Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

R. Luis Jalabert

Deep Learning Based FPGA-CPU Acceleration

Master's thesis in EMECS
Supervisor: Kjetil Svarstad
June 2019

NTNU
Norwegian University of
Science and Technology

R. Luis Jalabert

Deep Learning Based FPGA-CPU Acceleration

Master's thesis in EMECS
Supervisor: Kjetil Svarstad
June 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

To all my teachers and professors

Summary

The purpose of this project is to continue exploring new ways of accelerating sequential computer code, and finding out if the machine learning techniques available today are able to help us in this task. The core idea is trying to parallelize during run-time (in a way completely transparent to the programmer) the code that's being executed in the CPU, by snooping on the RAM-CPU communication bus, and, in case an artificial neural network considers this code to be parallelizable, it should be then converted into an FPGA module and executed on it instead. It build upon a previous project in which we used a very simple RISC CPU, called LT16x32, and created several Software applications to aid us in this enterprise. First, a parametric assembly code generator named LoopGen, allowed us to create many examples of code with different degrees of parallelism. Next, a program called LoopSim generated the neural network's input datasets by compiling the generated code and simulating its execution on the CPU. In this project we improved on the previous neural network architecture by switching from an LSTM to a convolutional model, achieving a much higher classification accuracy, while also having implemented two synthetizable hardware models of this network: one based in High Level C Synthesis (using C++ in Vivado HLS), and another one by traditional RTL means (using VHDL in Vivado). In addition, we have managed to integrate these hardware blocks with the rest of the system; namely, the CPU, memory and the instruction pre-processor, which successfully replicate the results obtained by the software model. While more work needs to be done in order to obtain a complete and more realistic system, this project paves the way for future endeavours in the quest for automatic run-time loop parallelization.

Table of Contents

List of Tables

List of Figures

Abbreviations

ANN	=	Artificial Neural Network
CNN	=	Convolutional Neural Network
CPU	=	Central Processing Unit
DOP	=	Degree Of Parallelism
DSP	=	Digital Signal Pprocessor
FC	=	Fully Connected
FIR	=	Finite Impulse Response
FPGA	=	Field-Programmable Gate Array
FSM	=	Finite State Machine
GPU	=	Graphics Processing Unit
HLS	=	High Level Synthesis
IC	=	Integrated Circuit
IIR	=	Infinite Impulse Response
IP	=	Intellectual Property
LSTM	=	Long Short-Term Memory
LUT	=	Look-Up Table
OpenCL	=	Open Computing Language
RAM	=	Random Access Memory
RAR	=	Read After Read
RAW	=	Read After Write
ReLU	=	Rectified Linear Unit
RISC	=	Reduced Instruction Set Computer
ROM	=	Read-Only Memory
RTL	=	Register Transfer Language
PC	=	Personal Computer, or Program Counter
SIMD	=	Single Instruction Multiple Data
VHDL	=	VHSIC Hardware Description Language
WAR	=	Write After Read
WAW	=	Write After Write

Chapter 1

Introduction

The concept of using co-processors as hardware accelerators for mainstream computers dates back to the late 70's, and enjoyed an increased popularity in the subsequent decades. The first version of such co-processors was Intel's 80387: launched in 1980 [20], it was a dedicated floating-point arithmetic IC (Integrated Circuit) which was fully compliant with IEEE's 754-1985 standard, and was later embedded in Intel's 80486DX processor. In more recent years, we have seen a dramatic increase in the use of other kinds of hardware as accelerators, such as GPUs, DSPs and FPGAs, which, with the aid of programming languages such as OpenCL or CUDA, can be utilized to off-load some of the most computationally intensive work from the CPU. In addition, the stagnation in the increase of clock frequencies, combined with the transition to multi-threaded, multi-core processor designs implies that sequential code will no longer achieve the historical performance gains from advances in technology that it has obtained in the past [19]. The "Dark silicon problem" poses new challenges as well as new opportunities to IC designers: System-on-Chip designs, containing many different, specialized accelerators can take advantage of this "extra" silicon, dynamically turning on and off these parts of the chip as needed. All of this points to an apparently unavoidable necessity of changing the sequential-programming-paradigm to a parallel one. However, many factors play against designers when trying to exploit these parallel technologies: Ahmdals law imposes a harsh theoretical upper-limit to the achievable performance speed-up (even when only a small fraction of the code is sequential in nature); the continued use of sequential legacy code that was written years before the advent of mainstream parallel computers; and the sharp increase of the difficulty in writing well-optimized parallel code, which requires experienced (and often expensive) programmers to make efficient use of these circuits.

Another idea, which has proven itself very successful in mainstream CPUs (and other kinds of ICs), is the idea of *learning* from previous executions of code. After all, if a computer is repeating the same thing over and over again, wouldn't it be clever to try to use this information when the CPU steps again into the same part of the program? And thus, the idea of space and time locality led to the implementation of ever-growing cache memories, branch predictors alleviated the burden of jumping back to the same program address, while speculative execution and simultaneous multi-threading made good use of otherwise idling units.

But what if CPUs were able to learn *more* from a program, what if they could *adapt* themselves to the current program execution? The USA' Defense Advanced Research Projects Agency (DARPA) has recently invested 1.5 billion USD in a project that aims to develop hardware and software that can be reconfigured in real time, based on the kind of data being processed, adapting the computing architecture for the workload in milliseconds [13]. The recent advances in machine learning and in deep learning in particular along with its astonishing results (often surpassing human capabilities in a growing spectrum of fields), poses the question of whether this technology could also be used to increase the performance of a CPU. It's certainly not a crazy idea to use neural networks in this way, neither is it new: today's most advanced x86 CPU architecture, AMD's "Zen", utilizes a Perceptron-based Artificial Neural Network (ANN) to implement its branch predictor [8].

1.1 Previous Contributions

All these ideas led to my previous specialization project at NTNU, a project that proposed a novel architecture which could help solve the aforementioned problems. The concept behind it, is in its essence quite simple: embedding in a single IC a CPU, an ANN, and real-time reconfigurable logic (such as an FPGA). The ANN should then detect serial code (at run-time, by snooping on the CPU-memory bus) and accelerate the CPU by re-programming a parallel version of this code in the FPGA. Then, the next time the CPU tries to execute this code, it will be done by the faster, and more efficient FPGA-based accelerator, possibly freeing up the processor to attend to other tasks. Rather than having a plethora of idling, highly specialized dedicated IP cores (which is the case in today's SoC architectures), a re-configurable circuit could lead to a faster, more powerful and efficient use of the same silicon area. The proposed architecture is illustrated in figure 1.1.

The accelerators running on the FPGA should be dynamically adapted to the code running on the CPU at any given time. Due to limitations in both time and human resources, the specialization project was limited to generating the training data set and developing an ANN that would detect parallelizable code and possibly extract useful features from it. To deal with the extreme complexity of this task, we only considered loop-level parallelism (as opposed to task-level parallelism), with loops fixed in length and with an upper-bound

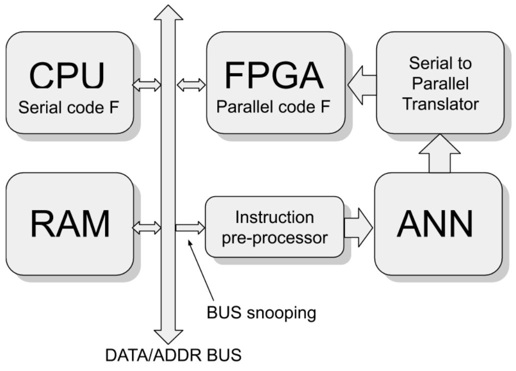


Figure 1.1: The Basic Blocks of the proposed Hardware

in the number of instructions per loop, while also avoiding nested loops. The focus on loop acceleration comes from the fact that programs spend a large percentage of their execution time in a small portion of their code (commonly known as the 90/10 rule) [1], and that most of this code is located within loop constructs. Additionally, not only is hardware loop parallelization completely transparent to the programmer, but it also allows parallelization of loops in cases where compilers fail to do so. The project made use of a very simple CPU called LT16x32 which was developed for academic purposes by the Chair of Electronic Design Automation of the Technische Universität Kaiserslautern. This is a uni-core, 16-bit, 3-stage pipeline, 20 instruction, in-order and non-speculative execution CPU. RAM and ROM can be conveniently accessed every clock cycle, therefore avoiding the stalling problems associated with memories running at slower clock speeds than the processor (thus requiring no cache memory). Even though these assumptions can be dangerously unrealistic, all these simplifications were a necessity that allowed us to focus on the core problems at hand, and if successful, future work could improve on the idea by lifting some or all of those restrictions.

1.2 Thesis Methodology and Contributions

This Master's Thesis builds upon the previously mentioned project, by adopting the same hypothesis, training data-sets (although it takes a different approach on the design of the ANN) and takes it a step further by implementing this neural network not only in software but also in hardware (as well as implementing a hardware block of the bus snooping unit), integrating it with the rest of the system. Additionally, this was done in parallel with another important goal: exploring one of today's mainstream high-level synthesis tools (Vivado HLS, which synthesizes a subset of the C, C++ and SystemC languages), and comparing its results with traditional RTL tools (Vivado, which supports VHDL development as well as integration of C and VHDL IP blocks). Furthermore, creating a functioning hardware Neural Network was a separate goal in itself. The working methodology was mostly experimental, but once the software ANN was finalized, this defined a golden model to which we were able to the compare results. Verification of the hardware modules was performed by simulating and comparing the outputs of the system to the outputs generated by the software model, for the same given input. A good amount of previous knowledge, along with the study of related articles, HLS tutorials, etc., provided a sufficient basis on which a first working model of the whole system (with the exception of the serial to parallel code translator) was successfully developed.

As a result, this work's contribution can be summarized as follows:

- Development of a PyTorch Loop-classifying Artificial Neural Network (ANN).
- Implementation of the ANN in C++.
- Synthetizable HLS version of the ANN.
- Synthetizable VHDL version of the ANN.
- Synthetizable HLS version of the instruction pre-processor.
- Synthetizable VHDL version of the instruction pre-processor.
- Synthetizable system integration of the LT16x32 with the instruction pre-processor, memory array controllers and ANN.
- Comparison of HLS vs RTL development of the same hardware blocks.
- Discovery of bugs in the Vivado HLS tool, as well as a discussion of its current limitations.

1.3 Thesis Structure

The rest of this document is organized as follows: chapter 2 discusses the theoretical background of ANNs as well as loop-level parallelism topics, while chapter 3 describes the pertaining elements of the previous project's work. Chapter 4 discusses and presents the results on the ANN's architecture and implementation of the software side, while chapter 5 does the same for the hardware implementations of the ANN,the instruction pre-processor and their integration into a system. The discussion of the results obtained so far is presented in chapter 6. Finally, chapter 7 points out recommendations towards possible future work, and concludes the work.

Chapter 2

Basic Theory

This project focuses on the improvement of an ANN architecture designed in a previous project (while still using the same datasets), as well as its hardware implementation in both HLS and RTL. The previous project on which this Thesis is based upon, will be further described in the next chapter.

2.1 Artificial Neural Networks

ANNs are digital structures loosely inspired on biological brains, consisting on many copies of a basic building block called *neuron* or *perceptron*. Their excellent ability to recognize patterns made them an extremely appealing choice for our ideas on runtime parallelizable loop detection. Even though our initial assessment on Recurrent Neural Networks (such as the LSTM) did not prove to be accurate, further investigation and testing has proven that CNNs have an outstanding performance when applied to our classification problem, at least in the terms that were previously decided upon. But before we move forward on explaining CNNs, we'll begin by explaining the simplest type of neural network: the multilayer perceptron.

2.1.1 Multilayer Perceptron

A *multilayer perceptron* is a type of ANN which consists of many neurons arranged in multiple layers, with the neurons of the l^{th} layer applying an affine transformation to the previous $(l-1)^{th}$ layer neuron's outputs, and mapped through some *activation function* [11]. Neurons arranged in this way are said to be *fully connected* (FC).

Mathematically, in the case of an FC feed-forward network, these outputs can be modelled as follows:

$$a_n^l = f\left(\sum_{i=1}^{S^l} w_{n,i}^l a_i^{l-1} + b_n^l\right) \tag{2.1}$$

where $w_{n,i}^l$ is weight of the i^{th} synapse connected to the input of the n^{th} neuron in the l^{th} layer; b_n^l is a bias term; f is the activation function, and S^l is the number of synapses connected to each neuron in the l^{th} layer. It could be graphically represented as shown in figure 2.1:

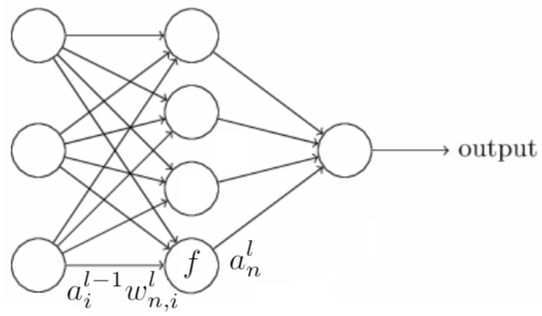


Figure 2.1: Graphical representation of a multi-layer fully connected ANN

The activation functions used in ANNs (such as the sigmoid or rectified linear unit) provide them with the non-linearities required so that they can learn any kind of arbitrary function. Among many other factors, the current success of deep learning can be attributed to some variants of these networks, the most prominent of them being the CNN, which will be described in the following subsection.

To train these ANNs, a large amount of training data is required, among their associated *targets* (i.e., the *desired output* for each given input). This training is done iteratively by measuring the error between the network's output for a certain input and the associated target value, by means of a *cost function*, and modifying the previously mentioned weights and biases accordingly, in order to minimize this difference.

2.1.2 Convolutional Neural Networks

Introduction

A *Convolutional Neural Network* (CNN) is a class of deep neural networks, most commonly designed to recognize image patterns directly from pixel representations (i.e., from two or three dimensional matrices) with minimal or no pre-processing required.

These networks were inspired in a connectivity pattern that resembles the organization of the brain's visual cortex. Each neuron responds to signals in a restricted region of the visual field known as the *receptive field*.

Like most neural networks, they are trained with some version of the back-propagation algorithm, that is, cleverly exploiting the chain-rule of differentiation, but since the hardware implementation of a CNN in this project does not deal with their training, this will not be further discussed; however, the interested reader can find more information about the subject in sources such as [11].

A typical CNN consists of an input and output layer, as well as multiple hidden layers sandwiched in between. The hidden layers of a CNN consist of convolutional layers (generally composed of multiple *channels* per layer), ReLU activation layers, pooling layers, and FC layers, all of which will be explained later. The following image shows the architectural representation of such typical CNN:

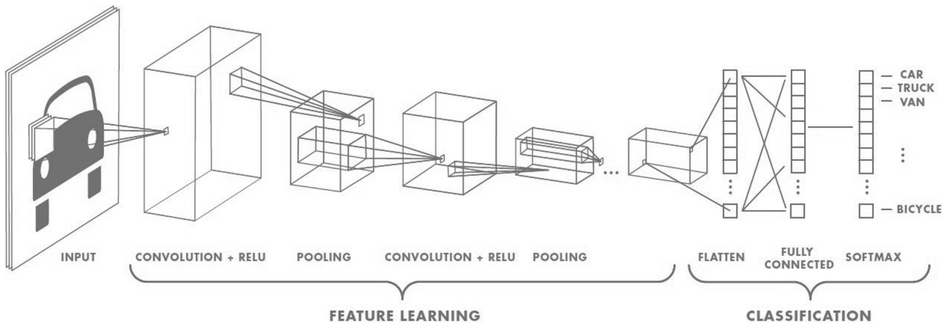


Figure 2.2: Graphical representation of a CNN. –Source: [18]

CNNs differ from the classic multilayer perceptrons mainly in their architecture: instead of connecting all neurons in one layer to all neurons in the following layer, and all connections having a different weight, neurons are instead connected locally from one layer to the next using the same weights, in the form of a *convolution* operation, hence its name. Strictly speaking, the actual mathematical operator used in CNNs is the *valid-cross-correlation*. This operation is what gives the CNN its translation invariance characteristics.

