Imperial College
London

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Optimising Convolutional Neural Networks for Reconfigurable Acceleration

Author:
Ruizhe Zhao

Supervisor:
Prof. Wayne Luk

Abstract

Convolutional Neural Network (CNN) is one of the most popular deep learning technique that has been used in many tasks, including image classification and machine translation. FPGA is a promising target hardware platform to deploy CNN models, because it has balanced performance and can be integrated with many platforms, from embedded devices to data-centre servers. Even though, FPGA is still less popular than GPU and CPU regarding CNN model deployment platform, which mainly due to the difficulty lies in converting high-level CNN descriptions to runnable FPGA hardware designs. This report aims at addressing this problem and provides two hardware libraries for constructing CNN on FPGA, one (RubyConv) is written in high-level language Ruby and the other one (MaxDeep) is written in OpenSPL. This report also presents a CNN model transpiler framework, Plumber, that can directly transform high-level models to FPGA designs with a novel model-hardware co-optimisation module.

The evaluation shows that the design generated and model-hardware co-optimised by the Plumber transpiler framework can achieve competitive performance.

Acknowledgements

I would like to sincerely thank Professor Wayne Luk for his help throughout my MRes study. His ideas are always insightful and practical, and he is always willing to give me all kinds of suggestions. Without his help, I can never successfully complete this report, two MRes research projects, two conference papers, and one conference poster. I would also like to thank Dr Xinyu Niu for his suggestions and discussions on my research topics and technical details. Moreover, I am grateful to many people in the Custom Computing Group, including Dr Timothy Todman, Paul Grigoras, and Nils Voss, who kindly taught me advanced knowledge in FPGA and the Maxeler platform.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Objective . . . 2
- 1.3 Challenges . . . 2
- 1.4 Contributions . . . 3
- 1.5 Report Organisation . . . 4

2 Background **5**
- 2.1 Deep Learning and Deep Neural Networks . . . 5
  - 2.1.1 Principles of Deep Learning . . . 5
  - 2.1.2 Deep Neural Networks . . . 6
- 2.2 Ruby and Neural Network . . . 9
  - 2.2.1 Ruby Basics . . . 9
  - 2.2.2 Neural Network in Ruby . . . 11
- 2.3 Efficient CNN Processing . . . 11
  - 2.3.1 Algorithmic Optimisation . . . 12
  - 2.3.2 Hardware Architecture . . . 14
  - 2.3.3 Network Model . . . 18
  - 2.3.4 Framework . . . 19
- 2.4 Summary . . . 19

3 RubyConv **20**
- 3.1 CNN Relations . . . 21
  - 3.1.1 Basic Relations and Functions . . . 21
  - 3.1.2 Layer Relations . . . 24
- 3.2 Evaluation . . . 27
  - 3.2.1 Symbolic Simulation . . . 28
  - 3.2.2 Case Study: LeNet-5 . . . 31
- 3.3 Summary . . . 31

4 MaxDeep **32**
- 4.1 OpenSPL-based Description . . . 33
  - 4.1.1 Convolution Layer . . . 34
  - 4.1.2 Binarised Convolution Layer . . . 37
  - 4.1.3 Depthwise Separable Convolution Layer . . . 39
  - 4.1.4 Other Layers . . . 41
  - 4.1.5 Network . . . 41
- 4.2 Analysis Model . . . 43

4.2.1 Datasheet: BRAM and DSP . . . 44
4.2.2 Statistical Method: LUT and Flip-Flop . . . 45
4.2.3 Performance . . . 46
4.2.4 Binarised and Depthwise Separable Convolution Layer . . . 47
4.2.5 Multiple Layers Analysis Model . . . 48
4.2.6 Summary . . . 49
4.3 Evaluation . . . 49
4.3.1 Single Convolution Layer Evaluation . . . 50
4.3.2 Binarised and Depthwise Separable Convolution . . . 55
4.3.3 Two Convolution Layers Evaluation . . . 57
4.4 Summary . . . 57

5 Plumber **58**
5.1 Dataflow Graph . . . 59
5.1.1 Nodes and Edges . . . 59
5.1.2 Implementation . . . 60
5.2 Plumber Transpiler . . . 60
5.2.1 Frontend: Model Parser and Optimisation . . . 61
5.2.2 Backend: Software Code and Hardware Design Generation . . . 61
5.2.3 Model-Hardware Co-Optimisation Module . . . 62
5.3 Evaluation . . . 62
5.4 Summary . . . 65

6 Conclusion and Future Work **66**
6.1 Summary of Achievements . . . 66
6.2 Comparison with MRes Projects . . . 67
6.3 Future Work . . . 67

Chapter 1

Introduction

In recent years, *Deep Learning* has become one of the most important and influential *Machine Learning* (ML) technologies. Deep Learning is mainly based on deep neural networks, among which *Convolution Neural Network* (CNN) is the one that has been applied to many different tasks, ranging from computer vision to machine translation. Because of the application domain of CNN is broad and essential, people are trying to push the performance boundary of CNN forward on various hardware platforms, including Graphic Processing Unit (GPU), Field-Programmable Gate Array (FPGA), Application-Specific Integrated Circuit (ASIC), and Central Processing Unit (CPU). Comparing these platforms, FPGA is the one that balances customizability, power-efficiency, performance, and prototyping speed. and is the promising platform to deploy CNN models for many use cases, such as robots and Unmanned Aerial Vehicles (UAVs).

However, it is not straightforward to deploy an arbitrary CNN model on an FPGA device. Programming on FPGA is already a difficult task, and programming a complex CNN computation logic on FPGA with acceptable performance is much harder. This issue greatly slows down the process from a newly trained CNN model to a runnable accelerator of that model on FPGA.

Resolving this difficulty is the major target of this report. In this report, we address this problem by proving a CNN acceleration hardware library, *MaxDeep*, and a CNN model transpiler framework, *Plumber*, together with an auxiliary high-level hardware description of CNN in Ruby, *RubyCNN*.

In this chapter, we first present the motivation of this report (Section 1.1) and the expected outcome (Section 1.2). Then, we mention the challenges to fulfil our goal (Section 1.3) and how they are addressed (Section 1.4). The organisation of this report is presented in the end (Section 1.5).

1.1 Motivation

Seamlessly deploying a trained CNN onto a runnable FPGA design is a feature in high demand. One reason is that researchers and developers need to immediately know whether their newly developed and trained CNN model can perform well on FPGA. Traditionally, model research group should hand over their model to the hardware team and wait for a long time until the model can be deployed and tested. This time-consuming loop should be broken to enhance the development efficiency. The other reason is more intuitive: comparing with platforms like GPU and CPU, FPGA does not have an easy-to-use deep learning *framework*, like Caffe (Jia et al., 2014) or TensorFlow (Abadi et al., 2016), that directly performs CNN inference by passing model definition files or writing several lines of code. Thus, a framework that can accelerate CNN model deployment on FPGA platforms is a topic that is

worth working on and beneficial.

In addition, unlike GPU and CPU that have fixed architecture, FPGA is more flexible and architectural impact from a slight change in the CNN model is possibly profound. It will be helpful if the feedback information of generated hardware design architecture can automatically give directions on CNN model selection, which can further become a model-hardware co-optimisation process. Therefore, the framework that will be presented in this report should better have a co-optimisation module.

In short, it is a necessary feature to transform a trained CNN model into FPGA hardware design, with an automatic co-optimisation based on feedback information.

1.2 Objective

According to the motivation above, this report aims at providing a framework that can end-to-end convert high-level CNN models into FPGA hardware designs with sufficient optimisation. To be specific, there are three major objectives in this report:

1. Describe essential building blocks of a CNN accelerator concisely in a high-level hardware description language, Ruby. It helps unravel the characteristics of CNN building blocks, such as configuration parameters, design metrics, and expected performance.

2. Devise a hardware library that implements CNN building blocks based on Ruby description. This library should be parameterised and flexible enough to construct most kinds of CNN models. Hardware designs built by this library should be able to synthesized and generated for real FPGA platforms. We choose the Maxeler platform and OpenSPL as the targeting FPGA platform and hardware description language. Also, the performance and resource usage of this library should be predictable only by design parameters, for efficient design space exploration and optimisation.

3. Design a transpiler that transpiles high-level CNN model description into hardware design. This transpiler should contain a dataflow graph IR for portability to different CNN description and different hardware description languages. This transpiler can also work together with a model-hardware co-optimisation module, which can provide optimisation guidance on the CNN model from hardware performance and resource usage feedback. and is an additional objective of this report.

Figure 1.1 presents the relation ship between the three objectives of this report. The whole report is about CNN framework on FPGA platforms. The top node is the transpiler that converts CNN models to hardware designs, and the bottom two nodes are candidates. RubyConv is a target hardware library, to which the input CNN model can only be partially transpiled, because RubyConv doesn't contain either platform specific or peripheral logic. But the partially transpiled RubyConv designs can be further converted to MaxDeep, which is a ready-to-use transpilation target, if the corresponding toolchain can perform well.

1.3 Challenges

To achieve the objectives of this report, we need to overcome three main challenges:

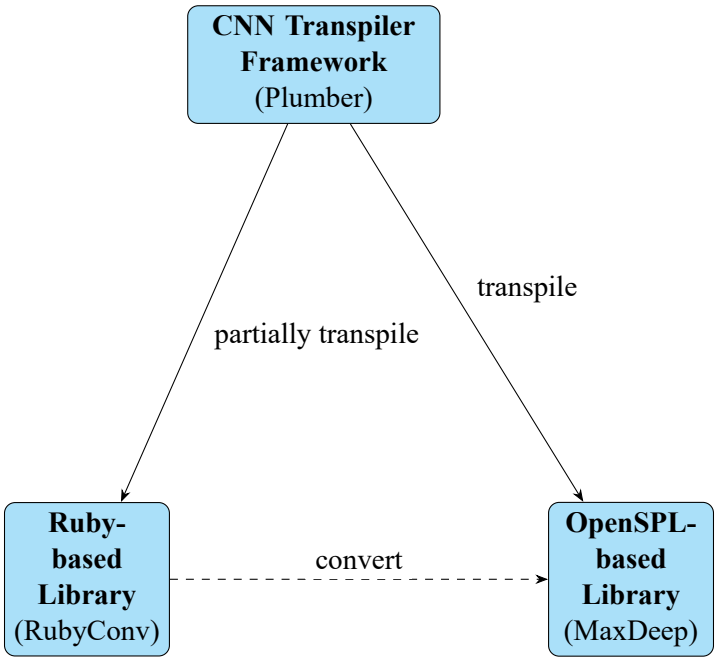

Figure 1.1: Relations among objectives in this report.

1. It is difficult to describe complex CNN architecture into parameterised hardware designs, written in high-level functional hardware description language Ruby. Building blocks in Ruby should not only process CNN computation correctly, but also be reusable and flexible to construct designs for different CNN topologies.

2. Devising a hardware library for CNN on FPGA platforms is also a challenging objective. For a single convolution layer, it is already a hard enough task achieving the highest performance and handling all its corner cases. It becomes even harder when considering constructing, connecting, and optimising a CNN accelerator design from general building blocks in the hardware library. Moreover, there are many variations of CNN, such as depthwise separable and binarised convolution, which should be integrated with the library.

3. The transpiler for transforming CNN models to FPGA hardware design is still a tough field of study. It needs a background in both deep learning framework, compiler principles, and low-level hardware design library. Also, the model-hardware co-optimisation has not been well-studied yet, and we need to propose several heuristic ideas to make it work.

1.4 Contributions

This report makes the following contributions to address the challenges mentioned above:

1. A Ruby-based CNN description library, *RubyConv*, is presented together with latency and resource usage analysis, and symbolic simulation evaluation. The possibility to generate a whole CNN hardware design from *RubyConv* on real FPGA hardware is also illustrated.

2. A hardware library to construct CNN accelerator design on the Maxeler FPGA platform, MaxDeep, is devised. MaxDeep contains essential CNN building blocks, such as convolution, fully-connected, max-pooling, and batch-normalisation layers. Besides, MaxDeep also supports

depthwise separable convolution and binarised convolution, to further enhance performance. The performance and resource usage of MaxDeep-constructed CNN hardware designs are highly-predictable, because most building blocks are associated with accurate analysis models. Besides, a constrained optimisation solver based performance optimisation flow for any CNN design constructed by MaxDeep is also presented with detailed evaluation of many cases.

3. Plumber, a CNN model to FPGA hardware design transpiler is another contribution. Plumber takes CNN model trained by TensorFlow as input, and generates hardware designs constructed by MaxDeep. The core of the Plumber transpiler is a dataflow graph intermediate representation, which contains sufficient annotated information about both high-level operator configuration and low-level design parameters. The generated hardware design and corresponding performance and resource usage information will be further proceeded by a model-hardware co-optimisation module, which optimises the input CNN model to generate better hardware design.

4. Evaluation on Maxeler MAX4 platform shows that, a single convolution layer generated by MaxDeep can reach GOp/s. And an end-to-end evaluation of MaxDeep and Plumber of LeNet shows that

1.5 Report Organisation

The rest of this report is organised as follows:

1. Chapter 2 presents essential background knowledge related to deep learning and CNN, and state-of-the-art techniques in hardware accelerator design for CNN.

2. Chapter 3 illustrates Ruby-based building blocks in *RubyConv*.

3. Chapter 4 introduces the MaxDeep library and its performance evaluation.

4. Chapter 5 contains the Plumber transpiler and the model-hardware co-optimisation framework, and their evaluation.

5. Chapter 6 discusses conclusions of this report and possible future work.

Chapter 2

Background

This chapter aims at providing knowledge for main chapters of this paper. This chapter first reviews the essential background of deep learning and convolutional neural networks (Section 2.1). Then, this chapter presents related background in using Ruby to implement neural network (Section 2.2). Details about how to efficiently process the CNN inference step are discussed in Section 2.3.

2.1 Deep Learning and Deep Neural Networks

Deep learning is a thriving subfield of *Machine Learning* (ML), which is a widely used technique in the field of *Artificial Intelligence* (AI). In contrast to the traditional knowledge-based approach, which is heavily relying on hard-coded knowledge such as logic-based rules, the machine learning approach to artificial intelligence is more flexible: the program constructed by a machine learning algorithm can be *trained* to enhance the performance based on a given dataset. There are many famous machine learning algorithms, such as *Support Vector Machine* (SVM) and *Gaussian Process*, and many successful applications built with these algorithms, such as spam detection and handwriting recognition.

However, the performance of a machine learning algorithm is restricted by the quality of *features*, which are extracted from the raw data. Although badly extracted features will misguide the training process and result in low performance, it is hard to avoid as good features can only be selected based on expert knowledge in the corresponding field. Deep learning provides an effective solution to the problems in machine learning, which will be introduced in the following sections.

2.1.1 Principles of Deep Learning

Deep learning origins from the idea of *representation learning*, which is a variation of machine learning that learns not only the mapping from extracted features to prediction, but also features themselves. The deep learning approach to representation learning is by expressing representations based on other simpler representations (Goodfellow et al., 2016): take a common image classification task as an example, simple representations that can be automatically learnt are colours, edges, and contours and the final representation for classification will consist of those simple representations. In the context of state-of-the-art deep learning models, simple representations mentioned above are usually known as representation *layers*, which can be concatenated together to form a feed-forward computation flow, such as *Multi-Layer Perceptrons* (MLP) and feed-forward *Neural Networks*.

The performance of a deep learning model depends on its *depth*. All the layers between the input and output of a deep learning model are often recognised as *hidden layers*, each of which contains features decomposed from the original data representation and will be reused in its following hidden layers. The deeper the deep learning model is, the more hidden layers that the model will contain. With more hidden layers, the deep learning model can construct more complex representation from those hidden layers, and its performance can be enhanced. In the next section, I will introduce the widely applied deep learning model - deep neural network, in which the layers are inspired by neurons and have several typical types.

2.1.2 Deep Neural Networks

A basic neural network contains two major computation units: one is a *weighted-sum* neuron that takes input from other neurons and outputs one value, the other one is a non-linear activation unit that integrates non-linearity to the network. These two units are generally inspired by the brain structure, especially the way that synapses work and connect to others. Neural networks can be innately grouped into layers, which are compatible with the concept of representation layers in deep learning, and further makes the idea of deep neural networks work.

In this report, I focus only on *feedforward* DNNs, which are composed of layers that are organised in sequences. There are five types of layers that are frequently used in recent DNN literature: *convolution*, *fully-connected*, *pooling*, *activation*, and *normalization*, which will be discussed as follows:

Convolution Layer

The computation in a convolution layer is based on convolution operations, which transform a multi-channel input image to another multi-channel output image. In the context of convolution layer, the input and output images of this layer are also known as *feature maps*, which denote that the output image of a convolution layer contains extracted features. There are two major properties of the convolution layer (LeCun et al., 1998), one is the *local receptive field*, which indicates that the features extracted from a convolution layer are based on localized information of the image; the other one is *shared weights*, which means that the convolution kernels (small coefficient matrices) have same values when computing all the pixels within a channel.

A convolution layer can be described by the parameters in Table 2.1. [1] Based on these parameters, the behaviour of a convolution layer can be described in Equation 2.1. $fmap_{in}$ and $fmap_{out}$ are input and output feature maps, and $weights$ is the coefficient 4-D matrix. This equation satisfies the two properties mentioned above, because for each pixel in the output feature map, it is only related to a receptive field of the input feature map (a $K \times K$ window), and the coefficient matrix will be shared.

$$fmap_{out}[f][h][w] = \sum_{c=0}^{C^{conv}} \sum_{i=-\lfloor \frac{K}{2} \rfloor}^{\lfloor \frac{K}{2} \rfloor} \sum_{j=-\lfloor \frac{K}{2} \rfloor}^{\lfloor \frac{K}{2} \rfloor} fmap_{in}[c][h+i][w+j] \times weights[f][c][i][j] \quad (2.1)$$

Dilated convolution is proposed by Yu and Koltun (2015) to aggregate contextual information for image semantic segmentation. The dilated convolution, which is referred as *atrous convolution* by

[1] Note that in this table the *stride* and *padding size*, which are two normally used parameters are ignored. These two parameters currently have no significant effect on the conclusion of this report.

Table 2.1: Parameters to describe a convolution layer

Name	Description
H^{conv}	The height of the input feature map
W^{conv}	The width of the input feature map
C^{conv}	The number of channels of the input feature map
F^{conv}	The number of channels of the output feature map
K	The size of the convolution kernel

Chen et al. (2014, 2016a). We will stick with dilated convolution in the following discussion. The feature of the dilated convolution is that, letting the *dilation factor* is l, the distance between two adjacent input activations collected by each sum-of-product step is $l-1$, rather than 0 which is used in normal convolution (1-dilated convolution). The motivation to introduce the dilation factor is that dilated convolutions support exponentially expanding receptive fields without losing resolution or exponentially increasing the size of parameters. This kind of convolution has been adopted in the *context module* (Yu and Koltun, 2015), which enhances the performance of semantic segmentation tasks. In Chen et al. (2016a), they discuss how the dilated convolution is equivalent to a downsampled convolution followed by an upsampling operation.

Convolution factorisation is an appealing idea that *factorises* a large convolution layer into several convolution layers with smaller kernels. Layers created after factorisation together perform a convolution computation that has same input and output dimensions as the original layer. This idea can reduce the parameters and computation resources required for a convolution layer. Jin et al. (2014) propose a factorisation method called *flattening*, which turns an original 3D kernel ($N_C \times K_H \times K_W$) into three 1D kernels ($N_C \times 1 \times 1$, $1 \times K_H \times 1$, and $1 \times 1 \times K_W$). This method can reduce parameters by a factor of $(N_C \times K_H \times K_W)/(N_C + K_H + K_W)$ for each output channel. Results show that both training and inference processes can be accelerated, and the accuracy can also be better on some datasets. However, the result should be further evaluated on large-scale datasets.

Convolution module is a recent developed concept that groups several **small** convolution layers, which have a relatively small amount of parameters, to perform the computation of a single **large** convolution layer, which has a relatively large amount of parameters. The major motivation of this approach is to reduce the total number of parameters required for a large convolution layer and increase the computational efficiency. Also, a CNN built on convolution modules is easier to be parameterised. Iandola et al. (2016) introduce the *Fire module*, which is comprised of two convolution layers: one has 1 kernels (*squeeze*) and the other one has both 1 and 3 kernels (*expand*). This architecture can be viewed as a normal CNN with 3 kernels partially replaced by 1×1 kernels to reduce number of parameters. Their results show that SqueezeNet can save at most 510x space comparing with the original architecture without losing accuracy. However, it will be better if they can provide deeper insight of SqueezeNet's performance, more than empirical numbers.

Depthwise Separable Convolution is a recent technique introduced by Chollet (2016) in the Xception network. Xception is derived from the idea behind the Inception network, which assumes that cross-channel correlations and spatial correlations of a convolution layer should be decoupled to increase efficiency. Xception pushes this idea to an extreme case with *depthwise separable convolution*, which performs *depthwise convolution*, i.e. a individual spatial convolution of each channel at first, and then *pointwise convolution* which is identical to 1×1 convolution. Depthwise convolution learns spatial correlations and pointwise convolution learns only cross-channel correlations. MobileNet (Howard et al., 2017) is another CNN architecture that is built upon *depthwise separable convolution* layers. Suppose there is a convolution layer with configuration $(H, W, C_{in}, C_{out}, K)$, it can

be replaced with a depthwise convolution (H, W, C_{in}, K) and a pointwise convolution (H, W, C_{out}), and the total number of operations is reduced by $1/C_{out} + 1/K^2$.

Fully-Connected (FC) Layer

An FC layer stands for a neural network layer that each of its output neurons is a weighted sum of *all* input neurons, which also indicates that the output neurons are *fully connected* to input neurons. If all the weights are stored in a coefficient matrix, and the weights used for computing each output neuron are grouped in each row, then the computation within the FC layer is a matrix-vector multiplication, where the vector contains all the values in input neurons. According to this behaviour, only two parameters (H^{fc} and W^{fc}) are required to describe an FC layer (Table 2.2).

Unlike the convolution layer, the FC layer has no local receptive fields and no shared weights, which further means that an FC layer cannot effectively learn local features in vision tasks and requires a large amount of space to store its weights. Thus, in recent DNN architectures, FC layers are often appended to a sequence of convolution layers and output classification probabilities.

Table 2.2: Parameters to describe a fully-connected layer

Name	Description
H^{fc}	The height of the weight matrix
W^{fc}	The width of the weight matrix

Pooling Layer

In order to make a convolution layer more robust, which means that features extracted from this convolution layer should be tolerant to small distortion and *translation-invariant*, a pooling layer is often applied on the output feature map. A pooling operation is based on a sliding window, which slides over the feature map and outputs the *maximum* or the *average* value of the region that the window covers. If the sliding stride is larger than 1, then this pooling layer can also reduce the dimensions of the output feature map.

In this report, I will fix the value of parameters in each pooling layer: the size of the sliding window will be 2×2, and the sliding stride will be 2. This configuration is widely used in many state-of-the-art DNN architectures.

Activation Layer

An activation layer adds *non-linearity* to a DNN. Convolution layers and fully-connected layers are linear combinations of the input image, and it is hard to fit non-linear target functions with only these layers. *Rectified Linear Unit* (ReLU) (Nair and Hinton, 2010) is a commonly used activation layer with a very simple functionality that gives an output of value 0 if the input value is less than 0.

Normalization Layer

A normalization layer is aiming at eliminating the distribution changes between the input and the output, which is one of the major problems while training a DNN. *Batch Normalization* is a popular normalization technique in recent years, which learns several hyper-parameters during training and

uses them in a very simple formula during inference. It is efficient in computation and effective in performance improvement.

Based on these DNN layers, several DNN *models* have been proposed and applied in real-world tasks. Most of these models are *Convolutional Neural Networks* (CNNs). CNN is a special type of feedforward DNN and is specifically designed for vision tasks. There are also DNNs of other types, such as *Recurrent Neural Network* (RNN) and *Long Short-Term Memory* (LSTM), which have quite different topologies and will not be covered in this report. **The rest of this report will focus only on CNN.** AlexNet (Krizhevsky et al., 2012) is the first DNN that applies to large-scale image classification problems and has achieved great success. VGG-16 (Simonyan and Zisserman, 2014) increases the number of convolution layers to 16 and proves that deeper and regular DNN can reach very high accuracy in image classification tasks. GoogLeNet (Szegedy et al., 2015) (Szegedy et al., 2016), also known as Inception, utilises *inception* modules that contain multiple branches of convolution layers, each of which has different kernel size and can extract different features. In this way, GoogLeNet has further explored the depth that a DNN could reach (up to 42 layers). ResNet (He et al., 2016) is the most recent DNN model that reaches up to 152 layers by using residual connections, which effectively resolve the vanishing gradient issue during training very deep DNNs.

