Iowa State University
Digital Repository

Graduate Theses and Dissertations

Iowa State University Capstones, Theses and Dissertations

2018

VHDL auto-generation tool for optimized hardware acceleration of convolutional neural networks on FPGA (VGT)

Muhammad K A Hamdan
Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/etd

Part of the Engineering Commons

Recommended Citation
Hamdan, Muhammad K A, "VHDL auto-generation tool for optimized hardware acceleration of convolutional neural networks on FPGA (VGT)" (2018). *Graduate Theses and Dissertations*. 16368.
https://lib.dr.iastate.edu/etd/16368

This Thesis is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Graduate Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.

المنارة للاستشارات

www.manaraa.com

VHDL auto-generation tool for optimized hardware acceleration of convolutional neural networks on FPGA (VGT)

by

Muhammad K.A. Hamdan

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTERS OF SCIENCE

Major: Electrical and Computer Engineering

Program of Study Committee:
Diane T Rover, Major Professor
Phillip H. Jones
Mani Mina

The student author, whose presentation of the scholarship herein was approved by the program of study committee, is solely responsible for the content of this thesis. The Graduate College will ensure this thesis is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University

Ames, Iowa

2018

Copyright © Muhammad Hamdan, 2018. All rights reserved.

المنارة للاستشارات

www.manaraa.com

DEDICATION

To my family, for your love and support

To my friends for your advice and good companionship

المنارة للاستشارات

www.manaraa.com

TABLE OF CONTENTS

Page

LIST OF FIGURES vi

LIST OF TABLES viii

ACKNOWLEDGMENTS ix

ABSTRACT x

CHAPTER 1. INTRODUCTION 1

CHAPTER 2. BACKGROUND 4
- Brief Introduction to Machine learning 4
  - Supervised learning 5
  - Unsupervised learning 5
- Neural Networks 5

CHAPTER 3. CONVOLUTIONAL NEURAL NETWORKS AND FPGAs 7
- Convolutional Neural Networks 7
  - Convolutional Layer 8
  - Non-linearity (Activation Function) 10
  - Normalization layer 10
  - Pooling layer 11
  - Fully-Connected layer 11
  - Other Layers 12
    - Dropout layer: 12
    - Classification layer: 12
- CNN Topologies 12
  - LNet-5-5 13
  - AlexNet 13
  - Network-in-Network(NIN) 14
  - VGG-Net 14
  - GoogLeNet-5 14
  - ResNet 15
- Introduction to Field Programmable Gate Arrays 15
  - Field-Programmable Gate Arrays 15
    - Programmable Logic 16
    - Programmable Interconnect 16
    - Programmable I/O 17
    - Specialized programmable functional blocks 17
  - FPGAs Versus Other Hardware Platforms 17
    - FPGAs versus General-Purpose Processors 17
    - FPGAs versus ASICs 18

المنارة للاستشارات
www.manaraa.com

CHAPTER 4. ACCELERATOR DESIGN AND CNN IMPLEMENTATION 19
CNN Training 19
CNN Optimization 20
Parallelism within convolution, ... 20
CNN Implementation 21
Hardware Accelerators Design 22
Potential Hardware Platforms for Accelerators Design 22
- Central Processing Units (CPUs) 22
- Graphic Processing Units (GPUs) 23
- Application-Specific Integrated Circuits (ASICs) 23
- Field-Programmable Gate Arrays (FPGAs) 24

Achieving High Performance with FPGA-Based Computing 25
- Method 1, use an algorithm optimal for FPGAs 25
- Method 2, use a computing mode appropriate for FPGAs 25
- Method 3, use appropriate FPGA structures 25
- Method 4, living with Amdahl's law 26
- Method 5, hide latency of independent functions 26
- Method 6, use rate-matching to remove bottlenecks 26
- Method 7, take advantage of FPGA-specific hardware 26
- Method 8, use appropriate arithmetic precision 27
- Method 9, use appropriate arithmetic mode 27
- Method 10, minimize use of high-cost arithmetic operations 27
- Method 11, create families of applications, not point solutions 28
- Method 12, scale application for maximal use of FPGA hardware 28

Hardware Description Language and High-Level Synthesis 29
Hardware Description Language 29
High-Level Synthesis 30

CHAPTER 5. VHDL GENERATION TOOL 33
Tool Overview 34
VGT Tool Flow 35
- Configuration process: 36
- Parameters Inclusion process: 36

VHDL Generation Example Using VGT 38
Model Setup 38
Model Configuration 40
- Platform and Network Selection 40
- Model Configuration Block 41
- Parameters Inclusion Block 43

External Files/Dependencies 45
- Configuration File Syntax 45
- Configuration File Syntax 46

VHDL code Generation Process 47
- Architecture constructor 47
- Graphical User Interface 47

المنارة للاستشارات
www.manaraa.com

- Parameterization Library Manager ... 48
- VHDL code generator and storage ... 48

Generated VHDL Details ... 50

CHAPTER 6. RELATED WORK ... 54
Survey on Hardware Implementations of CNNs ... 54
Custom Hardware Platform ... 55
GPU Platform ... 56
FPGA Platform ... 57
- Memory System Optimization ... 57

Generic Memory System Optimization ... 57
Resource Utilization ... 59
- Computation Engine Optimization ... 61

Parallelization Exploration ... 61
- Scalable Architectures ... 64

Related Hardware Implementations of CNNs ... 66

CHAPTER 7. HARDWARE ARCHITECTURE ... 68
Small-Scale Models Architecture ... 68
System Architecture Overview ... 68
Accelerator Architecture ... 69
- Convolution Module Architecture ... 69
- Pooling Module Architecture ... 71
- Matrix Multiplication (Fully-Connected Layer) Architecture ... 72

Large Scale CNN Architecture ... 73

CHAPTER 8. RESULTS AND EVALUATION ... 75
Implemented Models Details ... 75
LeNet-5 Model ... 75
AlexNet Model ... 77
Results ... 78
LeNet-5 Model ... 78
AlexNet Model ... 81

CHAPTER 9. CONCLUSION AND FUTURE WORK ... 82

REFERENCES ... 84

المنارة للاستشارات

www.manaraa.com

LIST OF FIGURES

Page

Figure 3.1 Left: A 3-layer feed-forward Neural Network. Right: A CNN layer that arranges its neurons in three dimensions (width, height, depth). The 3D input volume is transformed into a 3D output volume of neuron activations in every layer [15]. 8

Figure 3.2 Right: A mathematical representation of the convolution operation followed by a nonlinearity function. Left: Input value of size 7×7×1 with padding of 1, a stride of 2, and receptive field of 3×3 is convolved with a filter (In Red) of size 3 [15] 8

Figure 3.3 Convolution of a 5×5×3 filter with 32×32×3 input image [15] 9

Figure 3.4 Activation Functions: ReLU, Tanh, Sigmoid 10

Figure 3.5 Average and maximum pooling output for 2 x 2 filter with stride of two [15] 11

Figure 3.6 Dropout representation 12

Figure 3.7 Internal Architecture of FPGA [26] 16

Figure 5.1 VGT: Proposed Solution 34

Figure 5.2 VHDL Generation Tool Flow 35

Figure 5.3 CNN Model Implementation Process 37

Figure 5.4 VGTEST, example CNN model 38

Figure 5.5 Platform and network style selection 40

Figure 5.6 Manual CNN model configuration 41

Figure 5.7 Loading configuration file of a pre-configured CNN model 41

Figure 5.8 Incorrect configuration due to wrong stride size used in the 4th layer 42

Figure 5.9 Successful configuration check 43

Figure 5.10 Parameters inclusion block for the example model 44

Figure 5.11 Successful parameters inclusion 44

المنارة للاستشارات

www.manaraa.com

Figure 5.12 Generated VHDL files for VGTEST model 45

Figure 5.13 Configuration file syntax of the example model 45

Figure 5.14 Random parameters syntax file of the example model 46

Figure 5.15 VHDL code generation process 49

Figure 5.16 Actual generated VHDL code of header section 50

Figure 5.17 Snapshot generic module from model entity 51

Figure 5.18 Snapshot of signals declaration 51

Figure 5.19 Snapshot of weights and biases section 52

Figure 5.20 Snapshot of internal module instantiation 52

Figure 7.1 Top-level architecture of the system 68

Figure 7.2 Processing element details in a convolutional layer for a 3 x 3 filter 70

Figure 7.3 Hardware details of a complete convolutional layer 71

Figure 7.4 Max pooling architecture using filter size of 2×2 72

Figure 7.5 Hardware architecture of fully-connected layer 72

Figure 7.6 Fully-connected layer architecture of a large-scale CNN, (Adapted from [32]) 74

Figure 8.1 Original LeNet-5 architecture [19] 76

Figure 8.2 Implemented LeNet-5 architecture, (Adapted from [19]) 76

Figure 8.3 AlexNet architecture: ImageNet 2012 winning CNN model. (Adapted from [32]) 78

Figure 8.4 Post-implementation simulation results of LeNet-5 using 16-bit precision 79

المنارة للاستشارات

www.manaraa.com

LIST OF TABLES

Page

Table I List of different CNN topologies that participated in the ImageNet challenge.... 13

Table II VGTEST CNN model summary. 40

Table III Supported Configurations 42

Table IV Supported operations by the combinational process for CNN layers................ 53

Table V LeNet-5 model configuration.. 75

Table VI AlexNet architecture details 77

Table VII Hardware resource utilization of 8-bit LeNet-5 implementation on Zynq xc7z020.. 79

Table VIII Resource utilization of LeNet-5 implemented on Virtex-7 using 8 and 16-bit.. 80

Table IX LeNet-5 implementation in comparison to other related.................................. 81

Table X Resources Utilization by AlexNet model. 81

Table XI Comparison with other implementations of AlexNet model............................ 81

Table XII Comparison with other automatic HDL generation implementations 81

المنارة للاستشارات

www.manaraa.com

ACKNOWLEDGMENTS

I would like to thank my committee chair, Prof. Diane T. Rover, and my committee members, Dr. Phillip Jones, and Dr. Mani Mina for their guidance and support throughout the course of this research.

I also would like to thank my friend and coworker, Murad Qasaimeh for his help, advice, and thoughtful discussions with me. In addition, I would like to thank all friends, colleagues, the department faculty and staff for making my time at Iowa State University a wonderful experience.

المنارة للاستشارات

www.manaraa.com

ABSTRACT

Convolutional Neural Network (CNN), a popular machine learning algorithm, has been proven as a highly accurate and effective algorithm that has been used in a variety of applications such as handwriting digit recognition, visual recognition, and image classification. State-of-the-art CNNs are computationally intensive, yet their parallel and modular nature make platforms like Field Programmable Gate Arrays (FPGAs) well suited for the acceleration process. Typically, Convolutional Neural Networks take a very long development round to be implemented or accelerated using FPGAs, hence in this thesis, we propose a VHDL generation tool (VGT), which through VHDL code (CNN architecture) can be on the fly generated for different CNN models (benchmarked and hand-tuned). The generated code or architecture is highly optimized, where it is modular, highly parallel, reconfigurable, scalable, fully pipelined, and adaptive to different CNN models. We demonstrate the automatic VHDL generation tool and its adaptability by implementing a small-scale CNN model “LeNet-5” and a large-scale one “AlexNet”. The generated code for the small-scale model does not incorporate any external memory management for the CNN parameters, whereas parameters are automatically hard-coded as constants unlike how it is typically done for large-scale models. On a Xilinx Virtex-7 running at 200 MHZ, the system is capable of processing up to 125k 28×28 Images per second for LeNet-5 and achieved a peak performance of 611.52 GOP/s for AlexNet.

المنارة للاستشارات
www.manaraa.com

CHAPTER 1. INTRODUCTION

Convolutional Neural Networks (CNNs), a type of neural networks and a prominent machine learning algorithm, inspired by the visual cortex of the brain and a mathematical operation called convolution, currently represent the most viable approach to image understand ing. Indeed, CNNs have gained popularity not only in image and video classification [1][2][3][4], but also in many other applications such as speech recognition [5][6], textual analysis [1][2], and visual object recognition and self-driving cars [7].

The idea of neural networks has been around since the 20s of the 19th century, yet the latest generations of high-performance computing platforms have allowed the evolution of CNNs. In the past couple of years, many CNN models such as LeNet-5, AlexNet, VGG, GoogleNet, and ResNet were presented. For example, AlexNet model [8] won ImageNet Large-Scale Vision Recognition Challenge (ILS VRC) 2012, achieving a top-5 accuracy of 84.7%. The exceptional performance of convolutional neural networks comes as a trade off to the enormous computational cost they require, where a large CNN model requires over billion operations per image. With the availability of powerful platforms like graphic processing units (GPUs), this level of performance can be reached, yet due to the high-power consumption of GPUs it is infeasible to embed such solutions into small portable systems. Different platforms have been considered for efficient implementations of CNNs, and FPGAs were investigated as the most promising one [9]. Interestingly, FPGAs seem to well-fit the job because they are reconfigurable, take advantage of the inherent parallelism in CNNs, and power efficient.

المنارة للاستشارات
www.manaraa.com

CNNs are known for their frequent data access, computation complexity, and very long development round on FPGAs, hence an efficient implementation is required. In this thesis, we present a VHDL generation tool that reduces time and effort in the process of implementing CNNs on FPGAs. The tool allows users to easily configure a CNN model through a graphical user-interface and generate a highly optimized VHDL code for it. The generated VHDL reflects a modular, highly parallel, scalable, reconfigurable, and fully-pipelined implementation of the target CNN model. The key contributions of this work are listed as follows:

- A paper entitled "VHDL generator for a high performance Convolutional neural network FPGA-based accelerator" is published out of this work.
- A VHDL generation tool that offers a highly optimized auto-generated implementation of CNN models on FPGAs with the following features:
  - ✓ Support for configuration through a GUI and/or external configuration file.
  - ✓ Support for different CNN models in extremely short development round.
  - ✓ The tool is optimized to ensure flexibility, and adaptability with CNN models.
  - ✓ Support for test-bench for validation and testing purposes
- A High-performance FPGA-based accelerator that is highly parallel, scalable, reconfigurable, and operates in a fully-pipelined style.
- The VHDL generation tool was tested on two benchmarked models (LeNet-5 and AlexNet) and other hand-tuned models. The system can process up to 125K Images/s for LeNet-5 and achieved peak performance of 611.52 GOP/s for the AlexNet model
- An executable of the VHDL generator is made available at: HTTPS://GITHUB.COM/MHAMDAN91/CNN_VHDL_GENERATOR

المنارة للاستشارات

www.manaraa.com

The rest of this thesis is organized as follows, CHAPTER 2. introduces general concepts about machine learning and neural networks. In CHAPTER 3. , a thorough background about Convolutional Neural Networks and their topologies is presented, and a brief introduction to FPGAs is given. CHAPTER 4. describes accelerators design and CNNs implementation. In CHAPTER 5. , we present the main contribution of this work, the VHDL generation tool. CHAPTER 6. presents CNN accelerators from previous work as well as related work to HDL code generation for CNNs. CHAPTER 7. illustrates hardware architecture and implementation details. CHAPTER 8. shows the evaluation of this work and obtained results. Finally, conclusion and future work are presented in CHAPTER 9.

المنارة للاستشارات
www.manaraa.com

CHAPTER 2. BACKGROUND

This chapter introduces a brief background about machine learning and its types of learning, then covers some concepts about artificial neural networks

Brief Introduction to Machine learning

Machine Learning is an artificial intelligence approach, by which machines "Computers" learn in a similar way to how humans learn. Machine learning addresses how program systems can automatically learn and improve with experience. Learning in this context is not learning by heart but recognizing complex patterns and make intelligent decisions based on data. The difficulty lies in the fact that the set of all possible decisions given all possible inputs is too complex to describe. To tackle this problem the field of machine learning develops algorithms that discover knowledge from specific data and experience, based on sound statistical and computational principles.

The field of machine learning integrates many distinct approaches such as probability theory, logic, combinatorial optimization, search, statistics, reinforcement learning, and control theory. The developed methods are at the basis of many applications, ranging from vision to language processing, forecasting, pattern recognition, games, data mining, expert systems, and robotics [10]. Machine learning is usually divided into two main types: predictive (unsupervised) or supervised learning [11]. There is a third type of machine learning that is not widely used known as reinforcement learning. The latter type is useful for learning how to act when given occasional reward or punishment signals.

المنارة للاستشارات
www.manaraa.com

Supervised learning

In supervised training, inputs and outputs are provided, where inputs are processed by the network, and its resulting outputs are compared against desired outputs. Errors are then propagated back through the system causing the system to adjust weights that control the network. This process occurs over and over as the weights are continually tweaked to minimize the error. The dataset that enable training is called training set. During the training of a network the same set of data is processed many times as the connection weights are ever refined.

Unsupervised learning

In unsupervised training, the network is provided with inputs but not with desired outputs. The system itself must then decide what features it will use to group the input data. This is often referred to as adaptation. Unsupervised learning is usually used for clustering purposes.

Neural Networks

The development of neural networks dates back to the early 19th century. ANNs models are inspired by biological neural networks based on the functionality of neurons. Usually, neural networks consist of many artificial neurons that are interconnected with each other. The neurons are arranged in such a way to form a feed-forward neural network. Neurons are the basic building block of a neural network, where a neuron receives a number of input signals x_i from other neurons and these input signals are multiplied with weights W_i to simulate the synaptic interaction at the dendrites. The weighted inputs are summed up, biased with a value typically equals 1, and fed into a non-linear activation function that produces the neuron's output signal.

$$OUT^{i}_{neuron} = \sum_{j=1}^{K_{input}} INPUT^{i} \times weight^{ij} + Bias^{i} \qquad (2.1)$$

المنارة للاستشارات
www.manaraa.com

Why neural networks and not regular computer programming? The idea behind neural networks is that they do not require an explicit description of a problem neither need to be programmed to perform a particular task. The neural network adapts itself during a training phase, based on examples of similar problems. When a network has completed its training phase, the network is able to relate the problem data to the solutions, inputs to outputs, and able to offer a feasible solution to a new problem.

Before a neural network is deployed, the network must be trained on a particular set of examples, where parameters (weights and biases) in the neural network are not manually chosen, but learned during this training phase. As mentioned in supervised learning, a network is provided with a set of labeled training examples. The training starts with small and randomly initialized weights. Inputs are multiplied with weights and fed to a non-linearity function that produce the output to be compared with the labeled examples using a loss function that measures the difference between the true output (labeled examples) and the output of the non-linearity function. Error is minimized by optimizing the values of weights. Using the Back-propagation Algorithm [4] , outputs are propagated all the way back in the network. This is typically solved via Stochastic Gradient Descent (SGD) [12].

Stochastic gradient descent algorithm is perhaps the most commonly used optimization procedure for training deep neural networks [13], in which the network weights are moved along the negative of the gradient of the performance function. The term backpropagation refers to the manner in which the gradient is computed for nonlinear multilayer networks. The algorithm propagates the error, that is computed as the difference between the output of the forward pass and the expect output, back throughout the network to adjust the weights values in order to minimize the error.

المنارة للاستشارات
www.manaraa.com

CHAPTER 3. CONVOLUTIONAL NEURAL NETWORKS AND FPGAs

This chapter sheds the light on Convolutional Neural Networks, their structure and topologies. Further, the design and training of CNNs are illustrated, and a concise introduction to field programmable gate arrays (FPGAs) is provided.

Convolutional Neural Networks

Convolutional Neural Networks are a class of feed-forward neural networks that are suited for operations on 2-dimensional data such as images. CNNs are similar to ordinary neural networks, where they are made up of neurons that have weights and biases. Neurons in CNNs receive inputs, perform a dot product that is followed by a non-linearity, and then applies loss function on the classification layer. The major difference between CNNs and regular feed-forward NNs is that CNNs deal better with 2D input data and that is why they are mainly used in image classification.

CNNs usually start with a convolutional layer, where it takes input images and decompose them into different feature maps such as edges, lines, curves, etc. Multiple processes are applied to the extracted feature maps throughout the entire network. Extracted feature maps from the last layer (typically, a fully connected layer) are classified into output classes using a classifier like SoftMax classifier [14]. A typical Convolutional Neural Network consists of a number of convolutional and fully connected layers, where most of the operations are performed; pooling layers that are used to avoid overfitting; a classification layer, to classify final results into classes; and other as-needed layers. A layer in the CNN consists of 3D volumes of neurons as shown in Figure 3.1 (width, height, and depth and the word depth refer to what is called “Feature-maps or activation-maps” not the number of layers in the CNN).

المنارة للاستشارات
www.manaraa.com

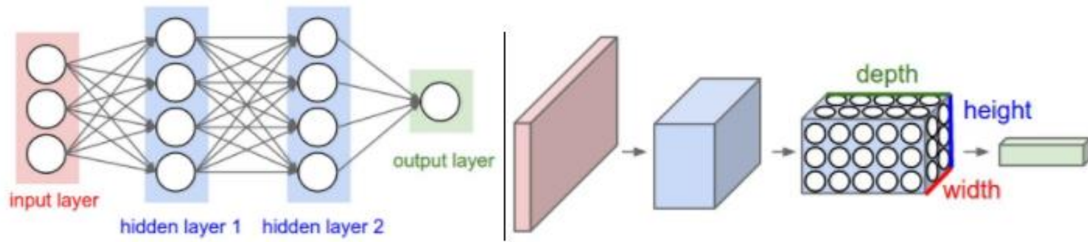

Figure 3.1 Left: A 3-layer feed-forward Neural Network. Right: A CNN layer that arranges its neurons in three dimensions (width, height, depth). The 3D input volume is transformed into a 3D output volume of neuron activations in every layer [15].

Convolutional Layer

The convolutional layer is considered as the main building block of a CNN, and it comprises most of operations in a CNN model. The convolutional layer essentially performs a mathematical operation called convolution that involves 3-dimensional multiply accumulate (MACC) operations. Shown in Figure 3.2, a kernel/Filter (Filter values selection depends on intended features, and input images should be divisible by 2 many times) of weights that is multiplied by a set of inputs (receptive region), and the weighted inputs are summed together.

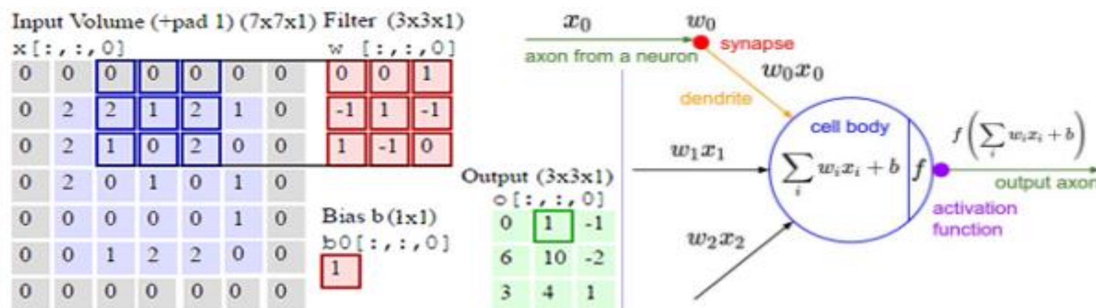

Figure 3.2 Right: A mathematical representation of the convolution operation followed by a nonlinearity function. Left: Input value of size 7×7×1 with padding of 1, a stride of 2, and receptive field of 3×3 is convolved with a filter (In Red) of size 3 [15]

A bias whose value usually one is added to the summed weighted inputs to ensure that neurons fire. An activation function such as rectified linear unit (ReLU) is applied to the accumulated sum to introduce nonlinearity and limit the output to a reasonable range. Results from the activation function are traversed to corresponding neurons in the next layer.

