Architectures and Implementations of the Neural Network LeNet-5 in FPGAs

Evangelou Georgios, Roussos Nikolaos

***Abstract* — With the increasing availability of computational resources, machine learning has seen skyrocketing popularity in data analysis and processing, since it offers solutions to problems in various scientific areas. Neural networks are one of the most researched and applied subdomains of machine learning, providing effective data classification methods. For instance, real-time image recognition systems are of great commercial interest, such as face identification in smartphones. Thus, there is a growing need to develop area and power efficient solutions that meet the constraints of such applications, while maintaining high performance. As a consequence, machine learning is drifting away from software solutions, as they generally utilize inefficient general-purpose processors. Following this trend, this dissertation aims to design a hardware realization as an alternative. The proposed framework is based upon the LeNet-5 neural network and is able to recognize hand-written digits with great accuracy. The main achievement is an FPGA implementation, which achieves this goal, while the authors managed to familiarize with and utilize design tools provided with the Vivado Design Suite. Expanding the horizons of this diploma thesis, a second objective is to improve the system design, with regard to efficiency and performance, by exploiting word-length compression mechanisms and architectural acceleration methods. A close inspection of the network's structure reveals solutions of different functional characteristics, in order for the overall system's Key Performance Indicators (KPIs) to be enhanced. This framework is based upon, but not limited to, the current application. Therefore, following a similar workflow, other neural networks can be implemented, as well.**

***Index Terms* — Machine Learning, Neural Networks, LeNet-5, Image Recognition, High-Level Synthesis, Hardware Acceleration, FPGA Design**

I. Introduction

During the past few years the industry has shown an increasing interest in *artificial intelligence*. One of the main branches of artificial intelligence is *pattern recognition*. This particular branch deals with extracting useful information and features from large amounts of data and has seen increasing popularity, as well.

One of the most common applications of pattern recognition is *image recognition*. This is also the subject of this diploma thesis. More specifically, the *neural network LeNet-5* has been studied and used in order to recognize handwritten digits from the MNIST dataset. The aim of this diploma thesis is the creation of a whole new design framework for the neural network LeNet-5 not only in software, but also in hardware. This design has been used as a reference for conducting statistical measurements and optimizations in order to compare different versions of the initial implementation. Last but not least, various methods for the compression of the neural net, via quantization of its coefficients, have been studied, while investigating the impact of different code transformations and architectures on the accuracy, speed, and hardware resources of the neural net.

This dissertation is organized as follows. Section II summarizes the important information regarding pattern recognition, neural networks and the neural net LeNet-5. Section III describes the software implementation and weight extraction of LeNet-5 in Python. Section IV deals with the initial hardware implementation of LeNet-5 using a High-Level Synthesis tool: Vivado HLS. Section V and VI examine various methods for network compression and architectural optimization, respectively. Section VII describes the physical implementation of LeNet-5 on a development board. Finally, Section VIII presents our conclusions.

II. Pattern Recognition And Neural Networks

Pattern recognition is the scientific field that involves extracting features from input data automatically by using appropriate computational algorithms. *Classification methods* in particular, categorize input data into separate *classes*. In the case of this diploma thesis, the classes are all possible numerical digits (0-9) and the input data are images depicting handwritten digits to be recognized.

A. Neural Networks

Neural networks are a type of classification methods and are comprised of three basic elements: the *input*, the processing *layers*, and the *output*. The input data are basically the numerical values that characterize an input pattern based on the features used for classification. Every processing layer is made up of *neurons*. A neuron is the basic processing unit

Georgios Evangelou is a graduate of the Department of Electrical and Computer Engineering in University of Patras, Greece (e-mail: george_evangelou [at] windowslive.com). He is currently working at Laboratory for Manufacturing Systems and Automation, Patras, Greece.

Nikolaos Roussos is a graduate of the Department of Electrical and Computer Engineering in University of Patras, Greece (e-mail: nroussos97 [at] gmail.com). He is currently working at Metricon S.A., Patras, Greece.

within a neural network which accepts an input vector and outputs a single numerical value (or a set of such) based on some learning parameters called *weights* and an *activation function*. The latter is used in order to prevent the saturation of the output. Fig. 1 shows the structure of a **neuron c** which accepts the outputs of **neuron a** and **b** which are part of the previous layer. Note that $\mathbf{w_{c,a}}$ stands for the weight within **neuron c** in the **neuron c** – **neuron a** connection.

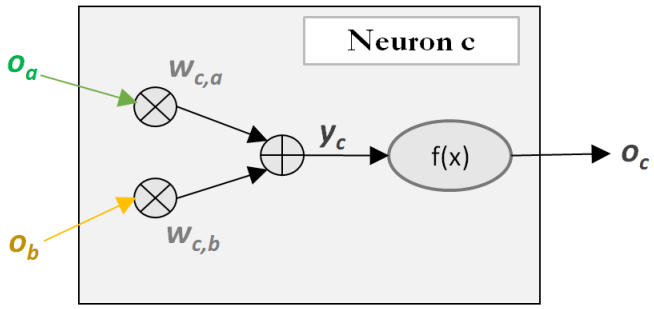

Fig. 1. The structure of a neuron. Here, **neuron c** accepts two inputs $\mathbf{o_a}$ and $\mathbf{o_b}$, which are later multiplied by $\mathbf{w_{c,a}}$ and $\mathbf{w_{c,b}}$ respectively. Finally, their sum is transformed into $\mathbf{o_c}$ by the activation function **f**.

A group of neurons accepting the same set of inputs is called a *layer*. There are many types of layers, such as *fully-connected*, *convolutional*, *pooling layers* etc.

As the name suggests, a fully-connected layer involves connecting all the neurons of the adjacent layers with its neurons. An example is depicted in Fig. 2.

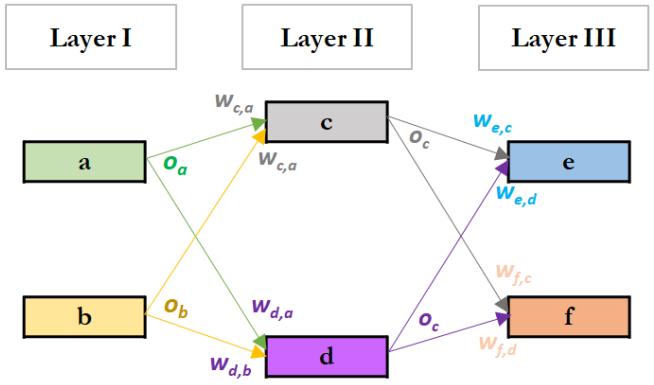

Fig. 2. A fully-connected neural network. Notice that every neuron belonging to layer II (c and d) is connected to every neuron of the adjacent layers I and III.

A convolutional layer is based on the signal convolution method [5]. In this case, the *feature map* plays the role of the 2D signal, on which a *kernel* (that is, the learning parameters-weights) slides. The function of a convolutional neuron is shown in Fig. 3. It should also be noted that the advantage of convolutional layers is the fact that they consist of a smaller number of weights and as such, decrease the computational complexity of the neural network [6]. From the hardware point of view this results in less memory requirements and reduced *data traffic* (data transfer/access) and, thus, *memory bandwidth*.

In order to reduce the required computational operations and *overfitting* [7], a pooling layer usually follows a convolutional layer. A pooling layer essentially down-samples a feature map by combining adjacent values (or pixels in the case of images). This is achieved either by *a)* calculating the average value (*average pooling)* or *b)* keeping the maximum value (*max pooling*). An example of an average pooling operation is shown in Fig. 4.

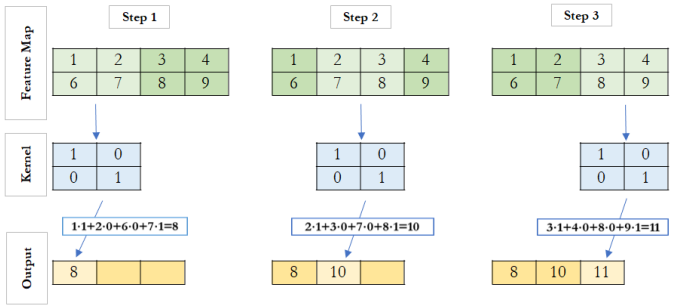

Fig. 3. An example regarding the function of a convolutional neuron. Note that the convolution *stride* is equal to 1. This means that the kernel slides by only 1 position at each step.

