UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Implementation and acceleration of a binary neural net on a reconfigurable platform

Diploma Thesis

Styliani Anastasiadou

Supervisor: Georgios Stamoulis

February 2023

UNIVERSITY OF THESSALY

SCHOOL OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Implementation and acceleration of a binary neural net on a reconfigurable platform

Diploma Thesis

Styliani Anastasiadou

Supervisor: Georgios Stamoulis

February 2023

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ

ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Υλοποίηση και επιτάχυνση ενός δυαδικό νευρωνικού δικτύου σε επαναπρογραμματιζόμενη πλατφόρμα ολοκληρωμένων κυκλωμάτων

Διπλωματική Εργασία

Στυλιανή Αναστασιάδου

Επιβλέπων: Γεώργιος Σταμούλης

Φεβρουάριος 2023

Approved by the Examination Committee:

Supervisor	**Georgios Stamoulis** Professor, Department of Electrical and Computer Engineering, University of Thessaly
Member	**Nestor Evmorfopoulos** Associate Professor, Department of Electrical and Computer Engineering, University of Thessaly
Member	**Konstantinos Kolomvatsos** Assistant Professor, Department of Computer Science, University of Thessaly

Acknowledgements

I would like to thank my family and friends for their unwavering support and encouragement.

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

Abstract

The purpose of this thesis is to provide a detailed description of the implementation and acceleration of a Binary Neural Network (BNN) on a Field-Programmable Gate Array (FPGA). BNNs are extensively used due to their robustness regarding both computation and memory performance, but their implementation on CPUs can be quite difficult due to the large number of arithmetic operations involved. FPGAs demonstrate positive prospects for platforms that are used for accelerating BNNs, as they can be customized to perform specialized computations quickly and effectively. In our implementation, we are using the Xilinx Vivado design suite and sds++ compiler in an attempt to reduce the power consumption of the system compared to similar implementations. We describe the steps that followed for the design of our BNN, and the optimization techniques we used, including pipelining and loop unrolling. Our results demonstrate the effectiveness of FPGAs for accelerating BNNs and suggest that they have the potential to significantly improve the efficiency and performance of deep learning applications.

Keywords:

Binarized Neural Networks, Field-Programmable Gate Array, Efficiency, Acceleration

Περίληψη

Σκοπός της παρούσας διπλωματικής εργασίας είναι να παρέχει μια λεπτομερή περιγραφή της υλοποίησης και της επιτάχυνσης ενός Δυαδικού Νευρωνικού Δικτύου (BNN) σε μια Επαναπρογραμματιζόμενη Πλατφόρμα Ολοκληρωμένων Κυκλωμάτων (FPGA). Τα BNN χρησιμοποιούνται εκτενώς λόγω της αποδοτικότητας τους όσον αφορά την υπολογιστική ισχύ και την κατανάλωση της μνήμης. Ωστόσο, η εφαρμογή τους σε CPU μπορεί να είναι αρκετά δύσκολη λόγω του μεγάλου όγκου αριθμητικών πράξεων που εμπλέκονται. Οι FPGA εμφανίζουν μεγαλύτερη συμβατότητα με μοντέλα που χρησιμοποιούνται για την επιτάχυνση BNN, καθώς μπορούν να προσαρμοστούν με ευελιξία, για να εκτελούν εξειδικευμένους υπολογισμούς γρήγορα και αποτελεσματικά. Στην συγκεκριμένη υλοποίηση, χρησιμοποιούμε το Xilinx Vivado για την σύνθεση του BNN και τον μεταγλωττιστή sds++ σε μια προσπάθεια να μειώσουμε την κατανάλωση ενέργειας του συστήματος σε σύγκριση με παρόμοιες υλοποιήσεις. Περιγράφουμε τα βήματα που ακολουθήσαμε για το σχεδιασμό του BNN μας και τις τεχνικές βελτιστοποίησης που χρησιμοποιήσαμε. Τέλος, παρουσιάζουμε τα συμπεράσματά μας, τα οποία καταδεικνύουν την αποτελεσματικότητα των FPGA για την επιτάχυνση των BNN και την σημασία των μεθόδων βελτιστοποίησης που εφαρμόστηκαν, καταλήγοντας στην βέλτιστη εκδοχή.

Table of contents

List of figures

List of tables

Abbreviations

FPGA	Field Programmable Gate Array
BNN	Binary Neural Network
CNN	Convolutional Neural Network
GPU	Graphics Processing Unit
RAM	Random Access Memory
LUT	Lookup Table
HLS	High Level Synthesis
CLB	Configurable Logic Block
PIP	Programmable Interconnect Point
SDS++	Slotted Drive System Plus Plus
GUI	Graphical User Interface
IoT	Internet of Things

Chapter 1

Introduction

Artificial intelligence [1] has evolved significantly in recent years thanks to machine learning and deep learning. These advanced technologies allow computers to learn from data and patterns, eliminating the need for explicit programming. Deep learning refers to an aspect of machine learning that uses neural networks to learn multi-level representations of data. These networks have facilitated significant advances in different applications, from speech recognition to natural language processing and game playing. Deep learning techniques have surpassed conventional machine learning methods by extracting feature representations from raw data automatically, without the need for human experts to manually engineer features.