المنارة للاستشارات
www.manaraa.com

The computation of the spatial size of the output is shown in Equation 3.1

$$Output_{size} = \frac{(Input_{width} - Filter_{size} + 2\times Padding)}{Stride} + 1 \qquad (3.1)$$

Three hyperparameters control the size of the output: Depth, Stride, Zero-padding, where the stride is the slide rate at which the filter slides (most common slide value is 1, where the filter is moved one pixel to the right at a time), and zero-padding is a process that is applied to the border of the input to help control the spatial size of the output and preserve the information on the boarder.

In CNNs the total number of parameters (weights and biases) is less than regular feed-forward networks, whereas not all neurons are connected to each other. The overall number of parameters is reduced because local field receptors (local connectivity) is applied, where neurons only connect to respective local field without the need to connect to all inputs (pixels in an image or neurons in a feature map). The field receptor is shared among all neurons in the next layer. For example, if there is N hidden layers and 5×5×3 receptor field, then the total number of parameters equals → (5×5×3×N) + (N biases). Figure 3.3 shows an input image convolved with a filter (in green), producing a corresponding activation maps (in blue). The other activation map (in green) is similarly was produced by a different filter with the same size, but different filter values

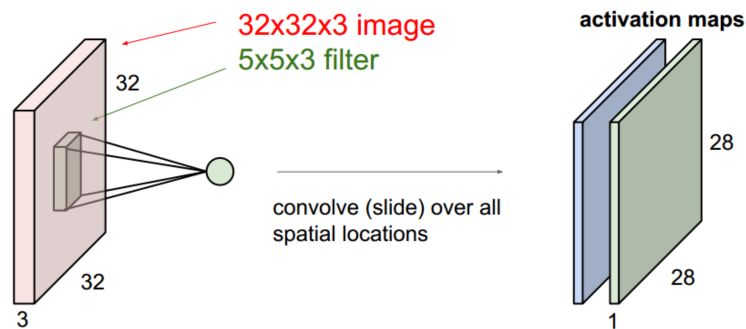

Figure 3.3 Convolution of a 5×5×3 filter with 32×32×3 input image [15]

المنارة للاستشارات
www.manaraa.com

Non-linearity (Activation Function)

Activation function is applied to each input pixel to ensure nonlinearity in the network as well as to get rid of unnecessary information. Among the various activation functions, Sigmoid, Tanh, and ReLU are the most commonly used activation functions. Sigmoid = $\frac{1}{(1+e_{-x})}$, and Tanh = tanh(x) activation functions require a longer training time in CNNs [16], unlike ReLU activation function which converges faster during training. Further, ReLU is simply defined as a zero-thresholding operation → ReLU = max (0, x). Figure 3.4 shows the different types of activation functions.

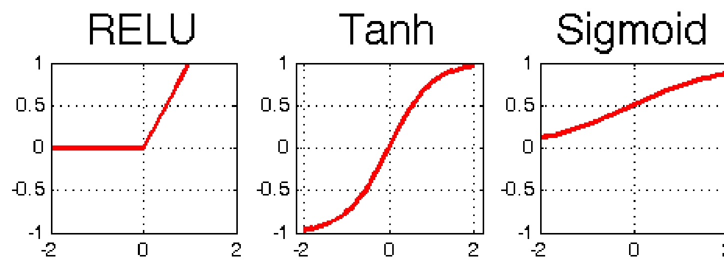

Figure 3.4 Activation Functions: ReLU, Tanh, Sigmoid

Normalization layer

Normalization or Local Response Normalization (LRN) implements the lateral inhibition [16] by damping the responses that are uniformly large in any given local neighborhood. Before sending the weighted inputs (outputs) of convolution to the nonlinearity, normalization layer normalizes the outputs depending on the neighboring neurons to help bring inputs to ReLU to a common scale. LRN layer was introduced in the AlexNet architecture [8], but are less common in recent CNNs.

المنارة للاستشارات
www.manaraa.com

Pooling layer

The importance of pooling layers comes to the fact that they prevent CNNs from overfitting [17]. Basically, spatial pooling is a form of nonlinear subsampling that is utilized to reduce the feature dimensions as we go deeper in the network. There are multiple methods to perform pooling and the most common ones are average and maximum pooling. In max pooling a set of neurons are subsampled based on the size of a pooling filter, whereas the maximum neuron value in that filter is passed to the corresponding neuron in the next layer and the rest of neurons are dropped out. In average pooling the forwarded value to the corresponding neuron in the next layer is the average of all neurons in the used filter as shown in Figure 3.5.

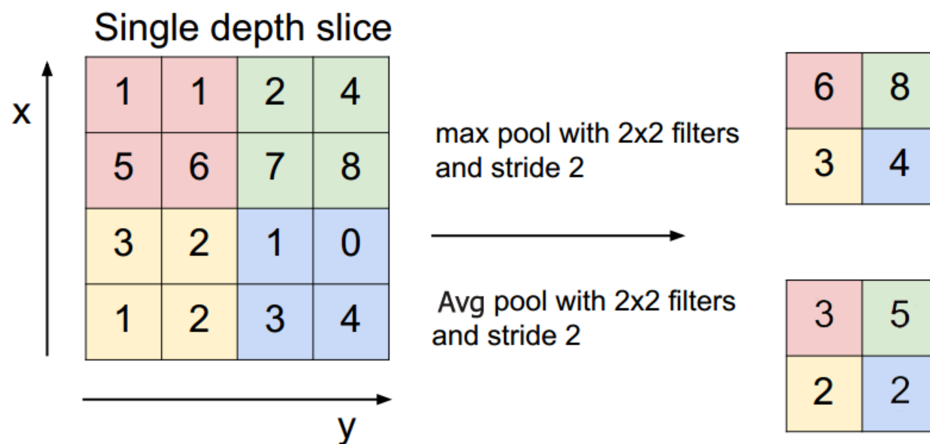

Figure 3.5 Average and maximum pooling output for 2 x 2 filter with stride of two [15]

Fully-Connected layer

The fully connected (FC) layer usually comes before classification layer and it comprises the highest number of parameters because every neuron in this layer is connected to all neurons in the previous layer, and parameters are translated on the connections between those neurons. Inputs in this layer are multiplied with corresponding weights, biases added respectively, and nonlinearity is applied similarly like convolutional layers.

المنارة للاستشارات
www.manaraa.com

Other Layers

Dropout layer: A method used to avoid overfitting in large CNNs. During training, this layer randomly drops a selectable percentage of its connections in order to prevent the network from learning very precise mappings, and force some redundancy to be built into the learned weights.

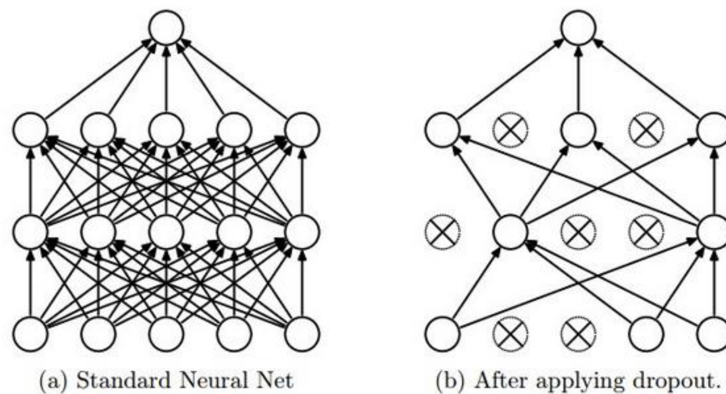

Figure 3.6 Dropout representation

Classification layer: This is last layer in a CNN and its main functionality is to classify the final output from the preceding layer into specific classes. In this layer a classification function such as SoftMax is used to perform the classification process. Basically, the SoftMax classifier converts raw class scores z_i of the nonlinearity in the preceding layer to a probability P_i in the range (0, 1) according to $P_i = \frac{e^{z_i}}{\sum_k e^{z_k}}$, to help bring the results to a common scale. The top probabilities from SoftMax classifier are then compared with actual labels of the available classes, hence evaluate the accuracy of the model.

CNN Topologies

Over the past couple of years many CNN topologies were introduced for their accuracy and performance. Image classification is an interesting and considerably the hardest task in computer vision, where the solver has to either label images, identify objects in images, or

www.manaraa.com

group images of the same characteristics in similar groups. This task seems to be solved efficiently using Convolutional Neural Networks.

An image classification competition called the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) is held annually, where participants compete with their developed CNN algorithms to classify images from ImageNet database. The ImageNet database consists of more than 14 million images, where each labeled with a corresponding class. The training set of ILSVRC comprises about 1.2 million images under 1000 different classes. Table I shows a summary for some of CNN algorithms that participated in ImageNet Challenge.

Table I List of different CNN topologies that participated in the ImageNet challenge

	Convolution layers	Parameters [millions]	Activations [millions]	ImageNet Top-5 error
AlexNet	5	60	2.4	15.4%
VGG-Net	16	138	29	11.2%
GoogLeNet-5	22	5	10.4	6.7%
ResNet-2015	50	25	46.9	3.6%

LNet-5-5

Among the early CNN models LeNet-5 was proposed by Yann LeCun et al [18] in 1998 to perform digit recognition on images contained digits. The model contained only two convolutional layers and two pooling ones along with two full connected layers. We demonstrated out VHDL generation tool by implementing this model.

AlexNet

AlexNet model is the winner of the ILSVRC challenge in 2012 and it was developed by Alex Krizhevsky et al [8]. The architecture comprises a total of 5 convolutional layer with three pooling layers and two full connected layers. AlexNet has about 60 million parameters and performs approximately 1.1 billion MACC operations for one forward pass. The SoftMax

المنارة للاستشارات
www.manaraa.com

classifier and dropout technique are adopted in this network, and the LRN layer was proposed in this architecture as well. This model achieved a top-5 error rate of 15.4%.

Network-in-Network(NIN)

Developed by Min Lin et al. [19] in 2013, NIN architecture consists of small multilayer perceptron working as convolutional filters, that slid over the respective input. In this network average pooling is adopted in the classifier instead of the fully-connected layers, hence the network has a smaller number of parameters. This model can be trained on ImageNet dataset and can reach the level of Alex-Net accuracy [20].

VGG-Net

The Visual geometry group (VGG) model designed by Karen Simonyan and Andrew Zisserman [21] was the winner of the ImageNet challenge in 2014. The deepest proposed model contained 19 convolutional layers, that is about 4x deep as AlexNet. Convolutional layers exclusively used 3×3 convolution filters and 2×2 max-pooling ones. This network has a very high number of parameters of about 138 million and a single forward pass requires approximately 16 billion MACC operations.

GoogLeNet-5

As the trend about CNN models was to develop deeper networks, Christian Szegedy et al [22] proposed a 22-layer deep CNN model called GoogLeNet-5, which won ImageNet challenge in 2015 with a top-5 error rate of 6.7%. The model has only 1.2 million parameters that is about 0.86% of VGG parameters. The massive reduction of parameters resulted into more complex architecture that employs what so-called Inception modules. An inception module is basically a network-in-network sub architecture that uses a 1×1 convolutional layer to reduce the number of input channels.

المنارة للاستشارات
www.manaraa.com

ResNet

Proposed by kaiming He et al [23] from Microsoft research, the residual network (ResNet) with a depth of 152 won the ImageNet challenge by achieving a top-5 error rate of 3.6%. Speaking of 152 layers means a hard training problem. To get over this problem, researchers included detours around each batch of subsequent convolutional layers. This topology can be viewed as $y = F(x) + x$ where the network has to learn a residual function F(x) only.

Introduction to Field Programmable Gate Arrays

This section gives a brief introduction to Field-Programmable Gate Arrays (FPGAs), then highlights characteristics, strengths, and weaknesses of FPGAs in comparison with other hardware platforms such as central processing units (CPUs), graphics processing units (GPUs), and application specific integrated circuits (ASICs).

Field-Programmable Gate Arrays

Field Programmable Gate Arrays are prefabricated semiconductor devices that consist of 2D arrays of configurable logic blocks (CLBs, or logic slices), which are connected via programmable logic. Interconnect resembles a network of wire bundles that run horizontally and vertically between the logic blocks with switchboxes (switches matrices) at each intersection between the horizontal and vertical bundles. The logic blocks, the fixed-function units as well as the interconnect are programmed electronically by writing a configuration bitstream into the device to implement any digital design. The configuration is typically held in SRAM memory cells and the FPGAs can be reprogrammed many times [24].

The first static memory-based (SRAM) FPGA was proposed by Wahlstrom in 1967. This architecture allowed for both logic and interconnection configuration using a stream of configuration bits. The first commercial modern-era FPGA was introduced by Xilinx in 1984. It contained arrays of configurable logic blocks and inputs/outputs (I/Os). Modern high-end

المنارة للاستشارات

www.manaraa.com

FPGA generations feature hundreds of thousands of configurable logic-blocks, and they include an abundance of hardened functional units that enable fast and efficient implementations of common functions.

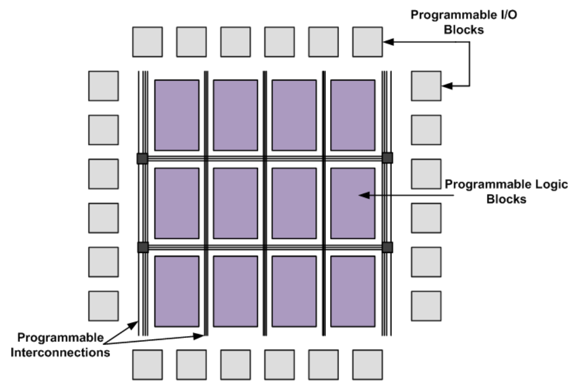

Figure 3.7 Internal Architecture of FPGA [25]

- **Programmable Logic**

Programmable logic blocks in FPGA are used to provide the basic computation and storage elements used in digital systems. A typical basic logic element contains some form of programmable combinational logic, a flip-flop or latch, and some fast carry logic in order to reduce area and delay cost. Additionally, modern FPGAs contain a heterogeneous mixture of different blocks some of which can be used for specific functions such as dedicated memory blocks, multipliers (DPS blocks), or multiplexers [25].

- **Programmable Interconnect**

Connections between logic blocks and I/O blocks are provided through programmable routing in FPGA. The interconnect consists of pass transistors, tri-state buffers, and multiplexers that achieve the desired connection. Generally, multiplexers and pass transistors are used within a logic cluster to connect the logic elements together while all three are used for more global routing structures. There are several global routing structures that have been used in FPGAs such as island-style, cellular, bus-based, and registered architectures.

www.manaraa.com

- **Programmable I/O**

Logic blocks and routing architectures are interfaced with external components of an FPGA through Input/output pads or programmable I/O. The I/O pad and supporting surrounding logic circuitry form important components that are called I/O cells. Due to variation in supply voltage and reference voltage standards, the design of I/O programmable blocks is kind of challenging. The choice of supported standard is one of the most important decisions in I/O architecture design. Supporting large number of standards can increase the silicon area required for I/O cells significantly [25].

- **Specialized programmable functional blocks**

FPGA architecture has been developed over the course of time through adding more specialized programmable functional blocks such as embedded memory (Block RAMs), arithmetic logic (ALUs), multipliers (MUXs), digital signal processors (DSP48), and embedded microprocessors. This made FPGAs heterogeneous platforms.

FPGAs Versus Other Hardware Platforms

- **FPGAs versus General-Purpose Processors**

The advantage of FPGA-based systems over traditional processing units-based systems such as desktop computers, smartphones, and GPUs, is the availability of freely programmable general-purpose logic blocks. FPGAs can be arranged into high performance specialized accelerators for very specific tasks, resulting in improved processing speed, higher throughput. Compared to GPUs, FPGAs are considered to be a much power-efficient devices where they fit better for mobile device-based applications. These advantages come at the price of increased complexity and reduced agility during development time, where designers need to carefully take into consideration the available hardware resources and the efficient mapping of target

المنارة للاستشارات
www.manaraa.com

algorithms onto the FPGA architecture. Further, FPGAs exceed the computing power of digital signal processors (DSPs) by breaking the paradigm of sequential execution and accomplishing more per clock cycle where they take full advantage of hardware parallelism. Controlling inputs and outputs (I/O) at the hardware level provides faster response time and specialized functionality to closely match application requirements. FPGAs usually do not use operating systems that actually minimize reliability concerns with true parallel execution and deterministic hardware that is dedicated to every task [26].

- **FPGAs versus ASICs**

Application-Specific Integrated Circuits (ASICs) are custom-tailored semiconductor devices. Unlike FPGAs, ASICs do not have any area or timing overhead that could be caused by configuration logic and generic interconnects, thus resulting in the fastest, most energy-efficient, and smallest systems. However, the sophisticated fabrication processes for ASICs results in a very lengthy and complicated development round and very high nonrecurring engineering upfront costs that demand a first-time-right design methodology and very extensive design verification. Therefore, ASICs are mostly suited for very high-volume, cost-sensitive applications where the non-recurring engineering and fabrication costs can be shared between a large number of devices. FPGAs with their reprogrammability are better suited for prototyping and short development cycles, where concepts can be tested and verified in hardware without going through the long fabrication process of custom ASIC design. FPGA chips are field-upgradable and do not require the time and expense involved with ASIC redesign. Digital communication protocols, for example, have specifications that can change over time, and ASIC-based interfaces may cause maintenance and forward-compatibility challenges. Being reconfigurable, FPGAs can keep up with future modifications that might be necessary [25][27].

المنارة للاستشارات
www.manaraa.com

CHAPTER 4. ACCELERATOR DESIGN AND CNN IMPLEMENTATION

Convolutional Neural Networks go through multiple phases before they get implemented in hardware to perform particular tasks for a particular application. In the previous section we illustrated various CNN topologies, where all of them are essentially based on the same design concepts of a typical CNN structure. The variations between the aforementioned topologies are driven by parameters that control the behavior of the network. This chapter will introduce CNN training, optimization, and Implementation.

CNN Training

For a Convolutional Neural Network to perform image classification of a particular dataset, the network has to be trained to perform classification for that dataset. CNNs are typically trained using backpropagation algorithm [4]. This is usually solved via Stochastic Gradient Descent (SGD) [12]. SGD is perhaps the most commonly used optimization procedure for training deep neural networks [13], in which the network weights are moved along the negative of the gradient of the performance function. The term backpropagation refers to the manner in which the gradient is computed for nonlinear multilayer networks. The algorithm propagates error, that is computed as the difference between the output of forward pass and expected output all the way back throughout the network to adjust weights values in order to minimize the error. Usually, CNN models are developed using libraries provided by well-known frameworks such as TensorFlow, Keras, or Caffe. Training is mostly done using GPUs as it is relatively easy to implement CNNs on a GPU, yet GPUs provide a very high training speed although a large CNN such as AlexNet [8] could take up to a week to be trained.

For a quick and easy start with CNN training, TensorFlow [28] is recommended to perform the training process, where a thorough guide on how to develop and train CNNs can

المنارة للاستشارات
www.manaraa.com

be found on TensorFlow website. TensorFlow offers different libraries for CNN training on GPUs or CPUs. Training a CNN model is relatively straightforward, where the model can be written in high-level language like python and then be trained using TensorFlow libraries.

CNN Optimization

CNN is a naturally parallel algorithm and to take full advantage of this natural phenomena, is it best to exploit the available parallelism as much as needed for the target application. Parallelism in a CNN can be explained as follows:

Parallelism within convolution, the convolution of a matrix $n \times n$ using $m \times m$ filter can be computed concurrently in parallel in one clock cycle; **Parallelism within pooling,** pooling operation can be parallelized by subsampling all of the individual submatrices at the same time; **Parallelism within output feature maps**, extracted features maps are totally independent of each other, hence all of them can be computed in parallel. In other words, if we are looking at X features in an image then it is possible to run X parallel processes to extract those features; **Parallelism within input feature maps, i**ncoming feature maps from previous layers can be processed in parallel as they can be combined to produce one single output.

Another optimization methodology that can be taken into consideration is limiting data precision. Lower precision can save a lot of hardware resources, hence an efficient reduction in model precision considering meeting application requirements can achieve a huge optimization. In [29] authors studied the effect of limited precision data representation and computation on neural network training. Within the context of low-precision fixed-point computations, they observed the rounding scheme to play a crucial role in determining the network's behavior during training. Their results show that deep networks can be trained using only 16-bit fixed-point number representation when using stochastic rounding, and incurred little to no degrada-

المنارة للاستشارات
www.manaraa.com

tion in the classification accuracy. They also demonstrated an energy-efficient hardware accelerator that implements low-precision fixed-point arithmetic with stochastic rounding. Other works such as [30], [31], and [32] adopted even smaller precisions and gained decent results.

CNN Implementation

Convolutional Neural Networks can be realized using the following platforms: General purpose central processing units (GPCPUs), graphic processing units (GPUs), and FPGAs. GPCPUs are the least favored platforms to run CNNs as they underutilize CNNs. Convolutional Neural Networks are naturally parallel and their end-use applications are mostly image-processing based applications, thus having a CPU in this equation does not fit at all as CPUs are sequentially based processing elements that are not fit for image processing nor taking advantage of the inherit parallelism in CNNs. While CPUs are not good for processing CNNs, GPUs are the most favored platforms for training CNNs, and that is obviously because with CNNs, GPUs process what they were actually created for. However, GPUs are not energy efficient because of their high-power consumption.

Since CPUs do not take advantage of the available parallelism in a CNN and GPUs are energy in-efficient devices, FPGAs manage to balance this equation. FPGAs are the type of reconfigurable devices that can be designed to match particular design requirements. Usually FPGAs are utilized as accelerators, where in the case of CNNs they seem to be an excellent fit as they take advantage of the inherit parallelism in the CNN with a much lower power consumption than that of GPUs.

Life cannot always be easy, and that is the case with FPGAs. To accelerate a CNN on FPGA, designer/researcher has to go through a very long and hectic development process using a hardware description language (HDL) such as VHDL or Verilog. Developing using HDL can result in the most optimized implementation of an accelerator; however, some designers would

المنارة للاستشارات
www.manaraa.com

prefer to trade off some performance in order to simplify implementation and reduce the development time. High level synthesis tools like HLS-Vivado offer an alternative methodology to implement hardware accelerators, where they replace hardware description languages with high level languages such as C, C++ or SystemC. High level synthesis and hardware description language will be explained in details in a later section.

Hardware Accelerators Design

Hardware accelerator design is a process that is subject to the requirements of target application and the end-implementation platform. Typically, embedded systems have a set of requirements and subject to particular constrains such as timing, power, and physical size. Those constrains require serious optimizations to be performed on algorithms prior to hardware implementation. To meet design requirements under hardware constrains, target algorithm must be investigated very well to identify the suitability for the acceleration process, the major optimization components, and the appropriate hardware acceleration platform.

Potential Hardware Platforms for Accelerators Design

Our target platform for our target algorithm implementation is FPGA; nevertheless, in this subsection we are providing a complementary introduction to an early brief one we mentioned in 0 on potential hardware platforms for accelerator design.

