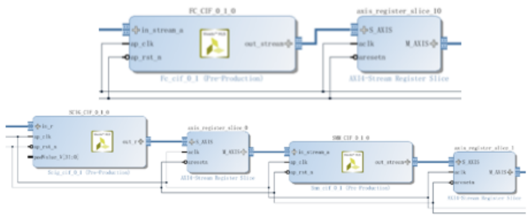
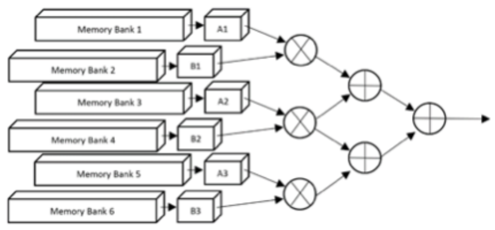
Imperial College London

Department of Electrical and Electronic Engineering

Final Year Project Report 2017





Project Title:	**PYNQ Classification - Python on Zynq FPGA for Neural Networks**
Student:	**Erwei Wang**
CID:	**00816456**
Course:	**4T**
Project Supervisor:	**Prof P.Y.K. Cheung**
Second Marker:	**Prof G.A. Constantinides**

ABSTRACT

Convolutional Neural Networks (CNNs) have achieved a significant amount of success in solving a wide range of classification problems. Traditionally, embedded CNN application prototypes have been implemented on CPU or GPU based machines due to short development time, but sacrificing performance and energy efficiency. However, recent advancements in high level synthesis (HLS) tools and PYNQ development boards are making the prototyping effort on FPGA comparable to that of CPUs or GPUs, making them a good option for prototyping embedded CNN applications. This report presents a fast FPGA prototyping framework, which is an Open Source framework designed to enable fast deployment of embedded CNN applications on FPGA platforms. My framework provides HLS CNN layers, which can be parameterised for a wide range of network specifications and provides state-of-the-art performance at low power consumption. By comparing with PYNQ ARM CPU implementation, my CIFAR-10 prototype shows up to 43x acceleration, while maintaining a 73.7% classification accuracy and 1.953 frames/J energy consumption.

Acknowledgement

I would like to express my sincere gratitude to my supervisor Professor P.Y.K. Cheung, who not only provided unstinting support and invaluable guidance throughout my four year's study in Imperial College, but also sparked my motivation to pursue a career in scientific research.

I would like to thank Dr. Peter Ogden for providing the PYNQ FPGA data transfer API design, which becomes the backbone of my project's architecture.

I would also like to thank Michaela Blott, Cathal McCabe, Giulio Gambardella and Andrea Solazzo from Xilinx Ireland Lab for the warm hospitality on our visit, as well as invaluable guidance on the techniques to optimise CNN implementation on FPGA. Thanks also to Patrick Lysaght from Xilinx Lab, San Jose, for initiating the Pynq project and provide all the support I needed to make this a successful project.

Special thanks to Stylianos Venieris and Junyi Liu from Circuits and Systems lab, as well as Aaron Zhao and Daryl Mah, who provided insightful ideas on the project and report.

Acronyms and Abbreviations

AI	Artificial Intelligence
API	Application Programming Interface
ASIC	Application-specific Integrated Circuit
BLAS	Basic Linear Algebra Subprograms
BRAM	Blocked Random Access Memory
BNN	Binarised Neural Network
CNN	Convolution Neural Network
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DSP	Digital Signal Processor
DMA	Direct Memory Access
FPGA	Field-programmable Gate Array
GPU	Graphics Processing Unit
HLS	High Level Synthesis
HPC	High-performance Computing
HTC	High-throughput Computing
IP	Intellectual Property
NIN	Network in Network
NN	Neural Network
OS	Operating System
PYNQ	Python Productivity for Zynq
RAM	Random Access Memory
ReLU	Rectified Linear Unit
RTL	Register Transfer Language
SDF	Synchronous Dataflow
SDFG	Synchronous Dataflow Graph
SoC	System on a Chip

Contents

1. Introduction

Deep Neural Networks have delivered state-of-the-art performances in both vision and auditory systems, showing its potential to revolutionise our existing technologies. The research on one Deep Neural Network, CNN, has been increasing over recent years. Similar to ordinary Neural Network (NN), CNN consists of interconnected neurons with learnable weights and biases. CNN specialises in processing images, which means CNN can exploit some properties of images to optimise its performance, making it more efficient in processing large image databases.

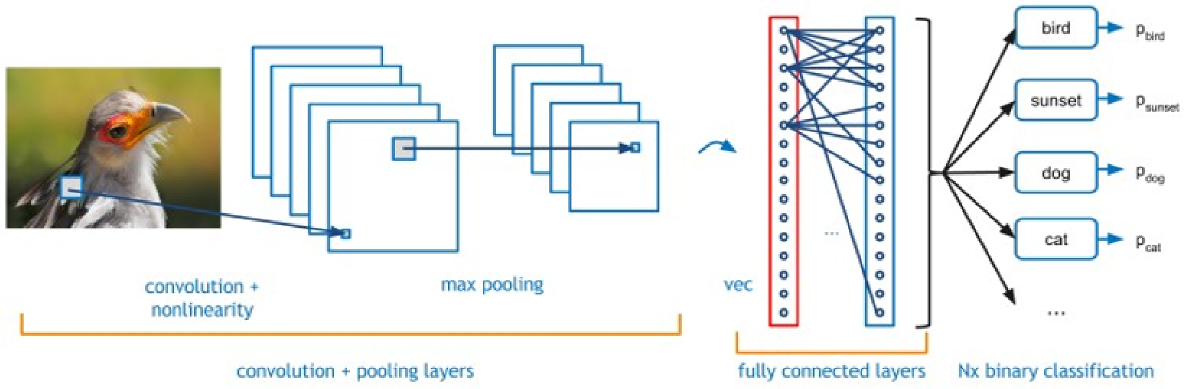


Figure 1.1: Illustration of a simple CNN, showing a convolution layer, a pooling layer, and a fully-connected layer [1]

The implementation of CNN on embedded systems allows real-time classification tasks to be performed more flexibly. Machine learning models can be trained off-line and then implemented onto embedded system, so that the system only needs to focus on improving the throughput of forward propagation (i.e. deployment). The development of high performance embedded CNN systems can have significant implications onto many areas of research, such as ADAS, UAV and robotics.

Meanwhile, CNN is also challenging the processing capability of our existing computing systems. For example, a well-known model, LeNet-5, requires $3.8 * 10^6$ operations in one forward propagation per image. Besides, real-time CNN classifications, such as video processing, require very high throughput to support a high processing frame rate. These requirements lead to the development of CNN hardware accelerators. [3] [4]

Figure 1.2: Embedded FPGA application: Aerotenna FPGA-based microwave radar sensors [2]

FPGA is a very promising accelerator for CNN. FPGA is an integrated chip which allows for gate-level hardware reconfiguration on the field. It contains a huge amount of small logic elements (or look-up-tables) which can be programmed into various and numerous digital modules for different custom applications. Below lists some of the major advantages of implementing embedded CNN applications on FPGA.

- FPGA's strong **parallel-processing capability** can be used to effectively exploit the inert parallelism in CNN algorithm, accelerating the CNN deployment on embedded systems.
- The **reconfigurability** of FPGA allows for the synthesis of hardware accelerators specially designed for each CNN model, allowing for higher optimisation in resource usage and greater flexibility for users' customisation in various applications.
- Research has shown that FPGA is able to provide high data processing throughput at **lower power consumption** than existing platforms. [5][6] Minimising power consumption is especially important for many embedded systems that have limited power supply (such as UAV and Automotive applications).

PYNQ is a development board which has recently been published by Xilinx^{TM}. The board features a Zynq XC7Z020 FPGA, with 512MB DDR3/FLASH memory and Dual-Core ARM Cortex-A9 CPU. PYNQ is a special FPGA platform which is very friendly to software engineers with limited experiences in working with CPU-FPGA heterogeneous architecture. The board provides a Linux Ubuntu 15.10 operating system with complete Python compiler support, and includes Python drivers that execute API for FPGA bitstream download and data transmission. It is a platform that aims at inviting engineers to explore the limitless design possibilities of FPGA.

With PYNQ platform, this project attempts to make the CNN deployment design flow on FPGA similar to the deployment on CPU or Graphics Processing Unit (GPU) platforms, so that engineers can deploy CNN on FPGA platform with intuitions that they are familiar with. My framework makes use of the Ubuntu OS on PYNQ's ARM core to execute Theano, which is currently one of the most popular AI designing frameworks. I packaged my FPGA API into a Theano CNN layer function, which can be instantiated in the same way as a Theano built-in function.

Figure 1.3: Embedded AI: Object detection for drone tracking [7]

In my framework, engineers construct CNNs using my pre-synthesised FPGA layer IP which are parameterisable to customise hardware resource usage. My IP designs employ the Synchronous Dataflow (SDF) model of computation as its basis.[8] Complete networks can be constructed as chains of layer IPs. With Xilinx^{TM} Vivado's "Block Diagram" utility, engineers can build CNNs graphically by simply chaining up IP blocks.

For less resource-demanding CNNs, my framework will execute all weights and layers onto FPGA, maximising the data parallelism in the form of heterogeneous computing. For more resource-demanding networks, my frameworks supports SDFG partitioning, where the entire network has been sliced into multiple sub-graphs, and each sub-graph being processed on FPGA sequentially.

Section 2 Project Scope explains the project's scope, detailing the project deliverable, main design choices and performance metrics. **Section 3 Background** demonstrates my research for the key technical aspects of the project and the relevant published works which my project can learn from. **Section 4 Implementation - Overall Architecture** gives an overview on the overall hardware architecture of my project. **Section 5 Implementation - ARM Linux OS Side** provides the technical details of my high level interface design, including the Python library interface and CPU-FPGA API. **Section 6 Implementation - Zynq FPGA Side** details the FPGA-side data streaming interface protocol design and IP library design. **Section 7 Testing** describes the test settings, including validation procedure and experimental setup for designing prototypes. **Section 8 Performance Evaluation** describes the three prototypes that I have constructed using my framework, demonstrating the performance of my framework. **Section 9 Conclusion and Further Plans** provides conclusions and further plans. **Section 10 About My Experience of Working with PYNQ** summarises my experience of working with PYNQ platform. **Section 11 User Guide** provides a sample user guide explaining to engineers how to use my framework.

2. Project Scope

This project aims at constructing a fast FPGA prototyping framework for high performance CNN deployment on PYNQ platform. I will design a framework that not only minimises engineers' efforts in prototyping CNN on FPGA platform, but also optimally utilises both the ARM core and FPGA hardware, delivering state-of-the-art CNN deployment speed, accuracy and power consumption.

My **target platform** is Xilinx^{TM} PYNQ Development Board. With a powerful Zynq FPGA and a Dual-Core ARM-Cortex A9 processor running Linux OS, PYNQ is an ideal platform for constructing my framework, which requires both an FPGA for delivering High-performance Computing (HPC) tasks, and a Linux OS for high-level-language design interface.

As the final **deliverable** of my project, the framework should include the following elements.

- A **library** of FPGA IP designs packaged in the format of block designs, which will be used by engineers as building blocks for their CNN models.
- A **Vivado project** which synthesises engineers' CNN model into FPGA bitstream.
- A **Python function** on PYNQ ARM Linux OS which deploys the CNN model on FPGA.

Since in embedded CNN deployment large amounts of operations are required for a prolonged period of time, this task can be categorised as a High-throughput Computing (HTC) task, and the **metric** for speed performance will be throughput, in operations per second (OP/s).

In order to achieve this goal, below are some of the design considerations which I will attempt to address throughout my project. [1]

[1] It is important to note that my framework does not provide solutions for automated FPGA design space optimisation. This means that users are required to hand tune parameters such as memory tiling factors and Blocked Random Access Memory (BRAM) usage for

- **CNN Framework on PYNQ OS:** Which currently available CNN framework to be implemented on PYNQ Linux OS as high level interface? The Linux machine has limited Random Access Memory (RAM), storage and processing power. An ideal framework should both demand less RAM and less computing power, and provide better support on user-customised layers (our FPGA layers will be instantiated as user-customised layers).
- **FPGA Data Transfer Protocol:** Currently PYNQ drivers support two data transfer protocols, namely Memory-Mapped Input Output (MMIO) and DMA. Which protocol will provide higher throughput?
- **Quantisation:** Should data and weights be quantised for FPGA computation? What are the gains and losses in data quantisation?
- **Memory Architecture:** Should data and weights be stored on-chip (BRAM) or off-chip (DDR)?
- **FPGA layer IPs Parametrisability:** When designing FPGA layer IPs, trade-offs exist between speed and BRAM usage, and between speed and Digital Signal Processor (DSP) usage. How to provide users with the freedom to control these trade-offs?
- **Scalability:** What happens when a CNN is too resource-demanding to fit in my FPGA platform?

The above design decisions will be made based on analysis of experimental results. As proof-of-concept, FPGA-accelerated CNN prototypes will be developed, which aim at delivering state-of-the-art deployment speed and accuracy on supervised image classification benchmarks, using my fast FPGA prototyping framework. The performance evaluation of these prototypes will be focusing on resource usage, throughput, power consumption and accuracy. Below are the performance evaluation metrics that my project will focus on.

- **Resource Usage:** The amount of FPGA on-chip resources utilised.
- **Throughput:** The amount of computations that can be carried out at a given period of time, often measured in MACC operations per second (OP/s).
- **Power Efficiency:** The sustained energy usage per frame.
- **Accuracy:** Supervised classification accuracy of existing performance benchmarks.

The above aspects will be considered when evaluating on the performance of this framework. Finally, future plans on this project will be made based on the performance evaluation.

Section 3 will explain in detail my background research on the various design questions posed in this section.

optimal FPGA implementation. My framework, however, do simplify users' optimisation efforts by providing appropriate generic parameters.

3. Background

This section demonstrates the background research on some of the key technical aspects relevant to the project. Various related works on these topics have been listed, which my design will refer to when making key design decisions.

Section 3.1 explains the fundamental algorithm of CNN. Section 3.2 shows why a high level interface is needed, as well as research on how to implement a user-friendly high level interface. Section 3.3 lists the advantages of implementing CNN on FPGA platforms, as well as my research on how to optimally implement embedded CNN on FPGAs based on existing published works. Section 3.4 details my research on the topic of CNN quantisation. Section 3.5 justifies my choice of PYNQ as my project's hardware platform.

3.1. What is a CNN?

The basic algorithm of a CNN is very similar to a common neural network, since they are both built from large amounts of neurons performing dot-product with learn-able parameters (also named **"weights"**), and they both require back propagation for training and forward propagation for testing. Different from a common neural network, a CNN assumes that the input data will always be in the form of images. With that assumption, the forward propagation of CNN becomes equivalent to a 2D convolution, which imply strong and deterministic parallelism that can be utilised for acceleration.

A typical CNN consists of many different layers that operates on the feature map sequentially. Each subsequent layer reads input features from output of previous layer. A convolution layer performs 2D convolution, a fully-connected layer performs dot-product, a Rectified Linear Unit (ReLU) layer performs activation thresholded at zero, and a pooling layer performs down-sampling by taking maximum or average. Finally, the last layer (normally a fully-connected layer) outputs an array of probabilities for the corresponding classes. The most frequently-used CNN layers are convolution layers, pooling layers, ReLU layers and fully-connected layers.

CONVOLUTION LAYER: A convolution layer performs 2D convolution between input feature map and weights. Weights are arranged in small parameter patches named kernels. Normally, the output of convolution layer often directly connects to a non-linearity layer to perform non-linear activation. Common non-linear activations include ReLU, sigmoid and linear functions.

$$f_i^{output} = \sum_{j=1}^{n_{in}} f_j^{in} \circledast g_{i,j} + b_i, \ (1 \le i \le n_{out}) \tag{3.1}$$

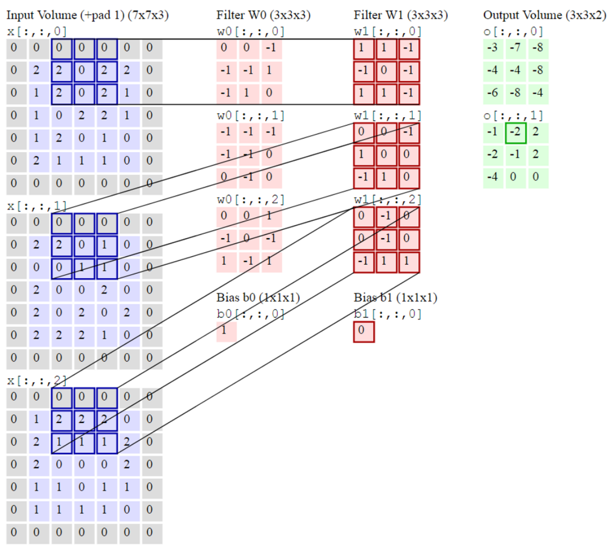


Figure 3.1: Illustration of 2D convolution [9]

In 2D convolution, there exhibits strong parallelism. Firstly, each pixel in a 2D convolution feature map is computed from dot-product between weight kernel and a patch of input feature map. Each dot-product is independent from each other, suggesting that all dot-products can be computed in parallel. Secondly, within each dot-product, all the multiply-accumulation operations are independent and hence can be computed in parallel as well.

POOLING LAYER: A pooling layer down-samples output feature map by outputting the maximum or average of sub-areas of output feature map. Pooling operation can significantly reduce the computation complexity of the network, as well as effectively preventing over-fitting. Equation 3.2 shows max-pooling operation.

$$f_{i,j}^{output} = max_{p*p}\begin{pmatrix} f_{m,n}^{input} & \cdots & f_{m,n+p-1}^{input} \\ \vdots & & \vdots \\ f_{m+p-1,n}^{input} & \cdots & f_{m+p-1,n+p-1}^{input} \end{pmatrix} \tag{3.2}$$

FULLY-CONNECTED LAYER: A fully-connected layer performs dot-product between input feature map and weights. Normally, fully-connected layers are placed at the end of the network to reduce the dimensions of output feature map.

$$f^{output} = W f^{in} + b \tag{3.3}$$

RECTIFIED LINEAR UNIT (RELU) LAYER: A ReLU layer performs linear activation but thresholded at zero. ReLU is a special activation function which becomes very popular in CNN mainly because of two reasons. Firstly, according to research done by Krizhevsky et al[10], ReLU can significantly accelerate the speed of convergence in the stochastic gradient descent algorithm, reducing training time. Secondly, as compared to tanh sigmoid or logistic sigmoid functions, the computation of ReLU is significantly simpler. [9] In my FPGA layer IPs, instead of a standalone ReLU layer, each layer IP will have the option of whether to directly rectify each output data as the output feature map streams out, with zero extra latency cost.

$$f^{output} = MAX(f^{in}, 0) \tag{3.4}$$

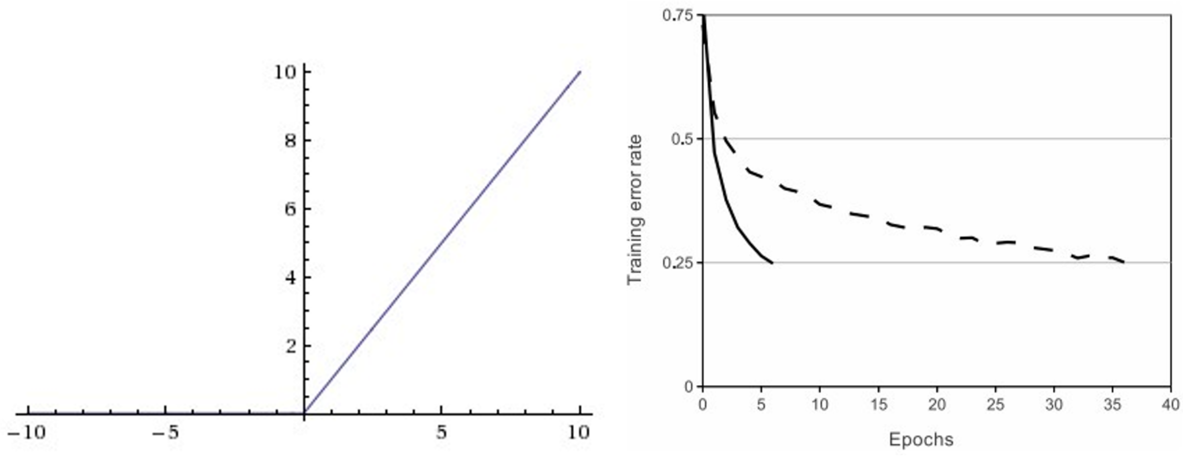


Figure 3.2: Left: Rectified Linear Unit (ReLU) activation function, which is zero when x < 0 and then linear with slope 1 when x > 0. Right: A plot from Krizhevsky et al. [10] paper indicating the 6x improvement in convergence with the ReLU unit compared to the tanh unit. [9]

In this project, since my target platform is embedded system, I always assume that CNN models are pre-trained off-line. Hence, my framework will only focus on CNN deployment (i.e. forward propagation), and CNN training (i.e. back propagation) do not fall into the scope of my project[11]. Section 6 will explain how each CNN layer algorithm has been implemented in the FPGA layer IP library.