The convolution operation can be interpreted as an FIR filter, but in contrast to traditional algorithms where these filters are hand-engineered, the network *learns* the filter's coefficients, also referred to as *kernel*. This independence from prior knowledge and human effort in feature design is not only a major advantage, but most likely a necessity in what seems an unattainable task for humans, considering the complexity of the filters in today's extremely deep neural networks.

CNNs have wide applications in many fields, not only in image recognition, but also in image and video classification, segmentation, natural language processing and many other fields. As it will be shown later in image 3.2, our input data can be interpreted as 2D black and white images, where time is nothing more than another spatial variable. It is therefore not a wild idea to think that CNNs could produce satisfactory results, and in fact, this was one of the questions that I asked myself in the previous project's conclusion [14], where I stated that *"perhaps (...) convolutional networks could achieve better results"*. And indeed, they did.

Convolution Layer

Each convolutional neuron processes data only for its receptive field. Although it's possible to use traditional multilayer perceptrons to learn features and classify data, it's not exactly practical when it comes to images, since a very high number of neuron connections (weights) would be necessary (even in a shallow architecture) due to the inmense input sizes associated with images. For example, an FC layer for a very low resolution, black and white picture of size 320 x 200 pixels, would require 64000 weights **for each neuron** in the second layer. The convolution operation solves this problem by drastically reducing the number of parameters; e.g., by using a kernel of size 3x3 with 3 output channels, only 27 learnable parameters are required (or 30, if we consider the biases), regardless of the image's size.

The convolutional layer can be better understood as an FIR filter, with which most electrical engineers are more accustomed. On a high level, this operation can be better visualized in the following picture:

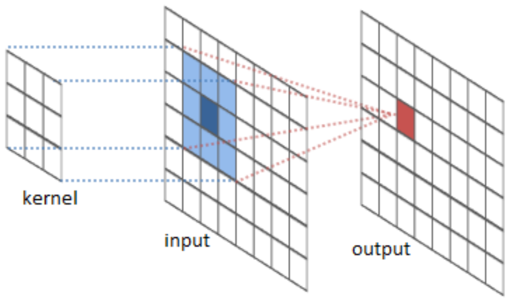


Figure 2.3: Convolution high level visualization. –Source: [16]

Each output is the sum of elements of the Hadamard product (i.e., entry-wise product) between the kernel and the tile composed of a window surrounding the input element in the input matrix. This process is repeated for all the input's elements to produce the final output of the convolution. In the case that no padding is used (as is the case of our project) the convolution product is only given for points where the signals overlap completely; values outside the signal boundary have no effect (i.e., it's a *valid* convolution, and not *full*). An horizontal or vertical stride can be defined, so that this window moves across the input jumping n_{stride} positions in the horizontal or vertical axis. A dilation factor can also be defined, and this dilates the input window so that it takes non-contiguous values. FIR filters as we know them, have a stride and dilation value of 1, which is generally the default value for CNNs and it's the value used in our design.

Let's consider an example by convolving a 4x4 matrix with a 2x2 kernel, as shown in the following figure:

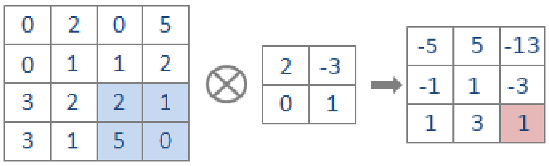


Figure 2.4: Convolution example

The weighted sum for the element highlighted in the output is:

$$o[3][3] = (2 \cdot 2) + (1 \cdot [-3]) + (5 \cdot 0) + (0 \cdot 1) = 1$$

If there is more than one input channel, the output is produced by convolving each channel with its own 2D kernel and adding the results to the same output value; analogously, if there are more than one output channel, the process is repeated, using the same input values but different kernels, for each output channel.

In the case that the input has dimensions (C_{in}, H, W) and the output has dimensions $(C_{out}, H_{out}, W_{out})$, the output values can be precisely described as [5]:

$$out(C_{out_j}) = bias(C_{out_j}) + \sum_{k=0}^{C_{in}-1} weight(C_{out_j}, k) \star input(k) \tag{2.2}$$

where $\star$ is the valid 2D cross-correlation operator previously explained, C denotes the number of channels, H is the height of the input planes, and W is their width.

Rectified Linear Unit Layer

In the context of ANNs, the rectifier is an activation function defined as the positive part of its argument, i.e.:

$$R(z) = max(0, z) \tag{2.3}$$

where z is the input to a neuron.

This is also known as a ramp function which is analogous to a half-wave rectifier, and can be visualized in the following figure:

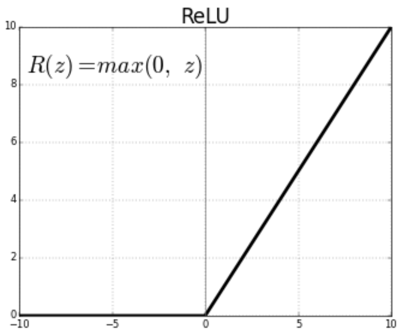


Figure 2.5: Rectifier function

The rectifier is, as of today, the most popular activation function for deep ANNs [17], not only because it's much simpler to calculate, but because it helps reduce the vanishing gradient problem: given that the derivative of this function for positive numbers is always equal to one, the successive product of derivatives that appears due to the chain rule, does not decrease the gradient in earlier layers. Moreover, the value of this derivative does not decrease as the weighted sum input z increases, which is the case with the sigmoid or with the hyperbolic tangent. In 2011, Xavier Glorot et al [9] demonstrated that using this unit as an activation function enables better training of deeper networks, compared to the other commonly used functions. Following our previous example, if we apply the ReLU layer to the output produced by the convolution, we'd have the following:

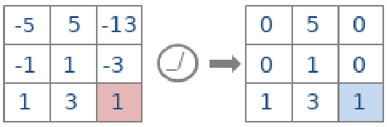


Figure 2.6: ReLU example

There are some variations of this function, such as the leaky-ReLU, which has a slope slightly greater than zero for negative values, instead of exactly equal to zero, but it hasn't been used in our project and will therefore not be further discussed.

Pooling Layer

Usually in a CNN architecture, a *Pooling layer* is inserted between successive convolutional layers. Its function is to progressively reduce the size of the feature maps, in order to reduce the amount of free parameters and computation in the network, and therefore to also control overfitting [4]. The Pooling Layer operates independently on every channel of the input and resizes it, using some fixed function, typically the max function (although it's possible to use avg, min, L2-norm, etc.). The most common form is a pooling layer with kernel size 2x2, with a stride of 2 along both horizontal and vertical directions, discarding 75% of the activations from the previous layer. Our architecture uses a 2x4 kernel size, with a horizontal stride of 2 and vertical stride of 4, discarding 87.5% of the activations.

In the case that the pooling function is max, the input has dimensions (C, H, W), the output has dimensions (C, H_{out}, W_{out}) and the kernel has size (kH, kW), the output values can be precisely described as [5]:

$$out(C_j, h, w) = \max_{\{m=0,...,kH-1\}} \max_{\{n=0,...,kW-1\}} input(C_j, stride_x \cdot h + m, stride_y \cdot w + n) \tag{2.4}$$

If we apply this to our original example, with a 2x2 kernel size and stride, we'd obtain the following output:

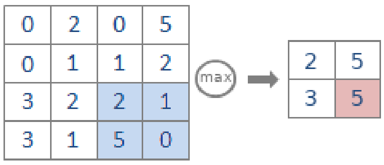


Figure 2.7: MaxPool example

Mathematically, for the highlighted values, the output value is:

$$out(1, 1) = \max\{2, 1, 5, 0\} = 5 \tag{2.5}$$

Fully Connected Layer

As it was shown in figure 2.2, a typical CNN consists of a sequence of convolutional, ReLU and pooling layers, after which there is a flattening operation and one or multiple FC layers. The flattening operation is nothing more than re-arranging the output tensor (3D matrix) from the previous convolutional layer into a single vector. This flattened output is then connected to an FC layer which was already described in section 2.1.1. The function of these FC layers is to perform classification based on the features extracted by the previous convolutional layers. Typically, the last FC layer uses a softmax activation function, which outputs a probability for each of the classification labels the model is trying to predict. However, in our project we chose to use a ReLU activation function for two reasons: first, it works just fine. Second, it's much less complex and easier to calculate, since the ReLU function doesn't have to deal with exponential functions.

2.2 Data Dependencies and Automatic Loop Parallelization

2.2.1 Loop-level parallelism and data dependencies

In order to define which high-level characteristics we want the ANN to extract from the code, we must first investigate the *meaning* of loop-level parallelism. As the term itself implies, *Loop-level parallelism* is a form of parallelism that seeks to extract certain tasks from loops which are in general independent from previous executions of the loop body, or in which the data dependencies involved allows the execution of these tasks in parallel or in a pipelined fashion. In case such tasks exist, there are various techniques that can be applied to decrease the execution time, for example: loop-pipelining, loop-unrolling, loop-merging and loop-splitting [6], which are currently being used in Vivado HLS. We will later present the first two of these techniques, since they have been extensively used in our ANN implementations.

Data dependencies in sequential statements

We have previously mentioned data dependencies, but we haven't discussed them yet, so let's introduce a few useful definitions. Given two consecutive, sequential statements S_1 and S_2, we define the following dependencies [10]:

- True (Flow, RAW) Dependence: S1 writes to a location later read from by S2.
- Anti Dependence (WAR): S1 reads from a location later written to by S2.
- Output Dependence (WAW): S1 and S2 write to the same location.
- Input Dependence (RAR): S1 and S2 read from the same location.

To preserve the sequential behaviour of a loop after it has been parallelized, the only kind of dependence that must be sequentially preserved is *Flow Dependence*; the others can be dealt with by making copies of the involved variables for each parallel process (this technique is known as *privatization*). Input dependencies are not truly dependencies, since input-dependent statements can be freely reordered without changing the result.

Dependencies in loops

When analyzing loops, we can distinguish between two types of dependencies:

- Loop-carried dependence
- Loop-independent dependence

In the case of loop-independent dependence, loops do not present dependence between iterations; therefore each iteration may be performed in parallel. Let's look at an example code for a 2-tap FIR filter:

```
#Iteration independent loop example:
for (int i = 1; i < n; i ++) {
    S1: tmp0 = x[i-1] * tap0;
    S2: tmp1 = x[i] * tap1;
    S3: out[i] = tmp0 + tmp1;
}
```

As we can see, there is a loop-independent, true dependence between (S1,S3) and (S2,S3), but iterations of the loop are independent of each other, therefore allowing for the execution of each iteration in a parallel or distributed manner.

In loop-carried dependencies however, statements in an iteration of a loop depend on the results of a previous iteration of the loop.

Let's take a look at an example of code which can be used for calculating the first *n* values of the Fibonacci sequence:

```
#Loop carried dependence example:
a[0] = 0;
a[1] = 1;
for (int i = 2; i < n; i ++) {
    S1: a[i] = a[i - 1] + a[i - 2];
}
```

In this case, it is clear that the statement S1 depends on the two previous executions of S1, and therefore cannot be distributed across different processing units.

Metrics of parallelism in loops

In our previous project, we had defined a metric with which we could determine the *Degree Of Parallelism* present in a loop, as simply 1 or 0 (parallelizable or non-parallelizable), depending on the ”kind” of loop (these kinds will be described in the next chapter). Given that our previous ANN architecture was not exactly successful in determining the DOP of a loop, introducing a more complex and precise definition would not only have taken some time to implement, but after all, if an ANN is unable to detect parallelism at this coarse level, then the prospects of detecting parallelism at a more refined level would be even lower. For these reasons, we have decided to keep this high-level definition, rather than trying to find the exact proportion of loop-carried dependent instructions.

2.2.2 Automatic Loop Parallelization

As we previously mentioned, there are various techniques that can be used to automatically parallelize loops, some of which will be discussed next.

Loop Pipelining

Pipelining allows operations to be computed concurrently, that is, it’s not necessary to wait for the completion of all operations before the next one can start. In a loop, this means that operations can be overlapped between consecutive executions of the loop body. Let’s suppose we have the following loop:

```
#Loop with dependencies:
for (int i = 0; i < N-1; i ++) {
    S1: a = x[i] * y[i];
    S2: b = f(a);
    S3: c = g(b);
}
```

where f(a) is some operation that depends on the result of a, etc. (if there were no dependencies, all three operations could be executed in parallel). Without loop pipelining, we would have the following execution pattern: `S1(i=0), S2(0), S3(0), S1(1), S2(1), S3(1), S1(2), S2(2), S3(2)`, etc, which would take $3 \cdot N$ clock cycles to complete. In contrast, when pipelining the loop, we can use the hardware that computes S2 and S3 in parallel with S1, allowing for the following execution pattern: `S1(0), S1(1) + S2(0), S1(2) + S2(1) + S3(0), S1(3) + S2(2) + S3(1)`, etc., which would take $N + 2$ clock cycles.

Graphically, this can be represented as follows:

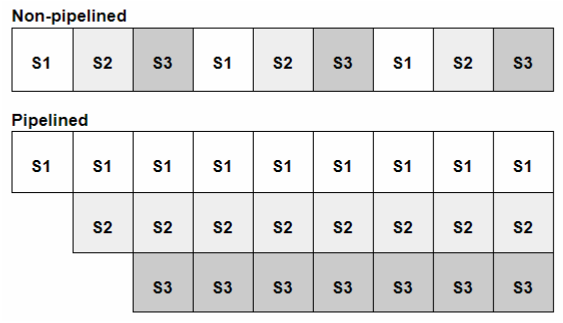


Figure 2.8: Loop Pipelining

In the case shown, pipelining the loop results in hardware roughly three times as fast (when $N >> 1$), at the expense of slightly more complex steering and control logic, more execution units (in case these instructions share some) and perhaps a few extra registers.

Loop Unrolling

Loop unrolling consists of making multiple copies of the loop body, and instead of using the same hardware in different clock cycles, copies of the same hardware are instantiated so that operations can be performed in parallel, if allowed by data dependencies and available resources. For instance, if we consider the following loop:

```
#Rolled Loop:
for (int i = 0; i < 3; i ++) {
    z[i] = x[i] * y[i];
}
```

leaving the loop rolled would require 4 clock cycles to complete, but if we partially unroll the loop with a factor of two, we end up with the following loop instead:

```
#Partially unrolled Loop:
for (int i = 0; i < 1; i ++) {
    z[i] = x[2*i] * y[2*i];
    z[i] = x[2*i + 1] * y[2*i + 1];
}
```

cutting the execution time in half.

If instead we unroll the loop completely, we end up with the following code:

```
#Completely unrolled Loop:
z[0] = x[0] * y[0];
z[1] = x[1] * y[1];
z[2] = x[2] * y[2];
z[3] = x[3] * y[3];
```

This would complete all the operations in a single clock cycle, at the expense of four times the hardware. It should be noted that memory accesses are also multiplied by the same factor, so care should be taken when applying this technique: if too many operations in a loop require memory accesses, then the speed limitation will be given by the available memory channels, and in the case of accessing external DRAM, the benefits of unrolling the loop won't be perceivable. In contrast, if a loop is compute-intensive and not many memory accesses are required, the improvements can be quite substantial. In the case of FPGAs, it's possible to achieve sizable speed-ups even when memory accesses are numerous, since they contain distributed memory blocks all across the chip.

Alas, given the short time available for this project, a hardware block that transforms the detected loops into parallel versions was not be implemented, but some ideas were discussed. In particular, columns of execution units, each unit implementing some of the CPU instructions could be connected to the next column through reconfigurable steering logic and registers, while a reconfigurable control unit would implement the execution sequence and branching instructions. And so, after a loop is detected, the array would be populated with the instructions in the loop body, trying to parallelize as many instructions as possible, while complying with the data dependencies.

Chapter 3

Previous Work

As it was already mentioned, this project builds upon the work done for a previous project called *Deep Learning Based CPU Acceleration* [14], and therefore, in order to understand and write a self-contained Thesis, it is necessary to describe some of the work previously done. The main goal of that project was to generate training datasets along with the desired training targets (i.e., degree of parallelization, head and tail of the loop, trip count, etc.), as well as the design and training of an ANN able to detect these parameters. For this purpose, various tools were developed, which will be briefly described next.