- **Central Processing Units (CPUs)**

CPUs are the most common processing elements that are found in electronic devices such as personal computers, smartphones, tablets, playstations, Xboxes, and even cars like Tesla model S. Most of these CPUs are called general-purpose CPUs, which means that they are designed to perform any task, where they can be easily and flexibly reprogrammed using software. GPCPUs offer decent performance on a wide range of computation workloads; however, CPUs are sequentially based computing devices, meaning that they underutilize parallel-

المنارة للاستشارات
www.manaraa.com

based tasks. CPUs are not ideal for high-parallelism dependent problems such as image processing, which is what we are doing in this work.

- **Graphic Processing Units (GPUs)**

The name of this processing element speaks for what it actually does. GPUs are found in nowadays personal computers and are dedicated to graphics related processing workloads. Recently, GPUs were investigated as an acceleration platform for machine learning problems and other general computing tasks. A high-end GPU such as NVIDIA TITAN XP contains 3840 floating-point processing cores that can run at a boost-frequency of 1.582 GHz, offering about 547.7 GB/s memory bandwidth with memory speed of 11.4 Gbps. TITAN XP can compute up to 11 TFLOP/s, but that comes at the cost of high power consumption which peaks at 250 W. Such very high-power consumption processor is not suitable for power constrained embedded devices. Further, with GPUs, 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. Hence, GPUs are not the optimal platform for our target algorithm.

- **Application-Specific Integrated Circuits (ASICs)**

When it comes to meeting system requirements, ASICs are the ideal solution, where they can achieve the highest performance and energy efficiency. However, ASICs are less suitable for irregular computation and dynamic algorithms that evolve with time, since they do not provide any reconfiguration once fabricated. CNN algorithm is an evolving algorithm and there is no fixed model that is considered as a representative model. Implementing a complete CNN on an ASIC is neither efficient nor effective, but implementing parts of a CNN is a much

المنارة للاستشارات
www.manaraa.com

better option. Convolutional layers are very computationally expensive, and accelerating some fixed modules using ASIC technology might be efficient. An example of efficient ASIC-based implementation of parts of a CNN is neuromorphic integrated circuits that use analog electronic circuits to mimic neurons and neural networks on custom-designed ICs [33]. Overall, for a dynamically changing CNN that requires reconfiguration, a seldom ASIC implementation is not preferred.

- **Field-Programmable Gate Arrays (FPGAs)**

While it is best to adapt algorithms to the parallel nature of the GPUs, FPGA architecture is tailored for the application, where custom processing engines can be built using the programmable logic blocks to meet the algorithm needs. In other words, there is less emphasis on adapting algorithms when it comes to developing machine learning techniques for FPGAs. This allows more freedom to explore algorithm level optimizations. The performance of FPGA design can be further increased by utilizing fixed-point or half- point precision data formats. Optimizations and techniques that require many complex low-level hardware control operations cannot be easily implemented in high-level software languages, thus is it more attractive to consider FPGA implementation. Further, in addition to the adaptiveness of FPGA implementation, FPGAs are reconfigurable and flexible that offer a wide scope of CNN models to be implemented on the same chip without spending any further design costs as it is the case in ASICs. Thus far, FPGA is the most suitable platform for our algorithm, but the downside of FPGA-based implementations is that designers have to use hardware description languages to perform their implementation which are not very friendly to program with and require decent programming experience.

المنارة للاستشارات
www.manaraa.com

Achieving High Performance with FPGA-Based Computing

Herbordt, Martin C., et al [34] designed 12 methods to avoid generating implementational heat while using FPGAs to accelerate high performance computing (HPC) applications. In this section we summarize those methods and highlight the ones that fit our application.

- **Method 1, use an algorithm optimal for FPGAs**

Prior to accelerating an algorithm using FPGA, it is necessary to check that this algorithm is worthwhile accelerating on FPGA and match the reconfigurable and parallel nature the FPGA can offer. Typically, the optimal algorithm for FPGA acceleration differs from that for serial computer when creating high performance computing FPGA applications.

- **Method 2, use a computing mode appropriate for FPGAs**

When talking about computing mode, it is referred to the differences between computation in software and hardware. FPGA configurations might resemble high-level language programs, they essentially specify hardware, not software. Meaning that good computing modes for software are not necessarily good computing modes for hardware, whereas restructuring an application can substantially improve its performance. For example, random access and pointer-based data structures are merely staples of serial computing, they may yield poor performance in FPGAs. Streaming, systolic, associative computing structures, and arrays of fine-grained automata are more preferable.

- **Method 3, use appropriate FPGA structures**

FPGAs support various data structures, yet certain data structures such as stacks, trees, and priority queues are ubiquitous in application programs, as are basic operations such as search, reduction, and parallel prefix. The analogous structures and operations usually differ from what is obtained by directly translating software structures into hardware.

المنارة للاستشارات

www.manaraa.com

- **Method 4, living with Amdahl's law**

Amdahl's law states that significant application speedup through an enhancement requires most of the application to be enhanced, but this is difficult to achieve sometimes especially with existing high-performance computing code.

- **Method 5, hide latency of independent functions**

latency hiding can contribute to achieving high performance in parallel applications, especially the latency introduced by the overlap between computation and communication. In FPGA implementations, rather than allocating tasks to processors that must communicate with one another, latency hiding lays out functions on the same chip to operate in parallel.

- **Method 6, use rate-matching to remove bottlenecks**

Multi-processor implementations offer some flexibility in partitioning by function or data, but on FPGA functions are laid out on the chip, meaning that function-level parallelism is already built in. This implies pipelining not only within but also across functions. Further, rate-matching can also be found across computational power offered in a design and the I/O bound on target FPGA, thus it is better to match I/O with desired parallelism to avoid performing unutilized parallelism.

- **Method 7, take advantage of FPGA-specific hardware**

FPGAs are often viewed as homogenous substrates that can be configured into arbitrary logic. Nowadays FPGAs include, DSP modules, on-chip memories, and other processing elements. Those processing elements can be utilized to perform specific tasks to result in an optimized implementation and a better utilization of FPGA resources. For example, the Xilinx

المنارة للاستشارات

www.manaraa.com

VP100 has 400 independently addressable, 32-bit, and quad-ported BRAMs; it achieves a sustained bandwidth of 20 terabytes per second at capacity. Using this bandwidth greatly facilitates high performance and is an outstanding asset of current generation FPGAs.

- **Method 8, use appropriate arithmetic precision**

We talked about this in the previous section, where it is an excellent optimization technique to consider when using FPGAs for hardware implementation. High-end microprocessors have 64-bit data paths, which in many applications are often overlooked as only a few bits of precision are needed. In contrast with microprocessors where data paths are fixed, FPGAs enable configuration of data paths into arbitrary sizes, allowing a tradeoff between precision and parallelism. An additional benefit of minimizing precision comes from shorter propagation delays through narrower arithmetic units.

- **Method 9, use appropriate arithmetic mode**

Microprocessors provide support for integers and floating point depending on multimedia features; however, in DSP systems cost concerns often require DSPs to have only integers. Although software can emulate floating point when required, it is not preferred to use floating point representation in FPGA because it is costly. Generally, it is rule of thumb to avoid floating-point in FPGAs and replace them with fixed-point representation.

- **Method 10, minimize use of high-cost arithmetic operations**

The relative costs of arithmetic functions on FPGAs are different than on microprocessors. For example, FPGA integer multiplication is efficient compared to addition, while division is orders-of-magnitude slower. It is highly recommended to replace costly arithmetic operations with simple operations. Even if the costly operation like division is fully pipelined to hide its latency, the cost remains high in chip area, especially if the logic must be replicated.

المنارة للاستشارات
www.manaraa.com

On FPGA, implementing unused functions is not necessary; recovered area can be used to increase parallelism. Thus, restructuring arithmetic with respect to an FPGA cost function can substantially increase performance.

- **Method 11, create families of applications, not point solutions**

High performance computing applications are often highly parameterized and complex resulting in variations in applied algorithms as well as data format. While it is easy to support these variation on object-oriented technology, it is far more difficult to implement in current hardware description languages. But if those variation are implemented in HDL, it reduces development cost over a larger number of uses, enables higher reuse of the design, and relies less on skilled hardware designers for each application variation.

- **Method 12, scale application for maximal use of FPGA hardware**

Parallelism is the most contributive component in increasing performance, yet part of accelerator design consists of instantiating as many processing elements (PEs) as the FPGA's computing fabric will support. For example, automated sizing of complex arrays will become increasingly important for porting applications among FPGA platforms, given the frequency at which larger FPGAs become available.

Adopting the aforementioned methods in HPC application implementation is a necessary step in order to achieve high performance and avoid underutilizing FPGAs. Not all methods are required to be adopted in every HPC application, rather choosing which methods to adopt is dependent on the target application and FPGA platform. Since Convolutional Neural Network can be parallelized, can be represented in different precisions, dynamic, parametrizable, scalable, computational intensive, memory dependent, and flexible it is highly recommended to consider most of the aforementioned methods.

المنارة للاستشارات
www.manaraa.com

In our implementation we took full advantage of the available parallelism in CNNs by utilizing the possible parallelism according to the available hardware resources in order to optimally utilize the FPGA. Further, we used suitable structures for our implementation and exploited the heterogenous resources available on the FPGA. We fully-pipelined our design in order to reduce worst slack time, i.e. hide latency. To consider appropriate arithmetic mode, and precision, we used fixed point representation for parameters instead of floating-point and bind that representation with appropriate precision that can reasonably meet desired accuracy. In our implementation we tried to minimize the use of high-cost arithmetic operations. Overall, our implementation is highly parallelized, fully-pipelined, reconfigurable, scalable, and highly optimized.

Hardware Description Language and High-Level Synthesis

Hardware description language and high-level synthesis are very correlated techniques as they share similar objectives that are achieved in different methodologies. Hardware description language and high-level synthesis are used to write code of algorithms that are intended to be implemented on digital logic circuits such as FPGAs and ASICs.

Hardware Description Language

Hardware description Languages include VHDL, Verilog, SystemC and Handle-C. Behavioral, register transfer level and structural levels of description can be used inter-changeably in these languages. VHDL and Verilog are matured as industry standards, while SystemC is a C++ based library used for modeling system level behavior, where processes can be easily modeled than in a more traditional HDL. Synthesis tools for SystemC are not as mature as VHDL or Verilog synthesis products. Handel-C is a relatively new product in comparison to VHDL or Verilog. Handel-C follows the Communicating Sequential Process (CSP) model.

المنارة للاستشارات
www.manaraa.com

Handel-C requires the designer to explicitly delineate parallel processing blocks within a process. It includes intrinsic for inter-process communication, as does SystemC 2.0 [25].

Register Transfer Level (RTL) is the description of hardware designs, where programmers specify their algorithm details using a number of parallel processes that operate on vectors of binary signals and simple integer data types derived from them. These parallel processes are driven by the rising and falling edges of a clock signal and they describe combinational logic, basic arithmetic operations and registers. RTL descriptions are very close to the wires and logic gates that are available in the underlying FPGA technology, and therefore the hardware that results from RTL synthesis can be closely controlled. However, the process of transforming a given algorithm into processes, logic blocks, and finite state machines on the register transfer level is very long, tedious, and error-prone. Designers have to consider and make many design decisions before attempting to write any code, whereas later changes are difficult and costly. This in fact prevent iterative optimizations, demand a lot of intuition, experience and expert knowledge from designers in order to have a fully optimized and functional implementation of their target algorithm [24][35]. Hence, HDL development is not highly preferred by some researchers as they lack the adequate HDL programming skillset.

High-Level Synthesis

In order to overcome the barriers introduced by development using HDL, decent research has been conducted to increase the level of abstraction, reduce development round, and simplify implementation. High-Level Synthesis (HLS) offer designers an alternative path to implement algorithms. In HLS, a lot of implementation details are abstracted away and handled by the HLS compiler, where it replaces the development using HDLs with high-level programming languages such as C, C++ or SystemC. The HLS compiler converts the code developed using high-level programming languages (sequential description) into a concurrent hardware

المنارة للاستشارات
www.manaraa.com

description, usually at the RTL level. HLS tools are grouped into five main categories: high-level language-based frameworks, model-based frameworks, HDL-like languages, C-based frameworks, and parallel computing frameworks (i.e. CUDA/OpenCL).

With HLS, designers can implement their designs through loops, arrays, floats, function calls, and other relevant arithmetic operations. The used loops, arrays, function calls, etc. are converted into counters, multiplexers, multipliers, memories, computation cores and handshake protocols. The compilation can be guided using scripted compiler directives or compiler pragmas, which are meta-instructions interpreted directly by the HLS compiler [36][37]. Vivado High-level Synthesis (Vivado HLS), offered by Xilinx, is the most common commercial HLS tool.

Although HLS can provide faster development cycles, easier track for hardware implementation, and higher productivity; HLS tools do not provide sufficient optimization for a lot of applications. Optimization in HLS is limited and defined by the directives and programs that are embedded in the tool. As a matter of fact, HLS tools have been on the market for about 15 years now, yet designers still use hardware description languages for their FPGA designs. The task of converting sequential, high-level software descriptions into fully optimized, parallel hardware architectures is extremely complex. Although companies have invested hundreds of millions of dollars and years of research into HLS [38][39], the results attained are still highly dependent on the coding style and intricate design details. Because flaws and deficiencies in the compiler are only discovered during the design, the decision for HLS is associated with a non-negligible risk [40]. Having said that the implementation of algorithms using HDLs is tedious and complicated and optimization levels are not met using HLS, designers find themselves bound and have to trade off optimization for development round or vice versa.

المنارة للاستشارات
www.manaraa.com

Convolutional Neural Network models vary in size, yet small models are still considered large to be implemented using HDL. Actually, it is impractical to implement large or deep CNN models using HDL. Further, implementing deep CNN models using HLS might result in underutilizing those models, hence not achieving the best possible performance. To overcome the issue of long development round introduced by hardware description languages and underutilization caused by the high abstraction introduced by High Level Synthesis, we present a graphic user-interface based tool that is designed to automatically generate an optimized VHDL code/implementation for Convolutional Neural Network models. The details of the generation tool are explained in CHAPTER 5.

المنارة للاستشارات
www.manaraa.com

CHAPTER 5. VHDL GENERATION TOOL

VHDL is one of the most common hardware description languages that is used to develop hardware circuits at the register transfer level (RTL). In VHDL, designers typically specify their algorithm details using a number of parallel processes that describe some combinational logic, and basic arithmetic operations and registers. These processes are driven by the rising and falling edges of a clock signal and they operate on vectors of binary signals and simple integer data types derived from them.

The process of transforming an algorithm into processes, logic blocks, and finite state machines on the register transfer level is very long, tedious, and error-prone. As later changes are difficult and costly in this process, designers have to consider and make many design decisions before attempting to write any lines of code. Developing using HDL requires a decent experience from the designer to reduce costly changes and ensure satisfying design results. Further, the difficulty of development in HDL prevents iterative optimization, demands a lot of intuition in order to have a fully optimized and functional implementation of algorithms. Thus, development using HDL is not highly preferred by a lot of researchers especially those who are not familiar with it. This actually makes FPGAs less attractive to accelerate an algorithm. Further, CNN is a massive algorithm and implementing even a small size model like LeNet-5 [18], could take months making the implementation of a large-scale model such as AlexNet [8] impractical and infeasible.

المنارة للاستشارات
www.manaraa.com

Tool Overview

Because it is impractical to implement CNN models especially large ones using HDL from scratch, we present a VHDL generation tool (VGT) based on Java language, that offers a parameterized implementation to achieve the following: First, overcome the barriers introduced by high description languages and the limitations of HLS tools; Second, achieve high performance and avoid underutilizing CNNs; Third and last, significantly shorten the development round and provide easy and iterative optimization.

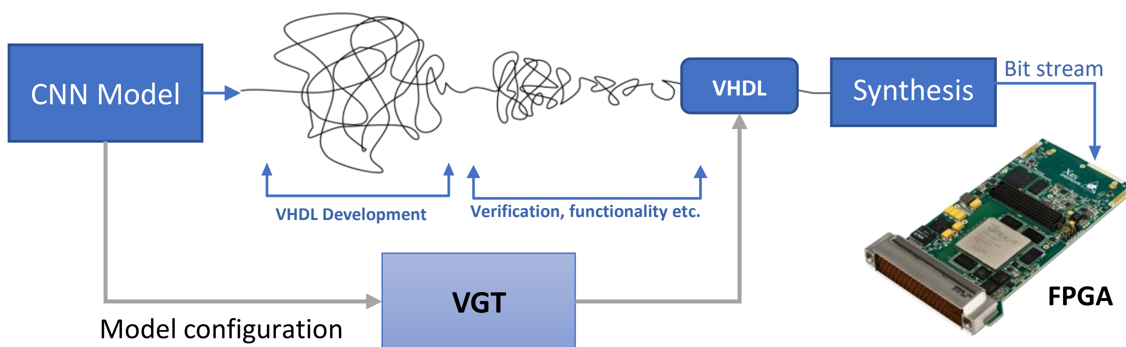

Figure 5.1 VGT: Proposed Solution

As shown in Figure 5.1, by passing only the configuration of a CNN model to VGT, users can configure their CNN model and generate optimized VHDL code in a few seconds or minutes depending on the size of the implemented model. As of now, VHDL is the supported HDL language for realizing our implementations. The generated implementation is fully-pipelined, each stage in the design is properly pipelined to hide latency; highly parallel, parallelism is highly utilized corresponding to the available hardware resources; scalable and reconfigurable, the implementation can easily be reconfigured either using VGT or directly using the generated VHDL code in accordance to desired changes; and modular, the generated code is broken down into multiple VHDL modules, whereas each module represents a particular layer in the targeted CNN model.

www.manaraa.com

VGT Tool Flow

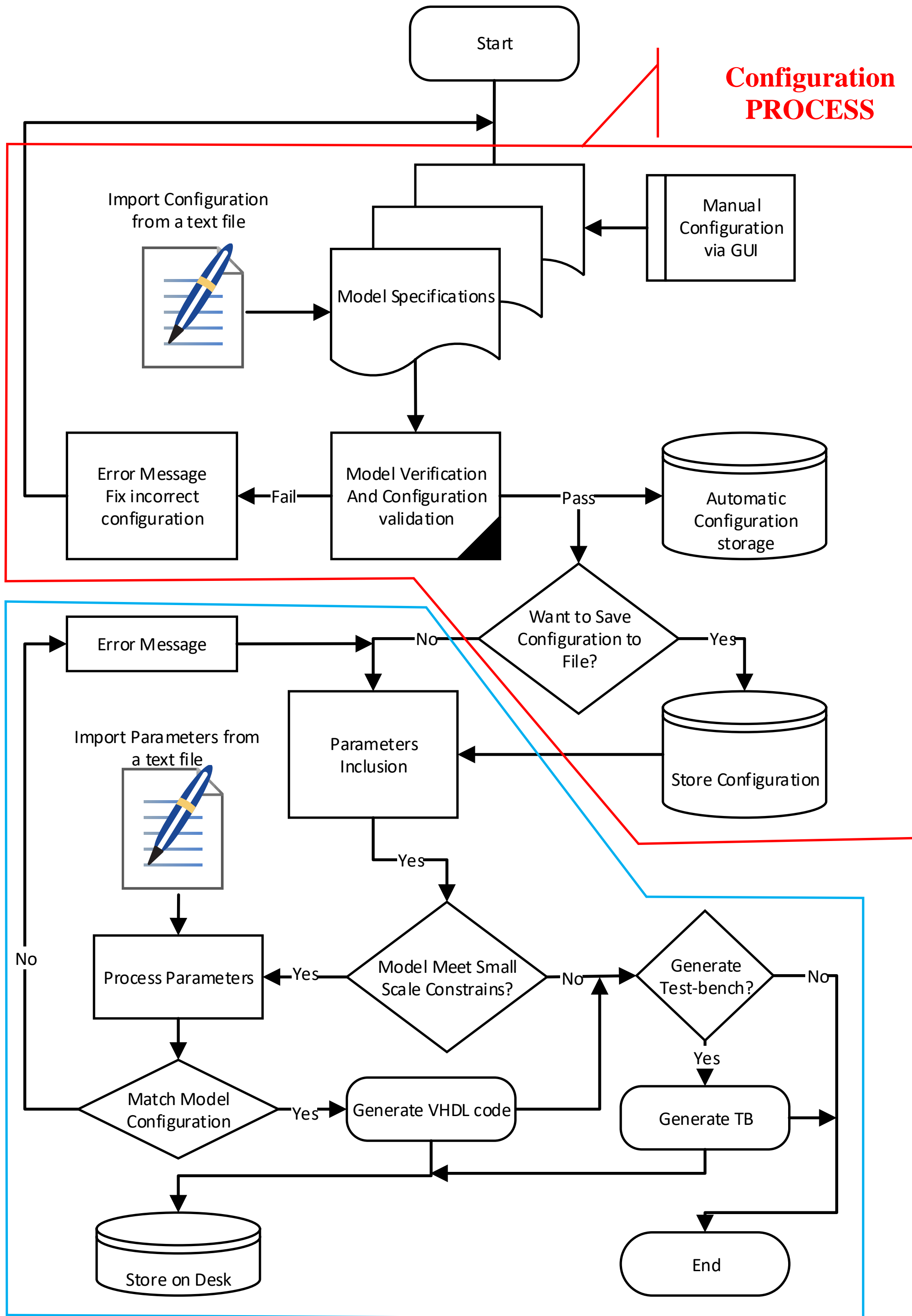

Figure 5.2 VHDL Generation Tool Flow

المنارة للاستشارات
www.manaraa.com

- **Configuration process:** In this process, users set model specifications through the tool's GUI (manual configuration) or through an external specification/configuration file. After configuration stage, users are prompted to verify their CNN model. Indeed, there are a set of rules that designers must adhere to when building a CNN model. In VGT, those rules are embedded to check the validity of the configured model. Only upon successful validation, users can proceed to the second process, where they can handle the parameters (weights and biases) of the target CNN model.
- **Parameters Inclusion process:** Upon successful configuration and validation of a CNN model, users are prompted to provide the parameters of the target model. In this process, users are asked to provide the representation of parameters as well as desired precision. Further, users are asked to select what to do with the parameters depending on the size of the target model. There are two options for handling model parameters, either to hardcode them as part of the programmable logic and this applies to small sized models, or store them on an external memory source and this is the case of large scale models. Upon selecting desired precision, parameters representation, and storage type, users are prompted to include the parameters through the tool GUI. The tool run validation checks on imported parameters to verify if they match up with the configured model and parameters representation. If the parameters file content violates any of the aforementioned rules, then the file will not be imported/loaded to the tool and an information message will be displayed to the user stating what errors should be fixed. Upon successful parameters inclusion, code generation module is enabled and users can generate VHDL code for their model as well as a test-bench for simulation and validation purposes.