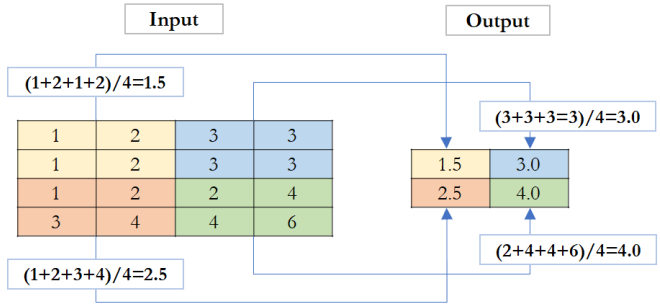

Fig. 4. An example regarding the function of a 2x2 average pooling layer. Note that the *pooling stride* is equal to 2.

B. Data and Structure of LeNet-5

The data used for the neural net LeNet-5 are part of the MNIST (Modified NIST) database [8]. The MNIST database consists of 28x28 grayscale 8-bit images of which 60,000 are training images (*training set*) and 10,000 test images (*test set*).

As shown in Fig. 5, the images are zero-padded in order to have a size of 32x32 and are then fed into the first convolutional layer (*CL1*) which consists of 6 kernels of size 3x3 and applies the activation function ReLU [9]. Note that the convolution reduces the size of the image to 30x30, because the 3x3 window is only applied to the internal pixels, in order to not get out of bounds. This layer is followed by a 2x2 average pooling layer (*AP1*). The same CL-AP block (*CL2*, *AP2*) is applied one more time and, then, a flattening layer reorders the 16 6x6 feature maps into a 576x1 vector. Afterwards, 3 fully-connected layers (*FC1*, *FC2*, *FC3*) follow using ReLU as an activation function, except for the last one which uses the Softmax [10][11] activation function for the final classification.

The layers which comprise LeNet-5 are summarized in Table I. Note that the total number of weights exceeds 80,000.

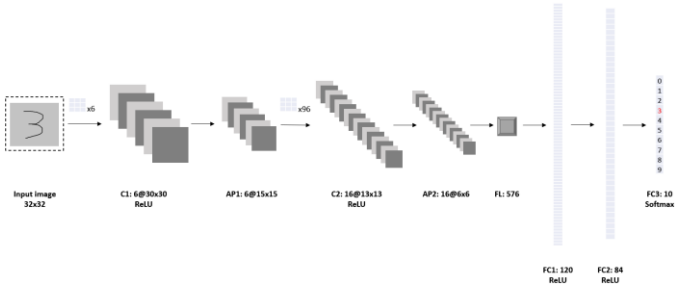

Fig. 5. The structure of the neural network LeNet-5. In every layer the number of feature maps and their dimensions are denoted. Note that the last layer uses 10 neurons which is equal to the number of classes (digits) to be recognized.

TABLE I
LENET-5 LAYER DIMENSIONS AND WEIGHTS

Layer	Input Dimensions	Number of Weights	Output Dimensions
Convolutional 1 (*CL1*)	1x32x32	60	6x30x30
Average Pooling 1 (*AP1*)	6x30x30	0	6x15x15
Convolutional 2 (*CL2*)	6x15x15	880	16x13x13
Average Pooling 2 (*AP2*)	16x13x13	0	16x6x6
Flattening (*FL*)	16x6x6	0	1x1x576
Fully-Connected 1 (*FC1*)	1x1x576	69240	1x1x120
Fully-Connected 2 (*FC2*)	1x1x120	10164	1x1x84
Fully-Connected 3 (*FC3*)	1x1x84	850	1x1x10

III. LENET-5 SOFTWARE IMPLEMENTATION

Due to the complexity of the algorithms used to train neural networks, the training of LeNet-5 was conducted via software. Thus, a software implementation was necessary. This was achieved by using Python and the Keras framework. It should be mentioned that the code and scripts developed can be found on the GitHub repository of this thesis [12]. The Keras implementation was based on [13] and follows the structure of Fig. 5. During training, 20% of the training set was used as a *validation set*. As a *loss function* the *categorical cross-entropy function* was used, because every image belongs to one and only one class (digit). The *Adam algorithm* with the default parameters provided by the authors of [14] was chosen in order to train the model. Using an image *batch size* of 128 and 10 *epochs*, the neural net performs as shown in Table II. Note that the average error in Table II is the final value of the loss function.

TABLE II
LENET-5 PERFORMANCE PER SET

Set	Recognition Accuracy (%)	Average Error
Training	99.47	0.0156
Validation	98.49	0.0811
Test	98.72	0.0444

In order to achieve the transition from a software to a hardware implementation, the architecture and the weights of the neural net need to be precisely reproduced. These two aspects were extracted from the Keras implementation to .json and .h5 files respectively. Subsequently, two Python scripts were developed in order to automate the process of transforming the weights into a form readable in C/C++ (.h file). Note that C/C++was the programming language used in the next step of the development process, because it was utilized later by the HLS tool to design hardware implementations (see Section IV). The Python scripts can be found at "2.LeNet_weights_extraction" of [12]. Finally, it is worth mentioning that the aforementioned scripts can work with neural networks with a different structure, as well.

IV. LENET-5 HARDWARE IMPLEMENTATION

Designing a hardware implementation poses certain challenges. First of all, developing using Hardware Description Languages – HDL, such as VHDL or Verilog, requires a lot of attention and effort, especially for a design as complex and vast as a neural network. Secondly, debugging is practically impossible considering the amount of data (28x28=784 inputs and 10 outputs), let alone the internal signals. Thus, this process needs to be automated.

A. Vivado HLS

In order to simplify the hardware implementation procedure, the High-Level Synthesis tool, Vivado HLS, was chosen. This tool has the capability of transforming a design described in C/C++ into VHDL or Verilog. Moreover, certain optimizations can be performed by means of *directives* and *pragmas*, both of which can transform certain blocks of code (see Section VI). As a result, these features speed up the process of developing the design's algorithm.

The three basic options that the tool offers during the design phase are the following [15]:

1. *C simulation*: Vivado HLS compiles the C/C++ code and checks its functionality by using a *testbench*. A testbench is a program, developed by the designer, which evaluates a design's functionality by utilizing known input-output pairs.
2. *Synthesis*: The tool transforms the C/C++ code into VHDL or Verilog, while providing an estimate of the hardware resources requirements, power dissipation and delay of the design. At the same time, certain constraints can be set by the user, such as a clock frequency constraint.
3. *C/RTL Co-simulation*: This is the final stage of the functionality check phase, where a concurrent simulation between the software and hardware model takes place. In essence, a consistency check is performed between the software (C/C++) and hardware (RTL, VHDL, Verilog).

B. LeNet-5 C Implementation

In order to successfully import LeNet-5 into Vivado HLS, it was necessary to model the whole network in C/C++. Thus, the structure of LeNet-5 (Fig. 5) was developed and tested for consistency with the Keras implementation (see Section III).

Apart from the consistency test, certain modifications in the source code were needed in order to successfully import the C/C++ model into Vivado HLS. The main modification was the usage of static data structures (arrays) exclusively, as the use of dynamic data structures and pointers rendered the design non-synthesizable. This is to be expected, as a hardware design cannot allocate resources dynamically.

Moreover, the library "ap_fixed" was used for the utilization of *arbitrary precision fixed point arithmetic* (more details in Section V). An important side-note is that this library posed unexpected issues on Windows 10 during development. Hence, Ubuntu 18.04 was preferred.

Last but not least, a testbench was developed in order to verify the design's functionality. As previously mentioned, known input-output pairs were first required. By using Keras, the images (inputs) and the corresponding outputs (values and image labels) were exported into separate files. These files were utilized by the testbench program, in order to calculate the *TOP-1*, *TOP-3*, and *TOP-5* accuracies. The developed testbench can be configured to accept inputs from both the training and test set. Using the entire test set (10,000 images) as an input, a TOP-1 accuracy of 98.72% was calculated, which is consistent with the performance shown in Table II.

Developing a testbench which validates the design's correct functionality is strongly suggested by [15]. Indeed, the testbench proved to be very important, because every optimization performed (see Section V and VI) could alter the functionality of the design.

It should be noted that the LeNet-5 C implementation is located at "3.LeNet_in_C" of [12].

