Hardware acceleration of Computer Vision Algorithms using Field Programmable Gate Arrays (FPGA)

A Design Project Report

Presented to the School of Electrical and Computer Engineering of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering, Electrical and Computer Engineering

Submitted by

Dengyu Tu

MEng Field Advisor: Van Hunter Adams

Degree Date: December 2024

Abstract

Master of Engineering Program

School of Electrical and Computer Engineering

Cornell University

Design Project Report

Project Title: Hardware acceleration of Computer Vision Algorithms using Field Programmable Gate Arrays (FPGA)

Author: Dengyu Tu

Abstract: This project focuses on leveraging Field-Programmable Gate Arrays (FPGAs) to accelerate the computational processes of Convolutional Neural Networks (CNNs), particularly the convolutional filter operations applied to images. The primary objective is to significantly enhance the processing speed of CNNs, thereby improving the performance and applicability of computer vision algorithms in real-time and resource-constrained environments. The project involves implementing various convolutional filters on FPGAs and demonstrating their effectiveness on a range of images. Additionally, a comparative analysis will be conducted to measure the performance differences between FPGA-based and traditional CPU-based CNN computations. The findings aim to underscore the advantages of using FPGAs for hardware acceleration in computer vision tasks, showcasing improvements in speed, efficiency, and scalability. By optimizing CNN operations with FPGAs, this work aspires to contribute to the advancement of high-speed, real-time computer vision applications.

Executive Summary

Field-Programmable Gate Arrays (FPGAs) are integrated circuits that can be configured by the user after manufacturing. Unlike fixed-function devices, FPGAs provide a versatile platform for implementing custom hardware designs tailored to specific applications. They are particularly well-suited for tasks requiring high-speed processing and parallel computation, making them ideal for accelerating the computationally intensive operations found in CNNs. This project investigates the use of Field-Programmable Gate Arrays (FPGAs) to accelerate convolutional operations in Convolutional Neural Networks (CNNs), specifically targeting computer vision algorithms. The primary goal is to enhance the computational speed and efficiency of CNNs, making them more suitable for real-time applications and resource-constrained environments.

The project spanned two semesters, each focusing on different aspects of the work. During the Fall 2023 semester, the primary focus was on building a strong foundation. With no prior experience in FPGAs or Verilog, I dedicated this semester to learning the basics of the Verilog hardware description language. I implemented fundamental digital components such as multipliers, counters, decoders, encoders, and asynchronous circuits. These projects were simulated using ModelSim to verify their functionality. This phase provided me with a solid foundation in Verilog, essential for more complex implementations in the following semester.

In the Spring 2024 semester, the focus shifted to implementation and optimization. I began by studying the System-on-Chip (SoC) data flow and learning how to transfer data between the FPGA and a Micro SD card. Using Python, I preprocessed several images and displayed them on a VGA display. The bulk of the semester was spent developing the matrix multiplication module and image buffer, key components of the convolution operation. This involved writing complex control logic and ensuring accurate data handling. After validating the convolution algorithm, I focused on parallelizing the design to maximize speedup. This involved addressing issues related to the M10K memory blocks and ensuring synchronized operation of the parallel compute modules. Once the design was optimized, I measured and documented the processing times of various designs, comparing FPGA-based acceleration with traditional CPU-based computations.

The project achieved several key accomplishments. The FPGA implementation demonstrated significant speedups in convolution operations compared to CPU-based calculations, showcasing the potential of FPGAs in accelerating computer vision tasks. By increasing parallelism and optimizing data handling, the project achieved notable improvements in processing speed. By providing a comprehensive understanding of FPGA hardware, Verilog HDL, and the principles of hardware acceleration, this project underscores the transformative impact of FPGAs in computational tasks, promising continued advancements in artificial intelligence and machine learning technologies.

Introduction

Computer vision algorithms are integral to a variety of modern technologies, including Face ID, autonomous driving systems like Autopilot, and cashier-less stores like Amazon Go. These algorithms are capable of classifying and localizing different objects within an environment, as illustrated in Figure 1, where computer vision systems identify and label various objects on a road. The backbone of these advanced applications is the Convolutional Neural Network (CNN), a deep learning model that excels in processing visual data.

As the demand for sophisticated AI applications continues to surge, the efficiency and performance of Neural Networks, particularly CNNs, have become crucial factors. One of the primary challenges in this field is the need to accelerate neural network computations to meet the requirements of real-time processing. Traditional approaches that rely on general-purpose processors (CPUs) often fall short in achieving the necessary speed and efficiency due to their inherent architectural limitations.

Figure 1. Example of Computer Vision Algorithms. Source: AUGMENTED A.I.

A complete Convolutional Neural Network (CNN) is depicted in Figure 2. CNNs are composed of multiple layers, each performing distinct types of calculations. These layers typically include the Convolution Layer, ReLU (Rectified Linear Unit), Pooling Layer, Flatten Layer, and Fully Connected Layer. This project specifically focuses on optimizing the Convolution Layer, which is the most computationally intensive part of the network.