3.2. Framework High Level Interface Design

3.2.1. Why Do We Need Another High Level CNN Framework?

One fundamental aim of this project would be to allow engineers to quickly and easily prototype embedded CNN applications. To achieve that, I propose a high-level-language user interface using existing CNN frameworks, so that engineers who are familiar with deploying CNN on CPU and GPU platforms can quickly get used to deploying CNN on FPGA platform using my framework.

To achieve that, I decided to install one existing CNN framework onto the PYNQ Linux machine, and package my FPGA-accelerated layers into user-defined functions (or **"customised layers"**) of existing CPU/GPU based CNN frameworks. This way, the network deployment process using my platform will be similar to deployment process that engineers are already familiar with.

Currently, the most popular CNN frameworks include Caffe, TensorFlow and Theano. Section 5 will compare these three frameworks and choose the one which best satisfies my project requirements.

3.2.2. Existing FPGA CNN Frameworks and Their High Level Interfaces

Stylianos I. Venieris and Christos-Savvas Bouganis propose *fpgaConvNet* which reads CNN descriptions from existing networks such as Caffe and Theano, and maps the CNN onto a particular FPGA-based platform. [5] The developed framework first takes as input a CNN model in the high-level, domain-specific scheme, then performs fast design space exploration by manipulating the SDF CNN model and finishes by generating a synthesisable Vivado High Level Synthesis (HLS) hardware design. Their designs achieve up to 1.62x the performance density of hand tuned designs. However, their design focuses more on an automated design methodology for the mapping of CNN onto FPGA platforms. After the FPGA bitstream has been generated, engineers still need to construct CPU-FPGA data transmission architecture, which requires extensive FPGA knowledge. I propose to design a framework which can simplify this process.

Nallatech's *FPGA Acceleration of Convolutional Neural Networks* has proposed framework with FPGA-CNN layer incorporated in Caffe framework. [1] User needs to only change the description of the CNN layer in the Caffe XML network description file to target the FPGA equivalent. This enables software designers to continue working on framework that they are familiar with, hence reducing the prototyping time. This concept satisfies my design requirements, and hence my framework will utilise similar model, targeting PYNQ platform.

3.2.3. Installing CNN Frameworks on Embedded ARM Chipset

I also reviewed literatures regarding how to install Caffe, TensorFlow and Theano onto ARM chipset. Some tutorials regarding installing CNN frameworks onto Raspberry PI has been used on PYNQ board because they both use similar ARM chipset[12][13]. Some tutiroals also explain how to solve Boost library version clashing issues[14], and guide me to implement Caffe/Theano Python 3 support (currently Caffe does not officially support Python 3, but PYNQ requires Python 3 libraries to interface with FPGA)[15].

3.3. FPGA Layer IP Library Design

3.3.1. Why Is FPGA Good at Accelerating CNN?

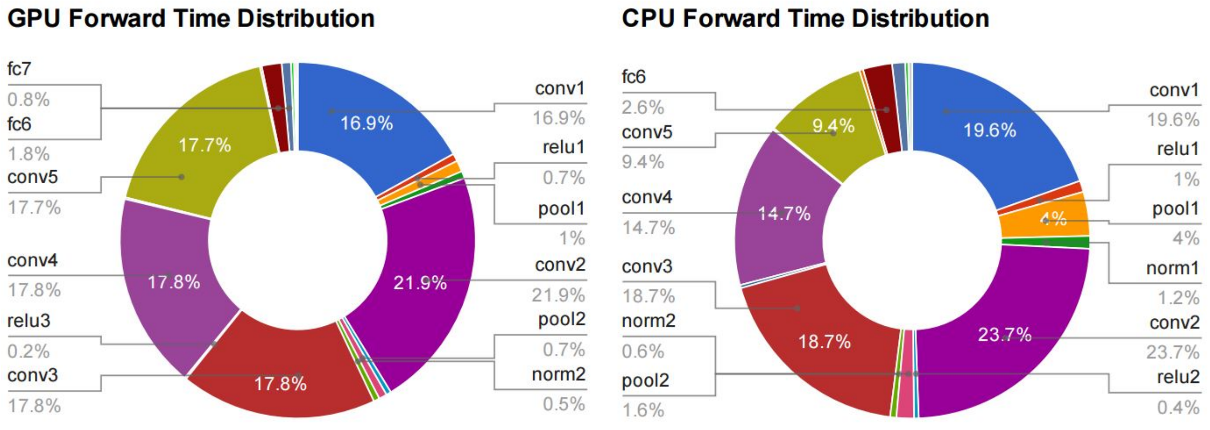


Figure 3.3: Computation time distribution of individual layers of AlexNet, on both GPUs and CPUs for the forward pass[16]

There has been many research regarding accelerating CNN on various accelerators based on FPGA, GPU, and even Application-specific Integrated Circuit (ASIC). FPGA has attracted more and more attentions due to its high energy efficiency, fast development cycles, and reconfigurability[17].

The majority of computation latency in one CNN forward propagation (deployment) lies in the convolutions. Figure 3.3 is the time distribution for CPU and GPU forward propagation (from research done by Yangqing Jia[16]). This shows that the convolution layer takes the majority of the CNN forward propagation processing time[18], and accelerating 2D convolution is the key to achieving more acceleration in the overall deployment.

The 2D convolution implies strong data parallelism. As shown in Section 3.1, in 2D convolution every multiply-accumulation operation can be processed in parallel. This parallelism can be easily exploited in an FPGA architecture. Four related works which provide outstanding performance are compared in the following section.

3.3.2. Related Works on Accelerating 2D Convolution on FPGAs

One design proposed by Chen Zhang et-al. implements a single processing engine style architecture by loading batches of input onto on-chip memory, followed by pipelining and unrolling 32-bit floating point multiply-accumulations in tiles.[17] This maximises the parallelism in the algorithm, but utilises a significant amount of on-chip BRAMs and DSPs, making it unsuitable for the limited resources on PYNQ board (4.9 MB on-chip BRAM and 220 DSP slices).

Another design proposed by Di Carlo S. et-al. performs convolution using line buffers and windows. This design was originally for 2D convolution, but can also perform 3D convolution with some minor changes. At each clock the buffer shifts down and a new pixel streams in. A window captures some pixels in the buffer and all pixels in the window perform MAC in parallel. This model fits the AXI4-streaming I/O interface very well, using minimum memory bandwidth and maximum data throughput[19].

Sharan Chetlur et-al. propose a design which uses CPU to firstly convert convolution into matrix multiplication using a function named **"im2col"** [20]. This design was originally proposed for CPU and GPU architecture. The FPGA acceleration of matrix multiplication has been extensively researched and high acceleration can be easily achieved. However, the input data stream is not just the input feature map, but the output of "im2col" function. This is especially problematic since the size of the feature map scales up significantly after "im2col", which is essentially a memory unrolling process. Hence huge memory transfer latency is expected with this design.

Stylianos I. Venieris and Christos-Savvas Bouganis' *fpgaConvNet* proposes a framework which interprets CNN as a streaming application. The proposed framework employs the SDF model of computation as its basis, where both sliding window unit (i.e. **im2col**) and matrix multiplication are processed on-chip using streaming interface. This model effectively eliminates the excessive off-chip memory transformation overhead from the previous design. Moreover, a streaming interface means output feature map from each layer does not need to be buffered on on-chip memory, hence reducing memory footprint. The streaming data interface enables the FPGA to exploit data parallelism in the form of pipelining. Thus, the SDF model proposed by this model is most ideal for my project specifications.

To sum up, the first two designs aim at maximising parallelism on FPGA architecture using the architecture of a single processing engine, but their use of floating point numbers result in very high on-chip memory usage. Besides, having a single processing engine means that feature maps need to be constantly transmitted in and out of DDR memory. Especially, the size of the feature map expands significantly in the middle of the CNN SDFG, meaning that even longer data transfer latency from DDR memory is expected. Both problems make them less suitable for an embedded system. The third design appears to solve that problem, but results in more off-chip memory operations and hence more latency. The fourth design's SDF streaming IO model provides good performance with minimal memory footprint, and in my framework I decide to implement this concept. Section 6 provides the technical details on how this concept has been implemented in hardware.

3.4. Data Quantisation

3.4.1. Why Do We Quantise CNN?

Recent research has shown that actually CNN is rather insensitive to noises in the input data and weight, and the precision of CNN forward propagation does not have to be **32-bit floating point**. Fixed point number representations are sufficiently accurate, because CNN forward propagation can be very robust to quantisation noises[21][22][23][24]. CNN models can be trained in 32-bit floating point for improved precision, and then quantised in deployment for acceleration and memory compression. Fixed point representation can significantly reduce both the on-chip memory usage and DSP usage.

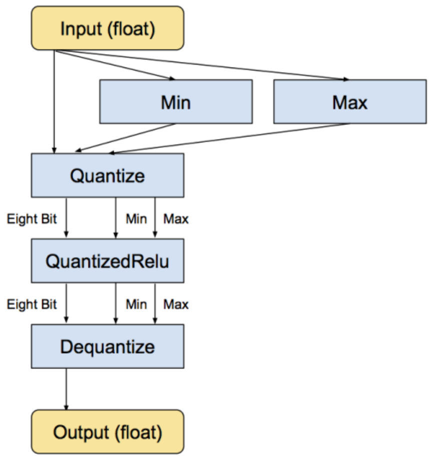


Figure 3.4: Graph Showing Dataflow of Quantized CNN Layers[21]

3.4.2. Existing CNN Quantisation Schemes

Figure 3.4 shows the data flow when **fixed-point quantisation** of a part of the network has been implemented. Similar data flow can be implemented before and after FPGA IP layers. There is a trade-off between accuracy and bitwidth, and the optimal bitwidth needs to be selected.

Besides using constant fixed-point bitwidth, research has shown that the **dynamic quantisation** of CNN can minimise bitwidth usage, while at the same time maximise accuracy. For example, Ristretto performs automated CNN quantisation using dynamic quantisation. [25]

Xilinx^{TM} FPGA also has **INT8 DSP hardware architecture acceleration** design, which outperforms other FPGAs when accelerating the MACC operations in INT8[22]. However, at the moment, Vivado HLS cannot exploit this feature. It can only be exploited by hand-tuned HDL.

In the recent FPGA 2017 conference, Xilinx^{TM} published their implementation of **binarised neural network inference**. By utilizing a novel set of optimizations that enable efficient mapping of binarised neural networks to hardware, they implemented fully connected, convolutional and pooling layers, with per-layer compute resources being tailored to user-provided throughput requirements. Currently, this design can provide some of the fastest classification rates reported on classification benchmarks. With binarised data representation, the FPGA design requires minimal BRAM and DSP, thus achieving high scalability.[6]

In FPGA implementation of CNN, quantisation can significantly reduce the resource usage, ensuring better scalability and more parallelism. Section 6.2 will compare the different quantisation schemes and select the most optimal design for CNN deployment.

3.5. PYNQ Platform

3.5.1. What is PYNQ Platform?

Traditional FPGA-CPU and FPGA-RAM interfaces requires designer to have knowledge on FPGA hardware implementations. This has created a barrier for potential software designers to try and use FPGA accelerator into their applications. This barrier has restricted the growth of the FPGA developer community, resulting in slower development of FPGA applications. The main counterpart of FPGA accelerator, GPU, on the other hand, has been welcoming software developers with C-like programming languages and frameworks (such as OpenCL[26]), resulting in the formation of a very successful GPU developer community and productive supply chain of GPU applications.

Nowadays, the FPGA industry has raised its focus on creating more user-friendly development frameworks. Recently, Xilinx^{TM} has published PYNQ project[27]. It is an innovative framework where embedded engineers can instantiate pre-synthesised FPGA IPs (or "Overlays") in Python, without digging into hardware level.

This platform is especially suitable for implementing embedded CNN system. The CPU core on PYNQ has built-in Linux Ubuntu 15.10 OS. This makes PYNQ capable of running software CNN frameworks such as Theano completely on the CPU. Layers of Theano which are parallelisable will be dispatched into PYNQ pre-synthesised "Overlays" and instantiated in standard Theano syntax. Thus, FPGA acceleration of CNN on Theano can be achieved without users doing extensive Register Transfer Language (RTL) design. Besides, the Zynq FPGA on PYNQ has 53200 LUTs, 4.9MB (140 36KB blocks) of BRAMs, and 220 DSPs (18 x 25 MACC). These on-chip resources are powerful enough for engineers to implement most of common CNNs.

3.5.2. Alternative Platforms for CNN Deployment

There are also many research on implementing CNN systems on embedded GPU boards. One of the most popular implementation platform would be Nvidia's Jetson mobile GPU board. Jetson is an embedded system equipped with Nvidia GPU. Many developers have been trying to deploy CNN on this framework, and have achieved great performance. However, Nvidia Jetson board is more expensive than PYNQ-Z1 board. Meanwhile, PYNQ-Z1 FPGA is more power-efficient than GPU systems, which is crucial to embedded applications with limited power supply.[28][11][6] Table 3.1 compares FPGA performance with embedded GPU, in scene labelling task, showing that FPGA platform achieves higher power efficiency and lower hardware cost.[5]

Implementation	Device	Sustained Performance	Performance per Power	Cost
fpgaConvNet	Zynq-7000 XC7Z020	12.73 GOp/s	7.27 GOp/s/W	£65
Scene Labelling	TK1	76.00 GOp/s	6.91 GOp/s/W	£196.14

Table 3.1: Performance Comparison with embedded GPU [5]

3.6. Summary

This section demonstrates my background research on key technical aspects of the design. Section 4, 5 and 6 will talk about the design implementation, detailing how the design considerations proposed in Section 2 have been approached using design space analysis.

4. Implementation - Overall Architecture

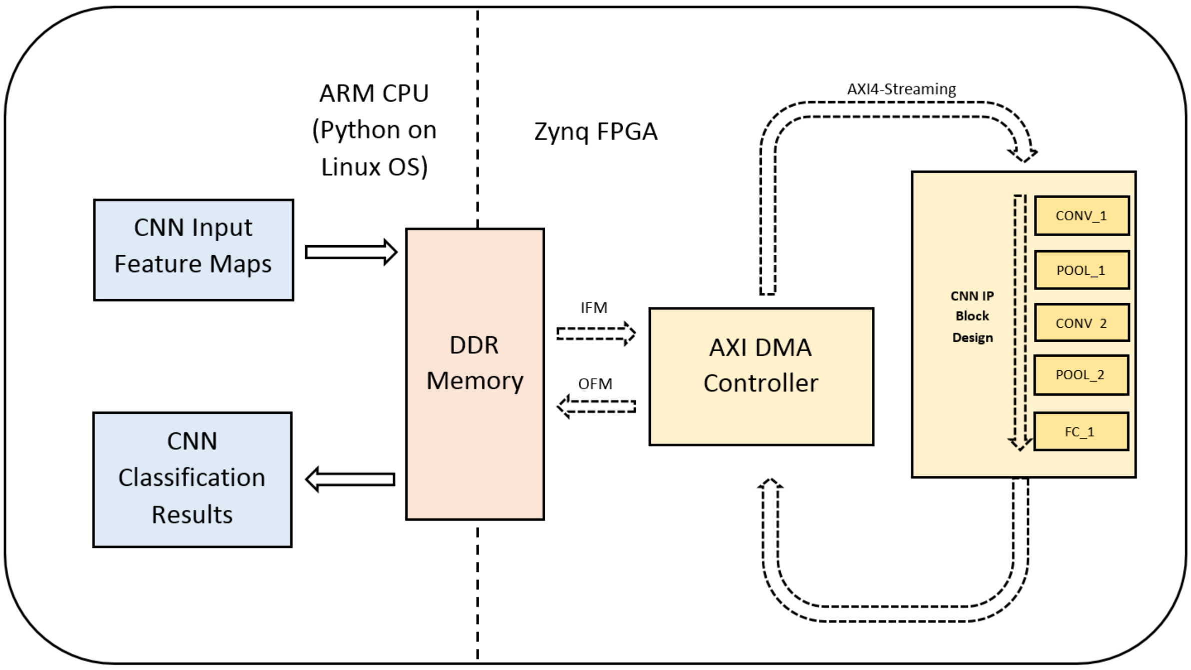


Figure 4.1: The overall hardware architecture of my design

Figure 4.1 summarises the overall architecture of my framework. The architecture of my design can be separated into two parts, namely ARM Linux OS implementation and Zynq FPGA implementation. The left half of the diagram shows the ARM CPU side which is controlled by Python on Linux OS. This side controls the high level interface of my framework, where the framework loads input feature maps to DDR memory and outputs the classification results. The right half of the diagram shows the Zynq FPGA side which is hardware-configured by overlay IP. This side performs high speed forward propagation of CNN in SDF paradigm.

Section 5 and 6 explains the ARM Linux side and Zynq FPGA side of implementation, respectively. Both halves of the implementation aim at delivering a fast prototyping framework for high performance CNN deployment. Hence, design decisions have been made to minimise engineers' deployment efforts, while in the meantime delivering optimal resource usage efficiency, throughput, power efficiency and accuracy.

5. Implementation - ARM Linux OS Side

In order for more engineers to get used to my framework, I choose some of the most popular frameworks (reported to date) as my framework's high level interface. Caffe, TensorFlow and Theano are some of the most popular CNN frameworks which target CPU/GPU based platforms. Table 5.1 provides a systematic comparison among these three frameworks.[29]

Framework	Creator	Open Source	Platform	Written In	Interface	Has Pre-trained Models
Caffe	Berkeley Vision and Learning Centre	Yes	Linux, Mac OS X, Windows	C++	Python, MATLAB	Yes
TensorFlow	Google Brain team	Yes	Linux, Mac OS X, Windows	C++, Python	Python, C/C++, Java, Go	Yes
Theano	Universite de Montreal	Yes	Cross-platform	Python	Python	Through Lasagne's model zoo

Table 5.1: Deep learning framework by name[29]

All three frameworks are open source, which means I have free access to them. They all support Linux platform, and all supports Python interface. This means they should all be able to operate on the Linux OS on PYNQ. Hence, in order to give a closer look at these frameworks in action, I attempted to install all three platforms on PYNQ.

5.1. Framework Installation and Setup on PYNQ Linux OS

5.1.1. Caffe

The first framework that I tried to install was Caffe. In the installation process I met several errors, which I was able to solve. The first error was Boost library installation. Linux **apt-get install** command was not able to install correct version of Boost library for ARM core cross-compile gcc compiler. Hence, I downloaded the correct version of Boost from its official website. A more difficult error was that as for the day that I wrote the report, Caffe did not have official support for Python 3. However, the data transmission API that PYNQ provided only supported Python 3. Hence, I manually installed **protobuf 3.0.0**, **OpenCV 3**, and **LAPACK**, and changed the Caffe installation makefile script to link to these new components. Eventually, I successfully installed Caffe onto PYNQ Linux machine.

Having installed the Caffe framework, I proceeded to setting up a small testbench which instantiates a FPGA-accelerated CNN layer from Caffe. For testing purpose, the FPGA-accelerated layer had only one convolution layer IP. (For details on FPGA layer IP design, please see Section 6) The testbench includes a testbench setup script,

"single_conv_layer.py", which calls for the setting up and execution of the testbench network in Caffe syntax, and a CNN configuration file, **"myconvnet.prototxt"** Protobuf configuration file, which declares the CNN structure and specifications. For contents of **"single_conv_layer.py"** and **"myconvnet.prototxt"**, please see Appendix A.

In **"myconvnet.prototxt"**, the layer named **foga_conv_im2col** is a customised Caffe layer that I added into the Caffe library, which contains API that executes the FPGA convolution layer IP. In Caffe terminology, this customised layer is named **Python Layer**, mainly because the layer is written in Python. However, since Caffe built-in library is written in C++, when Caffe calls the Python Layer, there will be some small overhead in executing Python compiler, making the performance less desirable. Besides, having embedded Python script inside a C++ library makes the overall library more difficult to maintain or upgrade.

5.1.2. TensorFlow

The second framework that I attempted was TensorFlow. I installed TensorFlow on PYNQ with no errors. However, whenever I tried to execute a CNN with **session.run()** command, I always got the "Illegal Instruction" error due to chipset clash. When I consulted with engineers in Xilinx^{TM}, they mentioned hacks which would enable PYNQ to execute TensorFlow. However, by that time I had already switched to Theano and there was no time left for me to attempt TensorFlow any more.