V. LeNet-5 Compression

The Keras model of LeNet-5 uses 32-bit *floating point arithmetic* for both data and weights. Applying this arithmetic to a hardware design would result in significant speed, energy and resource inefficiency. Thus, *fixed point arithmetic* was preferred in order to reduce the computational complexity of the algorithm. Another improvement was the usage of a smaller *word length* [16]. However, the above actions can negatively affect the accuracy of the neural net. Thus, the adoption of a quantization (compression) method, which reduces the computational complexity while maintaining an acceptable accuracy loss, is necessary.

A. Compression Methods

1) Uniform Quantization Method

This is the simplest method, as the exact same quantization over all layers is applied. However, in order to analyze this, and the other compression methods, the following definitions need to be made:

1. *IL* (*Integer Length*): The number of integer bits.
2. *FL* (*Fractional Length*): The number of fractional bits.
3. *BL* (*Bit Length*): The total number of bits (*IL+FL*).

The authors of [17] suggest that this method can pose certain problems during training, such as delay the convergence of the algorithm, or worse, introduce instabilities. These problems are caused due to the method's restrictive nature, as the most sensitive layer determines the lowest acceptable (*IL, FL*) parameter pair. However, the usage of the same *BL* in every layer allows for simpler implementations.

2) Dynamic Fixed Point Method

This is a more complicated technique as different (*IL, FL*) pairs are allowed per layer. There are two different versions:

Version 1. In each layer the following equation holds:

$$(IL, FL)_{in} = (IL, FL)_{wei} = (IL, FL)_{out} \quad (1)$$

where $(IL, FL)_{in}$, $(IL, FL)_{wei}$, and $(IL, FL)_{out}$ the (*IL, FL*) pair of the *input*, *weights*, and *output* of a particular layer, respectively.

Version 2. This is a generalization of the first version. In this case, the terms of (1) are not necessarily equal and, thus, (1) does not hold. As a result, this is a much more flexible method, albeit more complicated.

The authors of [18] and [19] suggest that the accuracy requirements differ from one layer to another. Consequently, a per-layer approach can lead to acceptable accuracy while allocating fewer resources. In [20] the same conclusion is reached, while it is also suggested that using a smaller word length reduces the probability of overfitting in convolutional neural networks.

It is evident that an increase in the depth of a neural network results in a rapid expansion of the solution space regarding the minimum (*IL, FL*) pairs for each layer. However, this technique takes full advantage of each layer's tolerance in terms of accuracy drop. Thus, the word length of each layer can be reduced independently from the other layers, enabling the use of smaller and faster memory units.

3) Ristretto Method

Ristretto [21] is a tool that can find the (*IL, FL*) parameter pair of each layer within a neural network. Apart from that, Ristretto enables designers to examine the impact of different arithmetic systems and word lengths on the behavior of their neural net [19]. Ristretto is a tool that depends on *Caffe* [22], another software tool, which is used for modeling and training neural networks, like Keras. A model is described using .prototxt files on Caffe in a similar way to how Python is used in conjunction with Keras.

It is worth mentioning that Ristretto supports three types of quantization [23]:

1. *Dynamic fixed point arithmetic.*
2. *Minifloat*: floating point arithmetic of word length ≤ 16-bit.
3. *Multiplier-free arithmetic*: the weights of the network are rounded towards the nearest power of *2*, in order to transform multiplications into bit shifts.

From the three types of quantization that Ristretto supports only dynamic fixed point arithmetic was used. Minifloat is not appropriate for hardware implementations, because of the floating point arithmetic. Finally, multiplier-free arithmetic is characterized by the largest quantization errors, overall [19].

Ristretto executes the following process in order to find the (*IL, FL*) parameter pairs for each layer [23]:

Step 1. Analysis of the dynamic range of the network's weights. An appropriate *IL* is selected in order to avoid saturation.

Step 2. Calculation of statistical parameters for effective quantization by activating the network multiple times.

Step 3. Finding the best $(IL, FL)_{in}$, $(IL, FL)_{wei}$, and $(IL, FL)_{out}$ for each layer, while keeping the rest of the layer in

normal accuracy.

Step 4. Test the accuracy using the training set.

Step 5. Fine-tuning. Retrain the (now compressed) neural network.

The last step is executed in order to compensate for the accuracy loss due to the quantization of the neural network. A key-detail of this step is the fact that the retraining of the network is performed at full word length (32-bit), although the data and weights have been quantized. This is done because minor weight changes cannot be detected by the compressed network. However, as retraining is a repetitive procedure, the accumulation of such small changes can have a significant impact on the accuracy of the network.

B. Compression Methods in LeNet-5

1) Uniform Quantization Method

An important observation of [18] is that the weights of a neuron typically range between *-1* and *1* (both inclusive). This enables fixing IL_{wei} to *1*. The authors of [18] also suggest that for LeNet-5, a BL_{wei} of*8* (that is, $FL_{wei}=7$), and $(IL, FL)_{in}$ = $(IL, FL)_{out}$ = (*8, 2*) will result in no accuracy drop. The (*IL, FL*) pairs are summarized in Table III. Note that in this thesis one extra layer is added, because in [18] the neural net LeNet-4 was used, which has one less layer.

TABLE III
INITIAL AND USED (*IL, FL*) PAIRS FOR APPLYING UNIFORM QUANTIZATION

Layer	Configuration from [18]			Adjusted Configuration		
	Input	Weights	Output	Input	Weights	Output
Conv. 1 (CL1)	(8,2)	(1,7)	(8,2)	(8,2)	(1,7)	(8,2)
Avg. Pl. 1 (*AP1*)	(8,2)	-	(8,2)	(8,2)	-	(8,2)
Conv. 2 (*CL2*)	(8,2)	(1,7)	(8,2)	(8,2)	(1,7)	(8,2)
Avg. Pl. 2 (*AP2*)	(8,2)	-	(8,2)	(8,2)	-	(8,2)
F.-Conn. 1 (*FC1*)	(8,2)	(1,7)	(8,2)	(8,2)	(1,7)	(8,2)
F.-Conn. 2 (*FC2*)	(8,2)	(1,7)	(8,2)	(8,2)	(1,7)	(8,2)
F.-Conn. 3 (*FC3*)	-	-	-	(8,2)	(1,7)	(8,2)

2) Dynamic Fixed Point Method

According to [18], performing a per-layer quantization can save more hardware resources, while maintaining an acceptable accuracy drop. The statistical analysis conducted by the authors of [18] showed *92%* data traffic drop with *1%* accuracy tolerance for LeNet-4. Depending on the sensitivity of each layer, the word lengths can be further reduced. Thus, every layer is adjusted independently. The (*IL, FL*) parameter pairs are summarized in Table IV. However, some adjustments were necessary in order to avoid saturation, which are shown in the "Adjusted Configuration" column of Table IV. The main deviation from the initial configuration is the use of 8 bits for the input of the 1st convolutional layer.

3) Ristretto Method

Following the guidelines of [24] the architecture of LeNet-5 was developed while making slight modifications in order to match the specific structure of Fig. 5. The training was performed using the same training algorithm (Adam), loss function (categorical cross-entropy), batch size (=128), and epochs (=10). As a result, although a different software tool was used (Caffe), the training of LeNet-5 was performed under the same conditions as those applied by Keras. The relative files can be found at "7.LeNet_in_Ristretto" of [12].

TABLE IV
INITIAL AND USED (*IL, FL*) PAIRS FOR APPLYING DYNAMIC FIXED POINT QUANTIZATION

Layer	Configuration from [18]			Adjusted Configuration		
	Input	Weights	Output	Input	Weights	Output
Conv. 1 (CL1)	(1,1)	(1,3)	(3,0)	(8,0)	(1,2)	(3,1)
Avg. Pl. 1 (*AP1*)	(3,0)	-	(3,0)	(3,1)	-	(3,1)
Conv. 2 (*CL2*)	(3,0)	(1,5)	(3,0)	(3,1)	(1,4)	(3,1)
Avg. Pl. 2 (*AP2*)	(3,0)	-	(3,0)	(3,1)	-	(3,1)
F.-Conn. 1 (*FC1*)	(3,0)	(1,7)	(3,0)	(3,1)	(1,6)	(3,1)
F.-Conn. 2 (*FC2*)	(3,0)	(1,5)	(3,0)	(3,1)	(1,6)	(3,1)
F.-Conn. 3 (*FC3*)	-	-	-	(3,1)	(1,4)	(3,1)

Compressing LeNet-5 using Ristretto with a *1%* accuracy drop margin, the configuration of the column "Ristretto Configuration" of Table V is extracted. It should be clarified that the most significant bit has negative weight, following the rules of the two's complement representation. As a result, when *IL=0*, the first fractional bit has negative weight, which translates into a dynamic range of [*-0.5, 0.5*).