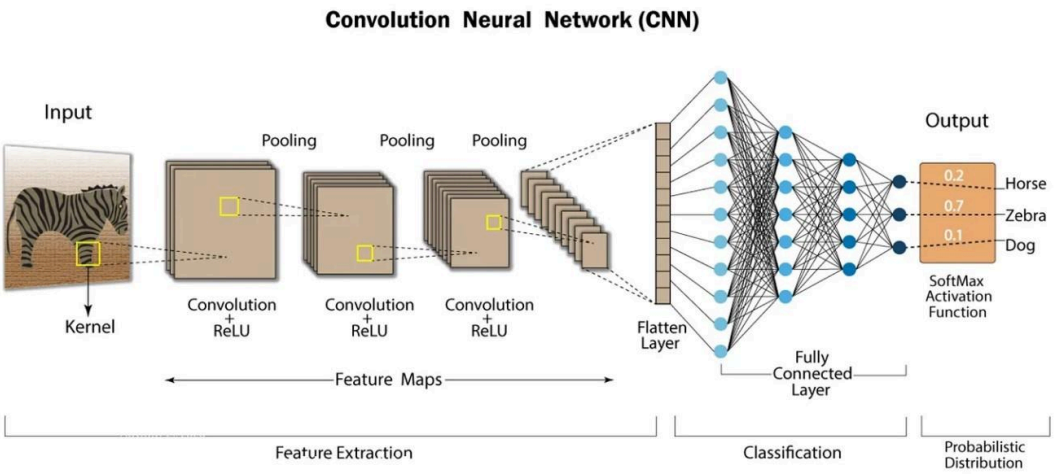

Figure 2. Different Layers of the Convolutional Neural Network. Source: Kh. Nafizul Haque

The convolution operation on an image is illustrated in Figure 3, where a filter (represented by a 3x3 blue matrix) is applied to the image. During this operation, the filter slides over the image, multiplying its values with corresponding pixel values in a 3x3 region of the image and summing the results to produce a single output value. This process is repeated as the filter moves across the entire image, generating a feature map. This step is commonly referred to as feature extraction because different filters can be used to extract various features from the image, such as edges, textures, and patterns.

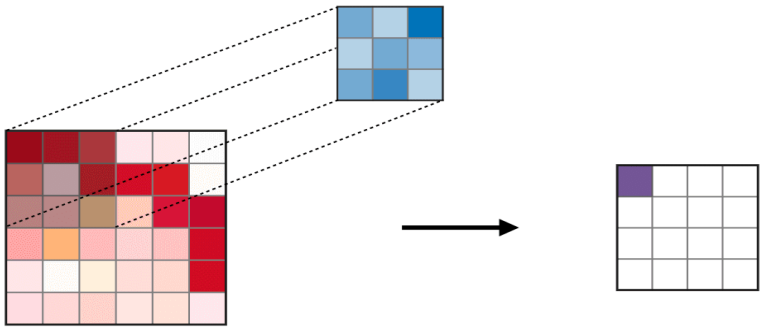

Figure 3. Convolution Calculation. Source: Google.

For example, as shown in Figure 4, applying vertical edge filters and horizontal edge filters to an image can highlight the vertical and horizontal features, respectively. These extracted features are crucial for the subsequent layers in the CNN, which use them to perform higher-level tasks such as object detection and classification.

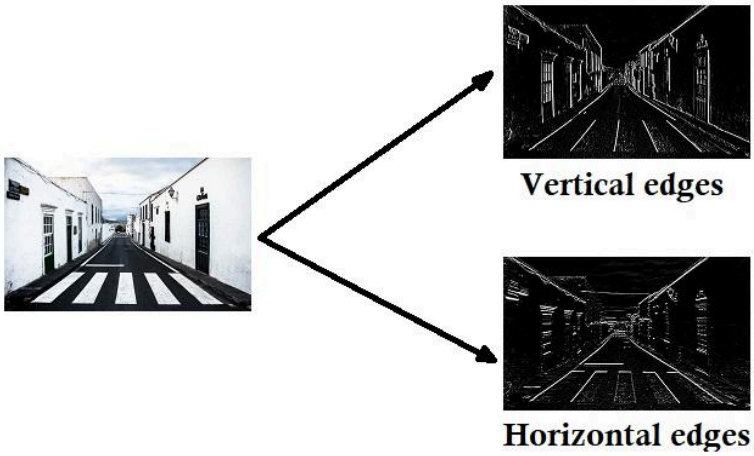

Figure 4. Edge detection. Source: https://oriont.net/posts/gan-intro.

Accelerating neural network calculations poses several challenges, including:

1. Computational Intensity: Neural networks involve complex mathematical operations, and as models grow in size and complexity, the computational workload increases significantly.
2. Real-Time Processing: Applications such as autonomous vehicles and live video analysis require real-time processing, demanding rapid inference capabilities that may exceed the capabilities of traditional computing architectures.
3. Energy Efficiency: Efficient utilization of computational resources is crucial, especially in resource-constrained environments or for applications running on battery-powered devices.

Field-Programmable Gate Arrays (FPGAs) emerge as a potential solution to address these challenges. FPGAs offer unique advantages for accelerating neural network calculations:

1. Parallel Processing: FPGAs inherently support parallel processing, allowing for the simultaneous execution of multiple operations. This parallelism aligns well with the inherently parallel nature of neural network computations.
2. Customization: FPGAs can be customized to implement specific hardware architectures tailored to the requirements of neural network operations. This customization enables the creation of highly optimized, application-specific hardware.
3. Low Latency: The parallel and customizable nature of FPGA implementations enables low-latency processing, making them suitable for real-time applications.
4. Energy Efficiency: FPGAs can achieve high energy efficiency by efficiently implementing parallel data pipelines and optimizing for specific neural network operations.