3.1 Dataset Generation

3.1.1 LoopGen: A Parametric Assembly Loop Code Generator

In order to train an ANN capable of recognizing parallelizable loops, we first had to create a dataset on which we could train it. Given that the CPU we were (and still are) working with was an in-house development, there exists no C-compiler for it and we were therefore forced to come up with our own assembly programs. Since the number of training examples required is in the order of hundreds (if not thousands) we came up with the idea of generating parametrizable assembly loops, in which the kind of loop, registers used, number of memory arrays, number of iterations, etc., are completely parametrizable and so, by randomizing these parameters (within certain restrictions), we were able to come up with hundreds, or even thousands of different assembly programs. Making it possible to change the registers was deemed necessary since we didn't want the ANN to be biased by arbitrarily chosen registers which it could learn to associate with certain kinds of loops. This way of creating different assembly programs is, in fact, nothing more than a clever trick to perform *data augmentation* [12]: a machine learning technique which consists in artificially inflating the training set with label-preserving transformations.

LoopGen consisted of three C++ functions that will be briefly described in the following paragraphs.

The first function, called "`instr_str`", simply takes the type of instruction and its parameters (such as destination register, source registers, labels, modes, etc.) and returns a string with the correct assembly syntax for the given instruction. This is necessary to make possible the parametrization described before while keeping the registers and labels consistent with the desired behaviour of the assembly program.

The second and third functions, called "`loop_independent`" and "`loop_dependent`" respectively, take a plethora of parameters, such as loop identifier, kind of loop, temporary registers, array registers, array names, etc., and returns a string containing fully working assembly code automatically generated with those parameters.

The function "`loop_independent`" creates four different kinds of loops that were deemed to be fully parallelizable. The first kind, "`add_constant`", generates a program that adds a random constant value between -128 and 127 to the elements of the given arrays. The second kind, "`add_arrays`" generates code that adds two or more arrays and stores the result in the first of those arrays. The third kind of loop, "`swap_arrays`", generates a program that, as the name suggests, swaps pairs of arrays. The fourth and last kind of parallelizable loop, "`fir_Filter`" creates a parametric, fully functional FIR filter for a CPU that does not possess a multiply instruction; an idea inspired on the work by Y. C. Lim and B. Liu. In [15], they demonstrated that an FIR filter with very small frequency response ripple magnitude can be realized using two power-of-two terms for each coefficient value. This allowed us to implement a fast filter using shifting instructions instead of multiplications, by hard-coding each filter coefficient into the immediate values of the `lsr` (Logic Shift Right) instructions themselves.

Analogously, the function "`loop_dependent`" creates four different kinds of loops that were deemed to be non-parallelizable. The first kind of iteration-dependent loop, "`fibonacci`" returns a program that calculates Fibonacci-like sequences, given by the recurrence formula:

$$f[n] = \sum_{i=1}^{N} f[n-i],$$

with $f[0] = f[1] = ... = f[N] = 1$. It writes the sequence in an array, and when N is equal to 2, it reduces back to the classic Fibonacci sequence. The second kind, "`dep_array_sum`", returns a program that calculates the sum of elements in one or many arrays, with each successive element being added or subtracted depending on whether the previous partial sum was positive or negative. The third kind, "`binom_coeffs`", returns a program that calculates the binomial coefficients, which count the number of ways of selecting k elements out of a set of n elements. This program actually returns a recursive algorithm, which is computed by a loop with variable bounds.

The last kind of iteration-dependent loop, "`IIR_filter`" was implemented using the same ideas as the FIR filter, that is, performing divisions with shift-right instructions. It can be shown that that the output value at a certain point in time n depends on previous outputs, therefore generating a loop whose output depends on previous iterations.

In both cases, whether it's an iteration-independent or dependent loop, after the main program ends, there is an infinite loop that does nothing but jump back to itself; the arrays are defined and memory is allocated for them using assembly directives.

3.1.2 Generating the datasets with LoopSim: from LoopGen to CPU execution traces

Since we wanted the ANN to detect loops by inspecting the signals inside a CPU, we needed not only to create hundreds of assembly programs, but to compile them to machine code and produce the traces that would be found in the CPU-memory bus (in reality, this signal would be found in the instruction register's output). Additionally, we had to be able to do all of this in an automated way. Fortunately, the assembly compiler was already implemented, as were some parts of the CPU simulator; solving this was then a matter of adapting the existing code to our needs, and creating what needed creation. The flow of LoopSim can be visualized in figure 3.1.

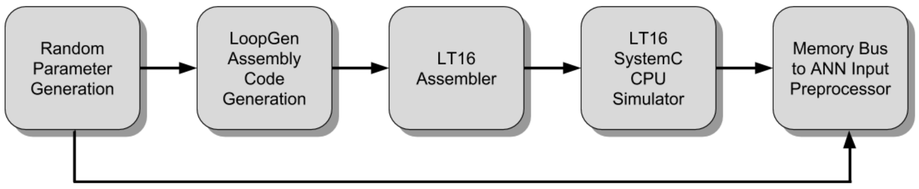

Random Parameter Generation
LoopGen Assembly Code Generation
LT16 Assembler
LT16 SystemC CPU Simulator
Memory Bus to ANN Input Preprocessor

Figure 3.1: LoopSim Data Flow

First, the parameters to be used with the functions in LoopGen were randomly generated, using uniform probability distributions, taking care of the restrictions imposed by each kind of loop. Some of these parameters also became the targets for the ANN: a Parallelizable loop (i.e., a loop created by the function `loop_independent`) was associated with the target (1,0) and a Non-Parallelizable one (i.e., a loop created by the function `loop_dependent`) with (0,1), that is, the classes we wanted the ANN to identify were (P, NP). Then, the assembly code generated by LoopGen was assembled into machine code using the compiler provided by the creators of the CPU, which was adapted and embedded into LoopSim. A SystemC CPU and its memory models were then instantiated, and the string output from the compiler was then loaded to the RAM model.

The execution of any instruction was performed by calling the associated function in the processor class; some of these instructions had to be implemented since they were missing from the model. Given that we were provided only with a SystemC memory model that can hold machine code, and with a SystemC CPU model which only contains the register file and the functions associated with the instructions, it was necessary to implement a *decoder/execution unit* which implements the operation of the CPU. This was done as follows: after the CPU object is instantiated, it is set into the reset state: the register file, including the Program Counter are set to 0. Then, an instruction is fetched from the memory address given by the Program Counter and it is decoded according to its opcode; afterwards, the corresponding processor function is called with the corresponding parameters. The function call generates a new status with which the CPU state is updated (in particular, the Program Counter is incremented), and the process begins once again with a new instruction. In the case of a branching instruction, the branch delay slot and branching address are taken care of.

3.1.3 ANN Input Pre-processor

After having generated the CPU execution traces, it was deemed convenient to represent this information using one-hot encoded vectors as data input for the ANN. Since the opcodes of the CPU instructions have no ordinal relationship, directly feeding the ANN with these 16-bits numbers and allowing the model to assume a natural ordering between instructions may have resulted in poor performance or unexpected results [3]. All instructions contain at least one of the following categories, each of which were encoded using one-hot vectors:

- **Opcode**
- **Mode**
- **Condition**
- **Destination Register**
- **Source Register A**
- **Source Register B**
- **Immediate value**

The first category, i.e., **Opcode** was encoded as follows:

```
(ADD, SUB, AND, OR, XOR, LSH, RSH, ADDI, CMP, LDIMM, LDPTR,

STPTR, BIMM, BREG, CIMM, CREG, TRAP, RETI, BRT, TST)
```

Therefore, as an example, an ”AND” instruction’s opcode would be encoded as: (0, 0, 1, 0, ... 0). This vector has 20 dimensions, since all ”load from pointer” instructions (LD08, LD16, LD32) are encoded to the same vector, and the same goes for the ”store pointer instructions”.

The remaining categories were encoded in a similar fashion.
In the previous project, the immediate value was also encoded with one-hot vectors; however, it was found that removing the immediate from the encoding did not result in any accuracy degradation of the new ANN’s accuracy.

All these one-hot vectors were then concatenated and fed to the ANN in the following way:

```
(Opcode, Mode, Condition, Destination Register, Source
Register A, Source Register B)
```

These vectorial time sequences were created and stored into a *comma separated values* (.csv) file as the CPU simulator executed each instruction. The execution of an IIR filter loop generates a sequence of vectors which can be visualized in figure 3.2:

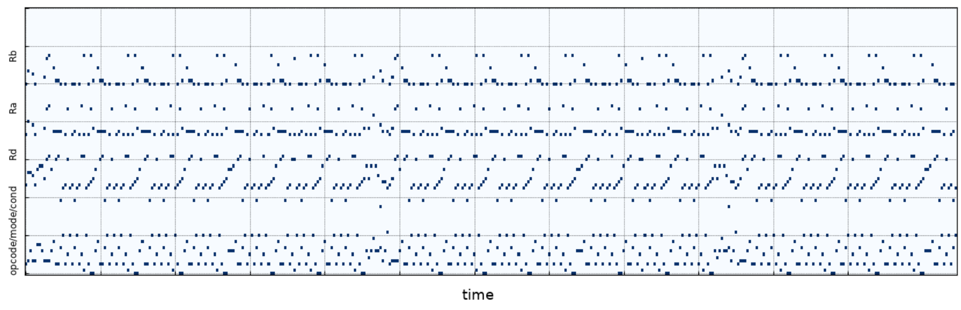


Figure 3.2: Graphical representation of the first 400 vectorized instructions executed by an IIR filter

It was perhaps this way of visualizing the execution traces, that led us to the hypothesis that CNNs could perform well in this problem, since this is nothing more than a black and white 2D image, and CNNs are known for their excellent performance with them.

3.1.4 LoopOracle v1: A Loop Classifier LSTM ANN

The ANN architecture implemented in our previous project consisted of an FC layer which takes the vectorized instruction sequences as inputs and creates an adequate encoding for an LSTM network. This input structure, given that it's fed with one-hot vectors, acts as a sort of embedding layer. The LSTM network consisted of two stacked LSTM layers with a hidden size of 256. At the output of this layer, a 256-to-2 FC layer classified the loop as either parallelizable or as non-parallelizable, i.e., the targets previously described (P, NP). Even though LSTMs seemed the obvious choice to pick, which were also recommended by experts on the field, the results achieved were quite underwhelming: an accuracy of around 80% in a problem where random chance would have a 50% success rate, was less than impressive.

The ANN was implemented using *PyTorch*, an open-source machine learning library for Python (based on Torch) which is used for applications such as natural language processing. Primarily developed by Facebook's artificial-intelligence research group, it provides a very high-level of abstraction for implementing deep ANNs and it's surprisingly easy to use.

Chapter 4

Software Architecture and Implementation

4.1 LoopOracle v2: Improving on the previous Loop Classifier ANN

Our new ANN architecture consists of a pretty typical CNN: three convolutional layers followed by a fully connected classifier. It uses almost the same input data as our previous ANN, but the number of instructions was cut down from 512 to 256 (this is around the maximum number of instructions found in any loop body generated by LoopSim), and after removing the one-hot encoded immediate value, the instruction encoding ended up with a dimension of 76. The architecture can be visualized in figure 4.1:

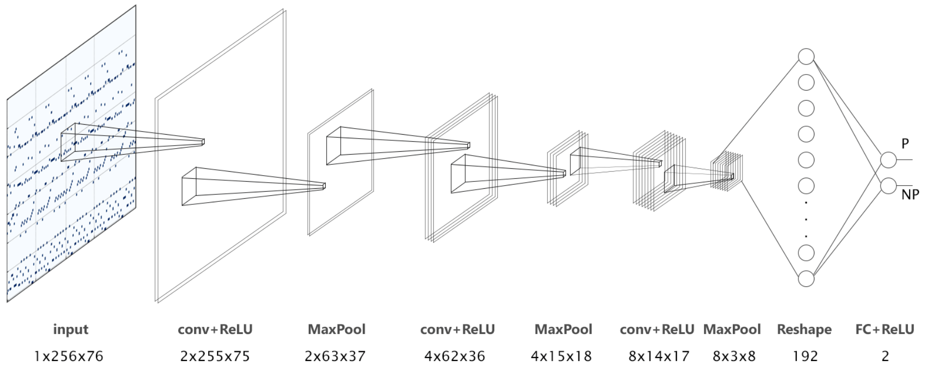


Figure 4.1: Graphical representation of the CNN architecture

4.1.1 Training a Pytorch CNN

As in our previous project, we started by developing the ANN in Python with the aid of the PyTorch library. Given that this allows for a very high-level design of ANNs, it's worth going through some of the most important lines of the code developed, starting with the definition of the model and forward pass:

```python
class ConvolutionalLoopOracle(nn.Module):
    def __init__(self):
        super(ConvolutionalLoopOracle, self).__init__()

        # Convolutional layers:
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=2, kernel_size=2)
        self.conv2 = nn.Conv2d(in_channels=2, out_channels=4, kernel_size=2)
        self.conv3 = nn.Conv2d(in_channels=4, out_channels=8, kernel_size=2)
        self.pool2x4 = nn.MaxPool2d(kernel_size=[2, 4], stride=[2, 4])

        # Output layer:
        self.fc = nn.Linear(8 * 8 * 3, 2)

    def forward(self, x):
        x = functional.relu(self.conv1(x))
        x = self.pool2x4(x)
        x = functional.relu(self.conv2(x))
        x = self.pool2x4(x)
        x = functional.relu(self.conv3(x))
        x = self.pool2x4(x)
        x = x.view(-1, 8 * 8 * 3)
        x = functional.relu(self.fc(x))
        return x
```

As shown in figure 4.1, the architecture consists of three convolutional + ReLU + MaxPooling layers, each layer duplicating the amount of channels of the previous one, while reducing the resolution of the feature space roughly by eight, since the MaxPooling operation has a kernel size of 2x4 and same stride. After these layers, only 192 activations make it to the end, which are then connected to an FC layer with this same number of inputs, and two outputs with a ReLU activation function, corresponding to the two classes previously discussed, (P, NP). After defining the CNN, we must define a function with an algorithm to train it:

```python
def train_model(model, learning_rate, num_epochs):
    criterion = nn.SmoothL1Loss()
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
    for i in range(num_epochs):
        # use 80% of dataset as training:
        for j in range(int(0.8*data.__len__()/5)):
            x_t, y_t = data.get_batch(j*5, (j+1)*5)
            optimizer.zero_grad()
            y_hat = model(x_t)
            loss = criterion(y_hat, y_t)
            loss.backward()
            optimizer.step()

        # Validation:
        with torch.no_grad():
            optimizer.zero_grad()
            # use 15% of dataset as validation:
            x_t, y_t = data.get_batch(int(0.8 * data.__len__()), int(0.95 * data.__len__()))
            y_hat = model(x_t)
            val_loss = criterion(y_hat, y_t)

        if val_loss < min_val:
            min_val = val_loss
            torch.save(model.state_dict(), 'best_model.pt')
```

As we said before, we won't go through the mathematical details of the back-propagation algorithm since this won't be done in hardware, but it's perhaps worth explaining some of the concepts involved in it. The first step of training an ANN consists of defining a *loss function*, that is, the function to be minimized. In our case, we have chosen the *Smooth L1 Loss*, defined as follows:

$$L_{1Smooth} = \begin{cases} |x| & \text{if } |x| > 1; \\ x^2 & \text{if } |x| \leq 1 \end{cases}$$

This function is basically the classic L1 loss, but with a smooth, differentiable behaviour for values around zero. Although the L2 norm is more precise and better in minimizing prediction errors, the L1 norm produces sparser solutions, ignores more easily fine details and is less sensitive to outliers. Sparser solutions are good for feature selection in high dimensional spaces, as well for prediction speed [2].