TABLE V
INITIAL AND USED (*IL, FL*) PAIRS FOR APPLYING RISTRETTO QUANTIZATION

Layer	Ristretto Configuration			Adjusted Configuration		
	Input	Weights	Output	Input	Weights	Output
Conv. 1 (CL1)	(8,0)	(1,3)	(8,0)	(8,0)	(0,4)	(8,0)
Avg. Pl. 1 (*AP1*)	(8,0)	-	(8,0)	(8,0)	-	(9,0)
Conv. 2 (*CL2*)	(8,0)	(0,4)	(8,0)	(9,0)	(0,4)	(8,0)
Avg. Pl. 2 (*AP2*)	(8,0)	-	(7,1)	(8,0)	-	(7,1)
F.-Conn. 1 (*FC1*)	(7,1)	(0,4)	(8,0)	(7,1)	(0,4)	(8,0)
F.-Conn. 2 (*FC2*)	(8,0)	(0,4)	(8,0)	(8,0)	(0,4)	(8,0)
F.-Conn. 3 (*FC3*)	(8,0)	(0,4)	(7,1)	(8,0)	(0,4)	(6,4)

The use of the initial (*IL, FL*) pairs in LeNet-5 resulted in incorrect functionality, due to the saturation of some internal signals. The most significant indication for this was the fact that the neural net recognized all images as the digit 0. Therefore, the word length of each layer was increased by 1 repetitively and different (*IL, FL*) pairs were tested per layer. From the above actions the configuration of the column "Adjusted Configuration" of Table V was derived.

The main differences between the initial and the adjusted configuration are *a)* the addition of 1 bit to the input of the 2nd convolutional layer, and *b)* the addition of 2 bits to the network's output (i.e. the output of the 3rd fully connected layer). However, this last addition introduces inconsequential computational overhead given that the final layer comprises of only 10 neurons. Moreover, it should be noted that the input of the first layer is unsigned, because the pixels of the images have no negative values. One final observation is the fact that smaller word lengths are used for the weights compared to the data (approximately half the word length of the data). This fact justifies the separation of (*IL, FL*) into $(IL, FL)_{in}$, $(IL, FL)_{wei}$, and $(IL, FL)_{out}$.

C. Compression Methods Comparison

1) C Simulation Results

Utilizing the testbench program mentioned in Section IV, the TOP-1, TOP-3, and TOP-5 accuracies for LeNet-5 are calculated. These accuracies (shown in Table VI) were calculated using 1,000 images of the test set. Setting the *floating point 32-bit* configuration as a reference point, it is evident that both *floating point 16-bit* and *uniform fixed point 32-bit* configurations do not introduce any accuracy loss. Meanwhile, *uniform fixed point 16-bit*, *uniform fixed point (from [18])*, and *dynamic fixed point (from Ristretto)* configurations are characterized by a small accuracy drop (up to 3.1%). Setting 90% as the threshold for acceptable accuracy, both *uniform fixed point 8-bit* and *dynamic fixed point (from [18])* configurations are discarded.

TABLE VI
LENET-5 ACCURACY COMPARISON FOR VARIOUS CONFIGURATIONS

Configuration	Accuracy (%)		
	TOP-1	TOP-3	TOP-5
Floating point 32-bit	98.3	100.0	100.0
Floating point 16-bit	98.3	100.0	100.0
Uniform fixed point 32-bit	98.3	100.0	100.0
Uniform fixed point 16-bit	97.5	99.6	99.6
Uniform fixed point 8-bit	51.7	74.8	77.0
Uniform fixed point (from [18])	95.2	97.6	97.6
Dynamic fixed point (from [18])	52.4	60.6	60.6
Dynamic fixed point (from Ristretto)	96.1	99.3	99.4

It is worth mentioning that although Ristretto uses only 4 bits for the weights, a much higher accuracy is achieved compared to the *uniform fixed point 8-bit* configuration, because of the more systematic digit distribution. Also, note that the uniform fixed point configurations use equal *IL* and *FL* bits. Finally, as the number of classes is only 10 (10 digits), TOP-1 accuracy is considered to be the only reliable performance measure.

2) Synthesis Results

The Synthesis option that Vivado HLS offers, apart from translating the design from C/C++ into a hardware description language, also calculates the resource usage, delay, and interval (time spacing between two successive inputs) of the design. The results that follow were obtained using the Kintex-7 FPGA, *xc7k160tfbg484-1*, and a *10 ns* clock.

In order to gain a better understanding of the resources allocated, a quick explanation of the basic functional units of an FPGA is necessary. The basic blocks of a Xilinx FPGA are the following [25]:

1. *LookUp Table* (*LUT*): A basic block that can realize logic functions. A LUT can also be used as a small memory, because it essentially holds a truth table.
2. *Flip Flop* (*FF*): The most basic memory unit that holds internal data.
3. *DSP48*: An *Arithmetic Logic Unit* (*ALU*) which contains a 25x18 bit multiplier and a 48-bit accumulator.
4. *Block RAM* (*BRAM*): BRAMs are dual-ported memories that are capable of storing large amounts of data (18 or 36 kbits). Dual-ported memories allow for two data accesses per clock cycle.

The resource usage per configuration is shown in Fig. 6. Notice that the *uniform fixed point 8-bit* and *dynamic fixed point (from [22])* configurations are missing, because they failed the functionality test. As expected, the more compressed an implementation is, the less the number of required BRAMs. It is worth mentioning that the initial *floating point 32-bit* configuration requires half of the FPGA's BRAMs, which justifies the necessity of the network's compression. Meanwhile, the fixed point configurations require 0 or almost 0% DSP48 units. As for FFs and LUTs, smaller differences are observed between the various configurations.

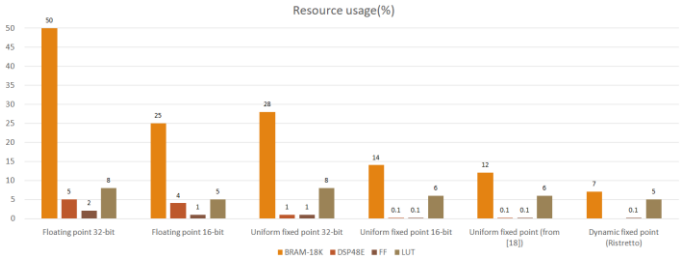

Fig. 6. LeNet-5 resource usage comparison for various configurations.

Apart from the resource usage comparison, Vivado HLS offers the ability to compare different designs in terms of speed. The delay per configuration is shown in Fig. 7. It is evident that the floating point configurations are approximately 300% slower than the fixed point ones and also require more resources as reported in Fig. 6.

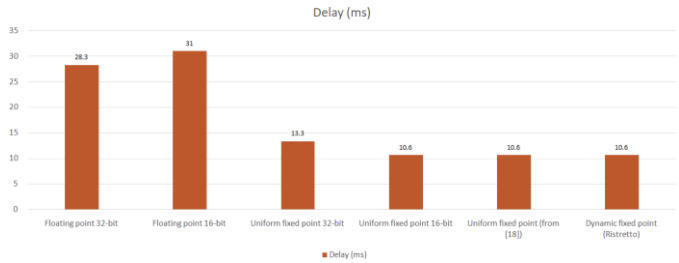

Fig. 7. LeNet-5 inference delay comparison for various configurations.

The configuration given by Ristretto requires the fewest resources according to Fig. 6. More specifically, only 14% of

the initial configuration's (*floating point 32-bit*) memories are needed, while no DSP48 units are necessary. The latter suggests that the arithmetic operations are performed by the smaller and faster LUTs. Furthermore, the reduction of the circuit's area results in a decrease in the *routing delay*.

The speed improvement and the smaller area are the main factors for choosing the Ristretto configuration in this work. Despite the 2.2%accuracy drop, this configuration is a good tradeoff between compression and performance.