The primary reason for choosing to accelerate CNNs using FPGAs is the inherent uniformity of filters across the various layers of CNNs. This uniformity facilitates straightforward parallelization, making FPGA acceleration a highly efficient solution. As illustrated in Figure 5, the image can be divided into different sections, with each section processed in parallel using the same set of filters. This parallel processing capability significantly reduces the computation time required for the convolution operations,

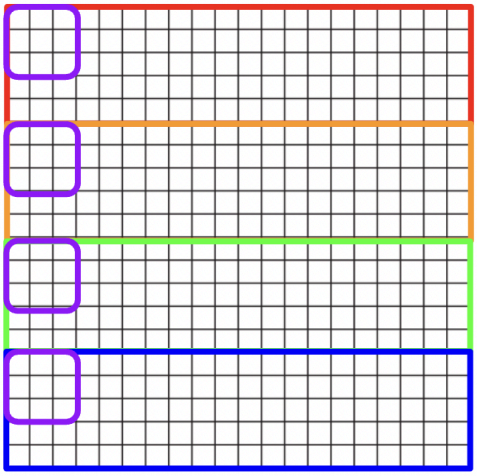

Figure 5. Image Segmentation for parallelization.

By leveraging these advantages, FPGA-based solutions can significantly accelerate neural network calculations, providing a pathway to address the challenges associated with computational intensity, real-time processing, and energy efficiency in AI applications. This project focuses specifically on accelerating Convolutional Neural Networks using FPGA, aiming to showcase the practical benefits of this approach.

This project serves as a valuable learning opportunity, especially considering the absence of prior experience with FPGA and Verilog. The exploration of FPGA design can contribute to a deeper understanding of hardware acceleration and potentially serve as a prototype for future Application-Specific Integrated Circuit (ASIC) designs.

Implementation

The hardware platform utilized for this project is the Intel DE1-SoC development board, which features a Cyclone V FPGA chip integrated with an ARM Cortex-A9 processor, as illustrated in Figure 6. This hybrid architecture provides a versatile environment where the ARM processor can handle high-level control tasks and Linux-based operations, while the FPGA can be dedicated to accelerating the convolution computations of the CNN.

To leverage the dual capabilities of the DE1-SoC, a Linux image was installed on the ARM Cortex-A9 processor. The Linux system facilitates the transmission of image data and orchestrates the overall algorithm control, while the FPGA is tasked with executing the convolution operations.

The implementation workflow is as follows:

1. **Data Storage and Transfer:** The pixel data of the input image is initially stored on a Micro SD card. This data is then transferred from the Micro SD card to the M10K memory blocks within the FPGA.

2. **Image Display via VGA:** A hardware-coded VGA driver within the FPGA reads the image data directly from the M10K memory and displays it on a VGA screen. This step allows for visual validation of the image data prior to and after processing.

3. **Convolution Processing:** The convolution filter, defined within the FPGA logic, processes the image data stored in the M10K memory. The filtered data, which represents the convolution results, is then written back to the M10K memory.

4. **Updated Image Display:** The VGA driver reads the updated pixel data from the M10K memory and displays the processed image on the VGA screen. This real-time display of the convolution results enables immediate visual comparison with the original image data.

In typical CNN implementations, a VGA display is not required. However, for this project, the VGA display serves as a valuable tool for validating the convolution calculations performed by the FPGA. By comparing the FPGA-processed image displayed on the VGA screen with the results obtained from CPU-based calculations, we can verify the accuracy and performance gains of the FPGA implementation.

The overall data flow, encompassing data storage, transfer, processing, and display, is depicted in Figure 7. This flowchart highlights the interaction between the various components inside and outside FPGA and stages of the implementation, ensuring a comprehensive understanding of the system operation.