المنارة للاستشارات
www.manaraa.com

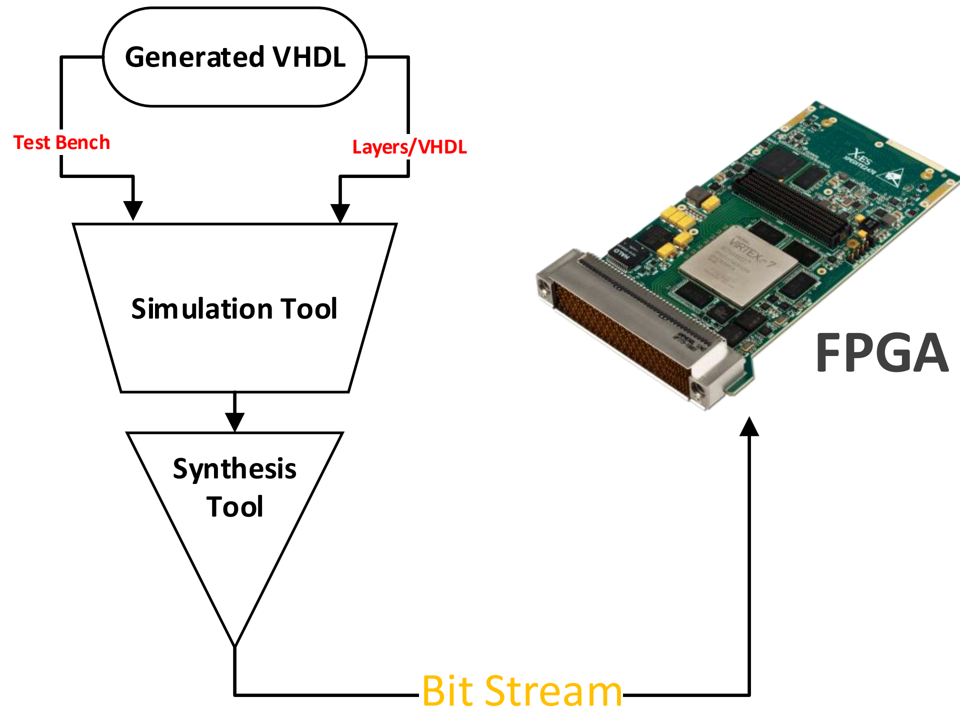

Figure 5.3 CNN Model Implementation Process

Now that users have generated an optimized code for their implementation, they can simulate the generated code using the provided test-bench to check the functionality of the model as well as get a clue of the model's performance. Also, users can skip simulation and can directly synthesis their model and generate a bitstream to run on an FPGA as shown in Figure 5.3.

For small scale models, users only need to put their target dataset images on an external memory and establish communication with the accelerator without worrying about weights and biases since they are hardcoded as part of the programmable logic. For large scale models, users will need to store both parameters and target dataset images on an appropriate memory source like external memory alone, or external and on chip memories. Currently the tool only generates code for the CNN layers, and users have to take care of accelerator-memory communication for loading/transferring images and parameters.

المنارة للاستشارات
www.manaraa.com

VHDL Generation Example Using VGT

In this section we will give a thorough example on how to configure a simple CNN model using VGT and generate VHDL code for it. The example CNN model (VGTEST) is shown in Figure 5.4. The model is composed of two convolutional layers, two pooling layers, one fully connected layer, and a classification layer of two classes, zero and one.

Model Setup

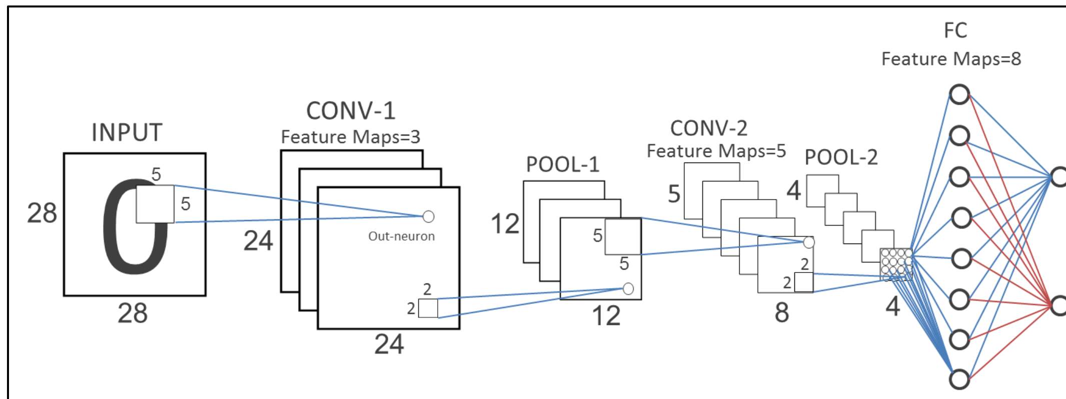

Figure 5.4 VGTEST, example CNN model

The configuration of the layers of our example model can be derived from Equation 5.1, where $\boldsymbol{FM_{size}}$ is the output feature map size, $\boldsymbol{IMAGE_{WIDTH}}$ is the input image size, $\boldsymbol{F_{size}}$ is the used filter size, **Stride** is the step size or shift of used filter over the input image, and **Padding** is the process of filling the edges of non-square images with zeros to make it divisible by used stride size.

$$FM_{size} = \frac{IMAGE_{WIDTH} - Fsize + 2 \times Padding}{Stride} + 1 \qquad (5.1)$$

The target model is used to classify images of handwritten 0s and 1s. The input image is grayscale and of size 28×28. In the first convolutional layer (Conv-1), three different feature maps are extracted through convolving the input image with kernels/filters of size 5×5. The stride of the convolutional filter is one, no padding is applied to the input image, and the applied activation function is rectified linear unit (ReLU). The output of the convolutional

المنارة للاستشارات
www.manaraa.com

layer is passed to the first pooling layer (Pool-1), where maximum pooling (max-pool) function is applied using a filter size 2×2. Basically, in max-pool the maximum neuron value in the filter is passed to the corresponding neuron in the next layer and the rest of neurons are dropped out as shown in the following Equation 5.2.

$$Passed_{neuron} \rightarrow max(2x, x, 0.5x, 3x) = 3x \qquad (5.2)$$

The second convolutional layer (Conv-2) takes in three input maps of size 12×12 and extracts 5 new feature maps for each input feature map. Conv-2 uses kernels of size 5×5 with a stride size of one and no padding, and ReLU is used as the activation function. The output of Conv-2 is 5 feature maps of size 8×8**.** The second pooling layer (Pool-2) is similar to the first one except that the input maps size is 8×8 and the output maps size is 4×4.

The Fully connected layer (FC) is similar to the convolutional one, but convolution is replaced with matrix multiplication. The FC layer constitutes most of the parameters/connections in the network, which are more than the total number of parameters for both of the convolutional layers combined. In this layer, 8 feature maps are extracted for each incoming feature map from the second pooling layer. The total number of parameters in the FC layer can be calculated as shown in Equation 5.3. The used activation function is ReLU.

$$FC_{parameters} = weights + biases \rightarrow {IN_{map}}_{size}^{\ 2} \times IN_{map} \times OUT_{maps} + OUT_{maps} =$$

$$4^2 \times 5 \times 8 + 8 = 648. \qquad (5.3)$$

Equation 5.3 can also be used to calculate the number of parameters for convolutional layers by replacing ${IN_{map}}_{size}^{\ 2}$ with $Filter_{size}^{\ 2}$, so the total number of parameters for the first and second convolutional layers would be 458. The last layer in the network is the classification layer, which is basically represented by the SoftMax classifier with normalizes the output of the fully connected layer into a probabilistic value between (0,1).

المنارة للاستشارات

www.manaraa.com

Table II VGTEST CNN model summary.

Layer	$MAP_{IN-SIZE}$	$MAP_{OUT-SIZE}$	$Filter_{size}$	Features	Activation	Stride	Padding
Input image	28×28	24×24	-	Grayscale	-	-	-
Conv-1	24×24	12×12	5	3	ReLU	1	0
Pool-1	12×12	8×8	2	-	Max-Pool	2	0
Conv-2	8×8	4×4	5	5	ReLU	1	0
Pool-2	4×4	1×1	2	-	Max-Pool	2	0
FC	1×1	1×1	1	8	ReLU	1	0
Classification	1×1	1×1	-	2	SoftMax	-	-

Model Configuration

VGT comprises three configuration stages that a user has to go through to generate a complete VHDL code for a CNN model. In this subsection, we will configure VGTEST model using VGT's graphical user-interface.

- **Platform and Network Selection**

In this block, users have to specify network style and target FPGA platform.

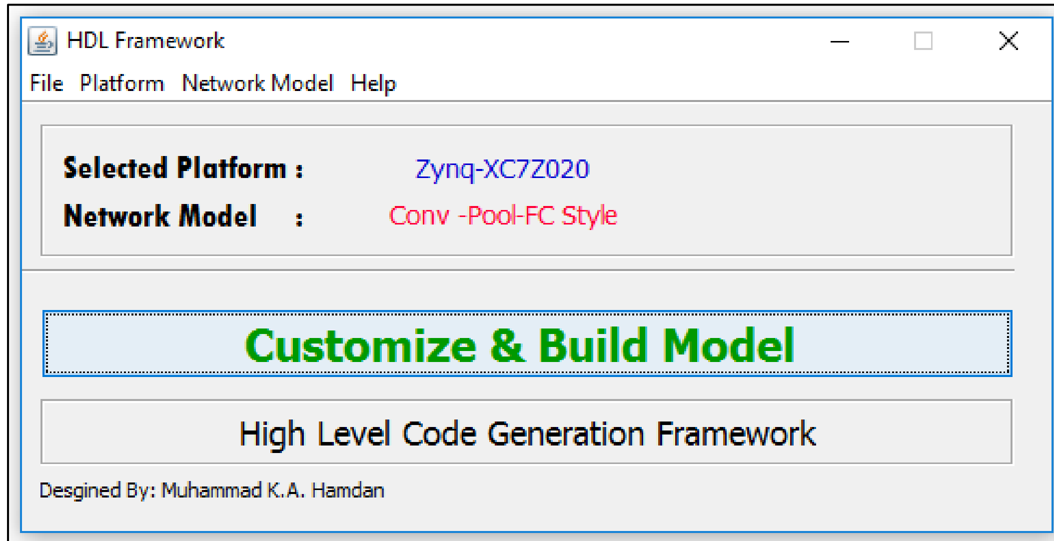

Figure 5.5 Platform and network style selection

المنارة للاستشارات
www.manaraa.com

- **Model Configuration Block**

In this block, users can configure their target CNN network manually using VGT GUI or load configuration of a pre-configured model from an external text file as shown in Figure 5.6 and Figure 5.7.

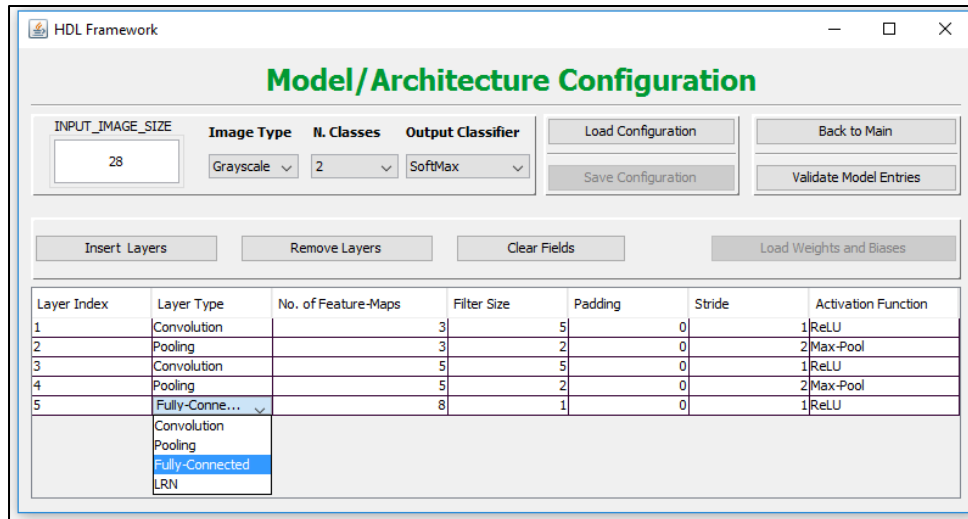

Figure 5.6 Manual CNN model configuration

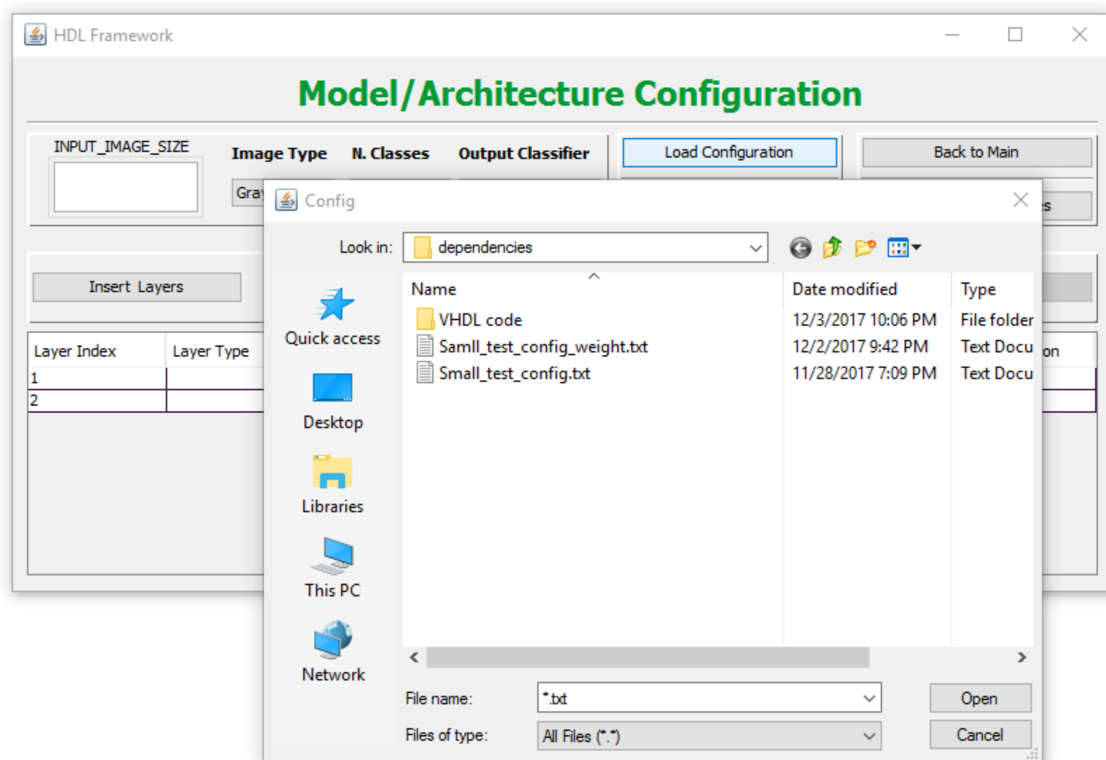

Figure 5.7 Loading configuration file of a pre-configured CNN model

www.manaraa.com

Once configuration is complete, user is prompted to validate their configuration to ensure it meets standard CNN configuration. On unsuccessful validation check, a prompted message is displayed to the user to inform them of what changes they have to make to fix errors.

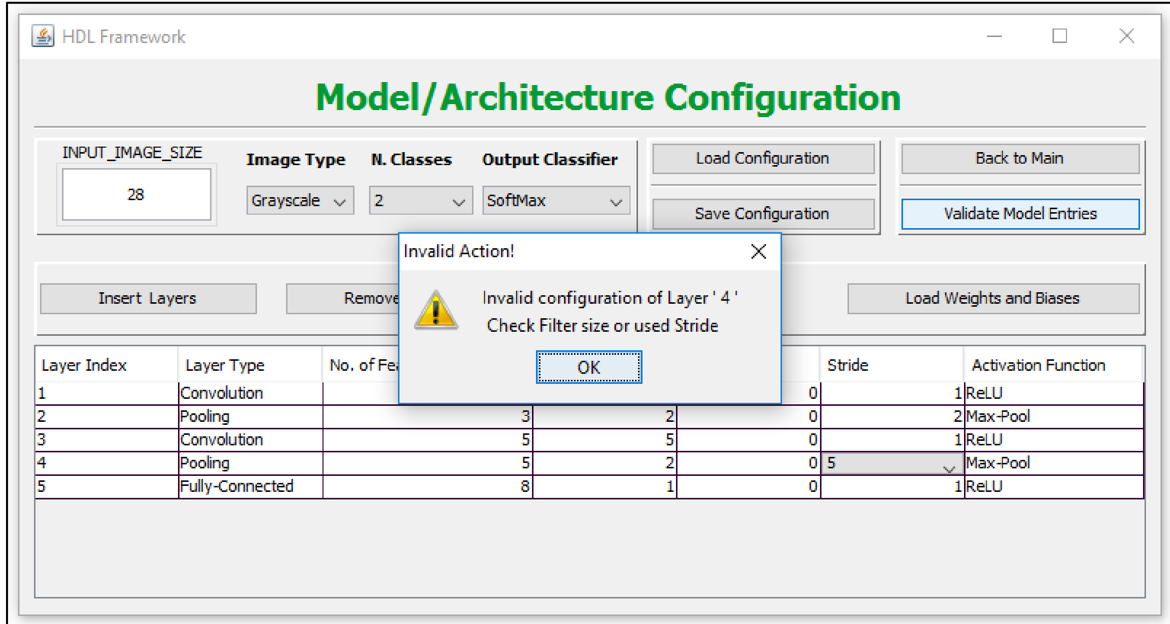

Figure 5.8 Incorrect configuration due to wrong stride size used in the 4th layer

On a successful validation check, the user can export their configuration to an external storage source in case they need to reuse the same configuration later. Load Weights and Biases button gets enabled only on successful validation check, then users can proceed to next stage, which is parameters inclusion. Supported configurations by the VGT are shown in Table III.

Table III Supported Configurations

Image Size	User-defined
Output Classifier	SoftMax
Filter Size	User-defined
Feature maps	User-defined
No. of Classes	User-defined
Layer type	Convolution, Pooling, FC, LRN
Activation Functions	ReLU, Tanh, Sigmoid, Average and Max Pool

المنارة للاستشارات
www.manaraa.com

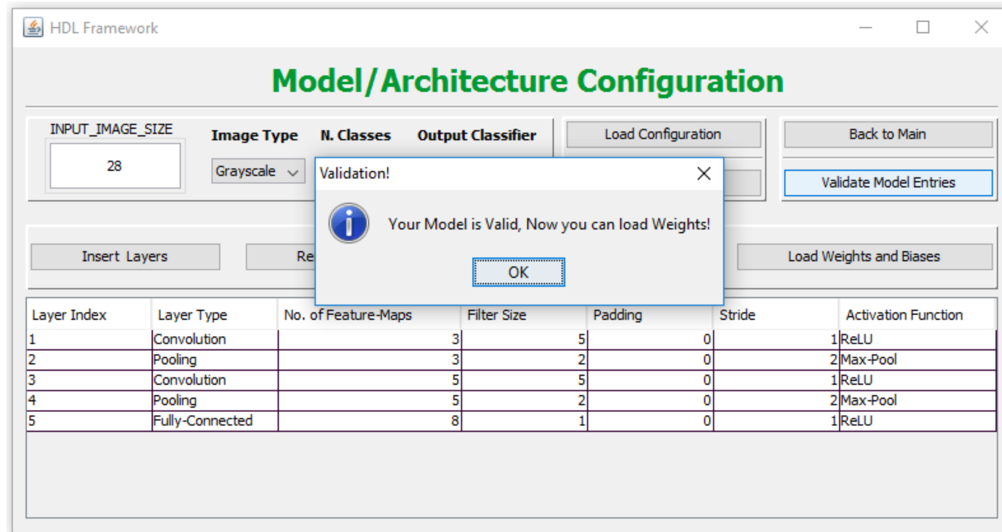

Figure 5.9 Successful configuration check

- **Parameters Inclusion Block**

In parameters inclusion block users are prompted to specify desired precision of target model, parameters representation, and parameters storage type. The tool supports three representations (decimal, hexadecimal, and binary) and the used representation in the generated VHDL is fixed point. The tool supports different precisions from 1-bit up to 32-bits. If users choose to the hardcoded-constants storage type, then parameters are consolidated within the generated VHDL code as part of the programmable logic (PL).

Parameters must be formatted according to model configuration in order to have a successful VHDL generation. In parameters file, users should specify layers name, list all kernels used in each feature map along with their weights, specify biases value, and end each line with a dollar sign. This will be explained in details in a later section. The sizes of weights and biases are specified in the GUI, so for our example the tool is expecting binary representation of a weight size of 5-bits and a bias size of 2-bits. If parameters file does not correspond to configuration, an error message will be displayed to the user highlighting the issue. Figure 5.10 illustrates the options given to incorporate parameters.