VI. Architectural Optimizations

The compression of the neural network LeNet-5 resulted in significantly less resource usage and delay compared to the initial floating point implementation. However, there is still room for improvement, considering that no architectural optimizations (such as loop unrolling, pipeline etc.) have been applied yet. Luckily, due to the resource savings achieved by the Ristretto configuration (Fig. 6), experimentation with architectural optimizations is less restricted. That is because such optimizations typically require considerably more resources.

A. Code Restructuring

The first step towards achieving better performance is restructuring the code. Considering that the code developed will eventually result in a hardware design, it is very important that the resources are utilized effectively. This includes optimizing read/write operations within tables (memory accesses) and reducing the number of processing stages required to produce the output. The latter essentially shortens the *critical path* of the design.

The examples that follow optimize the performance of fully-connected neurons. It should be noted that the principles shown apply to convolutional neurons, as well.

The algorithm for calculating the output of a fully-connected layer's neuron is shown in Fig. 8. Notice that both *init_loop* and *acc_1_loop* access the array *internal_state*. However, these two loops can be merged and, thus, the memory traffic can be reduced. The restructured code is shown in Fig. 9.

Further reduction in terms of memory accesses can be achieved by merging *out_loop* into *acc_1_loop*. The final code is depicted in Fig. 10.

B. Code Transformations

Before compiling the code, the preprocessor can analyze the code and perform certain transformations. This aims to alter the design in terms of structure, resources and performance, without changing its functionality, however [26]. In some cases, the aforementioned transformations do not affect the code, but rather how the code will be translated into hardware.

1) Loop Unrolling

Loop unrolling (*partial/complete*) transforms a loop into a series of separate commands. This either reduces or eliminates (depending on the type of the unrolling) the overhead introduced by the condition check at the start of the loop and also makes room for parallelization. Indeed, the separate commands can now be executed concurrently. Moreover, a loop can also be partially unrolled by a specific factor. An example is shown in Fig. 11.

```
// Data declaration
X: a neural network layer
Y: the layer previous to layer X
n: a neuron of layer X
m: a neuron of layer Y
w[n][m]: the weight connecting neuron m to neuron n

init_loop:
For each neuron n in layer X:
        internal_state[n] = bias[n]

acc_1_loop:
For each neuron n in layer X:
        acc_2_loop:
        For each neuron m in layer Y:
                internal_state[n] += output[m] * w[n][m]

out_loop:
For each neuron n in layer X:
        output[n] = f(internal_state[n])
```

Fig. 8. The initial algorithm for calculating the output of a fully-connected layer's neuron.

```
// Data declaration
X: a neural network layer
Y: the layer previous to layer X
n: a neuron of layer X
m: a neuron of layer Y
w[n][m]: the weight connecting neuron m to neuron n

acc_1_loop:
For each neuron n in layer X:
        internal_state[n] = bias[n]
        acc_2_loop:
        For each neuron m in layer Y:
                internal_state[n] += output[m] * w[n][m]

out_loop:
For each neuron n in layer X:
        output[n] = f(internal_state[n])
```

Fig. 9. The restructured algorithm for calculating the output of a fully-connected layer's neuron after merging *init_loop* into *acc_1_loop*.

```
// Data declaration
X: a neural network layer
Y: the layer previous to layer X
n: a neuron of layer X
m: a neuron of layer Y
w[n][m]: the weight connecting neuron m to neuron n

acc_1_loop:
For each neuron n in layer X:
        internal_state[n] = bias[n]
        acc_2_loop:
        For each neuron m in layer Y:
                internal_state[n] += output[m] * w[n][m]
        output[n] = f(internal_state[n])
```

Fig. 10. The final algorithm after merging *out_loop*.

```
For n from 1 to 20:
        A[n] = 1 + A[n]
```

(a)

```
For n from 1 to 20 with step 4:
        A[n] = 1 + A[n]
        A[n+1] = 1 + A[n+1]
        A[n+2] = 1 + A[n+2]
        A[n+3] = 1 + A[n+3]
```

(b)

Fig. 11. (a) Initial loop, and (b) loop unrolled by a factor of 4.

2) Loop Pipelining– Dataflow

When there are no data dependencies (*RAW*, *WAR*, *WAW* and *control dependencies*) [27], the *pipeline* transformation can be applied. This mechanism essentially divides a process into separate processing stages, which can be executed concurrently. An example is depicted in Fig. 12 and Fig. 13. Notice that after an initial delay (*latency*) of *3* clock cycles, an output is produced every cycle, and thus an *interval* equal to *1* is achieved. As a result, the loop can accept a new input every clock cycle.

		Clock cycle								
		1	2	3	4	5	6	7	8	9
Data	A	input	process	output						
	B				input	process	output			
	C							input	process	output

Fig. 12. Loop with no pipeline. An output is produced every *3* clock cycles.

		Clock cycle								
		1	2	3	4	5	6	7	8	9
Data	A	input	process	output						
	B		input	process	output					
	C			input	process	output				

Fig. 13. Loop with pipeline. After an initial delay of *3* clock cycles, an output is produced every cycle.

Similarly, *dataflow* pipelines functions by using intermediate buses. This allows the "consumer" functions to begin executing before the "producer" functions have finished.

3) Loop Flattening

Loop flattening compresses nested loops into one loop as shown in Fig. 14. Therefore, it eliminates the control overhead when transitioning between different loop levels.

In order to apply this transformation, it is necessary that the loop nest is either *perfect* or *semi-perfect*, which is defined by the following rules:

Rule 1. Only the innermost loop has loop body content.

Rule 2. There is no logic specified between the loop statements.

Rule 3. All loop bounds are constant, except maybe for the outermost in which case the loop nest is semi-perfect.

```
For n from 1 to N:
        For m from 1 to M:
                For o from 1 to O:
                        command_1
                        ...
                        command_i
```

(a)

```
For n from 1 to (N*M*O):
        command_1
        ...
        command_i
```

(b)

Fig. 14. (a) Initial loop nest and (b) flattened loop nest.

4) Array Partitioning

Most data are stored in arrays. Vivado HLS implements arrays as RAMs [28]. However, as mentioned in Section V, these memories can only access 2 bytes per clock cycle and can, thus, introduce a bottleneck.

Array partitioning is a technique that utilizes more memory units in order to accelerate memory accesses. This is achieved by splitting an array into smaller ones (see Fig. 15). Vivado HLS supports three type of array partitioning:

1. *Block*: The array is split into continuous blocks.
2. *Cyclic*: The array is split into blocks containing elements circularly.
3. *Complete*: In this case every element of the array is stored in a separate register and, thus, all elements can be accessed concurrently.

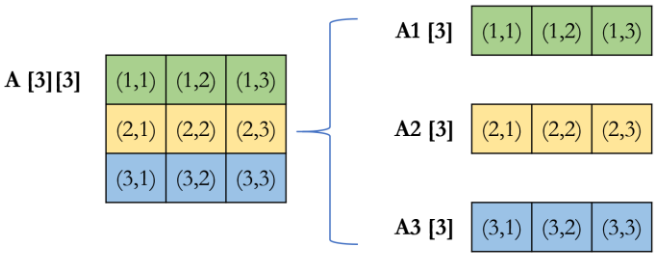

Fig. 15. Block partitioning of the first dimension of array A by a factor of 3.

5) Array Mapping – Array Reshape

Array mapping merges smaller arrays either horizontally (by concatenating arrays) or vertically (by increasing the word length of the elements) into a bigger one. As a result, fewer RAMs are used.

Array reshape combines both array partitioning and vertical array mapping by merging array elements into elements with increased word length. This results in fewer memory units while maintaining concurrent access to data, because with one access more than one element is fetched. Similarly to other methods, array reshape can be performed with three variations: i) block, ii) cyclic and iii) complete reshaping. An example for block array reshape with a factor of 3 for a two dimensional array A[3][3] is shown in Fig. 16.

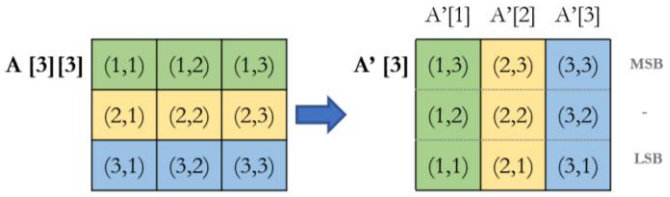

Fig. 16. Array reshape of the second dimension of array A by a factor of 3.