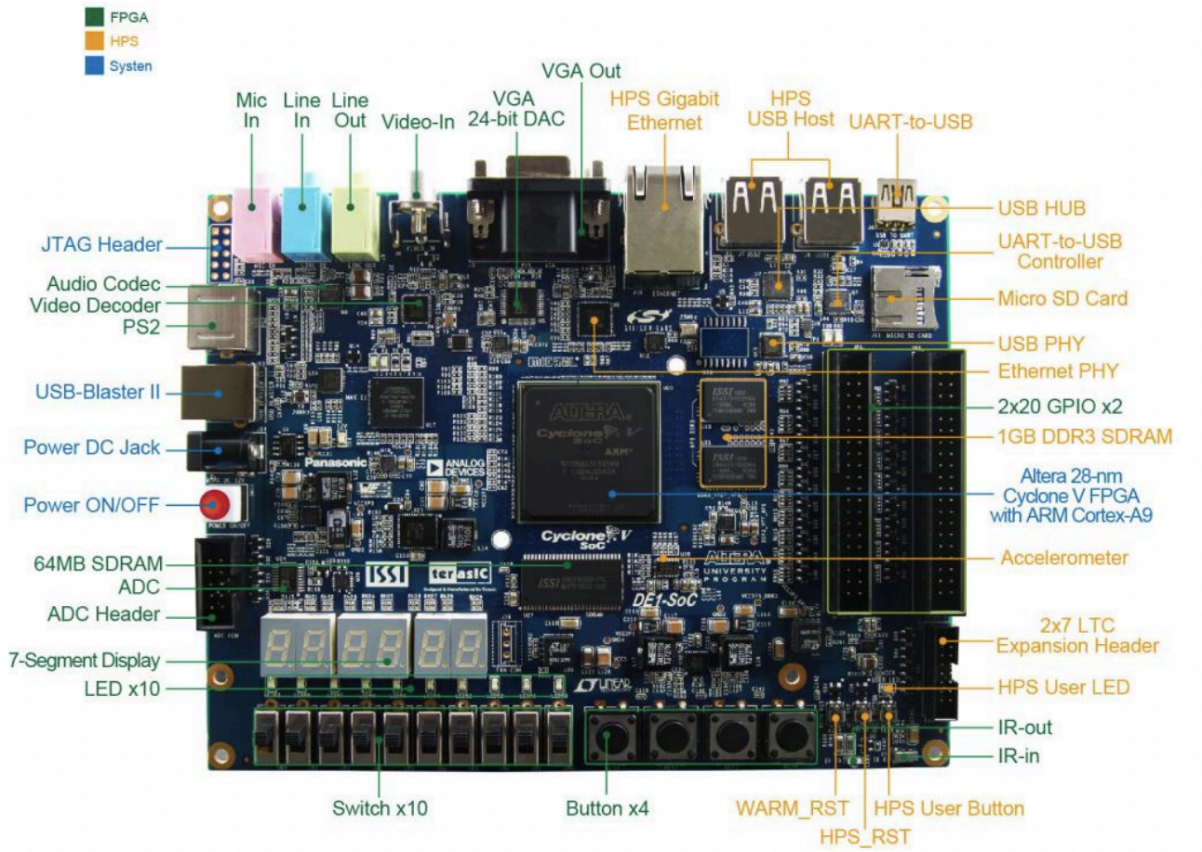

Figure 6. DE1-SoC board. Source: DE1-SoC User Manual.

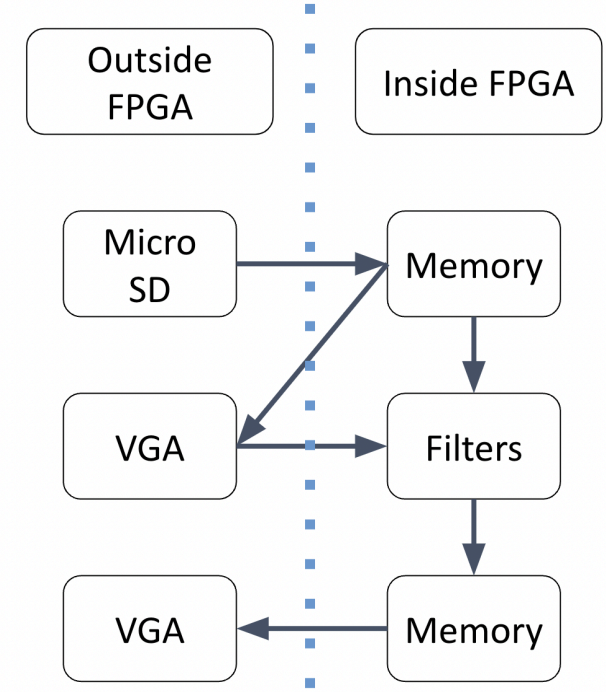

Figure 7. Data flow diagram

The image data is stored in M10k memory because the M10k is built inside the FPGA which costs less time to read the data. According to the datasheet, the FPGA has a total of 3900 kilobits of M10K memory. Given that the VGA display resolution is 640x480 pixels, this memory constraint allows us approximately 12 bits per pixel. To optimize memory usage while maintaining color representation, we chose to represent each pixel using 8 bits. In this 8-bit format, the most significant three bits correspond to the red color channel, the middle three bits to the green color channel, and the least significant two bits to the blue color channel as shown in Figure 8. The pixel data from the Micro SD card is pre-processed using Python to conform to this 8-bit format before being loaded into the FPGA.

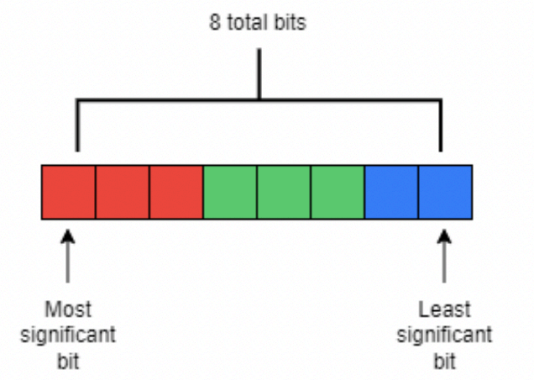

Figure 8, 8-bit color representation of image.

To preserve the original dimensions of the image during convolution operations, we applied zero padding to the edges. As described in the introduction, convolution involves sweeping a filter across the image, which reduces the image dimensions. For instance, convolving a 640x480 image with a 3x3 filter would result in a 638x478 image. To counteract this reduction, zero padding was added, effectively expanding the image size to 642x482 by wrapping zero value around the image before convolution as shown in Figure 9. This padding ensures that the output image retains the original dimensions of 640x480.