المنارة للاستشارات
www.manaraa.com

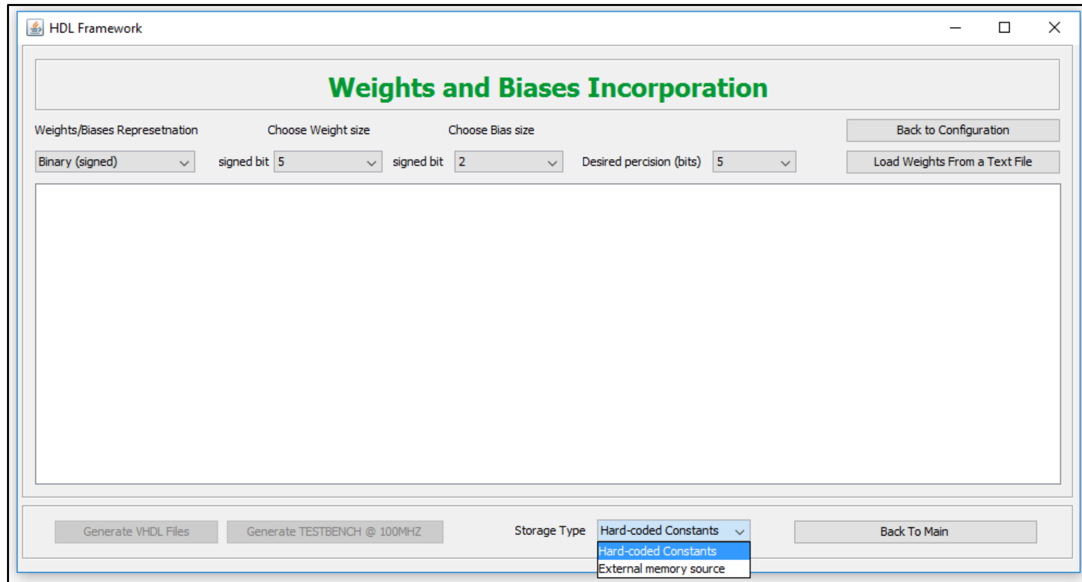

Figure 5.10 Parameters inclusion block for the example model

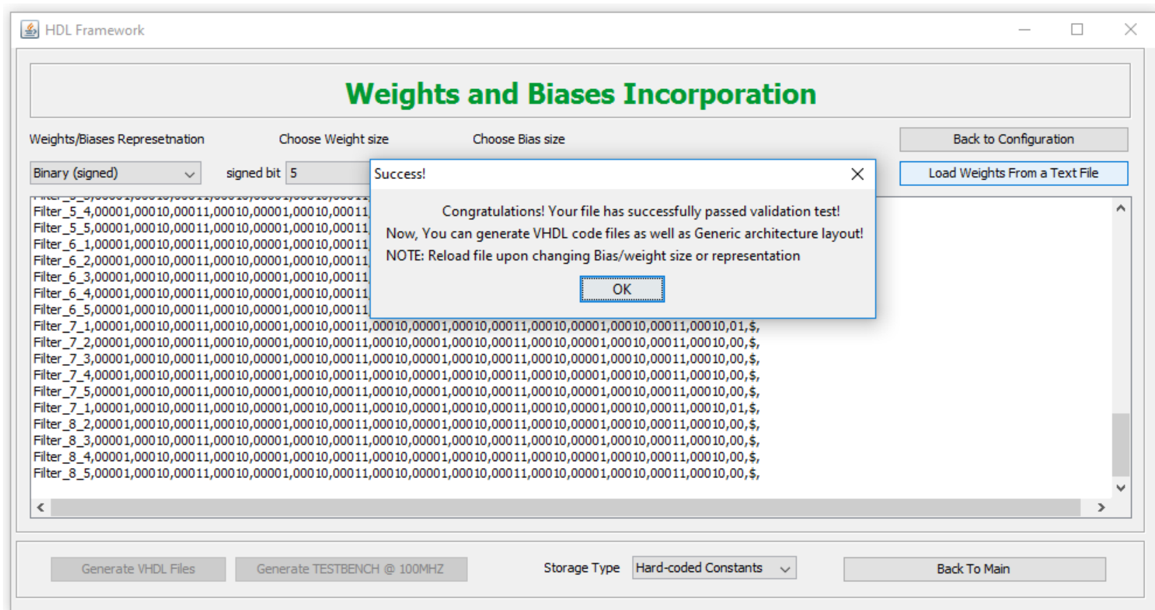

Figure 5.11 Successful parameters inclusion

On an unsuccessful incorporation of parameters from an external file, users are prompted with an error message stating what fixes they should do in order to proceed. On a successful load as shown in Figure 5.11, “Generate VHDL files and Generate test-bench” buttons are enabled and users can now generate VHDL code for the targeted model.

www.manaraa.com

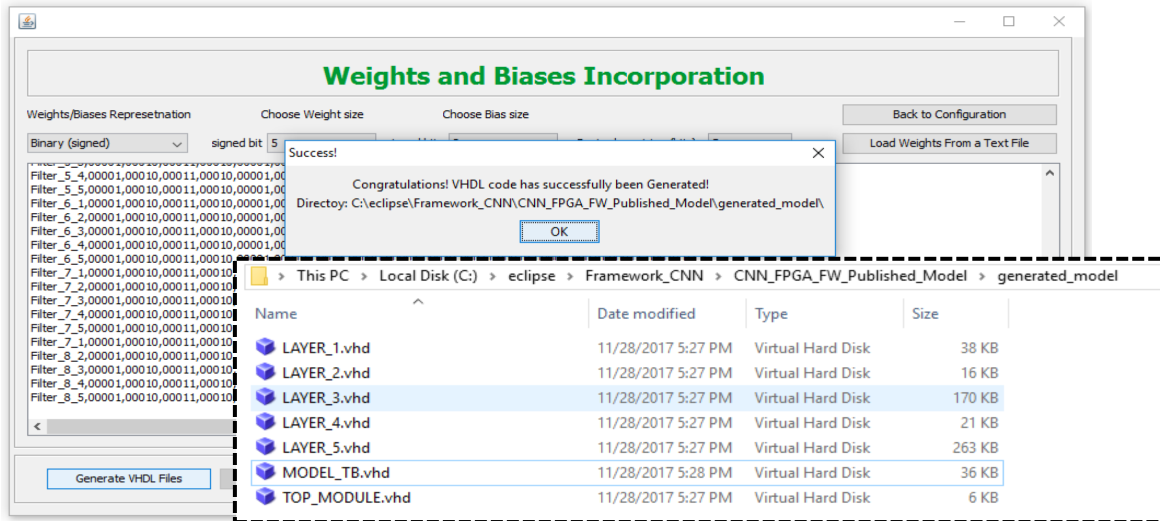

Figure 5.12 Generated VHDL files for VGTEST model

External Files/Dependencies

The VHDL generation process depends on two external files, one is required and the other is optional. The optional file is the external configuration file and the required file is the parameters file. The syntax for both files is explained in details in this subsection

- **Configuration File Syntax**

```
N_Layer,5,
Image_Size,28,
Image_type,Grayscale,8, or → Image_type,Colored,24,
N_Classes,2,
Classifier,SoftMax,
Convolution,3,5,0,1,ReLU, or → Sigmoid, Tanh
Pooling,3,2,0,2,Max-Pool, or → Avg-Pool
Convolution,5,5,0,1,ReLU,
Pooling,5,2,0,2,Max-Pool,
Fully-Connected,8,1,0,1,ReLU,
```

Features
Filter
Padding
Stride

Figure 5.13 Configuration file syntax of the example model

المنارة للاستشارات
www.manaraa.com

Figure 5.13 shows the configuration syntax for VGTEST, where **N_Layer** represents the number of layers in the network; **Image_Size** is the input image dimension; **Image_type** specifies the type of image if colored or grayscale and 8 represents the input data (pixels) width, where 24 is for colored and 8 is for grayscale; **N_classes** represents the number of output classes and **Classifier** is the used classifier function; **Convolution,2,2,0,2,Max pool** respectively represent layer name, number of output feature maps, filter size, padding, stride size, and used activation function; the same syntax applies to pooling and fully connected layers.

- **Configuration File Syntax**

Three Feature maps with filters of size 5x5
Weights → 5bit, Biases →2bits

```
Convolution,1
Fiter_1,00001,00010,00011,00010,00001,00010,00011,00010,00001,00010,00011,00010,00001,0
0010,00011,00010,00001,00010,00011,00010,00001,00010,00011,00010,00010,01,$
Filter_1_2,00001,00010,00011, …….. ,00010,01,$
Filter_1_3,00001,00010,00011, …….. ,00010,01,$
Pooling,1
Convolution,2
Filter_1_1,00001,00010,00011, …….. , 00010,01,$
Filter_1_2,00001,00010,00011, …….. ,00010,00,$
Filter_1_3,00001,00010,00011 , …….. , 00010,00,$
Filter_2_1,00001,00010,00011 , …….. , 00010,01,$
…
…
…
Filter_5_3,00001,00010,00011 , …….. , 00010,01,$
Pooling,2
Fully-Connected,1
Filter_1_1,00101 , …….. , 00111,01$
Filter_1_2,00111 , …….. , 00111,00$
Filter_1_3,00101 , …….. , 00111,00$
Filter_1_4,00110 , …….. , 00111,00$
Filter_1_5,00110 , …….. , 00111,00$
…
…
…
…
…
…
Filter_8_5,00110 , …….. , 00111,00$
```

5 x 3 Feature maps of size 5x5

8 x 5 Feature maps of size 4x4

Figure 5.14 Random parameters syntax file of the example model

المنارة للاستشارات
www.manaraa.com

Figure 5.14 shows the syntax of parameters file of our example model. Users must structure this file as follows: Start with layer name followed by its feature maps and filters with their weights and biases. Naming is case sensitive for layers, but not for filters. Must end lines with a dollar sign. This file should be consistent with the configured model in configuration stage as well as in parameters inclusion stage, otherwise it will not be accepted by the tool and will result in an error message. Possible reasons for not accepting a parameters file could be; invalid file extension or non-matching content; inconsistent filter size, number of layers/filters, or data representation; missing biases or dollar sign at the end of each line.

VHDL code Generation Process

This section illustrates the process of VHDL code generation using Java. The process of auto generation is based on the concept of parametrized design implementation. The configuration process is divided into three blocks; platform and network selection block, model configuration block, and parameters inclusion block.

- **Architecture constructor**

Architecture constructor is the core block of the generation tool, where it analyzes the specifications of configured CNN models and make design decisions based on these specifications. There are four modules that are connected to the architecture constructor, which are described as follows: graphical user-interface classes, parameterization library manager, VHDL code generator, and VHDL code storage.

- **Graphical User Interface**

Graphical user-interface provide the possible configurations of a valid Convolutional Neural Network to users and is divided into model configuration interface and parameters inclusion interface. In model configuration, a complete set of pre-defined layers configuration

المنارة للاستشارات

www.manaraa.com

are provided to help users easily build their models. Further, a sub-module for storing configuration is provided in this block. As for parameters inclusion, a set of -pre-defined parameters are provided in order to format weights and biases. Additionally, a complete parser is constructed to parse weights and biases in order to check their validity.

- **Parameterization Library Manager**

This module handles layer templets and supported functions in each templet. The main layers in a CNN are convolutional, pooling, and fully-connected layers, and there are three templets, one for each layer, that include needed functions to implement these layers. For example, a convolutional layer is composed of convolution operations followed by bias addition, and activation function operations. The convolution operation can be realized in different means that include but are not limited to, systolic array or sliding window. These two functions are stored in the convolutional layer templet and based on the incoming specifications that are passed by the architecture constructor, particular functions are selected and structured in a particular manner. Similarly, there are different activation functions that include but are not limited to, rectified linear unit, Tanh, or sigmoid, which can be formed/structured based on the specifications passed by the architecture constructor. This parameterization process also applies to pooling and fully-connected layers and that is according to their respective functions.

- **VHDL code generator and storage**

VHDL generation and storage is the last stage in the generation process. Upon passing model specifications to parametrization library manager, actual CNN layers are formed based on the available templets, hence generation process can take place. Once the parameterization process is complete, the code generator writes VHDL code and stores it in a designated folder relative to the location of the tool on hard-drive.

المنارة للاستشارات
www.manaraa.com

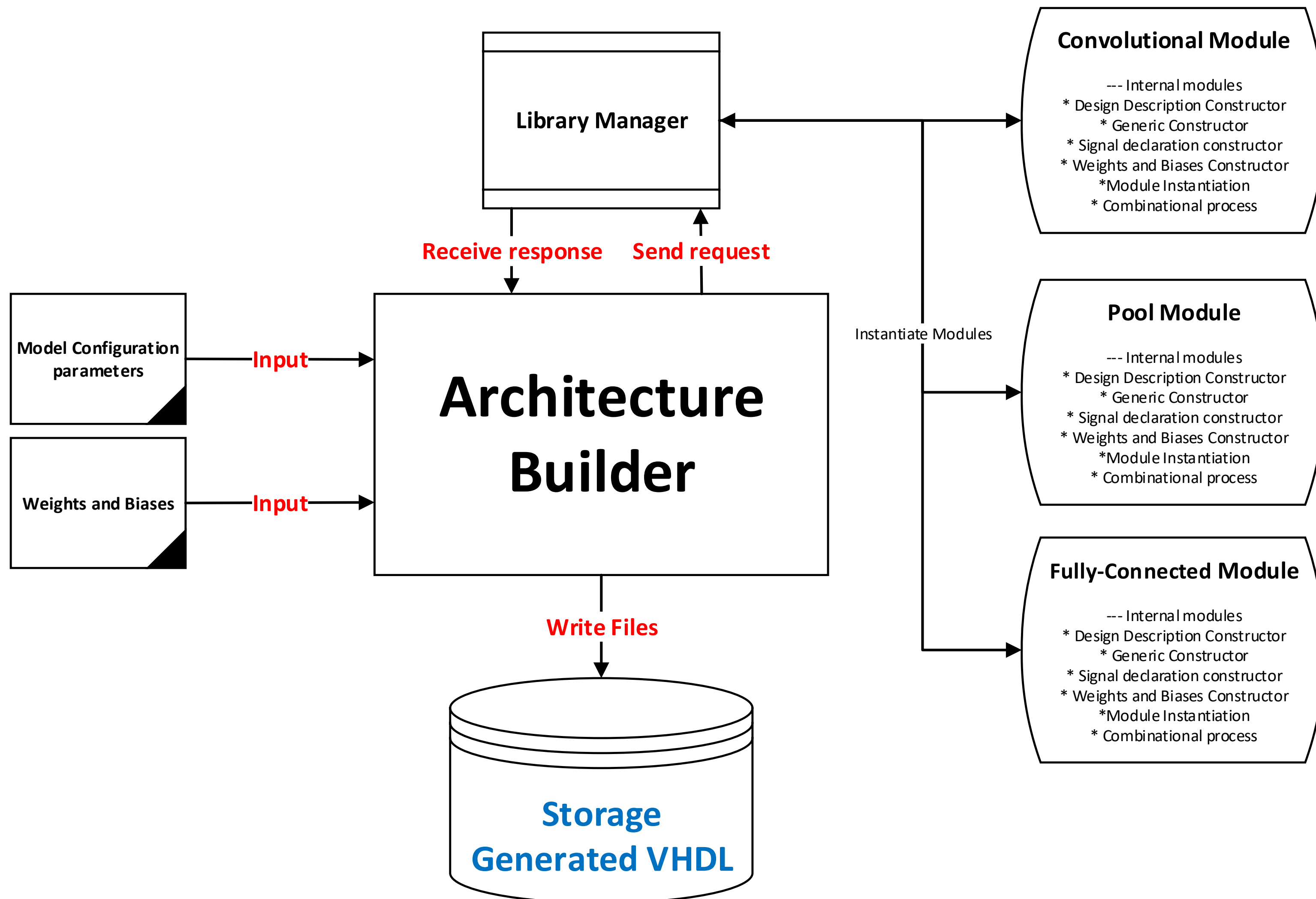

Figure 5.15 VHDL code generation process

المنارة للاستشارات
www.manaraa.com

Generated VHDL Details

This section explains the details of auto-generated VHDL, and provides example snapshots of the generated code for the first convolutional layer of VGTEST. The generated VHDL files are structured into six sections as shown in the following chart.

Sections	
	Design Details
	Generic Module
	Signals Declaration
	Weights and Biases
	Internal Module Instantiation
	Combinational Processes

Design details section is basically a header text that gives details about the generated network such as targeted FPGA platform, implemented network, performance estimation, and other useful guidelines to implement the network on FPGA.

```
--GENERATION DATE/TIME: Thu Dec 25 22:11:56 CST 2017
-- Engineer:    Muhammad Hamdan
-- Design Name:  HDL GENERATION - CONV LAYER
-- Module Name:  CONV_1 - Behavioral
-- Project Name:  CNN accelerator
-- Target Devices: Zynq-XC7Z020
-- Number of Operations: 30
-- Number of Clock Cycles: 6
```

Figure 5.16 Snapshot of generated VHDL code header section

Generic module specifies the parameters that can make the design reconfigurable without having to make significant changes to the design itself. For example, since FIFO size is parametrizable, the size can be changed through changing the representing constant of FIFO size in the generic module without re-writing any VHDL code. We provided this module in order to ensure reconfigurability of the design without having to reuse VGT to reconfigure the network by generating new VHDL code.

المنارة للاستشارات

www.manaraa.com

```
GENERIC (
  constant PERCISION    : positive := 5;
  constant DOUT_WIDTH    : positive := 5;
  constant BIAS_SIZE    : positive := 5;
  constant MULT_SIZE    : positive := 13;
  constant MULT_SUM_SIZE : positive := 6;
  constant DIN_WIDTH    : positive := 8;
  constant IMAGE_WIDTH   : positive := 13;
  constant F_SIZE      : positive := 2;
  constant WEIGHT_SIZE   : positive := 5;
  constant BIASES_SIZE   : positive := 2;
  constant STRIDE      : positive := 1;
  constant FEATURE_MAPS   : positive := 3;
  constant VALID_CYCLES   : positive := 144;
  constant STRIDE_CYCLES : positive := 12;
  constant VALID_LOCAL_PIX: positive := 12;
  constant ADD_TREE_DEPTH : positive := 2;
  constant INPUT_DEPTH   : positive := 1;
  constant FIFO_DEPTH    : positive := 12;
  constant USED_FIFOS    : positive := 1;
  constant ADD_1      : positive := 2;
  constant ADD_2      : positive := 1;
  constant LOCAL_OUTPUT   : positive := 5  );
```

Figure 5.17 Snapshot of generic module from model entity

```
---------------- ARCHITECTURE DECLARATION - START----------------------

architecture Behavioral of CONV_LAYER_1 is

------- INTERNAL FIXED CONSTANT & SIGNALS DECLARATION - START-----------
type    FILTER_TYPE  is array (0 to F_SIZE-1, 0 to F_SIZE-1) of
signed(WEIGHT_SIZE- 1 downto 0);
type    FIFO_Memory  is array (0 to FIFO_DEPTH - 1) of STD_LOGIC_VEC-
TOR(DIN_WIDTH - 1 downto 0);
type    SLIDING_WINDOW     is array (0 to F_SIZE-1, 0 to F_SIZE-1) of
STD_LOGIC_VECTOR(DIN_WIDTH- 1 downto 0);
signal   VALID_NXTLYR_PIX    :integer range 0 to STRIDE_CYCLES;
signal   PIXEL_COUNT       :integer range 0 to VALID_CYCLES;
signal   OUT_PIXEL_COUNT     :integer range 0 to VALID_CYCLES;
signal   EN_NXT_LYR_1      :std_logic;
signal   FRST_TIM_EN_1      :std_logic;
signal   Enable_MULT       :std_logic;
signal   Enable_ADDER      :std_logic;
signal   Enable_ReLU       :std_logic;
signal   Enable_BIAS       :std_logic;
signal   SIG_STRIDE       :integer range 0 to IMAGE_SIZE;
signal   PADDING_count      :integer range 0 to IMAGE_SIZE; --
```

Figure 5.18 Snapshot of signals declaration section

www.manaraa.com

Signals declaration is simply a section where all the design signals are defined. We put the signals all together in one section to ensure better readability and easy redefinition or addition of old or new signals if needed.

The weights and biases section handles hardcoding the weights and biases of the model. This section only applies to small-scale networks. In this section, all filters with their weights are stored in two dimensional matrices as constants followed by their respective biases.

```
------------- FILTER HARDCODED CONSTANTS -WEIGHTS START----------------
constant FMAP_1: FILTER_TYPE:=   (("00001","00010"),("00011","00010"));
constant FMAP_2: FILTER_TYPE:=   (("00001","00010"), ("00011","00010"));
constant FMAP_3: FILTER_TYPE:=   (("00001","00010"), ("00011","00010"));
constant BIAS_VAL_1: signed (BIASES_SIZE-1 downto 0):="01";
constant BIAS_VAL_2: signed (BIASES_SIZE-1 downto 0):="01";
constant BIAS_VAL_3: signed (BIASES_SIZE-1 downto 0):="01";
```

Figure 5.19 Snapshot of weights and biases section

Internal module instantiation section takes care of instantiating other layers since the code was based on a modular generation approach.

```
------------------ MAP NEXT LAYER - COMPONENTS START--------------------
COMPONENT POOL_LAYER_2
  port(  CLK,RST      :IN std_logic;
  DIN_1_2,DIN_2_2, DIN_3_2:IN std_logic_vector(LOCAL_OUTPUT-1 downto 0);
  VALID_OUT_2, EN_STREAM_OUT_2 :OUT std_logic;
  DOUT_1_2, DOUT_2_2 :OUT std_logic_vector(DOUT_WIDTH-1 downto 0);
  EN_STREAM ,EN_LOC_STREAM_2 :IN std_logic );
END COMPONENT POOL_LAYER_2;
begin
POOL_LYR_2 : POOL_LAYER_2
     port map(
     CLK           => CLK,
     RST           => RST,
     DIN_1_2         => DOUT_BUF_1_1,
     DIN_2_2         => DOUT_BUF_2_1,
     DIN_3_2         => DOUT_BUF_3_1,
     DOUT_1_2        => DOUT_1_2,
     DOUT_2_2        => DOUT_2_2,
     VALID_OUT_2       => VALID_OUT_2,
     EN_STREAM_OUT_2     => EN_STREAM_OUT_2,
     EN_LOC_STREAM_2     => EN_NXT_LYR_1,
     EN_STREAM        => EN_STREAM  );
```

Figure 5.20 Snapshot of internal module instantiation

المنارة للاستشارات
www.manaraa.com

Finally, the combinational process section is the actual implementation of the target layer. In this section, convolution, pooling, matrix multiplication, bias addition, adder tree, or activation function operations are applied. The combinational process takes care of processing data, and the synchronous process updates data every clock cycle.

Table IV Supported operations by the combinational process for CNN layers.

Function/Applicable Layer	CONV	POOL	First FC	Later FCs
Signals Reset	✓	✓	✓	✓
Matrix/Vector Multiply	✓	×	✓	✓
Max/Average Pooling	×	✓	×	×
Feature Maps Adder Tree	✓	×	✓	✓
Filter Values Adder Tree	✓	×	✓	×
Bias Addition	✓	×	✓	✓
Activation Function	✓	✓	✓	✓
Process End	✓	✓	✓	✓

المنارة للاستشارات
www.manaraa.com

CHAPTER 6. RELATED WORK

This chapter covers two main components, a thorough survey on hardware implementations of Convolutional Neural Networks and related FPGA-based implementations of Convolutional Neural Networks to this work.

Survey on Hardware Implementations of CNNs

Deep Neural Networks (DNNs) became popular algorithms recently in center-based services and standalone-embedded applications. A prominent type of DNNs that has attracted many researchers interest is Convolutional Neural Networks (CNNs). CNNs are used in various applications such as visual recognition, handwritten digit recognition, web search, speech recognition, etc. A huge work has been done over the years to improve the performance and increase the accuracy of CNNs to meet different application requirements. This made CNNs computationally very intensive. Thus, to accelerate CNNs and maintain their desired accuracy, an efficient implementation on hardware is needed. Because CNNs are naturally parallel, modular and dynamically adaptive, reconfigurable and custom architectures seem to be well suited for the job. A lot of work has been done in the area of CNNs acceleration, and many CNNs accelerators have been proposed for different purposes and with different techniques.

In this survey, we will present the hardware implementations (Accelerators) of CNNs in the following structure. Accelerators will be categorized into three primary platforms: (1) Custom Hardware Platform; (2) Graphics Processing Unit (GPU) Platform; (3) Field-Programmable Gate Array (FPGA) Platform. Under each platform, work of the same or similar objective will be grouped and presented all together.

المنارة للاستشارات
www.manaraa.com

Custom Hardware Platform

Architecture specialization is seen as a promising path to achieving high performance at low energy, provided it is possible to find ways to accommodate architecture specialization and flexibility. Designing a highly specialized and efficient hardware could likely benefit many of emerging high-performance applications [41].

DaDianNao, a custom multi-chip machine-learning accelerator implemented by Yunji, et al. [41], can outperform the NVIDIA K20M GPU by up to 450.65x, and reduce energy by up to 150.31x using 64 nodes. DaDianNao is based on DianNao, a small-footprint high throughput accelerator proposed by Chen et al [42]. The general architecture is a set of identical nodes, one per chip. Each node contains significant storage, especially for synapses, and neural computational units. Authors tackle bandwidth requirements issue through the following design principles: (1) create a fully distributed architecture; (2) create an asymmetric architecture where each node footprint is massively biased towards storage rather than computations; (3) transfer neurons values rather than synapses to minimize required external bandwidth; (4) break down the local storage into many tiles to enable high internal bandwidth.