C. Code Transformations with Vivado HLS

Vivado HLS offers the ability to create alternative architectures of a certain design which are called *solutions*. This feature enables experimentation with code transformations, such as those previously mentioned. Vivado HLS applies transformations by utilizing *pragmas* and/or *directives*. The main difference between them is that pragmas are part of the source code, and as a result are shared between solutions, whereas directives are solution-specific.

1) Code Structure

In order to understand the code transformations applied to LeNet-5, a brief description of the code structure (Fig. 17) is necessary.

1. *Convolutional layers*: Convolutional layers first access the neurons (*neurons_loop*) and then the feature maps. The latter require three loops (*kernel_0_loop – kernel_2_loop*), because feature maps form a set of 2D images. After that, the 2D

kernel is applied (*kernel_3_loop* and *kernel_4_loop*). Finally, the last two loops apply the activation function ReLU to the 2D image output of each neuron.

2. *Average pooling layers*: These layers access a set of 2D feature maps and, thus, require three nested loops.
3. *Flattening layer*: This layer reorders the elements of the 3D array holding the feature maps into a 1D array. Therefore, three nested loops are needed.
4. *Fully-connected layers*: As shown in Fig. 10 fully-connected layers require only two nested loops. However, the use of the Softmax function in the last layer demands an additional loop, because a sum calculation is needed [10].

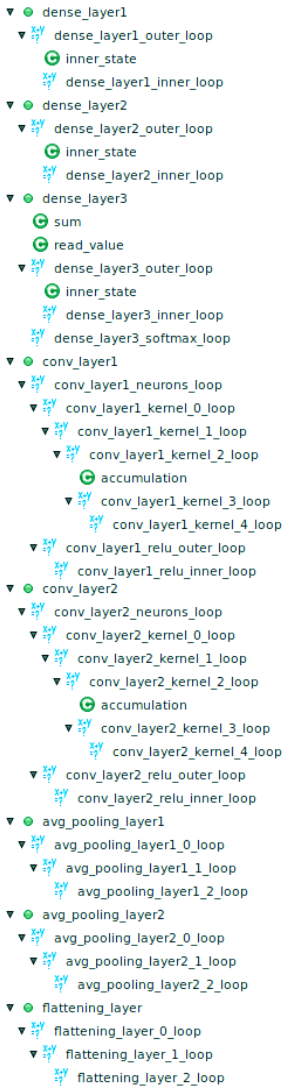

Fig. 17. Code structure of LeNet-5.

The selected directives and the order with which they were applied are based on [15]. It should be mentioned that the transformations that follow are additive, meaning that every solution extends its predecessor. Finally, the specific directives and transformations used, along with their corresponding synthesis reports can be found at "8.Optimizations" of [12].

2) Applying Loop Pipelining

First, the pipeline directive was applied to the innermost loop of a loop nest as suggested by [15]. This is done because loop pipelining requires the inner loops to be unrolled, which results in notable increase in resource usage. Apart from that, the default behavior of Vivado HLS is to try to flatten the loop with its outer ones.

Initially, the pipeline directive was applied to the innermost loops of the average pooling, flattening and fully-connected layers. However, the loop nest could not be flattened in the fully-connected layers, because of the commands that lie between the loops (*inner_state* initialization between *outer_loop* and *inner_loop*). Applying the pipeline directive to the outer loop results in 147% LUT usage (which, of course exceeds the FPGA limits), due to the fact that the inner loop is unrolled.

However, because of the data dependencies, a pipeline with an interval of 1was not possible for the average pooling layers. This problem was solved by using the partial array partition and array reshape directives on the input arrays, as they enabled concurrent access to the arrays' elements. The architecture encompassing the above proposed transformations is named *Pipeline 1*.

Using the Synthesis option of Vivado HLS a 14% delay drop (delay = 9.1 ms) is reported with only a 2% increase in LUTs compared with the initial Ristretto implementation (see Fig. 6 and Fig. 7).

Afterwards, the pipeline directive was applied to the *kernel_2* loops of the convolutional layers. These loops were chosen because they are the innermost loops of the perfect loop nest (*kernel_0 – kernel_2*) and because the inner loops (*kernel_3* and *kernel_4*) have a small number of iterations (3 each, 9 in total). However, the pipeline optimization was unable to achieve an interval of 1 which, again, was solved by transforming the input arrays. Finally, the loop *relu_inner* was pipelined, directly achieving an interval of *1*. We name this architecture *Pipeline 2*.

This time, using 5% more LUTs and 1% more FFs, an 85% delay drop (delay = 1.57ms) is achieved compared with the initial Ristretto implementation.

3) Applying Loop Unrolling

Further acceleration can be achieved by using the unroll transformation. However, extra caution is needed, as this transformation can result in much higher resource usage. Thus, the aim is a good tradeoff between acceleration and resource usage.

In order to make an effective use of the unroll technique, the bottlenecks of the design should first be determined. Vivado HLS reports synthesis statistics for every function, which in the case of LeNet-5 are its layers. Using this option, the results in Table VII are obtained. The fact that the 2nd convolutional layer in conjunction with the 1st fully-connected layer account for 79% of the total delay makes these two layers the best candidates for applying the unroll transformation.

TABLE VII
PER LAYER DELAY CONTRIBUTION

Layer	Delay (ms)	Delay (%)
Conv. 1 (CL1)	0.19	12.1
Avg. Pl. 1 (*AP1*)	0.01	0.6
Conv. 2 (*CL2*)	0.54	34.4
Avg. Pl. 2 (*AP2*)	0.01	0.6
F.-Conn. 1 (*FC1*)	0.70	44.6
F.-Conn. 2 (*FC2*)	0.10	6.4
F.-Conn. 3 (*FC3*)	0.01	0.6
Total	1.57	100.0

Beginning from the 1st fully connected layer, the loop pipeline transformation of *inner_loop* was replaced by loop unrolling. Tests showed that using a partial unroll with a factor of 36 is a good tradeoff between speed and hardware requirements. This architecture is named *Unroll 1*. Synthesis reports a 20% speed increase (delay = 1.26ms) with a 7% increase in LUTs compared with the *Pipeline 2* architecture.

Unrolling *outer_loop*, which also completely unrolls *inner_loop*, results in 110% LUTs usage, which, of course, is not a viable option.

Next, the unroll transformation was applied to the 2nd convolutional layer. More specifically, the pipeline directive was moved one level higher. That is, from *kernel_2* to *kernel_1*. As previously mentioned, the inner loops are automatically completely unrolled, which in this case includes *kernel_2* loop. Again, a pipeline with an interval of 1was not directly achieved. This time the problem was solved by completely partitioning the input array instead of partially. We call this architecture *Unroll 2*.

In comparison with the *Unroll 1* architecture, *Unroll 2* speeds up the neural net by 37% (delay = 0.80ms), while utilizing almost triple the number of LUTs (47% of the total available).

Afterwards, the unroll directive was applied to the 1st convolutional layer, which is the third slowest according to Table VII. Repeating the same actions taken for *Convolutional Layer 2* (moving the pipeline directive one level higher and completely partitioning the input array), the *Unroll 3* architecture was created.

This time a delay equal to 0.68ms is achieved, while using 88% of the FPGAs LUTs.

4) *Applying Dataflow*

Finally, the dataflow directive is applied to the design. This directive essentially performs a pipeline on the function level (or rather on the layer level in the case of LeNet-5). The dataflow transformation is applied to the architectures *Unroll1*, *Unroll2* and *Unroll3*. The new architectures are named *Dataflow 1*, *Dataflow 2*, and *Dataflow 3*, respectively. Synthesis reports a lower interval and relatively the same percentage of resource usage compared with the corresponding pipeline architectures.

D. *Architectures Comparison*

The architectures designed in the previous paragraph are compared in terms of hardware usage and speed in Fig. 18 and Fig. 19, respectively. The *Initial* architecture uses minimal resources but is also the slowest. However, applying the pipeline directive to all layers (*Pipeline 2* architecture), results in speeding up the neural net by approximately *7* times, while using only an additional 5% of the FPGA's LUTs.

The unroll architectures further reduce the delay, but they cannot achieve a big speed increase such as the one accomplished by *Pipeline 2*. Both *Unroll 2* and *Unroll 3* architectures result in a delay lower than 1ms, but they require almost 50% and 90% of the FPGAs LUTs, respectively.