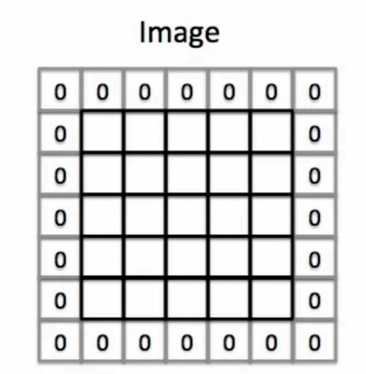

Figure 9, zero padding.

However, the zero padding is not stored in the M10K memory to avoid affecting the VGA display output. Instead, the padding is dynamically applied only during the convolution process, particularly when calculations involve the image edges.

Matrix Compute Module

A custom matrix compute module was designed to handle the convolution operations. This module features nine inputs and one output, corresponding to the 3x3 filter and the resulting convolved pixel value. The inputs are arranged to match the spatial order of the corresponding pixels in the image.

During each clock cycle, the matrix compute module performs the following steps:

1. **Input Multiplication:** Each of the nine inputs is multiplied by the corresponding hardcoded filter value.
2. **Summation:** The products are summed to produce the convolution result for the current pixel.
3. **Clamping:** The resulting value is clamped to the range of 0 to 255 to maintain color consistency and prevent data overflow.

Image Buffer

Due to the constraint of M10k memory, at each cycle, we can only read one value from M10k. However, at each convolution calculation, we utilized nine values each cycle. This big difference makes pipelines really hard and challenging. In addition, after each convolution, we cannot store the calculated values back to the M10k memory directly because the old value needs to be used for other calculations. We can only store the value back when it is no more used. The solution to these two problems is an image buffer as shown in the Figure 10. The image buffer is a long array of registers where some part of the image is read and stored beforehand so that we do not need to access them while calculating the value. I chose the image buffer to be two rows of image plus three extra pixel values. So that, after the whole buffer is filled with pixel data, we can start the calculation with the first 3*3 matrix. After the calculation is done, the value is stored back to the M10k and the next pixel data is read from M10k. At the next clock cycle, a new 3*3 matrix is formed and the convolution can be calculated again. With this method, we can pipeline the convolution calculation with the M10k read and write operation so that we need have to wait nine cycles after each convolution. In addition, we can store the calculated result back to M10k memory immediately without worrying if the old pixel data is lost. The trade off of this method is huge registers resources needed and big overhead before the convolution calculation. Luckily, the FPGA provides enough registers for parallelization and the overhead

can be decreased if we store the value in the imagebuffer when we store the image data from the Micro SD card to the M10k memory.

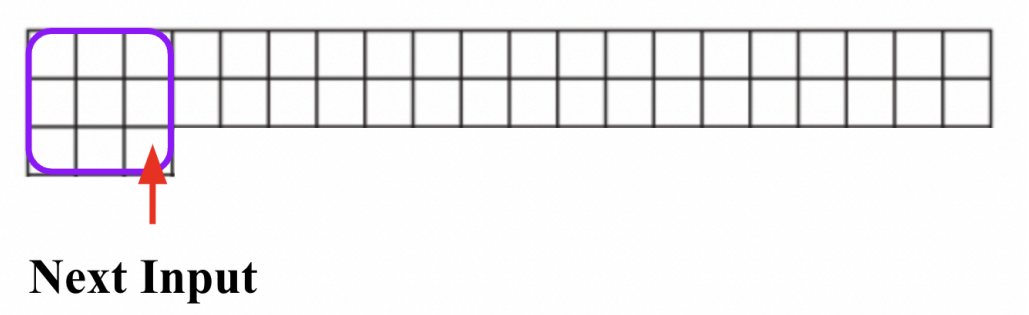

Figure 10, Image Buffer.

Handling Edge Cases in Parallelization

Once the compute module and image buffer configurations are established, parallelization becomes a straightforward process. Depending on the desired level of parallelization, multiple identical instances of M10K memory, matrix computation modules, and image buffers need to be implemented. Additionally, a large multiplexer is required before the VGA driver to select the appropriate M10K memory based on the position on the VGA display.

However, a challenge arises when dealing with pixel values at the edges of the parallelized M10K memories, as illustrated in Figure 11. When perfectly separating the image into distinct segments, calculating the 3x3 matrix at the edge of two adjacent M10K memories requires data from both memories. This inter-segment communication introduces significant overhead and complexity, making it difficult to manage.

To mitigate this challenge, the size of each M10K memory block is increased to ensure that adjacent blocks overlap by two rows, as depicted in Figure 12. This overlap allows the compute modules to access neighboring pixel values seamlessly, eliminating the need for complex inter-segment communication.

By increasing the size of each M10K memory block and allowing for overlap between adjacent segments, the parallelization process becomes more efficient and easier to manage. This approach reduces overhead and ensures smooth communication between compute modules, enabling seamless parallel execution of convolution calculations across the entire image.