While DianNao can perform 452 G-ops/s consuming only 485mW, DaDianNao can perform 5.58 T-ops/s with only a single node, consuming 15.97W for the whole chip. Having DaDianNao consuming this amount of power makes the system impractical to be implemented in mobile devices. ShiDianNao, an energy-efficient design of a visual recognition accelerator implemented by Du, Zidong, et al [43], is 60x more energy efficient than DianNao [42], where it can perform 194G-ops/s at its peak, consuming only 320.10 mW.

It is worth mentioning that designs in [41], [42], and [43] are hardwired, thus cannot efficiently adapt to different network sizes. Vinayak, et al. [44] present a scalable, low-power

المنارة للاستشارات
www.manaraa.com

coprocessor for enabling real-time execution of deep neural networks called nn-X (Neural Network Next). The system is composed of a coprocessor, a host processor (ARM Cortex-A9), and an external memory. The coprocessor has three main components: processing elements called collections, a system bus called memory router, and a configuration bus to control data flow. Each collection is comprised of one convolution engine, one pooling module, and one non-linear programmable operator. The convolution engine is implemented as fully pipelined logic and uses memory to cache incoming data. The non-linear operation computes a piecewise linear approximation of any arbitrary non-linear function. The performance of nn-X in deep learning applications peaks 200 G-ops/s while consuming less than 4 watts of power.

GPU Platform

Working with custom architectures requires special hardware skills that many researchers might not possess. Researchers who are familiar or well suited with high-level programming languages tend to accelerate CNNs using GPUs. GPUs are inexpensive, available in most recent computers, and easily programmable with standard development kits. Indeed, GPUs can achieve very high performance, but with high-energy consumption.

Fabian, et al [45] present a ConvNet GPU-based accelerator to tackle face detection under pose variation. The GPU is mainly used to explore and take advantage of the inherent possible parallelization of CNNs. On NVidia GeForce 8800 GT at 600MHz, the system processed 640×480 images at 209 - 497ms per frame on average for eight runtime measurements. In the same context, Li, Haoxiang, et al [46] also present a GPU-based accelerator for the same application under visual variations. They adopt a cascaded architecture that operates at multiple resolutions. The GPU can process VGA-resolution images at 100 FPS.

المنارة للاستشارات

www.manaraa.com

FPGA Platform

GPUs do not fit well in energy constrained and mobile embedded systems, because of their significant energy dissipation. Custom hardware offer a good performance and energy efficient solution at the disadvantage of significant fabrication cost and limited flexibility [47]. FPGA-based accelerators have attracted more and more attention of researchers because they have the advantages of good performance, high energy efficiency, fast development round, relatively moderate cost, and capability of reconfiguration [48]. A lot of work has been carried out to accelerate CNNs using FPGAs. Proposed CNN accelerators can be generally classified into two groups: computation engine optimization and memory system optimization.

- **Memory System Optimization**

Memory bandwidth bottleneck is a critical issue in the acceleration process. Overcoming this barrier can significantly improve the performance of CNN acceleration process.

Generic Memory System Optimization

A matrix multiplier based accelerator architecture was proposed by Yuran, et al [49] to accelerate the fully connected (FC) and convolutional (CONV) layers of a CNN. In their work they handle a couple of presented problems as follows: (1) use a stream mapper unit to handle the overhead of unrolling the convolutions to matrix multiplications; (2) use a prefetch unit structure to make the address stream to the external memory sequential; (3) optimize a blocking strategy to make matrix multiplications of different sizes perform efficiently. The accelerator consists of several processing-unit (PE) chains where each one has a stream-prefetcher, a stream mapper and a matrix multiplier where the latter accelerates the matrix multiplication in the CONV and FC layers. The stream store/load (S/L) loads operands to the PE chain and then stores the results. The stream mapper remaps the data stream to the stream S/L to unroll convolutions to matrix multiplication, and the stream prefetcher is used to ensure efficient external

www.manaraa.com

memory access. A host processor is used to handle workload except for the convolutional layers. It communicates with the accelerator through a system bus and they both share the external memory. Based on dual core ARM cortexA9 running at 800MHz (The host processor), and Zynqzq7045 FPGA chips using 1600 DSP48Es running at150MHz, the system achieved an average throughput of 77.8 GFLOPs.

Manoj, et al. [50] accelerate CNNs for speech recognition applications. They reduce total data transfer between layers through fusing the processing of multi-ConvNet layers (i.e. exploiting the locality in a convolution's data flow) to avoid using off-chip memory to store intermediate data between layers since the data is too large. The HLS tool (Xilinx Vivado HLS 2015.4.2) is used to transform C++ code into hardware and handle pipelining of the arithmetic units and DRAM transfers. The design employs loop transformations to reorder computations, increase throughput, and reduce data transfer. Loops are fully unrolled of dimensions Tm X Tn (adders and multipliers) and to be optimized to maximize the performance. The in, out, and weight arrays represent on-chip buffers for input, output, and weight data to reduce off-chip memory access. Copying data in or out of these buffers is done using double buffering to overlap data transfer with computation. Implemented on Virtex-7 and applied to AlexNet's first two layers, the fused layer archived 28% savings in off-chip data transfer, however, DSP, BRAMs, and about 50% in FPGA's LUTs and FFs increased. They claim that the increase is due to not fusing the non-linear layers. Further, the accelerator was applied to the first five convolutional layers of the VGG model. The fused layer accelerator minimized off-chip feature map data transfer, reducing the total transfer by 95%, from 77MB down to 3.6MB per image.

المنارة للاستشارات
www.manaraa.com

A memory-centric accelerator was developed by Maurice, et al [51] to improve performance without increasing memory bandwidth through a flexible memory hierarchy that supports complex data access through tiling. The accelerator uses BRAM-based multi-bank on-chip buffers to minimize the required bandwidth through data reuse. To ensure reconfigurability and programmability, a cluster of SIMD type of Multiply Accumulate (MACC) PE is used to accelerate the convolutions. Based on implementation on the Virtex-6 FPGA running at 150MHz, the accelerator shows a reduction of FPGA resources of up to 13x while maintaining the same performance.

Resource Utilization

Yongming, et al. [52] state that, the organization of computation modules in [51] depends on the number of output feature maps and their number of rows. Because both of these parameters can change drastically from layer to layer, an analogous resource underutilization problem occurs. Yongming, et al. present an accelerator where they partition FPGA resources into multiple convolutional layer processors (CLPs) to maximize resources utilization for a higher overall throughput and computational efficiency. A Typical CLP for a convolutional layer is structured as buffered inputs and weights that are forwarded to a vector dot product block, summed with previous output that is stored in an output buffer, and then stored in the output buffer. The accelerator operation timeline is segmented wherein each segment each CLP sequentially processes its layers. The segment ends when all CLPs finish. Applied to AlexNet on the Virtex-7 485T FPGA, the Multi-CLP accelerator yields a 3.33x higher throughput compared to the Single-CLP used in [47] using the same resources. The Multi-CLP achieved 99% dynamic utilization, where the single-CLP has dynamic utilization of less than 66%.

المنارة للاستشارات

www.manaraa.com

Chen, et al. [47] propose a design space exploration methodology for CNNs acceleration by optimizing both computation resources and external memory accesses. In this work, they only implement convolutional layers. They optimize the external memory transfers through data reuse. The computation engine is implemented as a tree-shaped poly-structure with 7 inputs from input feature maps, 7 inputs from weights, and one input from bias. 64 poly structures are duplicated for unrolling loop Tm. For efficient memory access, on-chip double-buffers are built to operate in a ping-pong manner to overlap data transfer time with computation. They use external data transfer engines to provide data transfer between the accelerator and the external memory and to isolate the accelerator from various platform and tool specific bandwidth features. Loop pipelining is applied to improve the system throughput by overlapping the execution of operations from different loop iterations. Implemented on Vivado HLS on VC707 board with Xilinx FPGA chip Virtex7 485t running at 100MHz, the accelerator achieved an overall performance of 61.62 GFLOPS.

Jiantao, et al. [53] state that optimization approaches that are done by [52] and [54] can be integrated with their work since they both work on the organization of computation units. Data reuse in the convolution layers is applied multiple times to reduce the bandwidth. FC layers weights are compressed through using Singular value decomposition (SVD). Floating-point numbers are converted into fixed-point ones. The FPGA programmable logic consist of a computing complex, on-chip buffers, a controller, and memory interface streaming engines (DMAs). The computing complex consists of PEs that do the computations of convolution, pooling, and FC layers and on-chip buffers prepare data to be used by the PEs and to store the results. The controller fetches instructions from an external memory and decodes them to orchestrate all modules except the DMAs on the PL. The DMAs transfer data and instructions

المنارة للاستشارات
www.manaraa.com

between the external memory on the processing system (PS) side and the on-chip buffers on the PL side. To fully utilize the bandwidth for FC computations a convolver complex in one of the PEs is used. They implemented VGG16-SVD on Zynq ZC706 running at150MHz, and achieved a frame rate at 4.45 fps using 16-bit quantization. The average performance of the convolutional layers and the full ConvNet is 187.8 G-ops/s and 137.0 G-ops/s.

- **Computation Engine Optimization**

Parallelization Exploration

A typical approach to optimize the computational engine is parallelism exploration, and there is a lot of work that has been carried out to optimize computation through this technique.

Generic Parallelization Exploration

Ning, et al. [55] present a multistage data-flow implementation of a complete CNN for high-speed object recognition. They use a 3D convolver that is connected with FIFO for convolution. A rectified linear unit is used as the activation function. The global summation is used to overcome regular multiplication in the FC layer to save memory and DSP resources. Accumulators are used to obtain the summation of each feature map. For memory access efficiency, the memory is used as a buffer to store the computation results and serve the input of the next layer and a recurrent ConvNet is used to improve the capability of object recognition. Based on Altera Stratix V 5SGSMD5K2F40C2 running at 130MHz, the design achieved a performance of 409.62 G-ops/s for image size of 32×32, consuming 1113.88 mW

An accelerator that advances the work in [47] was presented by Motamedi, Mohammad, et al [48]. They advance the work in [47] through the consideration of all sources of parallelism. A Parallel convolution engine (PCE) that is combined of parallel block multipliers with their corresponding adders, exploits intra-kernel parallelism in each convolution. Then, a combination of PCEs with their corresponding adders perform inter-kernel parallelism. Tiling

المنارة للاستشارات
www.manaraa.com

is used in the kernel and feature map levels to manage data transfer and increase performance. On-chip buffers are used to hold the necessary data. Implemented on the VX485T FPGA and applied to AlexNet with 32-bit float point precision, the accelerator achieved an overall performance of 84.2GFLOPs that is 1.9x speedup compared to work in [47].

Huimin, et al. [31] present a CNN accelerator with all layers working concurrently in a pipelined style to increase throughput. A batch-based computing method is implemented and applied on FC layers to increase the memory bandwidth utilization. Between each layer, there are two ping-pong buffers, where the former layer may write to/read from one of the ping-pong buffers while the next layer reads data from the other buffer. The ping-pong buffers are also utilized to store the intermediate data to handle the input data and weights, thus reducing data access workload. This work adopts the 3 types of parallelism (Intra-output parallelism, Inter-output parallelism and parallelism within a convolution operation) as proposed in [56]. Using Xilinx VC709 running at 156MHz and applied to AlexNet, the system achieved a peak performance of 565.94 G-ops/s and 391 FPS, consuming 30.2W by the FPGA board.

Systolic array implementations

Systolic array implementations seem to be a natural fit to CNNs because they are very efficient at filtering but are very inflexible. Systolic implementations support only convolutions up to the implemented kernel size [51].

Murugan, et al. [57] accelerate CNNs using a programmable massively parallel coprocessor coupled with off-chip memory. The coprocessor uses off-chip memory as a scratchpad to manage the large intermediate data between CNN layers. Memory load in this work is reduced through using low precision data by packing multiple works in every memory operation. They dedicate 20-bits fixed point for kernel weights and 16-bits for all other values. The system architecture is organized in parallel as a cluster of vector processing elements, which are arrays

المنارة للاستشارات
www.manaraa.com

of 2D convolvers. In this work, parallelism is mainly used within feature maps and convolution kernel. On Xilinx Virtex-5 LX330T running at 115MHz, the coprocessor processed 640×480 images with 16bit-pixel precision at 6 FPS, consuming 11 watts.

Srimat, et al. [56] present a dynamically configurable co-processor that automatically analyzes workloads and configure its hardware and software components to match the exact mix of different types of parallelism in the workload. In this work, they propose three types of parallelism, Intra-output parallelism, Inter-output parallelism and parallelism within a convolution operation; however, they apply only the first two mentioned types. The co-processor is a stateless processing core that consists of 20 2D-convolvers connected to an external memory and a memory subsystem that consists of three independent memory banks each has one single ported memory. No internal storage is used and an input switch is used to allow the convolvers to be dynamically grouped in different ways based on memory bandwidth; however, the size of the convolver is still fixed. Implemented on Virtex 5 SX240T FPGA running at 120MHz, the co-processor processed images of size 640 x 480 at 25 - 30 FPS rate and achieved a speedup of 4.8x compared to [58], consuming less than 14W.

Clément, et al. [58][59][59][59][58][58][59] developed an FPGA stream processor called CNP for real-time object recognition. The CNP contains a control unit, a parallel vector arithmetic, a logic unit (VALU), an I/O control unit, and a memory interface. The control unit is used to sequence the hardwired operations of the VALU (2D convolutions, spatial pooling/subsampling, point-wise non-linear functions, and other more general vector operators). Parallelization was achieved through an arbiter (multiplex/de-multiplex) that accesses the same memory location simultaneously through 8 FIFO buffered ports. All operations were performed with 16-bit fixed-point precision. On Xilinx virtex-4 SX35 FPGA running at 200MHz

المنارة للاستشارات
www.manaraa.com

consuming 15W, the system achieved a processing speed of 10 FPS, processing a full 512 × 384 greyscale images.

A study on the effect of limited precision data representation and computation on neural networks training was conducted by Suyog, et al [60]. This work is built upon the idea that algorithm-level noise tolerance can be leveraged to simplify underlying hardware requirements. The system consists of systolic array of multipliers, an on-chip memory configured as FIFO, and other controllers that orchestrate the movement of data and the communication with the off-chip memory within the FPGA. For a 28×28 systolic array implemented on KintexK325T FPGA, the Xilinx's Vivado synthesis tool estimated a maximum circuit operation frequency of 166MHz and a power consumption of 7W which translates to a throughput of 260G-ops/s at a power efficiency of 37 G-ops/s/W.

- **Scalable Architectures**

Clément, et al. present a runtime reconfigurable dataflow processor for vision called NeuFlow in [61] and [62]. This work is similar the work presented in [58], however, in [58] the architecture is presented as a data flow grid. This architecture is designed to process homogeneous streams of data in parallel, achieve high throughput and provide flexible processing framework. [63] and [61] architectures have the same components that perform the same tasks to some extent other than the latter architecture describes the used DMA as a smart DMA because it complements the work of the control unit. While consuming 10W when implemented on a Virtex-6 FPGA running at 200MHz, the system segmenting 20 categories on 500 × 375 frames at 12 FPS.

المنارة للاستشارات
www.manaraa.com

Paolo, et al. [64] propose a flexible and scalable architectural for CNNs acceleration based on the tightly-coupled cluster architectural paradigm followed by the PULP platform [65] that has been configured to suit the needs of CNNs acceleration and adapted to be implemented on a Zynq device. The accelerator presented in [64] is based on the cooperation between a set of software cores (SW) and a parallel convolution engine that communicate via a tightly coupled L1 shared scratchpad. The system is connected through an AXI cluster bus to a Zynq Processing System that hosts a dual core ARM-Cortex A9 and communicates with external storage. A shared tightly coupled data memory (TCDM) is implemented using dual-port block RAM primitives. This allows simultaneous access to the memory banks by the computation engine, the DMA, and the SW. Memory banks are partitioned into two sets, one dedicated to input features and one dedicated to output features. Each bank is partitioned into at least two sections to allow overlapping of computation and communication. The computation engine design is fully parametric and can be scaled in terms of a big set of parameters. Implemented on a Xilinx ZC-706 board running at 100MHz, the system delivers theoretical peak performance up to 80 GMAC/s, i.e. 160 G-ops/s at two ops per MA for 5×5 filters.

DLAU, a scalable deep learning accelerator unit presented by Chao, et al [66] to speed up the kernel computational parts of deep learning algorithms. Authors utilize the tile techniques to partition the large-scale input data, FIFO buffers to prevent data loss, pipelines to minimize memory transfer operations, and computing units reuse to implement the large-size neural networks. Implemented on XC7Z020 running at 200MHz, DLAU was able to achieve 19.2x and 36.1x speedups for 64×64 and 256×256 network sizes, respectively compared to Intel Core2 running at 2.3GHz, when DianNao [42] can achieve 117x for the 256×256 network. The accelerator consumes 234mW and the whole system consumes 1814mW.

المنارة للاستشارات
www.manaraa.com

Related Hardware Implementations of CNNs

The main drawback of accelerating a CNN on an FPGA platform is that developers have essentially to rebuild the CNN model from scratch, and that takes a long development round. A few implementations tackled this issue, for example, in [67] authors propose an FPGA framework that is based on Caffe framework [68] to map CNN layers to an FPGA platform. The framework mainly uses Xilinx FPGA SDAccel environment [69] to map CNN layers and generate the bit-stream file. To optimize the computational component, they increase the number of hardware units used to process a problem which in turns increase hardware resources linearly, making it an inefficient optimization method.

HLS tools such as OpenCL-Framework are a good alternative path away from low-level programming; however, such tools are not highly optimized to take full advantage of the available parallelism in CNNs and those tools abstract away a lot of the design details. In [70] authors use the OpenCL framework to implement the AlexNet model on P395-D8 board. Altera OpenCL SDK is used for compilation of OpenCL code to RTL to run on the FPGA accelerator. Running at 120MH the P395-D8 board achieved a peak performance of 72.4 GOPS; however, in our implementation of the same network, the system achieved peak performance of 611.54GOPs having the system running at 200MHz.

HDL automatic generation for convolutional neural networks was previously proposed. In [71], authors use high-level descriptive language to generate a Verilog HDL code for CNN models, where they specify the details of layers and generate each layer independently. Once all layers are generated, they combine all of them to have a complete accelerator. The generated code is generic for all different models scale wise. They did not state anywhere in their work that they store parameters on-chip or hard code them, meaning that they use an external memory source for small-scale models which is not an efficient way to handle parameters for

المنارة للاستشارات
www.manaraa.com

such models. Their accelerator can achieve a performance of 222.1 GOP/s for AlexNet, while ours can achieve 611.52 GOP/s for the same model. Further, our VHDL generation tool is designed to generate an optimized code/CNN implementation that is modular, scalable, reconfigurable, highly parallel, and fully pipelined.

In [72] authors avoid loading parameters from an external memory source by storing them in an on-chip memory. In their implementation, they adopt a parallel-serial style to increase the throughput; however, this strategy does not take full advantage of the available parallelism in the CNN as well as different layers do not work concurrently. They implemented a small-scale neural network that performs digits recognition on Xilinx XC7Z045. Under 172 MHZ, their system is capable of processing about 70K 28×28 images per second. In our implementation, we avoid using any sort of memory storage to store parameters, rather we hard code them as constants to maximize the utilization of the available hardware resources and reduce the use of the expensive ones, and get over memory bandwidth limitations. Our system is capable of processing up to 125K 28×28 Images/s, having the system running at 200 MHZ.

Optimizing computation in CNNs can significantly improve the overall performance of a CNN model. Many attempts have been made to optimize computation through various parallelism approaches. Authors in [57] and [73] use parallelism only in convolution operations and output feature maps. This work implements three types of parallelism: parallelism in convolution operations, parallelism in input feature maps, and parallelism in output feature maps. In addition, the design in this work is implemented in a pipelined style where all layers work concurrently that helped increase the throughput of the system, achieving a peak performance of 611.54 GOPs for AlexNet model.

المنارة للاستشارات
www.manaraa.com

CHAPTER 7. HARDWARE ARCHITECTURE

This chapter explains the details of the hardware architecture pieces which were used for small and large-scale models. The chapter is divided into two sections, the first section covers the details of small-scale architectures and the other section shows the architecture used to implement AlexNet model.

Small-Scale Models Architecture

Small scale models can be implemented on different FPGA boards based on their size and number of parameters they comprise. In this section we will target the Zedboard for our previously used example in CHAPTER 5. , VGTEST model.

System Architecture Overview

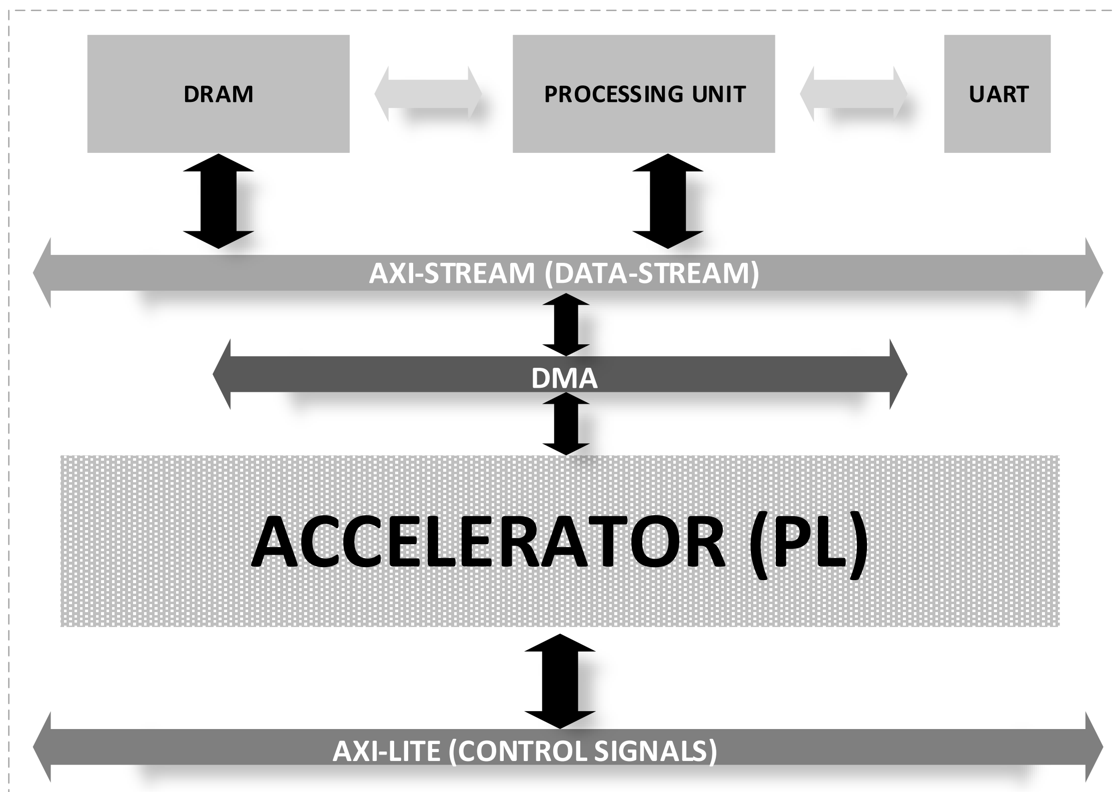

Figure 7.1 Top-level architecture of the system

المنارة للاستشارات
www.manaraa.com

Figure 7.1 describes the top-level architecture of the proposed system using the Zedboard. DRAM is used to store images of the target dataset; processing unit is used to perform classification operation and that is to avoid using expensive and complex FPGA operations like natural logarithm; AXI-Stream is used to increase the throughput of the system through continuous pixel streaming into the accelerator. Direct memory access(DMA) controller is used to manage communication or data transfer between the accelerator and the DRAM; Finally, the accelerator represents the developed core for the acceleration process. The details of the accelerator will be broken down into modules and presented in the following subsection.