In terms of neural networks, one of the most popular types is the Convolutional Neural Network (CNN), which is commonly used for the identification and classification of images. Binary Neural Network (BNN) is another type of neural network that has received considerable attention over the past few years for its ability to provide accurate estimations with lower power and memory consumption than a CNN, making it particularly useful for applications with constrained resources.

1.1 Background

In various implementations of Convolutional Neural Networks (CNNs), GPUs are used to achieve high accuracy. However, FPGA-based implementations have, increasingly, received high attention from researchers, since they are more efficient and consume less energy. Despite the high accuracy, CNNs need complex models and high-power consumption for their calculations in comparison to Binarized Neural Networks (BNNs), which have shown great potential in certain applications, in terms of accuracy, compared to CNNs. As a result, BNNs can reduce the bit-widths from 32-bit to a single-bit and give high performance. A further advantage is that they are perfectly combined with FPGAs, allowing the most efficient implementation to be achieved.

1.2 CNN

Convolutional Neural Networks (CNNs) [2] constitute a form of artificial neural networks that are considered to be very useful for a vast range of image and video recognition tasks, and as result, they are widely adopted in both industry and research. They use several different layers, with the most important being the convolutional layers, the pooling layers, and the fully connected layers Fig. 1.1. The first ones are responsible for extracting features from the input image, while the second ones are used to reduce the dimensionality of the feature maps. At last, the fully connected layers are applied to identify the input image dependent on the data extracted by the convolutional and pooling layers.

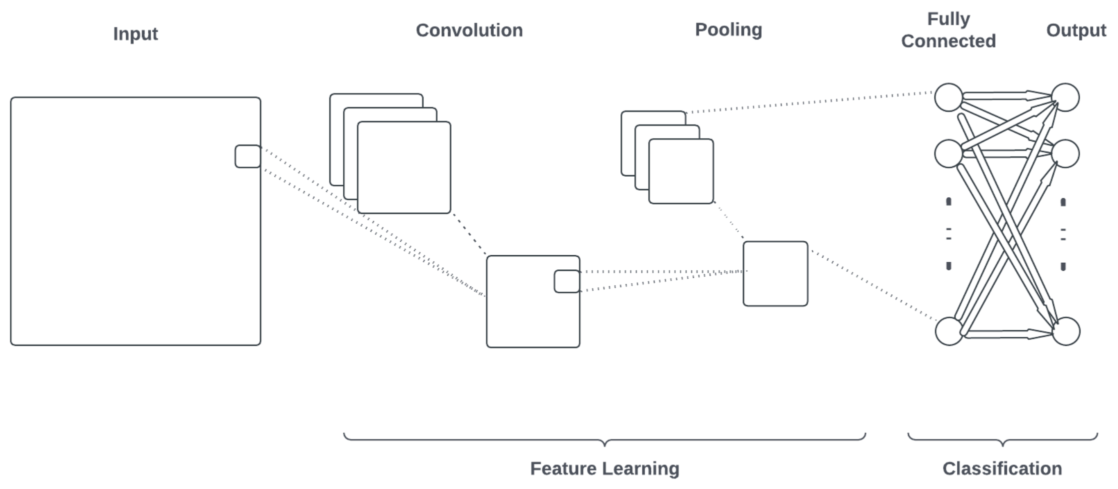

Figure 1.1: Example of a simple CNN

1.3 BNN

Binarized Neural Networks (BNNs) are similar to Convolutional Neural Networks (CNNs), however, the values of their weights and activation are binary, rather than continuous values fig. 1.2. This difference is vital for reducing the computational complexity and the model size of the network. As shown in many scientific approaches [3], despite their simplicity, BNNs can provide high accuracy and efficiency with less power consumption than a CNN which is a requirement for a system with limited power capability and memory space, like an embedded system or an edge device.

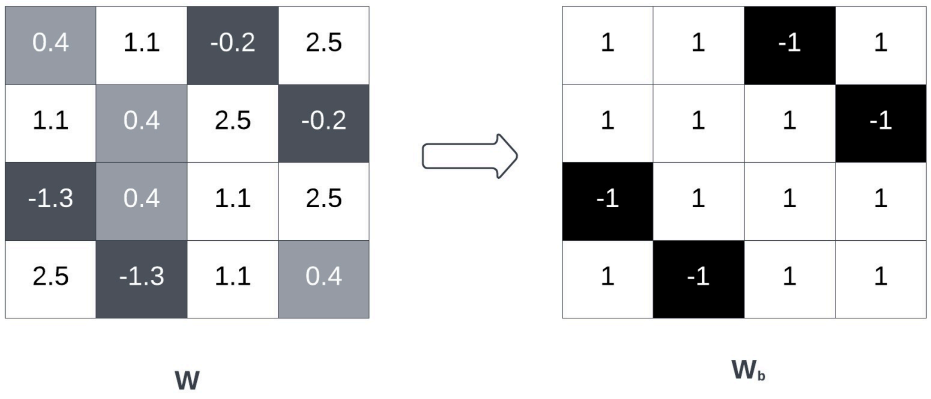

Figure 1.2: W_b = sign(W)

BNNs are well suited to be implemented on commercial FPGAs boards, since they can be easily optimized to perform specific computations with minimal power consumption. The flexibility of an FPGA seems to be useful for optimizations of BNNs, because it allows the developer to change the hardware topology according to the network's design, which could reach or even exceed the accuracy of more complex models, such as a CNN running on a GPU.

1.4 FPGA

Field Programmable Gate Arrays (FPGAs) are reconfigurable silicon devices with the ability to be converted into various integrated circuits from simple ones, such as logic gates, to more complicated systems. They consist of configurable logic blocks (CLBs), in an array, and programmable interconnect points (PIPs), as well as memory blocks that are used for Look Up Tables (LUTs), RAM, or even simple Flip Flops and can be configured by the user to produce the circuit that is suitable for their needs. Generally, a high-level synthesis (HLS) language will be used to determine the desired configuration of the blocks.

What is most important about FPGAs is their flexibility to be configured to run simultaneously different programs that require different hardware designs which can be reconfigured easily at low cost. Additionally, the parallelism of the processes running provides efficiency and acceleration. Finally, FPGAs are suitable for simple circuits because they consume less power than an ASIC. Considering all these factors, FPGAs are an attractive and efficient solution for low-volume production.

1.5 Vivado

Xilinx Vivado [4] is a comprehensive software suite for designing and programming FPGAs and other programmable logic devices. It provides a wide range of tools and features for all stages of the FPGA design process, from initial design creation to testing and debugging, as well as, a GUI for designing circuits and a suite of tools for simulating and testing the design. It also includes a synthesis tool that generates optimized gate-level netlists from high-level RTL designs and a place-and-route tool that maps the netlist onto the target FPGA and optimizes the routing of signals. Vivado also provides advanced debugging and analysis tools, including waveform viewers and performance analyzers, to help identify and solve design issues. Finally, Vivado supports a range of programming languages and design flows, making it a versatile and flexible tool for FPGA design.

1.6 SDS++

SDS++ is a toolkit developed by Xilinx that is included with the SDx development environment [5] and is intended for the programming of FPGA-accelerated applications. It is a C/C++ compiler that generates FPGA-optimized code directly from high-level designs, allowing software developers to take advantage of the FPGA acceleration capabilities without having to write low-level hardware description language (HDL) code. SDS++ supports a wide range of FPGA devices and includes a set of libraries and APIs for delegating processing efforts to the FPGA. It also includes tools for profiling and analyzing the performance of the accelerated application, enabling developers to optimize their code for the target FPGA. SDS++ is integrated with the Vivado design suite, allowing seamless design and verification of the FPGA-based acceleration. Overall, SDS++ provides an easy-to-use and efficient way to create FPGA-accelerated applications, enabling software developers to take advantage of the power and flexibility of FPGAs without requiring expertise in hardware design.

1.7 Contribution

This thesis tackles the problem of high power and memory consumption by implementing and accelerating a BNN on a Xilinx FPGA. The acceleration has been achieved by using optimization techniques such as pipelining and loop unrolling, which have been widely used to enhance the performance of digital circuits. These techniques enable the design to process more data in less time, thus reducing power and memory requirements. Also, a different number of convolvers were examined. Additionally, the study examines the impact of varying the number of convolvers in the BNN. The results of the experiments are presented and discussed in the following chapters, shedding light on the trade-offs between computational complexity and accuracy. This project contributes to the development of more efficient and effective hardware architectures for BNNs, with potential applications in various fields, like computer vision, robotics, and IoT.

1.8 Structure of thesis

The rest of the thesis is organized as follows. In Chapter 2, related approaches and solutions will be reported. In Chapter 3, the implementation of the BNN used for this thesis will be analyzed. In Chapter 4, the acceleration techniques that have been applied to the BNN will be presented. In Chapter 5 the results of the optimization experiments will be explained and finally, in Chapter 6 ideas for future work will be discussed.

Chapter 2

Related Work

As already noted, Binary Neural Networks (BNNs) are neural networks that replace floating-point weights and activations with binary values, resulting in memory conservation and complexity minimization of computational processes. They have gained popularity in recent years due to their ability to reduce memory requirements and computational complexity compared to traditional neural networks. Implementing BNNs in FPGAs can lead to further benefits such as improved power efficiency and reduced latency. Several studies [6, 7, 8] have produced encouraging findings in manifold applications, to name a few, image classification and natural language processing, while retaining minimal accuracy loss.

”FINN: A Framework for Fast, Scalable Binarized Neural Network Inference” by Umuroglu et al. This work [9] presents a framework for implementing BNNs on FPGAs using the Xilinx Vivado design suite. The authors use the ZC706 FPGA platform and perform experiments on image classification tasks.

”A GPU-Outperforming FPGA Accelerator Architecture for Binary Convolutional Neural Networks” by Li et al. In this work [10], the authors propose a hardware-friendly binary convolutional neural network architecture and implement it on a Virtex-7 FPGA. They demonstrate the effectiveness of their approach according to throughput and energy efficiency.

There is no doubt that the implementation and acceleration of BNNs in FPGAs have received considerable interest, primarily because of their potential advantages, reduced power consumption and enhanced performance. Thus, it is indispensable to examine the cutting-edge developments of BNNs on FPGAs to identify the limitations and challenges in this field and investigate plausible solutions to counteract them.

The research in this field has been extensive. However, prior studies have left some gaps that deserve attention. First of all, although BNNs are computationally less complex than traditional neural networks, their implementation on FPGAs still poses challenges owing to the restricted resources available on FPGAs, namely, on-chip memory and programmable I/O blocks. Additionally, they have been found to exhibit lower levels of accuracy [7] compared to classic neural networks, which is a primary bottleneck in certain applications. On the other hand, FPGAs are highly customizable, but the customization process is complex [11] and time-consuming. Finally, BNNs on FPGAs have been used for various applications, but their performance and accuracy under real-world scenarios still need to be evaluated and optimized. As a result, we need to conduct further research to implement BNNs on FPGA in a more cost-effective manner and to improve the accuracy of BNNs without eliminating their numerous advantages while attempting to make the customization process more efficient and user-friendly.

Chapter 3

Implementation of BNN

This chapter provides an overview of the required steps for a BNN implementation and the steps followed in our own BNN implementation.

The design of the architecture in which the number of layers of the network and the number of neurons in each layer is determined. The activation function for the network is also an important consideration, established in the beginning. The next step is about quantizing the weights and activations from their original floating-point values. Afterward, the BNN is ready to be mapped to the FPGA which means that the digital circuit for the network's implementation has to be created. Following that, the digital circuits have to be synthesized to generate the digital logic that will be implemented on the FPGA. Once the digital circuits have been created and synthesized, the FPGA will be configured according to the digital logic which has been produced in the previous step. The last three actions can be done using an HLS tool. The final step concerns the optimization of the BNN.

For this thesis, a BNN has been implemented using C++ and Vivado has been used as the HLS tool [12] for the generation of the RTL description of the network. Below there are, in detail, the steps that were followed for this model.

3.1 Designing the BNN architecture

As it was mentioned above, this stage identifies the number of layers, the number of neurons in each layer, and the type of activation function to be used. Furthermore, during the design phase, it is necessary to balance the complexity of the architecture with the required accuracy and performance. The main criteria for this design were efficiency as well as low power and memory consumption, in order to have a suitable model to run on the FPGA.

In our BNN, the number of layers and neurons has been chosen based on the simplicity of the project, thus a small number of layers are adequate. For more complex tasks a larger number of layers is required. Likewise, the number of neurons in each layer has a significant impact since it affects the computational operations required for forward and reverse passes. A large number of neurons in each layer will result in a highly complicated BNN, on the other hand, it may also increase the accuracy of the model.

The activation function that has been used is the sign function fig. 3.1 due to its simplicity and efficiency. It is the most popular and well-suited activation function for devices with limited power and memory capabilities. In particular, only a single bit is required to represent the activations, as opposed to multiple bits required by other activation functions. As a result, XNOR and bit-count operations have been used to perform the binary convolutions and pooling required for the BNN. A lookup table-based approach has been also used to implement the binary sign function required for the hidden layers. A further advantage of the sign function is its simplicity, which relies on a unique comparison operation that allows for precise quantization of activations and weights. Consequently, the network becomes more computationally efficient, since the conversion from floating-point to binary representations is straightforward and it, also, has a reduced memory footprint.

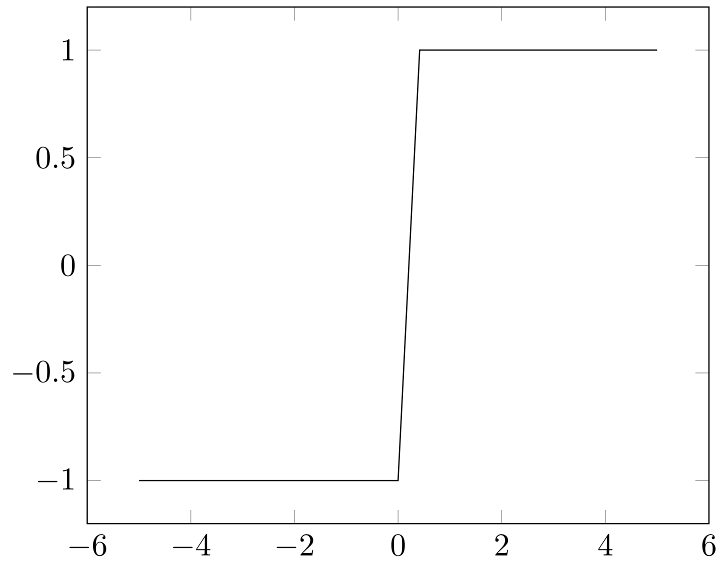

Figure 3.1: sign function

3.2 Quantizing the weights and activations

The quantization process is converting the values of weights and activations of a traditional neural network, from real numbers into binary values, most commonly +1 and -1. Initially, a threshold value needs to be selected above which the weights and activations are set to +1, and below which they are set to -1. In our case, this value is the ’zero value’ but it is possible to adjust the level of thresholding to optimize the ratio between the accuracy and performance of the BNN. Secondly, by comparing the weights and activations of the BNN with the threshold value, they are quantized to give the +1 or -1 status. The quantized weights and activations are therefore stored in memory and are immediately available to be used in the forward and backward passes of the BNN. This step can result in a loss of precision, which can affect the accuracy of the BNN. Therefore, it’s crucial to carefully select the threshold value and monitor the accuracy of the BNN after quantization. For our implementation, the post-training quantization technique was used to mitigate the accuracy loss.

3.3 Mapping the BNN to FPGA

As part of the mapping process, it is necessary to determine which specific digital circuits will be used for the FPGA implementation of the BNN. For example, the BNN’s computation functions, such as convolution, pooling, and fully connected layers, need to be implemented using appropriate digital building blocks like adders, multipliers, and registers. Furthermore, to store the weights and activations of the BNN, the memory architecture must be determined, for example, on-chip or external memory. Among these considerations are the required memory size, as well as the memory organization, such as the number of banks, the number of ports, and the method of accessing the memory. It is also necessary to determine the interconnect structure, such as routing resources, in order to connect the various digital building blocks and memory components. There are several factors to consider, including determining the number of interconnects, the topology of the interconnects, and the routing resources, such as switch boxes and routing tracks.

Mapping a BNN to an FPGA involves determining the specific digital circuits and memory architecture that will be used to implement the BNN, and determining the interconnect structure that will connect the various digital building blocks and memory components. This process is crucial to ensure that the BNN can be implemented on the FPGA efficiently and that the performance, power consumption, and area of the BNN implementation meet the requirements of the specific application.

3.4 Synthesis & Implementation

The synthesizing of the design has been created by Vivado HLS as well as the implementation of the BNN on the FPGA. At the beginning of this step, we chose the suitable hardware architecture that could provide us with all the resources needed for our model. Then, we inserted the Verilog code, which has been produced during the build process of our implementation, into our Vivado project. Finally, the tool could begin the synthesis of the code into a netlist which had to be optimized to reduce the number of logic elements and maximize the clock frequency. The optimized netlist was then mapped onto the target FPGA, which involves assigning the netlist to physical resources, such as lookup tables and flip-flops, and generating the routing connections. During this stage, the design had to meet the timing and resource constraints of the FPGA.

Chapter 4

Acceleration of BNN

In the following chapter, we will present the acceleration methodologies employed in our BNN. In addition to the binary weights and activations of our network, that were already reduced the hardware resources required, as we used an FPGA, we were able to parallelize the computations which allowed multiple calculations to be performed simultaneously. This greatly reduced the overall processing time and it was especially beneficial for further increasing efficiency in implementation. Overall, the acceleration provided significant performance improvements which are going to be presented in detail in Chapter 5.

4.1 Convolvers

Optimization of a BNN can be accomplished by adjusting the number of convolutional layers and the parameters within each layer, including the size of the filters, the stride, and the padding. The effectiveness of the convolutional layers depends on the complexity of the input data and the nature of the problem being solved. It is possible to capture more intricate patterns in the input data by employing a greater number of convolutional layers. The downside of this is that it also increases the computational cost and the likelihood of overfitting the data. Overfitting occurs when a model becomes too complex and starts to memorize the training data instead of learning the general patterns, which can lead to poor performance on unseen data. On the other hand, if the model has fewer convolutional layers than needed to capture complex patterns on the data, it may also have a limited ability to capture those patterns.

Hence, in order to choose the optimal number of layers for a BNN, the complexity of the model should be balanced against the computational burden. To determine the best number of convolutional layers, we experimented with different numbers and evaluated their performance. We observed that the number of convolutional layers in a BNN impacts its performance and efficiency, and for the optimal number of layers, we have taken into consideration the trade-off between model complexity and computational cost. As demonstrated by the results presented in Chapter 5, the optimal approach entails the selection of four convolvers in conjunction with a series of optimization techniques.

4.2 Pipeline

The first optimization technique that was used is Pipelining. Generally, it refers to a method that allows multiple instructions to be executed simultaneously in order to increase the performance of FPGAs. As a general principle, pipelining consists of segmenting a single instruction into multiple stages, each representing one step of the execution process. The division of the instruction into stages enables the execution of a new instruction to begin while the previous instruction is still in progress. It also allows parallelizing multiple instructions since the stages can be executed concurrently.

Stages in the pipeline correspond to discrete phases in the process of executing an instruction. As part of these phases, the instruction may be fetched from memory, decoded, executed, or written back to memory. When multiple instructions are executed in parallel, as outlined in /cite[pipeline], the FPGA is capable of achieving significant performance and efficiency gains as compared to sequential execution. As pipelines are structured in such a way as to enable parallel execution of different stages of the instruction execution process, idle time is reduced and FPGA resources are maximized. Designing a pipeline that balances the number of stages and their complexity will allow performance optimization while minimizing potential bottlenecks and hazards.

In order to apply all the above to our implementation we used HLS Pragmas which directive provided by Xilinx. Specifically, we used the ”#pragma pipeline” to specify that a loop in the code should be pipelined, meaning that each iteration of the loop should be executed in parallel with the previous iteration.

Formally, the ”pragma pipeline” directive is used to specify the pipeline depth of a loop, which is the number of loop iterations that can be executed in parallel. By using this directive, the designer can control the number of pipeline stages in the design, which can have a significant impact on the performance and resource utilization of the resulting hardware. The directive provides the compiler with information about the intended behavior of the loop, allowing it to optimize the code for pipelining and improve the performance of the design.

Algorithm 1 Reseting of the Convolutional Buffer with Pipelining

```
1: Procedure Pipeline each iteration of the loop
2: for ($i = 0$; $i < WORD\_SIZE$; $i$++) do
3:     $\#pragma\ HLS\ pipeline$        //Use the HLS pragma for pipelining
4:     $buffer[i][j] = 0)$;           //Reset Convolution buffer
5: end for
6: End Procedure
```

4.3 Loop Unrolling

Loop unrolling [13] is the second method that was used for acceleration, by reducing the number of iterations required to process the data. Loop unrolling is based on the concept that the loop body is reprised a fixed number of times in order to decrease loop control overhead, which involves checking loop conditions and incrementing loop indexes.

For our BNN, loop unrolling was applicable to the convolutional layers, where the filter weights are involved in overlapping patches of input data. Convolution loops were unrolled to reduce loop control statements and speed up processing. Further, loop unrolling also improved the utilization of the system resources, like the arithmetic units and memory, by allowing multiple operations to be performed in parallel. This had as a result increase in performance and latency reduction. It is important to note that loop unrolling could also increase memory usage, as the unrolled code may require more memory to store the duplicated loop bodies. and can as well lead to excessive code size and decreased performance due to increased overhead associated with instruction fetch and dispatch. Thus, we aimed to achieve optimal performance by finding the right balance between loop unrolling and code size

The procedure to apply this technique to our implementation was the same with the pipelining, We used the Xilinx pragma ”#pragma HLS unroll” [14]. As it was mentioned before, this optimization technique is used to specify that a loop should be unrolled, meaning that multiple iterations of the loop should be executed in a single pass.

Specifically, the ”#pragma HLS unroll” directive converts a loop consisting of a predetermined number of iterations into a series of equivalent code blocks, each representing one iteration of the initial loop. As a result of the directive, the compiler is provided with information regarding the calculated behavior of the loop, permitting it to optimize the code for unrolling and to enhance the performance of the design. By unrolling the loop, the compiler is able to generate optimized code that reduces the overhead of loop control instructions and improves the performance of the design.

Algorithm 2 Reseting of the Convolutional Buffer with Loop Unrolling

```
1: Procedure Unroll the iterations of the loop
2: for $(i = 0; i < WORD\_SIZE; i{+}{+})$ do
3:     $\#pragma\ HLS\ unroll$        //Use the HLS pragma for loop unrollning
4:     $buffer[i][j] = 0;$            //Reset Convolution buffer
5: end for
6: End Procedure
```

The aforementioned techniques aid in emphasizing the fundamental significance of implementing a robust and thorough approach, which combines the mentioned methodologies to ensure maximum efficacy and efficiency. This procedure is essential to achieving superior performance outcomes such as the ones that will be proposed at the next chapter.

Chapter 5

Results

In this chapter, we present the optimization results produced by Vivado HLS. To determine the optimal acceleration for our network, we conducted several experiments, always keeping in mind the theoretical background described in the previous chapters.

The FPGA used for the synthesis and implementation process was the Virtex-7, xc7vx485tffg1157-1. The choice has been made in order to fulfill the requirements for the resources that our project needed.

5.1 BNN with 2 Convolvers

In this synthesis and implementation run using Vivado, the design has used 21897 LUTs out of the available 303600. Additionally, the design has utilized 1568 of the Slice LUTs as memory, with 768 of them being distributed RAM and 800 of them being shift registers. The remaining 20329 LUTs are used as logic and the power consumption on-chip was 259.111 (W).

Table 5.1: BNN with 2 Convolvers

Slice Type	**Used**	**Available**	**Util%**
Slice LUTs*	24437	303600	8.05
LUT as Logic	22869	303600	7.53
LUT as Memory	1568	130800	1.20
LUT as Distributed RAM	768		
LUT as Shift Register	800		
Slice registers	25656	607200	4.23
Register as Flip Flop	25656	607200	4.23
Register as Latch	0	607200	0.00
F7 Muxes	2102	151800	1.24
F8 Muxes	985	75900	1.30
Total On-Chip Power = 259.111 (W)			

5.1.1 Pipeline

In this synthesis and implementation run using Vivado, the design has used 24005 LUTs out of the available 303600. Additionally, the design has utilized 1568 of the Slice LUTs as memory, with 768 of them being distributed RAM and 800 of them being shift registers. The remaining 22437 LUTs are used as logic and the power consumption on-chip was 311.614 (W).

Table 5.2: BNN with 2 Convolvers & Pipeline

Slice Type	**Used**	**Available**	**Util%**
Slice LUTs*	24005	303600	7.91
LUT as Logic	22437	303600	7.39
LUT as Memory	1568	130800	1.20
LUT as Distributed RAM	768		
LUT as Shift Register	800		
Slice registers	24761	607200	3.80
Register as Flip Flop	24761	607200	3.80
Register as Latch	0	607200	0.00
F7 Muxes	2098	151800	1.24
F8 Muxes	744	75900	1.12
Total On-Chip Power = 311.614 (W)			

5.1.2 Loop Unrolling

In this synthesis and implementation run using Vivado, the design has used 21897 LUTs out of the available 303600. Additionally, the design has utilized 1568 of the available LUTs as memory, with 768 of them being distributed RAM and 800 of them being shift registers. The remaining 20329 LUTs are used as logic and the power consumption on-chip was 281.296 (W).

Table 5.3: BNN with 2 Convolvers & Loop Unrolling

Slice Type	**Used**	**Available**	**Util%**
Slice LUTs*	21897	303600	7.21
LUT as Logic	20329	303600	6.70
LUT as Memory	1568	130800	1.20
LUT as Distributed RAM	768		
LUT as Shift Register	800		
Slice registers	23071	607200	3.80
Register as Flip Flop	23071	607200	3.80
Register as Latch	0	607200	0.00
F7 Muxes	1881	151800	1.24
F8 Muxes	850	75900	1.12
Total On-Chip Power = 281.296 (W)			

5.1.3 Pipeline & Loop Unrolling

In this synthesis and implementation run using Vivado, the design has used 24527 LUTs out of the available 303600. Additionally, the design has utilized 1568 of the Slice LUTs as memory, with 768 of them being distributed RAM and 800 of them being shift registers. The remaining 22959 LUTs are used as logic and the power consumption on-chip was 329.672 (W).

Table 5.4: BNN with 2 Convolvers & Loop Unrolling & Pipelining

Slice Type	**Used**	**Available**	**Util%**
Slice LUTs*	24527	303600	7.21
LUT as Logic	22959	303600	6.70
LUT as Memory	1568	130800	1.20
LUT as Distributed RAM	768		
LUT as Shift Register	800		
Slice registers	23071	607200	3.80
Register as Flip Flop	23071	607200	3.80
Register as Latch	0	607200	0.00
F7 Muxes	1881	151800	1.24
F8 Muxes	850	75900	1.12
Total On-Chip Power = 329.672 (W)			

5.2 BNN with 4 Convolvers

In this synthesis and implementation run using Vivado, the design has used 29678 LUTs out of the available 303600. Additionally, the design has utilized 816 of the Slice LUTs as memory, as shift registers. The remaining 28862 LUTs are used as logic and the power consumption on-chip was 441.492 (W).

Table 5.5: BNN with 4 Convolvers

Slice Type	**Used**	**Available**	**Util%**
Slice LUTs*	29678	303600	9.78
LUT as Logic	28862	303600	9.51
LUT as Memory	816	130800	0.62
LUT as Distributed RAM	0		
LUT as Shift Register	816		
Slice registers	31241	607200	5.15
Register as Flip Flop	31241	607200	5.15
Register as Latch	0	607200	0.00
F7 Muxes	2297	151800	1.51
F8 Muxes	912	75900	1.20
Total On-Chip Power = 441.492 (W)			

5.2.1 Pipeline

In this synthesis and implementation run using Vivado, the design has used 29678 LUTs out of the available 303600. Additionally, the design has utilized 816 of the Slice LUTs as memory, as shift registers. The remaining 28862 LUTs are used as logic and the power consumption on-chip was 370.496 (W).

Table 5.6: BNN with 4 Convolvers & Pipelining

Slice Type	**Used**	**Available**	**Util%**
Slice LUTs*	29317	303600	9.66
LUT as Logic	28501	303600	9.39
LUT as Memory	816	130800	0.62
LUT as Distributed RAM	0		
LUT as Shift Register	816		
Slice registers	30332	607200	5.00
Register as Flip Flop	30332	607200	5.00
Register as Latch	0	0	0.00
F7 Muxes	2163	151800	1.42
F8 Muxes	704	75900	0.93
Total On-Chip Power = 370.496 (W)			

5.2.2 Loop Unrolling

In this synthesis and implementation run using Vivado, the design has used 29678 LUTs out of the available 303600. Additionally, the design has utilized 816 of the Slice LUTs as memory, as shift registers. The remaining 28862 LUTs are used as logic and the power consumption on-chip was 221.630 (W).

Table 5.7: BNN with 4 Convolvers & Loop Unrolling

Slice Type	**Used**	**Available**	**Util%**
Slice LUTs*	27962	303600	9.21
LUT as Logic	27146	303600	8.94
LUT as Memory	816	130800	0.62
LUT as Distributed RAM	0		
LUT as Shift Register	816		
Slice registers	28749	607200	3.80
Register as Flip Flop	28749	607200	3.80
Register as Latch	0	607200	0.00
F7 Muxes	2061	151800	1.36
F8 Muxes	610	75900	0.80
Total On-Chip Power = 221.630 (W)			

5.2.3 Pipeline & Loop Unrolling

In this synthesis and implementation run using Vivado, the design has used 12346 Look-Up Tables (LUTs) out of the available 22346. Additionally, the design has utilized 882 of the available LUTs as memory, with 382 of them being distributed RAM and 500 of them being shift registers and the power consumption on-chip was 394.974 (W).

Table 5.8: BNN with 4 Convolvers & Loop Unrolling & Pipelining

Slice Type	**Used**	**Available**	**Util%**
Slice LUTs*	27307	303600	8.99
LUT as Logic	26491	303600	8.73
LUT as Memory	816	130800	0.62
LUT as Distributed RAM	0		
LUT as Shift Register	816		
Slice registers	27740	607200	4.57
Register as Flip Flop	27740	607200	4.57
Register as Latch	0	607200	0.00
F7 Muxes	2249	151800	1.48
F8 Muxes	896	75900	1.18
Total On-Chip Power = 394.974 (W)			

5.3 Summary

Based on these simulations, we have evaluated the design under a variety of configurations and settings. A different set of results was produced by each run, including the use of LUTs, memory usage, and power consumption. This analysis has enabled us to identify the most efficient implementation that offers the best performance. It was particularly evident in the implementation with four convolvers and loop unrolling that a relatively low number of LUTs was utilized, with 27962 LUTs being operated out of 303600 available. Moreover, this run met all timing constraints, while consuming the least amount of memory and power. In light of these findings, it is reasonable to conclude that this run is the most appropriate implementation of the design, giving the best balance between resource utilization, performance, and power consumption. Based on the results of this analysis, future iterations and improvements to the design can be guided.

Chapter 6

Conclusion & Future Work

We have presented the implementation and acceleration of a BNN on an FPGA. The implementation can be customized in order to test different optimization techniques. An HLS tool is used for bitstream generation and simulation of the model. The system can be trained with a number of images that the user defines and then it can be flashed on a real FPGA.

According to the results, pipelining, and loop unrolling has been essential for the acceleration of the BNN. Both techniques have reduced the number of LUTs used and improved the results.

The present implementation can be extended to be evaluated in a wide range of real-world applications. Some examples of the fields that can be applied are computer vision and robotics where the combination of high accuracy and low power consumption is vital. In more detail, BNNs can be used for edge devices like drones and surveillance cameras to perform object or face recognition and semantic segmentation, as well as, for robots for tasks such as object grasping and human-robot interaction. Another important application is in Healthcare, where BNNs can be utilized for wearable devices and medical equipment allowing heart rate monitoring and sleep analysis. Last but not least, in the Automotive industry BNNs can be applied in autonomous vehicles for lane detection, obstacle avoidance, and traffic sign recognition.

Bibliography

[1] Karan Aggarwal, Maad M. Mijwil, Sonia, Abdel-Hameed Al-Mistarehi, Safwan Alomari, Murat Gök, Anas M. Zein Alaabdin, and Safaa H. Abdulrhman. Has the future started? the current growth of artificial intelligence, machine learning, and deep learning. *Iraqi Journal For Computer Science and Mathematics*, 3(1):115–123, Jan. 2022.

[2] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. Understanding of a convolutional neural network. In *2017 International Conference on Engineering and Technology (ICET)*, pages 1–6, 2017.

[3] Maoyang Xiang and Tee Hui Teo. Implementation of binarized neural networks in all-programmable system-on-chip platforms. Feb. 2022.

[4] Vivado high-level synthesis user guide 2022.2. `https://docs.xilinx.com/viewer/book-attachment/NsrqATHzUj6if4Toia~ORQ/eysSTISAO7ZIMF3n0HIRrQ`. Last accessed: 20-02-2023.

[5] Sdsoc environment user guide. `https://www.xilinx.com/support/documents/sw_manuals/xilinx2019_1/ug1027-sdsoc-user-guide.pdf`. Last accessed: 20-02-2023.

[6] Wenyu Zhao, Teli Ma, Xuan Gong, Baochang Zhang, and David Doermann. A review of recent advances of binary neural networks for edge computing. *IEEE Journal on Miniaturization for Air and Space Systems*, 2(1):25–35, Mar. 2021.

[7] Haotong Qin, Ruihao Gong, Xianglong Liu, Xiao Bai, Jingkuan Song, and Nicu Sebe. Binary neural networks: A survey. *Pattern Recognition*, 105:107 – 281, Sep. 2020.

[8] Qingliang Liu, Jinmei Lai, and Jiabao Gao. An efficient channel-aware sparse binarized neural networks inference accelerator. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 69(3):1637 – 1641, Oct. 2022.

[9] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. Finn: A framework for fast, scalable binarized neural network inference. page 65–74, Feb. 2017.

[10] Yixing Li, Zichuan Liu, Kai Xu, Hao Yu, and Fengbo Ren. A gpu-outperforming fpga accelerator architecture for binary convolutional neural networks. 14(2), Jul. 2018.

[11] Hongwu Peng, Shanglin Zhou, Scott Weitze, Jiaxin Li, Sahidul Islam, Tong Geng, Ang Li, Wei Zhang, Minghu Song, Mimi Xie, Hang Liu, and Caiwen Ding. Binary complex neural network acceleration on fpga : (invited paper). In *2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 85–92, Jul 2021.

[12] Vivado high-level synthesis Tutorial. `https://docs.xilinx.com/v/u/2014.2-English/ug871-vivado-high-level-synthesis-tutorial`. Last accessed: 20-02-2023.

[13] J.C. Huang and T. Leng. Generalized loop-unrolling: a method for program speedup. In *Proceedings 1999 IEEE Symposium on Application-Specific Systems and Software Engineering and Technology. ASSET'99 (Cat. No.PR00122)*, pages 244–248, 1999.

[14] Vitis pragmas. `https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/HLS-Pragmas`. Last accessed: 20-02-2023.