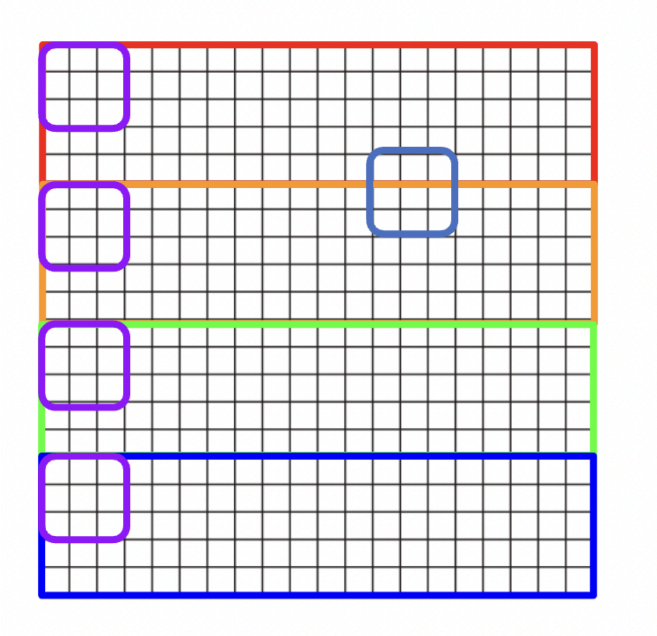

Figure 11. Calculation at the edges of the parallelized M10K memories

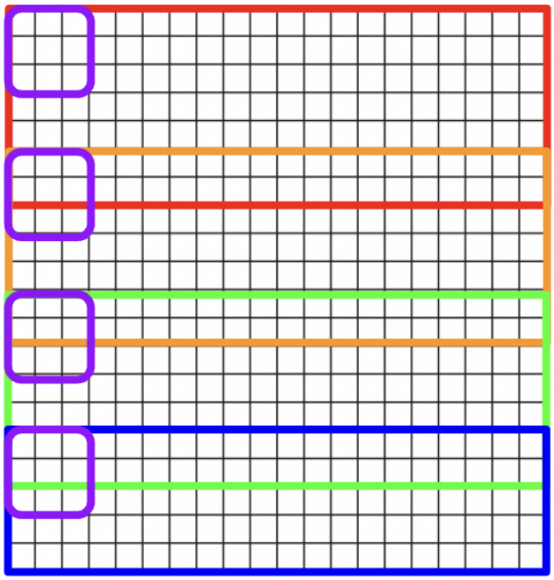

Figure 12. Overlap between parallelized M10K memories

Result

To evaluate the accuracy and efficiency of our hardware implementation, we selected an image of a wooden architecture in Germany, as depicted in Figure 13. This image features numerous distinct edges, making it an ideal candidate for testing edge detection algorithms, a common application of convolution calculations.

Figure 13. Image before processing. Source: Pixabay.

We applied three different filters to extract various features from the image. Firstly, a vertical edge detection filter was employed, characterized by a descent along the vertical axis as shown in the matrix form below. The resulting image, shown in Figure 14, predominantly highlights vertical edges while suppressing other features.

1	0	-1
2	0	-2
1	0	-1

Figure 14. Image after vertical edge detection.

Similarly, a horizontal edge detection filter was applied, featuring a descent along the horizontal axis. The matrix is shown below. Figure 15 illustrates the outcome, emphasizing horizontal edges across the image.

1	2	1
0	0	0
-1	-2	-1

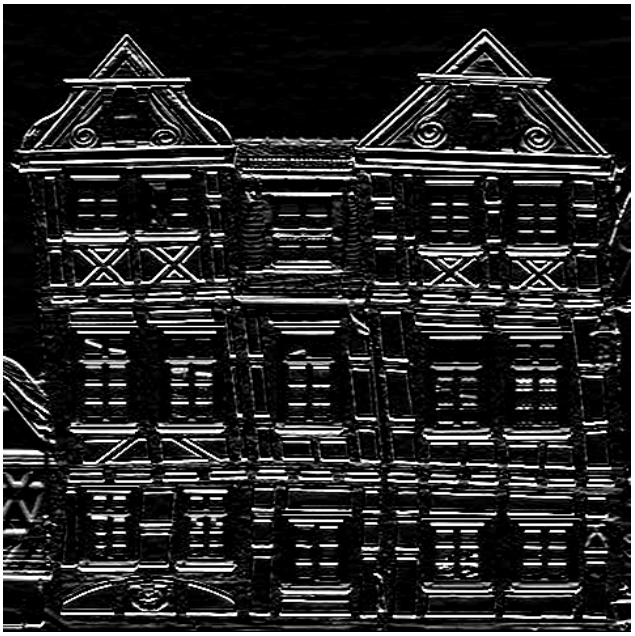

Figure 15. Image after horizontal edge detection.

Furthermore, an overall edge detection filter, combining both vertical and horizontal edge detection components, was utilized. The filter is shown below. Even the resulting image, depicted in Figure 16, is lighter than the previous two images, it exhibits a comprehensive representation of edges throughout the image.

0	-1	0
-1	4	-1
0	-1	0

The distinct outcomes obtained from applying different filters affirm the accuracy and effectiveness of the convolution layer calculations in our hardware implementation.

Figure 16. Image after edge detection.

We conducted a comprehensive analysis of the processing time for various design configurations, as illustrated in Figure 17. The "Python Calculation" represents the convolution calculations performed on the CPU, while "Baseline FPGA" denotes the non-parallelized implementation on the FPGA.

The number of iterations reflects the level of parallelization employed in each design. By applying four different filters to the same image and measuring their processing time, we observed that each filter exhibited varying processing times. However, consistently, the FPGA outperformed the CPU in terms of processing speed.

Moreover, we investigated the impact of increased filter iterations and parallelization on computation speed. As shown in Figure 18, the x-axis represents the number of iterations, while the y-axis indicates the speedup achieved by the parallelized FPGA implementation compared to the baseline.