Accelerator Architecture

The accelerator can be viewed as a combination of four different modules; top level interface module, used to interface with the AXI-stream; convolution module, used to perform computation and handle convolutional layers; pooling module, used to perform maximum or average pooling operation in pooling layers; and matrix multiplication module, used to perform matrix multiplication and handle fully-connected layers.

- **Convolution Module Architecture**

Convolutional layers account for most of the operations in the CNN, thus it is necessary to optimize the computational engine through maximizing parallelism and simplifying the computational operations. We designed a hardware architecture that takes advantage of the three parallelism techniques we mentioned in Generic Parallelization Exploration. The process in convolutional layers begins by streaming multiple or single vectors of data, then using a sliding window, we achieve the first parallelism technique through multiplying the input-data (receptive region) covered by the sliding-window (filter) parameters in a single clock cycle. To achieve the second technique of parallelism, each sliding window with its MACC operations is considered as a one processing element (PE) and the number of PEs is equal to the

المنارة للاستشارات
www.manaraa.com

number of input data vectors. The last parallelism technique is achieved through extracting multiple feature maps at the same time, where that is also achieved through the parallel PEs.

The complete process of a convolutional layer can be summarized as follows; First, stream pixels and perform convolution operation for multiple input data vectors; Second, add up the values of each filter (window) together, then add up all the input data vectors together in a pipelined style to form one data vector; Third, extract multiple feature maps from the unified data vector; Fourth, add biases to their corresponding extracted feature map; Fifth and last, apply ReLU activation function to all extracted feature maps, where it is basically a zero-thresholding operation.

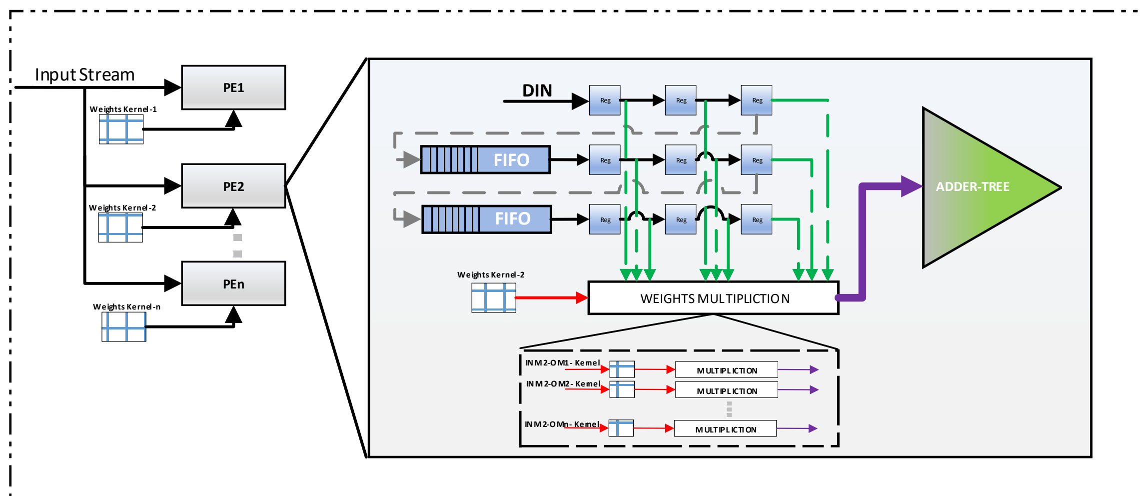

Figure 7.2 Processing element details in a convolutional layer for a 3 x 3 filter

Figure 7.2 shows the hardware architecture details of a processing element in a convolutional layer for an example filter of size 3×3. PEs are scalable to different filter sizes. INM2-OM1-kernel, is the kernel that includes the weights to extract sub-feature map 1 of feature map 1 from input map 2.

المنارة للاستشارات
www.manaraa.com

The architecture details of a complete convolutional layer are shown in Figure 7.3. The difference between the first convolutional layer and later convolutional layers is that the first layer might only have one processing element if the image is of grayscale type.

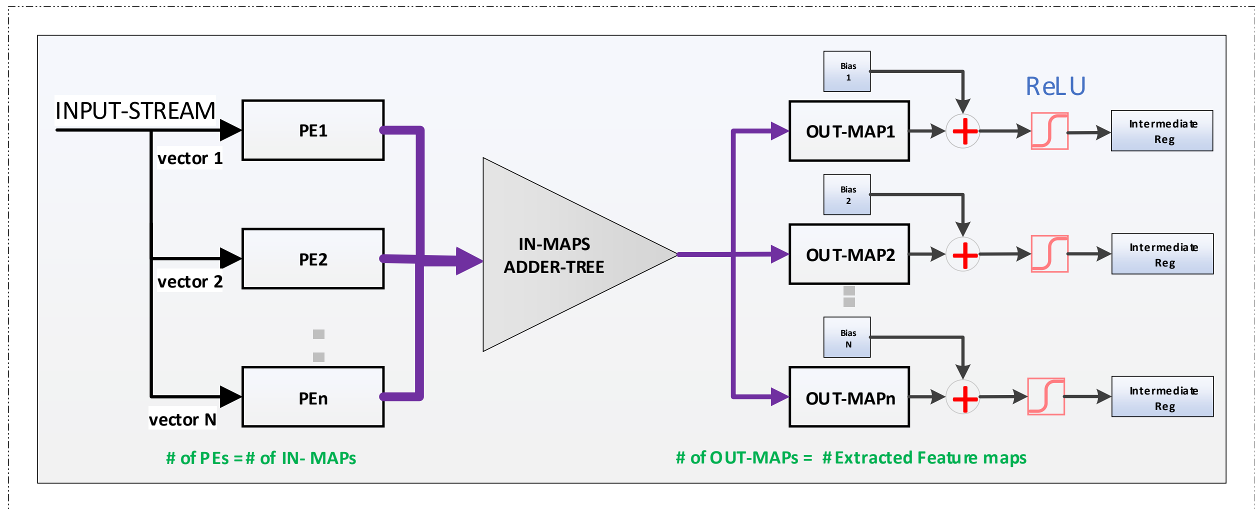

Figure 7.3 Hardware details of a complete convolutional layer

- **Pooling Module Architecture**

The architecture of pooling layer is the simplest. In fact, in this layer only one operation is performed, which is max or average pooling. Pooling layer takes up intermediate values stored in buffers from the previous layer and applies a sliding window that has the size of the pooling filter and a step size based on the specified stride value. This sliding window is similar to the one used in the convolutional layer, except that the performed operation is max pooling and no weights multiplication is performed. Results from max-pool are stored in buffers that feed the next layer. The max pooling operation is performed for all incoming feature maps in one clock cycle. Architecture details of the pooling layer are described in Figure 7.4.

المنارة للاستشارات
www.manaraa.com

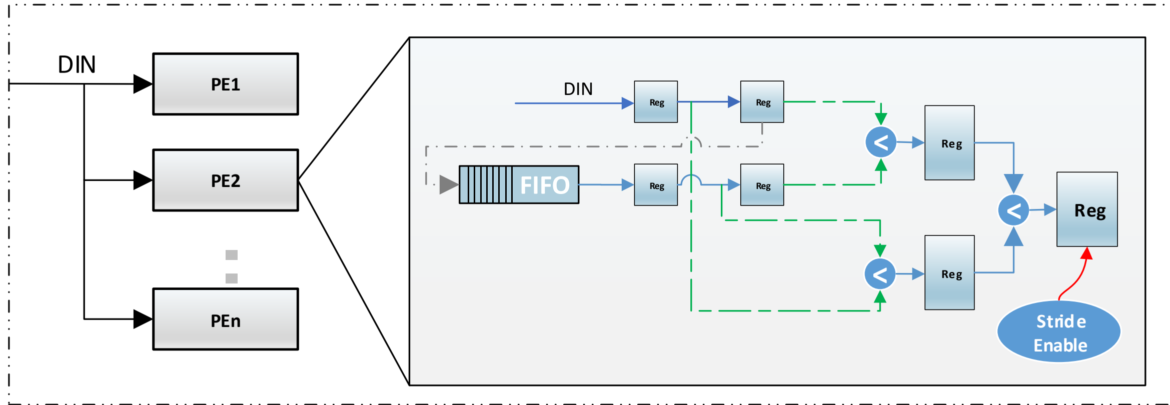

Figure 7.4 Max pooling architecture using filter size of 2×2

- **Matrix Multiplication (Fully-Connected Layer) Architecture**

The architecture of the FC layer is similar to the convolutional module architecture except that the convolution operation is replaced with matrix multiplication operation. The process starts by streaming data from the previous layer's intermediate buffers into the fully-connected layer. For the first FC layer, the sliding window has the size of the input feature map and that is to extract features for each neuron in the input feature map. For later FC layers, no sliding windows are used, and multiplication is applied directly. The architecture of a fully-connected layer is shown in Figure 7.5. The processing elements represent matrix multipliers.

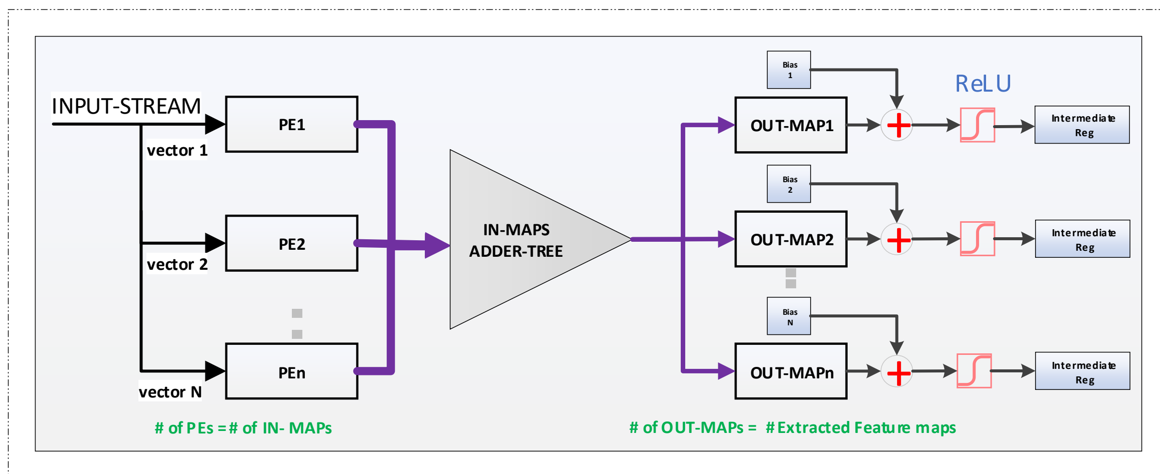

Figure 7.5 Hardware architecture of fully-connected layer

المنارة للاستشارات
www.manaraa.com

Large Scale CNN Architecture

The goal intended from implementing a large-scale CNN is to demonstrate the adaptability and capability of VGT. The tool generates VHDL code for the network layers; however, the top-level module is not covered yet by the tool. As a case study, we implemented the large scale AlexNet CNN [8].

The architecture of a large-scale CNN is slightly different from small-scale CNN architecture. Since we cannot hardcode parameters nor store them on an on-chip memory because of their massive size, we are ought to store them on an external memory source. This in fact, introduces a lot of complication in the process as we need to take into account memory accesses for loading parameters from the external memory source. The architecture of the convolutional and pooling layers is similar to those of small-scale CNNs, but the fully-connected layer architecture is different. The FC layer as we said earlier, accounts for most of the network parameters, hence we cannot simply perform a huge matrix multiplication operation.

As a matter of fact, the architecture of the fully-connected layer of AlexNet is adopted from the work in [31]. Authors use the same parallelism techniques we use for small-scale models and have similar approaches to what we have done for small scale networks, hence we followed their implementation of the large scale AlexNet.

The amount of parallelism in AlexNet is massive and is subject to the available hardware resources on FPGA. Authors in [31], introduced a parallelism space exploration approach to balance between parallelism utilization and the available hardware resources available on FPGA. Further, they proposed a decent methodology for optimizing memory bandwidth to achieve high performance. For more details, we recommend reading that paper.

المنارة للاستشارات
www.manaraa.com

Since the main difference between large and small-scale architectures lies within the fully-connected layer architecture, we will only describe the architecture of the fully connected layer. The first FC in AlexNet requires about 398 million multiplication operations. The weights matrix is of size ((6×6×256) × 4096) and the input vector is of size (1 × 9216). To perform such a massive matrix multiplication operation, the input vector should be divided into small and equal vectors ($1 \times X_n^i$), and weights matrix should also be divided into similar ($X_n^{ij} \times 1$) vectors. The multiplication operation is performed as shown in Equation 7.1. Results from the small vector multiplication are stored in a temporary output. When all multiplications for a complete input vector are done, final results are generated and stored in designated outputs: $Y^1 \rightarrow Y^j$. The multiplication operation is illustrated in Figure 7.6.

$$\sum_{j=1}^{m=4096} \sum_{i=1}^{k=\frac{9216}{n}} (1 \times X_n^i) * (X_n^{ij} \times 1) = Y_i^j \qquad (7.1)$$

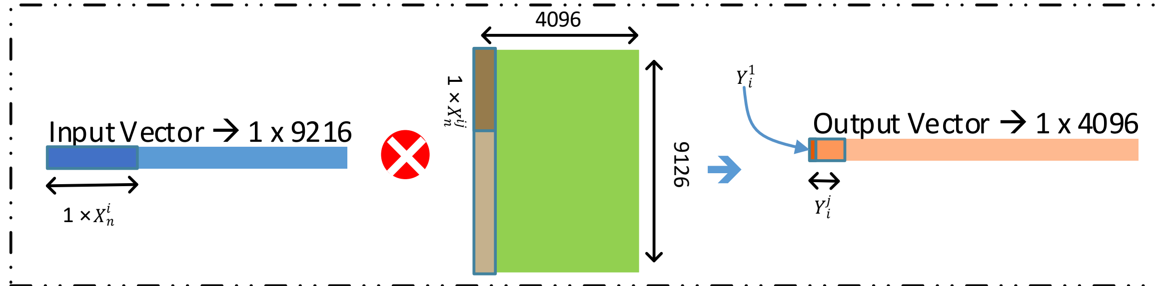

Figure 7.6 Fully-connected layer architecture of a large-scale CNN, (Adapted from [31] *)*

www.manaraa.com

CHAPTER 8. RESULTS AND EVALUATION

To demonstrate the functionality and scalability of VGT, we implemented two benchmarked models. This chapter illustrates the implementation details of LeNet-5 [18], and AlexNet [8]. As for AlexNet implementation as stated earlier in CHAPTER 7. , we adopted the strategy proposed in [31] in managing memory bandwidth and desired degree of parallelism.

Implemented Models Details

AlexNet implementation is not a fully-automated implementation, where the tool was only responsible for generating VHDL code for the layers without handling the storage of weights and biases. In AlexNet implementation, 16-bit fixed point precision is used for weights representation, and 8-bit and 16-bit fixed point precisions are used in LeNet-5 implementation.

LeNet-5 Model

LeNet-5 model comprises three convolutional layers, two pooling layers, and one fully connected layer. The number of parameters for the entire model is only ~1.25x times the parameters required for the first convolutional layer in AlexNet; however, this small model is good enough to perform digit recognition with decent accuracy.

Table V LeNet-5 model configuration

Layer	Dimensions	$Filter_{size}$	Feature Maps	Parameters
Input Image	28 × 28 × 1		-	-
CONV 1	24 × 24 × 1	5 × 5	6	156
POOL 1	12 × 12 × 1	2 × 2	-	0
CONV 2	8 × 8 × 1	5 × 5	16	2416
POOL 2	4 × 4 × 1	2 × 2	-	0
CONV 3	1 × 1 × 120	-	120	30840
FC 2	1 × 1 × 84	-	84	10164
Total	-	-	-	43576

المنارة للاستشارات
www.manaraa.com

Table V shows the details of LeNet-5 model; the operations performed in each layer, and the number of parameters each layer accounts for. The number of parameters shown in this table is different from what is reported in [18], where we consider input images of size 28×28 by which later feature maps sizes are changed. Further, we do not follow their strategy for extracting feature maps for C3, where we establish all connections. Although establishing all connections accounts for more parameters, but it actually improves accuracy and generalizes the model to be more parameterizable and similar to other generic CNN models. This indeed, puts away any special extra functions to be implemented for this single model.

Since the number of parameters in LeNet-5 is very small compared to AlexNet, we managed to have the parameters hard-coded. This strategy helped significantly improve the overall throughput of the system as we do not have to deal with external memory bottleneck for loading parameters, and it improved the overall utilization of hardware resources.

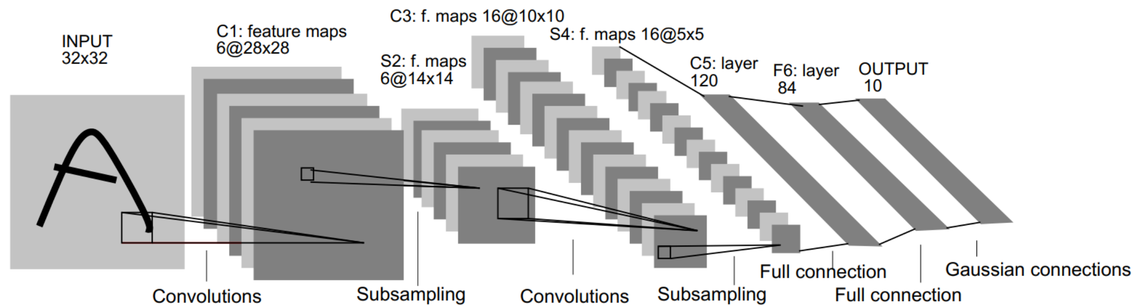

Figure 8.1 Original LeNet-5 architecture [18]

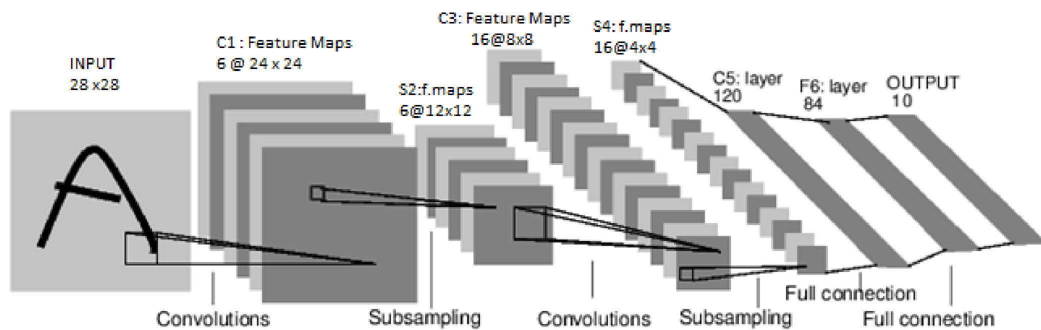

Figure 8.2 Implemented LeNet-5 architecture, (Adapted from [18] *)*

المنارة للاستشارات
www.manaraa.com

AlexNet Model

AlexNet model is one of the most prominent benchmarked large-scale CNN models. This model was proposed back in 2012 when it had won ImageNet Challenge (ILSVRC) 2012, achieving a top-5 accuracy of 84.7%, performing image classification of colored images of size 224x224 to 1000 different classes. Reported by the authors of AlexNet, the model takes between 5~6 days to be trained on two GTX580 3GB GPUs. This shows how large the model is, where it comprises about 60 million parameters. The model consists of five convolutional layers some of which are followed by max-pooling, and three fully-connected layers.

Hardcoding 60 million parameters is impractical because of the huge size those parameters account for, hence those parameters are stored in an external memory source. Table VI shows the details of AlexNet architecture, where **Layer** is the layer name, $\mathbf{IN_{Fs}}$ and $\mathbf{OUT_{Fs}}$ are the input and output feature maps, $\mathbf{Feature_{size}}$ is the size of a feature map, $\mathbf{Filter_{size}}$ is the size of filter that is used in the convolution operation, and **Stride** is the shifting stride of the used filer during the convolution operation.

Table VI AlexNet architecture details

Layer	IN_{Fs}	OUT_{Fs}	$Feature_{size}$	$Filter_{size}$	Stride	Parameters
Input Image	3		224×224			-
Convolution 1	96	96	55×55	11×11	4	34944
Pooling 1	96		27×27	3×3	2	-
Convolution 2	256	256	27×27	5×5	1	614656
Pooling 2	256		27×27	3×3	2	-
Convolution 3	256	384	13×13	3×3	1	885120
Convolution 4	384	384	13×13	3×3	1	1327488
Convolution 5	384	256	13x13	3×3	1	884992
Pooling 5	256		6×6	3×3	2	-
Fully-connected 6	4096	4096	1×1	--	--	37752832
Fully-connected 7	4096	4096	1×1	--	--	16781312
Fully-connected 8	1000	1000	1×1	--	--	4097000

المنارة للاستشارات
www.manaraa.com

The number of parameters of FC layers can be calculated as shown in Equation 8.1

$$FC_{parameters} = \text{weights} + \text{biases} \rightarrow {IN_{map}}_{size}{}^{2} \times IN_{map} \times OUT_{maps} + OUT_{maps} =$$

$$6^2 \times 256 \times 4096 + 4096 = 37752832 \quad (8.1)$$

Equation 8.1 can also be used to calculate the number of parameters for convolutional layers by replacing ${IN_{map}}_{size}{}^{2}$ with ${Filter_{size}}^{2}$, so the total number of parameters for the first convolutional layer = $11^2 \times 3 \times 96 + 96 = 34944$ parameter.

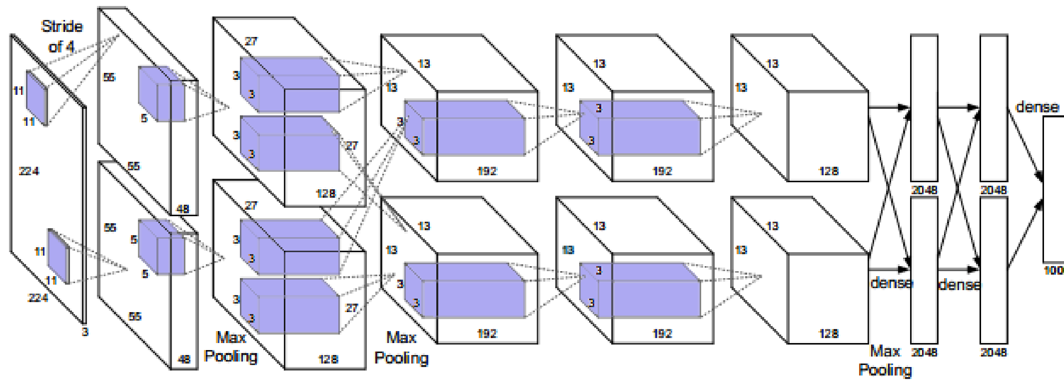

Figure 8.3 AlexNet architecture: ImageNet 2012 winning CNN model. (Adapted from [31] *)*

Results

The evaluation of the implemented models is based on simulation, synthesis and post implementation results obtained from ISE [74] and Vivado [37] tools .