The dataflow architectures use practically the same percentage of resources and achieve the same delay as their predecessors ($delay_{Unroll\ 1} = delay_{Dataflow\ 1}$ etc.) but reduce the interval. This means that the neural net can accept new inputs more frequently. Also, notice that *Dataflow 2* and *Dataflow 3* achieve the same interval (*0.39* ms), but the latter requires almost double the resources of the former. Despite the delay difference, these two architectures can accept new data at the same rate.

In case the margins in terms of hardware are tight, *Initial*, *Pipeline 1* and *Pipeline 2* are the most appropriate architecture choices. However, if the main demand is speed, one of the remaining architectures should be selected. As for the FPGA board implementation (Section VII), the *Unroll 2* architecture was chosen.

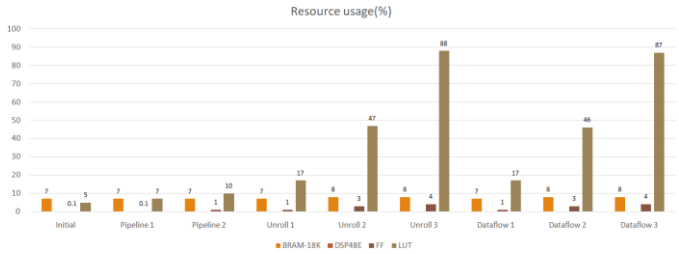

Fig. 18. LeNet-5 resource usage comparison for various architectures (solutions). Notice that no architecture requires the use of DSP48 units.

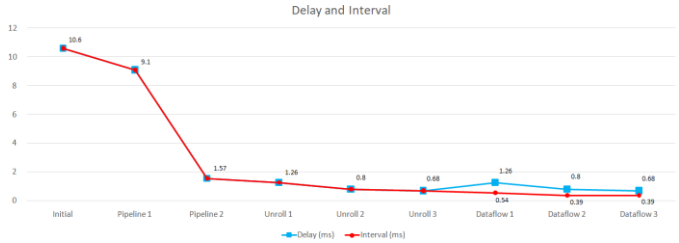

Fig. 19. LeNet-5 delay and interval comparison for various architectures (solutions).

VII. LENET-5 IMPLEMENTATION ON DEVELOPMENT BOARD

After studying and optimizing the behavior of LeNet-5 in a simulation environment, it is time to test its performance in a practical implementation. In order to accomplish this, the configuration created by Vivado HLS should be downloaded to a physical FPGA. In the case of this diploma thesis, the *ZYNQ-7000 AP SoC ZC702 Evaluation Kit* was used.

A. FPGA Programming

According to the Vivado design flow [30], after ensuring functionality in the simulation environment (C/RTL Co-Simulation option), the design should be exported as an *IP* (*Intellectual Property*) package. The IP-CPU communication is carried out using the *AXI* (*Advanced eXtensible Interface*) protocol. Using [31] as a guide and applying the appropriate directives, the design can now be accessed via the AXI-lite interface, which is a subset of the AXI protocol.

In order to minimize the overhead of data exchange between the IP and the processor, the output of the network was reduced from 10 elements (see Fig. 5) to only 1, which represents the best (TOP-1) class. This change is made by utilizing a simple algorithm for finding the maximum value between 10. Because of the small number of elements, this modification introduces little to no overhead in terms of hardware and delay.

The IP can now be imported to Vivado, which offers a block-level design environment. Following the guidelines of [32], the complete system designed is shown in Fig. 20. The blocks involved are the following:

1. *ZYNQ7 Processing System*: The CPU of the SoC. The configuration of this block was based on the default settings of the ZC702 Evaluation Kit.
2. *ActivateNetwork*: The LeNet-5 block with its AXI-lite interface.
3. *AXI Interconnect*: The block enabling the CPU-IP communication. It is configured to have a single master and a single slave port.
4. *Processor System Reset*: This block provides a mechanism for controlling the reset signals of the system.

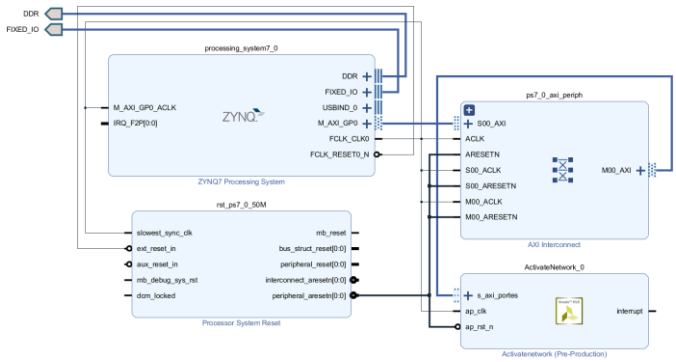

Fig. 20. The block diagram of the complete system. The system includes both the IP and CPU configurations as well as the CPU-IP AXI interconnection.

The last step is to write a software application that the CPU will execute. When the application needs to recognize a handwritten digit, this image will be fed to the FPGA via the AXI-lite interface and the corresponding output will be gathered in the same manner.

Using the Xilinx SDK tool, the software drivers of the IP were generated. These are essentially an API for performing various operations on the custom IP. Initially, the system was tested manually using a small number of images and appeared to be fully operational. However, an automated test was later developed, which is analyzed in the following paragraph.

B. Accuracy Calculation

The purpose of this test is to feed the network with multiple images and gather the corresponding outputs in order to calculate the practical accuracy of the neural net. The images were drawn from the test set, as this provides a stricter measure of accuracy. However, the 10,000 32x32 8-bit grayscale test images require a total space of approximately 10 MB, which exceeds the 256 KB RAM of the ARM Cortex-A9 CPU [33]. Therefore, the images could not be stored all at once.

As a workaround, a handshake-based protocol was designed for data exchange between a personal workstation (e.g. a PC) and the development board via asynchronous serial communication. This protocol is shown in Fig. 21 and described as follows:

Step 1. Personal Workstation (PW) transmits a *SOH* character to the Development Board (DB) which signals the start of the session. Then, for every image steps 2 to 6 are executed.

Step 2. PW transmits a *STX* character signaling the start of an image.

Step 3. PW transmits the label of the image (0-9) and a *CR* character.

Step 4. PW sends the image pixel by pixel using *CR* as a delimiter.

Step 5. PW transmits an *ETX* character signaling the end of the image.

Step 6. DB responds with either *ACK* or *NACK* depending on whether the output of the network matches the true label transmitted in Step 3.

Step 7. After receiving the response to the last image, PW transmits an *EOT* character, terminating the session.

It is evident that both PW and DB should adhere to the rules of the aforementioned protocol. On the DB side, the UART API provided by the Xilinx SDK tool was utilized. On the PW side, a Python script was developed which accesses the images stored as files and transmits them via a serial port. These scripts along with the LeNet-5 IP can be found at “9.Zynq” of [12].

The accuracy is simply calculated by counting the number of ACKs received and dividing by the number of images transmitted. Running the automated test for the first 1,000 images of the test set, a TOP-1 accuracy equal to 96.1% is calculated. The important thing to note is that the practical accuracy matches the simulated accuracy, which is shown in the last row of Table VI.

Finally, it should be mentioned that the application services the handshake signals using *polling*. This method was preferred due to its simplicity and because of the *blocking* nature of the custom protocol (meaning that PW must wait for the DB’s response before continuing).

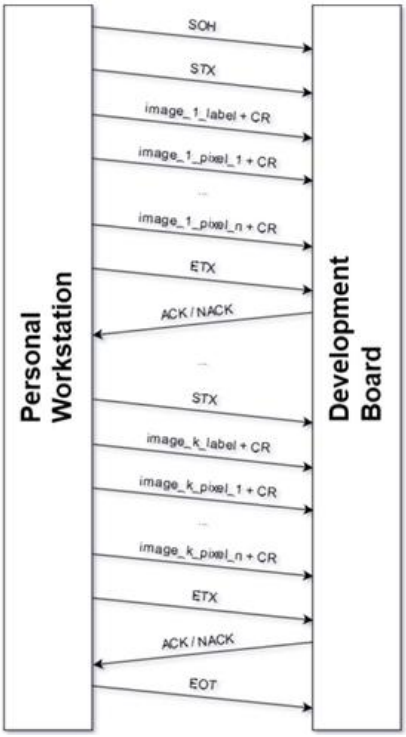

Fig. 21. The communication protocol used for transmitting the images from the personal workstation to the development board.