The *Adam* (Adaptive Moment Estimation) optimization algorithm is an enhanced version of the stochastic gradient descent algorithm, which in contrast to the latter, computes individual adaptive learning rates for the different parameters of the network using the first and second derivatives of the loss function; it also takes into consideration how fast the weights were changing in the previous steps.

The training loop itself runs `num_epochs` times, and it uses 80% of the whole dataset for the training of the ANN. The forward method is called with `model(x_t)` and the predicted results are stored in the variable `y_hat`. After the forward pass, the difference (loss) between the target values `y_t` and the predictions `y_hat` is calculated by calling the function `criterion(y_hat, y_t)`. When calling the forward function along the loss calculation, a computational graph of these operations is created, where each operation in this process is registered, allowing to perform the back-propagation pass by simply calling the function `loss.backward()`: this calculates all the gradients of the loss function respect to each parameter in the network. After the gradients are calculated, the Adam optimizer updates the values of the network's weights, and the network is ready to be trained again the next epoch. But before we do that, we first compute the loss in the validation set, which consists of 15% of the whole dataset (however, note that the network does not learn on this data). The idea behind this is having a measure of how well the ANN is performing on new, unseen data, so we can perform *early stopping* [11], which is another regularization technique used to avoid overfitting. For this reason, we keep track of the minimum value of the loss function on the validation set in each iteration, and in case this value decreases from the previous minimum, we store the model into a file that can be loaded in the future. Finally, after training the model `num_epochs` times, we perform a final test using the remaining 5% of the dataset.

An initial learning rate of 0.001 is a fairly common practise, but we have to keep in mind that when using Adam's algorithm, this value will be accordingly adapted and it only makes a significant difference in the initial training steps. A number of epochs of 250 is much more than it was found to be necessary, with the validation loss reaching its minimum after around 50 epochs, as can be seen in figure 4.2.

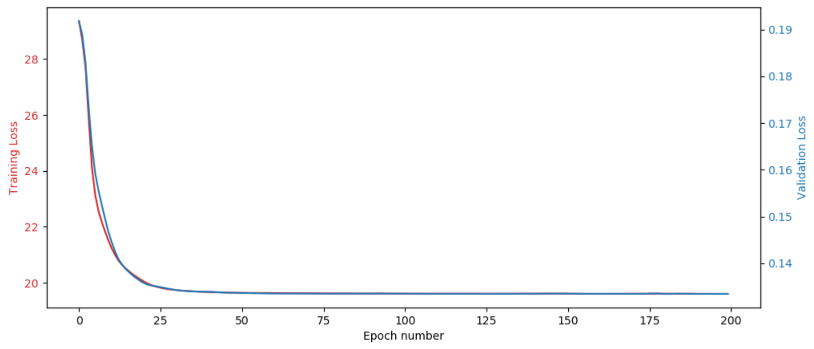


Figure 4.2: Training and validation loss

A final test accuracy of around 99.8% was reached. A lot of testing with different network sizes was done, and even though more optimizations and size reductions may be possible, the final size was deemed reasonable. Given that one of our objectives is to investigate the hardware implementation of a CNN, a very simple architecture would lead to a very small design, which would not appropriately reflect the technological challenges that a typical hardware CNN implementation would entail. On the other hand, it is necessary to keep the architecture small enough so that it fits in a typical FPGA at a reasonable speed, and the sizes chosen for the CNN's layers provide a satisfactory balance between these opposing objectives.

4.2 Lowering the abstraction level: transforming a PyTorch CNN to C++ code

After successfully developing and training the CNN in Python, the next step was to implement this architecture in a language that can be later synthesized by Vivado HLS. Using already developed C++ libraries was considered, but the complexity and generality of the code of the libraries found seemed excessive in comparison with our very limited needs, given that trying to understand these libraries would have taken considerable time and effort. As for the libraries and frameworks available for FPGA ANNs, they suffer a similar problem, with the addition that these libraries are developed with the intention of using the FPGA as an inference accelerator for the PC, much in the same way one would use a GPU. None of them, at least to the best of my knowledge, are tailored to be used as a standalone hardware block inside an SoC. Additionally, re-using an already developed library would have taken away all the experience and challenges that implementing a hardware neural network poses. And finally, the optimization possibilities and opportunities of an in-house developed neural network are significantly higher than what would be possible if an existing library was adapted.

Vivado HLS currently supports three high-level languages: C, C++ and SystemC. The initial choice was to use C as development language; however, it turned out that Vivado HLS does not support arbitrary fixed-point number representations in C, so the code was later ported from C to C++ instead of SystemC, since the syntax differences are minimal (though it must be said that both languages are quite finicky when passing an array to a function). Since the only thing we must implement in hardware is the forward pass, only the convolution, ReLU, max-pooling, reshape and affine functions had to be implemented.

If we recall equation 2.2, expanding the valid 2D cross-correlation operator, we obtain the following equation:

$$out(C_{out}, m, n) = b(C_{out}) + \sum_{C_{in}=0}^{N_{C_{in}}-1} \sum_{i=0}^{Kr-1} \sum_{j=0}^{Kr-1} in(C_{in}, m+i, n+j) \cdot w(C_{out}, C_{in}, i, j) \tag{4.1}$$

where:

- $out(C_{out}, m, n)$ is the output value of output channel C_{out} and coordinates m, n
- $b(C_{out})$ is the bias of channel C_{out}
- $in(C_{in}, m+i, n+j)$ is the input value of input channel C_{in}
- $w(C_{out}, C_{in}, i, j)$ is the weight of the corresponding kernel
- $N_{C_{in}}$ is the number of input channels
- Kr is the size of the kernel

Once this equation is fully understood, it's quite easy to implement in C++; so easy in fact that much to my amazement, the code written for the 2D convolution function worked on the first try after it had just been written!

```
template <size_t x_in_dim, size_t y_in_dim, size_t ch_in, size_t ch_out, size_t kernel_size>
void Conv2d(float (&dataIn)[ch_in][x_in_dim][y_in_dim],
            float (&dataOut)[ch_out][x_in_dim-kernel_size+1][y_in_dim-kernel_size+1],
            float (&C2dWeights)[ch_out][ch_in][kernel_size][kernel_size], float (&C2dBias)[ch_out])
{
    int i, j, m, n, in, out;
    for (out=0; out < ch_out; out++){
        for (m=0; m < x_in_dim-kernel_size+1; m++){
            for (n=0; n < y_in_dim-kernel_size+1; n++){
                dataOut[out][m][n] = C2dBias[out];
                for (i = 0; i < kernel_size; i++){
                    for (j = 0; j < kernel_size; j++){
                        for (in = 0; in < ch_in; in++){
                            dataOut[out][m][n] = dataOut[out][m][n] + dataIn[in][m+i][n+j]*C2dWeights[out][in][i][j];
                        }
                    }
                }
            }
        }
    }
}
```

As it can be seen, the lines in this code corresponds one to one with equation 4.1: the first three `for` loops correspond to traversing the coordinates of the output variable, while the remaining three `for` loops correspond each to one of the three summations.

Now that we have the code for the convolution operator, it is possible to calculate the amount of multiply-accumulate operations required by each layer, with a very simple equation:

$$\#MulAcc = ch_{out} \cdot (x_{in_{dim}} - ker_{size} + 1) \cdot (y_{in_{dim}} - ker_{size} + 1) \cdot ker_{size}^2 \cdot ch_{in} \quad (4.2)$$

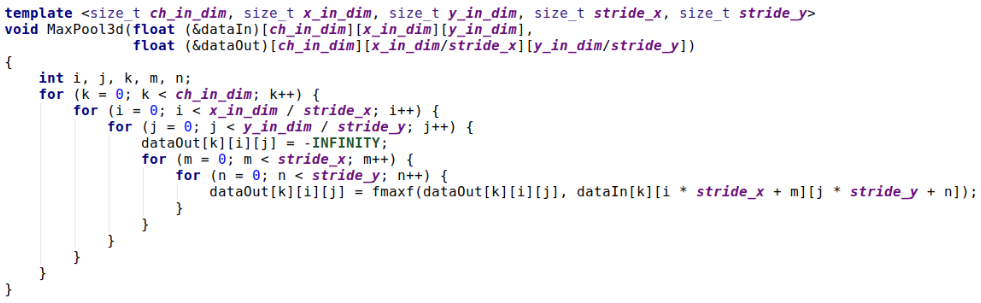
All layers have a kernel size of 2 (both in the horizontal and vertical dimensions), and the dimensions of each convolution operation are those shown in figure 4.1. Therefore, the number of multiply-accumulate operations for the first layer is 153000, 71424 for the second layer and 30464 for the third layer. A clever reader can already be thinking of ways to get rid of 153000 multiply operations, given the nature of our input data.

As for the implementation of the ReLU function, the code is pretty straight-forward, it just implements equation 2.3 for a 3D array.

```
template <size_t x_in_dim, size_t y_in_dim, size_t z_in_dim>
void ReLU3d(float (&dataIn)[x_in_dim][y_in_dim][z_in_dim], float (&dataOut)[x_in_dim][y_in_dim][z_in_dim])
{
    int i,j,k;
    for (i = 0; i < x_in_dim; i++)
        for (j = 0; j < y_in_dim; j++)
            for (k = 0; k < z_in_dim; k++)
                dataOut[i][j][k] = (dataIn[i][j][k]>0) ? dataIn[i][j][k] : 0;
}
```

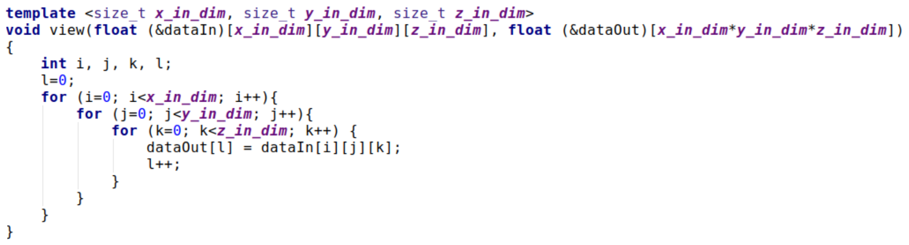
An analogous function was implemented for the unidimensional case, which is needed at the output of the FC layer.

As for the MaxPooling function, this is again a straight-forward implementation of equation 2.4. The MaxPool function traverses the input array of every channel and outputs, for each channel, the maximum value found in a window of size $[stride_x, stride_y]$.



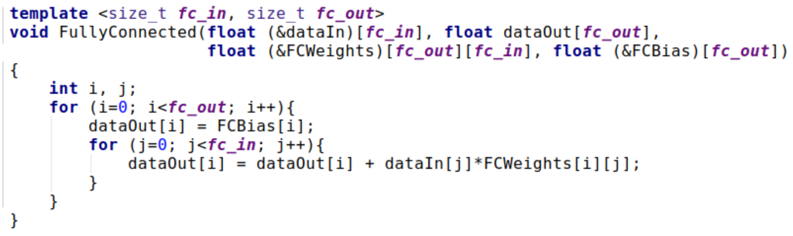
```
template <size_t ch_in_dim, size_t x_in_dim, size_t y_in_dim, size_t stride_x, size_t stride_y>
void MaxPool3d(float (&dataIn)[ch_in_dim][x_in_dim][y_in_dim],
               float (&dataOut)[ch_in_dim][x_in_dim/stride_x][y_in_dim/stride_y])
{
    int i, j, k, m, n;
    for (k = 0; k < ch_in_dim; k++) {
        for (i = 0; i < x_in_dim / stride_x; i++) {
            for (j = 0; j < y_in_dim / stride_y; j++) {
                dataOut[k][i][j] = -INFINITY;
                for (m = 0; m < stride_x; m++) {
                    for (n = 0; n < stride_y; n++) {
                        dataOut[k][i][j] = fmaxf(dataOut[k][i][j], dataIn[k][i * stride_x + m][j * stride_y + n]);
                    }
                }
            }
        }
    }
}
```

The function in charge of reshaping the output of the last convolutional layer, called `view`, traverses the 3D input array and writes those values into a 1D output array, as shown in the following code:



```
template <size_t x_in_dim, size_t y_in_dim, size_t z_in_dim>
void view(float (&dataIn)[x_in_dim][y_in_dim][z_in_dim], float (&dataOut)[x_in_dim*y_in_dim*z_in_dim])
{
    int i, j, k, l;
    l=0;
    for (i=0; i<x_in_dim; i++){
        for (j=0; j<y_in_dim; j++){
            for (k=0; k<z_in_dim; k++) {
                dataOut[l] = dataIn[i][j][k];
                l++;
            }
        }
    }
}
```

The last function implements an FC classification layer, which performs the affine transformation specified by equation 2.1:



```
template <size_t fc_in, size_t fc_out>
void FullyConnected(float (&dataIn)[fc_in], float dataOut[fc_out],
                    float (&FCWeights)[fc_out][fc_in], float (&FCBias)[fc_out])
{
    int i, j;
    for (i=0; i<fc_out; i++){
        dataOut[i] = FCBias[i];
        for (j=0; j<fc_in; j++){
            dataOut[i] = dataOut[i] + dataIn[j]*FCWeights[i][j];
        }
    }
}
```

It's easy to see that the FC layer requires 192x2 of multiply-accumulate operations; i.e., 384. The outputs of this FC layer are passed through the ReLU activation function which has been already described for the 3D case.

The forward function instantiates these functions as required, in order to implement the CNN architecture described before, as shown in the following code:

```
//..::First convolutional layer::..
// kernel size = 2x2, 1 input channel, 2 output channels
Conv2d<in_x, in_y, data_in_CH, C1_out_CH, kernel_size>(DataInTr, DataOutC1, C1W, C1B);
ReLU3d<C1_out_CH, C1_out_x, C1_out_y>(DataOutC1, DataOutC1);
MaxPool3d<C1_out_CH, C1_out_x, C1_out_y, stride_x, stride_y>(DataOutC1, DataOutMP1);

//..::Second convolutional layer::..
//kernel size = 2x2, 2 input channels, 4 output channels
Conv2d<R1_out_x, R1_out_y, C1_out_CH, C2_out_CH, kernel_size>(DataOutMP1, DataOutC2, C2W, C2B);
ReLU3d<C2_out_CH, C2_out_x, C2_out_y>(DataOutC2, DataOutC2);
MaxPool3d<C2_out_CH, C2_out_x, C2_out_y, stride_x, stride_y>(DataOutC2, DataOutMP2);

//..::Third convolutional layer::..
//kernel size = 2x2, 4 input channels, 8 output channels
Conv2d<R2_out_x, R2_out_y, C2_out_CH, C3_out_CH, kernel_size>(DataOutMP2, DataOutC3, C3W, C3B);
ReLU3d<C3_out_CH, C3_out_x, C3_out_y>(DataOutC3, DataOutC3);
MaxPool3d<C3_out_CH, C3_out_x, C3_out_y, stride_x, stride_y>(DataOutC3, DataOutMP3);

//..::Fully connected layer::..
view<C3_out_CH, R3_out_x, R3_out_y>(DataOutMP3, dataOutView);
FullyConnected<FC_in, out_x>(dataOutView, dataOut, FCW, FCB);
ReLU1d<out_x>(dataOut, dataOut);

if (dataOut[0] < dataOut[1]){
    CnnOut[0] = 0;
    CnnOut[1] = 1;
} else {
    CnnOut[0] = 1;
    CnnOut[1] = 0;
}
```

Figure 4.3: Forward function

In contrast to PyTorch, all dimensions of every array must be calculated and defined beforehand; the values of the weights and biases were imported from the trained PyTorch model into floating-point arrays. All these details were left out from the code shown here, for the sake of clarity.