LeNet-5 Model

We simulated LeNet-5 model using ISE and Vivado simulators, using test weights and biases. Simulation results of LeNet-5 are shown in Figure 8.4. Results matched up in behavioral, synthesis, and post implementation simulations. We verified the correctness of the obtained results from simulation by implementing the network in MATLAB.

المنارة للاستشارات
www.manaraa.com

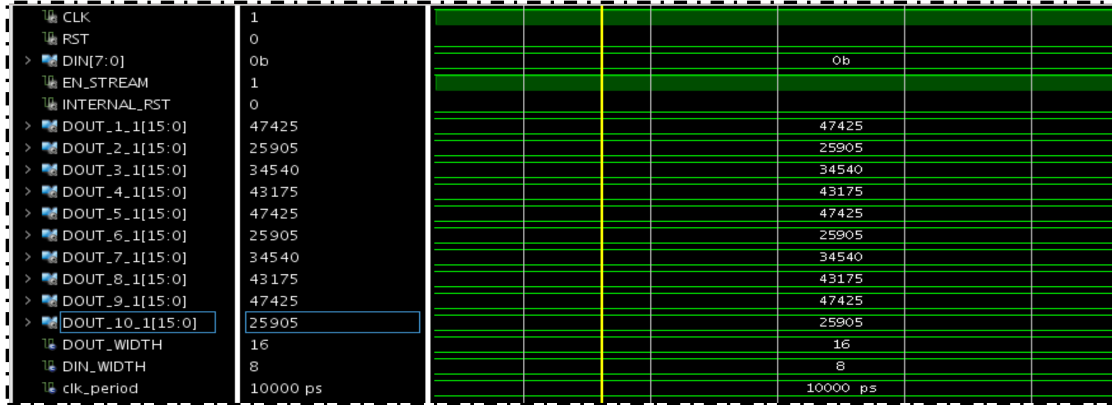

Figure 8.4 Post-implementation simulation results of LeNet-5 using 16-bit precision

Hardcoding model parameters as constants does not only help overcome memory bottleneck, but it also can optimize the use of DSP blocks. To test this hypothesis, we implemented LeNet-5 model with 8-bit precision using two different versions of Xilinx synthesis tools, having them running at their default synthesis/implementation design goal.

We found out that Vivado optimized out all DSP48 blocks, where no blocks were reported in the post-implementation report, while ISE reported a total DSP48 block utilization of 89% with a total reduction of 33% in other hardware resources compared to Vivado utilization report. Table VII shows utilized resources of LeNet-5 using 8-bit precision.

Table VII Hardware resource utilization of 8-bit LeNet-5 implementation on Zynq xc7z020

Resources	**Slice Registers**		**LUTS**		**DSP**	
Available Resources	**106400**		**53200**		**220**	
Layers/Tools	**Vivado**	**ISE**	**Vivado**	**ISE**	**Vivado**	**ISE**
Conv-1 (C1)	1448		953		0	
Pool-2 (S2)	1516		747		0	
Conv-1 (C1)	6892		4905		0	
Pool-4 (S4)	1844	**13839**	1057	**10015**	0	**196**
Conv-1 (C1)	5731		6112		0	
FC6 (F6)	1738		2661		0	
Total	**19169**		**16435**		0	
Utilization	18.02%	13.01%	30.89%	18.83%	0%	89.09%

المنارة للاستشارات
www.manaraa.com

Table VIII shows hardware resource utilization when implementing LeNet-5 on Virtex-7 (7vx690tffg1157). For both used bit representations, ISE tool used DSP48 blocks, and the used DSP48 blocks are 57% more than those were used in the Zedboard despite the fact that the used model and parameter representation are exactly the same in both cases.

Table VIII Resource utilization of LeNet-5 implemented on Virtex-7 using 8 and 16-bit

Resources	**Slice Registers (FF)**		**LUTS**		**DSP48 Blocks**	
Available	**866400**		**433200**		**3600**	
Precision	8-bit	16-bit	8-bit	16-bit	8-bit	16-bit
Used	13501	27530	9778	17162	309	553
Utilization	1.56%	3.18%	2.26%	3.96%	8.58%	15.36%

In conclusion, hardware utilization in the case of hardcoded constants can be optimized to target particular hardware resources by specifying special synthesis directives to synthesis tools. In this implementation, ISE synthesis optimization targeted performance, where the synthesis report showed a maximum operational frequency of 387MHz when implementing 8-bit LeNet-5 on the Zedboard and 434MHz when implemented on Virtex-7. The variation in speed here is because more DSP48 blocks were used in Virtex-7.

On the other hand, Vivado synthesis optimization targeted area, where we have seen less hardware resources used for both implementations, 8-bit and 16-bit LeNet-5, yet this was on the account of having the design running at maximum operational frequency of 200MHz, when the same design ran at maximum speed of 434MHz through ISE.

We evaluated LeNet-5 implementation by comparing it with other related work and a software implementation of our own. Evaluation is shown in Table IX.

المنارة للاستشارات

www.manaraa.com

Table IX LeNet-5 implementation in comparison to other related

Implementation	Platform	Frequency (MHZ)	FPS (28×28)	Speedup
Software	Intel® Core™ i7-6700HQ	2600	2.5K	baseline
[75]	Xilinx XC7Z045	172	70K	28x
This work	Virtex7	200	125K	50x

AlexNet Model

AlexNet was implemented on Virtex-7 only due to its large size. Table X Shows synthesis results of hardware resource utilization of AlexNet, different implementations of AlexNet model in comparison to this implementation are illustrated in Table XI , and Table III shows related implementations that are based on HDL generation.

Table X Resources Utilization by AlexNet model.

Resources (VirtexVC709)	FF	LUTS	DSP	BRAM
Available	866400	433200	3600	2940
used	269845	287461	2070	2023
Utilization	31.14%	66.35%	57.5%	68.8%

Table XI Comparison with other implementations of AlexNet model

	Platform	Frequency (MHZ)	GOP/s	Processing time (ms)
[76]	Altera Stratix-V	120	136.5	20.1
[31]	Virtex7- VX690T	156	565.9	2.56
[77]	Stratix-V GXA7	100	114.5	>12.5
[47]	Virtex7-VX485T	100	61.62	21.61
This work	Virtex7- VX690T	200	611.5	2.41

Table XII Comparison with other automatic HDL generation implementations

	Platform	Frequency (MHZ)	GOP/s / GMACs	Model
[67]	Virtex7- VX690T	200	45.8 GOP/s	AlexNet
[75]	Virtex 7-VX485T	150	16.42 GMAC/s	LeNet-5
[71]	Virtex7- VX690T	100	222.1 GOP/s	AlexNet
This work	Virtex7- VX690T	200	611.5 GOP/s	AlexNet

المنارة للاستشارات
www.manaraa.com

CHAPTER 9. CONCLUSION AND FUTURE WORK

In this work, we designed and implemented an FPGA-based VHDL generation tool (VGT) for Convolutional Neural Networks implementation. The tool was developed in Java, and is designed to facilitate the process of hardware acceleration of Convolutional Neural Networks models using FPGAs through parametrizing the implementation of those models.

The tool offers a graphical user-interface through which users can on the fly configure their target CNN model by providing model specifications. VGT reduces development time needed to implement a CNN significantly, overcomes barriers introduced by the complexity of development in hardware descriptive languages, and mitigates under-optimization caused by high level synthesis tools. The tool is optimized to generate a modular, scalable, reconfigurable, and parallel implementation of CNN models.

We demonstrated our VHDL generation tool by implementing a small-scale "LeNet-5" CNN model and a large-scale one "AlexNet" on virtex-7. Having the FPGA running at 200 MHz, the system is capable of processing up to 125K images of size 28×28 for the small-scale model and achieved a peak performance of 611.52 GOP/s for the large scale one.

Small-scale CNN models utilize what we call "hardcoded constants approach" in handling CNN parameters "weights and biases". This indeed, contributed to improving the overall performance of implemented models, and offered more flexibility with synthesis tool implementation in terms of area and performance optimization strategies. In one hand, we were able to optimize all DSP blocks out, where multiplication operations where replaced by shift register operations, for implementations ran on optimized area strategy. On the hand, DSP blocks we fully utilized for implementations ran on optimized performance strategy.

المنارة للاستشارات
www.manaraa.com

Thus far, the proposed tool does not provide smart automated decisions for used FPGA platform and CNN model. Hence, we aim to extend this work through incorporating a design space exploration methodology which will mainly handle matching provided CNN model and desired implementation strategy with the adequate FPGA platform. In other words, if a model is constrained by area, power, or performance, then the tool will generate an implementation that will meet those design constrains and choose the target FPGA platform that best suits the application. Moreover, we aim to support more benchmarked CNN models and other neural networks algorithms such as recurrent neural networks. Lastly, we might extend the generated implementation output form to include C or C++ languages besides VHDL to allow users, in exceptional cases, advance their implementation to meet their special desired implementation needs/constrains without having to hustle with hardware descriptive languages, i.e. VHDL.

المنارة للاستشارات
www.manaraa.com

REFERENCES

[1] Y. Bengio, R. Ducharme, P. Vincent, and C. Janvin, "A Neural Probabilistic Language Model," *J. Mach. Learn. Res.*, vol. 3, pp. 1137–1155, 2003.

[2] R. Collobert and J. Weston, "A unified architecture for natural language processing," *Proc. 25th Int. Conf. Mach. Learn. - ICML '08*, vol. 20, no. 1, pp. 160–167, 2008.

[3] A. Coates, A. Arbor, and A. Y. Ng, "An Analysis of Single-Layer Networks in Unsupervised Feature Learning," *Aistats 2011*, pp. 215–223, 2011.

[4] Q. V Le, A. Coates, B. Prochnow, and A. Y. Ng, "On Optimization Methods for Deep Learning," *Proc. 28th Int. Conf. Mach. Learn.*, pp. 265–272, 2011.

[5] G. E. Dahl, D. Yu, L. Deng, and A. Acero, "Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition," *IEEE Trans. Audio, Speech Lang. Process.*, vol. 20, no. 1, pp. 30–42, 2012.

[6] G. Hinton *et al.*, "Deep Neural Networks for Acoustic Modeling in Speech Recognition," *IEEE Signal Process. Mag.*, vol. 29, no. 6, pp. 82–97, 2012.

[7] J. L. F. Pereira and R. J. F. Rossetti, "An integrated architecture for autonomous vehicles simulation," *Proc. 27th Annu. ACM Symp. Appl. Comput. - SAC '12*, pp. 286–292, 2012.

[8] A. Krizhevsky, I. Sutskever, and G. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," pp. 1–9, 2012.

[9] G. Lacey, G. W. Taylor, and S. Areibi, "Deep Learning on FPGAs: Past, Present, and Future," 2016.

[10] "MACHINE LEARNING." [Online]. Available: http://www.mlplatform.nl/what-is-machine-learning/. [Accessed: 01-Apr-2017].

المنارة للاستشارات

www.manaraa.com

[11] P. G. Papadourakis, “Introduction To Neural Networks.”

[12] L. Bottou, “Large-Scale Machine Learning with Stochastic Gradient Descent,” *Proc. COMPSTAT’2010*, pp. 177–186, 2010.

[13] L. Bottou, “Stochastic Gradient Learning in Neural Networks,” *Proc. Neuro-Nımes*, vol. 91, no. 8, 1991.

[14] “SoftMax Classifier.” [Online]. Available: http://knet.readthedocs.io/en/latest/softmax.html. [Accessed: 07-Jul-2017].

[15] “CS231n Convolutional Neural Networks for Visual Recognition.” [Online]. Available: http://cs231n.github.io/convolutional-networks/. [Accessed: 01-Jan-2017].

[16] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet Classification with Deep Convolutional Neural Networks,” *Adv. Neural Inf. Process. Syst.*, pp. 1–9, 2012.

[17] D. Scherer, A. M??ller, and S. Behnke, “Evaluation of pooling operations in convolutional architectures for object recognition,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 6354 LNCS, no. PART 3, pp. 92–101, 2010.

[18] “LeNet-5.” [Online]. Available: http://yann.lecun.com/exdb/lenet/

[19] J. Dong, M. Lin, Y. Wei, Q. Chen, H. Lai, and S. Yan, “ Network in Network,” 2014.

[20] F. N. Iandola, K. Ashraf, M. W. Moskewicz, and K. Keutzer, “FireCaffe: near-linear acceleration of deep neural network training on compute clusters,” *2016 IEEE Conf. Comput. Vis. Pattern Recognit.*, pp. 1–13, 2016.

[21] J. Qiu *et al.*, “Going Deeper with Embedded FPGA Platform for Convolutional Neural Network,” *Proc. 2016 ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays - FPGA ’16*, pp. 26–35, 2016.

المنارة للاستشارات
www.manaraa.com

[22] C. Szegedy *et al.*, "Going deeper with convolutions," *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, vol. 07–12–June, pp. 1–9, 2015.

[23] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," *Arxiv.Org*, vol. 7, no. 3, pp. 171–180, 2015.

[24] *Digital Integrated Circuit design*. .

[25] coep.vlab.co.in, "Introduction to FPGA and Verilog Programming. Retrieved 26 December 2017," 2011. [Online]. Available: http://coep.vlab.co.in/?sub=29&brch=88&sim=228&cnt=1. [Accessed: 01-Jan-2017].

[26] "Texas-instruments White paper." [Online]. Available: http://www.ni.com/white-paper/6984/en/. [Accessed: 01-Jan-2017].

[27] "Xilinx-ASIC." [Online]. Available: https://www.xilinx.com/fpga/asic.htm. [Accessed: 01-Jan-2017].

[28] M. Abadi *et al.*, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems," 2016.

[29] "2015- Deep Learning with Limited Numerical Precision." .

[30] "LightNN : Filling the Gap between Conventional Deep Neural Networks and Binarized Networks," pp. 2–7.

[31] Huimin Li, Xitian Fan, Li Jiao, Wei Cao, Xuegong Zhou, and Lingli Wang, "A high performance FPGA-based accelerator for large-scale convolutional neural networks," *2016 26th Int. Conf. F. Program. Log. Appl.*, pp. 1–9, 2016.

[32] M. K. Hamdan and D. T. Rover, "VHDL generator for a high performance convolutional neural network FPGA-based accelerator," in *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2017, pp. 1–6.

المنارة للاستشارات
www.manaraa.com

[33] S.-C. Liu, T. Delbruck, J. Kramer, G. Indiveri, and R. Douglas, *Analog VLSI: Circuits and Principles*. MA, USA: MIT Press Cambridge, 2002.

[34] M. C. Herbordt *et al.*, "Achieving high performance with FPGA-based computing," *Computer (Long. Beach. Calif).*, vol. 40, no. 3, pp. 50–57, 2007.

[35] D. Gschwend, "ZynqNet : An FPGA-Accelerated Embedded Convolutional Neural Network," no. August 2016.

[36] Xilinx UG998, "Introduction to FPGA Design with Vivado High-Level Synthesis," 2013. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/ug998-%0Avivado-intro-fpga-design-hls.pdf. [Accessed: 01-Jan-2017].

[37] Xilinx UG902, "Vivado Design Suite User Guide, High-Level Synthesis," 2016. .

[38] P. Release, "Xilinx buys high-level synthesis EDA vendor," 2016. [Online]. Available: http://www.eetimes.com/document.asp?doc_id=1258504. [Accessed: 01-Jan-2017].

[39] J. Cooley, "Cadence to acquire Forte Cynthesizer at a rumored fire sale price.," 2014. .

[40] "Vivado High-Level Synthesis." [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html. [Accessed: 07-Aug-2017].

[41] Y. Chen *et al.*, "Dadiannao: A machine-learning supercomputer," *Microarchitecture (MICRO), 2014 47th Annu. IEEE/ACM Int. Symp.*, pp. 609–622, 2014.

[42] T. Chen *et al.*, "DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning," *Proc. 19th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, pp. 269–284, 2014.

المنارة للاستشارات
www.manaraa.com

[43] Z. Du *et al.*, “ShiDianNao: Shifting Vision Processing Closer to the Sensor,” *Isca*, pp. 92–104, 2015.

[44] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello, “A 240 G-ops/s mobile coprocessor for deep neural networks,” *IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit. Work.*, pp. 696–701, 2014.

[45] F. Nasse, C. Thurau, and G. a Fink, “Face Detection Using GPU-Based Convolutional Neural Networks,” *Comput. Anal. Images Patterns*, pp. 83–90, 2009.

[46] L. Haoxiang and Lin, “A Convolutional Neural Network Approach for Face Identification,” *IEEE Conf. Comput. Vis. Pattern Recognit.*, pp. 5325–5334, 2015.

[47] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, “Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks,” *Proc. 2015 ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays - FPGA ’15*, pp. 161–170, 2015.

[48] M. Motamedi, P. Gysel, V. Akella, and S. Ghiasi, “Design Space Exploration of FPGA-Based Deep Convolutional Neural Networks,” *21st Asia South Pacific Des. Autom. Conf.*, pp. 575–580, 2016.

[49] L. Adhianto *et al.*, “FPGA-accelerated deep convolutional neural networks for high throughput and energy efficiency,” *Concurr. Comput. Pract. Exp.*, vol. 22, no. 6, pp. 685–701, 2010.

[50] C. Draft and D. N. O. T. Distribute, “Fused-Layer CNN Accelerators,” 2016.

[51] M. Peemen, A. A. A. Setio, B. Mesman, and H. Corporaal, “Memory-centric accelerator design for convolutional neural networks,” *2013 IEEE 31st Int. Conf. Comput. Des. ICCD 2013*, pp. 13–19, 2013.

المنارة للاستشارات
www.manaraa.com

[52] Y. Shen and M. Ferdman, “Maximizing CNN Accelerator Efficiency Through Resource Partitioning,” vol. 1.

[53] J. Qiu *et al.*, “Going Deeper with Embedded FPGA Platform for Convolutional Neural Network,” *Proc. 2016 ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays - FPGA ’16*, pp. 26–35, 2016.

[54] Y. Shen, M. Ferdman, and P. Milder, “Overcoming resource underutilization in spatial CNN accelerators,” *2016 26th Int. Conf. F. Program. Log. Appl.*, pp. 1–4, 2016.

[55] N. Li, S. Takaki, Y. Tomiokat, and H. Kitazawa, “A Multistage Dataflow Implementation of a Deep Convolutional Neural Network Based on FPGA For High-Speed Object Recognition,” pp. 165–168, 2016.

[56] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, “A dynamically configurable coprocessor for convolutional neural networks,” *ACM SIGARCH Comput. Archit. News*, vol. 38, no. 3, p. 247, 2010.

[57] M. Sankaradas *et al.*, “A Massively Parallel Coprocessor for Convolutional Neural Networks,” *Icasap*, pp. 53–60, 2009.

[58] C. Poulet, J. Y. Han, and Y. Lecun, “CNP : AN FPGA-BASED PROCESSOR FOR CONVOLUTIONAL NETWORKS Cl ´,” vol. 1, no. 1.

[59] C. Farabet, C. Poulet, and Y. LeCun, “An FPGA-based stream processor for embedded real-time vision with convolutional networks,” *2009 IEEE 12th Int. Conf. Comput. Vis. Work. ICCV Work. 2009*, pp. 878–885, 2009.

[60] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, “Deep Learning with Limited Numerical Precision,” *Proc. 32nd Int. Conf. Mach. Learn.*, vol. 37, pp. 1737–1746, 2015.

المنارة للاستشارات
www.manaraa.com

[61] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. Lecun, “NeuFlow: A runtime reconfigurable dataflow processor for vision,” *IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit. Work.*, 2011.

[62] Y. Lecun *et al.*, “Large-Scale FPGA-based Convolutional Networks Chapter in Machine Learning on Very Large Data Sets ,” *Mach. Learn. Very Large Data Sets*, pp. 1–26, 2011.

[63] C. Farabet, B. Martini, P. Akselrod, S. Talay, Y. LeCun, and E. Culurciello, “Hardware Accelerated Convolutional Neural Networks for Synthetic Vision Systems,” *Proc. Int. Symp. Circuits Syst.*, pp. 257–260, 2010.

[64] P. Meloni, G. Deriu, F. Conti, I. Loi, L. Raffo, and L. Benini, “Curbing the roofline,” *Proc. ACM Int. Conf. Comput. Front. - CF ’16*, pp. 376–383, 2016.

[65] D. Rossi *et al.*, “A 60 GOPS/W, -1.8 v to 0.9 v body bias ULP cluster in 28 nm UTBB FD-SOI technology,” *Solid. State. Electron.*, vol. 117, pp. 170–184, 2016.

[66] C. Wang, Q. Yu, L. Gong, X. Li, and I. Y. Xie, “DLAU: A Scalable Deep Learning Accelerator Unit on FPGA,” vol. XX, no. X, pp. 1–5, 2016.

[67] R. DiCecco, G. Lacey, J. Vasiljevic, P. Chow, G. Taylor, and S. Areibi, “Caffeinated FPGAs: FPGA Framework For Convolutional Neural Networks,” *arXiv*, 2016.

[68] Y. Jia *et al.*, “Caffe: Convolutional Architecture for Fast Feature Embedding,” *ACM Int. Conf. Multimed.*, pp. 675–678, 2014.

[69] “Xilinx Inc. SDAccel Development Environment User Guide.” [Online]. Available: https://www.xilinx.com/support/documentation-navigation/development-tools/software-development/sdaccel.html. [Accessed: 07-Aug-2017].

المنارة للاستشارات

www.manaraa.com

[70] N. Suda *et al.*, "Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks," *Proc. 2016 ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays - FPGA '16*, pp. 16–25, 2016.

[71] Z. Liu, Y. Dou, J. Jiang, and J. Xu, "Automatic Code Generation of Convolutional Neural Networks in FPGA Implementation," in *International Conference on Field-Programmable Technology (FPT)*, 2016, pp. 61–68.

[72] J. Park and W. Sung, "Fpga Based Implementation of Deep Neural Networks Using on-Chip Memory Only," *Icassp 2016*, pp. 1011–1015, 2016.

[73] S. Cadambi, A. Majumdar, M. Becchi, S. Chakradhar, and H. P. Graf, "A programmable parallel accelerator for learning and classification," *Proc. 19th Int. Conf. Parallel Archit. Compil. Tech. - PACT '10*, p. 273, 2010.

[74] Xilinx, "ISE Design Suite." [Online]. Available: https://www.xilinx.com/products/design-tools/isim.html. [Accessed: 01-Jan-2017].

[75] Y. Zhou and J. Jiang, "An FPGA-based accelerator implementation for deep convolutional neural networks," *Proc. 2015 4th Int. Conf. Comput. Sci. Netw. Technol. ICCSNT 2015*, no. Iccsnt, pp. 829–832, 2016.

[76] "2016- Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale." .

[77] Y. Ma, N. Suda, Y. Cao, J. S. Seo, and S. Vrudhula, "Scalable and modularized RTL compilation of Convolutional Neural Networks onto FPGA," *FPL 2016 - 26th Int. Conf. Field-Programmable Log. Appl.*, 2016.

المنارة للاستشارات
www.manaraa.com