# Design of a Convolution Operation Accelerator for CNNs Using an FPGA

Capstone Report

Olzhas Nurman

Nazarbayev University
Department of Electrical and Computer Engineering
School of Engineering and Digital Sciences

This project report was created on TexStudio editing platform using LaTeX. All the figures were drawn using draw.io online software tool.

NAZARBAYEV UNIVERSITY

**Electrical and Computer Engineering**
Nazarbayev University
http://www.nu.edu.kz

**Title:**
Design of a Convolution Operation Accelerator for CNNs Using an FPGA

**Theme:**
Capstone Project Report

**Project Period:**
Spring 2025

**Participant(s):**
Olzhas Nurman

**Supervisor(s):**
Akhan Almagambetov

**Copies:** 1

**Page Numbers:** 28

**Date of Completion:**
April 24, 2025

**Abstract:**

Training convolutional neural networks (CNNs) demands significant computational resources and specialized hardware. Custom accelerators have emerged as an efficient alternative to traditional graphics processing units (GPUs), offering higher throughput, and decreased physical footprint at low costs. In this paper, we present a bit-serial multiplier based architecture for convolution acceleration. This design offers a hardware accelerator in a DE1-SoC board that utilizes an FPGA and ARM processor on the board. The main processing unit in the design is a novel bit-serial multiplier architecture that minimizes the hardware usage and overall area on the board.

# Contents

# Preface

This project was completed as part of my Capstone Project at Nazarbayev University in the Electrical and Computer Engineering program. The goal was to design and test a hardware accelerator for convolution operations using an FPGA.

During the project, I learned a lot about both hardware and software — and how they interact and cooperate to solve challenging engineering problems. There were many difficulties, especially with debugging the design, modifying the multiplier architecture, and handling communication between the ARM processor and the FPGA. But each step helped me better understand digital design and system integration.

I would like to express my sincere gratitude to my supervisor, Dr. Akhan Almagambetov, for his guidance, encouragement, and valuable insights throughout this project. I am also thankful for the support provided by the faculty and staff of the Department of Electrical and Computer Engineering. I also thank NU and the NU Library for providing access to a large database of research papers, which was essential for this work.

Nazarbayev University, April 24, 2025

Olzhas Nurman
<olzhas.nurman@nu.edu.kz>

# Chapter 1

# Introduction

Convolutional deep neural networks (DNNs) have gained increasing attention in machine learning (ML) in the past decade. Currently, it is one of the most efficient and widely used models in many fields, especially in those that involve tasks related to analyzing image data. As the number of hidden layers continues to increase, the model's accuracy improves but it results in higher computational complexity [1]. This is due to the need for an increased number of processing units that perform multiplication operations (as part of convolution and multiply-and-accumulate operations) and memory to store weights and activations in convolutional DNNs. This means that the increased accuracy of the model comes at the cost of increased hardware usage [2, 3]. To handle such workload next-generation hardware accelerators are required [3].

Because of the computationally expensive convolution operations training of a model and inference is traditionally performed on graphic processing units (GPUs). It performs computations in a highly paralleled fashion with minimal delay that assists in increasing the throughput. Another alternative is a CPU, although not as effective as GPU can be used for relatively small models. The main issue with these options is their inefficiency in terms of power usage and price. Field Programmable Gate Arrays (FPGAs) can be configured for ML training and inference, optimizing power consumption at a relatively low price [3, 4]. Also, FPGAs allow to deviate from Von-Neumann architecture [3], which means that computing-in-memory (CIM) architecture can be implemented [2]. In other words, FPGAs are highly customizable, which allows for different architecture choices to be realized in the hardware accelerator that potentially improves the performance. Performance enhancements can include improvement in area usage, power consumption, throughput, and delay. Also, they offer the possibility of implementing highly pipelined systolic array architecture, that allows parallel processing. This parallel processing option significantly improves the performance. Parallel processing is one of the things that compensates for the lack of resources in the FPGA. Another advan-

tage FPGAs offer over GPUs and CPUs is their large number of configurable logic blocks, which allow hardware designers to implement tailored designs. This allows to design of a hardware neural network accelerator to optimize the design for the particular neural network. This reduces the costs and offers the opportunity to test different hardware architectures for performance.

However, the option of using an FPGA comes with drawbacks. One of the main issues is the requirement for high memory storage, which FPGAs can not satisfy [5]. Therefore, outside memory should be used that adds additional overhead in terms of throughput and energy consumption [2]. However, different novel architectures have been suggested to optimize the memory access [6, 7, 8]. Another issue is the lack of resources, as the multiplication operation required in convolution is computationally intensive [9]. For example: assuming that the frame image loaded to the FPGA has a resolution of 640x480, represented by 16-bit data, we get approximately 300 thousand neurons in the input layer. Assuming a single kernel size of $3 \times 3$, and with a stride size of 1 there are approximately 3 million 16-bit floating point multiplications required. Even state-of-the-art FPGAs struggle to process such large amounts of data. Therefore, architectural design decisions should be made to decrease the load, but still keep the high accuracy at low latency rates.

The issue of limited resources becomes extremely acute considering the security of data. The security of data being processed in the hardware accelerator might not be important in simple image classification tasks. However with CNNs currently being used for analyzing sensitive data, that might include facial identification data [10], health records [11] the defense mechanism should be developed to ensure data security [12].

To ensure that the drawbacks described above are mitigated and have a minimal impact on the performance different design choices can be made, that can be made both on hardware and software level. One of the defining factors in the performance of hardware is the technology of manufacturing and their effects can be evaluated in advance [13]. However, since these factors can not be controlled by us we will rely on the architectural design choices. They can be considered in two categories, the ones that affect the CNN models' performance in terms of accuracy, and the ones that do not [2].

One of the widely used techniques in the acceleration of CNNs is quantization. Quantization is a method that decreases the precision of data used for training, and inference. Using half-precision data [2], or even data with bit-width of 8-4 bits [14] makes it possible to process data with FPGA resources. While it decreases the accuracy to a small amount as opposed to full precision data types, it is one of the options to consider [2, 14, 15, 16]. Also, the current trends show that Binarized Neural Networks (BNNs) is also a possible solution to decrease the load on the hardware, where weights and activations are constrained between positive

and negative 1 [17, 18, 2]. This way the multiplication operation which is an expensive operation on the hardware can be replaced by simple XNOR gates, and the memory requirement to store weights and activations can be significantly decreased. Apart from benefits in terms of limited hardware requirements, this leads to limited access to off-chip memory, which in turn contributes to high energy efficiency [19, 2]. However, BNNs bring a couple of issues as well. First of all, very high fan-in at the addition circuit for weight accumulation. Another one is the fact that LUTs in the FPGA are more capable than implementing XNOR gates [20]. As reviewed in [2], the concept of layer-wise quantization, which suggests the usage of different bit-width data for each layer, has shown promising results as well.