5.1.3. Theano

The third framework that I attempted was Theano. I installed Theano onto PYNQ using **apt-get install** command with only one line of script (*pip install Lasagne==0.1*). To further improve on the user interface of CNN deployment, I also installed **Lasagne** on top of Theano. Lasagne is a lightweight library to build and train neural networks in Theano, providing more succinct and intuitive code presentation than Theano. Lasagne also published a sub-repository named **Recipes**, which provides examples and tutorials explaining how to quickly and easily deploy pre-trained Caffe CNN models. I can easily build my testbench based on these tutorials. In Listing 1, a LeNet-5 CNN was implemented using Lasagne syntax. In Listing 2, an identical network model has been implemented using my framework, with the same syntax. From both Listing 1 and 2, I can conclude that using Lasagne, CNNs can be declared with simple and readable Python codes, providing user-friendly design interfaces.

Since Lasagne has good support for user-designed customised layers, I can simply use Lasagne customised CNN layers as my framework's high level interface. These layers can be instantiated identically to Theano built-in layer functions, providing performance which is as good as built-in layer functions. Based on a tutorial project provided by Lasagne, a testbench project was designed to test on the functionality of my framework. It consists of an iPython Notebook script (modified from Lasagne tutorial) and a customised Lasagne CNN layer which contains

API that executes the FPGA convolution layer IP. Listing 2 shows how my framework declares an FPGA-accelerated CNN model.

Listing 1: Lasagne LeNet Configuration

```
net = {}
# Input image with dimension 28 x 28
net['input'] = InputLayer((None, 1, 28, 28))
net['conv1'] = ConvLayer(net['input'],
    num_filters=20, filter_size=5,
    nonlinearity=linear)
net['pool1'] = PoolLayer(net['conv1'],
    pool_size=2, stride=2, mode='max',
    ignore_border=False)
net['conv2'] = ConvLayer(net['pool1'],
    num_filters=50, filter_size=5,
    nonlinearity=linear)
net['pool2'] = PoolLayer(net['conv2'],
    pool_size=2, stride=2, mode='max',
    ignore_border=False)
net['ip1'] = DenseLayer(net['pool2'],
    num_units=500, nonlinearity = rectify)
net['ip2'] = DenseLayer(net['ip1'],
    num_units=10, nonlinearity = None)
net['prob'] = NonlinearityLayer(net['ip2'],
    softmax)
```

Listing 2: My Framework's LeNet Configuration

```
net = {}
# Input image with dimension 28 x 28
net['input'] = InputLayer((None, 1, 28, 28))
net['lenet'] = FPGA_LENET(FPGA_net['input'])
# FPGA_LENET is the customised layer which
    wraps the API, calling for the execution
    of FPGA LENET IP









net['prob'] =
    NonlinearityLayer(FPGA_net['lenet'],
    softmax)
```

5.1.4. Evaluation

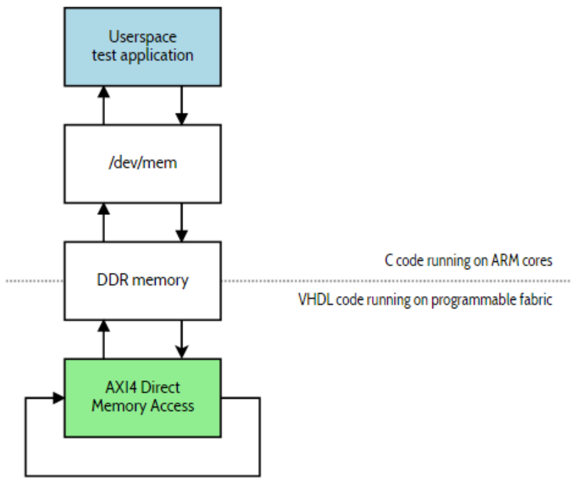
Having attempted all three frameworks, I evaluated these frameworks based on three fundamental aspects, namely installation difficulty on PYNQ, support for customised layer, and simplicity in design interface. Table 5.2 compares these three frameworks. From Table 5.2, I conclude that Theano (with Lasagne) requires the least installation efforts, while providing the most intuitive design interface and the best supports on embedded CNN deployment as well as customised layer design. Hence, Theano with Lasagne is chosen as the most optimal framework for my project.

Framework	Installation Difficulty on PYNQ	Support for Customised Layer	Simplicity in Design Interface
Caffe	Moderate. Requires Compilation which is more than 10 hours long	Low. Embedding Python function leads to extra Python interpreter latency	Moderate. layer instantiatoin in Protobuf text format
TensorFlow	NA. Failed to install	High	Simple
Theano (Lasagne)	Very Easy. Installed with a single line of command. No need for compilation	Very high. Well-designed customised functions' performances are comparable to built-in functions	Very simple. Directed Acyclic Graph (DAG) constructed in Python functions

Table 5.2: Evaluation of the listed CNN frameworks

5.2. API for FPGA-CPU Data Transmission

One crucial task of CPU-side framework is to be able to call for data transmission from DDR memory to FPGA on-chip BRAM. This process requires driver which is able to link to FPGA-connected ports and delegate to FPGA hardware for data transmission. Fortunately, PYNQ provides two types of such drivers, namely **MMIO** and **DMA**.


Userspace test application
/dev/mem
DDR memory
C code running on ARM cores
VHDL code running on programmable fabric
AXI4 Direct Memory Access
Figure 5.1: Block diagram of the minimal working hardware for DMA. The DMA output stream is looped back into input stream[30]

MMIO stands for memory-mapped input/output. This is a rather straight forward memory access protocol. In short, a pointer is initialised with the physical address of the FPGA control/status register(s) (i.e. port(s)), and the driver simply loads and stores to the FPGA device through that pointer. MMIO driver is simple to implement and is hence commonly used in applications with small amount of memory transmissions.

DMA stands for direct memory access. In DMA, data transmission is controlled and scheduled by an independent controller on FPGA. It focuses on transmitting data at maximum throughput possible, while CPU and other FPGA IPs can focus on other tasks. **AXI DMA** specialises in transmitting a stream of data in AXI4-Streaming protocol, between FPGA and DDR memory. AXI DMA focuses on boosting the throughput of transmitting a large stream of data, and is capable of supporting a throughput of one data word per clock cycle. [30]

For CNN deployment, large amounts of data transformations need to be performed at very high speed between

FPGA and DDR memory, and a combination of AXI4-Streaming protocol and DMA can provide higher data transmission throughput. Hence, in my project, DMA instead of MMIO has been selected as the FPGA data transfer architecture.

My implementation of PYNQ DMA API is largely inspired by Dr. Peter Ogden's **"Decorator"** design, which provides very clear tutorials on how to construct AXI DMA data transmission architecture on PYNQ.

6. Implementation - Zynq FPGA Side

The traditional workflow for designing FPGA IPs involve designing component modules by hand using RTL. While fine-tuned RTL can usually provide the best performance with minimum resource usage, large hand-written RTL designs usually have low readability and high difficulty in maintenance. Besides, writing RTL requires larger amount of development time, which is not feasible given the time limit. Hence, in this project I have decided to design FPGA IPs using **Vivado High-Level Synthesis (HLS)**.

Vivado HLS accelerates IP creation by enabling C, C++ and System C specifications to be directly targeted into Xilinx^{TM} All Programmable devices without the need to manually create RTL. Supporting both the ISE and Vivado design environments Vivado HLS provides system and design architects alike with a faster path to IP creation.[31]

Since my framework aims at fast prototyping of CNN, the FPGA IP design must be modular and parametrisable, delivering maximum reconfigurability and scalability while requiring minimal user efforts to re-design the network model. Apart from simplicity in prototyping, my hardware design should aim at maximising speed performance, exploiting maximum level of parallelism.

My framework is based on the SDF paradigm. In this paradigm, a network is represented as a directed graph named SDFG. In SDFG, nodes represent computations and lines represent data streams. Fundamentally, the principle of SDF is that whenever input data are available for a node, the node will immediately start processing and generating outputs. When the entire network is constructed using SDF, each component IP on the graph can independently drive the data streaming, forming a heterogeneous streaming architecture. With streaming IO, output data immediately streams out instead of being buffered in on-chip memory, hence saving the memory footprint of the entire network. [5]

So far, my framework provides FPGA IPs including convolution layer, pooling layer, fully-connected layer and ReLU layer. These layers all employ the SDF model of computation as its basis.[8] Complete networks can be constructed as chains of these three layer IPs. With Xilinx^{TM} Vivado's "Block Diagram" utility, engineers can build CNNs graphically.

Figure 6.1 shows how my framework maps LeNet-5, a popular CNN, onto FPGA, using my FPGA layer IPs. Given a CNN topology, engineers will **select** corresponding FPGA IPs from my IP library, **place** them onto one Vivado Block Diagram, **chain** them into one SDFG, and finally **customise** the IPs with appropriate parameters (such as

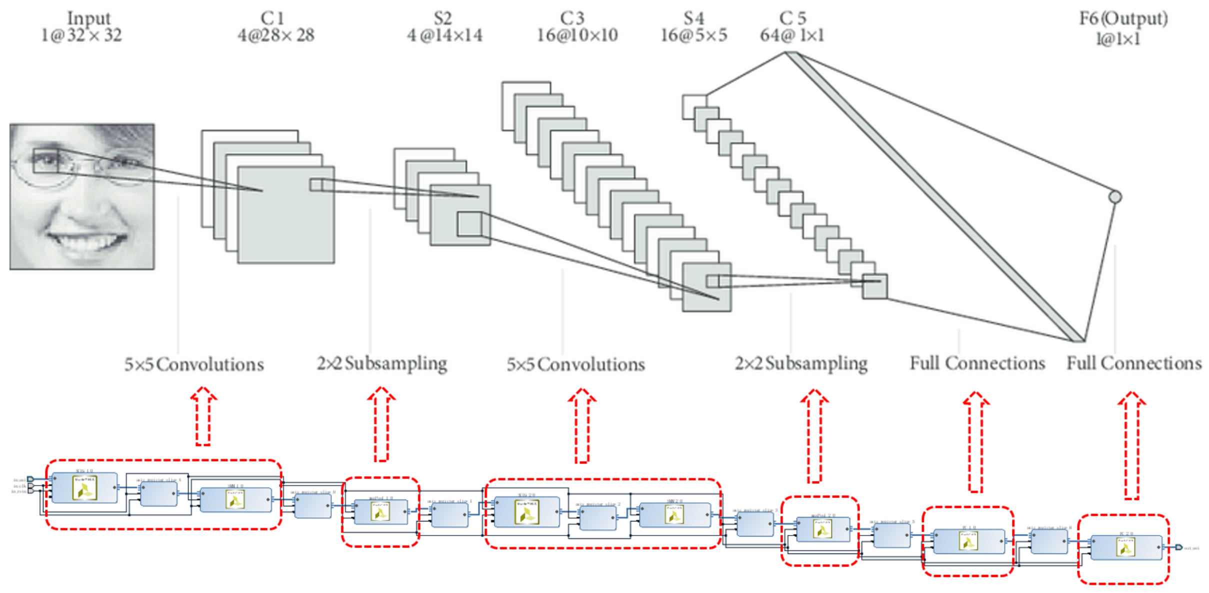


Figure 6.1: Illustration of LeNet-5, showing how my framework deploys CNN on FPGA using SDF. ABOVE: Structure of LeNet-5. BELOW: CNN constructed as a chain of graphical block diagrams in my framework. [32]

kernel dimensions, IO channels, and memory tiling factors).

Technical details of my FPGA IP designs will be explained in this section.

6.1. Data Streaming Architecture

My data I/O uses **AXI4-Streaming interface**, which supports a streaming speed of one word per clock cycle. For each layer IP, at each clock cycle the IP takes in a new word of input feature map, and outputs whenever a new output word is ready. Thus, each node on the SDFG drives the data stream forward independently, forming a heterogeneous computing architecture where all IPs in SDFG process data independently.

Figure 6.2 shows the waveform of AXI4-Streaming control signals. In my design, three control signals are required, namely **TLAST**, **TVALID**, and **TREADY**.

- **TLAST** is active when the last word of the current data stream is being transmitted. In data streaming applications, **TLAST** is required for AXI4 slave to know when the last word of the stream has been transmitted, so that the AXI4 slave knows when to terminate.
- **TVALID** is active whenever an AXI4 master is transmitting valid data to AXI4 slave.
- **TREADY** is active when an AXI4 slave is ready to receive new data from AXI4 master.

Vivado HLS will automatically manage **TVALID** and **TREADY** signals, but designer should manually specify when

TLAST should be active.

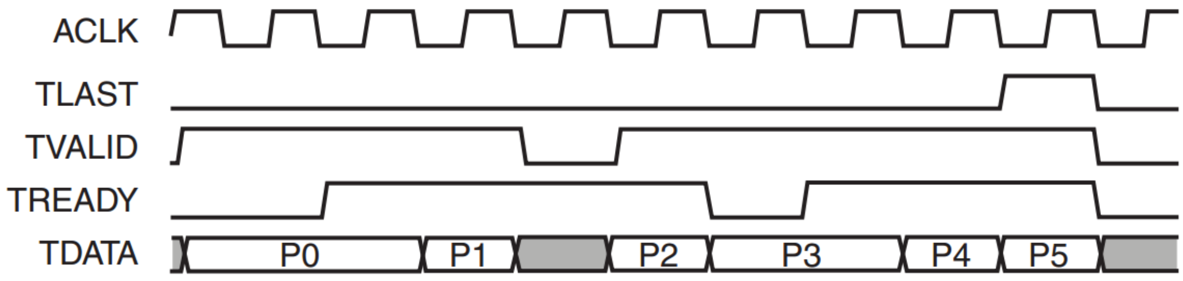


Figure 6.2: AXI4-Streaming Interface Waveform [33]

Since all IPs are connected with the same data stream, the overall data streaming throughput will be dominated by the IP with the slowest throughput. This IP will be known as the throughput **bottleneck**. Bottleneck in a SDF design could either be in data transmission between DDR and FPGA, or in one IP with the lowest data streaming throughput (even lower than data transmission throughput).

6.2. Quantisation

6.2.1. 32-bit Floating-point Format

In the first version of my FPGA layer IPs, I implemented convolution layers using 32-bit floating point data format. However, the resultant hardware resource usage is too large for PYNQ's Zynq FPGA. This section details the resource usage analysis of 32-bit floating-point format.

For resource analysis, I designed a convolution layer using **32-bit floating-point** format, with input feature map of dimension (1,3,32,32) (i.e. 1 3-channel 32x32 image) and kernel of dimension (32,3,5,5) (i.e. 32-output 5x5 kernels) at stride=1 and padding=2 (for implementation details on convolution layer IP, please see Section 6.3). This convolution can be converted into matrix multiplication of size (1024x75)*(75x32), and one forward-propagation of this layer contains 2.4576M MACCs. Table 6.1 shows the resource usage for this layer.

This layer is considered to be a small CNN layer, however from Table 6.1 I can see that the resource usage, in particular BRAM and DSP, is over one-third. Thus, with 32-bit floating point data format, PYNQ will not have enough hardware resources to accommodate more than three such layers on chip, meaning that the design will not accommodate complex CNN models.

6.2.2. Fixed-point Data Quantisation

As compared to 32-bit floating point arithmetic, **fixed-point arithmetic** on FPGA requires significantly less DSP and LUT, and most of the simple fixed-point arithmetic operations can be executed in one clock cycle. However, there is a trade-off relationship between the fixed-point quantisation bitwidth and classification accuracy, since

Resource	Usage	Available	Percentage Usage (%)
BRAM	54	140	38.6
DSP48E	76	220	34.5
LUT	1803	53200	3.39

Table 6.1: Floating point convolution layer resource usage comparison $((1,3,32,32) \circledast (32,3,5,5)$, generating $(1,32,32,32)$ output)

data quantisation results in data precision loss and consequently classification accuracy loss. In order to explore the optimal data format, I implemented the same convolution model mentioned in previous section, with floating-point and different bitwidths of fixed-point data formats, and compares their resource usage. For accuracies of various data formats, I used Philipp Gysel, Mohammad Motamedi and Soheil Ghiasi's published classification accuracies. Figure 6.3 show this trade-off relationship between FPGA DSP48E usage and classification accuracies both in LeNet-5 and CIFAR-10.

For LeNet-5, from Figure 6.3 (left), it can be concluded that **8-bit** and **16-bit** fixed-point data formats provide the most optimal resource-accuracy trade-off. In order to reduce BRAM usage, 8-bit instead of 16-bit fixed point format will be used for LeNet-5 deployment. For CIFAR-10, from Figure 6.3 (right), it can be concluded that 16-bit fixed-point data formats provide optimal resource-accuracy trade-off. Hence, 16-bit fixed-point data format will be used to implement CIFAR-10 model.

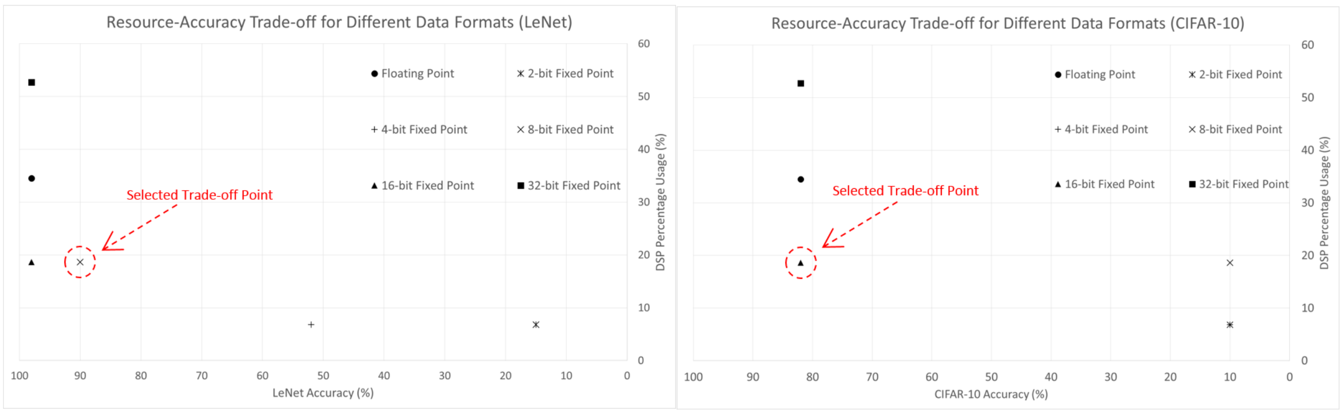


Figure 6.3: Trade-off between DSP usage and classification accuracies, with different data formats. Left: LeNet-5. Right: CIFAR-10.

6.3. FPGA Layer IPs

Three CNN layers, namely convolution layer, pooling layer, and fully-connected layer, have been constructed using Vivado HLS. ReLU layer was included in each of the three layers, as each layer IP has the option of whether to directly rectify each output data as the output feature map streams out.

6.3.1. Convolution Layer

The convolution between the input feature map X and the kernel W can be achieved by expanding the feature map into a large 2D matrix, M,, followed by matrix multiplication between M and unrolled 2D version of M. The first stage where X unrolls into M is named **"im2col"**, or in the streaming interface, **"sliding window"**.

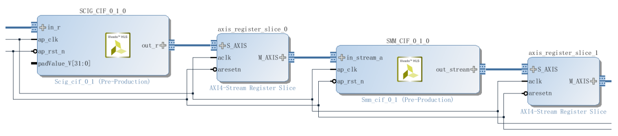


Figure 6.4: Block design of sliding window unit IP followed by matrix multiplication IP. AXIS register slices are placed to improve timing

Sliding Window Unit: As the name suggests, **"im2col"** expands the input feature map into column vectors, where each column vector contains the patch of input feature map which will perform dot-product with kernel. Although this operation essentially expands the memory usage of input feature map by replicating data, this operation has effectively transformed convolution into matrix multiplication. Matrix multiplication has many efficient implementations (for example, for CPU I have Basic Linear Algebra Subprograms (BLAS) API, and for FPGA Vivado HLS has introduced optimisations to implement matrix multiplications[34]), which I can take advantage of. The Sliding Window Unit has been implemented in Algorithm I.

Algorithm I Sliding Window with Streaming Data Interface

```
Read layer parameters contained in the first several data words
if Status is "Load Weight" then
    Pass input data stream to the subsequent layer
else {Status is "Deploy"}
    Stream in the input feature map and stream out subspaces of input feature map
end if
```

With AXI4-Streaming interface, **"im2col's"** memory footprint problem can be effectively eliminated. The outputs of **"im2col"** (i.e. **"sliding window"**[6][5]) block can immediately stream out to the subsequent block in SDFG, hence there is no need for buffering the output feature map.