There are two major discoveries from these DNN models: first, although these models have different topologies, most of the convolution layers are followed by a sequence of pooling, normalization, and activation layers, and fully-connected layers are appended at the end of the computation flow; second, all the DNN models are quite computation-intensive, they require million and even billion number of weights and *Multiply-ACcumulate* (MAC) operations per image. Table 2.3 summarises the resource usage of each DNN model (Sze et al., 2017). We will devise the hardware designs based on these characteristics in the following chapters.

Table 2.3: Summary of resource usage for each DNN model (Sze et al., 2017)

Resource Usage	AlexNet	VGG-16	ResNet-50	ResNet-152
# of weights	61M	138M	25.5M	60M
# of MACs per image	724M	15.5G	3.9G	11.3G

2.2 Ruby and Neural Network

This section reviews basic knowledge of Ruby and publications that use Ruby to describe neural networks.

2.2.1 Ruby Basics

Ruby is a unique hardware description language. It builds hardware designs based on *relations*, which describe building blocks based on their domain and range uith additional *composition* and *transformation* functions. Designs described in Ruby are often more concise and organised comparing with designs in other languages, such as Verilog and VHDL. However, we cannot directly synthesize designs from Ruby: we need to transpile designs into other languages first and then synthesize from those languages. This section presents several key concepts in Ruby and please review Jones and Sheeran (1990) for further details. Terminologies and symbols in this section are also referred from to Luk et al. (1994).

Relation

The fundamental element of Ruby is the *relation* that is often presented in the form $x\ R\ y$, in which R is the label for relation and $x\ y$ represent the *domain* and *range* of R respectively. For example, an adder, which is a relation that has two elements in the domain and their sum in the range, can be described as $\langle x, y\rangle\ add\ (x + y)$. Here $\langle x, y\rangle$ means a list of two elements. We also use $\langle x\rangle_n$ to represent a list of n elements.

There are several relations we use to operate on lists. π_1 (Equation 2.2a) and π_2 (Equation 2.2b) select the first and the second element of an input pair respectively. $swap$ (Equation 2.2c) swaps the positions of two inputs. Suppose ι is an *identity relation* that $x\ \iota\ x$, then fst (Equation 2.2d) and snd (Equation 2.2e) can be defined as abbreviations. At last, R^{-1} (Equation 2.2f) is the *converse* of relation R.

$$\langle x, y\rangle\ \pi_1\ x \tag{2.2a}$$
$$\langle x, y\rangle\ \pi_2\ y \tag{2.2b}$$
$$\langle x, y\rangle\ swap\ \langle y, x\rangle \tag{2.2c}$$
$$fst\ R = [R, \iota] \tag{2.2d}$$
$$snd\ R = [\iota, R] \tag{2.2e}$$
$$x\ R^{-1}\ y = y\ R\ x \tag{2.2f}$$

For more information on basic data structures and different shapes of relation blocks in Ruby, please refer to Jones and Sheeran (1990).

Composition

Relations can be *binary composited*, in sequential or in parallel. $x\ R; S\ y$ is a sequential composition of relations R and S, which infers that $\exists z.\ x\ R\ z \wedge z\ S\ y$. $\langle x_0, x_1\rangle[R, S]\ \langle y_0, y_1\rangle$ composites R and S in parallel, which infers that $x_0\ R\ y_0 \wedge x_1\ S\ y_1$. There are also two functions that *connects* two relations together: *beside* ($\leftrightarrow$) and *below* ($\updownarrow$), which are defined respectively in Equation 2.3a and Equation 2.3b.

$$\langle x, \langle y, z\rangle\rangle\ R \leftrightarrow S\ \langle\langle p, q\rangle, r\rangle \Rightarrow \exists s.\ \langle x, y\rangle\ R\ \langle p, s\rangle \wedge \langle s, z\rangle\ S\ \langle q, r\rangle \tag{2.3a}$$
$$\langle\langle x, y\rangle, z\rangle\ R \updownarrow S\ \langle p, \langle q, r\rangle\rangle \Rightarrow \exists s.\ \langle x, s\rangle\ R\ \langle p, q\rangle \wedge \langle y, z\rangle\ S\ \langle s, r\rangle \tag{2.3b}$$

There are also several other *high-order functions* that composite relations repeatedly: R^n (Equation 2.4a) places n number of relation R in sequential, map (Equation 2.4b) replicates a relation into an array without inner connections, row (Equation 2.4c) and col (Equation 2.4d) composite relations with inner connections in horizontal ($\leftrightarrow$) and vertical ($\updownarrow$) respectively.

$$x\ R^n\ y \Rightarrow \exists s_0 = x \wedge s_n = y.\ \forall i \in \{0, \ldots, n-1\}.\ s_i\ R\ s_{i+1} \tag{2.4a}$$
$$\langle x\rangle_n\ map_n R\ \langle y\rangle_n \Rightarrow \forall i \in \{1, \ldots, n\}.\ x_i\ R\ y_i \tag{2.4b}$$
$$\langle a, \langle x\rangle_n\rangle\ row_n R\ \langle\langle y\rangle_n, b\rangle \Rightarrow \exists s_0 = a \wedge s_n = b.\ \forall i \in \{1, \ldots, n\}.\ \langle s_i, x_i\rangle\ R\ \langle y_i, s_{i+1}\rangle \tag{2.4c}$$
$$\langle\langle x\rangle_n, a\rangle\ col_n R\ \langle b, \langle y\rangle_n\rangle \Rightarrow \exists s_0 = a \wedge s_n = b.\ \forall i \in \{1, \ldots, n\}.\ \langle x_i, s_i\rangle\ R\ \langle s_{i+1}, y_i\rangle \tag{2.4d}$$

Stream and Serialisation

Symbols in the domain and range of a relation R actually represent *streams*. A stream in Ruby is a collection of values at an input wire of all *ticks*, for example, x_t represents the specific element of stream x at tick t.

Latches in a circuit, which temporarily store values and are triggered by clock, should be explicitly specified by *delays* ($\mathcal{D}$) in Ruby (Equation 2.5a). Although it is not implementable and can only be simulated for most cases, *anti-delay* (Equation 2.5b) is adopted to "predict" the value of the stream in the next tick.

$$\forall t > 0.\, x_t \; \mathcal{D}\; x_{t-1} \tag{2.5a}$$

$$\forall t > 0.\, x_t \; \mathcal{D}^{-1}\; x_{t+1} \tag{2.5b}$$

$$x \; (loop \; R) \; y \Rightarrow \exists s.\, \langle x, s \rangle \; R \; \langle s, y \rangle \tag{2.5c}$$

$$cnt = loop \; (add; \mathcal{D}\; 0; fork) \tag{2.5d}$$

We can also implement *state machine* in Ruby by delays and $loop$, which redirects the range of a relation into its domain. See Equation 2.5c for what the $loop$ relation implies. Based on $loop$, it is straightforward to implement a basic state machine, such as *counter* (cnt). Equation 2.5d implements the counter by simply feed the output of an adder back to its input. Note that the output of the adder should be delayed to avoid *combinational loop*, the delay is initialised with constant zero, and the output of the delay will be forked (or duplicated) to provide an output to the outside, which can be viewed as the current value of the counter state, and an input back to the adder. This counter also has an input that is the step size of the counter.

Please find further details related to serialisation high-order functions in Jones and Sheeran (1990), such as cmx and $bundle$.

2.2.2 Neural Network in Ruby

Describe neural network in Ruby has been studied in the 1990s. Luk et al. (1994) present a thorough study on designing and implementing multi-layer perceptrons (MLP) into FPGA hardware by Ruby. In that paper, several designs with different architectures and properties are provided and compared. All these designs with complex structure are described within a few lines of Ruby code, and optimising them with pipelining can be done with straightforward transformation, which is impressive and shows the powerful ability of expression.

The difference between our report and that paper is that, in this report, we are considering deep convolutional neural networks. We provide more building blocks than that paper. Also, we use a different architecture to implement the MLP, which is equivalent to fully-connected layer in the current context, by using an array of multipliers followed by an adder tree.

2.3 Efficient CNN Processing

Processing a CNN has high computation complexity, both in time and space, which requires many hardware resources to finish a CNN computation within a limited time. Although this characteristic may not cause a significant problem for data centers that have high-end servers with GPUs installed,

how to efficiently process a CNN becomes an important challenge, because there is an increasing demand for CNNs on embedded hardware platforms that have limited hardware resources and real-time processing rate requirements. This section reviews the state-of-the-art efficient CNN processing approaches, which can be divided into four categories:

1. *Algorithmic Optimisation* (Section 2.3.1): Optimisation techniques in this category focus on reducing time or space complexity of CNN computations. These techniques can be effective as a reduction in complexity results in a *scalable* runtime and memory footprint reduction. Although an algorithmic optimisation is normally not restricted to a hardware architecture, the performance difference between a general implementation and a tuned implementation on a special hardware architecture. Moreover, some algorithmic optimisation techniques are intractable when there are limited hardware resources.

2. *Hardware Architecture* (Section 2.3.2): When the CNN processing algorithm is fixed, there are hardware architectures dedicated to accelerating the algorithm processing by efficiently utilising the computation resources. FPGA-based architectures will be mainly discussed in this section. Besides the processing speed, the design of hardware architecture for efficient CNN processing should also take the energy efficiency into account.

3. *Network Model* (Section 2.3.3): Different CNN topologies have different levels of resource efficiency, which can be evaluated by the number of operations performed per inference pass, or the number of parameters required per input image pixel. These metrics can be improved by applying low-precision data types, removing useless connections, or even changing the hyper-parameters of the network design. Note that a trade-off between the model efficiency and accuracy is a key problem to be discussed.

4. *Framework* (Section 2.3.4): A framework for efficient CNN processing aims at directly transform an original CNN model to an optimised system, which contains a generated hardware architecture and a refined network model. In this case, not only the processing performance, but also the development efficiency are important.

This section concentrates on the *inference* step of CNN and typical training algorithms like backpropagation will not be considered. However, it is worth to note that acceleration in the inference step will normally also boosts the training step, because they share many time-consuming procedures.

2.3.1 Algorithmic Optimisation

The core and most time-consuming computation in CNN is 2D (2-dimensional) convolution. 2D Convolution can be implemented by applying either a sliding-window algorithm or a single large matrix multiplication. The first approach is a direct implementation of the 2D convolution, which "slides" a window of filter values through an input image. Each step sums a *Hadamard product* between the filter and the image to a scalar. The second approach transforms the input image to a matrix at first, then multiply it by the filter matrix in one step. Convolution can be either optimised based on either method.

Sliding-Window Optimisation

Two approaches to optimise the sliding-window based computation are as follows:

1. *Fast Fourier Transform* (FFT): FFT is a classic signal processing algorithm that converts the representation of a signal between its original domain and its frequency domain in *linearithmic* complexity $\mathcal{O}(n \log n)$. Mathieu et al. (2013) prove that, the result of a 2D convolution is equivalent to the Hadamard product of two FFT-transformed inputs followed by an inverse FFT. Besides, this optimisation can reduce the time complexity from $\mathcal{O}(n^2k^2)$ to $\mathcal{O}(n^2 \log n) + \mathcal{O}(n^2)$ when computing a 2D convolution between a $n \times n$ feature map and a $k \times k$ kernel.

   However, the performance gain from the FFT optimisation depends on the configuration of the 2D convolution. Chetlur et al. (2014) argue that if the filter size is small or the stride size is larger than 1, then FFT may require a large amount of temporary memory and need to perform the less efficient sparse computation. Vasilache et al. (2014) also present evaluate results that small kernel size limits the performance gain from FFT, even their implementation of FFT-based convolution[2] can outperform other approaches in famous CNNs at that time. Note that state-of-the-art CNNs are constructed with small kernels (3×3) and deep structures, such as VGG and ResNet, which may not be able to take advantage of FFT-based convolution.

2. *Winograd Minimal Filtering Algorithm* (Winograd, 1980, p. 43): Letting $F(m, r)$ be a r-tap *Finite Inpulse Response* (FIR) filter with m-output, the Winograd minimal filtering algorithm states that the number of multiplications required in the standard algorithm of FIR equals to the number of the filter inputs, which is $m + r - 1$. And in a special case $F(2, 3)$, the number of multiplications can be reduced to 4 from 6 in this algorithm. Lavin and Gray (2016) propose to use this algorithm to reduce the complexity of convolution with 3×3 kernels by tiling input images to 4×4 blocks. These blocks can be efficiently optimised by the Winograd algorithm as the number of output equals to 2 in each dimension, which is a two-dimensional case of $F(2, 3)$.

   Although this optimisation method limits the kernel size of the target convolution layer to 3, it could not limit its usage as there is a trend to construct CNNs on 3×3 kernels.

Matrix Multiplication Optimisation

Convolution can be implemented as a single matrix multiplication, preceded by a conversion of the input feature map. The conversion is known as *im2col*, which stretches each window of the input image to a column in the result matrix. Suppose the input feature map (3D tensor) has shape $C \times H \times W$, then the converted matrix should have shape $(K^2C) \times (H - K + 1)(W - K + 1)$, where K is the edge length of the filter kernel. The time complexity of this algorithm is $\mathcal{O}(FK^2CHW)$, according to the standard algorithm of matrix-matrix multiplication, which is equivalent to the direct convolution algorithm. However, the memory usage is increased by a factor of $\mathcal{O}(K^2)$ due to the conversion of the input feature map. Thus, a plain implementation of this algorithm may not get a performance enhancement.

Fortunately, matrix-matrix multiplication is a well-studied field, both in theory and practice. In theory, the *Strassen* algorithm and the *Coppersmith-Winograd* algorithm (Coppersmith and Winograd, 1990) can reduce the time complexity of matrix-matrix multiplication from $O(N^3)$ to $O(N^{2.8074})$ and $O(N^{2.37})$ respectively. In practice, many *Basic Linear Algebra Subprograms* (BLAS) APIs, which are well-optimised for CPU (e.g. Intel MKL[3] and OpenBLAS[4]) and GPU (e.g. cuBLAS[5]), can boost

[2] fbcunn: `https://github.com/facebook/fbcunn`
[3] `https://software.intel.com/en-us/mkl`
[4] `http://www.openblas.net/`
[5] `https://developer.nvidia.com/cublas`

the performance of the single large matrix-matrix multiplication. Thus, this matrix multiplication based approach is still a promising alternative for algorithmic optimisation of convolution.

2.3.2 Hardware Architecture

This section reviews the CNN optimisation techniques at the hardware architecture level. Mainstream hardware platforms for computing CNN includes CPU, GPU, FPGA, and ASIC. We mainly focus on FPGA in this section, and discuss general architectures implemented on ASIC. Although Sze et al. (2017) comprehensively present and compare different hardware architectures with an explicit consideration of energy consumption in different dataflows, we decide to understand different architectural optimisation choices from a *bottom-up* perspective, which intends to build recent architectures from fundamental elements. We also consider all metrics (e.g. performance, power consumption, etc.) equally. Our approach is intuitive and straightforward, because architectures built on FPGA platforms follow the *spatial architecture* paradigm, which constructs hardware designs by *dataflows* and *Processing Engines* (PEs). In this way, these architectures share many common components and one can be viewed as an evolved version of another. Finally, we derive a *tree* of architectures, which uses edges to represent optimisation objectives.

The Root Architecture

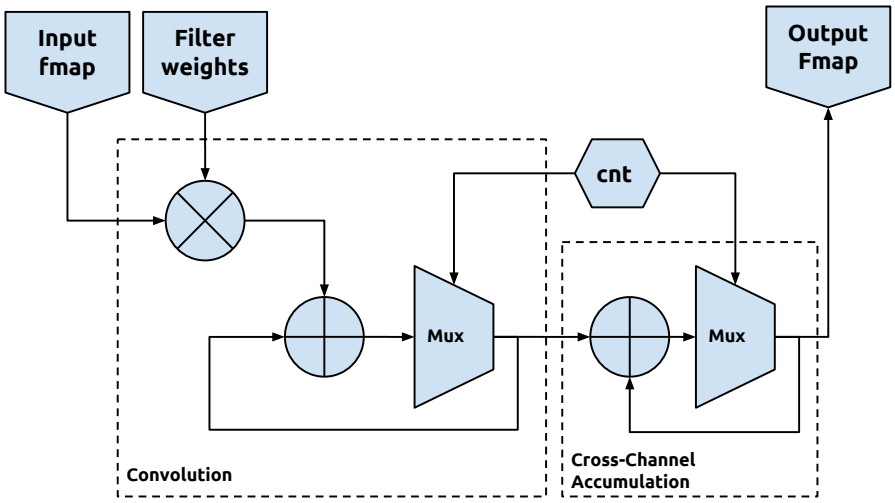

Figure 2.1: The root architecture.

Figure 2.1 illustrates the *root* architecture of CNN accelerator. This architecture uses minimal hardware resources, including two adders, one multiplier, control logics, and three streams. There are two PEs in this root architecture, one is *Convolution* which produces an accumulation result of element-wise product between a filter kernel and a window in the input feature map, the other one is *Cross-Channel Accumulation* which accumulates results from the convolution PE that contribute to one filter of one output activation. Input feature maps and filter weights are loaded from input streams, and the partial output feature maps are sent to the output stream. The data feed into the two input streams should be well-organised to give correct output. Windows of an input feature map should be organised in row-major, and an input map should be repeatedly fed into the stream. So for the filter weights.

Although this architecture uses minimal resources, it is obviously not the best choice. The maximal performance it can reach is considerably low, because it can only perform one multiplication in each clock cycle and the maximal clock frequency is limited on FPGA. Also, because input streams are connected to an off-chip memory, which has access latency and causes much energy consumption while reading from these streams, repeatedly reading identical data from these streams is not efficient.

Even though, because many existing designs can be viewed as derivations of this root architecture, which will be discussed in detail in following sections, this architecture is still worth being presented here.

Extension Points in the Root Architecture

The root architecture can be extended at different points within. Each point is an individual optimisation technique, and these points can be categorised into groups. These categories are listed as follows.

1. *Parallel PEs (Para)*: Placing multiple PEs of the same kind is a frequently used optimisation technique in spatial architectures. There are two levels of parallelisation for PEs: *fine-grained* and *coarse-grained* parallelisation. The fine-grained parallelisation places multiple ALUs within a PE, and the coarse-grained parallelisation places multiple PEs directly. Most of the time, these two levels are used together. For example, there can be multiple multipliers followed by an adder tree in the convolution PE to produce one convolution result in one cycle, and these inner-parallelised PEs will also be instantiated multiple times. Besides, the organisation of these parallel blocks is not fixed. It is possible to use a plain 1D array or a 2D matrix, such as a *systolic array*, to organise multiple PEs. Different organisations have different effect on both resource usage and latency. In the following discussion, we prefix extension points in this category with *Para-*.

2. *Registers in PEs (Reg)*: Data fetched from input streams can be reused for consecutive cycles. For example, the filter kernel remains the same for all input activations in the same channel while computing for the same output filter. It will be a waste of energy and bandwidth if it loads same values repeatedly. Temporarily storing read values in registers can effectively increase data reuse. Although it is an intuitive approach, when combined with parallelisation there will be more to be considered. For example, registers can be either a unified register file or several separated register files within each PE. Also, which stream should be temporarily stored in registers and how many registers will be stored are two open questions. Extension points in this category will be prefixed with *Reg-*.

3. *On-Chip Cache (Cache)*: Except using registers, using on-chip memory to store large blocks of data for further reuse is also an essential optimisation technique. Multiple channels of an input feature map can be completely cached in an on-chip memory block. Besides, it is also possible to cache multiple filters of the output feature map. These two choices are different in computation sequence, which are *filter-major* and *channel-major* respectively. These names are derived by choosing the major index of the multiple loops for computing convolution in the direct algorithm.

4. *Orthogonal Optimisation (Orth)*: There are several optimisation techniques that can be considered orthogonal to previous discussions, e.g. changing bit-width of the data type and using extra special PEs to compute sparse matrices, which will not significantly deviate from our

architecture. We consider these orthogonal optimisation at the bottom of our tree of architectures.

Figure 2.2 summarises the graph of architectures grown from the root. Each path from the root to the top is an existing design from previous publications.

Design Patterns

Most of the hardware architectures in recent publications follow several patterns that can be categorized. Each pattern is a specific combination of extension points. These patterns are summarised as follows, including their advantages and disadvantages.

1. *Systolic Array* (sys): Systolic array is a typical hardware architecture, in which multiple PEs are interconnected and communicate in a systolic style. Unlike other parallel architectures that connect all PEs directly to streams, a systolic array only connects *boundary* PEs to streams. Other PEs in a systolic array communicate data with only PEs. The major advantages of using systolic array are reducing global data communication and enhancing data reuse. Kung (1982) presents this advantage by discussing different designs for convolution based on systolic array. Wei et al. (2017) argues that a systolic array based design is easier for an FPGA to place and route due to its regular structure. This property is great for increasing the clock frequency of a massive parallel design. However, to maximise these advantages, a systolic array must be well organised. Streams connected to the systolic array should also be carefully arranged and controlled. Thus, it is more feasible to devise an automatic tool that can generate designs based on systolic array, rather than hand-tuning them. Wei et al. (2017) automate the process of building a systolic array based architecture for a specific CNN topology. The systolic array based architecture for processing CNN can either compute the sum-of-product of the input feature map and the filter within one PE (Wei et al., 2017), or perform a dot-product in the matrix-matrix multiplication (Gupta et al., 2015). These two approaches arrange the streams differently.

Categorise Architectures from Previous Publications

We select several architectures from previous publications to show that they fit our graph.

1. Zhang et al. (2015) propose an architecture that makes use of multipliers and adder trees in their PEs, and PEs are organised in a 1D parallel array. They use a large register file to temporarily store weights and input feature maps, which is considered as *No Local Reuse* regarding the register level of optimisation. At last, they use the *roofline model* (Williams et al., 2009) to search for the best design parameters for their architecture, which is an orthogonal optimisation technique. Qiu et al. (2016) devise a similar architecture. However, their architecture chooses to use a *line buffer* to enhance the cache performance, and they apply data quantization to improve the resource efficiency. The architecture of Origami (Cavigelli and Benini, 2016) also adopts a 1D array of PEs, while each PE has weights stationary in registers.

2. Du et al. (2015) build their architecture based on a 2D matrix of PEs, which contain ALUs and connections similar to the root architecture. Each PE can only perform a computation of one pair of elements in one cycle. Also, each PE stores an output activation before it is ready, which

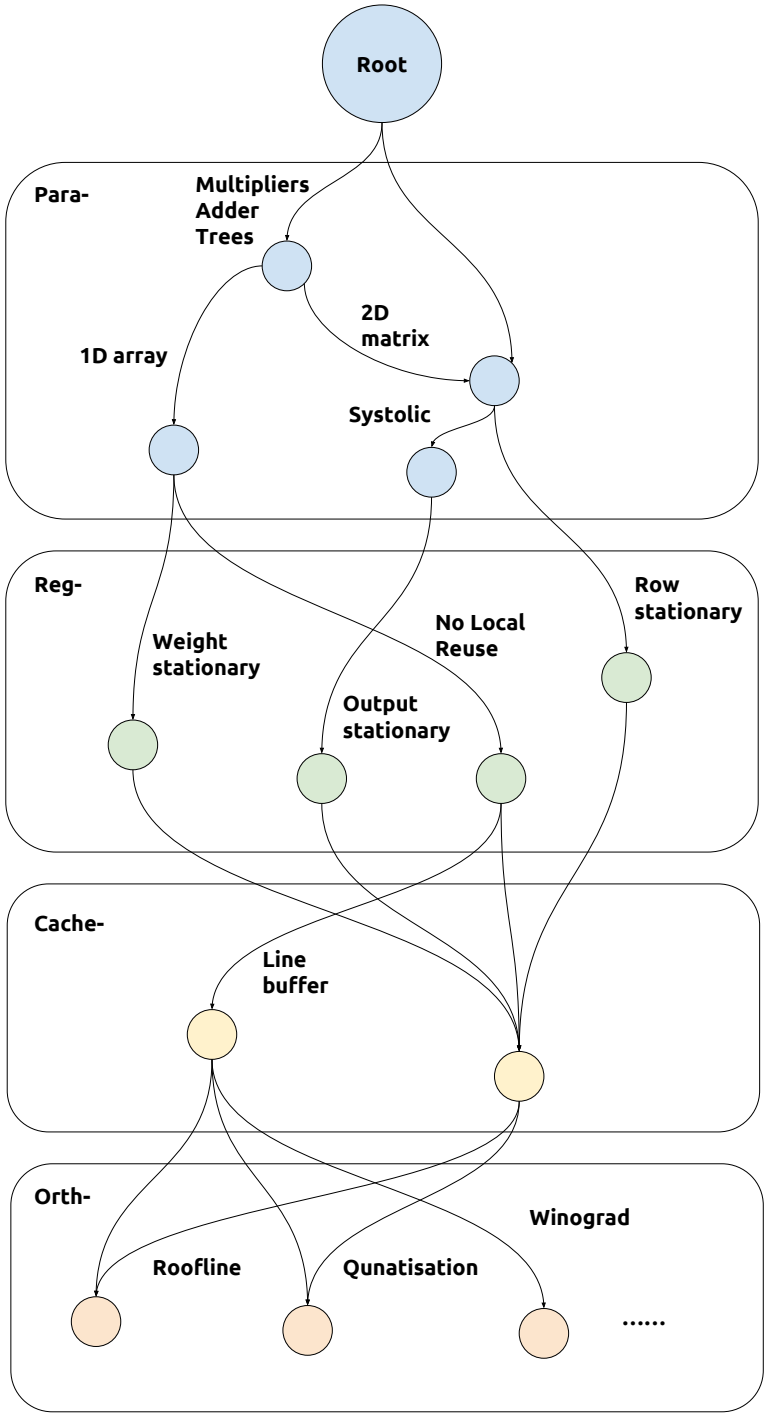

Figure 2.2: The graph of design decisions

requires registers to be placed in an output stationary style. Gupta et al. (2015) also organise PEs in a 2D matrix, while their inner connections are systolic. Chen et al. (2016b) further enhance the performance by separating register files into PEs based on the *row stationary* style. Zhao et al. (2016) is an example shows the potential of using FPGA to both train and inference CNN model, and they explicitly explore the possibility of using runtime reconfiguration on the FPGA platforms to load bitstream files of multiple layers at runtime. Alternatively, Li and Pedram (2017) studies a reconfigurable architecture that is deliberately designed for training, which explicitly considers the training algorithms that their communication patterns while running in a multi-core reconfigurable platform.