However, we used a different approach and went for a bit-serial multiplier-based accelerator, that would decrease the hardware footprint at the cost of a small timing delay. This architecture does not aim to compete with the GPU but offers an alternative to GPU in applications where the area is a crucial factor and additional delays can be tolerated.

This paper is organized as follows: Section 2 provides a background in convolution operation and outlines the computations involved. Section 3 introduces the architecture and discusses the performance metrics used to evaluate our model. Section 4 presents the results and examines their implications. Finally, Section 5 concludes the report and outlines potential future work.

## 1.1 Ethical and Professional Responsibilities

- **Ethical Responsibility:**
  One of the main issues to be considered is the privacy of the data being used for training a CNN model on a hardware accelerator. It is common for CNNs to be used for analysis of medical data, facial information, and other types of confidential data. Therefore, the implementation of a proper data protection system, that would encrypt the data to protect it from unauthorized access, should be considered. Another consideration to be made is the potential environmental hazards. The majority of power resources currently used for powering hardware devices are coming from unsustainable power sources. Therefore, the system should be developed in a way that would eliminate the sources of power waste. Additionally, there is a possibility that a hardware accelerator will be used to train models that use data that was acquired either illegally, or used without the original author's permission. In order to prevent such cases a proper guideline should be developed explaining user the ethical responsibilities, and this guideline should be integrated into licensing terms if the project is to be released for open-source or commercial use.

- **Informed Judgments:**
  To make sure that decisions made during the development of the hardware accelerator for CNN models on FPGA are well-informed, that is consider both technical and societal aspects, I will rely on a combination of discussions with my supervisor, and a research. Recent research papers in hardware design, AI, and FPGA technology will provide essential information into the technical aspects, ensuring that the design follows the latest industry standards and best practices. Comprehensive research will be the basis for each decision, including a detailed analysis of academic literature, industry case studies, and a comparison with existing hardware solutions to evaluate the project's technical and societal performance. The main goal of the project is to develop hardware that maximizes efficiency while considering the broader implications of its deployment across sectors. Societal and environmental impacts will be considered throughout the process, ensuring that no laws and regulations are violated. Additionally, I will prioritize sustainability by integrating energy-efficient design practices to minimize environmental impacts. Ultimately, by combining technical development with ethical considerations and sustainability, the project will provide a solution that not only performs effectively but also responsibly addresses societal concerns and long-term implications.

- **Global Context:**
  In the global context, the hardware accelerator designed for CNN could be

considered in two separate regions. In the context of developed and developing countries. In the developed countries where AI technologies are already widely used it could improve the AI infrastructure by offering more scalable and sustainable solutions for training AI models. FPGAs are widely known for being far more energy efficient as compared to, traditionally used for machine learning algorithm training, GPUs, and CPUs thanks to proper usage of every component in the design. Therefore, this allows them to continue to decrease their carbon footprint while continuing the improvement of their AI infrastructure. In the context of developing countries where not every sector can afford to incorporate AI solutions to improve their service, could potentially be able to do so with cost-effective FPGAs. This would help these countries improve healthcare, agriculture, and other sectors by exploiting the full potential of AI technologies. Therefore, this project would fit into the global context by providing a cost-effective, energy-minimizing solution for training widely used AI technologies.

- **Economic Impact:**
  In the short term, the hardware accelerator for CNNs implemented on an FPGA offers a cost-effective alternative to CPUs and GPUs. Most of the FPGAs are more affordable than other options, and their reconfigurability allows for them to be used for a long period. This minimizes the need for frequent replacement of hardware, resulting in potential cost savings for companies requiring hardware for machine learning model training and inference. This makes hardware accelerators based on FPGA an attractive option for companies that need to train models without significant expenses. In the long run, it is possible to substantially decrease the economic expenses thanks to the savings on the hardware. This would give an opportunity for companies to focus their budget on technological improvements and expansion. It should also be noted that cost savings associated with energy saving with the help of energy-efficient hardware could be substantial for companies with huge data centers. Also, FPGA adaptability can help cut expenses associated with frequent hardware updates as new architecture designs can be implemented on the old hardware. This way the company can keep up with the newest changes in the machine learning models, and adapt and implement new designs without substantial expenses.

- **Environmental Impact:**
  FPGAs are widely known for their energy efficiency, particularly when compared to GPUs and CPUs. With the increase in the utilization of AI models in the industry, energy consumption for such purposes has increased drastically, which in turn is increasing the carbon footprint in the environment. While hardware accelerator based on FPGA would help to create a sustainable in-

frastructure for machine learning model training and inference. Another important point to mention is the reconfigurability of FPGAs. As compared to other hardware accelerator designs, like ASIC, in case of the need for implementation of new changes, FPGA wouldn't need to be sent for manufacturing again. This results in a decrease in both electronic waste and energy waste that was used during the manufacturing of a new ASIC-based accelerator. This also makes the design future-proof, since the machine learning models are evolving every day, there might be a need for a new hardware accelerator design to be developed to optimize the training and inference phases. FPGA adaptability ensures that the hardware remains relevant with new architecture changes being made on the go. If my design succeeds in enhancing the performance of hardware it could lessen the environmental cost of scaling CNN solutions in the industry.

- **Societal Impact:**
  This project provides both cost and energy-efficient solutions as compared to traditional hardware used in AI model training like GPUs and CPUs. Meaning that it increases access to AI technologies in a broader range of industries, including those that have high social importance, but fail to get the necessary funding. This might include sectors such as healthcare, education, and agriculture where AI technologies can benefit by potentially improving early infection diagnostics, personalizing learning experiences for each individual student, etc. Moreover, the affordability of FPGAs can encourage small businesses and organizations to incorporate AI to improve productivity and customer experience which might potentially enhance competition in the market. Meaning that the average quality of services and products customers would be getting could improve. Additionally, the energy-efficient design of FPGA-based hardware accelerators helps reduce the environmental impact of data centers and AI-driven operations. Also, the energy-efficient design of FPGA-based hardware accelerators helps reduce the environmental impact. This might be particularly relevant in regions with strict environmental regulations. By reducing energy consumption, the project supports a more eco-friendly AI infrastructure. These advancements translate into a higher quality of life for communities, as they benefit from faster, more accurate, and tailored services across critical domains.

# Chapter 2

# Background

## 2.1 Convolution Operation in CNNs

Discrete convolution is a mathematical operation that is performed on 2 discrete sequences x[n] and h[n], and outputs a discrete sequence y[n]. It is described by the following equation:

$$y[n] = (x \circledast h)[n] = \sum_{m=-\infty}^{\infty} x[m] \times h[n-m] \tag{2.1}$$

However, in the context of CNNs convolution can be described as an operation used for feature extraction. In this context, K, an array of size usually smaller than input feature map X, referred to as convolutional filter or kernel, is applied to the input X, and output feature map Y is calculated using an operation similar to the one given in Equation 2.1. Since the both input and kernel are of finite size, the equation is adjusted accordingly and given as in Equation 2.2.

$$Y[n] = (X \circledast K)[n] = \sum_{m} X[m] \times K[n-m] \tag{2.2}$$

This equation can be further narrowed to the specific application of an image input, which will be considered as a 2-dimensional matrix. In this case, the convolution operation is described as in Equation 2.3. This is essentially an element-wise multiplication of the kernel matrix and matrix created by taking part of the input matrix and summing them all to find one entry in the output feature map. This kernel is then applied to the whole matrix by considering all small partial input matrices [21, 22]. The matrix resulting from this operation will be the output feature map. This feature map will have a size of $\lfloor \frac{i-l}{s} \rfloor + 1$ by $\lfloor \frac{j-k}{s} \rfloor + 1$ for a stride size of s.

$$Y[i][j] = (X \circledast K)[i][j] = \sum_{l} \sum_{k} X[i+l][j+k] \times K[l][k] \tag{2.3}$$

This equation can be adjusted accordingly to specific applications, like for the case of RGB image, where the kernel expands to the 3-dimensional matrix.

# Chapter 3

# Methodology

## 3.1 Overview of Accelerator Architecture

This chapter presents the design methodology for the proposed convolution accelerator. The design consists of 3 primary units: A data Scheduling and Output Storing Unit, a Bit-Serial Multiplier Unit, and a 2-bit Full Adder Tree Unit. Fig. 3.1 shows the high-level block diagram of the system and the interconnection between the modules. Each module's functionality is described in detail in the subsequent sections..

The system receives input feature maps and kernel weights via memory-mapped I/O (MMIO) registers and performs convolution using bit-serial multipliers. Key variables used throughout this design include:

- X: Input feature map of size $i \times j$ with each entry having a width of $w + 1$-bits.
- K: Kernel matrix of size $l \times k$.
- Y: Output feature map of size $\frac{(i-l+1)}{s} \times \frac{(j-k+1)}{s}$, where $s$ is the stride size.

In the diagram shown in Fig. 3.1, $\Delta$ denotes the delay block, and it was realized as a D flip-flop in RTL design. Subscripts in the variable names represent the bit position. Since 2 bits are generated in one clock cycle, the less significant bit was denoted with n, and the more significant bit was denoted with subscript $n + 1$.

## 3.2 Data Scheduling and Output Storing Unit

This unit handles software-driven scheduling and memory interfacing for the convolution process. It takes one bit of each value in the kernel and input matrices and creates a parallel bus out of all these bits. This bus is connected to FPGA using PIO (Parallel I/O) Intel FPGA IP in Quartus Platform Designer. PIO is a hardware

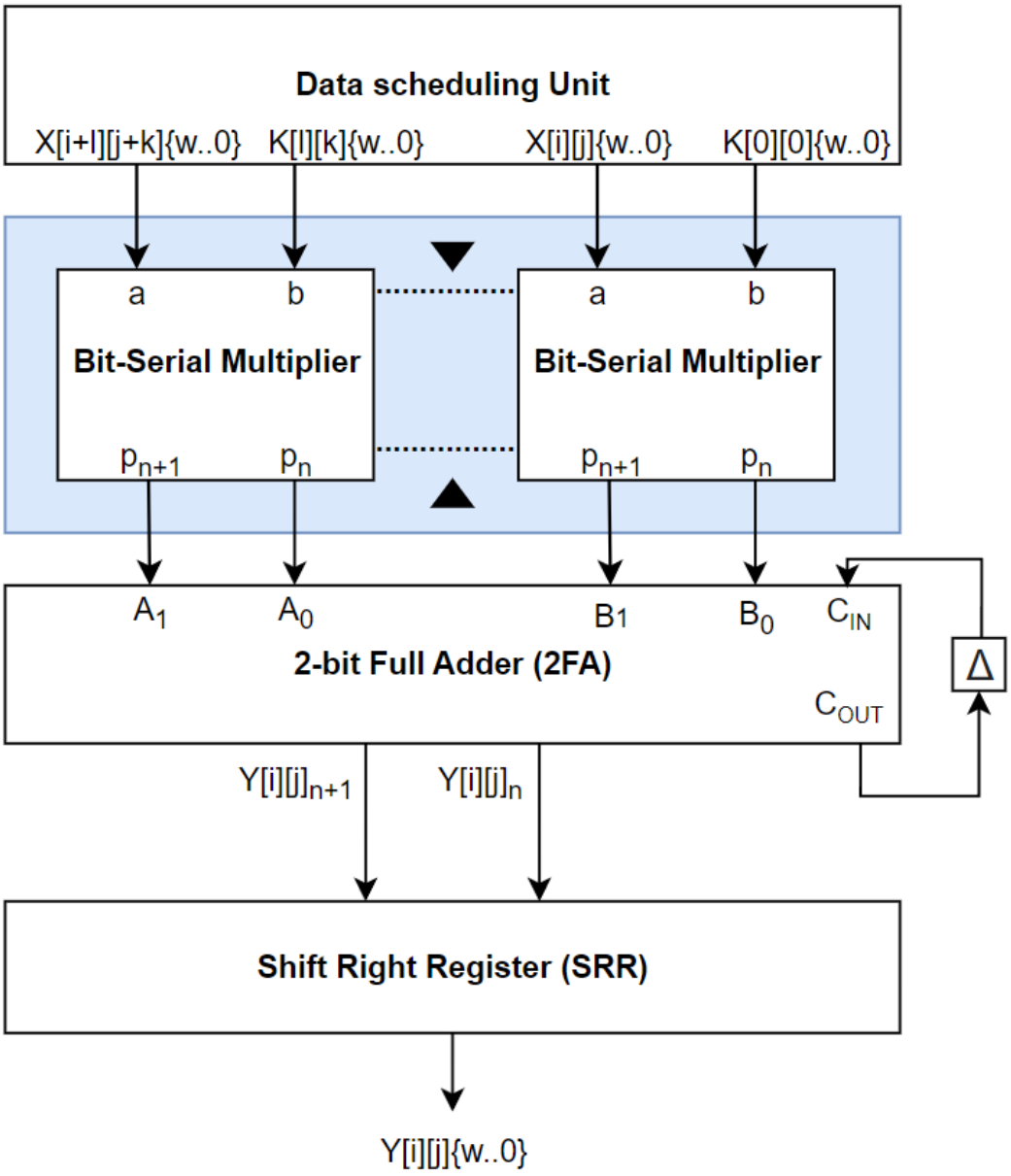


**Figure 3.1:** Accelerator schematic.

communication peripheral that connects the ARM processor with the FPGA via a lightweight HPS-FPGA bridge. Communication uses the AMBA AXI protocol and is fully automated by Quartus. Once the valid data is computed in FPGA the results are sent back using PIO and the C code reconstructs output value from output bits coming 2-bit at a time (starting from the least significant pair) from the FPGA using shifting.

This unit is realized in C code and runs in ARM Cortex A9 dual-core processor

in DE1-SoC board. Due to a lack of license to software tools, we could not implement this module using a bare-metal code. Instead, we booted a Linux operating system (OS) on an ARM processor and ran the code on it. The main difference between the bare-metal code and code running on top of an OS is that the latter includes an overhead on running an OS and requires the implementation of functions responsible for mapping virtual memory into physical memory space and the other way around. This physical memory is mapped to FPGA and provides control over MMIO which includes the PIO used in our design. The code from [23] was used as the basis for the memory mapping section of the code.

As mentioned above the communication with AXI protocol is fully automated by Quartus and we have no control over it except for reading and writing to variables in FPGA. This means that status of validity of data and readiness of the interface is unknown to us. Therefore we implemented our own additional handshaking mechanism on top of AXI and we now have more control over the communication. The scheduling algorithm implemented in C that uses custom handshaking for controlling the transactions is described below.

**Algorithm 1** Data scheduling algorithm

```
1: Extract kernel-sized portion of the input matrix
2: Reset convolution module
3: while bits_read < 2n do
4:    Wait for the ready signal
5:    Load input → PIO and assert valid
6:    Deassert valid
7:    if valid output then
8:       Store 2-bit output into the corresponding position of the result
9:       bits_read += 2
10:      Deassert 'ready' to acknowledge the successful read
11:      Reassert 'ready' to clear the acknowledgment flag
12:   end if
13:   bits_sent++
14: end while
15: Repeat for each sliding window position
```

## 3.3 FPGA Control Unit

On the FPGA side, a control unit is responsible for data transactions. We implemented it as an FSM and it follows an algorithm similar to the one described in section 3.2. Table 3.1 shows the FSM next state transition logic and Fig. 3.2 shows the FSM diagram retrieved from Quartus Netlist Viewer.

**Table 3.1:** FSM State Transition Table

| Current State | Condition | Next State |
|---|---|---|
| IDLE | valid = 1 | READ |
| IDLE | valid = 0 | IDLE |
| READ | - | WAIT_ST |
| WAIT_RES | output_valid = 1 | WRITE |
| WAIT_RES | output_valid = 0 | READ_DONE |
| READ_DONE | valid = 1 | READ_DONE |
| READ_DONE | valid = 0 | IDLE |
| WRITE | ready = 1 | WRITE |
| WRITE | ready = 0 | WRITE_DONE |
| WRITE_DONE | ready = 1 | IDLE |
| WRITE_DONE | ready = 0 | WRITE_DONE |

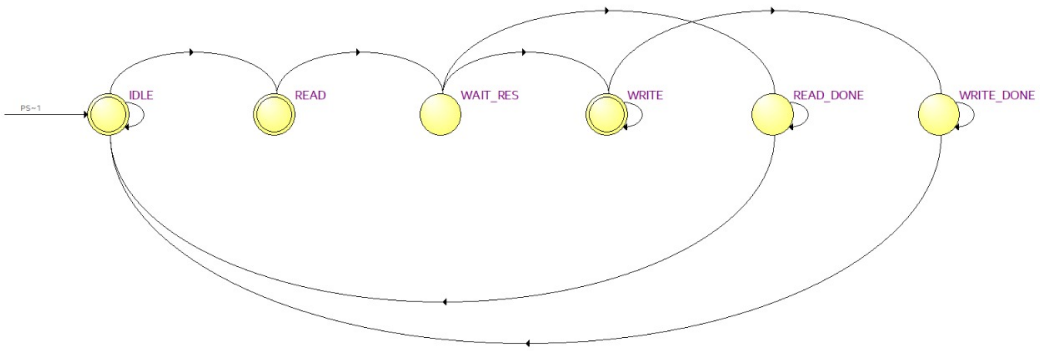


**Figure 3.2:** FSM next state transition logic.

## 3.4 Bit-Serial Multiplier

The bit-serial multiplier is the main unit that performs computations in our design. Several copies of this module are instantiated in the top module to compute all multiplications in convolution in parallel. The number of parallelized bit-serial multipliers depends on the size of the kernel. For a kernel given by the size of l by k, the design will have $l \times k$ parallel bit-serial multipliers. Fig. 3.1 displays the situation where $l \times k = 2$.

### 3.4.1 Original Design

Our design uses a novel bit-serial multiplier architecture proposed by [24] as a main module for performing multiplication operations in convolution. This architecture minimizes the overall area of the serial multiplier circuit and propagation delay. It

computes $n \times n$-bit multiplication in $2n - 3$ clock cycles with latency. The design is shown in Fig. 3.3.

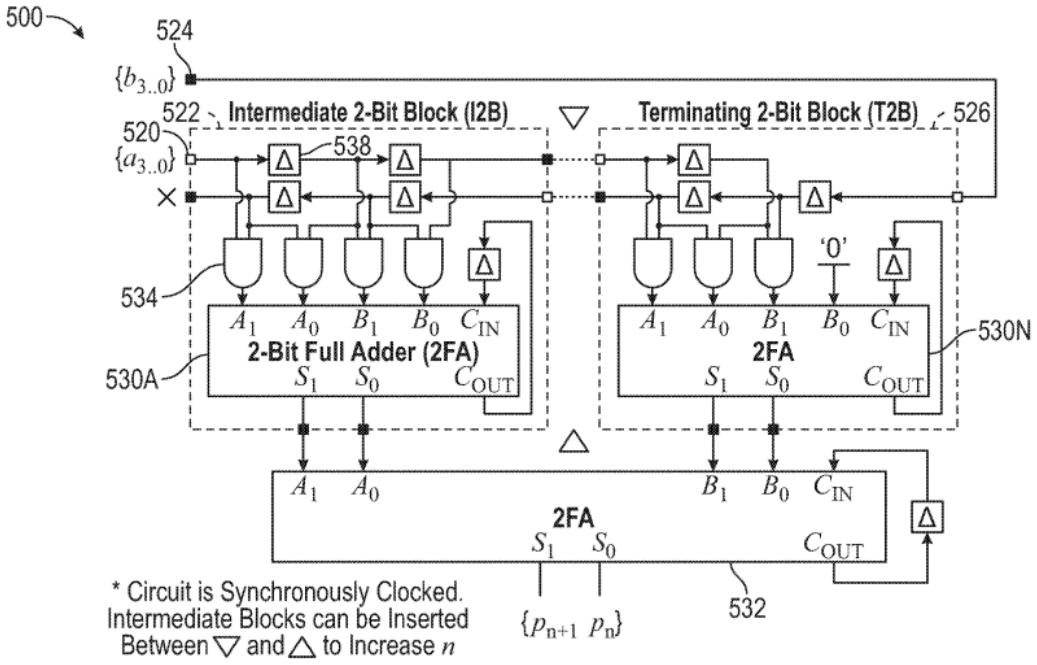


**Figure 3.3:** Bit-serial multiplier schematic with 2-bit synchronous output [24].

The design in Fig. 3.3 consists of 3 main blocks: an intermediate 2-bit block (I2B), a terminating 2-bit block, and a series of 2FAs. To increase the n (size of multiplicants) additional $(\frac{n-4}{2})$ I2B blocks should be inserted between I2B and T2B block [24]. The adder tree outside I2B and T2B blocks, consisting of 2FAs also should be adjusted accordingly. For each m number of I2B blocks in the diagram, there has to be a total of m 2FAs (outside of I2B, T2B blocks) to enable additional summation of intermediate products.

### 3.4.2 Design Modification

The design presented in the patent had some issues and was not computing correct multiplication: output didn't match the expected value, so we modified the design by removing one of the delay elements at b input in the T2B block. In modified architecture the delay changed to $(\frac{3n}{2} - 1)$ cycles with latency. This includes the latency of $(\frac{n}{2} - 1)$ cycles, and another n cycles to generate all n 2-bit output pairs. The modified serial-multiplier architecture is given in Fig. 3.4

The multiplier with the modified architecture was computing the multiplications correctly in the simulation, where inputs are given to the multiplier one bit per cycle. However, with PIO not being able to provide one bit per cycle new

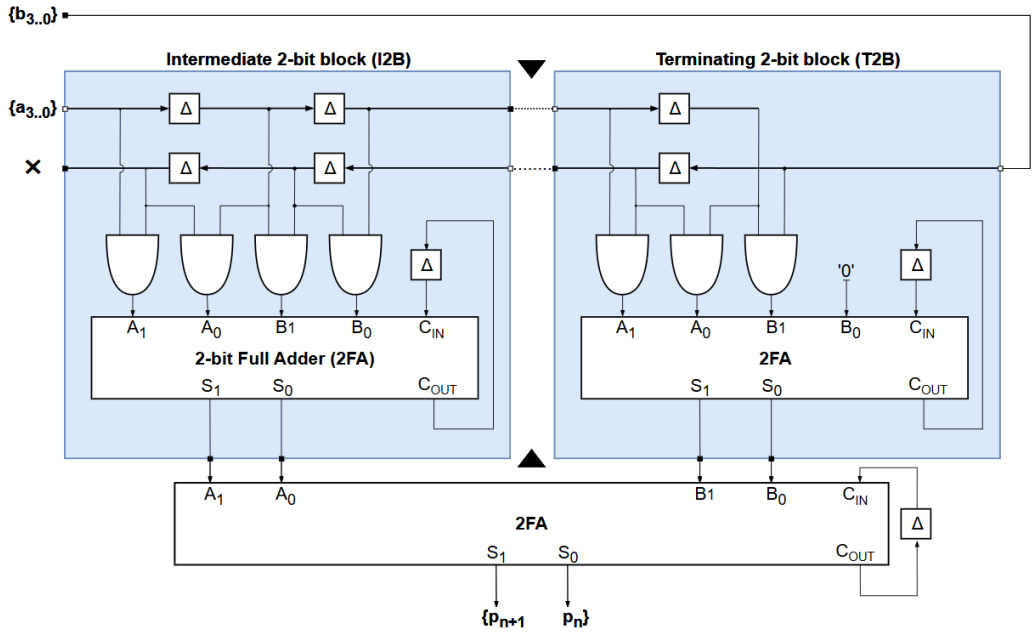


**Figure 3.4:** Modified bit-serial multiplier architecture.

issues arise with valid signal assertion. The first input, which performs computations without delay, corrupts the signals in the I2B block before other computations are performed one cycle after the valid is asserted. Therefore to avoid this problem we added additional delays in both a and b inputs. The final modified architecture is presented in Fig. 3.5 and was used in our design. This results in an additional delay of one cycle. So all 2-bit product pairs will be available at the output after $(\frac{3n}{2})$ cycles.

## 3.5 Adder tree (2-bit Full Adder Network)

This unit performs the role of an accumulator. 2-bit outputs from bit-serial multipliers are summed here to find the value in the output feature map 2-bit at a time. With the increase in the number of bit-serial multipliers, the number of 2FAs will increase to account for n input addition. Schematic in Fig. 3.1 displays the situation for n = 2. For n number of multipliers, we will need $(n - 1)$ instances of 2FA to sum all multiplication outputs. It will also create an adder tree for n higher than 2. For widely used kernel sizes like $5 \times 5$ the adder tree depth is only 5, meaning that the critical path does not increase drastically and there is probably no need for pipelining or other optimization methods aimed at decreasing the critical path.

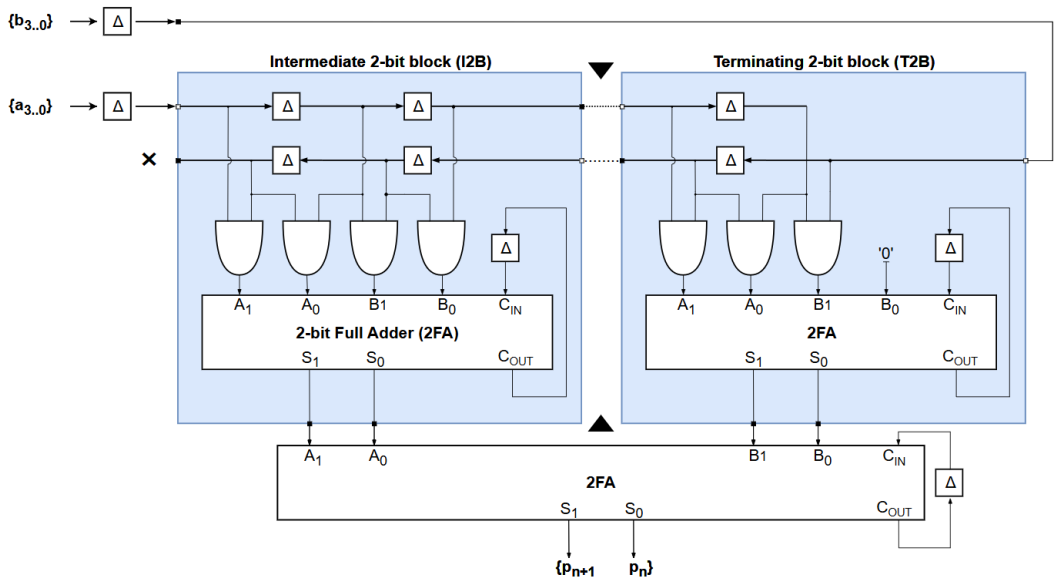


**Figure 3.5:** Modified bit-serial multiplier architecture with delayed inputs.

## 3.6 Implementation

The architecture described in section 3.1 (depicted in Fig. 3.1) was realized and implemented in Quartus Prime software. The RTL schematic of the design is given in Fig. 3.6 The presented design realizes the case when the kernel size is $5 \times 5$, therefore a total of 25 multiplier units were instantiated. These 25 units of bit-serial multipliers multiply the input feature map submatrix with corresponding values from the kernel matrix. The size of the kernel also implies that 24 2-bit full adders will be used to sum all products. This particular implementation also considers the data to be an 8-bit integer.

The code was written in a highly parameterized fashion such that variables such as data width, and kernel size could be easily modified, and the design would be adapted to new parameters. But since the output of one adder is forwarded into another one (as opposed to parallel instantiation in the case of multiplier) simple parameterization does not work for the 2FA block. The code for the adder tree has to be modified manually. Other possible solutions include a Python script and the generation of code with the help of an LLM model. Since the process of writing an HDL code for an adder tree is a simple and repetitive process we believe a solution with an LLM model is the simplest way. So we used the ChatGPT 4.0 model for a generation of HDL code for the adder tree whenever we needed a modification due to bit-width change. The prompt that we used for getting a fully functional HDL code for the adder tree from 3 attempts, at most, is given in Appendix A. The code of our implementation is also available in GitHub at [25].

A simplified RTL netlist of our convolution module is given in figure 3.6. To in-

crease readability majority of multipliers and adders were hidden in the schematic. Also, performance counters and control FSM signals are removed. From the netlist, we can see that delayed inputs X and K buses are passed to the serial multiplier units. The output of which passes to the adder tree and the output of the last level adder FA4_0 provides 2-bit output to PIO.

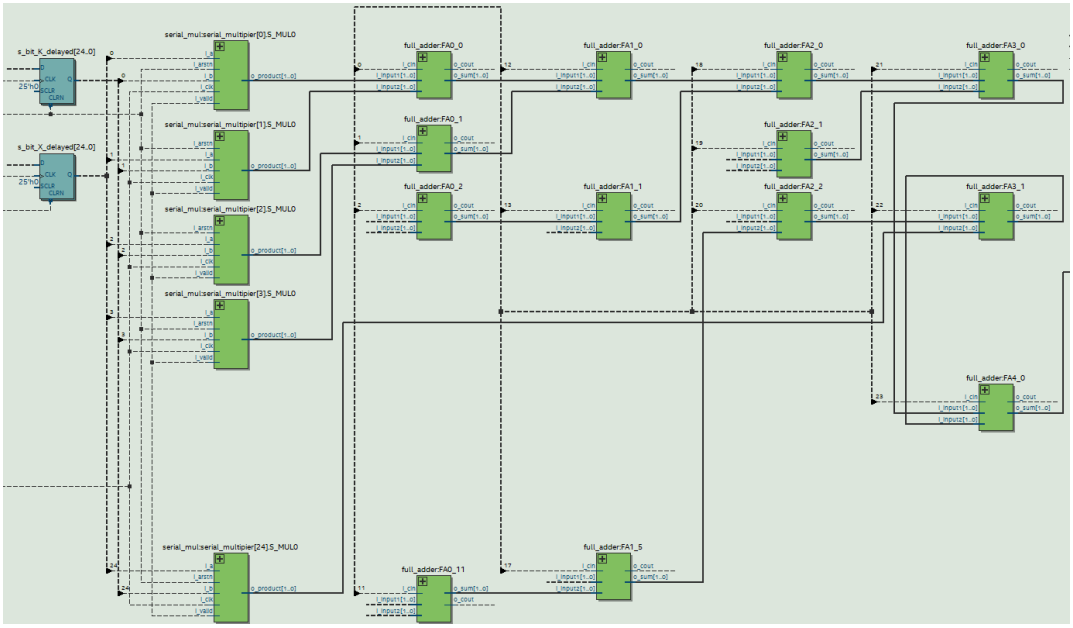

**Figure 3.6:** Modified bit-serial multiplier architecture with delayed inputs.

## 3.7 Performance metrics

### 3.7.1 Delay

The main performance metric used was the delay i.e. time required for performing the convolution operation. In C code we used `get_clock` function from time.h library to measure the time before starting the computation and after the computation. Then, we found the difference between these time moments to find the exact time the processor took to compute convolution. The overhead of calling this function was ignored, since it is significantly smaller than the time processor takes to compute millions of multiplication, and addition operations involved in the process.

To measure the time in the FPGA a simple counter was used. We measured time using two counters to measure performance with and without data transaction overhead. The first one increments only when the valid data is coming into the module ignoring the time when the multiplier is idle waiting for the next input.

The second counter starts before the first input arrives and stops counting after the last computation results are stored in C code.

The results are presented in seconds and number of cycles spent by the hardware. The number of cycles was calculated using the equation 3.1.

$$Number\ of\ Cycles = Time \times Clock\ Frequency \tag{3.1}$$

ARM Cortex processor is running at 800 MHz frequency, while the FPGA is using system clock of 50 MHz. We used these data for calculating number of cycles.

### 3.7.2 Clock Frequency

We conducted a static timing analysis to ensure that our design can operate at 50 MHz clock speed. Also, to find the maximum clock frequency at which the design will be working properly without it causing negative slack and clock skew. We constrained our design to operate at an unrealistic clock frequency of 1 GHz. Then found the worst negative slack (WNS) and used it to calculate the actual maximum frequency at which our design can operate using the equation 3.2.

$$F_{max} = \frac{1000}{1 - WNS} \tag{3.2}$$

Here calculated $F_{max}$ is given in MHz and the value WNS in ns should be used.

### 3.7.3 Area

Area or resources utilized in the FPGA are also an important aspect of performance evaluation. The resources in the DE1-SoC FPGA board include Adaptive Logic Units (ALMs), registers, DSPs, RAM blocks, and many more. However, our design requires the use of only ALMs, which are basic logic building blocks in Cyclone V family FPGAs, and registers. Our expected result for the number of D flip-flops can be calculated using the equation 3.3.

$$N_{ff} = m \times N + (m - 1) + 2m + 4 + 3 = m \times (N + 3) + 6 \tag{3.3}$$

Here m is the number of parallel instances of bit-serial multipliers and N is the number of DFFs in a single multiplier and it is given in equation 3.4. Terms $2m$ and $m - 1$ come from the registers required for delayed inputs and storing carry bits in 2FAs, respectively. While constants 4 and 3 are the bit-width of registers storing values for a counter, and FSM's present state.

$$N = 5 \times (\frac{n - 4}{2} + 1) + (\frac{n - 4}{2} + 1) + 3 = 3n - 3 \tag{3.4}$$

Here the term $(\frac{n-4}{2}+1)$ represents the number of I2B blocks in the design and since each I2B block has 5 DFFs we multiplied this number by 5. The number of 2FAs is equal to the number of I2Bs, so we added that number once more. Constant 3 is the number of DFFs in the T2B block and by summing all these terms we get the total number of DFFs in a single bit-serial multiplier unit that is equal to $3n-3$, where n is the bit-width of inputs.

Substituting equation 3.4 into 3.3 we get:

$$N_{ff} = 3 \times m \times n + 6 \tag{3.5}$$

In our case, m is equal to 25, and n is 8. So if we compute the total number of expected DFFs we get 606.

# Chapter 4

# Results and Discussions

## 4.1 Results

This section presents the experimental results obtained from the implementation of the proposed architecture and its comparison with the C code that performs the same operation. The performance results are presented for a convolution of an RGB image of size $1280 \times 720$ with a kernel of size $5 \times 5$ and a stride size = 1.

### 4.1.1 Delay

As described in section 3.7.1 we obtained two results for FPGA and a single result for C code. Results for the total time spent on processing the image with the above-mentioned dimensions are given in table 4.1.

| | Time, s | Number of Cycles |
|---|---|---|
| C | 8.801238860 | 7,040,991,088 |
| $FPGA_1$ | 0.657803520 | 32,890,176 |
| $FPGA_2$ | 9.094535880 | 454,726,794 |

**Table 4.1:** Total time.

Here results given as $FPGA_1$ represent the performance with computations done on an FPGA without additional overhead, while $FPGA_2$ represents the case with data scheduling and transfer overhead.

In addition to total time, average time per computation was calculated and the results are given in table 4.2. Here single computation refers to the calculation of a single entry in the output matrix. So one computation consists of 25 multiplications and 24 additions for a case where the kernel has dimensions $5 \times 5$ as in our example.

| | Time, $\mu$s | Number of Cycles |
|---|---|---|
| C | 3.211137159 | 2568.910 |
| $FPGA_1$ | 0.24 | 12 |
| $FPGA_2$ | 3.318146749 | 165.907 |

**Table 4.2:** Average time per computation.

Moreover, to ensure correctness we compared each entry in the output matrix obtained from the C code and the FPGA module. The results consistently matched each other for various input and kernel matrices.

### 4.1.2 Clock Frequency

The results of the timing analysis demonstrated that clock constrained to 1 GHz results in worst negative slack of $-4.323$ ns at slow 1100mV 85C model. This means that our module could run at maximum clock frequency of approximately 187.86 MHz if external oscillator was used instead of system clock of 50 MHz in DE1-SoC board.

### 4.1.3 Area

Another important aspect of performance analysis is area, or resource utilization that directly affects the area. Table 4.3 shows the list of utilized resources in the convolution module in the DE1-SoC board.

| | ALMs, units | Registers, units |
|---|---|---|
| FPGA | 453 | 835 |

**Table 4.3:** Resource utilization.

## 4.2 Discussions

The results presented in section 4.1 demonstrate that the proposed design and CPU take a similar time to perform the computations. We can see that computation takes a total of 8.8 seconds in CPU, while the hardware module in FPGA takes about 9.1 seconds when the data scheduling and transfer overhead is included in performance measurements. However, if we compare the raw number of cycles taken by these hardware units to compute the convolution we can see that our proposed architecture performs significantly better with a total of slightly above 450 million cycles as compared to 7 billion cycles taken by the ARM core. Despite running at a clock speed 16 times slower FPGA module loses by only about 0.3 seconds to the CPU demonstrating higher raw performance. Moreover, if the external oscillator with a maximum allowed clock frequency of 187.86 MHz was used, computation on an FPGA would have taken only 2.42 seconds. In this way, FPGA would be outperforming the CPU by a factor of more than 3.6.

If we compare the average time required for each hardware to compute a single convolution we can see that the ARM core and the FPGA show very similar results at 3.21 and 3.32 $\mu$s in terms of time and 2569 and 166 clock cycles, respectively.

It is also important to note that there is an additional overhead for disabling the counter. The performance counter used for counting the number of cycles with overhead is enabled before starting transactions and then disabled after a single computation is done. This repeats for each convolution. This means that the counter keeps working after the computations are complete. So while the counter start flag is being deasserted the counter keeps counting the number of cycles. This process involves reading the current value from the PIO port ( 7–8 cycles), AND-gating it with the corresponding mask, and writing results back into PIO ( 14–15 cycles). This creates an overhead of at least 21 cycles. So the taken average should be decreased to at least 145 cycles. It shows even higher performance improvement compared to the core, reaching approximately 18 times improvement. However, it is still lower than the parallel instances of multipliers used in our design, 25. The C code computes each multiplication and addition, involved in the convolution, one after another, sequentially. Our hardware module computes all operations involved in a single convolution in parallel, although input bits for each of the multipliers are received serially. If we compute the total time without the overhead for disabling the counter we would get approximately 7.9 seconds with the current clock, and outperform the C code by 0.9 seconds. As we can see a simple overhead for disabling the counter has introduced a delay of about 1.2 seconds. For future experiments, it might be advisable to test the design with a counter that has a smaller overhead to get more accurate results.

Let's analyze the performance of our hardware module without any overhead. As expected a single computation takes 12 cycles, which corresponds to $(\frac{3n}{2})$ cycles

for n = 8. For the total computation, it takes 32 million cycles, or 0.658 seconds with a 50 MHz clock in the FPGA. This shows better performance than the C code in terms of time by a factor of 13 and 214 in terms of the raw number of cycles spent by the hardware. It should also be noted that the computation itself takes less than 10% of the total time required for the hardware to compute the results. Meaning that the majority of delay is due to the data scheduling, transaction, and counter disable logic overhead rather than multiplication.

In terms of area, a total of 453 ALMs and 835 registers were used. This data is for the case where performance counter registers were removed from the design to get more accurate data on resource utilization. As it can be seen from table 4.3 the number of DFFs does not match the expected number that we calculated in section 3.7.3. However, out of 835 registers 609 are design implementation registers, meaning these are the D flip-flops actually instantiated or inferred in the design. While the remaining 226 are routing optimization registers added by Quartus Fitter during placement and routing. This is done to balance skew and improve timing closure. But still, the number is off by 3 units of DFFs. We made additional calculations, but could not figure out where these 3 additional DFFs can be coming from. We also checked Quartus warning messages to ensure that no additional latches, or DFFs have been inferred in the design by the EDA software. This might be due to optimizations by Quartus that are not listed in the utilization report, or miscalculations by us, but we believe that either way, the difference is not huge.

# Chapter 5

# Conclusion

This paper presented an FPGA-based convolution operation accelerator based on novel serial-serial multiplier architecture. This design utilizes the FPGA on the DE1-SoC board to realize the multiplier architecture, and ARM Cortex A9 dual-core processor to parse, and schedule the input data from external memory for processing. The novel bit-serial multiplier architecture used in the design computes a single convolution operation in $\frac{3n}{2}$ cycles. Our convolution module was able to show better performance than a simple C code thanks to highly parallelized instances of multipliers when overhead for data scheduling and transaction was ignored. But even with the overhead, we were able to achieve results similar to the C code.

This paper only compares the performance of our model with the CPU performance rather than the GPU that is traditionally used for such workloads. However, it is safe to assume that in comparison with GPU our proposed architecture would lose. This is because the GPU performs each multiplication in parallel and also all multiplications involved in the convolution would be performed at the same time. While our module implements only the latter parallelization and does not aim to compete with GPUs. It would rather be used in different applications like edge AI devices where time-sensitive data can be sent to the cloud for computing, while non-time-sensitive data can be analyzed directly on the hardware with a small footprint.

While the current implementation achieves promising results, there is still room for improvement. This design can be implemented in industry level FPGA board that could result in smaller communication overhead and higher clock frequency. Also, this design can be implemented on an SoC with FPGA on a custom PCB board, rather than evaluation board, and used as part of a different system.

# Bibliography


[1] Vivienne Sze et al. "Efficient processing of deep neural networks: A tutorial and survey". In: *Proceedings of the IEEE* 105.12 (2017), pp. 2295–2329.

[2] Cecilia Latotzke and Tobias Gemmeke. "Efficiency versus accuracy: a review of design techniques for DNN hardware accelerators". In: *IEEE Access* 9 (2021), pp. 9785–9799.

[3] Yudong Tao et al. "Challenges in energy-efficient deep neural network training with FPGA". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition workshops*. 2020, pp. 400–401.

[4] Griffin Lacey, Graham W Taylor, and Shawki Areibi. "Deep learning on fpgas: Past, present, and future". In: *arXiv preprint arXiv:1602.04283* (2016).

[5] Ahmad Shawahna, Sadiq M Sait, and Aiman El-Maleh. "FPGA-based accelerators of deep learning networks for learning and classification: A review". In: *ieee Access* 7 (2018), pp. 7823–7859.

[6] Jiajun Li et al. "SmartShuttle: Optimizing off-chip memory accesses for deep learning accelerators". In: *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2018, pp. 343–348.

[7] Shihui Yin et al. "XNOR-SRAM: In-memory computing SRAM macro for binary/ternary deep neural networks". In: *IEEE Journal of Solid-State Circuits* 55.6 (2020), pp. 1733–1743.

[8] Abhinav Podili, Chi Zhang, and Viktor Prasanna. "Fast and efficient implementation of convolutional neural networks on FPGA". In: *2017 IEEE 28Th international conference on application-specific systems, architectures and processors (ASAP)*. IEEE. 2017, pp. 11–18.

[9] Ciara Rafferty, Máire O'Neill, and Neil Hanley. "Evaluation of large integer multiplication methods on hardware". In: *IEEE Transactions on Computers* 66.8 (2017), pp. 1369–1382.

[10] Yunlong Mao et al. "A privacy-preserving deep learning approach for face recognition with edge computing". In: *Proc. USENIX Workshop Hot Topics Edge Comput.(HotEdge)*. 2018, pp. 1–6.

[11] Pradeep Kumar Mallick et al. “Brain MRI image classification for cancer detection using deep wavelet autoencoder-based deep neural network”. In: *IEEE Access* 7 (2019), pp. 46278–46287.

[12] Sparsh Mittal, Himanshi Gupta, and Srishti Srivastava. “A survey on hardware security of DNN models and accelerators”. In: *Journal of Systems Architecture* 117 (2021), p. 102163.

[13] Aaron Stillmaker and Bevan Baas. “Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm”. In: *Integration* 58 (2017), pp. 74–81.

[14] Eriko Nurvitadhi et al. “Can FPGAs beat GPUs in accelerating next-generation deep neural networks?” In: *Proceedings of the 2017 ACM/SIGDA international symposium on field-programmable gate arrays*. 2017, pp. 5–14.

[15] Soheil Hashemi et al. “Understanding the impact of precision quantization on the accuracy and energy of neural networks”. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE. 2017, pp. 1474–1479.

[16] Kaiyuan Guo et al. “[DL] A survey of FPGA-based neural network inference accelerators”. In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 12.1 (2019), pp. 1–26.

[17] Shuang Liang et al. “FP-BNN: Binarized neural network on FPGA”. In: *Neurocomputing* 275 (2018), pp. 1072–1086.

[18] Sparsh Mittal. “A survey of FPGA-based accelerators for convolutional neural networks”. In: *Neural computing and applications* 32.4 (2020), pp. 1109–1139.

[19] Vivienne Sze et al. “Efficient processing of deep neural networks: A tutorial and survey”. In: *Proceedings of the IEEE* 105.12 (2017), pp. 2295–2329.

[20] Erwei Wang et al. “LUTNet: Rethinking inference in FPGA soft logic”. In: *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE. 2019, pp. 26–34.

[21] Trung Pham-Dinh et al. “An FPGA-Based Solution for Convolution Operation Acceleration”. In: *International Conference on Intelligence of Things*. Springer. 2022, pp. 279–288.

[22] Rikiya Yamashita et al. “Convolutional neural networks: an overview and application in radiology”. In: *Insights into imaging* 9 (2018), pp. 611–629.

[23] Intel Corporation. *Using Linux* on DE-series*. Accessed: 2025-04-20. 2019. URL: https://ftp.intel.com/Public/Pub/fpgaup/pub/Teaching_Materials/current/Tutorials/Linux_On_DE_Series_Boards.pdf.

[24] Akhan Almagambetov and Holly Renee Ross. “Bit-Serial Multiplier for FPGA Applications”. Pat. US 10275219 B2. Retrived from https://commons.erau.edu/publication/2215. 2019.

[25] Olzhas Nurman. *conv_w_serial_mult*. Accessed: 2025-04-20. 2025. URL: https://github.com/olzhasnurman/conv_w_serial_mult.

# Appendix A

# Prompt to ChatGPT

In this section, the prompt to generate adder tree is presented.

```
Using the following template, create an adder tree that will
sum 25 (N) instances of s_product. There will be 24 (for N =
25, N - 1) instances of a full adder:

  logic [1:0] s_sum_0;
  logic [1:0] s_sum_1;

  full_adder FA2_0 (
    .i_input1 ( s_product[0]   ),
    .i_input2 ( s_product[1]   ),
    .i_cin    ( s_carry_in[0]  ),
    .o_sum    ( s_sum_0        ),
    .o_cout   ( s_carry_out[0] )
  );
  full_adder FA2_1 (
    .i_input1 ( s_product[3]   ),
    .i_input2 ( s_product[4]   ),
    .i_cin    ( s_carry_in[1]  ),
    .o_sum    ( s_sum_1        ),
    .o_cout   ( s_carry_out[1] )
  );

Follow the given instructions:
1. Generate all N-1 instances. Do not leave behind instances
saying: "continue like that"
2. First sum all s_product[i] instances and only then start
summing s_sum_*
3. Keep in mind that when N is odd, s_product_{N-1} will be
left for the end. You can ignore it and start summing s_sum_*,
and you will add that in the LAST level of the adder tree with
```

```
s_sum_{N-2}.
4. Remember to declare s_sum_* signals
```