While the ideal curve represents a straight line with a slope equal to the number of iterations, our results deviated slightly from this ideal curve. Nonetheless, they demonstrate significant speedup with increasing parallelization, indicating the effectiveness of our parallel computing approach.

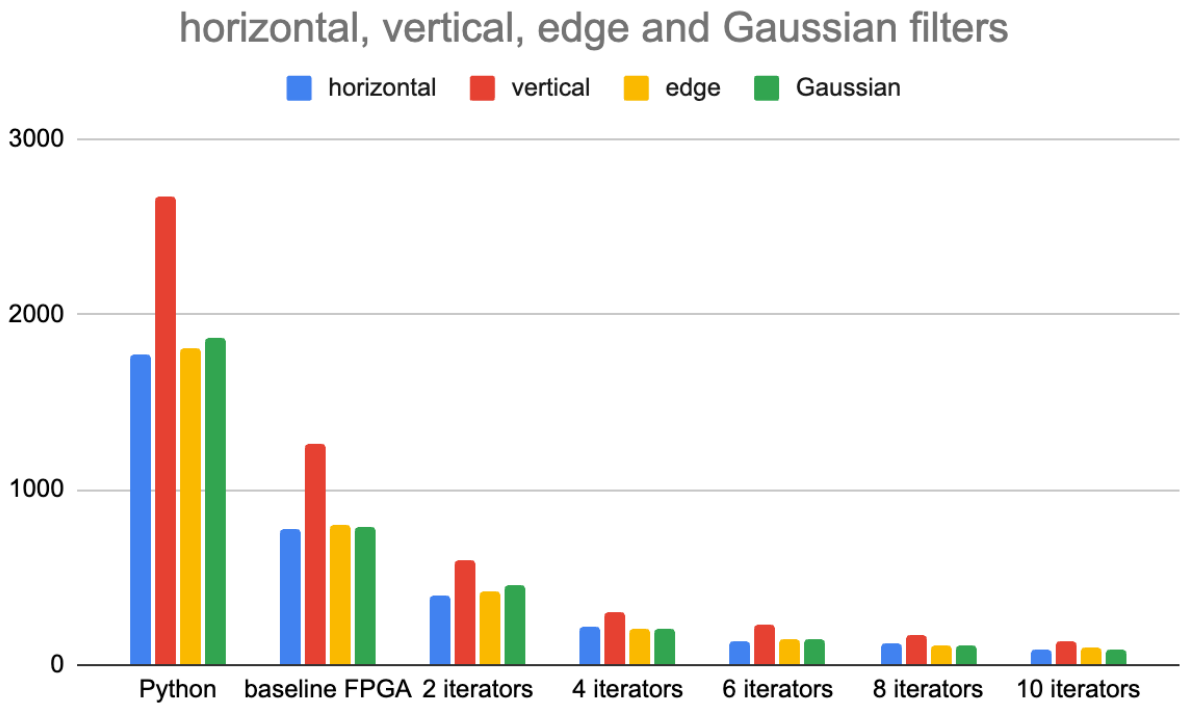

Figure 17. Processing time versus various design configurations

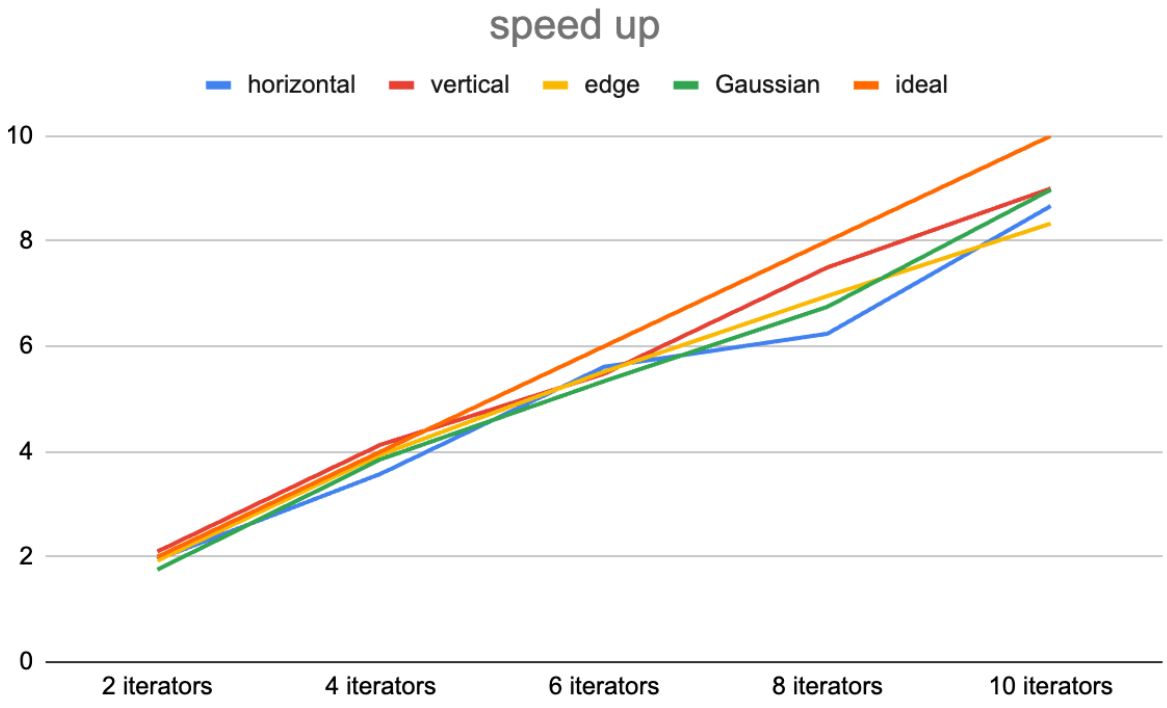

Figure 18. Speedup versus the number of iterators.

Future Work

If I have more time in the future, I want to expand this project on the following directions:

1. **Power Consumption Analysis:** While our current focus has been on performance improvements, future work could include a thorough analysis of the power consumption of the FPGA during hardware acceleration. Understanding the power requirements of FPGA-based implementations is crucial for assessing their overall efficiency and feasibility in real-world applications.

2. **Comparative Analysis:** Further comparisons between FPGA, CPU, and GPU implementations can provide valuable insights into the relative strengths and weaknesses of each platform. By systematically evaluating factors such as speed, power consumption, and resource utilization, we can better understand the optimal choice for specific applications and use cases.

3. **Energy Efficiency Assessment:** In addition to performance comparisons, it's important to assess whether FPGA offers superior energy efficiency compared to GPU implementations. By measuring energy consumption and performance metrics under various workloads, we can determine the energy efficiency trade-offs between different hardware acceleration solutions.

4. **Complete CNN Implementation:** While our current focus has been on accelerating convolution calculations, future work could involve implementing the complete Convolutional Neural Network (CNN) architecture, including additional layers such as the ReLU activation function, pooling layers, and fully connected layers. By extending our hardware acceleration approach to encompass the entire CNN pipeline, we can further enhance the efficiency and performance of deep learning algorithms for computer vision tasks.

5. **Optimization Techniques**: Exploring optimization techniques tailored specifically for FPGA implementations can lead to further improvements in performance, resource utilization, and energy efficiency. Techniques such as algorithmic optimizations, memory management strategies, and hardware design refinements can unlock the full potential of FPGA-based hardware acceleration for a wide range of applications.

6. **Real-time Applications:** Investigating the feasibility of real-time applications using FPGA-based hardware acceleration is another promising avenue for future research. By addressing latency constraints and optimizing throughput, FPGA implementations can enable high-speed, low-latency processing for time-critical tasks in fields such as autonomous driving, robotics, and medical imaging.

Conclusion

The results of our study demonstrate the superiority of FPGA-based hardware acceleration in convolution calculations, validating the potential of this approach for accelerating computer vision algorithms. By leveraging FPGA technology, we have achieved significant performance improvements compared to CPU-based calculations, highlighting the effectiveness of parallelization in accelerating convolution tasks.

As illustrated in Figure 17, increasing parallelism and utilizing parallelized filters lead to faster hardware acceleration. However, despite the benefits of parallelization, the overall speed is still constrained by the read speed of the M10K memory. Overcoming this limitation by enhancing memory access speeds or increasing data throughput per clock cycle could further optimize the efficiency of our FPGA implementation and achieve even greater speedups.

In conclusion, our hardware implementation effectively harnesses the power of parallelization to accelerate convolution calculations for image processing tasks. The diverse results obtained from applying different filters underscore the versatility and accuracy of our FPGA-based implementation. Furthermore, the observed speedup compared to CPU-based calculations reaffirms the superiority of FPGA-based acceleration for convolution tasks. Overall, our findings validate the suitability of convolutional neural networks for parallel computing and highlight the potential of FPGA technology in accelerating image processing algorithms, promising advancements that extend beyond the scope of this project and shape the future of deep learning and artificial intelligence.

Acknowledgment

I would like to thank my advisor Dr. Hunter Adams for his advice, encouragement, and continued support of this project.

Reference

- Jagreet Kaur (2023, June 15th), Convolutional Neural Network and its Latest Use Cases. https://www.xenonstack.com/blog/convolutional-neural-network
- Prashant Sharma (2023, August 22nd), Applications of Convolutional Neural Networks(CNN),https://www.analyticsvidhya.com/blog/2021/10/applications-of-convolutional-neural-networkscnn/
- Piotr Skalski (2019, April 12nd), Gentle Dive into Math Behind Convolutional Neural Networks.https://towardsdatascience.com/gentle-dive-into-math-behind-convolutional-neural-networks-79a07dd44cf9
- Augmented A.I. (2023, May 08th), The Best Object Detection Methods for 2023 | A Comprehensive Guide. https://www.augmentedstartups.com/blog/the-best-object-detection-methods-for-2023-a-comprehensive-guide
- Kh. Nafizul Haque (2023, April 03rd), What is Convolutional Neural Network — CNN (Deep Learning). https://www.linkedin.com/pulse/what-convolutional-neural-network-cnn-deep-learning-nafiz-shahriar/
- Alga Peng, Xiangyi Zhao, Sobel Edge Detection. https://people.ece.cornell.edu/land/courses/ece5760/FinalProjects/s2023/cp444_xz598/cp444_xz598/index.html
- Convolutional Neural Networks for Visual Recognition. https://cs231n.github.io/convolutional-networks/
- DE1-SoC User Manual. https://people.ece.cornell.edu/land/courses/ece5760/DE1_SOC/DE1-SoC_User_manualv.1.2.2_revE.pdf