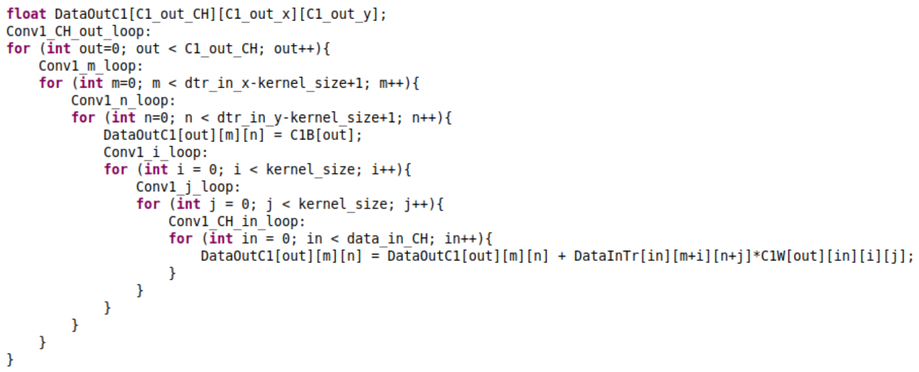
As anyone can intuit, this code is probably the most naive way to implement an ANN and does not contain any optimizations, but it is however quite readable. Even though it's not the final version of the C++ code used in Vivado HLS, it's necessary to show it here in order to better understand what ended up being implemented in Vivado, which is somewhat convoluted and nowhere near as readable.

Chapter 5

Hardware Architecture and Implementation

5.1 Vivado HLS implementation

Once we had a fully working C++ version of the CNN that produces identical results to the PyTorch golden model, we were able to start working on adapting the code so that it becomes synthetizable. The first step taken was to create specialized functions for each layer in order to be able to optimize each one of them. Inside each layer's convolutional function, loops were named with the number of layer and in accordance with the variables defined in equation 4.1, in order to make it easier to use pragmas and analyze the synthesis results. For example, in the case of the first convolutional loop, the code ends up looking like this:



```
float DataOutC1[C1_out_CH][C1_out_x][C1_out_y];
Conv1_CH_out_loop:
for (int out=0; out < C1_out_CH; out++){
    Conv1_m_loop:
    for (int m=0; m < dtr_in_x-kernel_size+1; m++){
        Conv1_n_loop:
        for (int n=0; n < dtr_in_y-kernel_size+1; n++){
            DataOutC1[out][m][n] = C1B[out];
            Conv1_i_loop:
            for (int i = 0; i < kernel_size; i++){
                Conv1_j_loop:
                for (int j = 0; j < kernel_size; j++){
                    Conv1_CH_in_loop:
                    for (int in = 0; in < data_in_CH; in++){
                        DataOutC1[out][m][n] = DataOutC1[out][m][n] + DataInTr[in][m+i][n+j]*C1W[out][in][i][j];
                    }
                }
            }
        }
    }
}
```

Given that the number of instructions analyzed by the ANN is 256, we set this number of clock cycles as a goal, so that we can analyze loops as fast as they are executed by the CPU.

5.1.1 Unoptimized Solution

A first synthetizable solution with no directives or any kind of optimizations (simply the C++ code made synthetizable) achieves a latency of 4086734, a far cry from our original objective. The FPGA chip selected for the project was the Xilinx Zynq UltraScale+ MPSoC ZU3EG A484, given that it is the chip found in the Ultra96 board, which is reasonably priced and available for use in the university's electronics department. The clock frequency selected was 100MHz, which is the frequency in which the LT16x32 processor was originally being used.

The hardware utilization estimates are summarized in the following table:

	BRAM18K	**DSP48E**	**FF**	**LUT**
Total	112	5	2512	5340
Utilization(%)	25	1	1	7

Table 5.1: First solution hardware utilization

Looking more closely at the scheduling report, we can see that the first convolutional loop takes 1759804 clock cycles, while the second and third take 839528 and 331592 clock cycles respectively. These numbers are roughly 11 times higher than our first estimates for the number of operations that should be performed by these loops, so it might be a good idea to take a closer look at the scheduling of operations in the innermost loop:

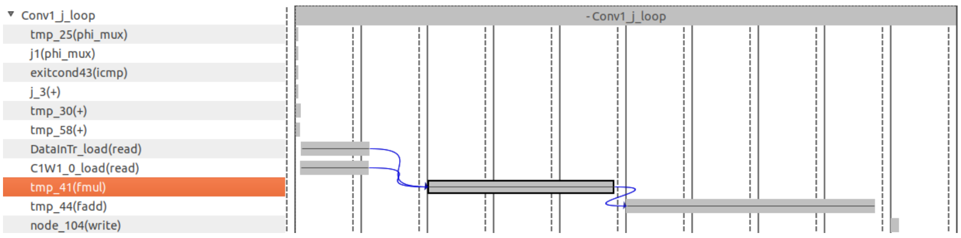


Figure 5.1: Scheduling of the first convolution innermost loop

As it can be appreciated in this figure, each multiply-accumulate operation in the loop takes ten clock cycles: two for reading the parameters, three for the floating point multiplication and strangely enough, four for the floating point addition. Therefore, given that each operation takes ten clock cycles, plus the initialization of the output with the bias and some additional overhead, we can see that these numbers are in agreement with our estimates. The remaining clock cycles correspond to the max pooling and ReLU operations, as well as the fully connected layer, etc.

5.1.2 Optimizing Data-types

Many conclusions and possible action plans can be drawn from the previous initial analysis: the first and most obvious is to change the data representation from floating point to fixed point, bit-arbitrary types. As it has been recently shown by Courbariaux et al in [7], the resolution of the data in ANNs doesn't have to be high at all, in fact, they show that with *Binarized Neural Networks*, cutting-edge accuracies can be achieved. Since I do not possess enough knowledge on the subject (nor time to further investigate it), a simpler approach consists of simply using Vivado HLS's arbitrary types instead of floating point and test the impact of bit accuracy on the CNN's prediction abilities. Given that the trained CNN's weights and biases are bounded in absolute value by a minimum of 0.000114 and a maximum of 1.1689, it seems reasonable to use at least two bits for the integer part, and the rest for the fractional part. Table 5.2 and figure 5.2 summarize the findings when testing different bit precision on a test set of 1000 elements.

float	**fixed16**	**fixed8**	**fixed7**	**fixed6**	**fixed5**	**fixed4**
99.8%	99.8%	99.9%	99.8%	99.6%	91.0%	53.3%

Table 5.2: Number representation accuracy

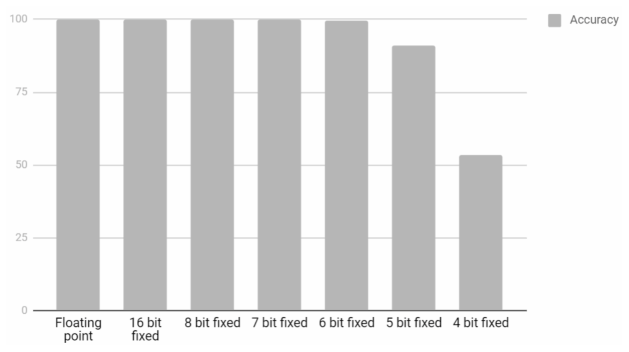


Figure 5.2: Number representation accuracy

Given that the accuracy loss of going from floating point to 6-bit fixed point is a mere 0.2%, this was the chosen resolution. However, further optimizations can be made if we consider that our inputs consists of a 2D array of zeros and ones, so for the first convolutional layer we can use 1-bit fixed point numbers.

Another conclusion that can be drawn from the previous analysis is that instead of using RAM banks to store the data and weights, we can use registers instead. This will undoubtedly become a necessity if we choose to increase the parallelism level later: it's worth noting that by default, Vivado HLS keeps the loops rolled and does not perform pipelining, nor does it partition arrays (unless the loops or arrays are very small).

After defining the fixed point data types as follows:

```
typedef ap_ufixed<1, 1, AP_RND, AP_SAT> ap1;


typedef ap_fixed<6, 2, AP_RND, AP_SAT> ap6;
```

and changing all data types in all functions and arrays (and changing the floating point function `fmaxf` to the conditional operator `"?"`), the latency of the innermost convolution loop decreases to 4, which cuts the forward function latency down to 2044558 clock cycles. This is practically half the time required initially, and even though it's far away from our goal of 256, it also decreased substantially the hardware usage, as can be seen in the following table:

	BRAM18K	DSP48E	FF	LUT
Total	28	3	1057	4141
Utilization(%)	6	˜0	˜0	5

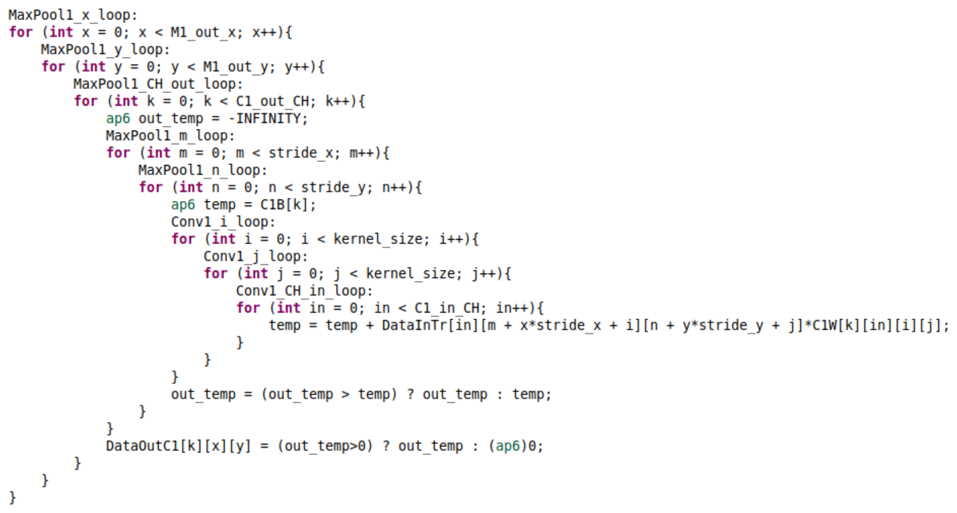
Table 5.3: Second solution hardware utilization

As we previously mentioned, it is possible to get rid of the 153000 multiplications in the first convolutional layer, by noting that one of the two operands is always one or zero, which degenerates to a simple `AND` function. However, by inspecting the scheduled operations one can see that Vivado HLS is smart enough to detect this (most likely because one of the two inputs of the multiplication is 1-bit data) and automatically exchanges the multipliers with `AND` gates.

5.1.3 Optimizing the Data-Flow

Vivado HSL creates memory blocks for every array that we define, and if we take a look at the data flow of the forward function in figure 4.3, we can see that each of the convolution, ReLU and MaxPool operations take an input and output array as parameters. In this way, the convolution processes the input array and writes the corresponding value onto another array, which is taken as input by the ReLU process and outputs an array to the MaxPool operation, for all layers, all of which can be completely avoided (thus accelerating the whole process) by merging these loops into one. While Vivado HLS has a built-in pragma that can potentially merge all these loops, it is unfortunately unable to do this automatically (or at least, despite many efforts, I was unable to find a way) since it detects data dependencies which it cannot solve. So the next step in order to avoid the unnecessary memory arrays and clock cycles associated with these loops, is to manually merge them. While it should be possible to merge *all* loops in the forward function, the code becomes quite complex and unintelligible very fast, and therefore we merged the three loops of each convolutional layer into one, that is, the `Conv2d`, `ReLU3d` and `MaxPool3d` functions will be computed in a single loop. Similarly for the FC layer's functions, we merged the `view` function, `FullyConnected` and `ReLU1d` into a single loop.

With these three loops merged in a single one, the code for the first layer consisting of a convolution+ReLU+MaxPool operations ends up looking like this:



```
MaxPool1_x_loop:
for (int x = 0; x < M1_out_x; x++){
    MaxPool1_y_loop:
    for (int y = 0; y < M1_out_y; y++){
        MaxPool1_CH_out_loop:
        for (int k = 0; k < C1_out_CH; k++){
            ap6 out_temp = -INFINITY;
            MaxPool1_m_loop:
            for (int m = 0; m < stride_x; m++){
                MaxPool1_n_loop:
                for (int n = 0; n < stride_y; n++){
                    ap6 temp = C1B[k];
                    Conv1_i_loop:
                    for (int i = 0; i < kernel_size; i++){
                        Conv1_j_loop:
                        for (int j = 0; j < kernel_size; j++){
                            Conv1_CH_in_loop:
                            for (int in = 0; in < C1_in_CH; in++){
                                temp = temp + DataInTr[in][m + x*stride_x + i][n + y*stride_y + j]*C1W[k][in][i][j];
                            }
                        }
                    }
                    out_temp = (out_temp > temp) ? out_temp : temp;
                }
            }
            DataOutC1[k][x][y] = (out_temp>0) ? out_temp : (ap6)0;
        }
    }
}
```

Figure 5.3: Merged convolution+ReLU+MaxPool loop C++ code

As can be seen, the code is quite convoluted (no pun intended) as it now consists of eight nested `for` loops. Further manual merging of the remaining loops could theoretically decrease the hardware usage and processing time, but the code would also be extremely difficult to read and debug, thus losing the appeal of HLS. This became our basic building block, which contains all three operations found in a single convolutional layer, and would latter be used as a starting point for our VHDL implementation. As it is, the execution time was cut in less than half, down to 893704 clock cycles, which is still far away from our 256 clock cycles goal.

The hardware utilization was also cut roughly in half, as can be seen in the following table:

	BRAM18K	DSP48E	FF	LUT
Total	6	4	543	2471
Utilization(%)	1	1	˜0	3

Table 5.4: Third solution hardware utilization

5.1.4 Pragma-based Parallelization

After all these optimizations were done, we started to parallelize the design, using the pragma: `#pragma HLS PIPELINE`. This pragma not only pipelines the design, but also unrolls the inner loops inside the level that the pragma has been placed. The outermost possible level of pipelining is in the loop `MaxPool_m_loop`; trying to pipeline even further results in failure because according to Vivado HLS, ”it may cause large runtime and excessive memory usage due to increase in code size.” At this point, the clock cycles required decreased substantially to 54823, that is, roughly 16 times faster. However, this comes at the expense of a sharp increase in hardware utilization, as can be seen in the following table:

	BRAM18K	DSP48E	FF	LUT
Total	10	152	7230	39402
Utilization(%)	2	42	5	55

Table 5.5: Fourth solution hardware utilization

In order to continue increasing the level of parallelism and not run into the problem previously mentioned, some loops had to be manually unrolled. Starting with the innermost loop (`Conv_CH_in_loop`), we can now use the pipeline pragma at the level of `MaxPool_y_loop`. If we wanted to achieve even more parallelism, we could keep unrolling the innermost loops, but further pipelining the functions at lower levels incurs in excessive hardware utilization. In the following table and figures, ’L2’ and ’L3’ means unrolling the convolutional layers at the second and third loop nests respectively, that is, at `MaxPool_y_loop` and `MaxPool_CH_out_loop`. For example, L2L3L3 means pipelining the first convolutional layer at the second level, and the second and third layers at the third level.