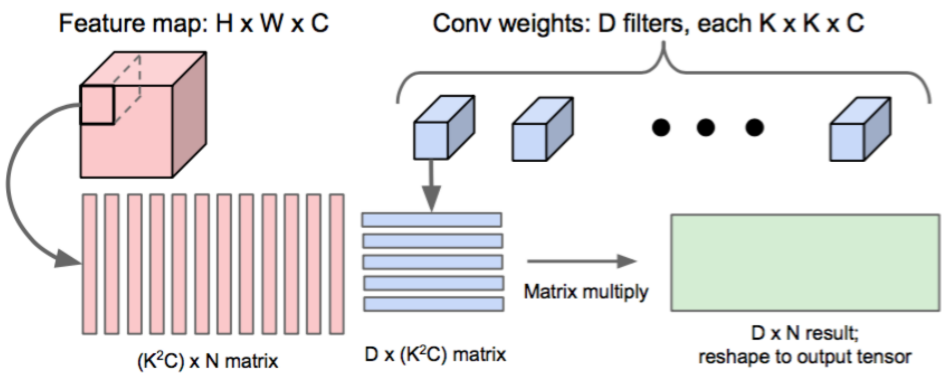


Figure 6.5: Illustration of "im2col" in action [35]

Figure 6.5 shows how the sliding window block expands the input feature map. Based on the specifications of the CNN layer (specifically, kernel dimension, padding dimension and stride), sliding window block generates column vectors which correspond to patches of input feature map which will perform dot-product with kernel. Note that up to the data of report, the framework only supports a stride of 1. In the future the framework will be updated to support more striding values. My implementation of sliding window unit is based on the sliding window unit function provided in Xilinx's BNN-PYNQ reference design. [6]

INTERLEAVING: From Figure 6.5, I can see that each column vector contains multiple channels of input feature map. Since addition is commutative, the order in column vector does not affect the output, as long as it corresponds to the unrolling order in kernel. Hence, in order to reduce buffer usage, the framework "interleaves" the weights off-line. Interleaving of input feature map will be performed on FPGA IP on-the-fly with no additional latency cost. [6]

Figure 6.6 shows how interleaving is performed. By default, input feature maps' dimensions are arranged in **(batch size, input channels, height, weight)** order, and kernels' dimensions are arranged in **(output channels, input channels, height, weight)** order. In both arrangements, input channels come before height and weight, and hence if I unravel these matrices into data streams, almost the entire feature map needs to be buffered before the buffer contains one entire patch. On the other hand, with interleaving, I transpose these dimensions such that input channel becomes the last dimension. In this way, after unravelling, data will be streaming across channels. This means that a buffer of size at most $K * W * C$ is required, significantly reducing buffer memory usage.

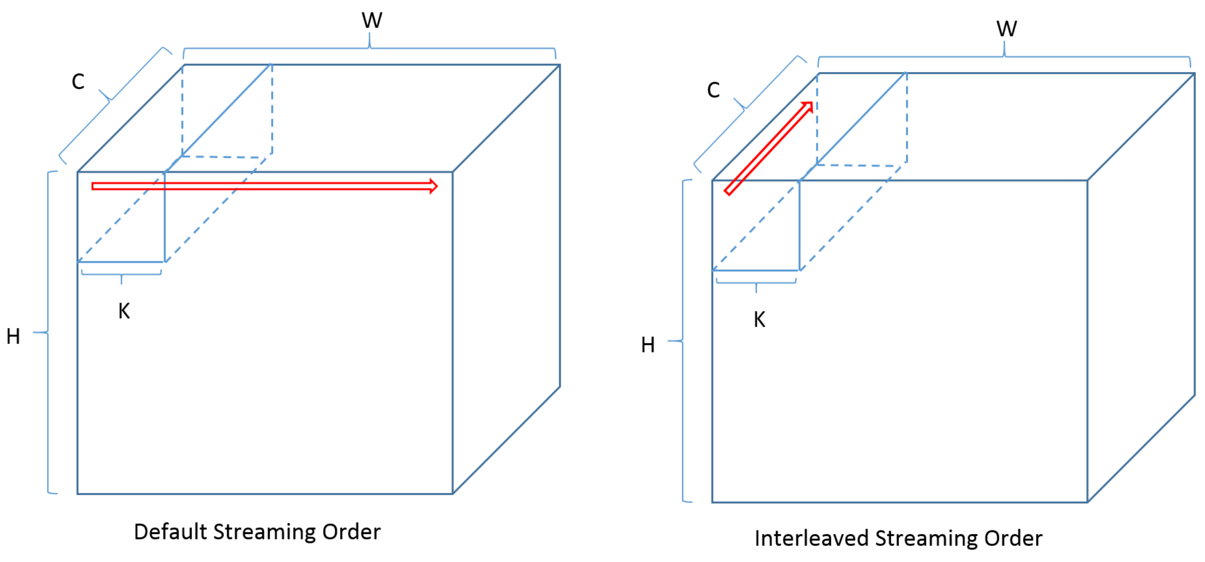


Figure 6.6: Effect of "interleave" operation on input feature map, where "H", "W", "C" and "K" refers to height, width, channel and kernel dimension respectively. The red arrow indicates the fundamental data streaming order, showing that with interleaving the amount of data that needs to buffer reduces.

MATRIX MULTIPLICATION: The second stage of convolution layer is matrix multiplication. Xilinx^{TM} Vivado HLS has extensive support for implementing AXI4-Streaming matrix multiplications on Zynq FPGA. As shown in Figure 6.5, in matrix multiplication $A * B = C$, each value in C is generated by dot-product between a row of A and a column of B. This dot-product can be implemented in parallel on FPGA architecture, and pipelined so that one output value can be generated per clock cycle. The algorithm for the matrix multiplication layer IP is shown in Algorithm II.

Algorithm II Matrix Multiplication with Streaming Data Interface

```
Read layer parameters contained in the first several data words
if Status is "Load Weight" & Target Layer ID Matches Current Layer ID then
    Instantiate BRAM for weight storage
    Write weights into BRAM
else if Status is "Load Weight" & Target Layer ID Does Not Match Current Layer ID then
    Pass input data stream to the subsequent layer
else {Status is "Deploy"}
    Load one row of input feature map into input buffer
    Compute dot-product between input buffer and each row of weight buffer in pipeline II=1
end if
```

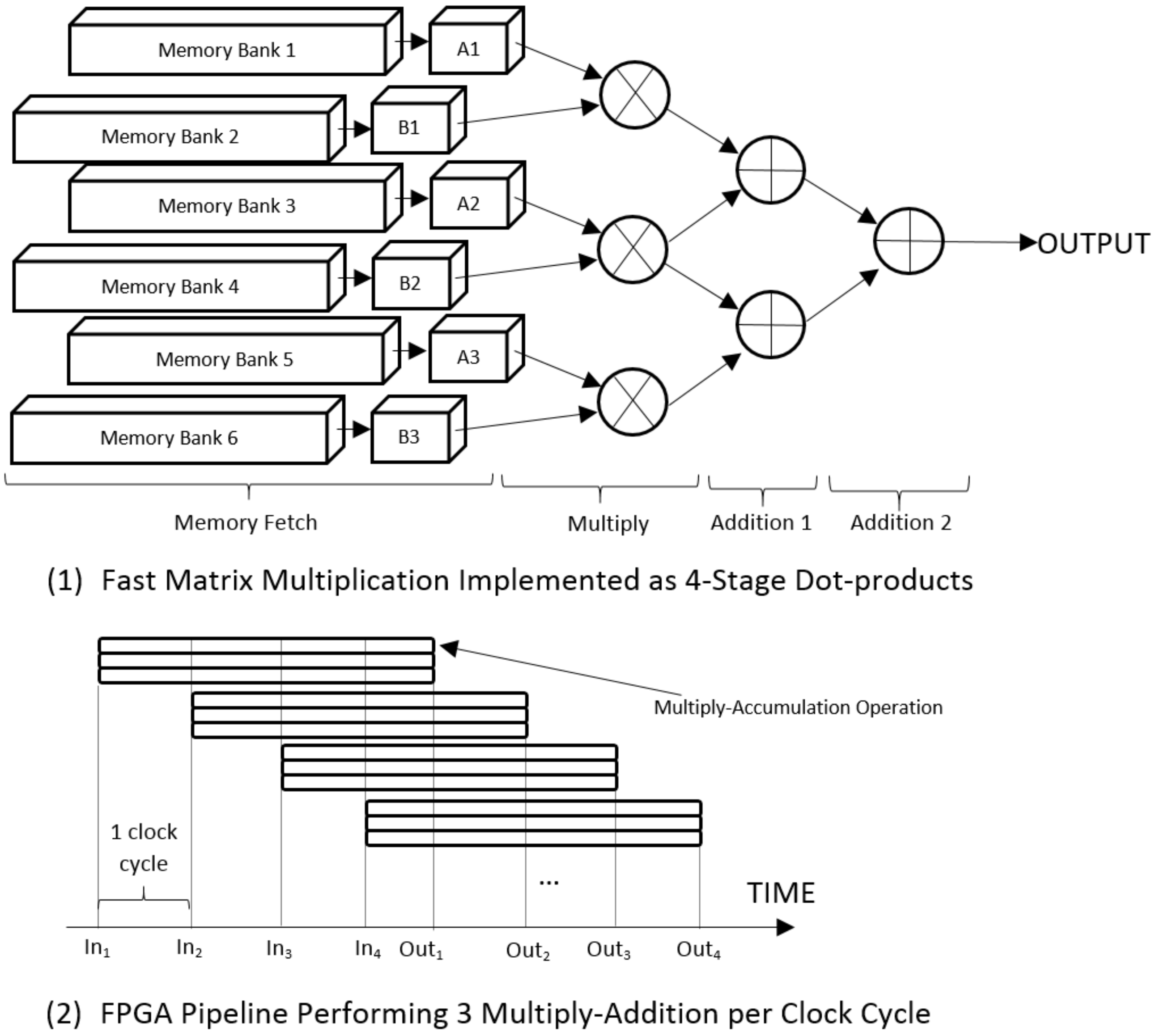


Figure 6.7: Illustration of how FPGA can accelerate matrix multiplication to high throughput

Figure 6.7 explains how FPGA accelerates matrix multiplication by **parallel computing** and **pipelining**. Assuming that the matrix multiplication is of dimension $(M * 3) * (3 * N)$. Figure 6.7 (1) shows the digital hardware generated on FPGA. The memory banks (1, 3, 5) store the input feature map and the memory banks (2, 4, 6) store the weights. Both input feature map and weight has been partitioned into smaller memory banks so that multiple input data can be fetched in parallel. At each clock cycle, three new pairs of operands for dot-product are fetched into registers A1-3 and B1-3, and computed in parallel using a tree of multipliers and adders. The computation process consists of many register stages, and instead of waiting for the current computation to finish, memory fetching continues at the next clock cycle, creating a data processing pipeline (as shown in Figure 6.7 (2)). Hence, output data will be generated at high throughput, leading to high overall data processing speed.

Figure 6.7 (1) also illustrates that since each BRAM bank has limited memory ports, in order to simultaneously fetch multiple operands, input data must be stored in different memory banks. Otherwise, extra latency is required for memory banks to fetch multiple data, leading to pipeline stalling. For details of trade-off analysis between memory bank partitioning, speed and resource usage, please see Section 6.6.

6.3.2. Pooling Layer

Pooling layer performs down-sampling on input feature map, by outputting maximum or average of sub-graphs. In AXI4-Streaming interface, pooling can be performed in similar concepts to sliding window unit. My implementation of pooling layer is based on the pooling layer design in Xilinx's BNN-PYNQ reference design, which provides a maximum throughput of one output word per clock cycle. [6] My max-pooling layer IP implementation is shown in Algorithm III.

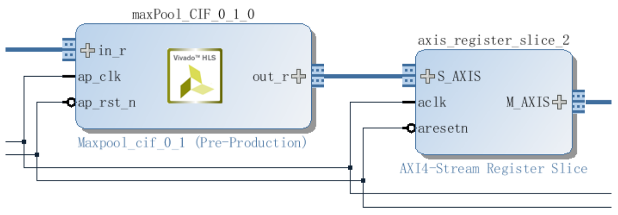


Figure 6.8: Block design of pooling layer IP. AXIS register slices are placed to improve timing

Algorithm III Max Pooling with Streaming Data Interface

```
Read layer parameters contained in the first several data words
if Status is "Load Weight" then
    Pass input data stream to the subsequent layer
else {Status is "Deploy"}
    Stream in the input feature map and stream out the maximum of each subspace of input feature map
end if
```

6.3.3. Fully-connected Layer

Fully-connected layer performs dot-product between input feature map and weights. Hence, in the off-line, a fully-connected layer can be easily converted into convolution layer.

For example, consider an input feature map of dimension $(512 * 7 * 7)$, which will go through a fully-connected layer with 4096 outputs. The weights of this fully-connected layer, which have dimension $(4096 * 25088)$, can be reshaped into $(4096 * 512 * 7 * 7)$, forming a convolution layer with kernel dimension = 7, input channel = 512, stride = 1, and zero-padding = 0.[9]

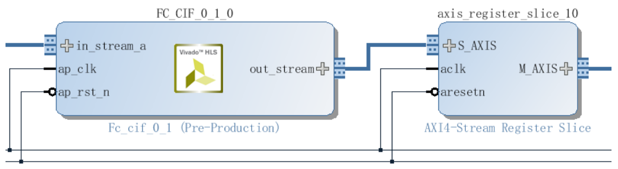


Figure 6.9: Block design of fully-connected layer IP. AXIS register slices are placed to improve timing

Since a fully-connected layer can be converted into convolution layer, my framework does not have IPs specifically for fully-connected layer. The algorithm of fully-connected layer IP is identical to matrix multiplication as shown in Algorithm II.

6.4. Weight Storage and Runtime Reconfigurability

A great deal of works exist in mapping CNN onto FPGA platforms, and these architectures have different memory usage schemes. Works that are based on a single processing engine usually only stores one layer's weights on BRAM at each time. After that, weights will be reloaded for the subsequent layers.[17][11]. However, for SDF-based architecture, all layers' weights should be stored on BRAM at the same time. Otherwise, if any layer stalls for memory operations, the entire data stream stalls, reducing overall performance.[5][6]

In order to simplify the CNN deploy efforts, my framework allows engineers to **reload weights of any layer at runtime**. As shown in Algorithm II, before deployment, weights of each layer can be loaded and reloaded by asserting the "Load Weight" status flag. This allows one FPGA CNN model to switch to perform multiple classification tasks at runtime, by simply reloading weights.

Apart from reloading weights in the runtime, my framework can even allow one layer to **reconfigure to different dimensions** when reloading weights. When weight reloads, my framework will transmit data stream which starts with dimension parameters of the target layer, followed by weights data. By changing the dimension parameters transmitted, the layer will reconfigure and change dimension. Figure 6.10 shows the format for data stream that reloads weights. This data stream contains the ID for target layer, and it will be transmitted throughout the entire SDFG. Layers with different ID will ignore the stream, and the target layer will reconfigure its dimensions based on the parameters carried at the start of the stream, and reload its weights with the new weights. Details of each parameter have been listed below.

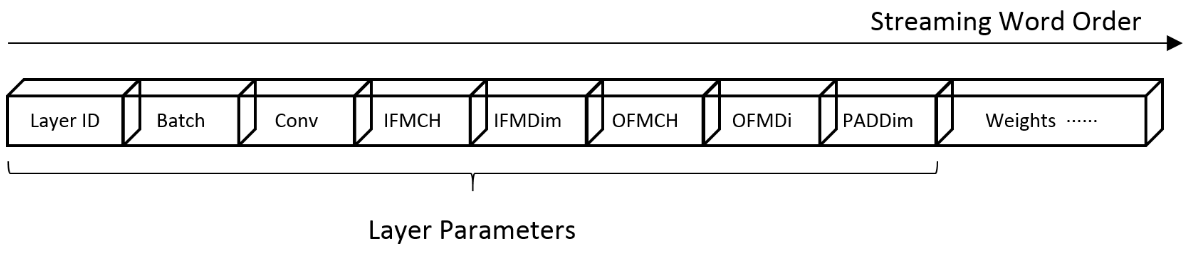


Figure 6.10: Format of data stream, where the first 8 words are parameters of the target layer

- **Layer ID** is the unique ID that engineer needs to assign to each convolution or fully-connected layer (i.e. layers with weights). At runtime weight reloading, layer ID is used to identify the target layer based on target layer ID. A layer ID of 0 means the current data stream is input feature map, and the system is in deployment mode.
- **Batch** refers to input image batch size. Many applications of CNN, such as Regional CNN (R-CNN) for object detection, require the processing of one batch of images per execution.
- **Conv** means convolution kernel dimension.

- **IFMCH** is the input feature map channel.
- **IFMDim** is the input feature map dimension.
- **OFMCH** is the output feature map channel.
- **OFMDim** is the output feature map dimension.
- **PADDim** is the padding dimension

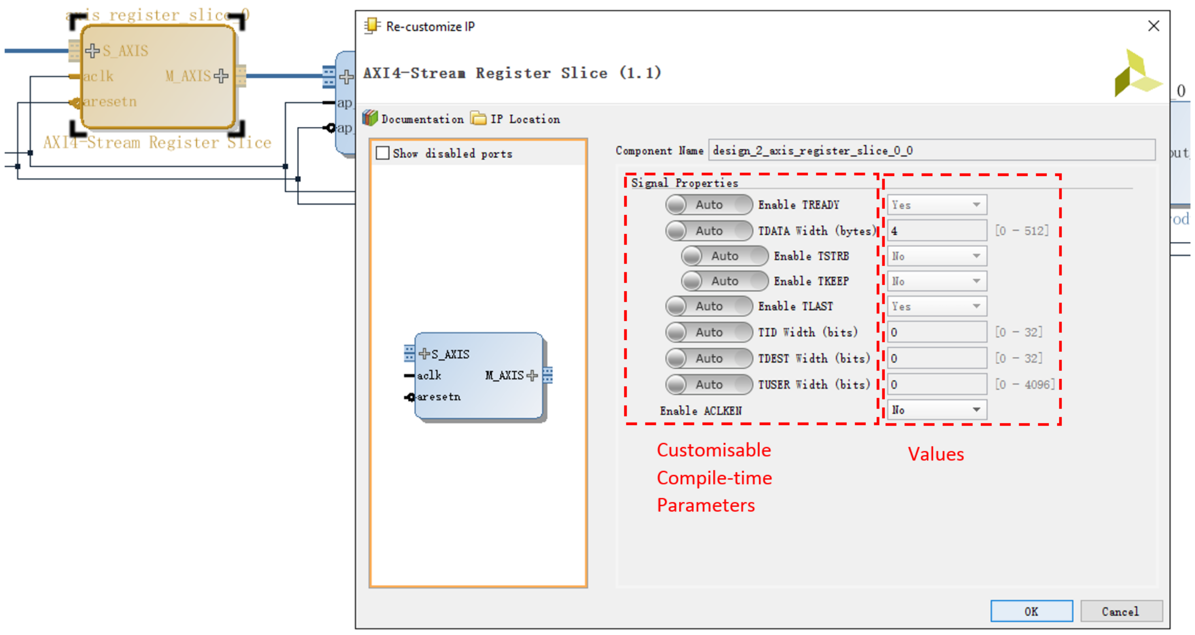
All these parameters can be used to reconfigure the layer dimensions during weight reloading. Note that the value of each parameter must not exceed the maximum memory size declared in compile-time parameters (See next section for details).

6.5. Compile-time Parametrisable IP Design

I aim to design my layer IP library such that engineers can apply my IP blocks to different CNN specifications, by simply changing block generic parameters before Vivado synthesis. Generic parameters are RTL macro parameters that can fine-tune and control the FPGA hardware design and resource usage before Vivado synthesis. I designed my FPGA layer IPs such that engineers can effectively fine-tune the IP resource usage and parallelism with only a small number of generic parameters, which can be easily defined either in a design header file or in Vivado Block Design **"Customise Block"** panel. Details of compile-time parameters are listed below.

- **BITWIDTH** defines the number of bits used for fixed-point data quantisation.
- **Layer ID** defines the unique ID assigned to the current layer. Note that the layer ID 0 has been reserved as flag to indicate the input data stream in deployment mode.
- **FACTOR** refers to the folding factor, which controls the trade-off between the IP's level of parallelism and on-chip resource usage.
- **OUTPUT_RECTIFY** is a switch that enables rectification at the output of current layer.
- **ROW_MAX and COL_MAX** define the on-chip memory size declared for weight storage. It is the maximum size of weights that can be stored in current layer.
- **POOL_SIZE** defines the filter size of pooling layer.
- **POOL_MODE** is a switch that toggles between "max-pooling" and "average-pooling".

My initial plan was to construct FPGA layer IPs in customisable block designs, where engineers can fine-tune the IP parameters on the block design GUI provided by Xilinx^{TM} Vivado. However, currently Vivado has not provided support for designing user-customisable block design using HLS yet. So far, only project written in full HDL can be packaged into customisable block design. Thus, in the current version of the framework, I can only parametrise IPs using a header file.



Component Name design_2_axis_register_slice_0_0

Signal Properties

Auto Enable TREADY	Yes	
Auto TDATA Width (bytes)	4	[0 – 512]
Auto Enable TSTRB	No	
Auto Enable TKEEP	No	
Auto Enable TLAST	Yes	
Auto TID Width (bits)	0	[0 – 32]
Auto TDEST Width (bits)	0	[0 – 32]
Auto TUSER Width (bits)	0	[0 – 4096]
Enable ACLKEN	No	

Figure 6.11: Illustration of concept for IP block design GUI

Figure 6.11 shows the "AXI4-Streaming Register Slice" block design customisation panel, where users can parametrise the hardware implementation of this IP using such panel. This illustrates the same concept as my proposed IP block design GUI. I hope that in the future I can use similar interface for my framework's IPs.