2.3.3 Network Model

A network model is a trained neural network. A model contains not only its architecture but also coefficients, which are trained from training datasets. Optimisation techniques at the network model level tune both the architecture and coefficients. We discuss the specific techniques applied to these two aspects in the following sections.

Architecture Optimisation

Network architecture has two types: *macro-architecture* and *micro-architecture*, which refer to the high-level organisation of CNN and the low-level organisation of CNN modules and layers respectively.

1. *Macro-architecture Optimisation*: New macro-architectures proposed in recent publications are motivated by enhancing accuracy rather than efficient processing of CNN. Typically, Highway Networks (Srivastava et al., 2015a,b) and ResNet (He et al., 2016) respectively use *bypass* connections and *residual* connections to effectively train very deep CNNs.

2. *Micro-architecture Optimisation*: Although many discussions on micro-architecture optimisation are also related to CNN accuracy, there is a trend to propose efficient CNN micro-architectures, in terms of a number of parameters used in the network. Inception (Szegedy et al., 2015) first introduces *modules* in CNN, which are groups of CNN layers, and they are functionally equivalent to convolution layers but with fewer parameters. Modules in Inception are called *inception* modules, which contains convolution layers with different kernel sizes and their results are concatenated together. SqueezeNet (Iandola et al., 2016) follows their idea and proposes *fire* modules, which "squeeze" input feature maps by a 1×1 convolution layer and "expand" its intermediate result by several 1×1 and 3×3 convolution layers. MobileNet (Howard et al., 2017) goes further by factorising convolution layers by *depthwise* convolutions, which contain a single filter, and *pointwise* convolutions, which are 1×1 convolutions.

Coefficients Optimisation

The purpose of coefficients optimisation is to reduce the memory footprint and increase the performance while maintaining the accuracy. Thus, optimisation techniques related to coefficients are deeply connected to data *quantisation* methods and *sparse* algorithms.

Data quantisation is the process of mapping values from a continuous set to a discrete set, and in the context of deep neural network, it converts coefficients from real numbers, which should be repre-

sented by floating-point, to integers, which can be represented by fixed-point or even raw bits. In this way, data quantisation can reduce the memory required to store coefficients and the hardware resources used by arithmetic. The latency will also be lowered.

Sparsity is an observation that the coefficient matrices in CNN contain many zero entries, especially after quantisation. It is also a natural outcome of *network pruning*, which removes useless connections within layers. This property can be utilised to enhance the performance of CNN processing because there are well-studied sparse algorithms that run faster than normal algorithms on sparse matrices. Several publications have studied these optimisation techniques:

1. *Quantisation*: Training and inferencing neural networks with limited precision have been studied since 1991 (Holt and Baker, 1991; Holt and Hwang, 1993). There is a trend to apply data quantisation on deep neural networks since when deep learning became popular. Courbariaux et al. (2014) evaluate the accuracy of fixed-point deep neural networks, which is also trained in low-precision. Gupta et al. (2015) propose using the stochastic rounding for training low-precision CNNs, which is implemented on an FPGA device.

   Computing with neural networks represented in raw bits is also an "historic" idea (Shoemaker et al., 1991). Courbariaux et al. (2016) propose to use binarised coefficients in CNNs. They convert typical CNN layers, such as convolution and batch normalisation, to their binarised version. Rastegari et al. (2016) extend the binarised network from using binary weights to using both binary weights and inputs.

2. *Network pruning and Sparsity*: *Optimal Brain Damage* (or *Surgeon*) is proposed by LeCun et al. (1989); Hassibi et al. (1993), which prunes network connections by the information from second order derivatives. Han et al. (2015) devise a framework called *Deep Compression* that can prune deep CNNs, and use sparse format to compute pruned networks.

2.3.4 Framework

Frameworks for efficient CNN processing work in similar steps. They first interpret CNN representations through configuration files, and then generates target hardware designs. Ma et al. (2016) generate scalable CNN designs from given CNN models to their parameterised RTL hardware designs. They tune the parameters while generating the hardware. DiCecco et al. (2016) is an alternative approach that provides fixed hardware designs that can be integrated with software deep learning frameworks. The second approach is easier to use but cannot tailor their hardware designs based on the network architecture.

2.4 Summary

In this chapter, we review the essential background of deep learning and CNN, and we concentrate on how to efficiently process CNN in four aspects: algorithm, hardware architecture, network model, and framework. It is worth to note that, although there are many tips and tricks about optimising CNN processing performance in recent publications, how to *systematically* integrate these techniques is still an open problem.

Chapter 3

RubyConv: Ruby-based CNN Hardware Library

This chapter describes *RubyConv*, a hardware library to build CNN designs on FPGA platforms, which is written in *Ruby*, a high-level hardware language based on *relations* and *functions*. The benefits of using Ruby to describe CNN designs are listed as follows.

1. From the background chapter (Section 2.2) we realise that comparing with other hardware description languages Ruby can describe complex designs more *concise* by compositions and transformations based on relations. Therefore, when a CNN design is constructed by RubyConv, its description also becomes more concise.

2. Designs in Ruby can be well verified because of the *symbolic simulation* feature provided by Ruby tools. This feature helps us discover the direct relation between the domain and range of a complex design in symbols, which further makes the verification process simple and sound. We provide symbolic simulation of RubyConv in Section 3.2.1.

3. Ruby is *platform agnostic*, we can use Ruby to write the core design *once* and transpile it into other hardware description languages on different platforms. Thus, CNN designs built by RubyConv can be ported to different platforms and reduce lots of efforts in developing and maintaining several codebases for various FPGA platforms.

This chapter has the following sections. Section 3.1 first presents essential relations in typical CNN described in Ruby, which is separated into two parts: *basic relations* (Section 3.1.1) and *layer relations* (Section 3.1.2). Section 3.2 shows the evaluation of RubyConv, which is about building the LeNet-5 topology (LeCun et al., 2015).

The major contribution of this chapter is that, as far as we know, this is the **first** work that uses Ruby to describe a complete CNN hardware design. Our designs are all **parameterised** and by optimising their values we can achieve good performance. This chapter also provides a case study evaluation of RubyConv, which can also guide further usage of this library. Besides, RubyConv also contributes to other chapters in this report: Chapter 4 presents an OpenSPL-based CNN hardware library, of which the core architecture refers to RubyConv.

3.1 CNN Relations

This section introduces relations in RubyConv, which are often known as building blocks in other languages, for constructing CNN designs from bottom to top. At the bottom level, we present *basic* relations and functions (Section 3.1.1), which are some customised high-order composition functions and core arithmetic units. At the top level, meanwhile, we devise *layer* relations (Section 3.1.2) that are mapped from typical CNN layers, which are built upon basic relation blocks. For Ruby basics, we recommend to read Section 2.2.1, and read Jones and Sheeran (1990) for the complete documentation.

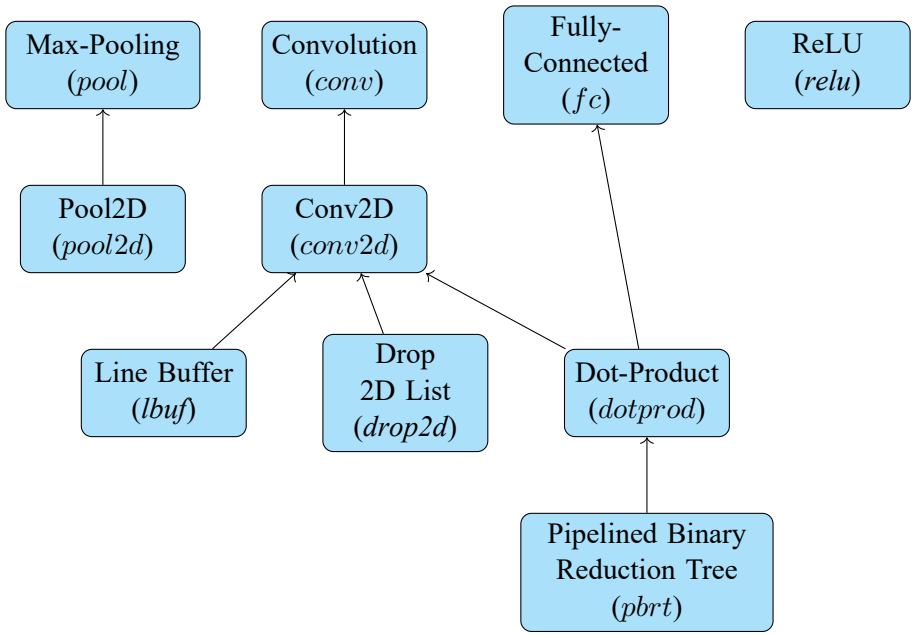

Figure 3.1: Dependencies among all relations and functions designed in this report.

3.1.1 Basic Relations and Functions

Ruby relations and functions belong to this category are introduced in a sequence of dependency that those discussed early are highly depended by others.

Pipelined Binary Reduction Tree

Pipelined Binary Reduction Tree ($pbrt$) is one of the most important and fundamental high-order composition function in RubyConv. It reduces an arbitrary length list of values into one by repeatedly applying a same *binary* relation. Two inputs of each relation instance is wrapped with delays to become pipelined. We restrict relations acceptable by $pbrt$ should be binary relations, which makes it simple and sufficient to use in the current context.

Equation 3.1 recursively defines $pbrt$: first two lines define the base cases and the rest define the *induction rules*. Note that every element in the domain, except the $n = 1$ case, is followed by a delay, and we ignore anti-delay relations in this equation.

$$\langle x\rangle_n \; pbrt_n R \; y \Rightarrow \begin{cases} x = y & \text{for } n = 1 \\ \langle x_1, x_2\rangle \; [\mathcal{D}, \mathcal{D}]; R \; y & \text{for } n = 2 \\ \forall i \in \{1, \ldots, m\}. \langle x_{2i}, x_{2i+1}\rangle \; [\mathcal{D}, \mathcal{D}]; R \; z_i \wedge \\ \qquad \langle z\rangle_m \; pbrt_m R \; y & \text{for } n = 2m \\ \forall i \in \{1, \ldots, m\}. \langle x_{2i}, x_{2i+1}\rangle \; [\mathcal{D}, \mathcal{D}]; R \; z_i \wedge \\ \qquad (x_n \; \mathcal{D} \; z_{m+1}) \wedge \\ \qquad \langle z\rangle_{m+1} \; pbrt_{m+1} R \; y & \text{for } n = 2m + 1 \end{cases} \tag{3.1}$$

Here are some properties of this function. The depth of the tree is $\lceil \log_2 n \rceil$, obviously. And the latency of $pbrt_n$, which is also known as *minimum cycle time*, is also $\lceil \log_2 n \rceil$ if there is no more delay in R, because the input of $pbrt$ can only go down one layer in the tree in each cycle due to delays in each layer. The number of R relations in the $pbrt_n R$ design is $n - 1$ which can be calculated by induction or the definition of reduction, and the *maximum* number of latches (delay relations) is $2(n - 1)$.

Dot-Product

The dot-product relation ($dotprod$) is widely applied in building CNN designs in RubyConv, such as constructing convolution layer and fully-connected layer. Dot-product ($\cdot$) is a well-known linear algebra operation, which produces the sum of element-wise products of two vectors of the same length. See Equation 3.2 for the formal definition.

$$\boldsymbol{x} \cdot \boldsymbol{y} \equiv \sum_{i=1}^{N} \boldsymbol{x}_i \times \boldsymbol{y}_i \tag{3.2}$$

The general form of the dot-product relation is $dotprod_{n,v}$, in which n is the length of the input vector and v is the width of a reduction tree. The reduction tree in $dotprod_{n,v}$ is implemented by $pbrt_v \; add$, that takes v number of elements in each cycle and produces the final result in $\lceil \log_2 v \rceil$ clock cycles, suppose there is no delay in add. The reason that we need two parameters to describe $dotprod$ in RubyConv is that the balance of resource usage and latency should be considered: when v is large, more resources will be used; otherwise, latency to produce a valid result will be longer.

Equation 3.3a illustrates the parameterised dot-product relation in RubyConv. The domain of $dotprod$ has two lists of length v and the range contains the final result (z) and a boolean (s) that identifies whether the result is valid. z is produced by R_v defined in Equation 3.3b, which first computes v multiplications in parallel and the reduces results by $pbrt$. Because the actual length of both input vectors is n, thus the result from $pbrt$ should be further reduced by an accumulator implemented by $loop$. $C_{n,v}$ is the other inner relation embedded in $dotprod$ to decide the value of the valid signal. It has a counter with wrap point at value M, which is the length of the latency from receiving the first chunk of input to producing the valid final result. The value of the valid signal is produced by a comparison between the counter value and $M - 1$. Note that the counter is implemented by a modulo accumulator, see $modcnt_m$ in Equation 3.3e.

$$\langle\langle x\rangle_v, \langle y\rangle_v\rangle \; dotprod_{n,v} \; \langle z, s\rangle \Leftrightarrow \langle\langle x\rangle_v, \langle y\rangle_v\rangle \; \pi_1^{-1}; \; [R_v, C_{n,v}] \; \langle z, s\rangle \tag{3.3a}$$

$$R_v = zip_v; \quad map_v\ mult; \quad \boldsymbol{pbrt}_v\ add; \quad \mathcal{D}\ 0; \quad loop\ (add; \mathcal{D}\ 0; fork) \tag{3.3b}$$

$$C_{n,v} = 1;\ modcnt_M;\ \pi_1^{-1};\ snd\ (M-1);\ eq \tag{3.3c}$$

$$M = \lceil \log_2 v \rceil + n/v \tag{3.3d}$$

$$modcnt_m = loop\ (modadd_m; \mathcal{D}\ 0; fork) \tag{3.3e}$$

$$modadd_m = add;\ \pi_1^{-1};\ snd\ m; mod \tag{3.3f}$$

Regarding properties of this $dotprod$ relation, the number of multipliers is v and the number of total adders is $v+1$, and the latency is M as mentioned before. The first term of M definition in Equation 3.3d is equivalent to the latency in $pbrt_v$, which is also the *initial interval* of the pipeline in $dotprod$. After the pipeline is fulfilled, $dotprod$ still needs n/v cycles, which is the second term in Equation 3.3d, to reduce to the final result.

Dot-Product with Bundle

There is an alternative way to implement dot-product in Ruby relation that reduces the valid signal s by using $bundle_n$, a relation that bundles n serial elements in the domain into a list of n elements. The motivation for this approach is that, although the valid signal is a common practice in handshaking between modules in hardware design, to the best of our knowledge, it is quite hard to implement the valid signal based handshaking scheme in Ruby. Equation 3.4 presents the $dotprod$ definition in this approach. Only the last element in the range of $bundle_M$ is the final output, and we use apr and π_2 to drop elements except the last one.

$$dotprod_{n,v} = R_v;\ bundle_M;\ apr_{M-1}^{-1};\ \pi_2 \tag{3.4}$$

In the following discussion, we assume that $dotprod_{n,v}$ refers to the implementation in Equation 3.4.

Summary

This section presents several important basic relations and functions, including $pbrt$ for pipelined binary reduction, and parameterised $dotprod$ to compute txhe dot-product between two vectors. Several other relations are also introduced in this section, such as $modcnt$. Table 3.1 summaries properties of each relation or function. Note that the number of units refers to the number of replicated relations of all kinds, it could be R in $pbrt$ or $mult$ and add in $dotprod$.

Name	Latency	Maximum Number of Latches	Number of Units
$pbrt_n$	$\lceil \log_2 n \rceil$	$2(n-1)$	$n-1$
$dotprod_{n,v}$	$\lceil \log_2 v \rceil + n/v$	$2v$	$2v$

Table 3.1: Summary of properties of each relation or function.

3.1.2 Layer Relations

This section introduces relations in RubyConv that can construct typical layers in CNN. For now, RubyConv supports four types of layer relations: $conv$, fc, $pool$, and $relu$. These relations are parameterised to support optimisation. We summarise properties of these relations at the end of this section.

Convolution Relation

The convolution layer is the most important layer in CNN. It is implemented as the $conv$ relation in RubyConv. To build $conv$, we first present the core computation relation of the convolution layer, $conv2d$, and then implement $conv$ based upon it.

The definition of $conv2d$ is listed in Equation 3.5a. $conv2d$ has one parameter k, which is either height or width of convolution filter kernel. The domain of $conv2d$ is a list of two 2D lists: a $k \times k$ window of the input feature map ($\langle\langle x\rangle_k\rangle_k$), and a $k \times k$ coefficient kernel ($\langle\langle w\rangle_k\rangle_k$), while the range is only a wire of the convolution output result. The implementation of $conv2d$ is based on $group$ and $dotprod$, as Equation 3.5b explains. $group_{k,k}^{-1}$ creates a flattened list from a $k \times k$ 2D list, and the first term of $conv2d$ implementation prepares the domain that is compatible with $dotprod$. $dotprod$ used in $conv2d$ is the fully parallelised version that $n = v = k^2$ and there is no accumulator based reduction.

Even with $conv2d$, it is still challenging to implement the convolution layer in RubyConv. First, the input feature map for convolution layer is not a stream of $k \times k$ 2D lists, it is actually a stream of single element generated by reading the input feature map in row major. Thus, we need to convert between these two structures with a relation. Line buffer ($lbuf_{k,w}$) implements this relation, here k is the kernel height and w is the width of the input feature map. In its definition (Equation 3.5c) , each element in the output $\langle\langle y\rangle_k\rangle_k$ is corresponding to a delayed x.

Next, both the domain and range of $conv$ should contain multiple channels to enable parallelisation and enhance the performance. We consider channel-wise parallelisation and filter-wise parallelisation and use p_c and p_f to quantify levels of these two types of parallelisation respectively. Equation 3.5d shows the definition of *conv*$_{p_f,p_c,k}$ when taking parallelisation into account, and Equation 3.5f presents its implementation.

The implementation of $conv$ has the following major components. T transforms input feature map in row major to sliding windows by using line buffer. R makes $p_f \times p_c$ number of sliding window streams, which will be passed to C to perform $p_f \times p_c$ parallel *conv2d* computation. For each output filter, p_c number of *conv2d* output should then be reduced to one value by A. At last, there is a cross-channel result accumulator S, which starts to give correct result after $(h - k + 1) \times (w - k + 1)$ for every filter output. Note that we need B (Equation 3.5m) to reset the counter after all channels for a filter have been accomplished. Practically, we also append a *bundle* after S and use *drop* to take the final output data.

$$\langle\langle\langle x\rangle_k\rangle_k, \langle\langle w\rangle_k\rangle_k\rangle \ \textit{conv2d}_k \ y \Rightarrow y = \sum_{i=1}^{k}\sum_{j=1}^{k} x_{i,j} \times w_{i,j} \tag{3.5a}$$

$$\textit{conv2d}_k = \left[\texttt{group}_{k,k},\ \texttt{group}_{k,k}\right]^{-1};\ \textit{dotprod}_{k^2,k^2} \tag{3.5b}$$

$$x\ \mathit{lbuf}_{k,w}\ \langle\langle y\rangle_k\rangle_k \Rightarrow \forall i \in \{1,\ldots,k\}.\, \forall j \in \{1,\ldots,k\}. \quad x\ \mathcal{D}^{(k-i)w+(k-j)}\ y_{i,j} \tag{3.5c}$$

$$\langle\langle x\rangle_{p_c}, W\rangle\ \mathit{conv}_{p_f,p_c,k}\ \langle y\rangle_{p_f} \Rightarrow \forall i \in \{1,\ldots,p_f\}.\, \forall j \in \{1,\ldots,p_c\}. \quad y_i = \sum_{j,p,q=1}^{p_c,k,k} z_{s,p,q} \times w_{i,j,p,q}\ \bigwedge\ x_j\ \mathit{lbuf}_{k,w}\ \langle\langle z_j\rangle_k\rangle_k \tag{3.5d}$$

$$W = \langle\langle\langle\langle w\rangle_k\rangle_k\rangle_{p_c}\rangle_{p_f} \tag{3.5e}$$

$$\mathit{conv}_{p_f,p_c,k} = [R,\ G^{-1}];\ C;\ G;\ A;\ S \tag{3.5f}$$

$$R = \mathit{mfork}_{p_f};\ G^{-1};\ \mathtt{map}_{p_f \times p_c}\ T \tag{3.5g}$$

$$T = \mathit{lbuf}_{k,w};\ \mathtt{bundle}_{h\times w};\ \mathit{drop2d}_{k-1,k-1,h,w};\ \mathtt{inv\_bundle}_{(h-k+1)\times(w-k+1)} \tag{3.5h}$$

$$G = \mathtt{group}_{p_f,p_c} \tag{3.5i}$$

$$A = \mathtt{map}_{p_f}\ (\mathit{pbrt}_{p_c}\ \mathit{add}) \tag{3.5j}$$

$$C = \mathtt{map}_{p_f \times p_c}\ \mathit{conv2d}_k \tag{3.5k}$$

$$S = \mathtt{map}_{p_f}\ \mathit{loop}\ \left(add;\ (\mathcal{D}\ 0)^{(h-k+1)(w-k+1)};\ fork; \mathtt{snd}\ B\right) \tag{3.5l}$$

$$B = \pi_1^{-1};\ \mathtt{snd}\ 0;\ \pi_2^{-1};\ \mathtt{fst}\ (\mathit{modcnt}\ p_c\ c;\ \pi_1^{-1};\ \mathtt{snd}\ (c - pc);\ eq);\ \mathtt{muxr}_2 \tag{3.5m}$$

$$\mathit{drop2d}_{m,n,h,w} = \mathtt{group}_{h,w};\ \mathtt{map}_h\ \mathit{drop}_{n,w};\ \mathit{drop}_{m,h};\ \mathtt{group}^{-1}_{h-m,w-n} \tag{3.5n}$$

The properties of the *conv* relation are listed as follows. Regarding number of latches, Most of the latches are used in $lbuf$, the number of which is $p_c p_f k(k-1)(w/2+1)$. Latches are also used in the *pbrt* of A and C, the maximum number of which is, according to Table 3.1, $2p_f(p_c-1)+2k^2$. The first term is the number of latches used in A and the second term is for the *conv2d* relation in C. In S, there are also $p_f(h-k+1)(w-k+1)$ number of latches need to be counted. The number of arithmetic units used in $conv$ is easier to evaluate, which is $2p_f p_c k^2 + p_f(p_c-1)+p_f$ from *conv2d*, A, and S. The latency of *conv* is the number of cycles it takes from the first input to the output of the first element in the output feature map. The length of the latency mainly depends on p_f and p_c, which is $c/p_c \times hw$, the total number of cycles to iterate through all elements in all channels of the input feature map. Note that the latency of the *dotprod* can be fully covered in this case. In short, all the properties of the *conv* relation are highly depending on the value of p_f and p_c. See Table 3.2 for the final result.

Fully-Connected Relation

Comparing with the implementation of *conv2d*, the fully-connected relation (*fc*) is much simpler. Its core relation is *dotprod*, however, we need to more parameters to indicate the level of parallelisation. Suppose p_r is the level of parallelisation in row and p_c is for column, then the definition of *fc* can be defined as Equation 3.6a. Note that the right hand side of Equation 3.6a explicitly shows the effect of serialisation, which is already considered in *dotprod*. The implementation of *fc* is based on *dotprod*, with additional relations to reshape its domain. Some fixed variables related to the shape of this fully-connected layer, such as m and n, are not placed in the parameter list of *fc*.

The properties of *fc* can also be evaluated straightforwardly. All the properties of *fc* equal to corresponding properties of *dotprod*, as the implementation of *fc* suggests. Only the configuration parameters for *dotprod* and the number of *dotprod* instances are different. These properties are also listed in Table 3.2.

$$\left\langle \langle x \rangle_{p_c}, \langle \langle w \rangle_{p_c} \rangle_{p_r} \right\rangle fc_{p_r,p_c} \langle y \rangle_{p_r} \Rightarrow \forall t \in \{1,\ldots,n,p_c\}.\, \forall i \in \{1,\ldots,p_r\}. \tag{3.6a}$$
$$(y_{0,i} = 0) \bigwedge$$
$$\left(y_{t,i} = y_{t-1,i} + \sum_{j=1}^{p_c} x_j \times w_{i,j} \right)$$
$$fc_{p_r,p_c} = \texttt{fst}\ \texttt{mfork}_{p_r};\ \texttt{zip}_{p_r};\ \texttt{map}_{p_r}\ dotprod_{n,p_c} \tag{3.6b}$$

Max-Pooling Relation

This section presents the implementation of the max-pooling layer in a RubyConv relation. Because max-pooling are often used with a 2×2 kernel configuration and the sliding stride is 2, the pooling relation in RubyConv only supports this case.

The core relation of the max-pooling is *pool2d*, which takes a stream of input feature map as its domain and generates *valid* maximum value of a window in its range. Equation 3.7a shows the definition. Note that it explicitly shows the current tick and under which conditions the result will be valid. Equation 3.7b further presents the implementation. In that implementation, the second row shows how the 2×2 kernel is collected by an approach similar to line buffer (Equation 3.5c), and how to find the maximum value among 4 elements. The third row of the implementation filters out invalid results by `bundle` and `inv_bundle`.

Based on *pool2d*, the implementation of the *pool* relation is obvious. *pool* takes input feature map in parallel, with the level of parallelisation specified as p, see Equation 3.7c for the definition. Also, the implementation of *pool*, which is listed in Equation 3.7d, just replicates *pool2d* into p instances.

The properties of *pool* is summarised in Table 3.2. This max-pooling relation takes at least $w + 2$ cycles to produce the first output. The number of latches depends on the second row of *pool2d* implementation (Equation 3.7b), which is $2(w + 1)$. Moreover, there is no significant usage in arithmetic elements.

$$x\ pool2d\ y \Rightarrow \forall t \in \{1,\ldots,h \times w\} \wedge \tag{3.7a}$$
$$(t\%w \equiv 0 \mod 2) \wedge (t/w \equiv 0 \mod 2).$$
$$y_t = \max\left(x_t, x_{t-1}, x_{t-w}, x_{t-w-1}\right)$$
$$pool2d = \texttt{mfork}_4; \tag{3.7b}$$
$$\left[\mathcal{D}^{w+1}, \mathcal{D}^{w}, \mathcal{D}^{1}, \iota, \right];\ \texttt{group 2 2};\ \left[\texttt{max}, \texttt{max}\right];\ \texttt{max};$$
$$\texttt{bundle}_{2w};\ \texttt{half}_{w};\ \texttt{group}_{w/2,2};\ \texttt{map}_{w/2}\ \pi_2;\ \texttt{inv\_bundle}_{w/2}$$
$$\langle x \rangle_p\ pool_p\ \langle y \rangle_p \Rightarrow \forall t \in \{1,\ldots,h \times w\} \wedge \tag{3.7c}$$
$$\forall i \in \{1,\ldots,p\} \wedge$$
$$y_{t,i} = \max\left(x_{t,i}, x_{t-1,i}, x_{t-w,i}, x_{t-w-1,i}\right)$$

$$pool_p = \texttt{map}_p\ pool2d \tag{3.7d}$$

ReLU relation

The relation that implements ReLU in RubyConv is just one line of code (Equation 3.8). It first creates a pair of the input value and constant 0, then based on this pair, generate a boolean that identifies whether the input value is larger than 0. At last, a multiplexer selects the input value if it is larger than 0, or it outputs 0. Note that because *relu* has very simple interface and it can be easily integrated with other relation, we don't need to give it a parameter to configure parallelisation.

This relation has no latency, no latch usage, and no typical arithmetic units.

$$relu = \pi_1^{-1};\ \texttt{snd}\ 0;\ \texttt{fork};\ \texttt{fst}\ \texttt{ltn};\ \texttt{muxr}_2 \tag{3.8}$$

Summary

In this section, we present four relations that are necessary to construct a CNN hardware design: *conv*, *fc*, *pool*, *relu*. These relations have clear definition and they are parameterised to be applied for different use cases and performance enhancement. Table 3.2 summarises these relations with key properties.

Name	Latency	Maximum Number of Latches	Number of Units
conv	hwc/pc	$p_f\Big[p_c \times k(k-1)(w/2+1)+ 2(p_c-1)+ (h-k+1)(w-k+1)\Big]$	$p_f\Big[2p_ck^2+(p_c-1)+1\Big]$
fc	$\lceil \log_2 p_c \rceil + n/p_c$	$2p_fp_c$	$2p_fp_c$
pool	$w+2$	$2(w+1)$	0
relu	0	0	0

Table 3.2: Summary of properties of each relation or function.

3.2 Evaluation

In this section, we evaluate RubyConv for its correctness by symbolic simulation and flexibility by constructing LeNet-5. The correctness of RubyConv relations should be evaluated by implementing, compiling, and simulating them with the Ruby tool-chain. It is risky and error-prone to trust in paperwork without direct implementation. The correctness of key relations is evaluated by *symbolic simulation* feature provided by the Ruby compiler. We post the result of simulation in some selected cases here to show that the relation is correctly implemented. The flexibility is then evaluated by constructing a typical CNN model, LeNet-5, in RubyConv.

The Ruby toolchain we choose to use for evaluation is named Rebecca, which compiles Ruby source code into *rbs* format that stores relations among wires directly in tables. Rebecca also supports symbolic simulation on generated rbs file. The Rebecca version we use is built from the `changeset 102:2a2bada07e2b` of the codebase managed in Mercurial.

3.2.1 Symbolic Simulation

We go through every relation mentioned before and post their symbolic simulation under a small but representative configuration. The explanation of each result is also attached.

Pipelined Binary Reduction Tree

pbrt is evaluated with a simple case of an adder tree. The input vector has length 9, which is a common case when using *pbrt* for 3×3 convolution layer reduction. Below is the test case and the corresponding simulation result. We simulate for 3 cycles and 4 extra cycles to make sure all simulated cycles can produce valid results.

The result shows that the value in the domain of $pbrt$ is a single value reduced from the values in the range in a binary tree.

```
# pbrt_test.rby
current = pbrt 9 add .

> re -s 3 --extra-cycles 4 "x1 x2 x3 x4 x5 x6 x7 x8 x9"
Simulation start :

    0 - <x1_0,x2_0,x3_0,x4_0,x5_0,x6_0,x7_0,x8_0,x9_0> ~
        <(x1_0 + (((x2_0 + x3_0) + (x4_0 + x5_0)) +
         ((x6_0 + x7_0) + (x8_0 + x9_0))))>
    1 - <x1_1,x2_1,x3_1,x4_1,x5_1,x6_1,x7_1,x8_1,x9_1> ~
        <(x1_1 + (((x2_1 + x3_1) + (x4_1 + x5_1)) +
         ((x6_1 + x7_1) + (x8_1 + x9_1))))>
    2 - <x1_2,x2_2,x3_2,x4_2,x5_2,x6_2,x7_2,x8_2,x9_2> ~
        <(x1_2 + (((x2_2 + x3_2) + (x4_2 + x5_2)) +
         ((x6_2 + x7_2) + (x8_2 + x9_2))))>

Simulation end :
```

Dot-Product

We evaluate a very simple case of *dotprod* with $v = 3$ and $n = 9$. We run the simulation for 1 cycle and 4 extra cycles. Here 4 is calculated by $\lceil \log_2 3 \rceil + 9/3 - 1 = 4$, same as expected from Table 3.1. The result shows that $dotprod$ can successfully handle multiple chunks of input, and there is no invalid data generated.

```
# dotprod_test.rby
current = dotprod 9 3 .

> re -s 1 --extra-cycles 4 "x1 x2 x3 w1 w2 w3"
Simulation start :

    0 - <<x1_0,x2_0,x3_0>,<w1_0,w2_0,w3_0>> ~
     (((x1_2 * w1_2) + ((x2_2 * w2_2) + (x3_2 * w3_2))) +
      (((x1_1 * w1_1) + ((x2_1 * w2_1) + (x3_1 * w3_1))) +
       (((x1_0 * w1_0) + ((x2_0 * w2_0) + (x3_0 * w3_0))) + 0)))

Simulation end :
```

Line Buffer

Here we explicitly evaluate the line buffer relation ($lbuf$) in $conv$. In the definition of line buffer (Equation 3.5c) we don't mention its implementation. We apply Hoare's Rule and `row` to optimise the latches usage in line buffers. The Rebecca code below shows about the detail.

```
# lbuf.rby
lbuf2d k w = fork;
snd (
    pi1^~1;
    row (k-1) (pi1; D^w; fork);
    pi1
);
apl (k-1);
map k (lbuf1d k);
rev k.
```

We also simulate the result of a 4×4 2D input feature map with a 3×3 kernel in 4 cycles and 11 extra cycles. The number 11 is the latency it takes to collect all elements. Results are listed below. Note that the data buffered in the range are all correct.

```
# lbuf_test.rby
current = lbuf 3 4 .

> re -s 4 --extra-cycles 11 "x"
Simulation start :

    0 - x_0 ~ <<x_0,x_1,x_2>,<x_4,x_5,x_6>,<x_8,x_9,x_10>>
    1 - x_1 ~ <<x_1,x_2,x_3>,<x_5,x_6,x_7>,<x_9,x_10,x_11>>
    2 - x_2 ~ <<x_2,x_3,x_4>,<x_6,x_7,x_8>,<x_10,x_11,x_12>>
    3 - x_3 ~ <<x_3,x_4,x_5>,<x_7,x_8,x_9>,<x_11,x_12,x_13>>

Simulation end :
```

Convolution Layer

The $conv$ relation is the most complex one in RubyConv out of no doubt. The following code snippet shows how $conv$ is finally implemented in Rebecca. Comparing with the definition defined in Equation 3.5f, the implementation here uses a `reset` relation, which is based on *cmx*, to reset the accumulated value in a period of $(h-k+1)(w-k+1) \times c/pc$ cycles.

```
# conv.rby
conv pf pc c h w k =
  ( [ ( convR pf pc h w k),
      ( ( convG pf pc )^~1;
      ( map (pf*pc)
          ( map (k*k)
              (mfork (h*w); inv_bundle (h*w))))) ];
    zip (pf * pc);
    # core conv2d
    map (pf * pc) (conv2d k);
    group pf pc;
    map pf (pbrt pc add);
    # cross-channel accumulation
    map pf (
```

```
      LET oh = h - k + 1 IN (
      LET ow = w - k + 1 IN (
        loop (
          add; (DI 0)^(oh*ow); fork;
          reset ((c/pc)*oh*ow)
        );
        AD^(oh*ow);
        ( bundle (c/pc*oh*ow);
        ( drop ((c/pc-1)*oh*ow) (c/pc*oh*ow));
          inv_bundle (oh*ow) )
      )
      END )
      END )
  ).
```

We also evaluate its correctness by symbolic simulation. Here we choose a very simple case which has $(p_f, p_c, c, h, w, k) = (2, 2, 4, 4, 4, 1)$. We evaluate it in 4 cycles to show the final results for the first 2 filters in parallel. Results are all correct.

```
# conv_test.rby
current = conv 2 2 4 4 4 1.

re -s 4 --extra-cycles 8 "x1 x2 w11 w12 w21 w22"
Simulation start :

    0 - <<x1_0,x2_0>,<<<w11_0>,<w12_0>>,<<w21_0>,<w22_0>>>> ~
        <(((x1_4 * w11_1) + (x2_4 * w12_1)) +
         (((x1_0 * w11_0) + (x2_0 * w12_0)) + 0)),
         (((x1_4 * w21_1) + (x2_4 * w22_1)) +
         (((x1_0 * w21_0) + (x2_0 * w22_0)) + 0))>
    1 - <<x1_1,x2_1>,<<<w11_1>,<w12_1>>,<<w21_1>,<w22_1>>>> ~
        <(((x1_5 * w11_1) + (x2_5 * w12_1)) +
         (((x1_1 * w11_0) + (x2_1 * w12_0)) + 0)),
         (((x1_5 * w21_1) + (x2_5 * w22_1)) +
         (((x1_1 * w21_0) + (x2_1 * w22_0)) + 0))>
    2 - <<x1_2,x2_2>,<<<w11_2>,<w12_2>>,<<w21_2>,<w22_2>>>> ~
        <(((x1_6 * w11_1) + (x2_6 * w12_1)) +
         (((x1_2 * w11_0) + (x2_2 * w12_0)) + 0)),
         (((x1_6 * w21_1) + (x2_6 * w22_1)) +
         (((x1_2 * w21_0) + (x2_2 * w22_0)) + 0))>
    3 - <<x1_3,x2_3>,<<<w11_3>,<w12_3>>,<<w21_3>,<w22_3>>>> ~
        <(((x1_7 * w11_1) + (x2_7 * w12_1)) +
         (((x1_3 * w11_0) + (x2_3 * w12_0)) + 0)),
         (((x1_7 * w21_1) + (x2_7 * w22_1)) +
         (((x1_3 * w21_0) + (x2_3 * w22_0)) + 0))>

Simulation end :
```

Fully-Connected Layer and Max-Pooling Layer

As we evaluate the correctness of $dotprod$, there is no need to explicitly evaluate fc in this section. Regarding *pool*, we evaluate an instance with $w = 4$ for 2 cycles and 6 extra cycles. The result is listed as follows, and shows that the *pool* relation can correctly extract the maximum value in the corresponding window.

```
# conv_test.rby
current = pool 1 1 4.
```

```
re -s 2 --extra-cycles 6 "x"
Simulation start :

    0 - x_0 ~ (max <(max <x_0, x_1>), (max <x_4, x_5>)>)
    1 - x_1 ~ (max <(max <x_2, x_3>), (max <x_6, x_7>)>)

Simulation end :
```

3.2.2 Case Study: LeNet-5

Below is the relation that defines LeNet-5 constructed by RubyConv. Except for those built-in relations in Ruby, this relation uses only relations from RubyConv. All these layer relations are connected through *beside* ($\leftrightarrow$), just like `row` in Ruby, and there is a π_2 at the end to project the final result. In this implementation, we use an implementation without parallelisation to make it clear. Note that we don't put symbolic evaluation of this relation here because they are hard to read and will take a lot of space. Also, we don't post symbolic simulation for `lenet5` is also due to the performance issue in Rebecca: our experience showed that the compilation process lasted for about 1h 14m on a server and exited with no verbose output.

```
lenet5 =
(
  ( conv 1 1 1 28 28 5; pi2^~1 )             <->
  ( pi1; (pool 1 1 24; relu) \ [-]; pi2^~1 ) <->
  ( conv 1 1 32 12 12 5; pi2^~1 )            <->
  ( pi1; (pool 1 1 8; relu) \ [-]; pi2^~1 )  <->
  ( fc 1 (4 * 4 * 64) 1; relu; [-]; pi2^~1)  <->
  ( fc 1 1024 1; [-]; relu; pi2^~1)
); pi2.
```

3.3 Summary

This section presents RubyConv, a CNN hardware library written in Ruby, a relation-based high-level hardware description language. RubyConv contains essential relations to build a typical CNN, including *conv*, *fc*, *pool*, and *relu*. These relations are well defined and listed with their detailed implementation. The properties of these relations are also well studied. In the evaluation section, we use symbolic simulation to check the correctness of each relation, and build LeNet-5 with only RubyConv supported.

Chapter 4

MaxDeep: OpenSPL-based CNN Hardware Library

MaxDeep is an OpenSPL hardware library providing building blocks to construct CNN hardware designs on the Maxeler FPGA platform. The primary difference between MaxDeep and RubyConv (Chapter 3) is that hardware designs generated by MaxDeep can be directly synthesised on real FPGA device while those generated by RubyConv cannot. This is also the key motivation for designing MaxDeep. Unlike RubyConv which provides the only essential building block designs for CNN, MaxDeep has 3 major components providing functionalities for a whole runnable FPGA system, including a collection of OpenSPL *packages* wrapping parameterised CNN hardware building block designs, *analysis models* to predict resource usage and performance from design parameters immediately, and an *optimisation flow* that explores the design space.

Although the difference between RubyConv and MaxDeep is significant, the connection between these two libraries cannot be ignored. MaxDeep and RubyConv are in different *levels of abstraction* for the same problem (CNN hardware library): RubyConv focuses more on *formal descriptions* of building blocks and their validation of correctness, while MaxDeep elaborates more on *systematic* and *architectural* parts. Based on this idea, we can discover that designs of core CNN layers are almost *identical* in both RubyConv and MaxDeep — even the parameter lists and interfaces are similar, although they use different hardware description languages. Besides, analysis models in MaxDeep can be viewed as advanced and extended versions of models used in discussing relation properties in RubyConv. What is more interesting is that, if the transpiler between Ruby and OpenSPL is ready to use, it is possible to *generate* the core of MaxDeep directly from RubyConv and we only need to provide peripheral and platform-specific modules in MaxDeep. This also gives us an insight that this *methodology* can be adapted to create a CNN hardware library in another language on another platform. These connections are also mentioned and discussed in detail in later sections.

It is worthwhile to clarify that MaxDeep is not just an extension or an implementation of RubyConv on the Maxeler platform — MaxDeep provides novel optimisation options in the core building block library. For the convolution layer, which commonly consumes most resources and computation time comparing with other blocks, MaxDeep offers two options: replace it with *depthwise separable convolution* or *quantize* it into shorter data types, even binary values. These optimisation techniques are remarkable, and their usage is highly relied on Chapter 5. Also, MaxDeep considers not only level of parallelisation but also computation sequence and layer-wise connection when maximising the performance of standard convolution layers. More parameters are introduced and it is more likely to get high performance in MaxDeep.

Ruby	OpenSPL
dotprod	`DotProductKernel`
conv	`ConvLayerKernel`
fc	`FullyConnectedLayerKernel`
pool	`PoolingLayerKernel`
relu	(no individual kernel, can be specified in other kernels)
×	`BatchNormLayerKernel`
×	`DepthwiseSeparableConvLayerKernel`
×	`BinarisedConvLayerKernel`

Table 4.1: Mapping from Ruby-based blocks to MaxJ-based blocks

Another point that MaxDeep outperforms RubyConv is the analysis model and optimisation flow. The model in RubyConv can only give a coarse idea about the performance and resource usage of a given design, measured in latency and number of latches plus key arithmetic units respectively. Meanwhile, analysis models in MaxDeep are based on reports from real hardware designs: they have solid assumptions discovered from patterns in reports, they have clearly specified the methodology to predict results for different metrics, and they are evaluated to prove their accuracy. Also, the optimisation flow built upon these models is also a considerable breakthrough comparing with RubyConv. We can use this flow to predict and generate the best performing design for given configuration without tedious experiments.

Thus, in short, contributions of MaxDeep described in this chapter are listed as follows:

1. We devise MaxDeep, an OpenSPL-based CNN hardware library, which not only contains essential CNN building blocks, but also specifically optimised ones like depthwise separable convolution and binarised convolution. For those layers that have been discussed in RubyConv, MaxDeep versions have more optimisation parameters and are more likely to achieve better performance.

2. The resource usage and performance can be predicted directly be a collection of analysis models, which are devised with solid assumptions and evaluated on real hardware builds. An optimisation flow can also cooperate with these analysis models to achieve high-performance when selecting configuration parameters.

3. Key design metrics of MaxDeep for given CNN models are evaluated on the MAX4 board. which belongs to the Maxeler platform and integrated with a Stratix V FPGA core. Results show that MaxDeep is both flexible and fast.

This chapter contains 3 sections. Section 4.1 presents OpenSPL building blocks in MaxDeep, Section 4.2 provides a thorough introduction on analysis models and optimisation techniques, and Section 4.3 illustrates evaluation results.

Additionally, the mapping from RubyConv relations to OpenSPL kernels are listed in Table 4.1.

4.1 OpenSPL-based Description

This section introduces all types of CNN layers that are supported by MaxDeep, including their optimised forms and connected layers. We first look at convolution layers: standard (Section 4.1.1),

binarised (Section 4.1.2), and depthwise separable convolution (Section 4.1.3) are all covered. Then, we review other types of layer (Section 4.1.4), most of which are already discussed in Chapter 3. At last, we discuss how to use MaxDeep to construct a CNN model (Section 4.1.5) and related properties. Each building block in this section will have a brief report on its design metrics. A more accurate and complete version is introduced in Section 4.2.

4.1.1 Convolution Layer

Convolution layer is implemented as `ConvLayerKernel` in MaxDeep, which is directly mapped from *conv* in RubyConv. `ConvLayerKernel` can be configured to compute different standard convolution layers, with supported data types and bit width. Unlike *conv* in RubyConv, `ConvLayerKernel` has three *levels of parallelisation* (P_C, P_F, P_K) rather than two. Also, `ConvLayerKernel` explicitly considers all three different *computation sequences* of convolution: *filter-major*, *channel-major*, *pixel-major*. Computation sequence is a critical property of the convolution layer hardware design. It affects all design metrics and the format of input and output streams. In general, the three computation sequences are three sequences of the triple for-loop in the convolution layer implementation: the suffix *-major* of each computation sequence name represents which index (filter, channel, or pixel) is the major index of the triple loop.

We start with the architecture of this layer.

Architecture

The architecture of convolution layer processing unit (Figure 4.1) is organised in three *levels* of inner blocks: *core* block implements 2D convolution; *array* block places multiple *core* blocks in parallel; *top* block wraps the *array* block with buffers and external interfaces. Following discussions for the architecture will be based on this high-level organisation concept.

The *core* block is directly mapped from *conv2d*. It's interface and inner design only depend on the kernel edge length (K) of the convolution layer that is going to be implemented. `DotProductKernel` is the OpenSPL implementation of these core arithmetic blocks.

The *array* block places *core* blocks in three dimensions, and each dimension is related to a level of parallelisation.

1. P_K number of *core* blocks in the *kernel* dimension process P_K number of 2D convolution operations within the same channel. For these P_K blocks, input feature map data chunks are adjacent in a sliding window series, and coefficient data chunks are identical.

2. P_C number of *core* block vectors, each of which has length P_K, are in the *channel* dimension and handle input for multiple channels. There are P_K number of adder trees to reduce output results across P_C channels.

3. P_F number of *core* block matrices ($P_C \times P_K$) are in the *filter* dimension and prepare results for multiple filters.

Based on this architecture, the *array* block takes $P_C \times K \times (K + P_K - 1)$ number of entries from the input feature map and $P_F \times P_C \times K^2$ number of coefficient values, and it outputs $P_F \times P_K$ number of intermediate output feature map values.

`Conv2DKernel` implements the *array* block.

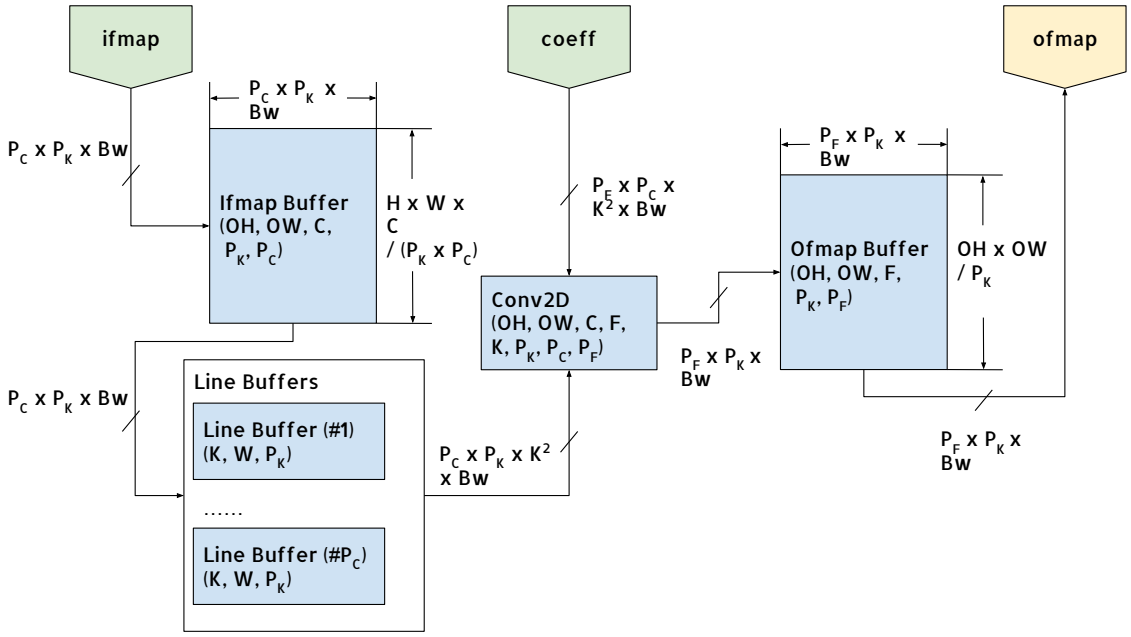

Figure 4.1: The architecture of standard convolution layer in MaxDeep.

The *top* block finally initialises interfaces to the external devices and creates buffers to reuse and prepare data for the *array* block. There are two input streams (*ifmap* for input feature map and *coeff* for coefficients) and one output stream (*ofmap*) for output feature map in the *top* block. Table 4.2 summarises properties of these streams.

Stream	Width
ifmap	$P_C \times P_K \times Bw$
coeff	$P_F \times P_C \times K^2 \times Bw$
ofmap	$P_F \times P_K \times Bw$

Table 4.2: Streams connected to and from the *top* block, Bw refers to bit width of the data type.

Besides, there are also *four* kinds of buffers used in the *top* block: *ifmap buffer*, *line buffer*, *coeff buffer*, and *ofmap buffer*. Among these buffers, the *line buffer* is a bit tricky: its definition is the same as Equation 3.5c but its implementation is through `stream.offset`, which only exists in OpenSPL. The properties of these buffers are highly related to the parallelisation parameters and computation sequences. Table 4.3 lists key properties of these buffers. Note that in MaxDeep, a buffer with depth 1 are implemented as a register file, while others are implemented in on-chip memory.

Design Metrics

This section relates design metrics, which are *latency*, *throughput*, *resource usage*, and *bandwidth*, with convolution layer parameters.

1. *Latency*: Latency of `ConvLayerKernel` is defined as the number of clock cycles taken from the

Buffer	Seq	Port Width	Depth	Usage
ifmap buffer	C	$P_C P_K Bw$	$\frac{HW}{P_C P_K}$	store one channel of the input fmap
	F	$P_C P_K Bw$	$\frac{HWC}{P_C P_K}$	store the whole input fmap
	P	$P_C P_K Bw$	$\frac{HWC}{P_C P_K}$	store the whole input fmap
line buffer	C	$P_C P_K Bw$ (in) $P_C P_K K^2 Bw$ (out)	$\frac{WK}{P_K}$	prepare data in windows for P_C channels
	F	$P_C P_K Bw$ (in) $P_C P_K K^2 Bw$ (out)	$\frac{WK}{P_K}$	prepare data in windows for P_C channels
	P	$P_C P_K Bw$ (in) $P_C P_K K^2 Bw$ (out)	$\frac{CWK}{P_C P_K}$	prepare data in windows for all channels
coeff buffer	C, F, P	$P_F P_C K^2 Bw$	1	cache the current coefficient kernel
ofmap buffer	C	$P_F P_K Bw$	$\frac{H_O W_O F}{P_F P_K}$	store the whole output feature map
	F	$P_F P_K Bw$	$\frac{H_O W_O}{P_F P_K}$	store one channel of the output feature map
	P	$P_F P_K Bw$	1	store only $P_K P_F$ number of elements

Table 4.3: Buffers and their description in the *top* block. **Seq** is the computation sequence chosen, can be *filter-major* (F), *channel-major* (C), or *pixel-major* (P).

start of the computation to the output of the first channel of the first pixel in the output feature map, the same as what we refer to in Table 3.2. Computation sequence is the major factor that determines latency, which is clarified in Table 4.4. The initial interval (II) includes the number of cycles of reading and writing buffers, propagating results through registers, and filling the pipelines within core arithmetic units. Comparing with the number of cycles for computation followed by initial interval, which is approximately hundreds to thousands cycles, II can be ignored. Intuitively, *channel-major* has the **longest** latency and *pixel-major* has the **shortest**.

Sequence	Latency
channel-major	$II + [(C-1) \times H \times W \times F]/(P_F \times P_C \times P_K)$
filter-major	$II + (C \times H \times W)/(P_C \times P_K)$
pixel-major	$II + (C \times F)/(P_C \times P_F)$

Table 4.4: Latency of `ConvLayerKernel`, categorised by computation sequence. II here still stands for initial interval.

2. *Throughput*: Unlike latency, throughput will *not* be affected by computation sequences. To be specific, the number of clock cycles to finish processing a single input feature map and ready to accept the next one, which is the definition of the throughput of `ConvLayerKernel`, is listed in Equation 4.1. Although throughput of a single `ConvLayerKernel` is unrelated to computation

sequence, the throughput of a series of layers is rather highly related. This property is discussed in Section 4.1.5.

$$Tp = \frac{F \times C \times H \times W}{P_F \times P_C \times P_K} \tag{4.1}$$

3. *Resource usage*: Buffers and arithmetic units (used in `DotProductKernel`) mainly contribute to the resource usage of `ConvLayerKernel`. BRAM is consumed by buffers, the usage of which can be resolved by Table 4.5. Note that the BRAM usage predicted here is the **lower bound** of the real BRAM usage, because there are limited numbers of BRAM blocks with fixed shape and there should be an increase in BRAM usage due to tiling in real build process. For the DSP usage and other resources, they are occupied largely by the dot-product kernel. This relationship is presented by Equation 4.2.

Sequence	BRAM Usage
channel-major	$U^C_{bram} = \frac{HWBw + P_C WKBw + H_O W_O FB_w}{\textit{bits per BRAM}}$
filter-major	$U^F_{bram} = \frac{HWCBw + P_C WKBw + H_O W_O B_w}{\textit{bits per BRAM}}$
pixel-major	$U^P_{bram} = \frac{HWCBw + CWKB}{\textit{bits per BRAM}}$

Table 4.5: BRAM usage of `ConvLayerKernel`, categorised by computation sequence

$$\begin{aligned} U^{seq}_t &= P_F \times P_C \times P_K \times U^{dp}_t \\ t &\in \{\textit{lut, ff, dsp}\} \\ seq &\in \{\textit{C, F, P}\} \end{aligned} \tag{4.2}$$

4. *Bandwidth*: The bandwidth here refers to the maximum number of elements that should be transmitted between CPU and FPGA. For the standard convolution, this value can be computed by Equation 4.3. The conclusion of that equation can also be discovered from Figure 4.1, by adding up the width of each FIFO that connects to the external system.

$$Bd = (P_C \times P_K + P_C \times P_F \times K^2 + P_F \times P_K) \times Bw \tag{4.3}$$

Summary

This section presents the architecture of the standard convolution layer in MaxDeep and its design metrics. Comparing with *conv* presented in Section 3.1.2, `ConvLayerKernel` explicitly uses on-chip memory as buffers to schedule or reshape streams while *conv* can only use delays to model. Although it is not discussed before, the computation sequence of *conv* is indeed *filter-major*. Other minor differences include additional parameters (P_K, Bw) and design metrics.

4.1.2 Binarised Convolution Layer

Binarised convolution layer is an extremely quantized convolution layer in the *bit width dimension* in MaxDeep. It is almost a standard convolution layer: the binarised convolution layer takes the same list

of parameters and has the same interface as the standard one, such as the number of parallel blocks and computation sequence. Major differences are: Operands are in binary data type; Array of multipliers and adder tree in `DotProductKernel` is replaced by XNOR ($\oplus$) and *popcount* respectively; A *batch normalisation* unit implemented by threshold should be appended in the end of processing.

Architecture

The architecture of binarised convolution layer is constructed by replacing the standard dot-product kernel with binarised dot-product kernel. The binarised dot-product kernel uses an array of binary XNOR units to replace multipliers. This replacement can be reasoned by Table 4.6. Note that the representation of -1 is converted to 0 in hardware. By replacing multipliers with XNOR units, the latency and resource usage of the binarised dot-product block can be reduced.

Original Operands	Multiply Result	Converted Operands	XNOR Result
$(1, 1)$	1	$(1, 1)$	1
$(1, -1)$	-1	$(1, 0)$	0
$(-1, 1)$	-1	$(0, 1)$	0
$(-1, -1)$	1	$(0, 0)$	1

Table 4.6: Reasoning the replacement from multipliers to XNOR units.

An adder tree that counts the number of positive ones in the output of the XNOR units array is appended. Counting the number of ones is equivalent to accumulating values in the binary vector. Although there are various optimisation techniques related to this operation, which is known as *popcount*, MaxDeep applies a straightforward version that uses an adder tree to accumulate number of positive ones in one cycle. Note that while accumulating the the result, the data type of the operand cannot remain binary and should at least have bit width $1 + \log_2(K^2)$. In order to have a binarised result, a comparison between the accumulated value and a threshold is required: if the accumulated value is above the threshold, the final output value should be positive one; otherwise negative one. This is the trick used by Umuroglu et al. (2017) to implement binarised batch normalisation.

Design Metrics

The design metrics are almost the same as those for standard convolution, but several functions should be revised and are listed as follows.

1. The resource required for dot-product is not related to multipliers and is related to XNOR units in this case, and adders in the accumulation tree will consume resource base on fixed bit width $1 + \log_2(K^2)$. No DSP will be used because no multipliers are used.

$$
\begin{aligned}
&U^{res}_{bdp}(H, W, C, F, K, P_F, P_C, P_K) = \\
&\quad P_F \times P_C \times P_K \times \left[U^{res}_{xnor}(K^2) + (K^2 - 1) \times U^{res}_{add}(1 + \log_2(K^2))\right] + \\
&\quad P_F \times P_K \times (P_C - 1) \times U^{res}_{add}(1 + \log_2(K^2))
\end{aligned} \tag{4.4}
$$

2. Input feature map buffer and line buffers consume BRAM based on binary data type, while the output feature map buffer stores value with data type that has bit width $1 + \log_2(K^2)$, as the output feature map buffer caches accumulated data.

4.1.3 Depthwise Separable Convolution Layer

Depthwise separable convolution layer (Section 2.1.2) is supported in MaxDeep to be an ultimate scalable case [1] . Depthwise separable convolution layer greatly reduces number of parameters required to perform a convolution computation, and its performance can be *scaled* to a larger extent comparing with standard convolution. In MaxDeep, it is straightforward and seamless to take advantage of depthwise separable convolution due to the features as follows.

1. MaxDeep discovers *architectural similarities* between standard and depthwise separable designs: all the building blocks in the depthwise separable convolution design are reused from blocks in the standard convolution design (Section 4.1.1), and MaxDeep only need to reorder and change parameters of these blocks. Also, it is easy to derive an analysis model of depthwise separable convolution by using models of those blocks that are well-studied.

2. The interface of depthwise separable convolution is the same as standard convolution. Parameters for standard convolution has almost the same meaning to depthwise separable convolution, even level of parallelisation and computation sequence. Slight differences will be introduced later.

Thus, it is possible to convert a design for convolution layer from the standard kernel to the depthwise separable kernel with limited revision in MaxDeep, and architectural optimisation algorithm can also be easily tweaked to accept this type of convolution layer as a special scenario.

This section first presents the architecture of depthwise separable convolution layer kernel (Section 4.1.3). And then the analysis model to analyse design metrics will be illustrated (Section 4.1.3).

Architecture

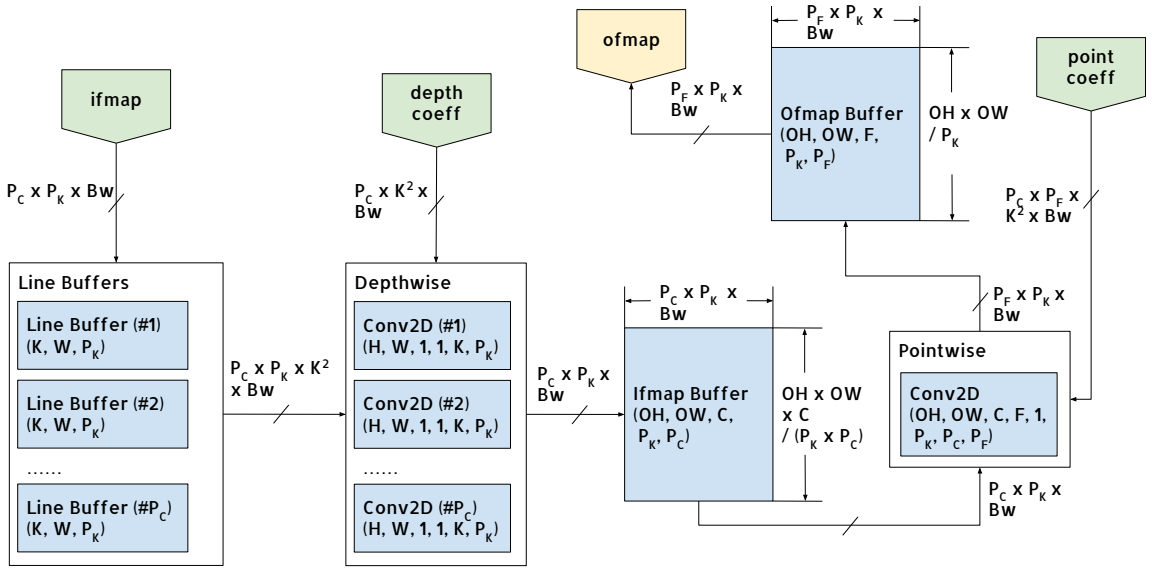

Figure 4.2: Depthwise separabel convolution architecture.

[1] `DepthwiseSeparableConvLayerKernel` implements this type of convolution layer.

The hardware architecture of depthwise separable convolution is illustrated in Figure 4.2. There are only 3 differences when we are comparing depthwise separable convolution with the standard architecture:

1. *Blocks*: Line buffers are moved to the first part of the sequence, and a spatial `Conv2DKernel` array is placed between the group of line buffers and the input feature map buffer.
2. *Streams*: Two separate streams are required to receive coefficients, one for depthwise convolution and the other one for pointwise convolution.
3. *Parameters*: The computation sequence of the depthwise separable convolution only covers the pointwise convolution, and in the current version of MaxDeep only *filter-major* sequence is supported.

Note that with the same convolution layer parameters, standard convolution design has the same interface as the depthwise separable convolution design in MaxDeep, which makes it seamless to convert from one type to another while performing architectural optimisation.

Design Metrics

This section lists design metrics based on the convolution layer parameters. We only add discussion to resource usage, other metrics are easy to reason about.

1. *Resource usage*: LUT, flip-flop, and DSP are mainly consumed by `Conv2DKernel` in depthwise convolution and pointwise convolution, which are mainly composed by `DotProductKernel`. Suppose res is a resource type and $U_{dp}^{res}(L, Bw)$ is the resource usage of `DotProductKernel` with given vector size L and bit width Bw, then the resource usage function for type LUT, FF, and DSP can be derived as Equation 4.5. Note that the last term in the equation is the resource used by adder trees to sum results cross P_C channels in depthwise and pointwise convolution.

$$\begin{aligned} U_{dsc}^{res}(H, W, C, F, K, P_F, P_C, P_K, Bw) = {} & P_C \times P_K \times U_{dp}^{res}(K^2, Bw) \\ & + P_C \times P_F \times P_K \times U_{dp}^{res}(1, Bw) \\ & + (P_F + 1) \times P_K \times (P_C - 1) \times U_{add}^{res}(Bw) \\ res \in {} & \{lut, ff, dsp\} \end{aligned} \tag{4.5}$$

   BRAM is mainly used by buffers and summarised in Equation 4.6:

$$U_{dsc}^{bram}(H, W, C, F, K, P_F, P_C, P_K, Bw) = \frac{\Big[P_C \times W \times K + OH \times OW \times (C + P_F)\Big] \times Bw}{\textit{number of bits per BRAM}} \tag{4.6}$$

2. *Latency*:

$$Lt_{dsc} = \frac{H \times W \times C}{P_C \times P_K} \tag{4.7}$$

3. *Throughput*

$$Tp_{dsc} = \frac{H \times W \times C \times F}{P_C \times P_K \times P_F} \tag{4.8}$$

4. *Bandwidth*

$$Bd_{dsc} = (P_C \times P_K + P_C \times K^2 + P_C \times P_F \times K^2 + P_F \times P_K) \times Bw \tag{4.9}$$

4.1.4 Other Layers

In this section, we briefly introduce how layers other than convolution layers are implemented in MaxDeep. Fully-Connected layer in MaxDeep has the same definition and implementation as the one in RubyConv. There are two parallelisation parameters P_R and P_C in the FC layer, same as p_r and p_c in *fc*. We still use dot-product to implement the core arithmetic in the FC layer. Regarding the Max-Pooling layer, we still only implements 2×2 pooling with stride equals to 2. This configuration is commonly used in many CNN models. We use `stream.offset` to extract data from far points in the stream, just like the implementation of the line buffer. Batch Normalisation is a new layer, but its implementation is quite straightforward. We use *ROM* to store average and variance from the trained CNN model and read them while performing computation.

In the next section, we show how to use convolution layers and layers mentioned in this section to construct a network.

4.1.5 Network

This section discusses how to run a complete CNN model in the MaxDeep framework. There is no straightforward solution to this problem, because each layer can either be placed on an individual hardware module or share a module with other layers of the same type. Also, values of hardware module parameters are highly correlated, such as the parallel parameters in convolution layers should match layers adjacent to it, which makes the problem harder to resolve. This section first focuses on connection patterns between two convolution layers and presents their design metrics. Next, this section shows an example about how to deploy a simple CNN in MaxDeep.

Regarding implementation, layers in MaxDeep are connected through *inter-kernel streams*, which are more than scheduling models: each stream should be instantiated by FIFO. Inter-kernel streams can be specified in OpenSPL managers.

Conv2Conv

The connection between two convolution layers is the most important one among all layer-wise connections in MaxDeep, because they are the most time-consuming blocks and they have the most complex architectures. P_C, P_F, and P_K, which are three parallel parameters in both convolution layers as mentioned before, should satisfy Equation 4.10. This equation aims at matching the width of the first layer's output and the second layer's input. If this equation cannot be satisfied, then the performance of the whole network will be reduced to the slowest layer's performance, and will not benefit from pipeline. Based on Equation 4.10, the total number of parameters can be decreased: there are four parameters ($P_C^{(1)}, P_K^{(1)}, P_F^{(1)}, P_F^{(2)}$) required for two adjacent convolution layers.

$$\begin{cases} P_K^{(1)} = P_K^{(2)} \\ P_F^{(1)} = P_C^{(2)} \end{cases} \tag{4.10}$$

Computation sequence is another important factor that affects design metrics. There are three patterns that have good performance: *filter-channel*, *channel-filter*, and *pixel-pixel*. Latter layers in these patterns can consume output from previous layers instantly, thus the depth of interconnect FIFOs will be small and the pipeline will be more efficient.

It is also worth to note that, placing two adjacent convolution layers both on hardware can increase the performance by at most 2 times, because while the computing the second layer, the first layer is also taking input and computing, and then the computation cost of the second layer can be covered. We put the result of this property in Equation 4.20.

Conv2Pool, Pool2FC, FC2FC

Here we select 3 other typical layer-wise connections in CNN model, and discuss their requirements on parallelisation parameter. For the Conv2Pool case, the level of parallelisation in the filter dimension of the convolution layer should match the channel dimension of the pooling layer, which is $P_F^{(1)} = P_C^{(2)}$. The pooling layer in MaxDeep will reduce P_K by half, due to the effect of stride. When the pooling layer or the convolution layer is connected to the FC layer, the output stream from the previous layer should only be parallelised within the feature map, i.e. $P_F^{(1)} = P_C^{(1)} = 1$. Regarding two FC layers, we simply make sure that $P_R^{(1)} = P_C^{(2)}$, then the output from the first layer can be immediately processed by the second layer.

Case Study: LeNet-5 in MaxDeep

Here we present a case study of building LeNet-5 in MaxDeep. After analysing LeNet-5 with all conditions mentioned in this section, we notice that there are 3 tunable parallelisation parameters, which are P_F in the first convolution layer, P_C in the first FC layer, and P_C in the second FC layer.

Next, to put these layers in the hardware, we only need to configure parameters for each layer, just like the prototxt approach in Caffe. The following code snippet shows how to construct LeNet-5 parameter list that is compatible to MaxDeep. This list of parameters will be passed to create and connect kernels.

```
List<LayerParameters> lp = new ArrayList<LayerParameters>();
ConvLayerParameters cp0 =
  new ConvLayerParameters.Builder(28, 28, 1, 32, 5)
    .name("conv0")
    .BW(ep.getBW())
    .PK(2)
    .PF(PP.get(0))
    .pool(new PoolingLayerParameters(2, 2, Mode.MAX))
    .seq(seq0)
    .type(ep.getUseDepth() ? Type.DEPTHWISE_SEPARABLE : Type.STANDARD)
    .dbg(false)
    .build();
lp.add(cp0);

ConvLayerParameters cp1 =
```

```
  new ConvLayerParameters.Builder(12, 12, 32, 64, 5)
    .name("conv1")
    .BW(ep.getBW())
    .PK(1)
    .PC(PP.get(0))
    .PF(1)
    .seq(CompSeq.FILTER_MAJOR)
    .pool(new PoolingLayerParameters(2, 2, Mode.MAX))
    .type(ep.getUseDepth() ? Type.DEPTHWISE_SEPARABLE : Type.STANDARD)
    .dbg(false)
    .build();
lp.add(cp1);

FullyConnectedLayerParameters fp0 =
  new FullyConnectedLayerParameters(
    "fp0",
    ep.getBW(),
    1024,
    4 * 4 * 64,
    1,
    PP.get(1));
lp.add(fp0);

FullyConnectedLayerParameters fp1 =
  new FullyConnectedLayerParameters(
    "fp1",
    ep.getBW(),
    10,
    1024,
    PP.get(1),
    PP.get(2));
lp.add(fp1);
```

4.2 Analysis Model

The analysis model of MaxDeep aims at predicting resource usage of a design only by its configuration parameters before the design is built. The motivation of this analysis model is to accelerate the design parameter optimisation process. This model can immediately decide whether a set of parameters is feasible on hardware while optimising the design, or several hours are needed to verify parameter values and the whole optimisation process will take ages.

The methodology of building an analysis model is combining *datasheet* with *statistical* methods:

1. *Datasheet* (Section 4.2.1): The usage of some hardware resource types, such as DSP and BRAM, can be statically calculated only by referencing the *datasheet* of the board. A datasheet contains model that directly evaluates resource usage of basic hardware blocks, such as adders, multipliers, and memory blocks.

2. *Statistical method* (Section 4.2.2): The usage of other hardware resource types, such as LUT and FF, cannot be calculated directly, because these resources are scattered in many different blocks and it is very difficult to aggregate resource usage information from all these blocks. However, according to our analysis in previous sections, candidate models of these resource types are *linear* on some hardware parameters, which makes it possible to deduce hyper-parameters of these models by statistical methods, such as linear regression.

The analysis model is mainly derived for the convolution layer, because it occupies most of the resource capacity on board. This argument will be evaluated in Section 4.3.

Besides resource usage model, Section 4.2.3 also presents performance analysis model, which is based on the roofline model, Equation 4.1 (throughput) and Equation 4.3 (bandwidth).

4.2.1 Datasheet: BRAM and DSP

The feasibility of the datasheet method on resource usage of BRAM and DSP is based on the following heuristic assumptions:

1. The BRAM usage is mainly consumed by the buffers of the convolution layer, including input feature map buffer, line buffer, and output feature map buffer.
2. Most of the DSP units are used by multipliers in dot-product blocks.

These assumptions can be reasoned from the convolution layer architecture in Section 4.1.1, and they will also be analysed with real build data in Section 4.3.

BRAM Usage

To utilise the datasheet method with assumptions above, the BRAM usage model of standard convolution layer can be derived by adding up number of bits required for on-chip storage in three types of buffers, which are listed previously in Table 4.3, and dividing it by the number of bits per BRAM block, which can be found in the board datasheet.

Equation 4.11 shows the final model for BRAM usage of standard convolution layer. Note that because the BRAM usage is related to computation sequence, there are three cases in the model. NB_{bram} is the number of bits per BRAM block that can be found in board datasheet. For example, the Stratix V FPGA uses M20K BRAM, which contains 20Kbits in each block. Thus, $NB_{bram} = 20 \times 1024 = 20480$ on Stratix V.

$$
\begin{aligned}
U_{conv}^{bram} &= U_{ibuf}^{bram} + U_{lbuf}^{bram} + U_{obuf}^{bram} \\
&= \begin{cases}
\dfrac{(HW + P_C WK + H_O W_O F)Bw}{NB_{bram}} & (\textit{channel major}) \\
\dfrac{(HWC + P_C WK + H_O W_O)Bw}{NB_{bram}} & (\textit{filter major}) \\
\dfrac{(HWC + CWK)Bw}{NB_{bram}} & (\textit{pixel major})
\end{cases}
\end{aligned}
\tag{4.11}
$$

DSP Usage

DSP usage model can be derived in an alike approach: first retrieve the DSP usage of a multiplier from the datasheet, and then insert this value into the analysis model already built in Section 4.1.1. DSP usage in adders are ignored because most fixed-point adders don't contain DSP.

Equation 4.12 presents this idea. $U_{mul}^{dsp}(Bw)$ is the number of DSP required for a multiplier that has operands with bit width Bw. This value is clearly listed in board datasheet, for example, a Stratix V board uses 2 DSP blocks for a multiplier takes two 32 bits operands. The coefficient factor before U_{mul}^{dsp} is the number of multipliers required for building all dot-product blocks in the design.

$$U_{conv}^{dsp}(P_F, P_C, P_K, K, Bw) = P_F \times P_C \times P_K \times K^2 \times U_{mul}^{dsp}(Bw) \tag{4.12}$$

4.2.2 Statistical Method: LUT and Flip-Flop

Usage of LUT and FF cannot be modelled with datasheet, because almost every block in a hardware design uses them, and it will be quite tedious to discover the relationship among design parameters and all these blocks.

An alternative approach is adapting statistical methods. Intuitively, LUT and FF usage are scaling *linearly* with the number of blocks that consume the major amount of hardware logic — in the convolution layer of MaxDeep, these blocks are *adders* and *multipliers* in dot-product. Thus, it is reasonable to train a linear regression model between the number of arithmetic units and the usage of LUT and FF from a dataset of typical real builds.

This statistical method contains the following steps, which are all integrated into a self-developed Python package named `maxlabor`. The `maxlabor` package is built upon NumPy (Walt et al., 2011), Pandas (McKinney et al., 2010), scikit-learn (Pedregosa et al., 2011), and Matplotlib (Hunter, 2007).

1. *Prepare dataset*: The dataset contains resource usage information of a set of designs built with typical parameters. These designs are all originated from a `Conv2DKernel` wrapped with key interfaces. A set of parameters are selected to bring sufficient varieties to these designs, see Table 4.7.

Parameter	Values
Bit width	`[8, 16, 32]`
K	`[1, 3, 5]`
P_F	`[1, 2, 4, 8, 16]`
P_C	`[1, 2, 4, 8, 16]`
P_K	`[1, 2]`

Table 4.7: Values of parameters for those designs in the dataset.

   This set of parameters includes different data types with different bit width, different kernel size, and different parallelise parameters. In total, there are $3 \times 3 \times 5^2 \times 2 = 450$ designs to be built, and in practice each build pass takes about 1 hour to complete. With 18 design builds running in parallel, the total time to generate a dataset will be $450/18 = 25$ hours and it is endurable.

2. *Training with cross-validation*: Once the dataset is ready, the linear model of LUT and FF can be trained with linear regression. Three linear models will be trained *separately* for each bit width value (8, 16, and 24), because the relationship between the bit width and the LUT or FF usage of a single block is more like a *piecewise function*, rather than a linear function, from the basic background knowledge of FPGA architecture. Thus, separately training three linear models will increase the general performance.

   In order to further enhance the performance of trained models, *cross-validation* is utilised to increase the accuracy and reduce overfitting. The dataset will be split into several folds and in each pass of cross-validation, 1 fold will be selected for testing and the rest will be used for training (*Leave-One-Out*). Cross-validation will stop until all folds have been tested and trained.

The final output of this step contains 3 models for each bit width value, and these models can predict LUT and FF usage of the standard convolution layer from a given list of 4 parameters (K, P_F, P_C, P_K). Table 4.8 and Figure 4.3 show these models. Note that values of coefficients scale by the bit width.

Bit width	C^{lut}	B^{lut}	C^{ff}	B^{ff}	Cross-validation Mean Accuracy
32	52.3	2.50	50.0	18.4	99.96%
16	26.0	12.6	25.6	14.1	99.99%
8	13.0	−3.1	13.6	5.0	99.98%

Table 4.8: Linear models and their hyper-parameters.

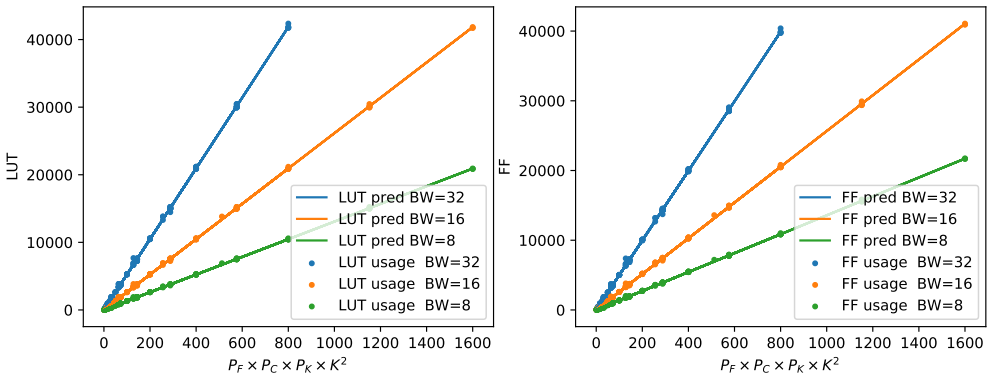

Figure 4.3: Linear model of LUT and FF usage in `Conv2DKernel`.

3. *Model evaluation*: Trained models will be evaluated for its accuracy on the test dataset split from the original dataset, and how large are the differences between resource usage of the standard convolution layer and the prediction result.

   Table 4.8 also shows the result of the first part of evaluation. Mean accuracy from cross-validation of all models are higher than 99.9%.

   Evaluation of the second part is listed in Section 4.3.

4.2.3 Performance

The overall performance of a single convolution layer is highly predictable from design parameters. According to the roofline model (Williams et al., 2009), the performance of a hardware design is bound by either computation speed or interface bandwidth.

Computation speed is the product of *throughput* and *clock speed*. Throughput is defined as the number of cycles required for a single convolution layer to accept next input feature map. Equation 4.1 already presents the throughput model. And clock speed is the reciprocal of clock frequency ($\mathcal{F}$). Thus, computation speed is the total time required to process a complete input feature map, which is summarised in Equation 4.13.

$$T_{conv}^{comp} = \frac{H \times W \times C \times F}{P_F \times P_C \times P_K} \times \frac{1}{\mathcal{F}} \tag{4.13}$$

The overall performance is also restricted by interface bandwidth. Equation 4.3 shows the maximal number of bits that should be transferred within one clock cycle If the interface bandwidth ($\mathcal{S}_{bd}$) cannot satisfy this requirement, then the time to process a single input feature map will be T_{conv}^{mem}, see Equation 4.14.

$$\begin{aligned} T_{conv}^{mem} &= \frac{H \times W \times C \times F}{P_F \times P_C \times P_K} \times Bd \times \frac{1}{\mathcal{S}_{bd}} \\ &= (H \times W \times C \times F)(\frac{1}{P_F} + \frac{K^2}{P_K} + \frac{1}{P_C}) \times \frac{Bw}{8\mathcal{S}_{bd}} \end{aligned} \tag{4.14}$$

According to the roofline model, the overall performance of processing one convolution layer (T_{conv}) is the minimal value of T_{conv}^{comp} and T_{conv}^{mem} (Equation 4.15) This equation shows that T_{conv} can be calculated by design parameters directly. Hyper-parameters $\mathcal{F}$ and $\mathcal{S}_{bd}$ can be set or checked by benchmarking.

$$\begin{aligned} T_{conv} &= \min\left(T_{conv}^{comp}, T_{conv}^{mem}\right) \\ &= (H \times W \times C \times F) \max\left(\frac{1}{P_F P_C P_K \mathcal{F}}, \left(\frac{1}{P_F} + \frac{K^2}{P_K} + \frac{1}{P_C}\right) \times \frac{Bw}{8 \times \mathcal{S}_{bd}}\right) \end{aligned} \tag{4.15}$$

The total number of operations to be performed is $H_O \times W_O \times C \times F \times K^2 \times 2$, then the overall performance measured in GOp/s is:

$$\begin{aligned} \mathcal{P}_{conv} &= \frac{H_O \times W_O \times C \times F \times K^2 \times 2}{T_{conv}} \\ &\approx \frac{H \times W \times C \times F \times K^2 \times S^2 \times 2}{T_{conv}} \\ &= 2K^2S^2 \times P_F P_C P_K \min\left(\mathcal{F}, \frac{8 \times \mathcal{S}_{bd}}{(P_K P_C + K^2 P_F P_C + P_F P_K)Bw}\right) \end{aligned} \tag{4.16}$$

Please see Section 4.3.1 for detailed evaluation of this performance model.

4.2.4 Binarised and Depthwise Separable Convolution Layer

Analysis models of binarised and depthwise separable convolution layer have been introduced in Section 4.1.2 and Section 4.1.3, and they are compatible with the general analysis model presented in this section: (1) BRAM and DSP usage can also be modelled by datasheets; (2) usage models for LUT and FF can also be trained by linear regression; (3) performance model stays the same.

The key difference between binarised and standard convolution layer in terms of analysis model is that there is no DSP usage in the binarised version. All arithmetic operations should be implemented in LUT and FF. Moreover, LUT and FF usage of binarised convolution layer are also linear to $P_F \times P_C \times P_K$. By running linear regression on a similar dataset as Section 4.2.2, we can get hyper-parameters for predicting LUT and FF usage.

Regarding depthwise separable convolution, the most significant and useful change in the analysis model is the DSP usage model. Previously in standard convolution, DSP usage of a given design follows $\mathcal{O}(P_F \times P_C \times P_K \times K^2)$, which scales really fast when the level of parallelisation increases. Now in depthwise separable convolution, DSP usage follows $\mathcal{O}(P_C \times P_K \times K^2 + P_F \times P_C \times P_K)$.

Figure 4.4 illustrates the difference between these two models. Thus, it is highly possible that a depthwise separable convolution layer can achieve higher performance by placing more computation units on board. Note that LUT and FF usage will not be the bottleneck in this case.

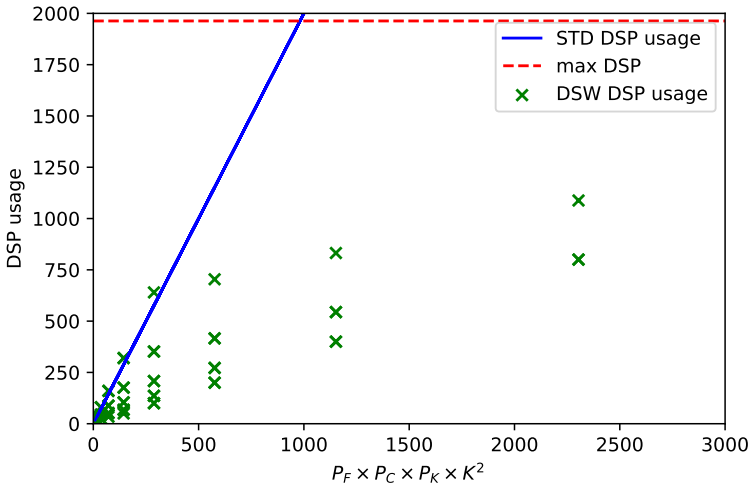

Figure 4.4: DSP usage of both standard convolution (STD) and depthwise separable convolution (DWS).

4.2.5 Multiple Layers Analysis Model

The multiple layers analysis model is designed for analysing full CNN design in MaxDeep. Because convolution layers consume most of the resources on board, this multiple layer analysis model only covers convolution layers in the given CNN topology. Most of the previously discussed models are reused in this section, and only important differences are discussed.

The resource usage model of a multiple layers design is a summation of results from single layer models applied to all convolution layers, for example, Equation 4.17 shows the LUT usage of the whole network with N convolution layers.

$$U^{lut} = \sum_{i=1}^{N} C^{lut}(Bw^{(i)}) \times P_F^{(i)} \times P_C^{(i)} \times P_K^{(i)} + B^{lut}(Bw^{(i)}) \tag{4.17}$$

Equation 4.18 shows another example, the DSP usage of a CNN, which is still the resource bottleneck in the current case.

$$U^{dsp} = \sum_{i=1}^{N} P_F^{(i)} \times P_C^{(i)} \times P_K^{(i)} \times \left(K^{(i)}\right)^2 \times U_{mul}^{dsp}(Bw^{(i)}) \tag{4.18}$$

As mentioned in Equation 4.10, to make sure that all layers are processing at the same rate, there are constraints on parallel parameters and many of them can be reduced. For a network with N layers, total number of parallel parameters required is $N + 2$: 1 for P_K, 1 for P_C, and N for P_F. Also, assume all layers use the same bit width Bw. Then Equation 4.18 can be updated in a form as the following equation:

$$U^{dsp} = P_K \times \left(P_C P_F^{(1)} (K^{(1)})^2 + \sum_{i=2}^{N} P_F^{(i-1)} P_F^{(i)} P_K (K^{(i)})^2 \right) \times U_{mul}^{dsp}(Bw^{(i)}) \tag{4.19}$$

Besides, the performance model can also be derived from the one from the single layer model. Suppose there are N convolution layers, the performance bound by *computation* is Equation 4.20.

$$\mathcal{P}^{comp} = \frac{2 \times N_b \times \sum_{i=1}^{N} H_O^{(i)} W_O^{(i)} C^{(i)} F^{(i)} (K^{(i)})^2}{N_b \frac{H^{(1)} W^{(1)} C^{(1)} F^{(1)}}{P_K P_C P_F^{(1)}} + \sum_{i=2}^{N} \frac{H^{(i)} W^{(i)} C^{(i)} F^{(i)}}{P_K P_F^{(i-1)} P_F^{(i)}}} \times \frac{1}{\mathcal{F}} \tag{4.20}$$

In this equation, the batch size N_b is explicitly included, and obviously with larger batch size, the expected performance will be better. And when the N_b is really large, the overall performance can be at most N times of the maximal performance of one single layer. As there are many streams included in a multiple layers design and the bandwidth-based model is much complex than single layer, only the computation bound case will be used for optimisation and prediction.

4.2.6 Summary

The general analysis model is summarised in Table 4.9.

Resource	Usage
LUT	$U_{conv}^{lut} = C^{lut}(Bw) \times P_F \times P_C \times P_K \times K^2 + B^{lut}(Bw)$
Flip-Flop	$U_{conv}^{ff} = C^{ff}(Bw) \times P_F \times P_C \times P_K \times K^2 + B^{ff}(Bw)$
BRAM	$U_{conv}^{bram} = U_{ibuf}^{bram} + U_{lbuf}^{bram} + U_{obuf}^{bram}$
DSP	$U_{conv}^{dsp} = P_F \times P_C \times P_K \times K^2 \times U_{mul}^{dsp}(Bw)$
Perf	$\mathcal{P}_{conv} = 2K^2 S^2 P_F P_C P_K \min\left(\mathcal{F}, \frac{8 \times \mathcal{S}_{bd}}{(P_K P_C + K^2 P_F P_C + P_F P_K) Bw}\right)$

Table 4.9: Summary of the analysis model

4.3 Evaluation

This section presents evaluation results of the MaxDeep framework. The major purpose of evaluation is to find out whether the *resource usage* and *performance* of real hardware builds match our expectation. MaxDeep is first evaluated on *single convolution layer* in several different aspects (Section 4.3.1), including resource usage of designs with different parameter values, the difference between model prediction and resource usage of real builds, and comparison among different configuration that can shed a light on what a good performing optimisation technique should be under some situations. Moreover, an evaluation of two adjacent convolution layers is illustrated with detailed experimental data. A manually constructed small but complete CNN, LeNet-5, is evaluated both in resource usage and performance at the end of Chapter 5.

Table 4.10 summarises properties of our evaluation system.

Metric	
CPU	Dual Intel Xeon E5-2640, 6 cores per CPU
FPGA	Stratix V 5SGSMD8N1F45C2
FPGA DRAM	48 GB
CPU to FPGA Bandwidth	38 GB/s
LUT	262400
FF	524800
BRAM	2567
DSP	1963

Table 4.10: System Properties of the Maxeler MAX4 Maia

4.3.1 Single Convolution Layer Evaluation

A single convolution layer with different parameter values are first evaluated, to make sure that the analysis model for design metrics has limited deviation from real hardware designs. To avoid combinatorial explosion of possible parameter values and reduce building time as little as possible, the evaluation in this section has the following settings: (1) height, width, number of channels, and number of filters are all 32; (2) kernel size is 3; (3) interface with CPU is PCIe rather than DRAM with memory controller; (4) P_K takes only 1 and 2, and both of P_C and P_K can take a value in $\{1, 2, 4, 8\}$; (5) bit width BW can choose from $\{1, 16, 32\}$; (6) only filter major sequence is considered in this section.

General Evaluation

Figure 4.5 shows the resource usage of the designs generated from a set of parameters following the settings above. This figure discovers that all four types of resource have their usage scaled linearly. To be specific, LUT, FF, and DSP usage scale by $P_F \times P_C \times P_K$ and BRAM scales by P_C, which are compatible with the analysis model. Note that the BRAM usage seems also scaled by P_K and there is deviation in the data collected, this circumstance is due to the memory block tiling while the FPGA programming tool synthesizing the design, and this error is currently ignored.

Figure 4.6 evaluates the prediction accuracy of analysis models upon the build dataset. In general, prediction of all resource types are quite close to the real build data. Table 4.11 lists the prediction error of each resource type.

LUT and FF usage prediction are all below the real usage, because only the usage in `Conv2DKernel` are considered in prediction. DSP usage is almost the same as real DSP usage, and the deviation is caused by computation logic in output feature map buffer for address calculation. BRAM usage is much complex due to tiling. Original prediction model in Equation 4.11 doesn't consider tiling and can cause large deviation (about 22%). Thus, by assuming the depth of each tile is 1024 and considering tiling while computing BRAM usage, the mean error of BRAM prediction can be reduced to 5.1%. However, to keep the analysis model simple and clean, Equation 4.11 stays unchanged.

Regarding performance, Figure 4.7 illustrates evaluated performance, predicted performance by computation speed and by interface bandwidth. Result shows that when the variable $P_F \times P_C \times P_K$ is small, the performance is bound by computation. Then as the variable increases, the performance is bound by bandwidth. The prediction results are calculated by replacing $\mathcal{F}$ with 100×10^{-3} (giga-cycles per second) and $\mathcal{S}_{bd}$ with 3.2 (GB/s). The bandwidth value is decided from the PCIe bus (x8 gen2) metrics. This discovery fits what the roofline model expects. Because the performance model

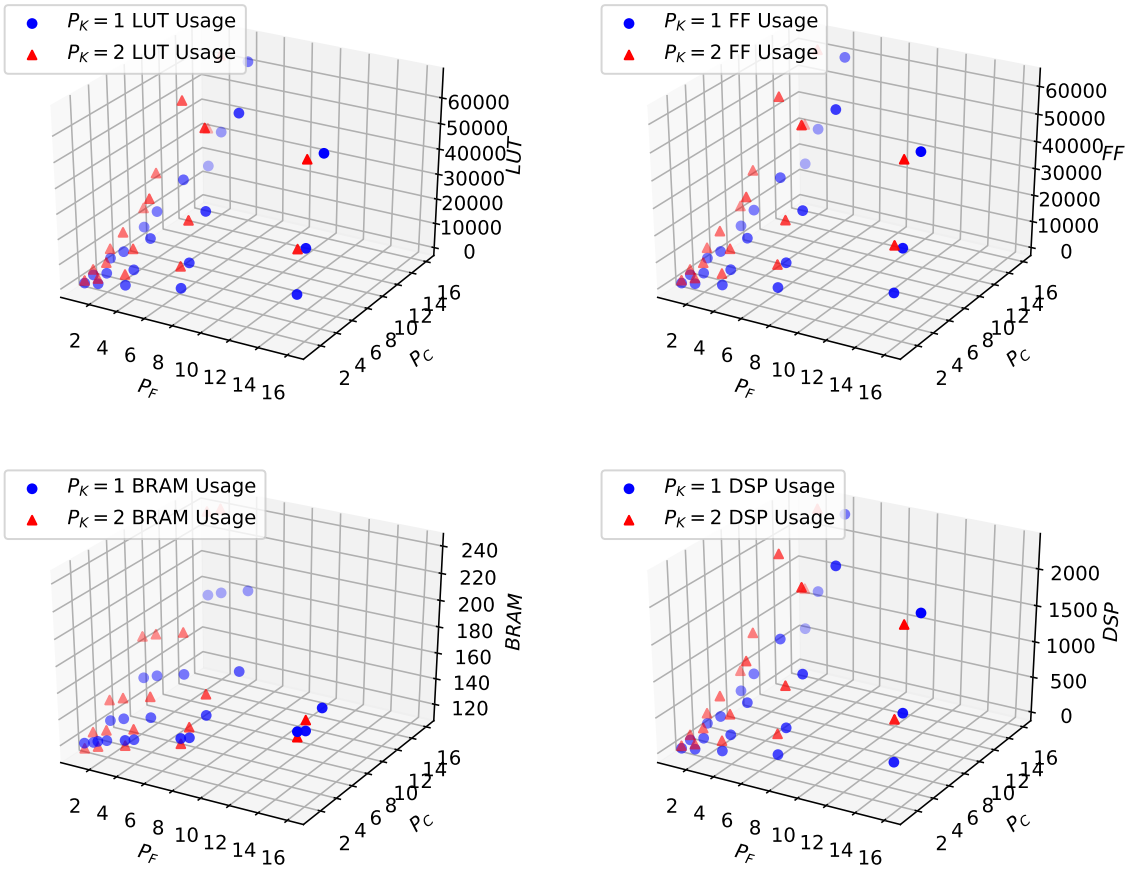

Figure 4.5: Evaluation of a single convolution layer with 32 bit fixed-point data type and filter major computation sequence.

by bandwidth is coarse and contains approximate assumptions, the overall performance predicted has an error rate of 17.9% (variance 0.027).

High-Performance Scenario Evaluation

This section presents the evaluation of a *high-performance scenario* for single convolution layer. This scenario aims at exploring the design configuration that can achieve the best performance. Designs in this scenario have the following features: (1) use DRAM-FPGA interface rather than PCIe to achieve higher bandwidth; (2) select larger *batch size* — the number of input feature maps to be processed in one pass — to cover initialisation latency and overhead; (3) consume most of the resources on board to increase level of parallelisation.

The analysis model, which is already evaluated in the general evaluation section, is used as the core component of the optimisation process in the scenario. And the optimisation process will examine all possible parameter combinations, which are vectors of $\langle Bw, P_F, P_C, P_K \rangle$, to select the combination with best performance.

However, the performance model can be much simpler in this scenario. Because the DRAM to FPGA interface has very high bandwidth (38.2 GB/s) and each transfer burst in each cycle has quite large size (384 B), at most $384 \times 8/32 = 96$ number of elements can be transferred without delaying the

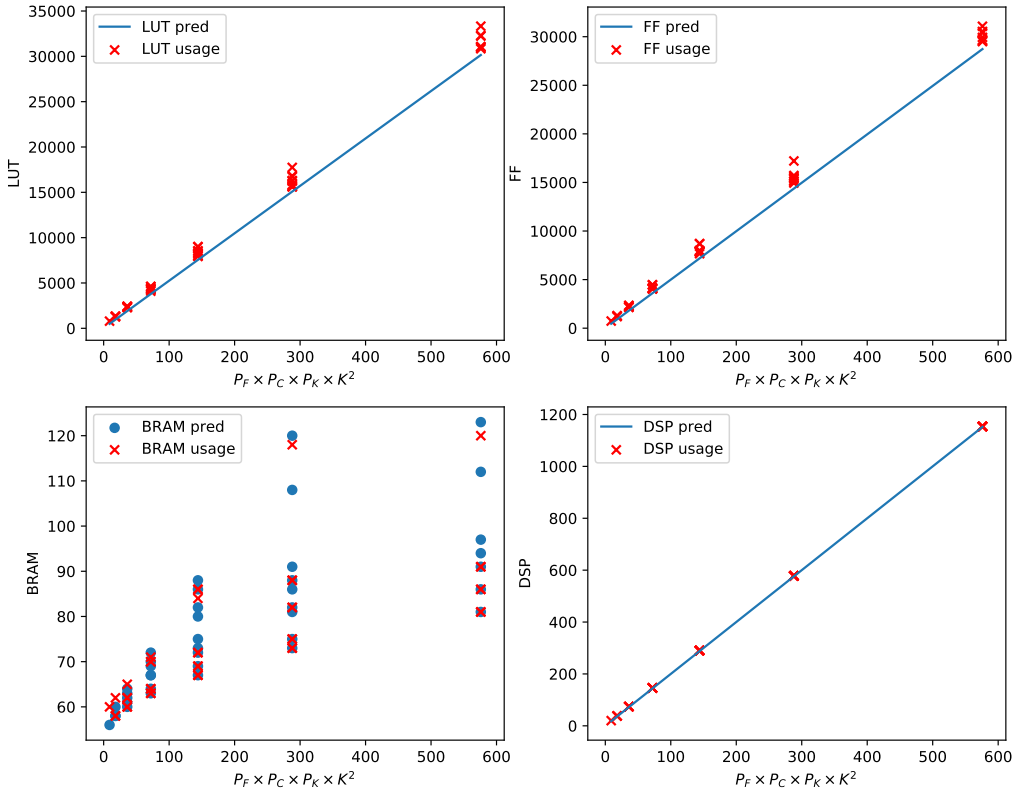

Figure 4.6: Evaluation the resource usage prediction of a single convolution layer.

Resource type	Error Mean	Error Variance
LUT	12.1%	0.006
FF	12.4%	0.006
BRAM	5.1%	0.006
DSP	1.4%	0.000

Table 4.11: Prediction error

computation process. Thus, in this scenario, the performance model can be simplified to $\mathcal{P}^{comp}$.

Figure 4.8 shows performance evaluation of valid builds for a convolution layer with shape ($32 \times 32 \times 32 \times 32$) and kernel size $K = 3$. Filter major sequence is used, and the data type is set to 32bit fixed-point. In this case, the best parameter vector is $\langle P_F, P_C, P_K \rangle = \langle 16, 4, 1 \rangle$, and the expected performance is **115.2** GOP/s. This result is close to the value found by running the build in hardware, which is **113.7** GOp/s.

Is there any chance to further increase the performance? There are two approaches: reduce precision and increase clock frequency. After examining the resource usage, it is discovered that the maximal performance is bound by DSP capacity, and change the bit width from 32 to 16 can enhance the performance by a factor of 2. Although reducing the precision to 8 has no effect on DSP usage per arithmetic unit, because both 16bit and 8bit fixed-point multipliers use 1 DSP, the total DSP usage can be reduced due to resource tiling and alignment. In this case, the best parameter vector is $\langle Bw, P_F, P_C, P_K \rangle = \langle 8, 16, 8, 1 \rangle$, and the expected performance is **230.4** GOP/s. Increase the clock frequency can also enhance the performance. If the frequency is changed from $100MHz$ to

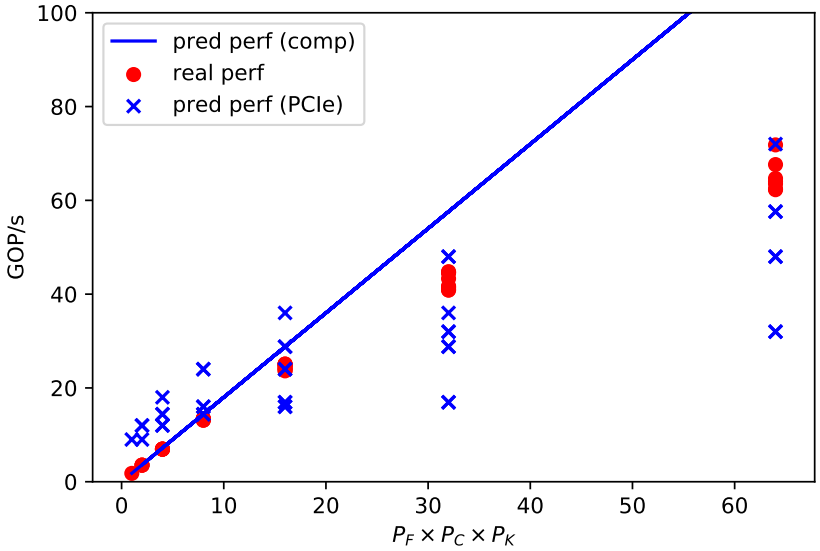

Figure 4.7: Performance evaluation of a single convolution layer.

$150MHz$, then the overall performance will become $230.4 \times 1.5 = \mathbf{345.6}$ GOp/s. An experiment with a design $\langle Bw, P_F, P_C, P_K, \mathcal{F} \rangle = \langle 8, 8, 8, 2, 150 \rangle$ shows that the result performance is **343.6** GOp/s.

Finally, we can push the clock frequency to 200 MHz. This design is named as `CNV-BEST` and its performance is further compared with existing publications. The two publications selected present designs that are state-of-the-art in 2015 and 2016 respectively. Note that it is reasonable to choose 8-bit fixed point as the target data type. We will show in Section 5.3 that 8-bit will have similar accuracy as other longer data types. The result shows that our design is better in terms of resource efficiency and only a bit less power efficient than Qiu et al. (2016), which might due to the difference in FPGA platforms: Zynq is targeting embedded devices and is more energy efficient than Stratix V.

	Zhang et al. (2015)	**Qiu et al. (2016)**	`CNV-BEST`
FPGA	Virtex VX485T	Zynq XC7045	Stratix V 5SGSDB
Technology	28 nm	28 nm	28 nm
Data Type	32-bit float	16-bit fixed	8-bit fixed
Freq. (MHz)	100	150	200
Power (W)	18.61	9.63	25.3
Perf. (GOp/s)	61.62	187.8	453.3
Resource Efficiency (GOp/s/Slice)	0.81×10^{-3}	3.58×10^{-3}	6.91×10^{-3}
Power Efficiency (GOp/s/W)	3.31	19.50	17.91

Table 4.12: Evaluation results of high-performance single convolution layer generated by MaxDeep, compared with two previous publications.

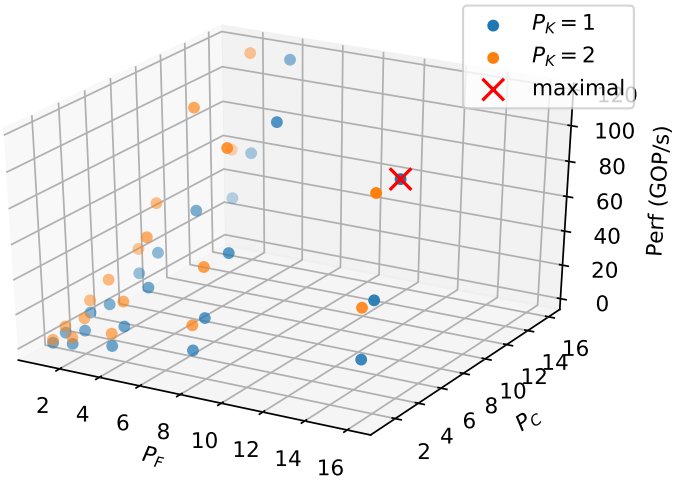

Figure 4.8: Evaluation of the high-performance scenario.

Multi-Pumping

In a previous version of MaxDeep, which was developed in the 2nd MRes project, *multi-pumping* is supported to enhance the performance of the convolution layer. Multi-pumping is a hardware resource saving technique that runs parts of a design at a integer multiple of the global clock frequency. Suppose the integer multiple is M, then the resource can be saved by $1/M$ at most. This technique is suitable for enhancing the performance of CNN designs, especially the convolution layer, because the major resource bottleneck for performance scaling of the convolution layer is DSP, which has quite high maximal frequency and the value of M can also be large. Thus, multi-pumping can help the convolution layer consume less DSP blocks per unit and further increase the level of parallelisation.

Although this technique is promising, the current MaxDeep version doesn't support multi-pumping, for now, because most of previous codes have been rewritten. However, we can still predict the performance enhancement by applying multi-pumping on `CNV-BEST`. The current `CNV-BEST` uses 57.16% of total DSP resources and the global clock frequency is 200 MHz. According to the datasheet of Stratix V, the maximum frequency of a DSP block is 500 MHz, which means that M, as defined above, could reach 2. In this case, we can place double amount of processing units in the convolution layer, which is the `Conv2DKernel`, and thus the total performance can also double, which becomes **906.6** GOp/s at the ideal circumstance.

There are some issues that might arise and prevent us getting this performance, such as bandwidth limitation of the memory and failed timing while the FPGA tool running Place and Route. In the 2nd MRes project, we have proven that multi-pumping works in enhancing the performance of convolution layers, but because the design is different between two versions, we need to further verify the conclusion on the current version. Even though, it is still a promising technique that worth trying in the future.

4.3.2 Binarised and Depthwise Separable Convolution

The purpose of this section is to show that binarised convolution layer and depthwise separable layer can achieve higher theoretical peak performance than a standard design.

Binarised Convolution Layer

We evaluate the resource usage of the binarised convolution layer at first, and then show a comparison between the best binarised convolution layer design we can build with the state-of-the-art design from a previous publication (Zhao et al., 2017).

Regarding binarised convolution layer, its performance is mainly bound by LUT and FF, and no DSP usage is involved. Figure shows the resource usage from a set of real builds similar to the ones presented in Section 4.3.1, except that the bit width of each build is 1. Figure 4.9 shows the result of resource usage prediction through linear models trained in a way similar to Section 4.2.2. Results show that accuracy of both models are above 96.6%. Assuming the performance of binarised convolution layer is only bound by *computation*, which is listed in Equation 4.13, then it is feasible to find the optimal set of parameters $\langle P_F, P_C, P_K \rangle$ through constrained optimisation. The objective of the optimisation is performance and the constraints are LUT and FF usage.

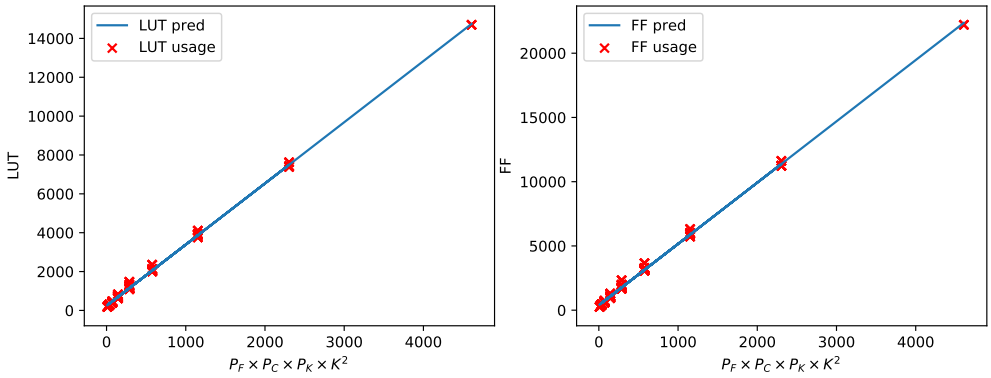

Figure 4.9: Evaluation of LUT and FF usage in binarised convolution layer

After running through all valid parameter combinations, it is discovered that $\langle 64, 64, 2 \rangle$ can achieve the best performance of a single binarised convolution layer. The predicted performance is **14.7** TOp/s. However, it is really hard for the FPGA toolchain to build a design as large as this one. In the end, the best binarised convolution layer from MaxDeep is named as `BCV-BEST`, which has $\langle P_F, P_C, P_K, \mathcal{F} \rangle = \langle 32, 32, 2, 150 \rangle$.

Table 4.13 shows a comparison between `BCV-BEST` and one of the state-of-the-art binarised CNN publication (Zhao et al., 2017). Our design is better in speed but the power consumption is higher. It is possible due to the difference between the FPGA platforms. Also, our design is better in both power efficiency and resource efficiency.

Depthwise Separable Convolution Layer

In terms of depthwise separable convolution layer, its performance is still bound by DSP, which usage can be calculated by Equation 4.21. Following similar approach to binarised convolution, the set of

	Zhao et al. (2017)	BCV-BEST	Comparison
FPGA	Zynq 7Z020	Stratix V 5SGSDB	
Num. of LUT	46900	106034	2.26
FREQ	143 MHz	150 MHz	1.04
Speed (GOp/s)	318.9	1166.12	**3.65**
Power (W)	4.7	13.1	2.78
Eff. (GOp/s/W)	44.2	89.1	**2.01**
Eff. (GOp/s/kLUT)	4.43	10.99	**2.48**

Table 4.13: Performance comparison with a previous binarised CNN publication.

parameters with the best performance can be found, which are $\langle 16, 16, 2\rangle$ and the performance is 115.31 GOp/s as predicted. Note that this number is derived from a different performance formula rather than the one for standard convolution, because the number of operations is greatly reduced in this case. It is not surprising to see the GOp/s number of the best depthwise separable convolution layer less than the one of the best standard convolution layer.

$$U_{dws}^{dsp} = (P_C \times P_K \times K^2 + P_C \times P_F \times P_K) \times U_{mul}^{dsp}(Bw) \tag{4.21}$$

Based on this prediction, we build a design (DWS-BEST) with parameters above and report its performance and resource usage in Table 4.14. Note that we use 16 bit fixed-point data type here, because the FPGA build tool cannot handle very wide FIFO interface [2]. We also compare the result of DWS-BEST with CNV-BEST. It shows that the processing rate of the best depthwise separable design is 2.61 times faster than the standard convolution layer. Because the depthwise separable convolution has smaller input feature map buffer, the BRAM usage of DWS-BEST is also smaller. The increase of LUT, FF, and DSP usage is due to higher level of parallelisation in DWS-BEST.

	DWS-BEST	CNV-BEST
$\langle Bw, P_F, P_C, P_K\rangle$	$\langle 16, 16, 16, 2\rangle$	$\langle 16, 8, 8, 2\rangle$
FREQ (MHz)	150	200
LUT	26.52%	25.67%
FF	24.23%	23.35%
BRAM	40.96%	58.79%
DSP	49.20%	33.93%
Time per frame (μs)	21.1	41.6
Power (W)	26.9	25.3

Table 4.14: Report of resource usage and performance of depthwise separable convolution, compared with CNV-BEST.

[2] In our OpenSPL code we simply assign $P_F \times P_C \times K^2$ as the number of elements of the FIFO interface for coefficients input of the depthwise convolution. Because we can achieve quite high level of parallelisation here, the width will become approximately 73728 b if we use 32 bit data type, which is of no chance feasible to be built on real hardware. However, we can reduce the width and read multiple cycles to fill in the coefficient registers to workaround this issue. We will not cover this fix for now and evaluate this case in the future.

4.3.3 Two Convolution Layers Evaluation

In this case, Equation 4.20 is used to predict performance and Equation 4.18 is applied to predict DSP usage. For a design that runs two convolution layers in pipeline, there are 4 parameters to be optimised: $\langle P_F^{(1)}, P_F^{(2)}, P_C, P_K \rangle$. By searching all possible solutions, the best performing set of parameters is $\langle 4, 4, 8, 2 \rangle$ and the predicted performance is **189**.**12** GOp/s. However, according to the real performance on hardware, which is **107**.**9** GOp/s, we can find out that when the number of layers is larger than 1, the bandwidth will become an issue that cannot be ignored. In this case, the bandwidth required is 1824 B per cycle, which is much larger than the burst size (384 B).

4.4 Summary

In this chapter, we present MaxDeep, an OpenSPL-based CNN hardware library on the Maxeler platform. MaxDeep contains essential parameterised building blocks to construct most CNN models. These blocks, unlike relations in RubyConv, can be optimised with novel and practical strategies, such as depthwise convolution, binarised convolution, and parameter tuning in level of parallelisation and computation sequence. The resource usage and performance of these blocks can be predicted through integrated analysis models. Analysis models for MaxDeep is devised with reasonable assumptions and statistical methods, and their accuracies are guaranteed by evaluation results on real hardware builds. Based on these models, we also provide a flow to find the best set of parameters for given CNN models by searching for the solution under a constrained optimisation problem. Our evaluation results show that the performance of designs built by MaxDeep is as what we expect.

Chapter 5

Plumber and Model-Hardware Co-Optimisation

Plumber is a special *transpiler* that converts high-level CNN model to low-level hardware design. The aim of plumber is to accelerate the migration process from a newly designed and trained CNN model to a optimised and runnable hardware design, which is indeed tedious at the moment. Plumber is targeting at providing compatibility with most CNN models developed by TensorFlow (Abadi et al., 2016) or Caffe (Jia et al., 2014) with little pre-requisites, and porting models to different FPGA platforms under various scenarios. Till now, the Plumber transpiler supports basic layers in CNN, such as convolution layer and fully-connected layer. Also, it can transpile depthwise separable convolution to the optimised hardware block in MaxDeep. Besides, the target hardware platform is limited to the Maxeler platform with MaxDeep as the hardware library.

Besides, Plumber is integrated with a *model-hardware co-optimisation* module. This module not only optimises hardware design parameters based on the given network model, but also changes the network model itself. Thus, the optimisation problem has a joint objective, which is a score that combines hardware performance, power consumption, and model accuracy as a weighted sum function. Currently, model optimisation focuses on several high-level options, such as quantisation and convolution layer replacement.

This chapter first describes the concept of *dataflow graph* (Section 5.1), which serves as Intermediate Representation (IR) that can be compiled from high-level CNN models (frontend), and is a single source to different compilation processes that target different FPGA platforms and scenarios (backend). Then, this chapter presents details about the frontend process (Section 5.2.1) and the backend process (Section 5.2.2). Section 5.2.1 also introduces high-level model optimisation strategies supported in Plumber at the moment. Section 5.2.3 discusses the co-optimisation module.

At last, we provide two evaluations of Plumber based on LeNet-5 (Section 5.3):

1. We first compare (Table 5.2) a *baseline* design (`STD-BASE`) of LeNet-5, which is generated directly from MaxDeep with sufficient hardware-level optimisation, with the *best* design (`Q8b-BEST`) achieved after model-hardware co-optimisation. Result shows that `Q8b-BEST` is 1.59 times faster than the baseline and its power consumption is 0.81 times less.

2. We then compare the performance of Plumber with TensorFlow on CPU and GPU, see Table 5.3 for comparison details. In general, `Q8b-BEST` can achieve the best performance for 8bit quantised LeNet-5.

For now, we cannot make a comparison with previous publications in the field of CNN framework on FPGA platforms because they are using network models different from us. In the future we will support those models and make further comparison.

Plumber is implemented in Python with the latest version 3.6.

5.1 Dataflow Graph

Dataflow graph is a type of directed graph that represents dataflow in a specific computation. Each node of the graph represents an operation and each edge represents a flow of data. In Plumber, nodes of a dataflow graph can either be implemented in hardware or software, and its edges are on-chip streams or streams between CPU and FPGA.

As mentioned before, dataflow graph in Plumber is actually an IR of the transpiler, which is necessary for compatibility and furture optimisation. For example, although TensorFlow natively uses dataflow graphs to represent high-level CNN models, operations of nodes in TensorFlow dataflow graph are neither implementable on hardware nor can be replaced by operations that are easier to be implemented. Also, models sources could come from other deep learning frameworks, such as Caffe and Keras, which define CNN models in different ways. Thus, first transforming them into a unified dataflow graph IR and then optimising it is a more efficient process.

This section describes the core concepts of Plumber dataflow graph IR, which include properties of graph nodes and edges (Section 5.1.1), and how it is implemented in Plumber (Section 5.1.2). Frontend and backend process of Plumber that respectively creates and utilises this dataflow graph IR will be discussed in next sections.

5.1.1 Nodes and Edges

Nodes represent computational operations. Each node has three major properties:

1. *Type* specifies which operation the current node will run. Plumber currently supports 3 types of nodes: `CONV2D` for convolution layer, `MATMUL` for matrix multiplication (fully-connected layer), and `DECONV2D` for transposed convolution layer. Note that `DECONV2D` is not yet supported in MaxDeep.

2. *Device* points where the operation will run on (CPU or FPGA). If the device is CPU, then the operation will be executed by a software function. If the device is specified as FPGA, then additional device information is required, such as the index of the FPGA board and the index of the hardware block instance (there might be several instances for the same functionality).

3. *Edges* list input/output connections of the node. Each edge represents a data transfer from one node to another, and it records the amount of data to be transferred and the source and destination address of the transfer if off-chip memory is involved.

Based on nodes and edges, it is possible to construct how a complete CNN model will be processed in the MaxDeep framework.

5.1.2 Implementation

Plumber implements its dataflow graph IR in Google Protobuf [1], because its portability to many different languages and interfaces. The code snippet below shows an example dataflow graph generated from the LeNet-5 network model. Note that essential information are attached to each node, such as shape, data type, and whether to use pooling and ReLU. Also, Plumber provides convenient library to operate the dataflow graph, which enables further optimisation algorithm development.

```
nodes {
    name: "conv1"
    op: "CONV"
    data_type {
        bit_width: 32
    }
    conv2d_def {
        height: 28
        width: 28
        in_channels: 1
        out_channels: 32
        kernel_height: 5
        kernel_width: 5
        num_channels: 1
        num_filters: 32
        kernel_size: 5
        has_2x2_maxpool: true
        has_relu: true
    }
}
...
nodes {
    name: "fc1"
    op: "FC"
    type: MATMUL
    data_type {
        bit_width: 32
    }
    matmul_def {
        num_rows: 1024
        num_cols: 3136
        has_relu: true
    }
}
...
```

5.2 Plumber Transpiler

The system architecture of Plumber is illustrated in Figure 5.1. There are three major passes: *model optimisation* (pruning and quantising), *model parsing*, and *backend synthesising*. Note that parsing Caffe models is not supported at the moment.

[1] https://github.com/google/protobuf

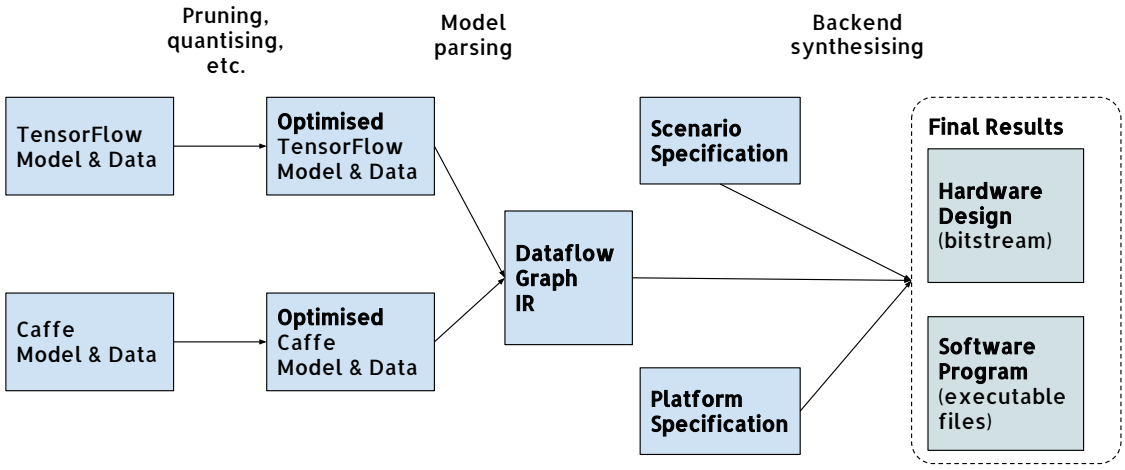

Figure 5.1: System architecture of Plumber.

5.2.1 Frontend: Model Parser and Optimisation

The model parser transforms raw TensorFlow model data, which is object of class `GraphDef`, into the dataflow graph IR. Its main process has two steps: run TensorFlow built-in utilities and then run specifically defined functions.

TensorFlow built-in utilities are defined in the `graph_utils_impl` package, and we choose to use the following functions in sequence: `convert_variables_to_constants` *freezes* the network coefficients into the model definition, and `remove_training_nodes` drops training nodes (such as gradient calculation nodes) that are useless while inferencing. Besides, if selected, Plumber will transform specific types of nodes into their quantized version. Quantized nodes use 8bit or 16bit integer arithmetic operations rather than floating point, which can reduce the model size and allow better hardware optimisation. Software implementation of quantized nodes are partially implemented by TensorFlow — only a subset of all node types are supported, such as `QuantizedConv2D` and `QuantizedMatMul`, and they are all implemented in MaxDeep.

We provide two additional utility functions, mainly for building and optimising the dataflow graph IR. The first one is called `convert_to_separable`, which can convert a standard convolution layer node to a depthwise separable convolution node. This function is the key to model-hardware co-optimisation (Section 5.2.3).

Another function is named as `filter_and_group_nodes`, which filters out nodes that will not be deployed on hardware (like a simple addition operation), and group nodes into the 3 supported node types of dataflow graph. For example, if a convolution node is followed by max pooling and relu, then the dataflow graph node generated will has its `has_2x2_max_pool` and `has_relu` fields have **true** value assigned.

5.2.2 Backend: Software Code and Hardware Design Generation

In general, both software code and hardware design code are generated through *templates*.

Regarding software code generation, the template contains control logic to interactively walk through

all nodes in the dataflow graph and generate corresponding function calls. Besides function call generation, the software code generator also organises the memory space of the DRAM connected directly to FPGA. Also, there will be a script file to link and compile the software side of the design.

Hardware code generation is more complex. There are two steps: search for the best set of design parameters and generate hardware based on it. The first step is also known as *design space exploration*, which mainly solves a constrained optimisation problem, similar to the one introduced in MaxDeep evaluation (Section 4.3). The objective of the optimisation is to maximise the performance. Constraints include resource usage and power consumption. According to discoveries in Section 4.3, the design space for exploration is quite small and the performance function is easy to evaluate, thus in Plumber we still use a brute-force solver.

Once the design parameters are settled, they can be further used in the second step to generate hardware design code. In MaxDeep, hardware design codes are written in OpenSPL. And because all kernel codes are parameterised, only a template file for specifying hardware manager is required. This template file will also walk through all nodes in the dataflow graph and connects them through streams.

All the templates and related rendering mechanics are provided by the *Jinja library*[2].

5.2.3 Model-Hardware Co-Optimisation Module

Alongside with the transpiler, there is an integrated module in Plumber called *Model-Hardware Co-Optimisation* (MHCO). Once the original model has been transpiled to hardware design and successfully deployed, this module will first evaluate the performance and accuracy of the generated hardware accelerator, and then give further optimisation directions. Figure 5.2 illustrates how this module and the transpiler work together: the MHCO module accepts *feedback* information from the built hardware, such as whether the design can be built or not and whether the performance is satisfiable.

Based on the feedback information, MHCO can make the following decisions for now: replace standard convolution layers to depthwise separable convolution layers, or quantize the data type from 32bit into 8bit or 16bit. These two functions are already implemented in the Plumber frontend and can be called while optimising the original model. After the decision has been made, the model will be retrained based on the new topology or precision. The output of the MHCO module will then become a retrained optimised model. Although it is possible to replace the type of convolution layer or change the precision in a layer by layer approach, MHCO chooses to perform one selected optimisation strategy on all layers at once, to reduce the complexity of the whole optimisation flow.

Note that only the layer replacement optimisation includes retraining, the data quantisation optimisation only converts the trained weights into specified data type before inference.

5.3 Evaluation

Evaluation of Plumber focuses on one problem: could the co-optimisation module really enhance the performance without hurting the model accuracy? To answer this question, we select datasets, MNIST, which is designed for hand-writing recognition, and three networks, *standard* implementation (STD), *quantised* to 8 bit (Q8b), and *depthwise separable* replaced implementation (DWS).

[2] `http://jinja.pocoo.org/`

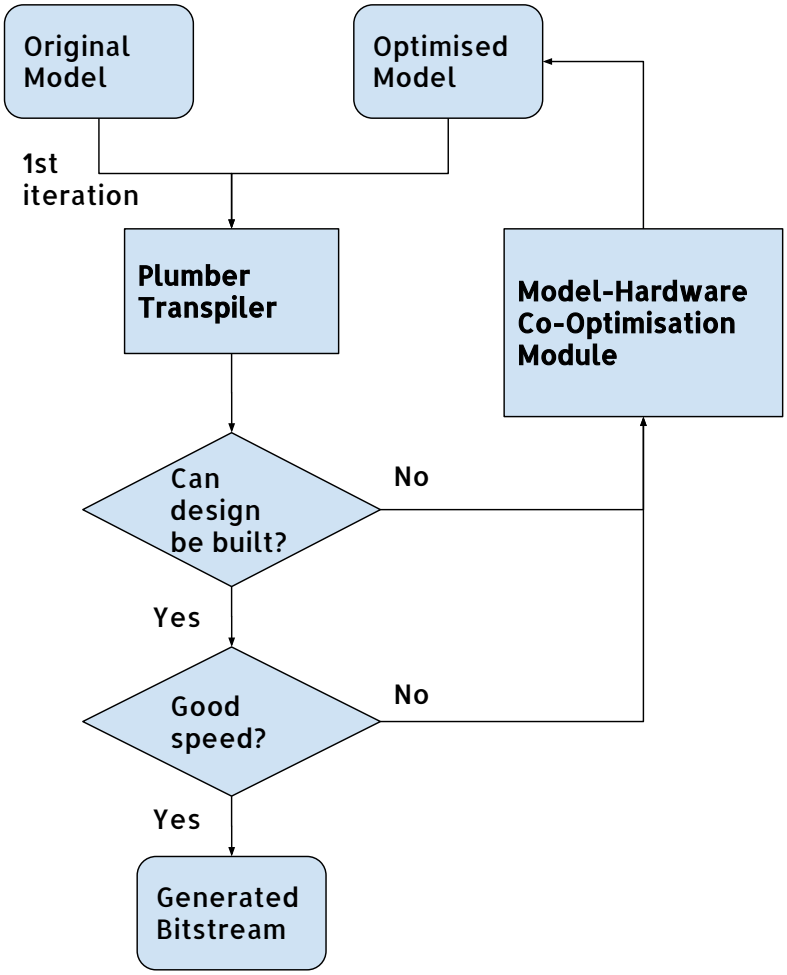

Figure 5.2: Working flow of the Plumber transpiler and the model-hardware optimisation module.

The standard implementation is derived directly from the LeNet-5 topology[3]. As mentioned before, retraining is not available for quantized networks, because TensorFlow has not yet supported this feature. Also, based on a similar reason, it is not possible to combine depthwise separable layers with quantisation. Thus, these three networks are what we can experiment with for now. Figure 5.3 shows the relationship among all three types of networks.

The first step of evaluation is training. STD and DWS can be trained with Adam optimiser (Kingma and Ba, 2014) on the MNIST dataset until good accuracy is achieved. And then, through TensorFlow built-in *graph transform* tool [4], a quantized model of the standard convolution can be created. Table 5.1 shows the accuracy of all trained networks and their model size. Note that on MNIST, quantised LeNet-5 model can achieve the best accuracy with the smallest model.

Next, we deploy all three networks on real hardware, by passing their model files through the Plumber transpiler and building the generated hardware design source codes. The Plumber transpiler can recognise those optimisation options in raw TensorFlow models and annotate the dataflow IR with these options. Then, the MaxDeep backend of the transpiler can process the dataflow IR and have

[3] `http://yann.lecun.com/exdb/lenet/`
[4] TensorFlow quantisation tutorial: `https://www.tensorflow.org/performance/quantization`

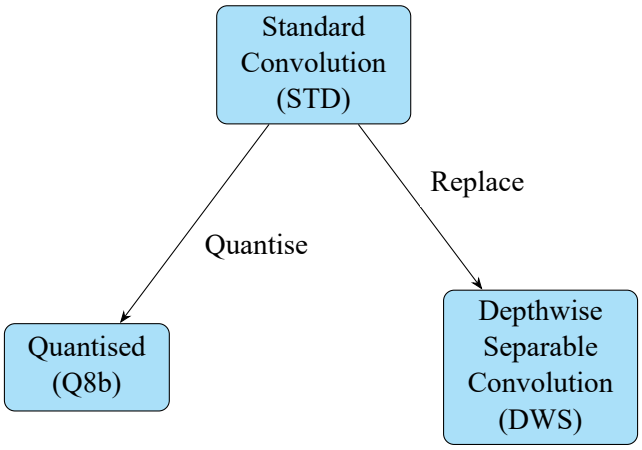

Figure 5.3: Relationship between four evaluation networks.

Network	Accuracy	Model Size
STD	99.21%	13.1 Mb
DWS	98.76%	12.9 Mb
Q8b	99.25%	3.2 Mb

Table 5.1: Evaluation results of three networks on MNIST for their accuracy and model size.

all those optimisation options implemented in the generated hardware code. The optimiser of the Plumber transpiler backend first discovers that there are *four* tunable parameters: $P_K^{(1)}$, $P_F^{(1)}$, $P_R^{(3)}$, $P_R^{(4)}$ in the generated LeNet design, and then locate the best set of parameters by solving a constrained optimisation problem. Details about this optimisation process are introduced in Section 4.3.

Results show that a **Q8b** network model with optimised design parameters is the **best** case for LeNet-5 hardware deployment. We name this design `Q8b-BEST`. To explicitly show the performance enhancement, we also create another design built directly from MaxDeep, without the co-optimisation process in Plumber. This design is named as `STD-BASE`, it targets 32 bit data type and the standard LeNet-5 network without layer replacement or quantisation.

Table 5.2 summarises statistics of the best built network and the base design. The platform we use to evaluate is same as Table 4.10. Note that the `Q8b-BEST` is better than `STD-BASE` in all aspects.

Metric	`Q8b-BEST`	`STD-BASE`	Comparison
FREQ	200 MHz	150 MHz	1.0
LUT	23.03%	27.38%	0.839
FF	20.25%	25.63%	0.789
BRAM	31.24%	53.29%	0.586
DSP	62.71%	70.91%	0.884
Speed (FPS)	5034.71	3156.35	1.59
Power (W)	25.9	31.7	0.81

Table 5.2: Design statistics of the best built network.

We also compare `STD-BEST` and `Q8b-BEST` with results collected from CPU and GPU platforms. The

objective of this evaluation is to compare Plumber generated designs with CPU and GPU software in both standard network and quantised network, to show that `Q8b-BEST` is an optimal choice for deploying LeNet-5. We use the latest version of TensorFlow (1.3.0), which is directly downloaded from the officially published PyPI package [5], to run the software version and GPU accelerated version of each network mentioned above. We use Intel i7-950 and NVIDIA Titan X as evaluation platforms for CPU and GPU respectively. Result shows that `Q8b-BEST` can achieve the best performance and power consumption in terms of quantised LeNet-5. Note that the quantised nodes in the current TensorFlow version are not supported on GPU, and they are not well-optimised for the CPU platform we use. Thus, the performance of Q8b on CPU is worse than standard implementation, and there is no performance data for GPU in this case. We will try get a fair comparison result in the future.

	Intel i7–950	**Titan X**	**Stratix V**
Technique (nm)	45	16	28
Clock Freq. (MHz)	3.06×10^3	1537	150
Num. of Cores	8 cores	3072	—
Framework	TensorFlow (1.3.0)		Plumber (MaxDeep)
STD Speed (FPS)	1252	1.23×10^5	3156
STD Power (W)	114	243	31.7
Q8b Speed (FPS)	157	—	5035
Q8b Power (W)	114	—	25.9

Table 5.3: Comparison for LeNet-5 performance among CPU, GPU, and FPGA.

5.4 Summary

Plumber is a transpiler framework that end-to-end converts high-level CNN models into hardware designs. It has a frontend that transforms CNN models into dataflow graph, an intermediate representation in Plumber, and a backend that ports those dataflow graphs into hardware designs. The CNN model can in different formats and the targeting hardware platforms can also be various. Plumber is also integrated with a model-hardware co-optimisation module, which handles feedback information from hardware builds and decides what the best optimisation strategy for the original CNN model. Evaluation shows that, for a TensorFlow trained LeNet-5 model, Plumber can produce a hardware design that can achieve 99.25% accuracy on the MNIST dataset and 2521 frames per second.

[5] `https://pypi.python.org/pypi/tensorflow`

Chapter 6

Conclusion and Future Work

This chapter summarises achievements of this report, including the Ruby-based CNN building block library, RubyConv, the OpenSPL-based CNN hardware library on the Maxeler platform, MaxDeep, and the Plumber transpiler with model-hardware co-optimisation in Section 6.1. Section 6.2 lists differences among this report and previous MRes projects. Section 6.3 then discusses possible future work than can be extended from the current stage of the report.

6.1 Summary of Achievements

The ultimate target of this report is to accelerate the process that converts trained high-level CNN models into hardware designs on FPGA platforms. To achieve this goal, we list and accomplish three major objectives as follows:

1. RubyConv provides essential CNN building blocks written in Ruby, a high-level functional hardware description language. By using RubyConv, we can construct hardware designs for CNN in a concise approach, and analytically calculate their performance and resource usage.

2. Regarding hardware library on real FPGA platforms, we devise and implement MaxDeep, a hardware library for CNN on the Maxeler platform. MaxDeep provides parameterised, flexible, and easy-to-use interfaces to construct CNN accelerator designs. MaxDeep also embeds accurate analysis models that can predict the performance and resource usage of a complex CNN accelerator design. Based on these analysis models, we provide an optimisation methodology to find the best design parameters for given CNN topologies, by solving constrained optimisation problems. Moreover, MaxDeep supports building blocks for not only standard convolution layers, but also depthwise separable and binarised convolution layers.

   Our evaluation shows that MaxDeep can achieve xx GOp/s for a single convolution layer, better than xx. A complete LeNet-5 topology built and optimised by MaxDeep can achieve xx FPS.

3. We also design and implement a novel CNN model transpiler framework, Plumber, that can transform high-level CNN topology description and trained weights into FPGA hardware designs. The Plumber transpiler is accompanied with a model-hardware co-optimisation module that directs CNN model optimisation based on feedback information of generated hardware designs. For now, this co-optimisation module can re-train and compare several CNN models created by replacing standard convolution layers with depthwise separable and quantizing them to 8 bit representation.

Our evaluation of Plumber, including the transpiler and the co-optimisation module, shows that it can automatically optimise a standard LeNet-5 topology to its quantized version, based on feedback information provided by MaxDeep-generated LeNet-5 hardware designs. The best performance of LeNet-5 design generated by Plumber is achieved by quantization and depthwise separable replacement, which is GOp/s.

6.2 Comparison with MRes Projects

Comparing with the first MRes project, which first presents the idea of MaxDeep, this report mainly optimises the convolution layer architecture in many different aspects, such as increasing level of parallelisation, discussing three different computation sequences, and dives into the several cases when connecting different layers. This report also provides more thorough and detailed analysis models for performance and resource usage prediction, with sound evaluation on their accuracy. Moreover, MaxDeep in this report can construct more layers, such as depthwise separable and binarised convolution layers

In terms of enhancements comparing with the second MRes project, which focuses on implementing multi-pumping for CNN hardware designs on FPGA platforms, this report provides broader hardware optimisation options. This report considers multi-pumping as an orthogonal optimisation techniques. Also, we suppose that multi-pumped designs created from MaxDeep described in this report can achieve better performance comparing to those generated in the second MRes project, because levels of parallelisation in this report are multi-dimensional and can become larger than the previous version.

6.3 Future Work

Future work of this report include extensions on all three objectives accomplished, which are listed as follows:

1. For RubyConv, the most exciting work is to use Ruby as the target language of our CNN transpiler. The benefit of this approach is that, generating CNN designs written in high-level languages is much more concise and easier to implement regarding CNN model transpiler. And the hardware library will be less tedious to maintain. To achieve this goal, we need to enhance the current transpiler that converts designs written Ruby to other hardware description languages, which can be synthesized directly on FPGA platforms, and generate CNN designs from RubyConv.
2. MaxDeep also has several aspects that can be enhanced:
   (a) It is better that MaxDeep can support more CNN layer types, ranging from recently developed layers like astrous convolution and deformable convolution to support latest CNN topologies, to old-fashioned layers, such as LRN layer, to be compatible with typical CNN architectures like AlexNet. It will also be beneficial to enable deconvolution layer in MaxDeep, which is widely applied in many CNN architectures designed for computer vision tasks.
   (b) Because it is a trend that all CNN architectures are growing deeper, and cloud FPGA platforms are becoming popular, we need MaxDeep to support constructing CNN designs on multiple FPGA boards. MaxRing can be adopted to create multi-FPGA designs. Also, we need to revise the analysis model to take this case into account.

(c) Moreover, it is interesting to mix several optimisation techniques, for example, mixing precision of data representation or mixing convolution layer types. This enhancement should be considered together with the Plumber transpiler and the co-optimisation module. Also, we could consider not using external streams to read weights and alternatively use on chip ROM for small layers.

3. Regarding Plumber, the major enhancement is to explore the possibilities of supporting several CNN model sources, such as Caffe and MXNet, and FPGA platforms other than the Maxeler platform. The latter target is challenging but beneficial, because many CNN hardware designs are targeting embedded FPGA platforms.

In short, although this report has achieved several objectives, it will be more interesting when these future work are accomplished.

Bibliography

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., et al. (2016). Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*. pages 1, 58

Cavigelli, L. and Benini, L. (2016). Origami: A 803 gop/s/w convolutional network accelerator. *IEEE Transactions on Circuits and Systems for Video Technology*. pages 16

Chen, L.-C., Papandreou, G., Kokkinos, I., Murphy, K., and Yuille, A. L. (2014). Semantic image segmentation with deep convolutional nets and fully connected CRFs. *arXiv preprint arXiv:1412.7062*. pages 7

Chen, L.-C., Papandreou, G., Kokkinos, I., Murphy, K., and Yuille, A. L. (2016a). Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected CRFs. *arXiv preprint arXiv:1606.00915*. pages 7

Chen, Y.-H., Krishna, T., Emer, J. S., and Sze, V. (2016b). Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*. pages 18

Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., and Shelhamer, E. (2014). cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*. pages 13

Chollet, F. (2016). Xception: Deep learning with depthwise separable convolutions. *arXiv preprint arXiv:1610.02357*. pages 7

Coppersmith, D. and Winograd, S. (1990). Matrix multiplication via arithmetic progressions. *Journal of symbolic computation*, 9(3):251–280. pages 13

Courbariaux, M., Bengio, Y., and David, J.-P. (2014). Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024*. pages 19

Courbariaux, M., Hubara, I., Soudry, D., El-Yaniv, R., and Bengio, Y. (2016). Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*. pages 19

DiCecco, R., Lacey, G., Vasiljevic, J., Chow, P., Taylor, G., and Areibi, S. (2016). Caffeinated fpgas: Fpga framework for convolutional neural networks. *arXiv preprint arXiv:1609.09671*. pages 19

Du, Z., Fasthuber, R., Chen, T., Ienne, P., Li, L., Luo, T., Feng, X., Chen, Y., and Temam, O. (2015). Shidiannao: Shifting vision processing closer to the sensor. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 92–104. ACM. pages 16

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. `http://www.deeplearningbook.org`. pages 5

Gupta, S., Agrawal, A., Gopalakrishnan, K., and Narayanan, P. (2015). Deep learning with limited numerical precision. In *ICML*, pages 1737–1746. pages 16, 18, 19

Han, S., Mao, H., and Dally, W. J. (2015). Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*. pages 19

Hassibi, B., Stork, D. G., et al. (1993). Second order derivatives for network pruning: Optimal brain surgeon. *Advances in neural information processing systems*, pages 164–164. pages 19

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778. pages 9, 18

Holt, J. L. and Baker, T. E. (1991). Back propagation simulations using limited precision calculations. In *Neural Networks, 1991., IJCNN-91-Seattle International Joint Conference on*, volume 2, pages 121–126. IEEE. pages 19

Holt, J. L. and Hwang, J.-N. (1993). Finite precision error analysis of neural network hardware implementations. *IEEE Transactions on Computers*, 42(3):281–290. pages 19

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. (2017). MobileNets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*. pages 7, 18

Hunter, J. D. (2007). Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95. pages 45

Iandola, F. N., Han, S., Moskewicz, M. W., Ashraf, K., Dally, W. J., and Keutzer, K. (2016). Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size. *arXiv preprint arXiv:1602.07360*. pages 7, 18

Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. (2014). Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*. pages 1, 58

Jin, J., Dundar, A., and Culurciello, E. (2014). Flattened convolutional neural networks for feedforward acceleration. *arXiv preprint arXiv:1412.5474*. pages 7

Jones, G. and Sheeran, M. (1990). Circuit design in ruby. *Formal methods for VLSI design*, 1. pages 9, 10, 11, 21

Kingma, D. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*. pages 63

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105. pages 9

Kung, H.-T. (1982). Why systolic architectures? *IEEE computer*, 15(1):37–46. pages 16

Lavin, A. and Gray, S. (2016). Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4013–4021. pages 13

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324. pages 6

LeCun, Y., Denker, J. S., Solla, S. A., Howard, R. E., and Jackel, L. D. (1989). Optimal brain damage. In *NIPs*, volume 2, pages 598–605. pages 19

LeCun, Y. et al. (2015). Lenet-5, convolutional neural networks. *URL: http://yann. lecun. com/exdb/lenet*. pages 20

Li, Y. and Pedram, A. (2017). Caterpillar: Coarse grain reconfigurable architecture for accelerating the training of deep neural networks. *arXiv preprint arXiv:1706.00517*. pages 18

Luk, W., Lawrence, A., Lok, V., Page, I., and Stamper, R. (1994). Parametrised neural network design and compilation into hardware. In *VLSI for Neural Networks and Artificial Intelligence*, pages 197–206. Springer. pages 9, 11

Ma, Y., Suda, N., Cao, Y., Seo, J.-s., and Vrudhula, S. (2016). Scalable and modularized rtl compilation of convolutional neural networks onto fpga. In *Field Programmable Logic and Applications (FPL), 2016 26th International Conference on*, pages 1–8. IEEE. pages 19

Mathieu, M., Henaff, M., and LeCun, Y. (2013). Fast training of convolutional networks through FFTs. *arXiv preprint arXiv:1312.5851*. pages 13

McKinney, W. et al. (2010). Data structures for statistical computing in python. In *Proceedings of the 9th Python in Science Conference*, volume 445, pages 51–56. SciPy Austin, TX. pages 45

Nair, V. and Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814. pages 8

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830. pages 45

Qiu, J., Wang, J., Yao, S., Guo, K., Li, B., Zhou, E., Yu, J., Tang, T., Xu, N., Song, S., et al. (2016). Going deeper with embedded FPGA platform for convolutional neural network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 26–35. ACM. pages 16, 53

Rastegari, M., Ordonez, V., Redmon, J., and Farhadi, A. (2016). XNOR-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer. pages 19

Shoemaker, P. A., Carlin, M. J., and Shimabukuro, R. L. (1991). Back propagation learning with trinary quantization of weight updates. *Neural Networks*, 4(2):231–241. pages 19

Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*. pages 9

Srivastava, R. K., Greff, K., and Schmidhuber, J. (2015a). Highway networks. *arXiv preprint arXiv:1505.00387*. pages 18

Srivastava, R. K., Greff, K., and Schmidhuber, J. (2015b). Training very deep networks. In *Advances in neural information processing systems*, pages 2377–2385. pages 18

Sze, V., Chen, Y.-H., Yang, T.-J., and Emer, J. (2017). Efficient processing of deep neural networks: A tutorial and survey. *arXiv preprint arXiv:1703.09039*. pages 9, 14

Szegedy, C., Ioffe, S., Vanhoucke, V., and Alemi, A. (2016). Inception-v4, inception-resnet and the impact of residual connections on learning. *arXiv preprint arXiv:1602.07261*. pages 9

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2015). Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9. pages 9, 18

Umuroglu, Y., Fraser, N. J., Gambardella, G., Blott, M., Leong, P., Jahre, M., and Vissers, K. (2017). FINN: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 65–74. ACM. pages 38

Vasilache, N., Johnson, J., Mathieu, M., Chintala, S., Piantino, S., and LeCun, Y. (2014). Fast convolutional nets with fbfft: A GPU performance evaluation. *arXiv preprint arXiv:1412.7580*. pages 13

Walt, S. v. d., Colbert, S. C., and Varoquaux, G. (2011). The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30. pages 45

Wei, X., Yu, C. H., Zhang, P., Chen, Y., Wang, Y., Hu, H., Liang, Y., and Cong, J. (2017). Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs. In *Proceedings of the 54th Annual Design Automation Conference 2017*, page 29. ACM. pages 16

Williams, S., Waterman, A., and Patterson, D. (2009). Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76. pages 16, 46

Winograd, S. (1980). *Arithmetic complexity of computations*, volume 33. Siam. pages 13

Yu, F. and Koltun, V. (2015). Multi-scale context aggregation by dilated convolutions. *arXiv preprint arXiv:1511.07122*. pages 6, 7

Zhang, C., Li, P., Sun, G., Guan, Y., Xiao, B., and Cong, J. (2015). Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 161–170. ACM. pages 16, 53

Zhao, R., Song, W., Zhang, W., Xing, T., Lin, J.-H., Srivastava, M. B., Gupta, R., and Zhang, Z. (2017). Accelerating binarized convolutional neural networks with software-programmable fpgas. In *FPGA*, pages 15–24. pages 55, 56

Zhao, W., Fu, H., Luk, W., Yu, T., Wang, S., Feng, B., Ma, Y., and Yang, G. (2016). F-CNN: An FPGA-based framework for training convolutional neural networks. In *Application-specific Systems, Architectures and Processors (ASAP), 2016 IEEE 27th International Conference on*, pages 107–114. IEEE. pages 18