If we pipeline at L2L2L2, the utilization of LUT units climbs up to 141%, exceeding the available units in the FPGA. Moreover, the difference in clock cycles between L2L2L3

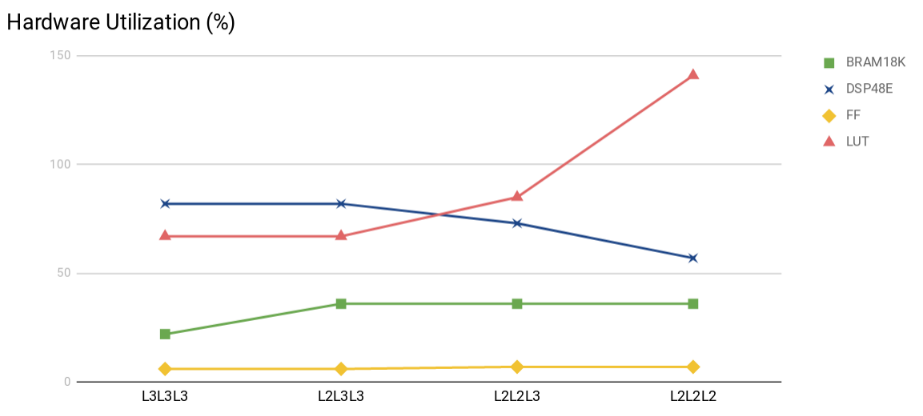


Figure 5.4: Percentage of Hardware Utilization for different levels of parallelism

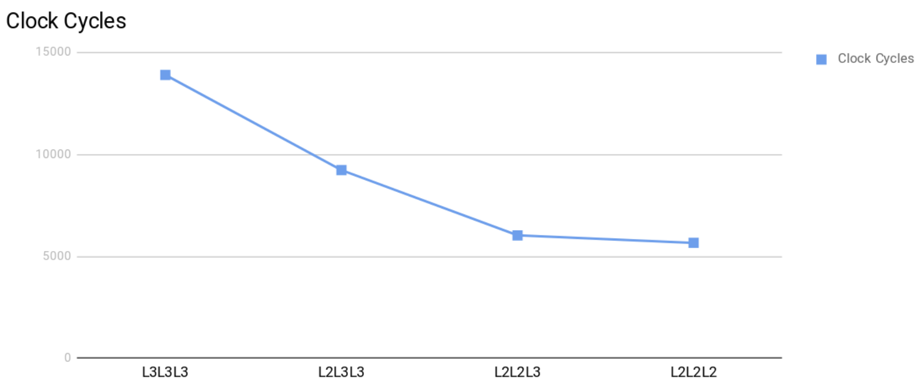


Figure 5.5: Clock cycles for different levels of parallelism

Level	**BRAM18K**	**DSP48E**	**FF**	**LUT**	**TCk**
L3L3L3	22%	82%	6%	67%	13895
L2L3L3	36%	82%	6%	67%	9233
L2L2L3	36%	73%	7%	85%	6035
L2L2L2	36%	57%	7%	141%	5668

Table 5.6: Hardware utilization of the different parallelization levels

and L2L2L2 is quite small, so the obvious choice is to pipeline at levels L2L2L3, which achieves a latency of 6035 clock cycles, roughly 23.5 times higher than our goal of 256, while using a similar number of hardware units as slower alternatives.

5.1.5 Increasing throughput with Array Partitioning

Besides loop pipelining, another directive that was used in order to achieve the throughput previously shown is the `HLS ARRAY_PARTITION` pragma. This is suggested by the tool itself, since during the synthesis process, we get warnings like the following:

```
WARNING: [SCHED 204-69] Unable to schedule 'load' operation
on array 'DataInTr_V' due to limited memory ports.
Please consider using a memory core with more ports or
partitioning the array 'DataInTr_V'.
```

The `HLS ARRAY_PARTITION` pragma breaks down an array into smaller arrays or even individual elements, which results in RTL with multiple small memories or multiple registers instead of one large memory. This effectively increases the amount of read and write ports for the storage units, and potentially improves the throughput of the design. However, it requires more memory instances or registers and more data buses. Multidimensional arrays can be partitioned multiple times in the different dimensions, and partitioning can be partial or complete. Completely partitioning an array of N elements creates N memory banks, while block and cycle partitioning with a factor M creates M memory banks, each with N/M addresses. Block and cyclic partitioning differ in the order that the array elements are placed in each bank, but perhaps it is easier to visualize the different kinds of partitioning methods available and their effects on the original array in figure 5.6:

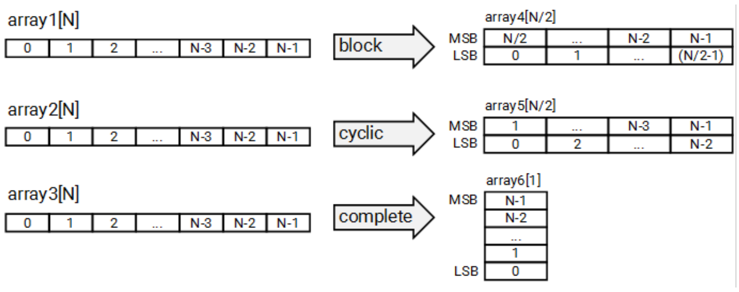


Figure 5.6: Array Partitioning

While it was possible to achieve a latency slightly lower than the initial target of 256 clock cycles, the hardware utilization climbed up even further, with a LUT utilization reaching over 500% of the available units. Also, it must be noted that a substantial amount of time was spent trying many other unsuccessful approaches, such as exploring loop merge pragmas, HLS dataflow pragmas (which is in theory, similar to pipelining but between functions instead of operations); analyzing memory access patterns to cache array rows to a local array (also referred to as 'line buffering' in the context of image filtering), so that the input arrays are read only once and can be implemented as FIFO buffers, allowing for the functions to start processing the data even if not all inputs are available; etc., but all these techniques did not provide any advantages, or in some cases, performed much worse, and in others they didn't work at all. Finally, during the system integration phase (which will be discussed later on this chapter), a new optimization was found, which (while still being an L2L2L3 pipelining) increased the execution speed by 77% (3406Tck vs 6035Tck), and at the same time decreased the BRAM18K utilization from 36% down to 22% and the LUT utilization from 85% down to 80%. This is roughly 1200 times faster than the unoptimized code, and 13 times slower than our original goal.

5.2 VHDL implementation

After finishing our work with the HLS version of the CNN, we had a C++ code which we could, with some effort, adapt to VHDL. This code can be more or less ported directly to VHDL, but three major challenges were faced when doing this.

5.2.1 VHDL challenges

Synthetizable Fixed-point Libraries

First, unlike in Vivado HLS, the fixed point library is not available by default in Vivado anymore and must be manually compiled and installed. Furthermore, and much to my surprise, the IEEE fixed point library has not yet been standardized, and while there is a proposed version of it, there are different versions which may be synthesized by some tools, but not by others. Multiplications of two `sfixed`, M-integer-bit and N-fractional-bit numbers, produce a 2M-integer and 2N-fractional bit `sfixed` number. Similarly, an addition of two M-bit `s_fixed` numbers produce a result with size M+1 bits. In order to avoid increasing the bit size of the data after each operation, the `resize` function must be used. This function can be called with the following parameters: `overflow_style` can be either `fixed_saturate` or `fixed_wrap`; and `round_style` can be either `fixed_round` or `fixed_truncate` (I assume the reader has knowledge on the differences between these methods). Similarly, the same methods are also available in Vivado HLS when defining an `ap_fixed` type, along with other additional rounding and saturation styles. However, no matter the combination of methods chosen for both parameters, the result of a resize operation (or the mere conversion of a 'real' number to a fixed point representation) yields different results when converting to `ap_fixed` and `sfixed` in Vivado HLS and Vivado respectively. I have so far not been able to determine the cause behind this, but in any case, the differences become minimal as the resolution is increased.

Multidimensional Arrays

The second major challenge faced in the conversion to VHDL was dealing with multidimensional arrays and data endianness. VHDL does not support array slicing for arrays with more than one dimension. Additionally, there are two ways of defining multidimensional arrays: the first is to simply define a new data type with all the dimensions, similar to what one would do in C++, e.g.:

```
type data2D_t is array (0 to rows -1, 0 to columns -1)
of STD_LOGIC;
```

However, this definition style does not allow accessing a whole row in a single statement, since as we mentioned, array slicing is only permitted in one-dimensional arrays. A work-around to this problem is to use a `for` loop to copy every single element of the row, which results in a quite cumbersome code. Another way of defining the same array would be to first define a row subtype:

```
subtype data2D_row is STD_LOGIC_VECTOR(0 to columns -1);
```

and then define the array type as:

```
type data2D_t is array (0 to rows -1) of data2D_row;
```

While this does allow for array slicing, it is only possible for the last dimension. To add to the confusion, calling an array element is done differently depending on the definition used. Combine this with the fact that you can mix both definition types, different endiannesses are possible (using `X-1 downto 0` instead of `0 to X-1`); throw four- and five-dimensional arrays to the mix, and you can have quite the headache! Needless to say, special attention and care must be taken when dealing with VHDL arrays.

Loop Unrolling

The third challenge relates to loop unrolling (or loop rolling, to be more accurate). In VHDL, `for` loops are executed in one clock cycle, so in order to fit the ANN in the hardware available, it is necessary to execute these loops in multiple clock cycles. For example, the following loop executes in a single cycle:

```
for i in 0 to M-1 loop
    temp(i) := 0;
    for j in 0 to N-1 loop
        temp(i) := temp(i) + x(i, j)*y(i, j);
    end loop;
end loop;
```

If we wanted the IC to execute a single multiply-accumulate statement in each clock cycle, these loops must be manually rolled, taking special care not to overwrite variables which should only be initialized in the first iteration of the loop. With these things in mind, our previous example would become:

```
if i < M then
    if (i = 0) and (j = 0) then
        temp(i) := 0;
    end if;
    if j < N then
        temp(i) := temp(i) + x(i, j)*y(i, j);
        j := j+1;
    end if;
    if j = N then
        j := 0;
        i := i+1;
    end if;
    if i = M then
        i := 0;
        loop_ended := '1';
    end if;
end if;
```

As anyone can imagine, in the case of our code (which consists of eight nested non-perfect `for` loops for each layer), the code can become quite cumbersome and difficult to control: a loop that takes 20 lines in C++ now takes around 80 lines of VHDL code, a four-fold size increase, which carries with it a similar increase in the risk programming of errors. Nonetheless, having precise control of what happens in each clock cycle is a very welcome difference with Vivado HLS. We can take advantage of the fact that each convolutional layer duplicates the number of output channels, while the MaxPooling layer divides the number of outputs by eight, which amounts to a net activation reduction factor of four. This allows us to run each successive layer four times slower than the previous one (thus saving quite a lot of transistors) at almost no cost in the overall CNN speed. If we look at the original code present in figure 5.3, we can calculate how many clock cycles it would take for a layer to complete its processing, if we decide to unroll it at a certain level: in table 5.7, a cell with a white background means the loop associated with that variable is left rolled, while a cell with a gray background means that the associated loop (and the loops lower in the hierarchy) have been unrolled. Therefore, the number of clock cycles that it takes each layer to complete is the product of the values in the white cells.

The following table summarizes the relevant values for the loops higher in the hierarchy:

Layer	x_out	y_out	ch_out	stride_x	stride_y	**#Cycles**
Layer 1	63	37	2	2	4	2331
Layer 2	15	18	4	2	4	2331
Layer 3	3	8	8	2	4	2331

Table 5.7: Loop limits and clock cycles

By unrolling the loops in each layer at the levels mentioned, it's possible to schedule the operations for each one so that it takes them exactly the same time to compute. As we mentioned, only the second and third layers require multipliers; the second one requiring:

```
stride_y * kernel_size^2 * C2_in_CH = 32
```

and the third one only:

```
kernel_size^2 * C3_in_CH = 16
```

This is an insignificant amount compared to the 360 DSP48 multiply-accumulate units present in the FPGA, which in addition have support for a SIMD mode. In theory, each DSP48 could support up to four 6-bit multiplications, for a total of 1440 6-bit multiply-accumulate units, so this is a possible source of optimizations which could lead to a significant speed-up. If we distributed all operations into 256 clock cycles, the first layer would require 583 multiply-accumulate units, while the second and third layer would require 270 and 96 respectively, for a total of 949 (these numbers come from multiplying all the loop limits and dividing by 256). Two additional units would be required for the FC layer, so it's theoretically possible to perform the analysis synchronously with the processor without exceeding the FPGA's capabilities. Unfortunately, there was not enough time to optimize and investigate this any further, so with the current code, the synthesis tool does not infer any of these DSP48 modules, therefore implementing multipliers through LUTs.

5.2.2 Loop Pipelining

Given the precise, clock cycle accurate control that VHDL provides, it was possible to easily perform ’function pipelining’ (similar to the ’dataflow pragma’ in Vivado HLS) between the different layers. By analyzing the memory access patterns and slightly changing the order of the loop hierarchy, it can be shown that each layer requires five rows of data (one of them being re-utilized from the previous iteration) to start processing once a new output row is being calculated. So, for example, the second layer can start working on the data after the following condition becomes true: `y1 > 4*(y2 + 1);` while the third layer can start after an analogous condition. The same can be done with the first layer, but here we must be a little bit more careful, since we must cache lines of data read from an external memory with a latency of one clock cycle. Once the cached data has been processed, the fifth row is moved to the first position. The following code takes care of updating the cache every time the `y` variable is increased, which indicates that four new rows are required:

```
-- Update cache with 5 lines of data:
if (update_ena = '1') then
    for x in 0 to input_x- 1 loop
        cache_lines(x, cache_ptr_old - (y*stride_y)) :=
            data_in(x);
    end loop;
    update_ena := '0';
end if;
if (cache_pointer < 1 + (y+1)*stride_y) then
    cache_ptr_old := cache_pointer;
    addr_in <= std_logic_vector(to_unsigned(cache_pointer,
                                    addr_in'length));
    update_ena := '1';
    cache_pointer := cache_pointer+1;
-- End Update cache with 5 lines of data
else (...)
```

This requires only one memory block with a single channel, a big contrast with the dual channel memory block and three single channel memory blocks required by the HLS version of the CNN.

5.2.3 Controlling the CNN with a simple FSM

Finally, a very simple Finite State Machine (FSM) was created to control the whole network, which can be seen in the following code:

```
control_FSM: process(rst, clk)
begin
    if (rst = '1') then
        state <= wait_ena;
        ready <= '0';
    elsif rising_edge(clk) then
        case state is
            when wait_ena =>
                ready <= '1';
                if enable = '0' then
                    state <= wait_ena;
                else
                    state <= process_CNN;
                end if;
            when process_CNN =>
                ready <= '0';
                if FC_done = '0' then
                    state <= process_CNN;
                else
                    state <= wait_ena;
                end if;
            when others =>
                state <= wait_ena;
                ready <= '0';
        end case;
    end if;
end process control_FSM;
```

While the `reset` signal is being asserted, the `ready` output is set to zero and the `state` is set to `wait_ena` (wait for an enable input signal). When in the initial state `wait_ena`, the `ready` output is set to one, to signal to the system that the ANN is ready to process new data. Once the module is requested to start processing (by setting its `enable` input to one) the state is set to `process_CNN`, in which the output ready is set to 0, and we wait for the FC layer to finish processing the data. Once the ANN is done processing, it goes back to the `wait_ena` state, ready to start processing a new assembly loop. The control of the dataflow between the different layers is done implicitly by the conditions imposed to the row variables, as previously mentioned.

This design needs 3215 clock cycles to analyze one loop, roughly the same time as the HLS version (2.3 times faster than it would take without function pipelining); however hardware utilization is significantly lower, with LUT utilization at 34.3%, flip-flops at 18.6%, 0% of DSP48 and 0% of BRAM18K blocks.

5.3 Loop detection and the CNN Input Pre-processor

After we had a functioning CNN, the next step was integrating it with the rest of the hardware; that is, the CPU and memory. In order to do this, a hardware block that snoops on the instruction memory bus, detects a loop and vectorizes its instructions (as described in section 3.1.3) was designed. As a first approach, we tried to detect the head and tail of the loop with the aid of another neural network. However, all attempts were unsuccessful, neither the LSTM nor the CNN provided good results. The input data of these neural networks were simply the sequences of addresses generated by the loops. While the CNN performed significantly better than the LSTM, the addresses predicted were in some cases correct, but in some other cases the address was off by one or a few positions.

5.3.1 Detecting Loops with a FSM

Given that loops are always defined by a jump back to a previous address, an FSM capable of detecting this is quite simple to develop. A very interesting work related to this was performed by de Alba and Kaeli in [1], in which their loop prediction hardware consists of four basic mechanisms; namely, the Path-to-Loop shift register (similar to a Branch Target Buffer pattern register), a Path-in-Iteration Table (which records state transitions through the sequence of unique path-in-loop patterns generated in a loop visit), a Loop Prediction Table (which stores the number of predicted iterations, etc.), and a Loop Cache (where the loop body is written). Our version is nowhere near as sophisticated, but again, this is a very simplified version of the problem. We mentioned already that we haven't considered nested loops, and so the assumption we work under is that there is a main program which starts at address 0, and it contains a few instructions after which the first loop starts. When this loop is done, some other instructions are executed and a second loop begins its execution. After the last loop is executed, the main program executes a few instructions and jumps back to an address near to the beginning.