6.6. Folding

Section 6.3.1 mentioned that due to limit in BRAM memory ports, weights must be stored in multiple BRAM banks in order to fetch multiple data simultaneously, suggesting that the more BRAM banks I instantiate on-chip, the more MACCs will be processed in parallel (and consequently using more DSPs). The instantiation of BRAN banks can be controlled by changing the folding of matrix multiplication. Hence, by varying the folding factor there exists a trade-off between resource usage and speed.

In Xilinx^{TM} Vivado HLS, memory bank partitioning is implemented via **directive** statements. Directives provide extra hardware requirements to the compiler, allowing the creation of specific high-performance hardware implementations. [31]. The Listing 6.6 shows how the folded matrix multiplication is implemented in Vivado HLS.

In Listing 6.6, each column of input streams in and is stored in a temporary LUT-based memory A, while weights are stored in BRAM-based memory B. Both A and B are partitioned into **k** folds, where **k** is the folding factor. For each column of A and for each column of B, output is calculated as the dot-product between these two columns. Each dot-product operation is folded into **k** folds, where each fold is computed in parallel, and all folds are computed in complete pipeline (as explained in Setion 6.3.1).

Listing 3: Folding Matrix Multiplication

```
//////////////////////////////////////////////////////////
// BRAM Instantiation
static ap_int<wordwidth> A[A_COL_MAX][A_ROW_MAX], B[B_COL_MAX][B_ROW_MAX];
#pragma HLS RESOURCE variable=A core=RAM_S2P_LUTRAM
#pragma HLS RESOURCE variable=B core=RAM_S2P_BRAM
int const FACTOR = 5; // Folding Factor
// Memory objects A and B partitioned along dimension 2
#pragma HLS array_partition variable=A block factor=FACTOR dim=2
#pragma HLS array_partition variable=B block factor=FACTOR dim=2
// Matrix multiplication, A*B
L1:for (int ia = 0; ia < A_COL_MAX; ++ia)
{
   L2:for (int ib = 0; ib < B_COL; ++ib)
   {
      ap_int<bitwidth> sum = 0;
      L3:for(int ic = 0; ic < B_ROW_MAX/FACTOR; ++ic){
#pragma HLS PIPELINE II=1 // Pipelining at II=1
         L4:for(int id = 0; id < FACTOR; ++id){
            sum += A[ia][id*B_ROW_MAX/FACTOR+ic] *
               B[ib][id*B_ROW_MAX/FACTOR+ic];
         }
      }
      valOut.data = sum;
      if (ia+iter*A_COL_MAX==A_COL-1 && ib==B_COL-1 && num_imag==batch_size-1)
         valOut.last = 1;
      else valOut.last = 0;
      out_stream.write(valOut);
   }
}
```

When folding factor increases, although more MACC operations are processed in parallel, more BRAMs and DSPs are required as well. Hence, a trade-off between resource usage and speed exists when different folding factors. Figure 6.12 shows this resource-latency trade-off, by comparing the latency and resource usage of a $(144 * 500) * (500 * 50)$ matrix multiplication using different folding factors. As shown in Figure 6.12, the folding factor can effectively fine-tune the resource usage and speed performance across a very large dynamic range, providing users with maximum freedom in customising the FPGA CNN implementation.

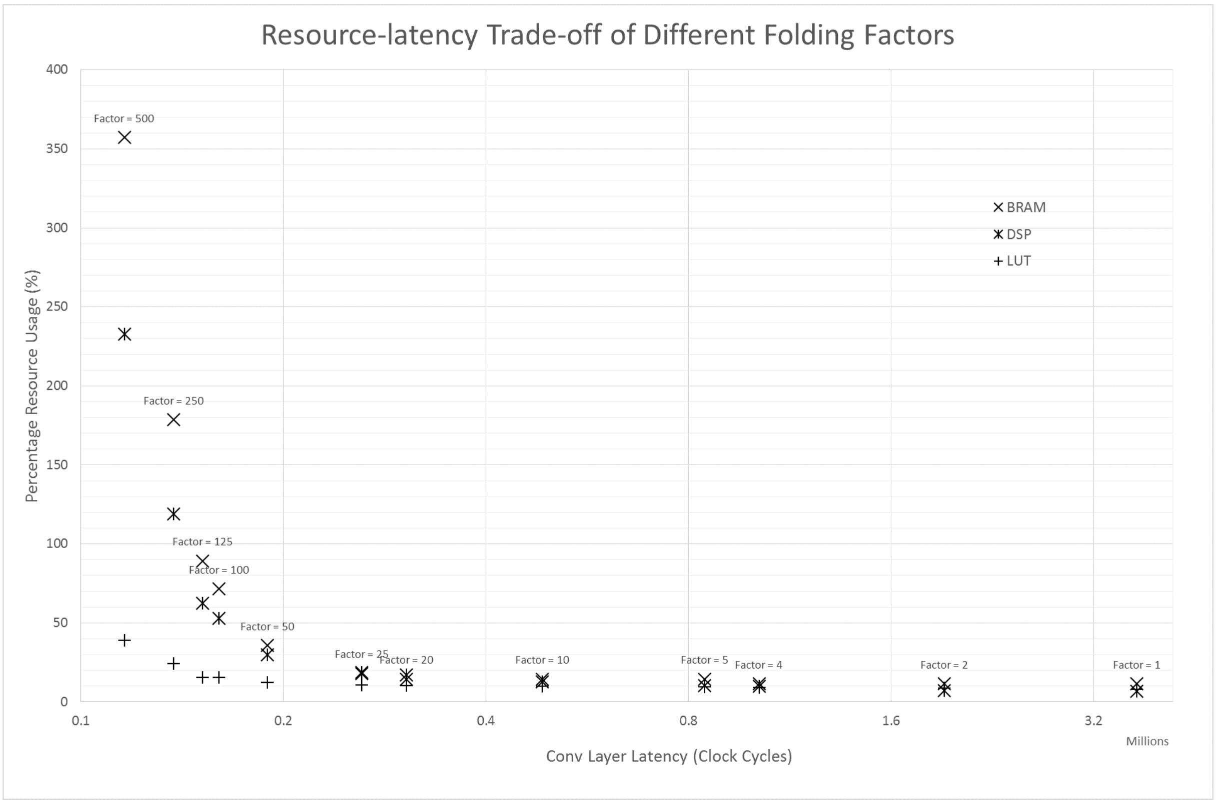


Figure 6.12: Trade-off between convolution layer latency and resource usage, with different folding factors

6.7. Timing Constraints

In Xilinx^{TM} Vivado, when I design a pipeline chain of IPs interconnected with AXI4-Streaming interface, the transition of many control signals are synthesised as asynchronous signals. This means that a long critical path, with its length approximately proportional to the number of IP blocks along the design pipeline, has been created. The result of this long critical path is a timing constraint violation. In order to remove this timing violation, **AXI4-Streaming Register Slice** has been inserted between adjacent IPs in the data stream. AXI4-Streaming Register Slice creates timing isolation between an AXI4-Streaming master and slave by inserting registered buffers of depth 2. Figures 6.4, 6.8 and 6.9 show how Register Slice IP has been inserted in the block design.

6.8. SDF Subgraphs for Larger CNNs

So far, I have only considered implementing a complete SDFG on FPGA in one go. However, the scalability of such implementations will eventually be limited by the amount of on-chip resources. In order to further expand the FPGA implementation's scalability, I also explored the possibility of partitioning a SDFG into SDF subgraphs, and implementing each subgraph at a time, using the FPGA's reconfigurability. This method was originally proposed by fpgaConvNet[5]. Below lists some advantages and disadvantages of such implementations.

- Advantages

  - **Scalability:** Theoretically, by partitioning the SDFG into SDF subgraphs, there is no upper limit in the complexity of CNN models to be implemented.
  - **Parallelism:** The FPGA implementation can focus on completely parallelising one partition at a time, hence increasing the operation parallelism and throughput.

- Disadvantages
  - **Reconfiguration Time:** Inevitably, such implementations will incur extra FPGA reconfiguration latency. The more partitions I make, the more reconfiguration latency.
  - **Less Feasible for Real-time Applications:** With such implementation, the increase in parallelism will only outweigh the reconfiguration latency when the batch size (i.e. number of images processed at one forward propagation) is large enough. For real-time applications, however, the input batch size is likely to be quite low, for frame rate requirements.
  - **Memory Transfer Latency:** Between adjacent SDF subgraphs, feature maps need to be transmitted to and from the DDR memory for temporary storage, resulting in extra latency.
  - **Memory Footprint:** As mentioned in the previous point, feature maps need to be temporarily stored in DDR memory. In the middle of CNN, the size of feature map expands significantly. Since the PYNQ DMA driver specifies that the maximum buffer size on DDR is around 8.4MB, the maximum batch size is limited to a small value, meaning that the design will not be able to implement applications requiring large input batch size, such as Regional CNN (R-CNN).

In order to evaluate the performance of SDF subgraph implementation, I implemented a prototype using such implementation. For performance evaluation of SDF subgraph prototype, please see Section 8.3.

7. TESTING

This section describes two testing environments that I designed for the project, namely validation setup and actual experimental setup.

7.1. VALIDATION

My project includes two stages of validation. The first stage is testing Vivado HLS C++ source using a C++ testbench project. The second stage is testing the HLS generated RTL source using simulation in Xilinx^{TM} Vivado.

In order to progressively validate my framework, I firstly implemented testbench for single CNN layers, followed by testbench for entire CNN SDFG. Towards the end of the project, I also implemented testbench for implementing SDF subgraph (a portion of SDFG with multiple CNN layers).

Xilinx^{TM} Vivado HLS allows users to design C++ simulation testbenches and use them to test and debug the HLS source codes. After each layer's HLS source code had been designed in Vivado HLS as a C++ function, I designed appropriate software testbenches in C++, which interfaces with the layer IP function and tests for its correctness. The testbench needs to be able to randomly generate input feature maps, as well as correct reference outputs (sometimes known as "Golden Data") which checks my IP output's correctness. In each test, 500 random input data has been generated and tested with my IP source code, such that a wide range of corner cases have been tested, ensuring functional reliability of the IP.

After my IPs passed the C++ simulation testbench, they will be packaged as **Block Designs**. These block designs will be instantiated in Xilinx^{TM} Vivado, and a HDL simulation testbench simulating AXI4-Streaming I/O feature maps has been prepared. There, the testbench validates the HDL generated from Vivado HLS, making sure that the AXI4-Streaming I/O signals are correctly created, and the HDL behaves as expected.

7.2. Experimental Setup

After relevant IPs are designed and validated, I used Xilinx^{TM} Vivado to synthesise bitstream, which was downloaded to PYNQ via Jupyter Notebook. Finally, my framework's Lasagne customised layer links with the bitstream, and the FPGA IP design is ready for deployment.

In order to design the testing environment for CNN deployment, I modified a tutorial provided by Lasagne Recipes, which guides users into implementing a CNN model that classifies CIFAR-10 image dataset. This tutorial provides CNN model specifications, pre-trained parameters, and testing images and labels. To test my framework, I simply changed the script from calling Theano built-in CNN layer to calling my framework's customised layer, allowing me to quickly implement my FPGA-accelerated CNN, validate output correctness, and benchmark design performances.

This report provides performance analysis of two forms of CNN deployments, namely complete SDFG implementation and SDF subgraph implementation. A number of prototypes have been created to demonstrate the FPGA acceleration of CNN deployment. In order to compare with related works, performance benchmarks including MNIST hand-writing digit dataset[36] and CIFAR-10 image classification dataset[37] have been used as testing inputs. Popular CNN models such as LeNet-5[4] and CIFAR-10 (Caffe "quick" version[38]) were implemented with my framework and used in performance analysis.

Some important technical specifications of the experimental setup have been listed in Table 7.1. Speed-up has been measured between ARM CPU execution speed and my FPGA execution speed. For a detailed performance evaluation, please see the next section.

	Dual Core ARM Cortex-A9	Zynq XC7Z020
Clock Frequency	0.8 GHz to 2 GHz	100 MHz
Number of processor cores	2	NA
Memory size	512 MB	4.9 MB (140 36KB BRAMs)

Table 7.1: Technical specifications of the hardware used for performance measurements

8. Performance Evaluation

This section evaluates three CNN prototypes implemented using my framework. The three prototypes are **complete SDFG implementation of LeNet-5 for MNIST hand-writing dataset**, **complete SDFG implementation of CIFAR-10 (Caffe "quick" version) for CIFAR-10 image classification benchmark**, and **SDF subgraph implementation of Network in Network (NIN) for CIFAR-10 image classification benchmark**. For LeNet-5 model topology in Lasagne syntax, please see Appendix C.

8.1. Complete SDFG Implementation - LeNet-5

In this section, LeNet-5 has been completely implemented on FPGA using my framework. It will be used to classify MNIST hand-writing digit dataset. Its performance will be benchmarked against ARM CPU performance using Theano, as well as other related works on GPU and FPGA.

Using parameters provided by my framework, my LeNet-5 implementation has been hand-tuned for a balance between parallelism and on-chip resource limitation. Table 8.1 lists the optimal compile-time parameters selected for each tunable layer.

Layer	ID	Maximum Weight Height	Maximum Weight Width	Folding Factor
CONV_1	1	20	25	25
CONV_2	2	50	500	20
FC_1	3	500	800	25
FC_2	4	10	500	20

Table 8.1: Parameters hand-tuned for optimal LeNet-5 deployment

8.1.1. Resource Usage

Table 8.2 lists the FPGA on-chip resource usage for LeNet-5 deployment. It can be observed that the BRAM usage has approached maximum capacity, showing that it has been fine-tuned for maximum resource usage, and this framework will struggle with fitting larger CNN designs on board in complete SDFG. Hence, this framework may have problem with the scalability of CNN model implementation, and Section 8.3 will propose SDF subgraph implementation, which aims to solve this issue.

Resource	Usage	Available	Percentage Usage (%)
BRAM	139	140	99
DSP48E	130	220	59
LUT	38304	53200	72

Table 8.2: Resource Usage of LeNet-5 deployment as complete SDFG

8.1.2. Theoretical Maximum Throughput

From Section 6.6, I know that the maximum number of parallel MACC operations per layer is equal to the layer's folding factor. Based on the folding factors provided in Table 8.1, I can estimate the maximum possible parallel MACC operations per second.

$$Throughput = \sum_{i}^{layers} folding\ factor_i * clock\ frequency \tag{8.1}$$

$$= (25 + 20 + 25 + 20) * 100M \tag{8.2}$$

$$= 9\ GOP/s \tag{8.3}$$

Thus, without considering the data stream scheduling, the maximum theoretical throughput that can be achieved by my design should be 9 GOP/s. This is assuming that all MACC operators are computing at the same time. However, in reality, since each layer has different throughput, the backward pressure will stall the entire data stream, causing the actual performance to be worse than this.

8.1.3. Actual Throughput and Bottlenecks

I benchmark my framework's LeNet-5 deployment latency against ARM CPU latency using Theano with Lasagne, by timing the total CNN prototype deployment latency, including Python script interpreter initialisation time, FPGA memory transfer time and FPGA execution time. Figure 8.1 plots their latency against input image batch size.

In general, my framework shows a sustained 30x speed up as compared to ARM CPU performance. Both plots show upward slope. This is because with small batch sizes, the constant initialisation time (including Python interpreter initialisation time) dominates the overall latency, whereas with bigger batch sizes, the increasing execution latency dominates the overall latency. Hence, the larger the batch size, the higher the deployment throughput.

Based on the latency plot in Figure 8.1, I can estimate the sustainable throughput of my framework. From published analysis reports on LeNet-5, I know that for each input image, LeNet-5 performs 0.0038 GOPs. [8][4]. From Figure 8.1, I know that my framework can process at a sustained frame rate of 493.9 images per second. Thus, the maximum sustainable throughput for my framework's deployment of LeNet-5 is

$$0.0038\ GOP/image * 493.9\ images/s = 1.877\ GOP/s$$

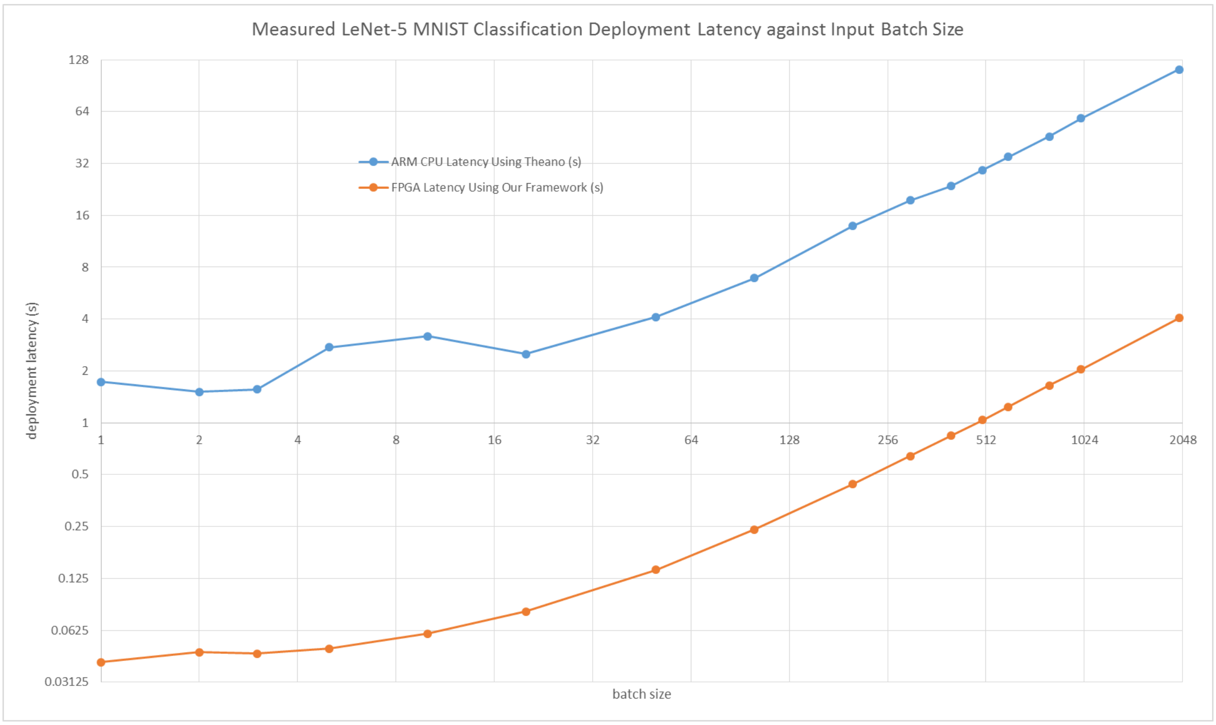


Figure 8.1: Plot of deployment latency for CPU using Theano and FPGA using my framework, against input batch size

It is important to note that the theoretical throughput is significantly higher than actual throughput that I have obtained in my experiment results. This is because the entire data stream has been bottlenecked by the **FC_1** layer. In deployment, **FC_1** layer continuously takes in 800 input data, then halts the entire data stream for about (800/25*500 = 16000) clock cycles, and the process repeats. Thus, this layer produces the greatest backward pressure on the data stream, dominating the overall throughput.

Having shown that my framework provides good acceleration compared to ARM CPU performance, I also compared my performance with other published performances on MNIST hand-writing dataset classification, using Zynq FPGA. Table 8.3 compares my framework's performance against some published LeNet performance benchmarks on FPGA platforms of similar capacity.

From Table 8.3, it can be shown that my framework's LeNet-5 deployment performance is at the same scale with some published works, and my framework delivers state of the art performance.

8.1.4. Power Efficiency

For embedded system with limited power supply, it is important to minimise the energy spent per image classification. With a frame rate of 493.9 frames per second, and an average power consumption of 1.895 W, my implementation of LeNet-5 achieves an energy usage of 493.9 $frames/s/1.895\ J/s = 260.6\ frames/J$.

Design	Platform	Clock Frequency	Sustained Throughput (GOP/s)
Sina Ghaffari[39] & Saeed Sharifian	Kintex-7 XC7K325T	150 MHz	0.216
fpgaConvNet (2016)[5]	Zynq-7000 XC7Z020 System on a Chip (SoC)	100 MHz	0.48
Magnus Halvorsen[40]	Zynq-7000 XC7Z020 SoC	100 MHz	0.988
My Framework	Zynq-7000 XC7Z020 SoC	100 MHz	1.877
BNN on PYNQ[41]	Zynq-7000 XC7Z020 SoC	100 MHz	32.76

Table 8.3: Comparison to other published performance benchmarks on LeNet-5 throughput

8.1.5. Accuracy

In this LeNet-5 prototype, I implemented a fixed-point 8-bit quantisation in order to optimise on-chip resource usage, which has resulted in some classification accuracy loss.

I implemented the LeNet-5 network to classify the testing dataset of MNIST hand-writing dataset, using a pre-trained model. The testing dataset of MNIST hand-writing dataset contains 10000 hand-writing images which are different from MNIST training dataset. With the MNIST testing dataset, the pretrained model can achieve an accuracy of 99.11% using 32-bit floating-point format. Figure 8.2 shows that my FPGA implementation introduces an accuracy loss of 0.24% due to 8-bit fixed-point quantisation.

This level of accuracy loss is reasonable considering that the FPGA implementation also achieved a sustained 30x acceleration as compared to ARM CPU implementation.

Figure 8.3 and Figure 8.4 show examples of correct and wrong classifications in MNIST hand-writing dataset respectively. The correctly-classified images have some common properties. Firstly, they are all clearly written with solid lines. Secondly, they are mostly upright. Thirdly, they are not ambiguous as each of them do not show similarity to other digits.

The wrongly-classified examples shown in Figure 8.4, on the other hand, are mostly not written in very solid lines. Some of them are not upright (such as the "0" in ninth image). Finally and most importantly, almost all of them are ambiguous and can be easily recognised as other digits. For example, in the first image, it is quite reasonable for the classifier to recognise it as a "4" since many people do write "4"s like that.

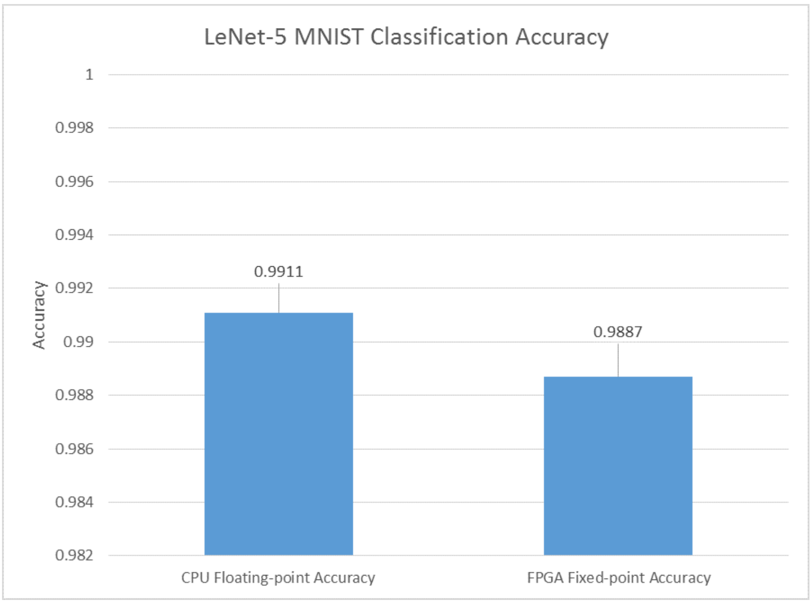


Figure 8.2: Comparison between CPU implementation accuracy using floating-point format, and FPGA implementation accuracy using 8-bit fixed-point format

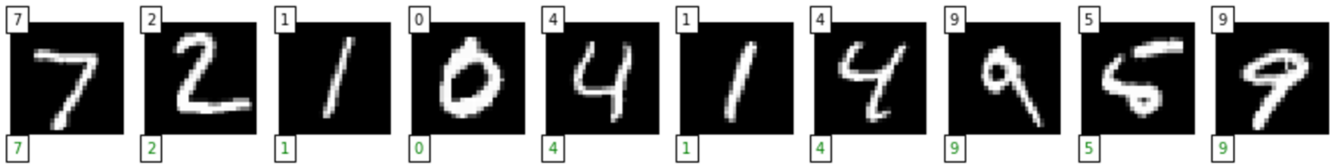


Figure 8.3: Examples of correct classifications in MNIST dataset. Digits on the upper-left corner are correct labels, and digits on the lower-left corner are predicted labels

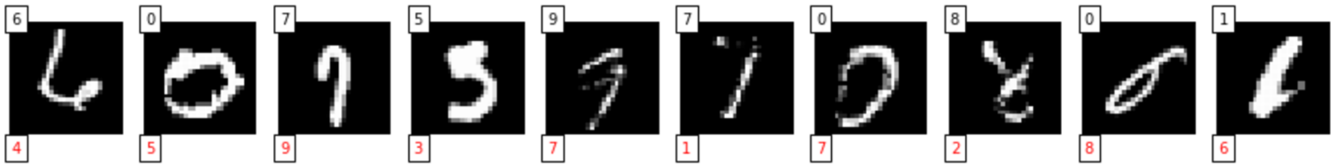


Figure 8.4: Examples of wrong classifications in MNIST dataset. Digits on the upper-left corner are correct labels, and digits on the lower-left corner are predicted labels

8.2. Complete SDFG Implementation - CIFAR-10

Similar to previous section, CIFAR-10 model (Caffe "quick" version[38]) has been completely implemented on FPGA using my framework. This model has been used to classify CIFAR-10 image set, and its performance will be benchmarked in the same manner as in the previous section. The hand-tuned compile-time parameters for each layer of CIFAR-10 are listed in Table 8.4. For CIFAR-10 model topology in Lasagne syntax, please see Appendix D.

Layer	ID	Maximum Weight Height	Maximum Weight Width	Folding Factor
CONV_1	1	32	75	25
CONV_2	2	32	800	25
CONV_3	3	64	800	25
FC_1	4	64	1024	32
FC_2	5	10	64	16

Table 8.4: Parameters hand-tuned for optimal CIFAR-10 deployment

8.2.1. Resource Usage

Table 8.5 lists the FPGA on-chip resource usage for CIFAR-10 deployment. It can be observed that the DSP usage has approached maximum capacity, and this framework will struggle with fitting larger CNN designs on board in complete SDFG. Again, Section 8.3 will propose SDF subgraph implementation which aims to solve the scalability issue.

Resource	Usage	Available	Percentage Usage (%)
BRAM	102	140	73
DSP48E	198	220	90
LUT	43092	53200	81

Table 8.5: Resource Usage of CIFAR-10 deployment as complete SDFG

8.2.2. Theoretical Maximum Throughput

The theoretical maximum possible parallel MACC operations per second is given by:

$$Throughput = \sum_{i}^{layers} folding\ factor_i * clock\ frequency \tag{8.4}$$

$$= (25+25+25+32+16)*100M \tag{8.5}$$

$$= 12.3\ GOP/s \tag{8.6}$$

Thus, without considering data stream stalling, the maximum theoretical throughput that can be achieved by my design should be 12.3 GOP/s. The actual performance should be worse than this.

8.2.3. Actual Throughput and Bottlenecks

I benchmark my framework's CIFAR-10 deployment latency against ARM CPU latency using Theano with Lasagne, and Figure 8.5 plots their latency against input image batch size.

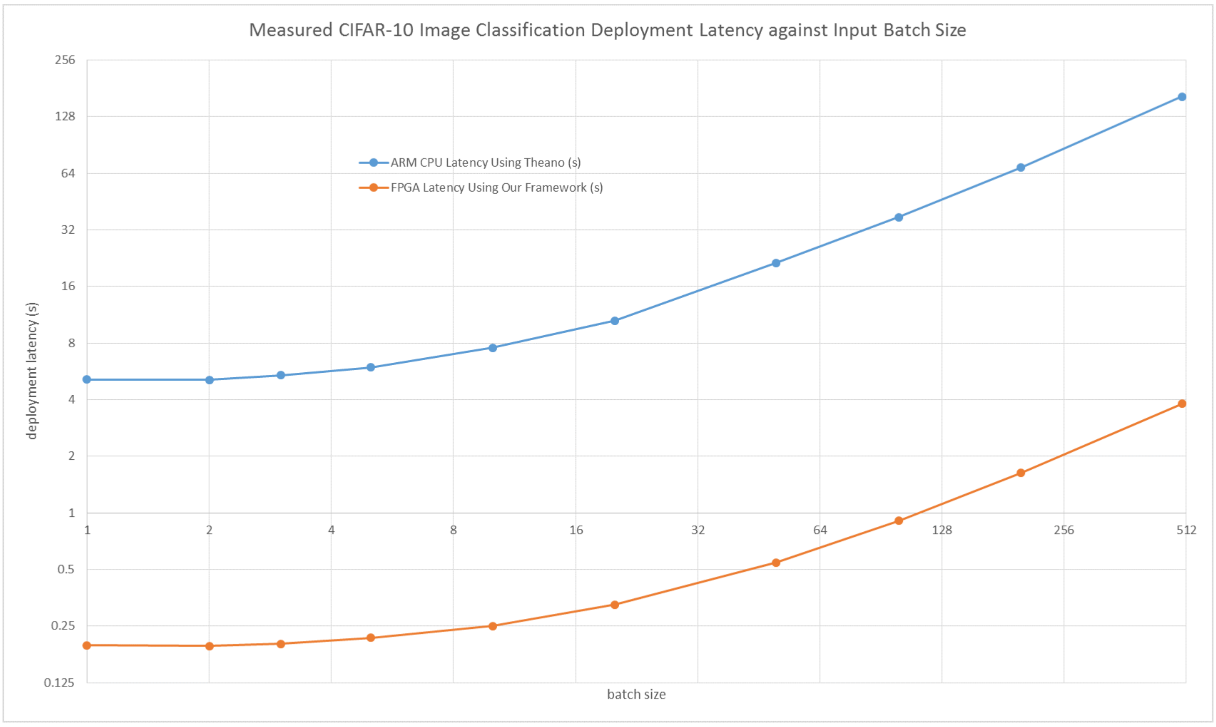


Figure 8.5: Plot of deployment latency for CPU using Theano and FPGA using my framework, against input batch size

In general, my framework shows a sustained 43x speed up as compared to ARM CPU performance. Both plots show upward slope. Both plots are show upward sloping, suggesting that for both implementations, with small batch sizes the python framework initialisation latency dominates total deployment latency, suggesting that increasing batch size increases the deployment throughput.

From Figure 8.5, I can estimate that the sustained CIFAR-10 deployment frame rate is 132 frames per second. From published analysis reports on LeNet-5, I know that for each input image, the CIFAR-10 model performs 0.0248 GOPs. [8][38]. Thus, the maximum sustainable throughput for my framework's deployment of CIFAR-10 is

$$0.0248\ GOPs/image * 132\ images/s = 3.2736\ GOPs/s$$

The reduction in actual throughput compared to theoretical throughput is due to bottleneck in **FC_1** layer. In deployment, **FC_1** layer continuously takes in 1024 input data, then halts the entire data stream for about (1024/32*64 = 2048) clock cycles to generate output stream, and then the process repeats. Thus, this layer produces the greatest backward pressure on the data stream, stalling the data stream.

Although increasing the data throughput of **FC_1** layer can effectively increase the overall throughput, since the CIFAR-10 IP is already using up 90% of total DSP, there is little room for more parallelism in this layer. Partitioning the SDFG into SDF subgraphs can allow **FC_1** layer to utilise more resources and exploit greater parallelism, but at the expense of reconfiguring the FPGA at runtime between adjacent SDF subgraphs. For details of SDF Subgraph implementation, please see Section 8.3.

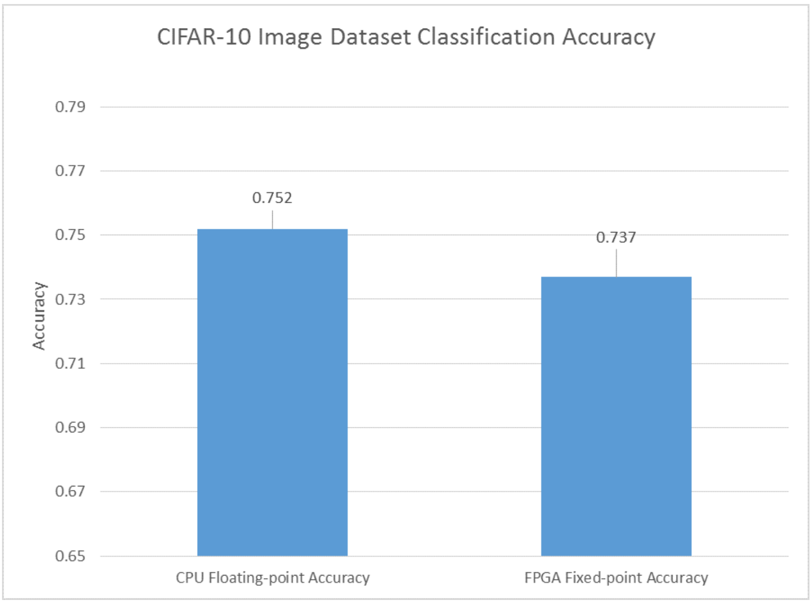


Figure 8.6: Comparison between CPU implementation accuracy using 32-bit floating-point format, and FPGA implementation accuracy using 16-bit fixed-point format

8.2.4. Power Efficiency

With a sustained frame rate of 132 frames per second, and an average power consumption of 2.063 W, my implementation of CIFAR-10 model achieves an energy usage of $132\ frames/s / 2.063\ J/s = 63.98\ frames/J$, which is a low power consumption per frame.

8.2.5. Accuracy

In this CIFAR-10 prototype, I implemented a fixed-point 16-bit quantisation in order to optimise on-chip resource usage, resulting in some classification accuracy loss.

Similar to LeNet-5 implementation, I implemented the CIFAR-10 model to classify CIFAR-10 image dataset, using pretrained weights. The testing set of CIFAR-10 image set contains 10000 32x32 images which are different from CIFAR-10 training dataset. With the CIFAR-10 testing dataset, the pretrained model weights can achieve a test accuracy of 75.2% using 32-bit floating-point format. Figure 8.2 shows that my FPGA implementation incurs an accuracy loss of 1.5% due to 16-bit fixed-point quantisation.

Again, as a trade-off between accuracy and deployment speed, this level of accuracy loss is reasonable considering that the FPGA implementation also achieved 42x acceleration as compared to ARM CPU implementation.

Figure 8.7 and Figure 8.8 show examples of correct and wrong classifications in CIFAR-10 image dataset respectively.

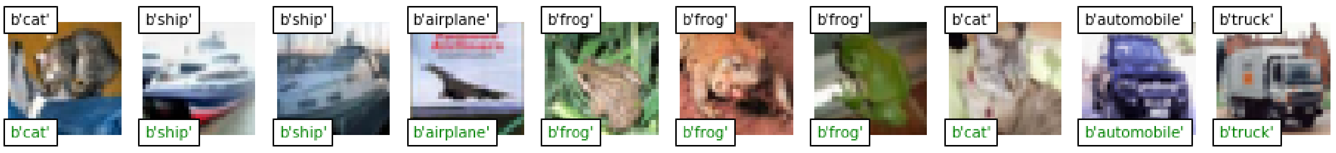


Figure 8.7: Examples of correct classifications in CIFAR-10 dataset. Digits on the upper-left corner are correct labels, and digits on the lower-left corner are predicted labels

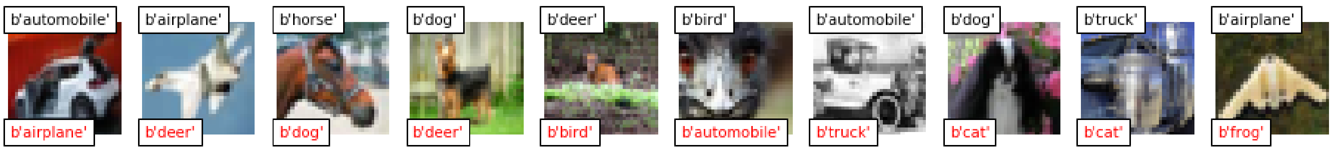
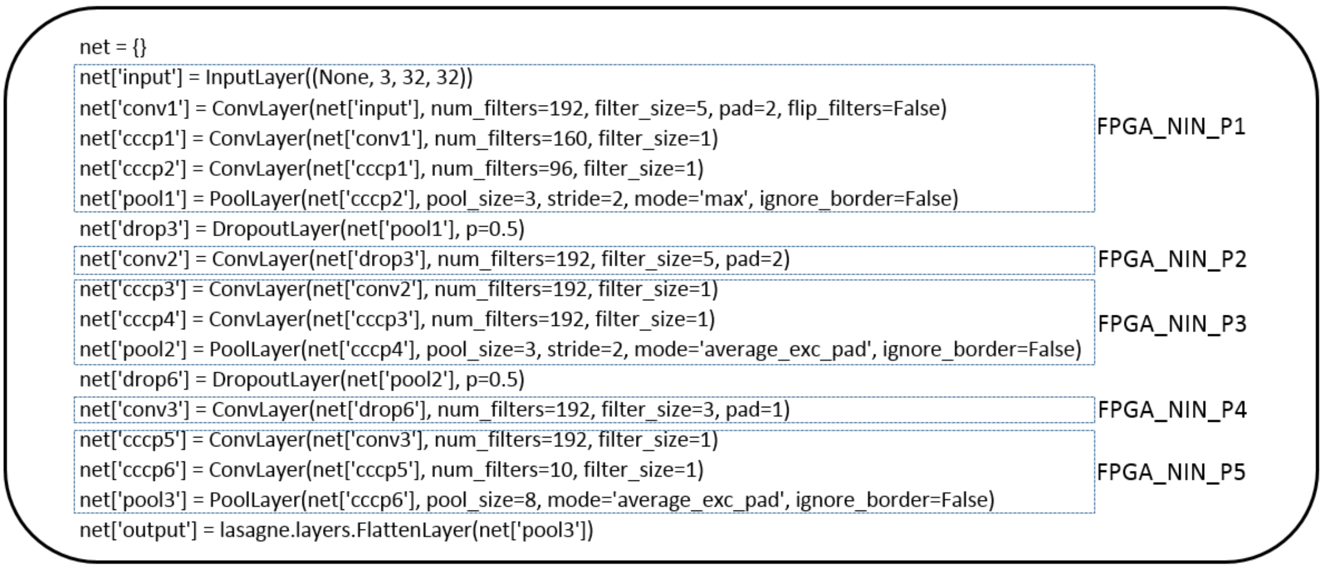


Figure 8.8: Examples of wrong classifications in CIFAR-10 dataset. Digits on the upper-left corner are correct labels, and digits on the lower-left corner are predicted labels

8.3. SDF Subgraph Implementation

As explained in previous prototypes, implementing complete SDFG on chip results in a limit in scalability due to limitation of FPGA on-chip resources. In order to address that, in the third prototype I attempted SDF subgraph implementation. My third prototype partitions a complex CNN model and implements each SDF subgraph individually on chip. During deployment, these SDF subgraphs will be executed sequentially, with the FPGA reconfiguring at runtime for each new SDF subgraph.

The complex CNN model that I have implemented is named NIN for CIFAR-10 classification. In NIN, the network is made of multiple micro CNNs as function approximators stacking together. Global average pooling instead of fully-connected layers is used to avoid over-fitting[42]. With greater depth and complexity, NIN is reported to be able to provide better classification accuracy as compared to my previous CIFAR-10 model (Caffe "quick" version). For NIN-for-CIFAR-10 model topology in Lasagne syntax, please see Appendix E.



```
net = {}
net['input'] = InputLayer((None, 3, 32, 32))
net['conv1'] = ConvLayer(net['input'], num_filters=192, filter_size=5, pad=2, flip_filters=False)
net['cccp1'] = ConvLayer(net['conv1'], num_filters=160, filter_size=1)
net['cccp2'] = ConvLayer(net['cccp1'], num_filters=96, filter_size=1)
net['pool1'] = PoolLayer(net['cccp2'], pool_size=3, stride=2, mode='max', ignore_border=False)
net['drop3'] = DropoutLayer(net['pool1'], p=0.5)
net['conv2'] = ConvLayer(net['drop3'], num_filters=192, filter_size=5, pad=2)
net['cccp3'] = ConvLayer(net['conv2'], num_filters=192, filter_size=1)
net['cccp4'] = ConvLayer(net['cccp3'], num_filters=192, filter_size=1)
net['pool2'] = PoolLayer(net['cccp4'], pool_size=3, stride=2, mode='average_exc_pad', ignore_border=False)
net['drop6'] = DropoutLayer(net['pool2'], p=0.5)
net['conv3'] = ConvLayer(net['drop6'], num_filters=192, filter_size=3, pad=1)
net['cccp5'] = ConvLayer(net['conv3'], num_filters=192, filter_size=1)
net['cccp6'] = ConvLayer(net['cccp5'], num_filters=10, filter_size=1)
net['pool3'] = PoolLayer(net['cccp6'], pool_size=8, mode='average_exc_pad', ignore_border=False)
net['output'] = lasagne.layers.FlattenLayer(net['pool3'])
```

Figure 8.9: The NIN for CIFAR-10 model topology (in Lasagne syntax) and my partitioning of this model for SDF subgraph implementation

Figure 8.9 shows the structure of NIN model. Since my framework currently cannot implement dropout layer, my partitioning will be primarily drawn to exclude dropout layers (which will be processed on ARM CPU). Besides, the layer "conv2" and "conv3" require large amount of on-chip BRAMs, each of them will be implemented independently as a single subgraph.

8.3.1. Resource Usage

Table 8.6 lists the FPGA on-chip resource usage for NIN-for-CIFAR-10 model deployment. Each partition is aiming to achieve the maximum parallelism, utilising the maximum amount of resources in the process.

8.3.2. Throughput and Bottlenecks

I benchmark my framework's NIN-for-CIFAR-10 deployment latency against ARM CPU latency using Theano with Lasagne, and Figure 8.10 plots their latency against input image batch size.

As shown in Figure 8.11, my framework shows a sustained 15.5x speed up as compared to ARM CPU performance. However, this speedup is only achieved when the batch size is very high. As the batch size reduces, the speedup drops significantly. This is because as the batch size becomes small, the fixed FPGA reconfiguration latency dominates the overall execution latency, reducing the overall throughput. Thus, in real-time applications, where the batch size tends to be smaller for high output frame rate, the throughput will be limited.

Another interesting observation from Figure 8.10 and Figure 8.11 is that my performance test stops when the batch size is 40. This is because as batch size increases, the DDR buffer size required for temporary feature map

Partition	Resource	Usage	Available	Percentage Usage (%)
P1	BRAM	63	140	45
	DSP48E	154	220	70
	LUT	23940	53200	45
P2	BRAM	140	140	100
	DSP48E	35	220	16
	LUT	14896	53200	28
P3	BRAM	52	140	37
	DSP48E	128	220	58
	LUT	18620	53200	35
P4	BRAM	140	140	100
	DSP48E	62	220	28
	LUT	17024	53200	32
P5	BRAM	52	140	37
	DSP48E	128	220	58
	LUT	16492	53200	31

Table 8.6: Resource Usage of NIN for CIFAR-10 using SDF subgraph implementation

storage increases. In forward propagation of CNN, the size of feature map can expand significantly in the process. Since the maximum DDR buffer for DMA, as defined by PYNQ DMA driver, is around 8.3MB, the maximum batch size is limited to 40 input images. This batch size restriction significantly limits the design's potential for higher throughput.

From Figure 8.10, I can estimate the sustained deployment frame rate as 3.81 frames per second. I know that for each input image, NIN-for-CIFAR-10 model performs 0.2224 GOPs [42]. Thus, the maximum sustainable throughput for my framework's deployment of CIFAR-10 is

$$0.2224\ GOPs/image * 3.81\ images/s = 0.8477\ GOPs/s$$

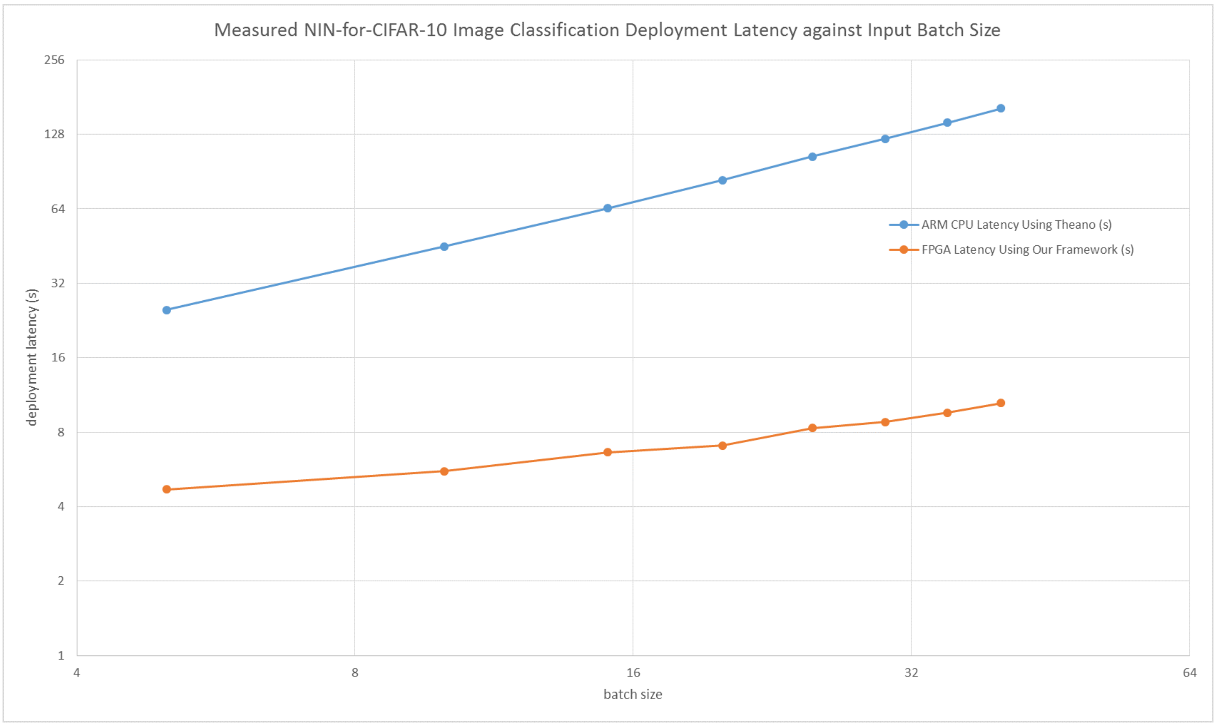


Figure 8.10: Plot of deployment latency for CPU using Theano and FPGA using my framework, against input batch size

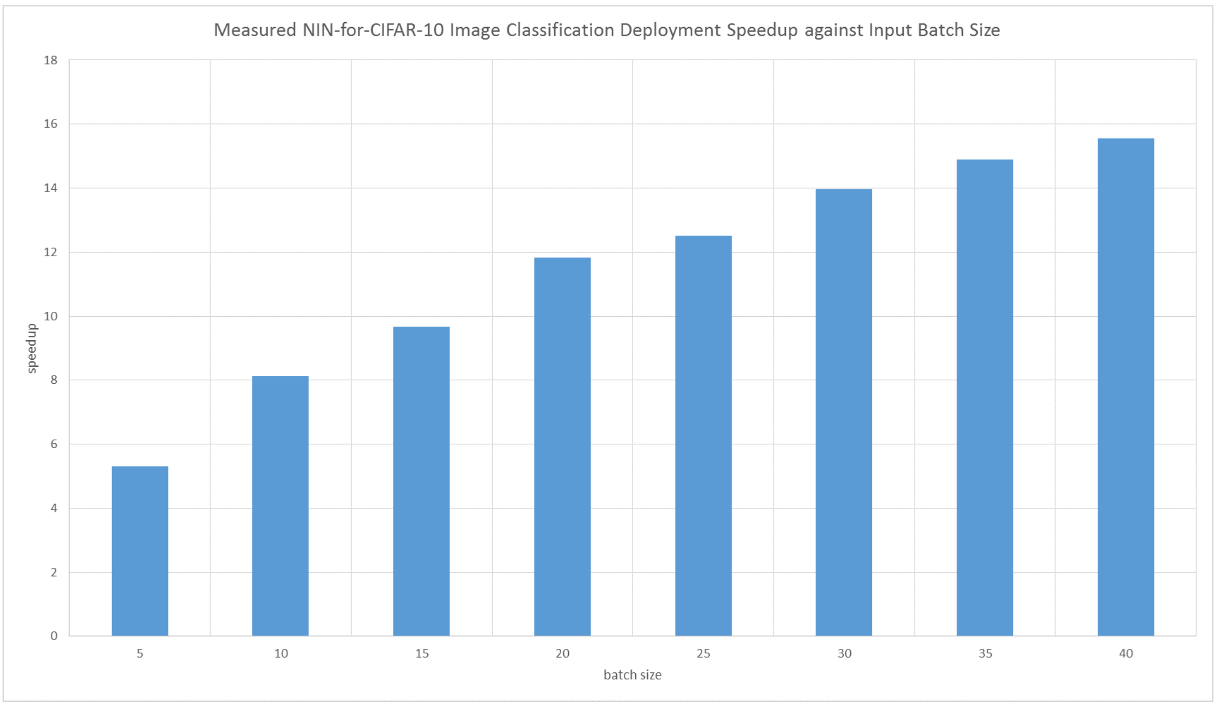


Figure 8.11: Plot of speedup of my framework (compared to ARM CPU implementation), against input batch size

In conclusion, although SDF subgraph implementation has expanded my framework's scalability, making my design capable of accommodating deeper and more complex networks, it also significantly reduced the throughput performance and increased the memory footprint for buffering intermediate feature maps. Consequently,

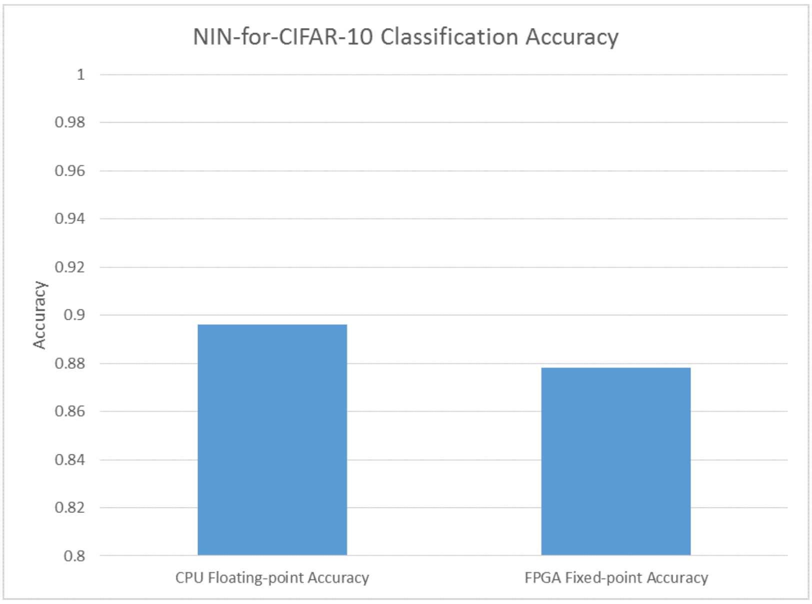


Figure 8.12: Comparison between CPU implementation accuracy using 32-bit floating-point format, and FPGA implementation accuracy using 8-bit fixed-point format

the design has lower throughput for real-time applications with smaller input batch size, and is incapable of implementing applications with large batch size (such as Regional CNN (R-CNN)).

8.3.3. Power Efficiency

With a frame rate of 3.81 frames per second, and an average power consumption of 1.951 W, my implementation of CIFAR-10 model achieves an energy usage of 3.81 $frames/s/1.951\ J/s = 1.953\ frames/J$, which is a low power consumption.

8.3.4. Accuracy

Finally, I implemented the NIN-for-CIFAR-10 model to classify the same CIFAR-10 image dataset as previous test, using pretrained weights which achieve a test accuracy of 89.6% using 32-bit floating-point format[42]. My design implements the model with a fixed-point quantisation of 8-bit to save BRAM and DSP resource usage, losing some accuracy in the process. Figure 8.12 shows the accuracy loss of FPGA implementation due to 8-bit fixed-point quantisation. Note that this implementation has achieved significant accuracy improvement as compared to previous CIFAR-10 model (Caffe "quick" version), which has a testing accuracy of 0.737. Hence, with SDF subgraph implementation, my framework is able to implement deeper and more complex CNNs, which can achieve better classification accuracy.

8.4. Summary

To sum up, the three tests that I have implemented in this section demonstrate the following characteristics of my framework:

High Speed: All three prototypes demonstrate high speed performance and operation parallelism, in the form of high throughput performance. In the LeNet-5 test, my implementation has been compared with published benchmarks, showing that my framework is capable of delivering state-of-the-art throughput performance, with appropriate tuning of design parameters.

Low Power Consumption: The prototypes show low power consumption per frame, making them suitable for embedded applications.

Efficient and Customisable On-chip Resource Utilisation: In NIN-for-CIFAR-10 implementation, each subgraph has been fine-tuned to utilise high parallelism at the expense of high resource usage. In LeNet-5 and CIFAR-10 implementations, on the other hand, the design has been fine-tuned to balance between resource limitation and parallelism. Hence, with appropriate fine-tuning of design parameters, my framework is capable of delivering highly-customisable implementations, suitable for various application requirements.

Fast Prototyping: The framework enables engineers to fully and easily control the resource-parallelism trade-off via fine-tuning a folding factor. This greatly simplifies the engineers' efforts in optimising the FPGA implementation. Meanwhile, the APIs that execute the FPGA IPs are wrapped into Lasagne-format layers, and engineers can instantiate my FPGA-accelerated layers in the same manner as Lasagne built-in layers. This allow engineers to quickly get used to prototyping with my framework. As an evidence for my argument, all three prototypes above were implemented by me within less than one month's time.
However, from the above performance analysis, my framework also demonstrates the following problems:

Scalability: The scalability of complete implementation of SDFG on FPGA will inevitably be limited by the on-chip resources. In order to extend my framework's scalability, I attempted SDF subgraph implementation, which can theoretically implement very deep and complex CNNs. However, from the test on NIN-for-CIFAR-10 implementation, I observed problems such as reduced throughput for small batch size and increased memory footprint. These problems make SDF subgraph implementation less desirable for my project since my scope is embedded system applications.

Hand-tuned Fixed-point Quantisation: In the three tests above, the quantisation was implemented using hand-tuned scaling factors. Automated dynamic quantisation algorithms, such as Ristretto, can be implemented on FPGA IPs to reduce quantisation efforts and minimise accuracy loss. [25]

Section 9 explains some future plans which aim to solve these problems.

9. Conclusion and Further Plans

This project proposes a fast FPGA prototyping framework for high performance CNN deployment on PYNQ platform. In order to achieve this aim, a set of design considerations (listed in Table 9.1) have been proposed, and the appropriate performance metrics have been clarified. With these design considerations in mind, background research has been carried out, which explains the relevant technical aspects and lists the related works which my project can refer to. Then, the report details how the design should be implemented, explaining how the design considerations have been approached based on design space analysis. After that, the report describes the testing environment that I constructed for validation and experimental setup. Finally, in order to evaluate on the performance of the framework, three CNN classifier prototypes have been constructed, and their performances have been compared against published benchmarks, showing that my framework is able to deliver state-of-the-art CNN deployment performance, while requiring minimal prototyping efforts from engineers.

Table 9.1 provides a list of the design considerations, as well as my design decisions which try to address them.

	Design Consideration	Design Decision
LINUX-SIDE	High level interface framework	Theano with Lasagne has been selected, since it requires the least installation efforts, while providing the most intuitive design interface and the best supports on embedded CNN deployment as well as customised layer design.
	FPGA data transfer API	The PYNQ's built-in DMA API has been selected as the CPU-FPGA data transfer API due to high data-streaming throughput.
FPGA-SIDE	Data streaming interface	The data streaming interface is constructed as a combination of AXI4-Streaming protocol and DMA, since this structure can provide higher data transmission throughput.
	Quantisation	Fixed-point data representations are implemented because of smaller on-chip memory footprint and lower DSP usage. User can parametrise the implementation bitwidth.
	Memory Architecture	For optimal speed performance, all weights are stored in on-chip memory to minimise memory transfer latency. Weights can be reloaded at runtime.
	Design Parametrisability	A number of compile-time parameters have been provided, which give engineers the maximum freedom to customise on the hardware's resource usage and throughput
	Performance Optimisation	A number of strategies for improving parallelism have been implemented. With the limited on-chip resources, the design aims to provide maximum deployment speedup at the expense of minimum accuracy loss and power consumption.
	Scalability	With SDF subgraph implementation, my framework is able to deploy very large CNN models at the expense of reduced throughput performance.

Table 9.1: Project Design Decision checklist

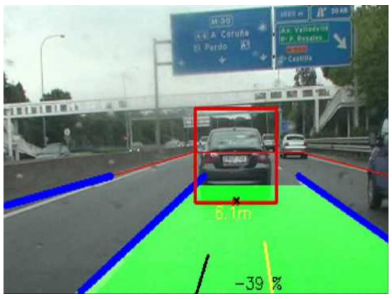
In general, all the design considerations listed have been covered with appropriate responses in the project, and based on the performance evaluation of my three design prototypes, I can conclude that the aims of **fast FPGA prototyping** and **high performance CNN deployment** are well met. However, if more time has been provided, more implementations could have be attempted which could potentially deliver even better performance. Below lists some alternative designs which are worthwhile attempting in the future.

- **Automatic Quantisation:** Currently, in my framework the quantisation needs to be hand-tuned and hard-coded for each CNN model. For improved simplicity, the quantisation process can be implemented in the FPGA hardware, where each layer registers the range of each channel of input feature map, and scale it up to the maximum range defined by the current bitwidth.

- **Block Design User Interface:** Xilinx Vivado currently does not support HLS designs packaged as user-customisable block design IPs. Such block design IPs can only be generated by complete RTL projects. Currently, my IP library can only be parametrised by using a header file in Vivado HLS One workaround could be to export the automatic generated RTL source code from Vivado HLS, then use this RTL source code to generate user-customisable block design IPs in Vivado.

- **Binarised Neural Network (BNN):** Currently, with complete SDFGs implementation, my framework can only accommodate models up to the scale of LeNet-5 or CIFAR-10 Caffe "quick" version. My current solution for the scalability issue is SDF subgraph implementation, which is capable of implementing very large CNNs, at the expense of reduced overall speed and increased memory footprint. Another possible method to increase scalability is BNN implementation. A network model can be converted to binarised network and then implemented on FPGA. Binarisation can significantly reduce memory footprint since each weight data has been represented by one single bit. The dot-product operation has been simplified into batchnorm-activation and thresholding, reducing the requirement for DSP. Thus, with BNN implementation my framework should be able to accommodate deeper and more complex CNN models. Research from Xilinx Ireland Lab has shown very promising results on BNN. Their implementation of BNN on PYNQ has a sustained throughput of 8620.69 images per second for LeNet-5 and 446.03 images per second for VGG-16, which are the best speed performances reported to date. Hence, I plan to implement a variation of my framework which converts CNN models to BNN models, then deploy them on FPGA.[6][41]

- **Real-time Application Prototypes:** In order to showcase the potential of my framework, applications which perform real-time classifications can be designed. For example, with some changes, R-CNN can be implemented based on my existing CNN image classifier prototypes. An automotive front-view road sign detector or vehicle detector can be implemented using R-CNN.



(a) Real-time application: front view camera - car detector

(b) Real-time application: front view camera - road sign detector

10. About My Experience of Working with PYNQ

PYNQ is a perfect platform for my project, which has greatly reduced my designing efforts. Below lists the two innovative concepts introduced by PYNQ, which are invaluable to my project.

- **Embedded OS:** When designing an embedded application, having an embedded OS can allow engineers to work with powerful and user-friendly programming environments. PYNQ provides a Linux Ubuntu embedded OS installed on the SoC ARM processors, providing full support for Python compilers, Jupyter Notebooks and any other software that is compatible with Ubuntu. This has greatly simplified my workload for the project. Thanks to the Linux OS, I can easily install a Theano with Lasagne onto the board as my framework's high level interface, or demonstrate the performance of my prototype with Jupyter Notebook, or even visualise the classification results of my prototypes using "matplot" package in Python.
- **"RTL-free" FPGA Design:** PYNQ provides us with the possibility of designing an FPGA design without actually going through RTL designing process. Due to the difference in programming workflow and design intuitions, RTL has deterred many engineers from ever exploring FPGA platforms. PYNQ provides a concept of "RTL-free" FPGA design workflow, where engineers can use well-packaged FPGA IPs (overlays) instead of designing IPs by themselves. This is actually where the idea of my project actually comes from: to design an open source library of parametrisable FPGA IPs for CNN deployment, so that engineers who wish to accelerate CNN on FPGA no longer need to design RTL themselves.

Currently, since PYNQ has just been published, there are limited number of reference designs available, and the library of open source FPGA overlays has not been really developed. In my opinion, for PYNQ to be really well-acknowledged, more eye-catching PYNQ reference designs need to be published. These reference designs should not only show high performance provided by FPGA, but also the simplicity in developing embedded designs on PYNQ platform. In this way, a community of PYNQ developers can be formed, which will contribute more high-quality overlays and reference designs. I believe that PYNQ has the potential to become a very successful educational platform, if there is a community of engineers supporting it.

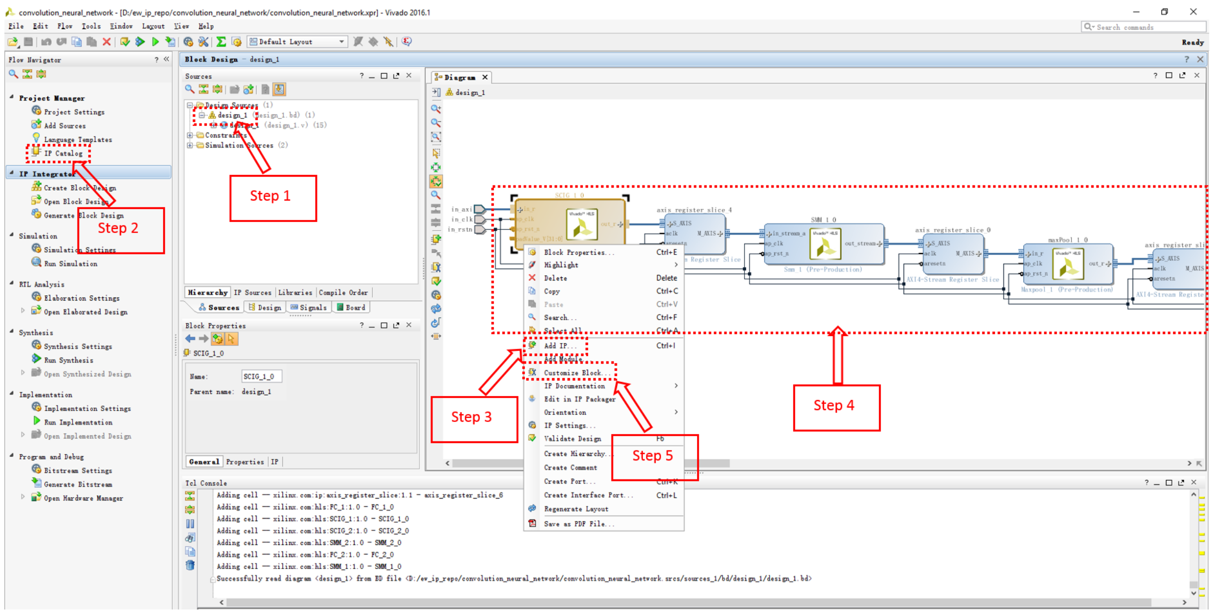
11. User Guide

This user guide introduces how to deploy FPGA-accelerated LeNet-5 model on PYNQ board using my framework. The source code and PYNQ SD card image will be available on the github soon, and will be open for free download. The user guide has been tested on Xilinx^{TM} Vivado 2016.1. Before trying this user guide, user should be able to set up the connection to PYNQ and should flash the PYNQ micro SD card with the image provided in my source code.

11.1. Constructing CNN using my IP Blocks in Xilinx^{TM} Vivado

Firstly, the CNN model topology needs to be built and fine-tuned in Xilinx^{TM} Vivado as a block design IP. The model topology will be constructed by chaining up CNN layer block design IPs provided by my framework.

1. Open **"CNN_BLOCK_DESIGN"** project in Xilinx^{TM} Vivado. In **"Project Manager"** panel, open block design **"Design_1"**. The block design should be empty except for "axis_in", "axis_out", "clk_in" and "rsn_in" ports.
2. In **"Flow Navigator"**, click on **"IP Catalogue"**. Right click on **"Vivado Repository"** and select **"Add Repository"**. Browse to my framework's directory and click **"Select"**.
3. Return to the block design **"Design_1"**. Right click and select **"Add IP..."**. Add component layers of LeNet-5 onto the block design, in the correct order as the CNN SDFG. Between adjacent IPs, insert a **"AXI4-Streaming Register Slice"** IP to avoid timing violation.
4. Connect all "ap_clk" ports on IP blocks to **"clk_in"** port. Connect all **"ap_rst_n"** ports on IP blocks to **"rsn_in"** ports. Chain up all IPs with AXI4-Streaming connections.
5. For each CNN layer IP, right click on its block design and select **"Customise Block..."**. Fine-tune the design parameters based on your design requirements.
6. Click on **"Tools"**, then click **"Create and Package IP..."**. In **"Packaging Options"**, choose **"Package a block design from the current project"** and select **"Design_1"** as the target block design. Choose the packaged IP location and go to **"Next"**. The packaged block design IP containing your CNN topology should be generated at the specified location.



```
Adding cell -- xilinx.com:ip:axis_register_slice:1.1 - axis_register_slice_6
Adding cell -- xilinx.com:hls:FC_1:1.0 - FC_1_0
Adding cell -- xilinx.com:hls:SCIG_1:1.0 - SCIG_1_0
Adding cell -- xilinx.com:hls:SCIG_2:1.0 - SCIG_2_0
Adding cell -- xilinx.com:hls:SMM_2:1.0 - SMM_2_0
Adding cell -- xilinx.com:hls:FC_2:1.0 - FC_2_0
Adding cell -- xilinx.com:hls:SMM_1:1.0 - SMM_1_0
Successfully read diagram <design_1> from BD file <D:/ew_ip_repo/convolution_neural_network/convolution_neural_network.srcs/sources_1/bd/design_1/design_1.bd>
```

Figure 11.1: Constructing CNN using my IP Blocks in Xilinx^{TM} Vivado

11.2. Synthesise Bitstream

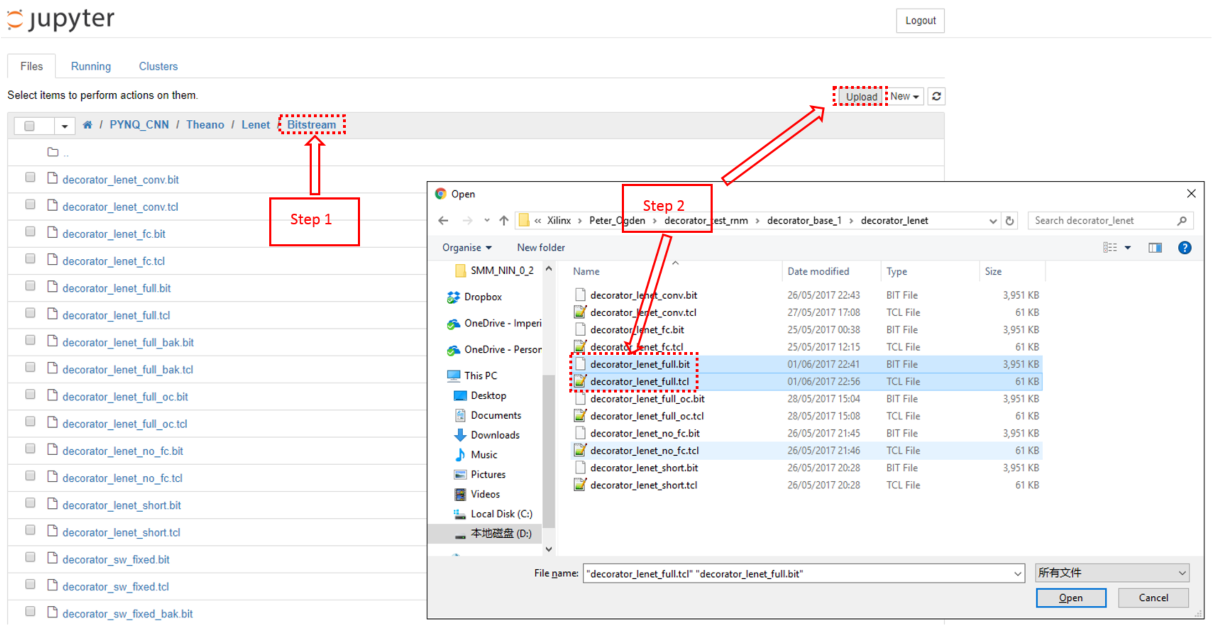
Secondly, the block design IP packaged from previous step needs to be incorporated into the PYNQ **"base"** project, which contains the base overlay architecture including key hardware components such as DMA controller and DDR port.

1. In Xilinx^{TM} Vivado, open the "base" project provided by my source code.
2. Replace the place-holding block design with the CNN block design IP that you have just packaged.
3. Click on **"Generate Bitstream"**.
4. Go to "File"->"Export"->"Export Bitstream" to export the bitstream. Go to "File"->"Export"->"Export Block Design" to export the block design tickle file, which is required by PYNQ DMA driver.

11.3. Download Bitstream to PYNQ and Link to Lasagne FPGA Layer

Thirdly, the synthesised bitstream needs to be downloaded to the PYNQ linux machine. The bitstream will be linked in the PYNQ API, which is packaged in a Lasagne customised layer function. The bitstream will be downloaded and the FPGA IPs will be executed when the Lasagne layer function is executed.

1. Connect to PYNQ board using Jupyter Notebook. Browse to the **"Bitstream"** folder under my project directory. Upload both the bitstream and tickle file into the **"Bitstream"** folder.
2. If both files are using the default names, the Lasagne customised layer should have automatically linked to the bitstream. Otherwise, go to the "conv_layer.py" file which defines this Lasagne function, and correct the bitstream file name.



decorator_lenet_conv.bit
decorator_lenet_conv.tcl
decorator_lenet_fc.bit
decorator_lenet_fc.tcl
decorator_lenet_full.bit
decorator_lenet_full.tcl
decorator_lenet_full_bak.bit
decorator_lenet_full_bak.tcl
decorator_lenet_full_oc.bit
decorator_lenet_full_oc.tcl
decorator_lenet_no_fc.bit
decorator_lenet_no_fc.tcl
decorator_lenet_short.bit
decorator_lenet_short.tcl
decorator_sw_fixed.bit
decorator_sw_fixed.tcl
decorator_sw_fixed_bak.bit

Name	Date modified	Type	Size
decorator_lenet_conv.bit	26/05/2017 22:43	BIT File	3,951 KB
decorator_lenet_conv.tcl	27/05/2017 17:08	TCL File	61 KB
decorator_lenet_fc.bit	25/05/2017 00:38	BIT File	3,951 KB
decorator_lenet_fc.tcl	25/05/2017 12:15	TCL File	61 KB
decorator_lenet_full.bit	01/06/2017 22:41	BIT File	3,951 KB
decorator_lenet_full.tcl	01/06/2017 22:56	TCL File	61 KB
decorator_lenet_full_oc.bit	28/05/2017 15:04	BIT File	3,951 KB
decorator_lenet_full_oc.tcl	28/05/2017 15:08	TCL File	61 KB
decorator_lenet_no_fc.bit	26/05/2017 21:45	BIT File	3,951 KB
decorator_lenet_no_fc.tcl	26/05/2017 21:46	TCL File	61 KB
decorator_lenet_short.bit	26/05/2017 20:28	BIT File	3,951 KB
decorator_lenet_short.tcl	26/05/2017 20:28	TCL File	61 KB

File name: "decorator_lenet_full.tcl" "decorator_lenet_full.bit"

Figure 11.2: Download Bitstream to PYNQ and Link to Lasagne FPGA Layer

11.4. CNN Deployment

Finally, instantiate this CNN in a python script, by declaring a Lasagne CNN that includes my FPGA-accelerated CNN function.

1. Load the weights of each layer, using **"FPGAWeightLoader"** function provided in **"conv_layer.py"**. Below is its function prototype, showing the arguments required.

```
FPGAWeightLoader(W, index, IFMDim, OFMDim, PadDim, flip_filters=True)
```

2. Declare the Lasagne CNN which includes the FPGA-accelerated layer. Below is an example showing how FPGA-accelerated LeNet-5 has been instantiated (**FPGA_LENET** is the name of the customised layer containing FPGA API).

```
FPGA_net = {}
FPGA_net['input'] = InputLayer((None, 1, 28, 28))
FPGA_net['lenet'] = FPGA_LENET(FPGA_net['input'])
FPGA_net['prob'] = NonlinearityLayer(FPGA_net['lenet'], softmax)
```

3. Execute the forward propagation in Lasagne syntax.

```
prob = lasagne.layers.get_output(FPGA_net['prob'], floatX(test_data[0:batch_size]),
    deterministic=True).eval()
```

References

[1] Nallatech. Fpga acceleration of convolutional neural networks. 2016.

[2] Aerotenna. Aerotenna high performance radar sensors.

[3] Yuran Qiao, Junzhong Shen, Tao Xiao, Qianming Yang, Mei Wen, and Chunyuan Zhang. Fpga-accelerated deep convolutional neural networks for high throughput and energy efficiency. *Concurrency and Computation: Practice and Experience*, 2016.

[4] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[5] Stylianos I Venieris and Christos-Savvas Bouganis. fpgaconvnet: A framework for mapping convolutional neural networks on fpgas. In *Field-Programmable Custom Computing Machines (FCCM), 2016 IEEE 24th Annual International Symposium on*, pages 40–47. IEEE, 2016.

[6] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. Finn: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA ’17, pages 65–74. ACM, 2017.

[7] SSRR2013. Demos: Aerial leashing with an ar.drone 2.0.

[8] Stylios Venieris. fpgaconvnet.

[9] Stanford University. Cs231n: Convolutional neural networks for visual recognition.

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

[11] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, et al. 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, 2016.

[12] Knight of Pi. Deepdreaming on a raspberry pi 2.

[13] benjibc. caffe-rpi.

[14] OSDEVLAB. How to install latest boost library on raspberry pi.

[15] encodeTS. Ubuntu14.04+cuda8.0+anaconda3 compile caffe.

[16] Yangqing Jia. Learning semantic image representations at a large scale. 2014.

[17] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 161–170. ACM, 2015.

[18] Pete Warden. Why gemm is at the heart of deep learning. 2015.

[19] Stefano Di Carlo, Giulio Gambardella, Marco Indaco, Daniele Rolfo, Gabriele Tiotto, and Paolo Prinetto. An area-efficient 2-d convolution implementation on fpga for space applications. In *Design and Test Workshop (IDT), 2011 IEEE 6th International*, pages 88–92. IEEE, 2011.

[20] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[21] Pete Warden’s Blog. How to quantize neural networks with tensorflow.

[22] Xilinx. Deep learning with int8 optimization on xilinx devices. 2016.

[23] Tim Dettmers. 8-bit approximations for parallelism in deep learning. *arXiv preprint arXiv:1511.04561*, 2015.

[24] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[25] Philipp Gysel, Mohammad Motamedi, and Soheil Ghiasi. Hardware-oriented approximation of convolutional neural networks. *arXiv preprint arXiv:1604.03168*, 2016.

[26] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66–73, 2010.

[27] Xilinx. Pynq.

[28] Ben Cope et al. Implementation of 2d convolution on fpga, gpu and cpu.

[29] Wikipedia. Comparison of deep learning software.

[30] Lauri's blog. Axi direct memory access.

[31] Xilinx. Vivado high-level synthesis.

[32] Shan Sung Liew, MOHAMED KHALIL HANI, SYAFEEZA AHMAD RADZI, and Rabia Bakhteri. Gender classification: a convolutional neural network approach. *Turkish Journal of Electrical Engineering & Computer Sciences*, 24(3):1248–1264, 2016.

[33] Xilinx. Axi reference guide. 2011.

[34] Daniele Bagni and et al. A zynq accelerator for floating point matrix multiplication designed with vivado hls. 2016.

[35] NamHyuk Ahn blog. Cnns in practice.

[36] Li Deng. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[37] Vinod Nair Alex Krizhevsky and Geoffrey Hinton. The cifar-10 dataset.

[38] BAIR. Alex's cifar-10 tutorial, caffe style.

[39] S. Ghaffari and S. Sharifian. Fpga-based convolutional neural network accelerator design using high level synthesize. In *2016 2nd International Conference of Signal Processing and Intelligent Systems (ICSPIS)*, pages 1–6, Dec 2016.

[40] Magnus Halvorsen. Hardware acceleration of convolutional neural networks. Master's thesis, NTNU, 2015.

[41] Xilinx. Bnn pynq.

[42] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *arXiv preprint arXiv:1312.4400*, 2013.

A. Caffe Testbench Setup Script

Listing 4: single_conv_layer.py

```python
import sys
sys.path.insert(0, '/home/xilinx/caffe/python')
import numpy as np
import matplotlib.pyplot as plt
import caffe
import cv2
import time

# Work is separated between CPU and FPGA, hence CPU mode is selected
caffe.set_mode_cpu()

# Declare a neural net based on configurations in myconvnet.prototxt
net = caffe.Net('myconvnet.prototxt', caffe.TEST)

# Input image is a 32x32 image input_32.png
img = cv2.imread('input_32.png', 1)
img_blobinp = img[np.newaxis, :, :, :]
img_blobinp = img_blobinp.transpose(0,3,1,2)
net.blobs['data'].reshape(*img_blobinp.shape)
net.blobs['data'].data[...] = img_blobinp

start = time.process_time()
# Execute forward propagation
net.forward()
print("Total forward time")
print(time.process_time()-start)

batch = net.blobs['conv'].data.transpose(0,2,3,1)

# Output the first 10 output images (in total 32 outputs)
for i in range(10):
   cv2.imwrite('output_image_' + str(i) + '.jpg', batch[i, :, :, :])

# Save the weights in myconvmodel.caffemodel
net.save('myconvmodel.caffemodel')
```

B. Caffe CNN Protobuf Configuration File

Listing 5: myconvnet.prototxt

```
# Declaring data layer with blob size of 1x3x32x32:
# 1 image per batch
# 3 colour channels
# 32x32 in dimension
name: "myconvolution"
input: "data"
input_dim: 1
input_dim: 3
input_dim: 32
input_dim: 32

# Declaring a Python Layer that executes convolution on FPGA
layer {
  type: 'Python'
  name: 'conv'
  top: 'conv'
  bottom: 'data'
  python_param{
    module: 'fpga_conv_im2col'
    layer: 'FPGAConvLayer'
  }
}
```

C. LeNet-5 Model Topology

Listing 6: LeNet5.py

```
net = {}
net['input'] = InputLayer((None, 1, 28, 28))
net['conv1'] = ConvLayer(net['input'], num_filters=20, filter_size=5, nonlinearity=linear)
net['pool1'] = PoolLayer(net['conv1'], pool_size=2, stride=2, mode='max', ignore_border=False)
net['conv2'] = ConvLayer(net['pool1'], num_filters=50, filter_size=5, nonlinearity=linear)
net['pool2'] = PoolLayer(net['conv2'], pool_size=2, stride=2, mode='max', ignore_border=False)
net['ip1'] = DenseLayer(net['pool2'], num_units=500, nonlinearity = rectify)
net['ip2'] = DenseLayer(net['ip1'], num_units=10, nonlinearity = None)
net['prob'] = NonlinearityLayer(net['ip2'], softmax)
```

D. CIFAR-10 (Caffe "Quick" Version) Model Topology

Listing 7: CIFAR10.py

```
net = {}
net['input'] = InputLayer((None, 3, 32, 32))
net['conv1'] = ConvLayer(net['input'], num_filters=32, filter_size=5, pad=2, nonlinearity=None)
```

```
net['pool1'] = PoolLayer(net['conv1'], pool_size=2, stride=2, mode='max', ignore_border=False)
net['relu1'] = NonlinearityLayer(net['pool1'], rectify)
net['conv2'] = ConvLayer(net['relu1'], num_filters=32, filter_size=5, pad=2,
    nonlinearity=rectify)
net['pool2'] = PoolLayer(net['conv2'], pool_size=2, stride=2, mode='average_exc_pad',
    ignore_border=False)
net['conv3'] = ConvLayer(net['pool2'], num_filters=64, filter_size=5, pad=2,
    nonlinearity=rectify)
net['pool3'] = PoolLayer(net['conv3'], pool_size=2, stride=2, mode='average_exc_pad',
    ignore_border=False)
net['ip1'] = DenseLayer(net['pool3'], num_units=64, nonlinearity = None)
net['ip2'] = DenseLayer(net['ip1'], num_units=10, nonlinearity = None)
net['prob'] = NonlinearityLayer(net['ip2'], softmax)
```

E. NIN-for-CIFAR-10 Model Topology

Listing 8: NIN.py

```
net = {}
net['input'] = InputLayer((None, 3, 32, 32))
net['conv1'] = ConvLayer(net['input'], num_filters=192, filter_size=5, pad=2,
    flip_filters=False)
net['cccp1'] = ConvLayer(net['conv1'], num_filters=160, filter_size=1)
net['cccp2'] = ConvLayer(net['cccp1'], num_filters=96, filter_size=1)
net['pool1'] = PoolLayer(net['cccp2'], pool_size=3, stride=2, mode='max', ignore_border=False)
net['drop3'] = DropoutLayer(net['pool1'], p=0.5)
net['conv2'] = ConvLayer(net['drop3'], num_filters=192, filter_size=5, pad=2)
net['cccp3'] = ConvLayer(net['conv2'], num_filters=192, filter_size=1)
net['cccp4'] = ConvLayer(net['cccp3'], num_filters=192, filter_size=1)
net['pool2'] = PoolLayer(net['cccp4'], pool_size=3, stride=2, mode='average_exc_pad',
    ignore_border=False)
net['drop6'] = DropoutLayer(net['pool2'], p=0.5)
net['conv3'] = ConvLayer(net['drop6'], num_filters=192, filter_size=3, pad=1)
net['cccp5'] = ConvLayer(net['nin4'], num_filters=192, filter_size=1)
net['cccp6'] = ConvLayer(net['cccp5'], num_filters=10, filter_size=1)
net['pool3'] = PoolLayer(net['cccp6'], pool_size=8, mode='average_exc_pad', ignore_border=False)
net['output'] = lasagne.layers.FlattenLayer(net['pool3'])
```