C. Speed Comparison

In order to calculate the speed of the on-board LeNet-5 implementation, a timer was used. The timer was configured to start ticking just before signaling the LeNet-5 IP to begin processing the input image and stopped ticking just after the output was stabilized. After repeating this test for different images, an average of 2.5ms delay per image was calculated.

It should be noted that this value is almost three times greater than the expected of 0.8ms of the *Unroll 2* architecture (see Fig. 19). This difference is firstly attributed to the delay overhead introduced by the AXI-lite interface. However, the main reason for the above deviation is the fact that Vivado doubled the clock period from 10ns, which was the initial target, to 20ns, despite meeting the timing constraints for 10ns in the simulation environment. This practically halved the net's speed.

Finally, the practical delay achieved is compared with the theoretical delays. This comparison is shown in Fig. 22, where:

1. *Initial* is the first hardware implementation (Section IV).
2. *Network compression* is the implementation after compressing the neural net using the Ristretto method (Section V).
3. *Architecture optimization* represents the *Unroll 2* architecture after performing the code transformations mentioned in Section VI.
4. *Practical* is the physical (on-board) implementation.
5. *Software* represents the initial C/C++ software implementation executed on a regular CPU.

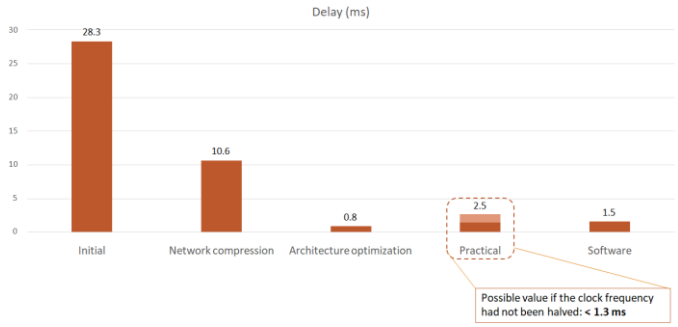

Fig. 22. Comparison of the delays achieved by the implementations of this diploma thesis.

It is evident that the practical implementation is slower than the software one. As mentioned earlier, though, the theoretical results suggest that the LeNet-5 per image delay can be almost halved (*0.8*ms).

VIII. Conclusion

In this diploma thesis a framework for developing the neural network LeNet-5 in hardware accelerators has been designed and tested. The workflow of this framework is summarized as follows:

1. Define the desired functionality of the system.
2. Select a neural net (LeNet-5) and adjust its parameters to the desired specifications.
3. Implement and train the neural net in software.
4. Describe the model using a low-level programming language and translate the neural net into hardware description language.
5. Compress the initial implementation, while thoroughly investigating various quantization methods.
6. Optimize the algorithm and the architecture of the implementation, while comparing different designs.
7. Physically implement one of the previous architectures on an FPGA board and design a fully functional embedded system.

It is evident that this workflow is general and, thus, the proposed framework can be easily modified to work with other neural networks, as well. Besides, extendibility was one of the initial goals of this thesis.

As for the tools used, an important observation is the ease that Vivado HLS offers in terms of translating software code into hardware description languages. Moreover, both the configurability and the testability provided by the tool enable fast experimentation with different design approaches. Last but not least, the simulation estimates proved to be adequately accurate compared to the practical data gathered from the on-board implementation.

Apart from implementing a neural network on a development board, an important aspect of this work is optimizing the design in terms of delay and efficient hardware usage. Indeed, starting from the initial 32-bit implementation, with a delay of ~30ms, a plethora of arithmetic, structural and code optimizations were carried out resulting in a much faster

design of ~1ms delay. Thus, an acceleration of approximately 30 times was achieved.

Finally, as for future work, it is of interest to extend the proposed framework in order to develop various neural networks.

REFERENCES

[1] C. M. Bishop, "Pattern Recognition and Machine Learning,"2006.
[2] R. Dass, "Pattern Recognition Techniques: A Review,"2018.
[3] S. Thedoridis. K.Koutroumbas, "Pattern Recognition," 2008.
[4] S. Ellie, "An overview of Pattern Recognition," 2013.
[5] A. V. Oppenheim, A. S. Willsky, S. H. Nawab, "Signals and Systems," 1997.
[6] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, "Gradient-Based Learning Applied to Document Recognition," 1998.
[7] R. Caruana, S. Lawrence, L. Giles, "Overfitting in Neural Nets: Backpropagation, Conjugate Gradient, and Early Stopping," 2001.
[8] "The MNIST Database of Handwritten Digits," [Online]. Available: http://yann.lecun.com/exdb/mnist/.
[9] A. F. M. Agarap, "Deep Learning using Rectified Linear Units (ReLU)," 2019.
[10] Google, "Multi-Class Neural Networks: Softmax," [Online]. Available: https://developers.google.com/machine-learning/crash-course/multi-class-neural-networks/softmax.
[11] B. Gao, L. Pavel, "On the Properties of the Softmax Function with Application in Game Theory and Reinforcement Learning," 2017.
[12] G. Evangelou, N. Roussos, "Thesis GitHub Repository," 2020, [Online]. Available: https://github.com/georg-e-v/A-framework-for-developing-Neural-Networks-in-hardware-accelerators.
[13] M. Gazar, "MNIST Keras Example," 2018, [Online]. Available: https://colab.research.google.com/drive/1CVm50PGE4vhtB5I_a_yc4h5F-itKOVL9.
[14] D. P. Kingma, J. L. Ba, "Adam: A Method for Stochastic Optimization," Presented at ICLR, 2015.
[15] Xilinx, "Vivado Design Suite Tutorial High-Level Synthesis," 2019, [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug871-vivado-high-level-synthesis-tutorial.pdf.
[16] Y. Cheng, D. Wang, P. Zhou, T. Zhang, "A Survey of Model Compression and Acceleration for Deep Neural Networks," 2020.
[17] S. Gupta, A. Agrawal, K. Gopalakrishnan, P. Narayanan, "Deep Learning with Limited Numerical Precision," 2015.
[18] P. Judd, J. Albericio, T. Hetherington, T. Aamodt, N. E. Jerger, R. Urtasun, A. Moshovos, "Reduced-Precision Strategies for Bounded Memory in Deep Neural Nets," 2016.
[19] P. Gysel, J. Pimentel, M. Motamedi, S. Ghiasi, "Ristretto: A Framework for Empirical Study of Resource-Efficient Inference in Convolutional Neural Networks," 2018.
[20] S. Anwar, K. Hwang, W. Sung, "Fixed Point Optimization of Deep Convolutional Neural Networks for Object Recognition," 2015.
[21] P. Gysel, "Ristretto, CNN Approximation," [Online]. Available: http://lepsucd.com/ristretto-cnn-approximation.
[22] BAIR, "Caffe: Deep Learning Framework," [Online]. Available: https://caffe.berkeleyvision.org.
[23] P. Gysel, "Ristretto: Hardware-Oriented Approximation of Convolutional Neural Networks," M.Sc. thesis, Dept. Elect. and Comp. Eng., Univ. of California, Davis, 2016.
[24] BAIR, "Training LeNet on MNIST with Caffe," [Online] Available: https://caffe.berkeleyvision.org/gathered/examples/mnist.html.
[25] Xilinx, "Introduction to FPGA Design with Vivado High-Level Synthesis," 2019, [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/ug998-vivado-intro-fpga-design-hls.pdf.
[26] D. F. Bacon, S. L. Graham, O. J. Sharp, "Compiler Transformations for High-Performance Computing," 1994.
[27] D. Patterson, J. Hennessy, "Computer Organization and Design," 2010.
[28] Xilinx, "Vivado Design Suite User Guide: High-Level Synthesis," 2019, [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug902-vivado-high-level-synthesis.pdf.
[29] Xilinx, "SDAccel Pragma Reference Guide," 2018, [Online]. Available: https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/igd1504034366745.html.
[30] Xilinx, "Vivado Design Suite User Guide: Design Flows Overview," 2018, [Online]. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug892-vivado-design-flows-overview.pdf.
[31] Xilinx, "Vivado Design Suite: AXI Reference Guide," 2017, [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf.
[32] Xilinx, "Zynq-7000 All Programmable SoC: Embedded Design Tutorial," 2019, [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_1/ug1165-zynq-embedded-design-tutorial.pdf.
[33] Xilinx, "Zynq-7000 SoC Data Sheet: Overview," 2018, [Online]. Available: https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf.