Chapter 1

Introduction

Convolution is one of the most fundamental building blocks in CNNs and indeed perhaps the most basic operation for computer vision tasks in general. Reformulating convolution as matrix multiplication and using GeMM has, therefore, turned out to be an extremely strategic advantage. The GeMM has been specially targeted for this particular project for optimizing parallelism and resource allocation to the FPGA for increased computational efficiency. Using the im2col technique together with an FSM tailored to the specification of the required application, it also addresses the memory constraints and challenges of real-time processing. Such applications gain increased throughput as well as low latencya requirement for dealing with complex applications that manipulate volumes of data efficiently over hardware platforms.

1.1 Background

This project deals with the acceleration of the convolutional operations on FPGAs with the use of the General Matrix-Matrix Multiplication algorithm. GeMM is imperative in accelerating computer vision tasks including image and object recognition, and segregating images in Convolutional Neural Networks. These CNN architectures evolve towards achieving the maximum accuracy obtained and this indirectly means large computation complexity and data handling requirements of this process. This slightly poses a challenge to traditional CPUs and GPUs that have limited power efficiency and also have poor real-time processing ability, especially in power-constrained environments like mobile vision applications. The reconfigurable architecture of FPGAs has tremendous promise in their inherent ability for parallel processing through custom implementations targeting particular operations. This project applies the GeMM algorithm in the conversion of convolutions into matrix multiplications in order to optimize the resources of the FPGA and achieve better performance. The method proposed uses this technique to

also address one of the shortcomings of traditional hardware in high-performance designs for a balancing factor of performance versus energy efficiency, especially excellent for real-time, power-sensitive applications on an FPGA.

1.2 Problem Definition

- Real-time convolution operations are computationally expensive, making softwarebased implementations on embedded processors inefficient for deep learning applications.
- Traditional processors struggle with high execution times due to sequential processing and limited parallelism.
- Performing multiple convolutions simultaneously increases computational demands, leading to bottlenecks in resource-constrained environments.
- Efficient acceleration techniques are required to reduce execution time while balancing hardware resource utilization and power consumption.

1.3 Scope and Motivation

The motivation for this work, there is heavy use of convolution operations that do not fit well with the traditional processing units. Still, CPUs as well as GPUs can be used for such resource-intensive operations to a small extent. However, they miserably fail regarding power efficiency and real-time processing in such applications, and to a large extent, they are rendered useless when the applications are massive or complex in nature. This challenge is more specifically pertinent in the case of CNNs that have multiple layers of convolutions requiring huge processing power and memory bandwidth in deep learning models.

By utilizing the parallelism and reconfigurability available from FPGAs, it is now possible to design a system specifically targeting convolution operations. This allows implementing the GeMM algorithm structured and resource-efficiently, transforming the problem of multi-dimensional convolution into matrix operations that can be managed

with much greater efficacy. The im2col transformation is also used in this project for data management in minimizing the latency that thus minimizes the energy required for computation-an important aspect in the case of mobile and embedded applications. Such a system's outcome would therefore have good performance, thus making FPGAs a very viable alternative for deep learning tasks compared to traditional hardware, with great power and efficiency considerations in the environment.

1.4 Objectives

1. Design and Implement FPGA-Based Convolution Using GeMM
2. Compare convolution operations between hardware and software. .
3. Optimize Execution Time Using Segmented Frame and Parallel Processing
4. Analyze Resource Utilization and Performance Trade-Offs
5. Implement Multi-Kernel Convolution for Enhanced Throughput

1.5 Challenges

Challenges of acceleration of convolution operations through GeMM algorithms on FPGAs: It requires sophisticated design skills as the development of FPGAs lies in proprietary technology; thus it is more complex and time-consuming than a software solution. Memory is a factor of the convolution operation, which is memory-intensive; thus memory management with limited resources of on-chip in FPGAs further complicates things. Detailed pipelining and parallelism are hard to implement at the best achievable latency and throughput for real-time processing. Scaling the accelerator to support various CNN architectures along with energy efficiency, even at the edge, would be far more challenging. Lastly, broad debugging and verification for reliable performance adds up to further development time and effort, hence underlining the requirement for optimized and advanced design strategies.

1.6 Assumptions

Grayscale images were used instead of RGB to reduce computational complexity in convolutional lowering by decreasing the number of input channels from three to one. In the FPGA implementation, image pixel elements are of 8 bit and the kernel coefficients have a 9 bit signed precision. This approach efficiently manages data size while minimizing output truncation effects, thereby preserving result accuracy. It is also assumed that the size of the image is 224 x 224 and the kernel size is 3 x 3 and the stride is equal to 1 and with no zero padding.

1.7 Societal / Industrial Relevance

The implementation of the FPGA is a GEMM convolution operation with an acceleration model of social and industrial importance. Optimized processing in such convolution operations will make CNN deployments easy and efficient in various industries such as medical, self-driving cars, security among others. This, in turn, can achieve quicker and more reliable analysis of information helping industries that require high speed manipulation of huge amounts of intricate data be it into AI integrated systems or virtual assistants.

Moreover, with the FPGAs ' energy efficiency, edge computing and IOT applications allow on-site real-time processing of the data and hence reduce its dependency on cloud services. This can be applied if A: Smart cities, industrial automation, and wireless devices all involve power and area constraints.

In addition to that, FPGAs are also cost-effective, and hence, contribute to making supercomputing resources accessible to startups and other small institutions out there thus leveling the playing field of AI progress. What's more, the goals of optimization of convolution for FPGA mentioned above also help in lessening the carbon footprint of data centers by clean energy consumption thereby conforming to climate goals set forth for technological progress.

1.8 Organization of the Report

The report is systematically structured to present a comprehensive analysis of FPGAbased convolution using the GeMM algorithm. The introduction provides an overview of the importance of hardware acceleration for convolution operations, discussing the advantages of FPGA implementations over traditional CPU and GPU-based approaches. It highlights the challenges in optimizing execution time, power consumption, and resource utilization. The methodology section details the implementation of convolution using the GeMM algorithm, explaining the design choices such as tiling and parallel processing to maximize computational efficiency. It also describes the experimental setup and testing procedures used to evaluate system performance.

The results and discussion section presents key findings from hardware and software evaluations, comparing execution speed, resource utilization, and accuracy trade-offs. The analysis highlights the benefits of FPGA-based acceleration while addressing challenges such as memory bandwidth limitations and hardware constraints. The conclusion summarizes the major insights gained from the study, emphasizing the effectiveness of the proposed approach. It also suggests potential future enhancements, including optimizing memory access patterns and improving power efficiency. The final sections of the report include references and appendices that provide additional details on implementation, datasets, and performance metrics.

Chapter 2

Literature Survey

Our team conducted an extensive literature review to provide context and insight for the investigation. We collectively examined and evaluated numerous relevant scholarly publications to construct the theoretical foundation of our study. These publications were carefully selected after a comprehensive assessment process and serve as critical references that support our understanding of the topic. Through our collaborative effort,

we aim to address gaps in the current literature, contribute to the advancement of existing knowledge, and present significant findings to the academic community by integrating concepts and methodologies from these scholarly works.

2.1 "Optimizing Hardware Accelerated General Matrix-Matrix Multiplication for CNNs on FPGAs" [1]

In this paper, we propose a hardware acceleration approach for convolutional neural networks by transforming the computationally intensive 2D convolution operation into General Matrix-Matrix Multiplication, or GeMM, via Tensor Rank Reduction. This is done by reshaping high-dimensional tensors representing the kernel and input tensors to lower-dimensional matrices. We have implemented an optimized GeMM engine on an FPGA and demonstrated its effectiveness in accelerating convolutional layers of small lightweight models, such as ShuffleNet, designed for mobile and embedded applications. This approach achieved an inference time of 1.27 ms with competitive performance with state-of-the-art hardware implementations. The architecture shows potential for scaling toward supporting complex neural networks and optimization to ASICs; dynamic quantization may further enhance the accuracy-performance trade-offs.

2.2 "Optimizing the Convolution Operation to Accelerate Deep Neural Networks on FPGA" [2]

This paper applies the improvement of convolution speeds in deep neural networks for FPGA hardware. In this scenario, as it is primarily the case with convolutions that dominate processing, optimizing the same should directly influence the maximum speed attained. Previous methods like breaking tasks into smaller loops (loop unrolling and tiling) lack full optimization of memory usage or data processing, and hence slow down performance.

To address these, this study introduces a BUF2PE (Buffer to Processing Element) data bus that simplifies the flow of data from memory to processing units, especially using different strides and paddings. This system allows good reuse of data with reduced memory accesses and increased efficiency. Besides, data movement and processing at different

levels adjustment methods from which techniques like loop unrolling and tiling based on optimization of FPGA resources use are applied. Such methods tested on Intel FPGA platforms like Stratix V and Arria 10 improved the speed of the DNN models VGG-16 and ResNet-50/152, so the approach seems to be promising for applying complex DNNs on an FPGA.

2.3 "Methodology for CNN Implementation in FPGA-Based Embedded Systems" [3]

Federico G. Zacchigna *et al.* Presented a methodology to improve the efficiency of implementing CNNs on FPGAs for embedded systems. This brings to light the potential of FPGAs as a prospective substitution for GPUs with substantial gains in energy efficiency and lower latency, apart from saving money if high performance is a critical criterion. However, the methodology proposal overcomes problems of using FPGAs, such as complexity in the development of a system and poor reusability of the design of heterogeneous architectures. It provides a structured, high-level architecture that partitions CNNs into manageable parts of PBs, DFBs, and MOBs. Such modular components can improve flexibility, simplify the design, and allow real applications with reduced latency and wastage of resources. The architecture also highlights the effective usage of DSP resources and cache memory toward the highest throughput, as can be viewed in a CNN model implementation referred to as LeNet in the module, wherein the extensive improvements of performance are identified in various metrics when compared with the usual implementation. The approach being adaptive to new CNN models, the integration of new modules is easier to carry out; thus, it is relevant to applications that require continuous advancement in models related to machine learning on FPGA-based platforms.

2.4 "An Energy-Efficient GeMM-Based Convolution Accelerator With Onthe-Fly im2col" [4]

This paper proposes a GeMM-based accelerator architecture that introduces several key modifications to improve efficiency in executing computation-intensive operations, particularly for dilated convolutions. The Key components include the SA GeMM engine,

double-buffered SRAM memories, a data feeder module, and a partial sum manager. The SA GeMM engine is designed with added efficiency through a zero detection and gating circuit, minimizing unnecessary operations, and a reserve register to reduce latency. The data feeder module manages the convolutional lowering process in real-time, aligning the kernel pattern with data in SRAM, and handling non-contiguous data access efficiently for dilated convolutions. The ability to handle data inflow and computations simultaneously, while ensuring synchronization through the staggered feed registers, improves the throughput, reduces latency, and optimizes resource utilization while maintaining low power overhead. All these features make the architecture highly suitable for energyefficient neural network inference and training.

Chapter 3

Methodology

The FPGA-based approach to hardware accelerated 2D convolution via the GEMM algorithm overcomes computational issues typically faced in convolution operations in computer vision and neural networks. The design starts with converting input image patches into a linear array and buffering them in a 1 × 9 buffer, then multiplying by the kernel. A finite state machine (FSM) controls the execution order, minimizing resource usage and hardware performance. The result of convolution is saved in a block memory of the FPGA to be processed further if necessary. This technique increases the efficiency of FPGA by maximizing resource usage and data flow, providing high performance and power efficiency, which is particularly useful for resource-restricted applications.

3.1 Conventional 2D Convolution

2D convolution is the most basic operation in the processing of images and computer vision, and it is an important technique in applying the process for feature extraction, image enhancements, and even transformations. In this process, systematically, the kernel

or filter is slid across the image where at each position element-wise multiplications are followed by summation. This results in a transformed output image; sometimes these are called feature maps.

3.1.1 Steps of Sliding Convolution Technique

1. **Initialization**:The input image as a 2D matrix of pixel intensity values. The kernel is a smaller matrix, often of size 3×3, 5×5 or similar dimensions, with predefined values that determine the type of operation (e.g., edge detection, sharpening).

2. **Padding**: For the output image to have certain dimensions or characteristics, zeropadding is applied to the edges of the input image. Padding helps the kernel in processing edge pixels efficiently without severely reducing the image size.

3. **Convolution Sweep**: The kernel aligns to the particular part of the image matrix from the top-leftmost corner. Element-wise multiplication is performed at each position between the kernel values and the corresponding image pixel values in the overlapping region.

4. **Elementwise product result**: The products obtained from the element-wise multiplication get added together to give a single value. The value is then written to the kernel's corresponding position in the output feature map.

5. **Shift the Kernel**: This kernel is horizontally shifted along each complete row, then to the next row, and repeats until it covers the entire image matrix. The stride, in this example, is the step size through which the kernel moves. If the stride is taken to be 1, then the kernel moves one pixel. If the stride is some greater number, then the kernel can move much faster and output dimensions are smaller.

6. **Output Generation**: The process produces an output as a single feature map, where each value is the filtered result when the kernel is applied to some portion of the input image. If there are multiple kernels, which is the case for CNNs, then there are multiple feature maps, one for each of the filters.

3.2 2D Convoltuion using GeMM

In this approach, the input image is reshaped into a matrix using the Im2Col method. This method allows for efficient parallel computation by converting convolution into matrix multiplication. Likewise, the kernel is reshaped in the form of a matrix where all the filters are flattened to make its application all along the image possible. It is then convoluted using GeMM, which converts the output matrix back into the spatial dimensions of the feature map and results in a filtered image or feature map. This method significantly improves computational efficiency, especially when dealing with large images and multiple filters in CNNs.

3.2.1 Steps Of 2D Convolution Using GeMM Algorithm

1. **Input Image Transformation (Im2Col)**: An input image is reorganized in a matrix form as shown in figure 3.1 where each column is an overlapping patch of the image with the kernel in the convolution process. For an image of size $H \times W$ and a kernel of size $K \times K$, a sliding window of $K \times K$ is extracted across the image with strides. Each $K \times K$ patch is flattened into a column vector, creating a large matrix. In this manner, the convolution is reduced to a large matrix multiplication operation, allowing for parallel computation.

Input Image Transformation: Im2col (image to column)

19	44	16	12	47
41	28	24	9	22
27	32	12	12	10
18	30	43	22	46
47	11	10	16	49

19	44	16	41	28	24	27	32	12
41	28	24	27	32	12	18	30	43
27	32	12	18	30	43	47	11	10
44	16	12	28	24	9	32	12	12
28	24	9	32	12	12	30	43	22
32	12	12	30	43	22	11	10	16
16	12	47	24	9	22	12	12	10
24	9	22	12	12	10	43	22	46
12	12	10	43	22	46	10	16	49

Figure 3.1: performing im2col(convolutional lowering) on 5 × 5 matrix

2. **Kernel Reshaping**: The kernel is also reshaped into a matrix as shown in figure 3.2. For a kernel with dimensions $K \times K \times C_{in}$ (channels in) and C_{out} filters, each filter is flattened into a row vector. The resulting kernel matrix has dimensions $C_{out} \times (K \cdot K \cdot C_{in})$.

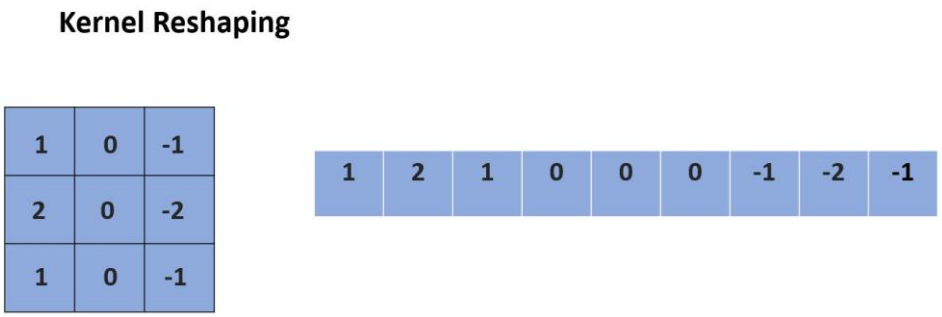

Figure 3.2: Reshaping 3 × 3 kernel

3. **Matrix Multiplication**: The transformed image matrix from the Im2Col transform is multiplied by the reshaped kernel matrix using standard GeMM operations as shown in figure 3.3:

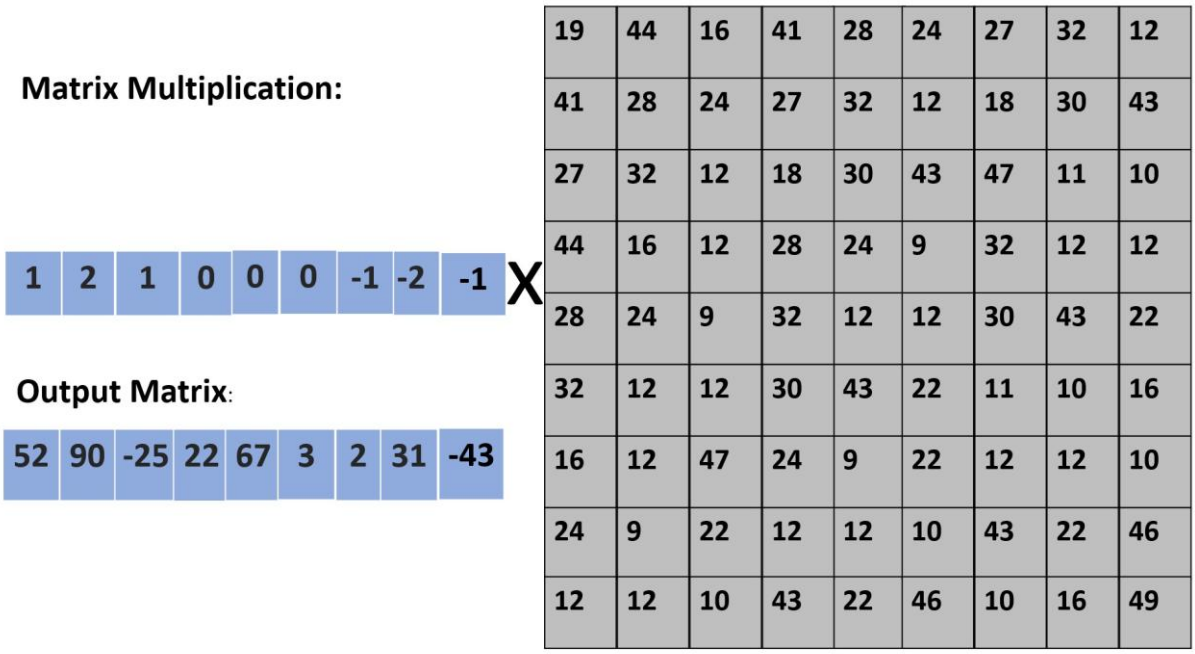

Figure 3.3: Matrix Multiplication to obtain the convolution result

$$\text{Output Matrix} = \text{Kernel Matrix} \times \text{Image Matrix}$$

This step has low computational complexity and relies on highly optimized GeMM implementations.

4. **Computing the Output Feature Map**: The GeMM result represents the convolution in a flat form. It is reshaped back into the spatial dimensions of a feature map.

3.4 System Architecture

The system architecture is based on an SoC, integrating a Processing System (PS) and Programmable Logic (PL). These components work together, as illustrated in Figure 3.5, which details the system architecture. The design leverages the PS for control and high-level processing, while the PL handles accelerated computations, ensuring efficient performance and optimized resource utilization.

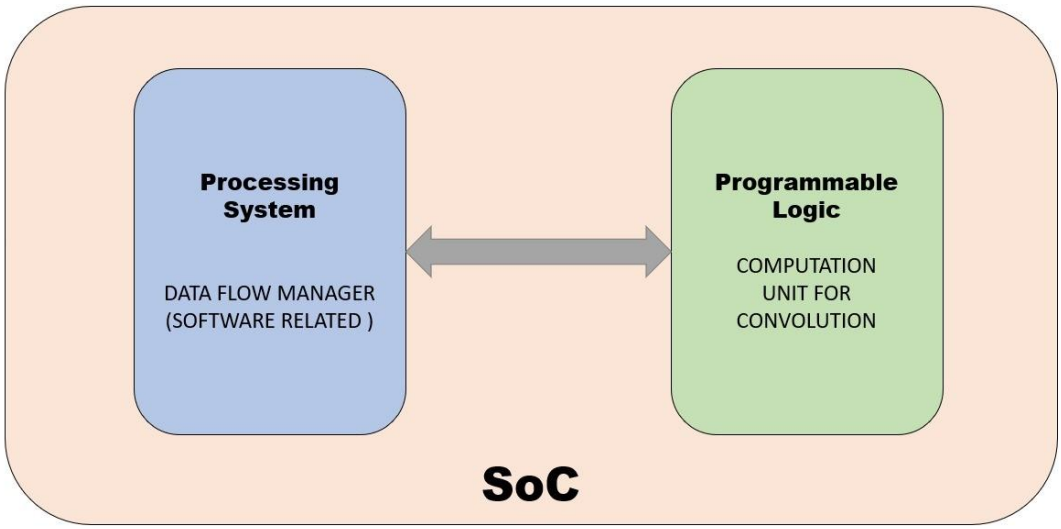

Figure 3.5: Overall system block

The PS is tasked with controlling the overall system control, data flow, and communication with the PL. The PL, however, conducts high-speed computations, taking advantage of hardware resources like block memory (BRAM) and special engines, like the GeMM engine, to speed up operations. This segregation of tasks makes the PS perform control and communication while the PL speeds up computation-intensive tasks like matrix multiplication in convolution operations.

3.6 GeMM-Based 2D Convolution Implementation On A Programmable Logic

Figure 3.7 describes the FPGA implementation of 2D convolution by the GeMM approach. The system processes image data efficiently with the use of block RAM (BRAM), buffering, and parallel processing. This is one of the essential operations for machine learning and image processing involving convolution.

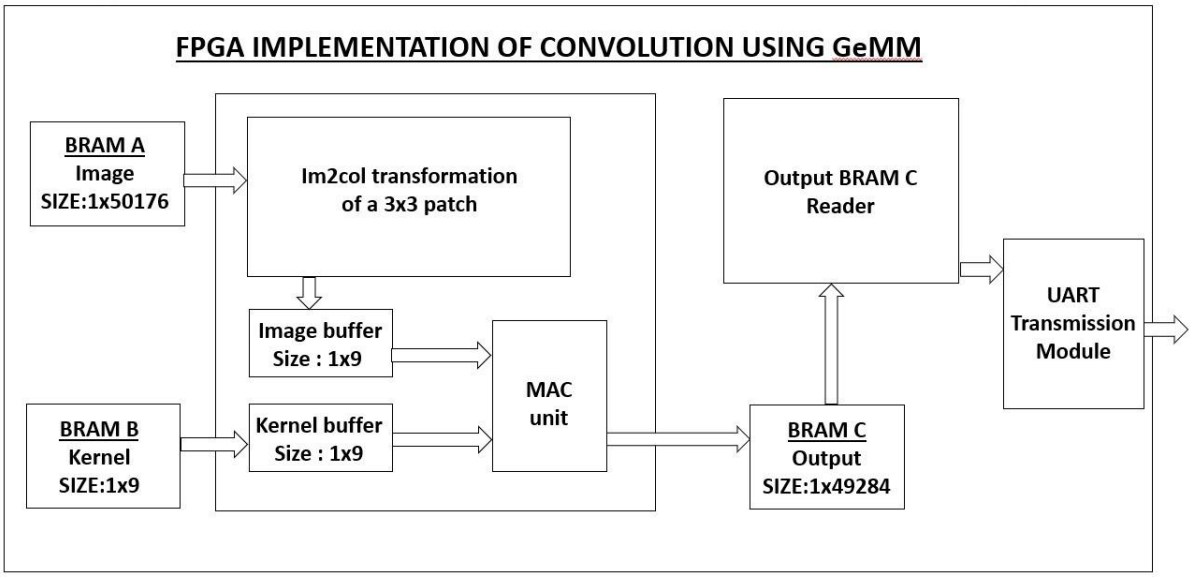

Figure 3.7: PL block

Below is the breakdown of each component in the PL system:

BRAM A

BRAM A is used to store the input image data before processing. The image is stored in this block RAM as a 1 × 50176 array, which is the flattened form of a 224 × 224 image. This storage facilitates easy access during the convolution process.

BRAM B

BRAM B holds the convolution kernel, utilized for feature extraction. The kernel is 1 × 9, equivalent to a flattened 3 × 3 filter. This kernel is utilized in the matrix multiplication step later on.

IM2COL Transformation

This module extracts 3 × 3 patches from the image and rearranges them into a linear array format suitable for GeMM-based convolution.This module processes the patches until entire 49284 patches are processed, considering 224 × 224 image.

Buffers

The image buffer holds the image transformed temporarily by im2col which is extracted patch. The extracted patch is in the form of a 1 × 9 vector to fit the kernel size for element-wise multiplication.

Likewise,the kernel buffer holds the flattened 3 × 3 kernel, ensuring that both the image and kernel data are in the same format before entering the computation stage.

MAC Unit

The MAC unit is responsible for the fundamental computation of the convolution process. It computes the product of corresponding elements of the image buffer and kernel buffer, adds the products together, and produces one output value for every processed image patch. The process is iterated over all patches to accomplish the convolution.

BRAM C

The computed convolution results are stored in BRAM C as a 1 × 49284 array, which corresponds to the output dimensions after applying the convolution operation on the input image.

Output BRAM C Reader

This module reads the stored output data from BRAM C once the convolution is completed. It ensures that the processed data is efficiently retrieved from BRAM C before being sent to the UART Transmission Module.

UART Transmission Module

The UART (Universal Asynchronous Receiver-Transmitter) module is tasked with sending out the processed final image data to an outside system (e.g., computer or display).
Seamless transfer of data to be further analyzed or visualized is ensured by this module.

3.7 FPGA Implementation Of 2D Convolution

Step 1: Loading the Data and Kernel on the Block Memory of FPGA

- The input image, initially in a 224 × 224 matrix format, is reshaped into a onedimensional array of size 50176 × 1 . This transformation allows efficient memory handling and compatibility with the FPGA's processing structure.
- The reshaped image and the kernel (a 3 × 3 filter) are stored in separate Block RAMs (BRAMs) on the FPGA.
- BRAM A stores the input image, while BRAM B stores the kernel, ensuring fast memory access during the computation process.

Step 2: Initializing the Parameters

- The following parameters are initialized to control the convolution process:
  - Number of vertical sliding steps: (*Image matrix size* − *Kernel size* + 1) = (224 − 3 + 1) = 222
  - Number of horizontal sliding steps: (*Image matrix size*−*Kernel size*+1) = (224 − 3 + 1) = 222
  - Kernel size: The 3 × 3 filter used for convolution.
  - Image matrix size: 224 × 224, representing the input image dimensions.
- These values define how the kernel moves across the image during convolution, ensuring complete coverage while maintaining valid boundaries.

Step 3: Defining the FSM States and Data Flow

- The convolution operation is controlled by a Finite State Machine (FSM), which transitions between different states to manage the data flow and computation sequence.
- The FSM consists of the following states:

– **IDLE STATE:**

∗ The FSM starts in this state and remains idle until a start signal is received.

∗ It ensures that the kernel has been completely loaded into BRAM B before proceeding to the next state.

– **SET VERTICAL:**

∗ A vertical pointer moves downwards through the input image, selecting a new row each time for convolution.

∗ This pointer keeps incrementing until it reaches the maximum number of vertical sliding steps (222 steps).

∗ Once the vertical movement is completed for one row, the FSM transitions to the SET _HORIZONTAL state.

– **SET HORIZONTAL:**

∗ A horizontal pointer moves from left to right across the selected row of the image, defining the starting position of each 3×3 patch for convolution.

∗ The pointer increments until it reaches the maximum horizontal sliding steps (222 steps).

∗ When a new position is selected, the FSM transitions to the PATCH PROCESSING state.

– **PATCH PROCESSING:**

∗ The FSM extracts a 3×3 patch from the input image by using the current vertical and horizontal pointer values.

∗ A patch counter is initialized to track the elements inside the 3×3 patch.

∗ The im2col transformation is applied, where the patch elements are rearranged into a 1×9 vector for matrix multiplication.

∗ The processed patch is then fed into the Multiply-Accumulate (MAC) unit for convolution.

- **DELAY (2X):**
  - A delay of two clock cycles is implemented to compensate for the read latency of BRAM.
  - This ensures that data is correctly fetched before proceeding with computations.
- **DATA STORAGE:**
  - The computed convolution result (a single pixel value) is stored in BRAM C at the appropriate position in the output matrix.
  - The FSM then determines whether additional patches need to be processed before moving to the next state.
- **DONE:**
  - Once all vertical and horizontal sliding operations are completed, a done flag is raised.
  - This indicates that the convolution process is finished, and the FSM returns to the IDLE state to wait for the next image.

3.7.1 Summary Of PL Work Flow

1. **Loading Image and Kernel:** The kernel and image data are loaded onto the block memory (BRAM) of the FPGA.
2. **Kernel Buffering:** The kernel is buffered in an intermediate buffer to enable efficient computation.
3. **Im2col Transformation:** Each of the 3×3 patches of the image is being extracted in an im2col manner and being stored in a second intermediate buffer. ¡/enumerate¿.
4. **MAC Calculation:** Multiply-Accumulate (MAC) processing module calculates the convolution between the kernel buffer and the image. The calculated result is saved to the block memory (BRAM C) on the FPGA.

5. **Transmission of Retrieved Output:** BRAM reader module reads out the convolved result from FPGA storage, and UART transmission module transfers the output to an outside system.

Note: The BRAM reader module begins reading the output data from memory only after the convolution process is complete.

This configuration of the PL is highly suitable for accelerating matrix multiplication. The parallelism within the PEs and the high-speed access through BRAM make the system efficient for convolution operations. This design is particularly useful for supporting computationally intensive tasks, making machine learning applications more practical on edge platforms.

3.8 FSM Designed For IPICU Unit

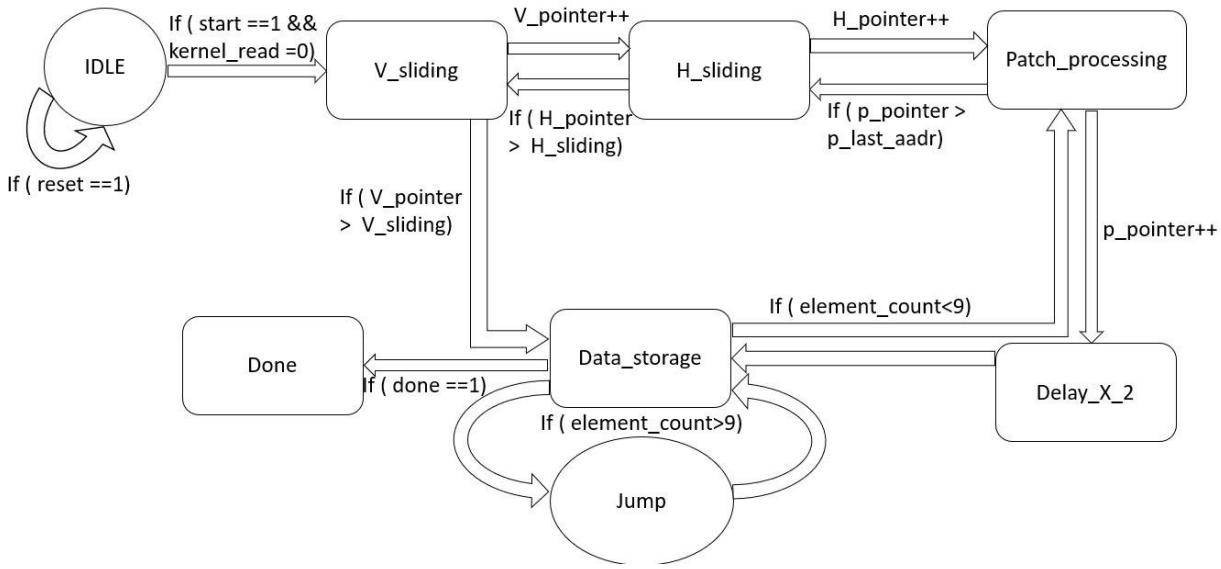

Figure 3.8: FSM

State-by-State Description Of FSM

The given Finite State Machine (FSM) represents a methodology for performing convolution operations using a sliding window approach. Each state transitions based on conditions related to pointer updates, element counts, and completion flags. Below is a detailed breakdown of each state:

1. **IDLE State**

- The FSM starts in the IDLE state.
- It waits for a start signal (start == 1 && kernel read == 0) to begin execution.
- If a reset signal is received (reset == 1), it loops back to itself.
- Once the start condition is met, it transitions to V sliding.

2. **V _sliding (Vertical Sliding)**

- This state manages the vertical movement of the convolution window.
- The V pointer is incremented (V pointer++).
- If the V _pointer exceeds the V sliding limit, it transitions to the Data storage state.
- Otherwise, it moves to the H sliding state.

3. **H sliding (Horizontal Sliding)**

- This state manages the horizontal movement of the convolution window.
- The H pointer is incremented (H pointer++).
- If the H pointer exceeds the H sliding limit, it transitions back to V sliding.
- Otherwise, it proceeds to Patch processing.

4. **Patch processing**

- This state processes each patch of the input matrix for convolution.
- The p pointer is incremented (p pointer++).
- If the p pointer exceeds p last _addr, it returns to H sliding for further processing.
- Otherwise, it moves to Delay X 2.

5. **Delay _ X 2**

- This state introduces a delay to manage data dependencies.
- Once processing is complete, it transitions to Data _storage.

6. **Data storage**

- This state manages storing the computed convolution output.
- If element _count < 9, it proceeds to Jump.
- Otherwise, it directly transitions to Done.

7. **Jump**

- If element _count > 9, it loops back to Data storage for further processing.
- Otherwise, it moves to Done.

8. **Done**

- If done == 1, the process is complete.
- The system remains in this state unless it is reset.

3.8.1 Summary Of Control Flow

1. IDLE: The system waits for start == 1 and kernel _read == 0. If reset == 1, it remains in IDLE.
2. V sliding: Increments V _pointer. If it exceeds the limit, transitions to Data _storage, else moves to H _sliding.
3. H sliding: Increments H pointer. If it exceeds the limit, returns to V _sliding, else proceeds to Patch processing.
4. Patch processing: Increments p pointer. If it exceeds p _last addr, returns to H sliding, else moves to Delay X 2.

5. Delay_X_2: Introduces delay and transitions to Data storage.

6. Data_storage: Stores results. If element_count < 9, moves to Jump, else transitions to Done.

7. Jump: If element_count > 9, loops back to Data storage, else moves to Done.

8. Done: If done == 1, execution is complete. The system remains here unless reset.

3.8.2 Important Features Of The FSM Design

- **Synchronization:** Operations begin only after reaching a stable state in the DELAY state.
- **Data Handling Efficiency:** Address pointers for matrices A and B are dynamically incremented to ensure smooth data flow.
- **Conditional Control:** Iterations are controlled based on the matrix sizes, *M* and *N*.
- **Scalability:** The FSM design is scalable, accommodating different matrix dimensions.

3.9 IPICU Unit (Integrated Patchwise IM2COL Convolutional Unit)

The IPICU unit is designed to efficiently perform patchwise convolution using the IM2COL method, which takes 3x3 image pixel elements sequentially, multiplies them with the kernel, and stores the result in the BRAM-C until the entire 49284 patch (if the image size is 224x224) is processed . The figure 3.9 illustrates the memory architecture and finite state machine (FSM) control flow for this unit.

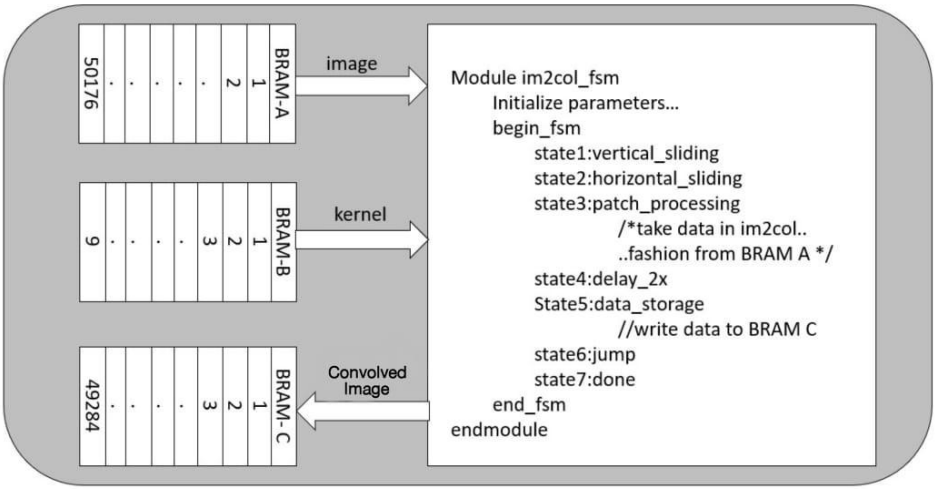

Figure 3.9: IPICU Unit - Integrated Patchwise IM2COL Convolutional Unit

3.9.1 Functional Breakdown

Memory Blocks (BRAM)

- BRAM-A: Stores the input image.
- BRAM-B: Stores the convolution kernel.
- BRAM-C: Stores the output convoluted image.

Finite State Machine (FSM) Operations

The FSM initializes parameters and enters sequential states for convolution processing:

- State 1 - Vertical Sliding: Moves the sliding window vertically across the input image.
- State 2 - Horizontal Sliding: Moves the window horizontally.
- State 3 - Patch Processing: Extracts patches from BRAM-A and arranges them in IM2COL format.
- State 4 - Delay: Introduces a delay to handle data dependencies.
- State 5 - Data Storage: Writes processed convolution results into BRAM-C.

- State 6 - Jump: Ensures that all patches are processed.
- State 7 - Done: Marks the end of execution.

Data Flow

- The image patches are read from BRAM-A and transformed into IM2COL format.
- The kernel values are fetched from BRAM-B for convolution operations.
- The resulting convoluted image is stored in BRAM-C.

3.10 Full Frame and Segmented Frame Processing

3.10.1 Full Frame Processing

In Full Frame processing, convolution on a 224×224 image is performed sequentially by processing each 3×3 image patch, resulting in a total of 49,284 patches. This approach is well-suited for resource-constrained environments as it minimizes on-chip memory usage. However, since each patch must be accessed sequentially, the execution time becomes a
bottleneck.
To improve performance, techniques such as parallel processing and optimized memory access patterns can be employed. By processing multiple patches concurrently or leveraging data reuse strategies, execution time can be significantly reduced while maintaining efficient resource utilization.

3.10.2 Segmented Frame Processing

Segmented Frame Processing is an optimization technique designed to accelerate convolution operations by enabling parallel processing. Instead of sequentially processing each
3×3 patch across the entire 224×224 image, the image is divided into multiple horizontal segments and stored in separate block memory units. This segmentation allows multiple

processing elements (PEs) to operate in parallel, significantly reducing execution time. Each segment is assigned to a dedicated processing unit, which independently performs convolution while ensuring data continuity at segment boundaries. Overlapping pixels are stored to maintain accuracy, and once all segments are processed, the results are merged to reconstruct the full image.

This approach optimizes memory access by distributing data across multiple memory blocks, reducing contention and improving throughput. Compared to full-frame processing, where patches are processed sequentially, segmented frame processing achieves faster execution by leveraging parallelism, making it highly efficient for real-time FPGA-based convolution operations.

3.11 IPICU ENGINE (Integrated Patchwise IM2COL Convolutional ENGINE)

3.11.1 Problem with Convolution using Full-Frame Processing

Performing convolution on a 224 × 224 image sequentially means that each 3 x 3 patch is processed one at a time, leading to high computational latency. Since convolution involves processing each image pixel patch as shown in Figure 3.9 and performing a MAC operation, the computation time increases significantly for large images.

Solution: Segmented Frame Processing(dividing the image into horizontal segments)

To speed up the convolution process, the image is divided into 8 (segment size is varied according to the availability of resources) horizontal segments(or stripes). Each segment consists of a portion of the original image, and all segments are processed in parallel as shown in figure 3.10, significantly reducing the overall computation time.

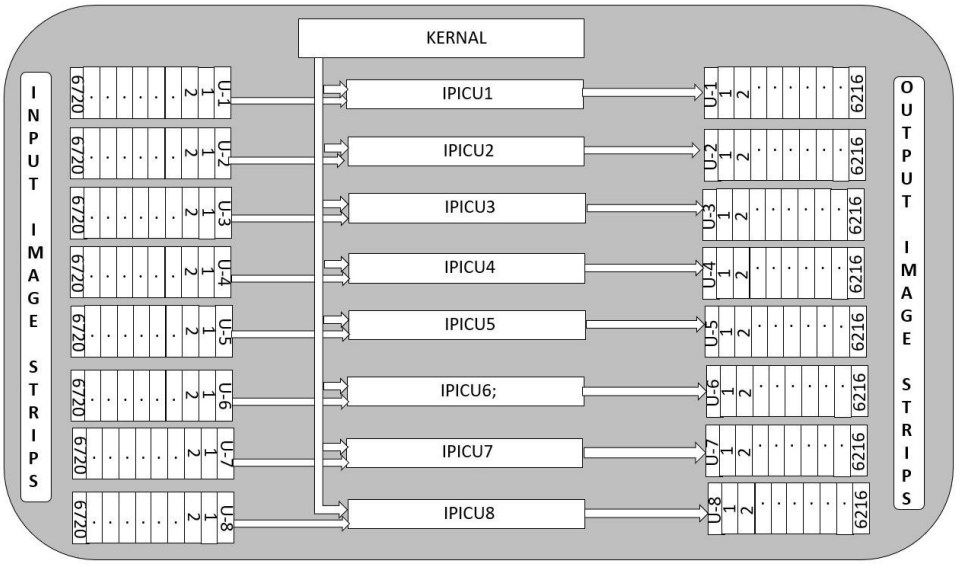

Figure 3.10: IPICU ENGINE - Integrated Patchwise IM2COL Convolutional ENGINE

- **Input Image Segments(or stripes)**: The left side of the diagram represents the input image split into 8 horizontal segments.
- **Kernel Sharing**: A common convolution kernel is shared across all processing units.
- **IPICUs (Image Processing Independent Convolution Units)**: Each segment is assigned to a separate IPICU (Image Processing Independent Convolution Unit), which performs convolution independently.
- **Parallel Processing**: Each IPICU operates on its assigned segment, applying the same convolution kernel in parallel.
- **Output Image Strips**: The processed image segments are merged back to form the final output image.

3.11.2 Benefits of Convolution using Segmented Frame Processing

- **Parallelism**: Since multiple processing units work simultaneously, the execution time is reduced.

- **Efficient Hardware Utilization**: FPGA or specialized hardware accelerators can be used to distribute the workload.

- **Faster Execution**: Instead of processing the entire image sequentially, processing is done in parallel across multiple units.

3.11.3 Considerations

- **Boundary Handling**: Since convolution requires neighboring pixels, a small overlap between adjacent segments may be required.

- **Synchronization**: The final output needs to be merged carefully to maintain continuity across segments.

This approach is useful in FPGA-based CNN acceleration, embedded vision applications, and real-time image processing tasks.

Chapter 4

Results and Discussion

5.1 Results

This work demonstrates the effectiveness of FPGA-based acceleration for convolution operations using the General Matrix Multiplication (GeMM) technique. Implementations on both Boolean FPGA and Zynq UltraScale+ ZCU104 showcase significant improvements in computational efficiency and parallel processing. The ZedBoard FPGA achieved a speedup of order 2.64 , while the Zynq UltraScale+ ZCU104 achieved a speedup of order 3.79 compared to software-only execution. The tiled processing approach enabled efficient handling of multiple convolution kernels in parallel, reducing execution time. Optimization strategies such as DSP block utilization, LUT efficiency, and memory access patterns further enhanced performance. These results confirm the viability of FPGA-based accelerators for deep learning and real-time image processing applications.

Implementation of Multi-Kernel Convolution

In our initial implementation, a single kernel was mapped on the input image, resulting in a single output feature map. To utilize the computational power of the Zynq UltraScale FPGA, we extended our design to accommodate simultaneous processing of six kernels. This optimization enables the system to produce six output feature maps in the same amount of execution time taken by a single kernel, resulting in a highly improved throughput.

Because of the greater resource availability of the UltraScale architecture, this implementation was conducted utilizing Full-Frame Processing to maximize execution efficiency. Segmented Frame Processing, employed in this case, would have resulted in overwhelming BRAM and DSP usage, which makes Full-Frame Processing a better alternative for

multi-kernel execution.

5.3 Comparative Analysis

To rigorously test the effect of our design decisions, we made performance comparisons on various configurations:

1. Zynq UltraScale PS vs. PL: We also compared the performance of Full-Frame Processing in the Processing System (PS) and the Programmable Logic (PL) of Zynq UltraScale. Since the PS lacks hardware-level parallelism, this comparison highlights the acceleration achieved through FPGA-based computation.

2. Boolean FPGA: Full-Frame Processing vs. Segmented Frame Processing, comparing execution time and resource usage in a low-resource setting.

3. Zynq UltraScale FPGA: Full-Frame Processing vs. Segmented Frame Processing Comparison, utilizing the increased resources to compare execution time benefits.

4. Zynq UltraScale FPGA (Multi-Kernel Execution): 1-kernel vs. 6-kernel Full-Frame Processing Comparison, illustrating the efficiency and scalability of multi-kernel convolution on FPGA.

5.3.1 Comparison of 2D convolution in Software (PS) and Hardware (PL)

In order to compare the performance benefit of hardware acceleration, 2D convolution operation was performed and compared on both the Processing System (PS) and Programmable Logic (PL) of Zynq-based FPGAs. As it is not possible to perform segmented frame processing in PS because of its sequential model of execution and inability to perform parallelization, this comparison was made with the full-frame processing method in both PS and PL.

Key findings

1. Execution on ZedBoard PS

- The convolution operation was executed purely in C on the ZedBoard PS, operating at 100 MHz, resulting in an execution time of 52 ms.
- The high execution time was primarily due to instruction fetching overhead, sequential computation, and memory access bottlenecks.

2. Execution on Zynq Ultrascale+ PS

- The same C-based implementation was executed on the Zynq UltraScale PS at 125 MHz, requiring 7,479,565 clock cycles, corresponding to an execution time of 59.84 ms.
- Despite the higher clock frequency, the execution time remained significant, further demonstrating the inherent limitations of processor-based computation for intensive operations like convolution.

3. Execution on Zynq Based PL (Hardware Acceleration)

- The implementation using FPGA in the PL part showed a significant decrease in execution time.
- At 100 MHz, the convolution operation took 1,971,804 clock cycles with an execution time of 19.72 ms.
- At 125 MHz, the time of execution was again cut down to 15.77 ms, indicating the performance benefit through parallel processing, pipelined execution, and optimized memory management in FPGA-based computation.

The comparison simply proves the superior effectiveness of hardware acceleration. The implementation using PL achieved a speedup of 3.28× compared to PS-based execution on Zynq UltraScale and a speedup of 2.64× compared to ZedBoard PS implementation. The outcomes strongly underscore the important function of FPGA-based parallelism and efficient handling of data to expedite computationally intensive operations like convolution.

By utilizing full-frame processing, we made a straightforward and equitable comparison between PS and PL since segmented frame processing is not feasible in PS because it cannot parallelize computations among the various segments of the image. The results validate that FPGA-based acceleration is necessary for real-time performance in highresolution convolution operations.

5.3.2 Comparative Analysis of Segmented Frame Processing vs. Full-Frame Processing on Boolean FPGA

RESOURCE METRIC	Segmented Frame (8 x Segments)	Full - Frame
TOTAL ON-CHIP POWER	0.225 W	0.093 W
DYNAMIC POWER	0.150 W	0.019 W
STATIC POWER	0.075 W	0.074 W
LOGIC UTILIZATION (LUTs)(52160)	3631	458
BLOCK RAM(75)	56	48.5
DSP USAGE(120)	88	11
CLOCK FREQUENCY	100MHz	100MHz
EXECUTION TIME	1.8 ms	19.7180 ms
REQUIRED CLOCK CYCLE	180000	1971804

Table 5.1: Comparative Analysis of Segmented Frame Processing vs. Full-Frame Processing on Boolean FPGA

Key Findings

The results highlight a significant tradeoff between execution time and resource utilization when using segmented frame processing instead of full-frame processing for 2D convolution on a Boolean FPGA.

1. Execution Performance:

    - Segmented frame processing attains an execution time of 1.8 ms, more than 10× faster than full-frame processing (19.7180 ms).

- The decreased processing time is caused by the image being divided into many small horizontal pieces, enabling good parallelization and quick processing.

2. Resource Utilization:

   - The segmented method takes 3631 LUTs, a drastic rise over 458 LUTs for full-frame processing.
   - DSP utilization is also much increased (88 DSPs compared with 11 DSPs), implying that parallel implementation is paid for in terms of extra arithmetic hardware.
   - Conversely, Block RAM usage is higher at 56, versus 48.5 in full-frame processing.

3. Power Consumption:

   - Total on-chip power consumption is 0.225 W for segmented frame processing, which is more than double the power used in full-frame processing (0.093 W).
   - This increase is primarily due to the higher dynamic power consumption (0.150 W vs. 0.019 W) caused by increased switching activity in the FPGA fabric.

Segmented frame processing on a Boolean FPGA substantially reduces execution time, making it highly beneficial for real-time applications. However, this speed improvement comes at the expense of higher resource utilization and power consumption. If resource constraints are critical, full-frame processing remains a viable alternative despite the longer execution time.

5.3.3 Comparative Analysis of Segmented Frame Processing vs. Full-Frame Processing on Zynq UltraScale + MPSoC

RESOURCE METRIC	Segmented Frame (16 x Segments)	Full - Frame
TOTAL ON-CHIP POWER	0.842W	0.62W

DYNAMIC POWER	0.226W	0.006W
STATIC POWER	0.616W	0.615W
LOGIC UTILIZATION (LUTs)(230400)	6089	1170
BLOCK RAM(312)	64	46
DSP USAGE(1728)	176	11
CLOCK FREQUENCY	125MHz	125MHz
EXECUTION TIME	0.9947ms	15.7744ms
REQUIRED CLOCK CYCLE	124348	1971804

Table 5.2: Comparative Analysis of Segmented Frame Processing vs. Full-Frame Processing on Zynq UltraScale + MPSoC

The results reveal a significant tradeoff between execution time and resource utilization when using segmented frame processing instead of full-frame processing for 2D convolution on a Zynq UltraScale + MPSoC.

1. Execution Performance:

    - Segmented frame processing achieves an execution time of 0.9947 ms, which is over 15× faster than full-frame processing (15.7744 ms).
    - The faster execution is attributed to the efficient segmentation of the image into 16 strips, allowing parallelization and reduced data dependency.

2. Resource Utilization:

    - The segmented approach uses 6089 LUTs, a significant increase compared to 1170 LUTs in full-frame processing.
    - DSP usage rises to 176 DSPs from 11 DSPs, emphasizing the computational overhead of the parallelized approach.

- Block RAM consumption is also higher in segmented processing (64 blocks vs. 46 blocks), as additional memory is required to store the results of each segment.

3. Power Consumption:

   - Segmented frame processing consumes a total on-chip power of 0.842 W, which is significantly higher than full-frame processing (0.62 W).
   - The dynamic power consumption is particularly notable, with segmented processing using 0.226 W compared to just 0.006 W in full-frame processing. This increase is due to the elevated switching activity from multiple concurrent op-
     erations.
   - Static power remains almost identical, with segmented processing at 0.616 W and full-frame processing at 0.615 W.

Segmented frame processing on a Zynq UltraScale + MPSoC provides a significant reduction in execution time and is therefore very beneficial for real-time applications. The accelerated processing, however, is at the expense of higher resource usage and power consumption. In resource-constrained environments where energy efficiency is a top priority, full-frame processing is still a viable option despite its longer execution time.

5.3.4 Comparative Analysis of Single-Kernel vs. Multi-Kernel Convolution on Zynq Ultrascale+ MPSoC

RESOURCE METRIC	Multi-Kernel (6 kernels)	Single-Kernel (1 kernel)
TOTAL ON-CHIP POWER	0.792W	0.62W
DYNAMIC POWER	0.175W	0.006W
STATIC POWER	0.617W (78%)	0.615W
LOGIC UTILIZATION (LUTs)(230400)	1549	1170
BLOCK RAM(312)	213.5	46

DSP USAGE(1728)	56	11
CLOCK FREQUENCY	125MHz	125MHz
EXECUTION TIME	15.7744ms	15.7744ms
REQUIRED CLOCK CYCLE	1971804	1971804

Table 5.3: Comparative Analysis of Single-Kernel vs. Multi-Kernel Convolution

This analysis compares the trade-offs between multi-kernel convolution (6 kernels) and single-kernel convolution (1 kernel) on a Zynq Ultrascale+ MPSoC. The key focus areas are execution time, resource utilization, and power consumption.

1. Execution Performance:
    - Single-Kernel Execution Time: 15.7744 ms
    - Multi-Kernel Execution Time: 15.7743 ms
    - Required Clock Cycles: Identical at 1,971,804 cycles for both implementations
2. Resource Utilization:
    - LUT Usage: Multi-kernel requires 1549 LUTs, which is higher than 1170 LUTs in single-kernel.
    - DSP Usage: Multi-kernel significantly increases DSP consumption to 56, compared to only 11 in single-kernel.
    - Block RAM Usage: Multi-kernel uses 213.5, whereas single-kernel only requires 46, indicating a large increase in memory demand.
3. Power Consumption:
    - Total Power Consumption: Multi-kernel consumes 0.792 W, which is higher than the 0.62 W of single-kernel.
    - Dynamic Power: Multi-kernel requires 0.175 W, while single-kernel only uses 0.006 W, showing a massive increase due to increased switching activity.

- Static Power: Nearly the same, with 0.617 W for multi-kernel and 0.615 W for single-kernel.

Despite a significant increase in resource usage (DSPs, LUTs, and memory), the execution time for multi-kernel convolution is nearly the same as single-kernel convolution. If power efficiency and lower resource consumption are priorities, the single kernel is the better option. If the design goal is to utilize available FPGA resources (e.g., DSPs) with increased throughput, then multi-kernel can still be useful.

5.4 Hardware Platforms Used For Implementation

For the hardware implementation of convolution, we utilized two FPGA platforms: the Boolean Board and the Zynq UltraScale+ ZCU104 Board. These boards were selected based on their computational capabilities, flexibility, and suitability for accelerating image processing tasks.

5.4.1 Boolean FPGA Board

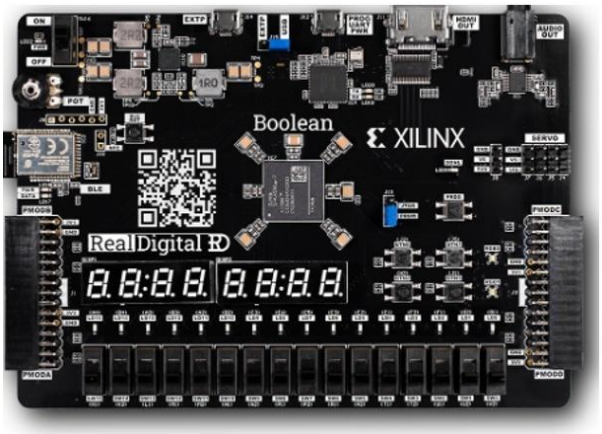

Figure 5.8: Boolean FPGA

The Boolean Board as shown in figure 5.8, is a versatile FPGA development platform designed for prototyping and implementing digital logic circuits. It provides essential resources for handling computational tasks, making it suitable for testing convolution operations at a fundamental level.

RESOURCE AVAILABLITY	SPECIFICATION
LOGIC CELLS (LUTs)	52160

BLOCK RAM TILES (36 Kb each)	75
DSP SLICES	120
CLOCK FREQUENCY	100MHz
PROCESSING SYSTEM	NO
PROGRAMMABLE LOGIC	SPARTAN 7 (XC7S50)

Table 5.4: Boolean Board

Part Number : XC7S50CSGA324-2

5.4.2 Zynq UltraScale+ ZCU104 FPGA Board

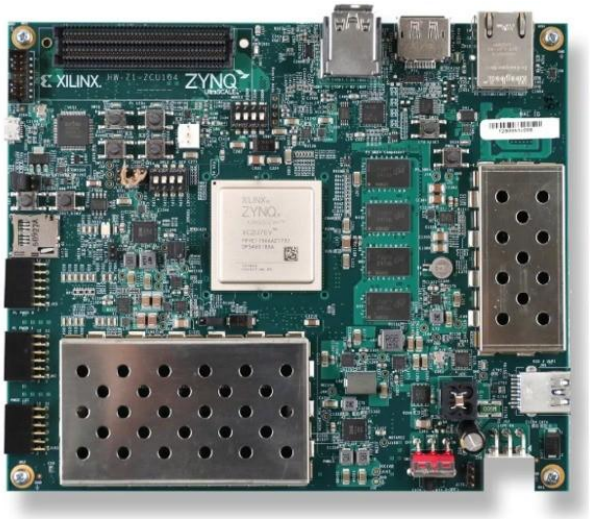

Figure 5.9: Zynq ultrascale+ZCU104 FPGA

The ZCU104 is a powerful FPGA board featuring Xilinx's Zynq UltraScale+ MPSoC, which integrates an FPGA fabric with an ARM-based processing system. It offers high-performance DSP slices, extensive memory bandwidth, and hardware acceleration capabilities, making it ideal for real-time convolution processing. The presence of programmable logic and embedded processing units allows for efficient parallel execution of computational tasks.

RESOURCE AVAILABLITY	**SPECIFICATION**
LOGIC CELLS (LUTs)	230400

BLOCK RAM TILES (36 Kb each)	312
DSP SLICES	1728
CLOCK FREQUENCY	125MHz
PROCESSING SYSTEM	quad-core ARM® Cortex™-A53
PROGRAMMABLE LOGIC	ZYNQ ULTRASCALE+ (16nm FinFET)

Table 5.5: Zynq UltraScale+ ZCU104 Board

Part Number: XCZU7EV-FFVC1156-2-E

These boards were used to evaluate the efficiency, performance, and feasibility of implementing convolution operations in hardware, ensuring optimized execution compared to traditional CPU-based processing

Chapter 6

Conclusions & Future Scope

6.1 Conclusion

The comparative study of segmented frame processing and full-frame processing on various FPGA platforms exposes a basic tradeoff between execution speed, resource usage, and power draw.

On Boolean FPGA and the Zynq UltraScale+ ZCU104, segmented frame processing dramatically enhances performance, reducing execution time by a factor of 10 to 15 compared to full-frame processing. This is mainly because parallel processing of smaller segments of images minimizes data dependency and maximizes computational throughput. Although this enhances the performance, the overhead cost is a significant increase in resource utilization. LUTs, DSP, and block RAM usage all come out to be significantly higher in segmented processing, as parallelism increases complexity. Also,

the dynamic power consumption is increased considerably because of greater switching activity and hence an increase in overall power usage is seen.

These findings are further supported by the comparison of single-kernel versus multikernel convolution. Multi-kernel processing indeed enhances the computational efficiency due to the workload being distributed across several processing units. However, it increases power and resource consumption. The trade-off is clearly evident here—where execution is nearly the same, LUT and DSP utilization is higher, thereby affecting the hardware efficiency in totality.

Finally, between segmented and full-frame processing and single- vs. multi-kernel architectures, the specific application requirements determine the choice. For real-time processing applications where performance is a priority, segmented and multi-kernel schemes are more suitable in spite of their greater resource needs. However, for energy-constrained environments where minimizing energy usage is a priority, full-frame and single-kernel processing are still valid options, trading off execution time with minimum power usage.

6.2 Future Scope

Although the project's developments demonstrate the potential of FPGA-based hardware acceleration for convolution operations, there are still a number of directions that need to be investigated and improved. Future research can concentrate on improving the hardware accelerator's scalability and efficiency through memory access patterns, data movement optimization, and computational bottleneck reduction.

Furthermore, by integrating specific floating-point arithmetic units into the FPGA fabric, floating-point support can be incorporated, allowing for more precise calculations and enhancing the numerical stability of convolution operations—particularly for deep learning and scientific applications.

Extending support for bigger and more intricate convolution kernels is another crucial area of development that will allow deep neural networks (DNNs) to be seamlessly accelerated on FPGA platforms. High-bandwidth memory architectures and effective data

reuse techniques could be used to reduce memory latency and increase computational throughput.

Furthermore, contrasting the hardware-accelerated method with cutting-edge GPUbased accelerators can offer more profound understanding of trade-offs between resource usage, performance, and energy efficiency. Performance can be further optimized for a variety of workloads by investigating hybrid architectures that combine CPU/GPU processing with FPGA acceleration.

Last but not least, the accelerator's practical implementation and validation in edge computing, autonomous systems, and embedded AI can yield insightful input for improving the design. FPGA-based convolution accelerators have enormous potential to transform high-performance computing for deep learning and image processing tasks with ongoing advancements.