This behaviour is depicted in figure 5.7:

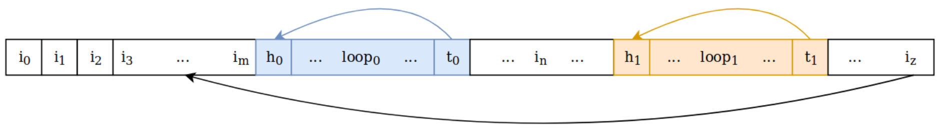


Figure 5.7: Assumed program behaviour with two loops

Under this hypothesis, our instruction pre-processor contains a stack which keeps track of the various loops present in the main program, along their head and tail addresses, number of loop iterations, etc. The behaviour of the FSM can be depicted in the following diagram:

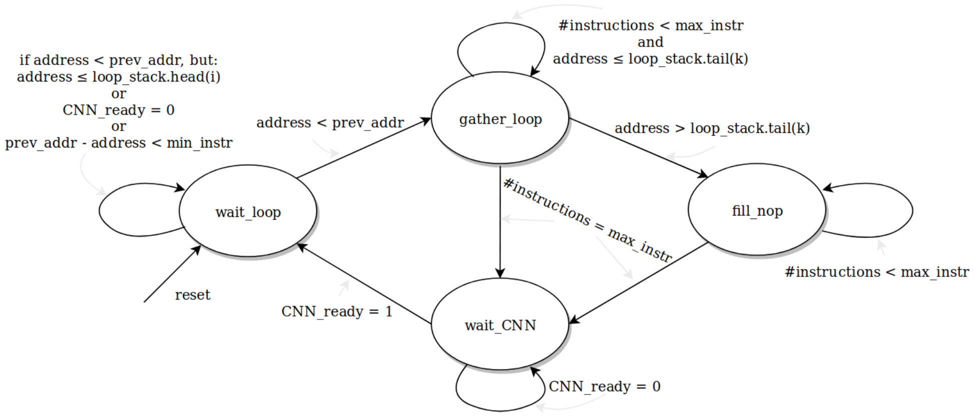

if address < prev_addr, but:
address ≤ loop_stack.head(i)
or
CNN_ready = 0
or
prev_addr - address < min_instr
wait_loop
reset
address < prev_addr
gather_loop
#instructions < max_instr
and
address ≤ loop_stack.tail(k)
address > loop_stack.tail(k)
fill_nop
#instructions < max_instr
#instructions = max_instr
wait_CNN
CNN_ready = 1
CNN_ready = 0

Figure 5.8: Pre-processor Finite State Machine

After a reset, the FSM goes to the state `wait_loop` and waits for a jump back to a previous address. If a jump back is detected, and the loop is not too small (fewer than a certain minimum threshold number of instructions), and the jump is not of a hierarchy higher than the previously detected loops, that is, it's not the jump back of the main program (remember, the only loops we consider are nested inside a main program), and the CNN is not busy processing a previous loop, then we switch the FSM to the `gather_loop` state. When the loop has been detected, we push the head and tail addresses onto the stack, and we gather instructions until one of two conditions is fulfilled. It can happen that either the loop executes less than the 256 instructions needed by the CNN for processing, or it can execute more than 255. In both cases, instructions are gathered, vectorized by purely combinatorial logic, and stored into a memory array. In case that the instructions executed are less than 256, we fill the remaining of the array with vectorized nops (this is done in the state `fill_nop`). After the array is full, we switch to the `wait_CNN` state, in which we tell the CNN to start processing, and we remain in this state as long as the CNN is busy with its process.

Once the CNN has finished working and made its prediction on whether the loop is parallelizable or not, this result is pushed onto the stack. The information in the stack can now theoretically be used by the serial-to-parallel code converter. The FSM goes back to its initial state, and starts analyzing loops that are still not present in the stack.

5.3.2 Implementing the FSM in Hardware

Both a synthetizable C++ version and a VHDL version were implemented, having an almost one-to-one correspondence in the source code. However, for some reason unknown the C++ HLS version does not work correctly: as it can be seen in figure 5.9, the vectorized instruction output of the HLS version (named `vector_instr_array_V_d0_0`) contains a value of X in some of its bits, which does not happen in the VHDL version (whose output is named `vector_instr_0`, and is delayed one clock cycle respect to the HLS signal). This error does not appear when running the C simulation in Vivado HLS, where the outputs are correct and the same as the VHDL version; it only appears when it's exported as an IP block and simulated in Vivado. The position of the bit (or bits) that change their value to X is not fixed, and debugging the VHDL code generated by the HLS synthesis tool is impossible. What is even more strange, is the fact that this vectorial output is purely combinatorial, and when this combinatorial function is synthesized alone from C++ it works perfectly in the RTL simulator, but when called from the FSM (which calls *exactly the same function*), the outputs do not behave as expected.

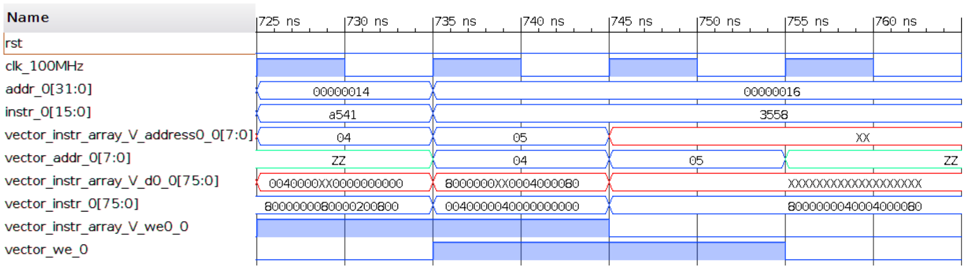


Figure 5.9: FSM simulation waveform: VHDL vs C

As mentioned previously, the VHDL version of this IP block is a one-to-one translation from the C++ code, and given that the VHDL version works as it should, along with the fact that the C simulation does not show this behaviour (making it impossible to debug), the C++ instruction pre-processor was discarded and the error was regarded as a bug in the HLS tool, so no additional time was allocated to finding the root cause of the problem. Finally, it should be noted that both versions consume very similar amounts of hardware.

5.4 System integration

Once we had designed and tested all the building blocks required for the final system, it was time to put all the pieces of the puzzle together. We'll start by describing the VHDL system, as it is simpler and easier to understand than the HLS system.

5.4.1 VHDL System

Using Vivado's IP integrator, it is possible to create a system by simply dropping IP cores in the Block Diagram GUI. Connections between different modules can be made by simply dragging a wire between I/O ports; the final result can be seen in figure 5.11:

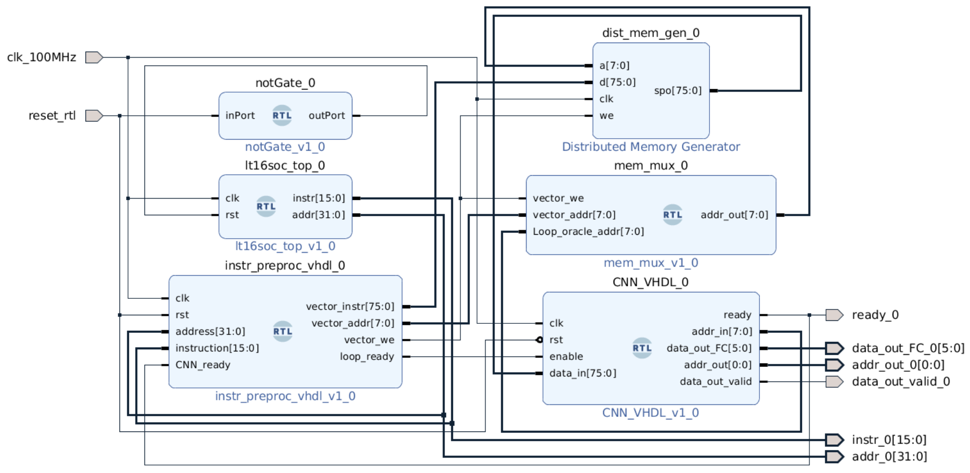


Figure 5.10: VHDL System

The system consists of six IP blocks: the first box in the figure is simply a `NOT` gate providing the required reset signal polarity of the LT16 SoC, which outputs the current instruction and its address; the instruction pre-processor and CNN blocks previously explained; a single channel distributed memory generator and a memory controller. The memory simply stores the vectorized instructions array and it has therefore 256 addresses, and a word width of 76 bits. The memory controller simply chooses which address should be the input of the distributed memory block: when the instruction pre-processor is asserting its write enable signal, then the input address is the one given by then instruction pre-processor; if not, the address is taken from the CNN block, so that it can read the array when it needs to. The LT16 SoC has its own memory inside the block, this system uses a wrapper which hides its complexity and outputs only the required signals. The SoC is running a program randomly generated by LoopGen. Rather than testing all different programs from the dataset (which would be extremely complex to implement and slow to simulate), the testing was done by using a single program, but instead of converting the two FC layer's outputs to two one-bit outputs, [0, 1] or [1, 0], we can simply compare the original 6-bit values with the software CNN's 6-bit outputs. It will soon become useful to remember that the final output is [1, 0] if `CnnOut[0] > CnnOut[1]`, and [0, 1] in the opposite case.

5.4.2 HLS System

The system created with the HLS version of the CNN is very similar, but instead of requiring a unique single-channel memory block, it requires one dual-channel memory block, and an additional three single-channel memory blocks.

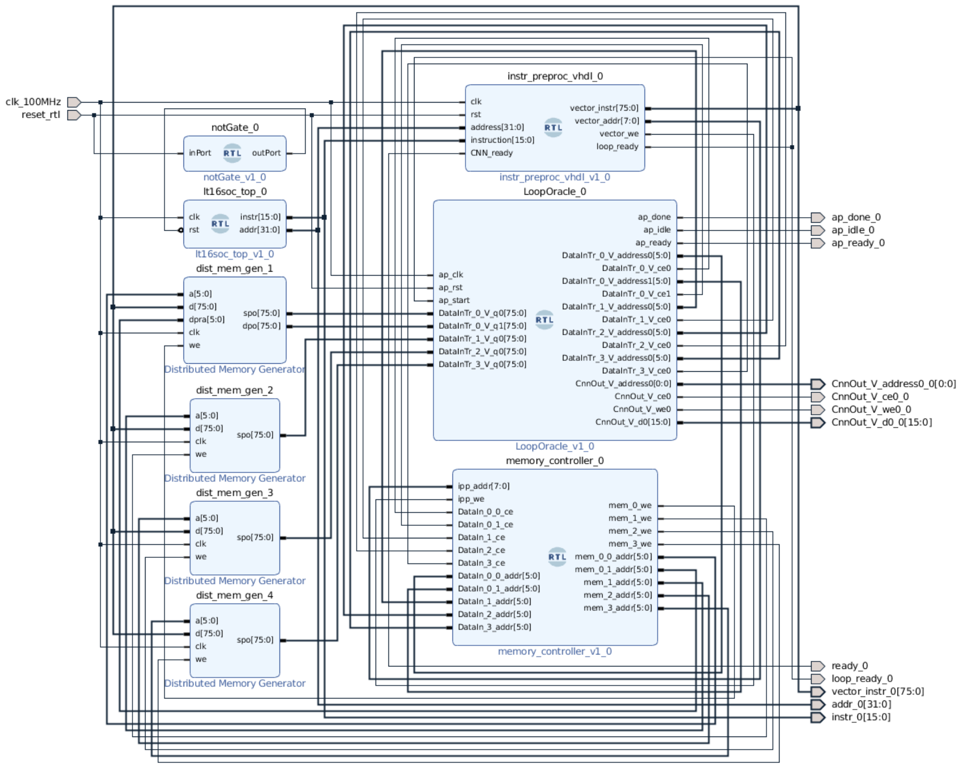


Figure 5.11: HLS System

Each memory block has a depth of 64 instead of 256, since the array is partitioned in four. Initially, the input data was a two-dimensional array of 76x256 1-bit `ap_ufixed` values, but this required 76 input channels, resulting in a need for 76 memory blocks, each with its own chip enable, write enable, and 8-bit address input. The obvious solution to this problem was to create a 76 bit wide, one-dimensional array with depth 256, which when correctly partitioned, not only does it consume less resources, but as mentioned in section 5.1, it led to a significant speed-up of 77%. Given that the array partitioning is cyclical, when the instruction pre-processor writes to an address whose least significant bits are 00, the controller sends this data to the first memory block; when the address ends in 01, it goes to the second, and to the third and fourth blocks when 10 and 11 respectively. The input address of these memory blocks is given by the instruction pre-processor when its write enable signal is asserted, and by the CNN when it asserts its chip enable signals. This results in quite a lot more wiring compared to the VHDL version. At this point it must be noted that yet another bug had been discovered: when Vivado HLS generates the IP package for the CNN, so that it can be integrated with other IP in Vivado, the signal `DataInTr_3_V_ce0` appears as a `STD_LOGIC_VECTOR` of a single bit in the automatically generated VHDL wrapper, but inside the generated VHDL code itself, the output was defined as a `STD_LOGIC` signal, making it impossible to integrate with the memory controller: if we defined the input signal `DataIn_3_ce` as `STD_LOGIC_VECTOR`, it would complain that it was incompatible with the type `STD_LOGIC`, and the opposite was true when `DataIn_3_ce` was defined as a `STD_LOGIC_VECTOR`. The only solution was then to directly import the VHDL code generated by the tool into the project, rather than adding the IP block to the library as suggested by Xilinx. Alas, our problems with the HLS version of the CNN did not end here, as it did not produce the same outputs as the C++ simulation. The VHDL version too was generating different results (although this was to be expected, given the previously mentioned discrepancies between `sfixed` and `ap_fixed`), and therefore, suspecting that this might have been a problem with the data precision, we increased the number of bits to 16. The following table shows a comparison between the outputs produced by each CNN for different bit precisions:

Precision	Output	C++ in Vivado HLS	C++ in Vivado	VHDL in Vivado
Floating Point	CnnOut[0]	0.3944	-	-
	CnnOut[1]	0.4316	-	-
16-bit Fixed	CnnOut[0]	0.3948	0.5273	0.3936
	CnnOut[1]	0.4333	0.2891	0.4309
8-bit Fixed	CnnOut[0]	0.3750	-	0.3594
	CnnOut[1]	0.5000	-	0.4688
6-bit Fixed	CnnOut[0]	0.1875	0.3750	0.3750
	CnnOut[1]	0.4375	0.2500	0.1250

Table 5.8: Bit-precision and tool comparison

In the case of the HLS CNN, the results remained completely different to the floating point results, while the VHDL 16-bit version converged to the floating point values, with a difference of around 0.2%. When the VHDL version is scaled down in precision to 8 bits, the difference with the floating point values remained within 10%, but more importantly, the ordinal relationship between both outputs was preserved. However, when scaling down to 6 bits, the difference becomes too great and the ordinal relationship between the outputs is inverted, which results in an incorrect prediction.

Regarding the C++ version, it *did* produce the same outputs as the HLS C simulation when the CNN was fed with the same vectorized instruction repeated 256 times, so this may suggest that there is a problem either with the memory read timings, or with the partitioning of the array. Unfortunately, there was not enough time to further investigate this problem, but at least the VHDL version is working as intended when integrated in the system.

Chapter 6

Discussion

6.1 PyTorch Neural Network

While the CNN achieved an astonishing test accuracy of 99.8%, it is unclear whether this success can be carried over to a more realistic scenario. It is also unclear whether the CNN is truly detecting parallelism or if it's simply being able to detect unique characteristics of each type of loop and memorizing them. Working with an in-house developed CPU inevitably led to having to create our own examples, and while theoretically the parametric randomly generated loops are all different from each other, in the end, they're nothing more than eight different programs (allegedly a very low number to successfully train a neural network), which have been augmented to hundreds of different examples, and it's very well possible that this accuracy is nothing more than the result of over-fitting. This is supported by the fact that the validation loss, as shown in figure 4.2, does not increase after reaching a minimum. Even though the validation set was not used for the training phase, if the characteristics that the CNN is able to learn are identical in the examples of the train set and validation set (e.g., a specific sequence of instructions present in all loops of the same kind), then this is exactly what one would expect to find: a validation loss closely following the behaviour of the training loss. Moreover, the failure of the ANNs to detect a loop's head and tail (which is a very simple task) casts even more doubt to the approach and the validity of its results. On the other hand, this does not imply that the task is impossible or that it cannot be solved by a neural network; rather, that the current approach is possibly not the right one. Finally, whether the CNN is successful in a broader sense or not, it can be said that the design works extremely well within the restrictions and for the tasks that it was set to perform, and that CNNs are not only orders of magnitude faster to train than LSTMs, but at least in my limited experience, they perform much better.

6.2 C++ CNN

Implementing the forward method for a CNN in C++ was surprisingly easy, and in hindsight I believe it was the right choice to develop the CNN from the ground-up rather than trying to understand and use an existing library. This was not only (most likely) the fastest approach, but also gave me control and low-level knowledge of the inner workings of this type of network. There isn't much to highlight about the results, since it performed exactly the same as its high-level PyTorch counterpart. Working with arrays in C++ can be a bit tedious when coming from Python, but it's not a challenge that can't be overcome with some work.

6.3 Vivado HLS CNN

As it was mentioned in section 5.1.5, it was possible to achieve a latency slightly lower than the initial target of 256 clock cycles, but with a hardware utilization that far exceeded the hardware units available in the selected FPGA. Many attempts to achieve better parallelization of the algorithm were approached, such as exploring loop merge pragmas, HLS dataflow pragmas; analyzing memory access patterns to cache array rows to a local array, etc., but none were successful. While it's quite possible that this failure was due to my inexperience with HLS tools, I believe that if one must learn to code in a very different way in order to make these pragmas work correctly, then there is no point to the whole HLS approach. We will come back to this point later, but overall, it seems that the results achieved by the HLS tool are the outcome of a brute-force approach in trying to achieve the desired latency, and quite underwhelming in comparison to the expectations. Nevertheless, after optimizing the original C++ code, we obtained a considerable speed-up of 1200, only 13 times slower than our ideal goal.

6.4 VHDL CNN

One of the big differences found when trying to implement the same CNN algorithm in VHDL was the behaviour of fixed-point numbers. Even when converting a 'real' number to a fixed point representation (allegedly using exactly the same rounding and saturation methods), the results produced were not the same as the HLS arbitrary-precision fixed-point data types. However, it was found that the results converged to the floating-point values once the resolution was sufficiently increased. The VHDL version of the CNN attempted to achieve a latency similar to that of its HLS counterpart, and while it was harder to code and debug, it also provided a level of control not possible to achieve (at least, to the best of my knowledge) in HLS. Many of the unsuccessful attempts to achieve better parallelization in the HLS version were successfully implemented in the VHDL version (these ideas probably served as a guide as to what techniques could be attempted). The study of the memory accesses in HLS was key to implementing a *line cache* in VHDL, while the *function dataflow* was achieved with the usage of a simple conditional statement. Moreover, for this algorithm working at roughly the same speed, and even though the effort and time spent in optimizing the HLS version was far greater than the optimization efforts put into the VHDL version, the amount of hardware required by the latter was only a fraction of the hardware that the HLS version requires:

Utilization(%)	**BRAM18K**	**DSP48E**	**FF**	**LUT**	**#Cycles**
HLS	22	57	7	80	3406
VHDL	0	0	19	34	3215

Table 6.1: HLS vs VHDL hardware utilization

While it's still unclear whether 6-bit precision is also enough for the VHDL version of the CNN, at least it has been verified that for eight bits and more, the neural network works as expected and reproduces the results of the golden model.

6.5 Instruction pre-processor and System Integration

To integrate the CNN with the CPU in the same system, an instruction pre-processor and memory controllers had to be implemented. It was at this point that we started to observe the divergence of the simulations performed in Vivado HLS with the algorithms packed in IP blocks, imported and simulated in Vivado. First, for some unknown reason, the conversion of the 16-bit instructions to their 76-bit vectorized representations worked perfectly fine in Vivado HLS, resulting in identical values to those generated by LoopSim, but when the same block was simulated in Vivado, this transformation gave the wrong results, with some of the bits switching to X, as it was shown in section 5.3.2. What's more puzzling, is that the same vectorizing function worked just fine when implemented as a stand-alone block. The reader is encouraged to compare the synthetizable C++ code with the VHDL and see that there is practically a one-to-one correspondence between them. While the VHDL version of the instruction pre-processor worked flawlessly with the rest of the VHDL system, the same cannot be said when integrating the HLS CNN and the VHDL instruction pre-processor, since we found that the CNN was not producing the same results as it did in Vivado HLS. It is unclear why this is the case, but we suspect that it might be due to either incorrect memory access timings, incorrect order or partitioning of the input array, or yet another Vivado HLS bug.

6.6 HLS vs VHDL

It shouldn't be a surprise that coding the same algorithm in VHDL can be significantly harder, starting with the non-standardized fixed-point libraries, the difficulty of working with VHDL arrays, rolling and unrolling imperfect loops, and dealing with the debugging process, which is nowhere near as evolved as it can be in a C++ environment. However, while it's significantly harder to manually unroll, pipeline and merge loops in VHDL, there is also the advantage that one has perfect knowledge and control of exactly what does and doesn't happen in each single clock cycle, something that can't be said when working in HLS. Additionally, as we have shown, the hardware usage for the same algorithm can be significantly higher when coding in HLS, something that could be forgiven if it worked correctly and performed as expected. But this leads us to the main issue and perhaps the biggest drawback of HLS we found so far: what happens if the algorithm does *not* perform as it does in the HLS simulations? I don't think it would be far-fetched to assume that most of us would be instinctively tempted to inspect the RTL code generated by the tool. However, after a minute or so of inspection, it becomes painfully clear that the code generated is completely unintelligible: the tool creates several thousands of signals and constant values (over *twenty thousand* in the case of our CNN algorithm), which are impossible to follow or understand. And not only that, but if we were to attempt to debug this by modifying a line here and there in the C++ code, it must be noted that it can take several *hours* for a computer to create the RTL code for a given C++ program. Finally, another major problem we found, is that the HLS pragmas do not work as well as expected, and the help we get from the tool's warnings can be quite cryptic and not really useful. Perhaps, having access to more support, or working with an engineer experienced with the tool, could lead us to better results, but with things as they are today, the HLS experience so far has been quite underwhelming.

Chapter 7

Conclusion

To finalize this document, we will go through some recommendations on how to possibly continue and improve this project, and we will end with a brief discussion of the main conclusions that can be drawn from the results obtained so far. All the source code for every software and hardware module implemented in this project can be found at: https://github.com/LuisJalabert/Deep-Learning-Based-FPGA-CPU-Acceleration

7.1 Future Work

This project opens up a significant amount of questions, each of which could be enough for a whole project or master's thesis, and therefore, in order to advance in the aspects that I find interesting, some of these questions have been left somewhat unanswered. In particular, the neural network approach to the problem of loop parallelism detection requires much more work to be done with a C compatible processor (either a RISC V or an ARM CPU), in order to determine if it really works, or if the excellent accuracy obtained by the CNN is a simple mirage, an illusion caused by a poor definition of the problem. To improve in this area, employing a C compatible processor would allow us to use real benchmarks (such as SPEC ACCEL or similar) which would give us a much more ample and realistic set of programs to work with. In addition, this would allow to compare the results obtained by other methods presented in some of the work mentioned throughout this document.

Regarding the hardware implementations, I would discard the usage of HLS, in favour of a C++ model and directly translating the C++ code to VHDL. The CNN implemented in VHDL requires more testing to see if a 6-bit precision is enough to reproduce the re-

sults obtained by the golden model. A way to do this would be implementing a VHDL test-bench which would read an assembly program, load it in the CPU instruction memory, reset the CPU, perform the CNN analysis, and start over with the next program until all of them are tested. This can be quite complex and time-consuming to implement in VHDL, and it was therefore not attempted. There is a plethora of optimizations that can be implemented in order to improve speed and hardware consumption of the CNN, such as using binarized ANNs, or at least investigate how to make the tool instantiate DSP48 multipliers, and if timing allows, some data arrays could be implemented in BRAM18K memory blocks rather than using registers, which would greatly increase the size of the neural networks that could be deployed in the FPGA. In regards to the actual implementation, the CNN does not start until all 256 instructions have been vectorized, but it should be possible to start after only five instructions have been vectorized. However, care must be taken if the CNN is accelerated to the point that it analyzes one instruction per clock, since the CPU can be stalled under certain circumstances. Another possible optimization comes from noting that each time the CNN's first layer finishes producing an output row, four more input rows have to be read from memory, and the CNN is stalled during that reading process. Implementing a dual-port input channel could shave off 128 clock cycles, and doubling the line cache size could remove the 256 clock cycles completely. Many more optimizations should be possible, but unfortunately there was not enough time to implement and test them.
The final missing piece of the puzzle is the Serial-to-Parallel translator, which could not be implemented due to a lack of time. As we mentioned in section 2.2.2, the idea is to have a reconfigurable matrix of 'instruction execution units' (each unit would have two inputs and one output), with each column corresponding to a clock cycle, and each column connected to the next through reconfigurable steering logic and pipeline registers. This could easily allow transforming a serial algorithm to a parallel one (and even allow for loop pipelining), by simply populating as many instruction units as possible in the same column, while obeying data dependencies.

7.2 Conclusions

Regarding the main goals of the project, it can be said that we have managed to achieve all of them: the ANN was vastly improved from the previous generation, reaching a 99.8% accuracy; the ANN was successfully implemented in hardware and it works as expected when integrated with the rest of the system. We have also been able to explore the differences and problems which arise when developing hardware in HLS, and in this regard, my personal experience leads me to believe that HLS is not worth it. While it should be possible to achieve better results with the tool, structuring the C++ code in a way that the

compiler infers a satisfactory hardware becomes too complex, at which point VHDL starts to appear as a better solution. Moreover, there are many additional sources of optimizations and fine-tuning possible in VHDL, making it seem that the HLS approach to solving the problem, is a brute force solution. Another problem with the current implementation of the HLS tool, is that it's still extremely buggy (it crashes the whole system if a compilation is stopped) and there isn't a way to ensure a formal high-level equivalence with the hardware generated. And related to this, is the fact that it has very serious limitation: it is impossible to debug the VHDL code that the tool generates, which makes it impossible to find out where the differences between the HLS simulation and the RTL simulation are. Everything is fine if both simulations produce the same output values, but if these values are different, we are left with no tools to solve the problem. I honestly can't see a use-case in which one would opt for an HLS design, because if speed is not an issue, then using a processor should be good enough, but if speed and transistor use are relevant, then the extra work of porting the C++ code to VHDL is not that difficult and it's well justified. One positive aspect that I can mention is that the attempt of using HLS led to a design methodology of going from a very high level Python model, to a medium level C++ model, and a low level VHDL model which is, in my opinion, a very good approach to hardware design. And in light of this, I think that a tool that translates C++ code to readable VHDL could be much more useful. We have shown in section 5.2.1 how to transform a C++ loop into a rolled VHDL version using `if` statements, and we have used this technique in our project to unroll different loops at different levels. It should be possible to unroll C++ loops at the desired level using the same pragmas that Vivado HLS implements, and translate into VHDL code which could serve as a good starting point for the design.
With respect to the broader question on whether this approach to automatic parallelization *can* solve the problem, in principle there is nothing in the results which says that it shouldn't be able to. However, whether this is the *right* approach or not, is a much more difficult question to answer, and a lot more work needs to be done in order to have a clearer picture. While I still firmly believe that at some point, it will become a necessity to develop much less rigid CPUs, which should be able to reconfigure themselves towards the task they are performing at any moment, I'm not so sure that neural networks will be able to help in solving the problem. Just by noting that the number of transistors required by the CNN is at least one order of magnitude larger than the number required by the CPU it is trying to accelerate, and given the failure of ANNs to solve a task so simple such as detecting a loop's head and tail, this casts serious doubts about their usefulness on tasks where the results must often be exact and not approximate. However, I am by no means the first person to cast doubts about the usefulness of ANNs and I certainly won't be the last, but in a world where neural networks seem to become 'smarter' by the day, only time will tell where their real limits are.

Bibliography

[1] Alba, M. D., Kaeli, D. R., 2002. Characterization and evaluation of hardware loop unrolling. Tech. rep.

[2] Bourez, C., 9 2016. About loss functions, regularization and joint losses. `http://christopher5106.github.io/deep/learning/2016/09/16/about-loss-functions-multinomial-logistic-logarithm-cross-entropy-s html`.

[3] Brownlee, J., 7 2017. Why one-hot encode data in machine learning? [Online; accessed 5-December-2018].
URL `https://machinelearningmastery.com/why-one-hot-encode-data-in-machine-learning/`

[4] Contributors, S. U., 2018. Cs231n: Convolutional neural networks for visual recognition. `http://cs231n.github.io/convolutional-networks/`.

[5] Contributors, T., 2018. Pytorch documentation. `https://pytorch.org/docs/stable/nn.html`.

[6] contributors, X., 2 2015. Loop pipelining and loop unrolling. [Online; accessed 14-September-2018].
URL `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_2/sdsoc_doc/topics/calling-coding-guidelines/concept_pipelining_loop_unrolling.html`

[7] Courbariaux, M., Bengio, Y., 2016. Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. CoRR abs/1602.02830.
URL `http://arxiv.org/abs/1602.02830`

[8] Fabien Le Floc'h, Aug. 2017. The neural network in your cpu. [Online; accessed 11-September-2018].
URL https://chasethedevil.github.io/post/the_neural_network_in_your_cpu/

[9] Glorot, X., Bordes, A., Bengio, Y., 11–13 Apr 2011. Deep sparse rectifier neural networks. In: Gordon, G., Dunson, D., Dudk, M. (Eds.), Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics. Vol. 15 of Proceedings of Machine Learning Research. PMLR, Fort Lauderdale, FL, USA, pp. 315–323.
URL http://proceedings.mlr.press/v15/glorot11a.html

[10] Goff, G., 6 1991. Practical dependence testing. PLDI '91 Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation, 15–29.

[11] Goodfellow, I., Bengio, Y., Courville, A., 2016. Deep Learning. MIT Press, [Online; accessed 11-September-2018].
URL http://www.deeplearningbook.org

[12] Grochowski, A. M. M., 2018. Data augmentation for improving deep learning in image classification problem. 2018 International Interdisciplinary PhD Workshop (IIPhDW).

[13] HardOCP, Jul. 2018. Darpa launches 1.5 billion program to reinvent chip technology. [Online; accessed 11-September-2018].
URL https://www.hardocp.com/news/2018/07/30/darpa_launches_15_billion_program_to_reinvent_chip_technology

[14] Jalabert, L., 12 2018. Deep learning based cpu acceleration. [Online; accessed 03-June-2019].
URL https://github.com/LuisJalabert/Deep-Learning-Based-FPGA-CPU-Acceleration/blob/master/Deep_Learning_Based_CPU_Acceleration_-_Luis_Jalabert_December_2018.pdf

[15] Lim, Y. C., Liu, B., Nov 1988. Design of cascade form fir filters with discrete valued coefficients. IEEE Transactions on Acoustics, Speech, and Signal Processing 36 (11), 1735–1739.

[16] Olah, C., 2014. Understanding convolutions. http://colah.github.io/posts/2014-07-Understanding-Convolutions/.

[17] Ramachandran, P., Zoph, B., Le, Q. V., 2017. Searching for activation functions. CoRR abs/1710.05941.
URL `http://arxiv.org/abs/1710.05941`

[18] Shyamal Patel, J. P., 2017. Introduction to deep learning: What are convolutional neural networks? `https://www.mathworks.com/videos/introduction-to-deep-learning-what-are-convolutional-neural-networks html`.

[19] T. Moseley, D. Grunwald, D. A. C. R. R. V. T., Peri., R., 2006. Loopprof: Dynamic techniques for loop detection and profiling. Proceedings of the 2006 Workshop on Binary Instrumentation and Applications, WBIA 14.

[20] W. Warner, 12 2004. Great moments in microprocessor history. [Online; accessed 14-September-2018].
URL `https://www.ibm.com/developerworks/library/pa-microhist/index.html`