Cornell University

Department of Electrical and Computer Engineering

ECE 6775

High-Level Digital Design Automation

Final Report:

Canny Edge Detection Project

Submitted by:

Junze Zhou(jz2275)

Yibin Xu(yx623)

Yiqi Sun (ys2257)

Zhongqi Tao (zt88)

Dec. 14th, 2023

Section 1: Introduction

Edge detection, a fundamental technique in the realm of computer vision and image processing, plays a critical role in extracting essential features from images. By precisely outlining the boundaries of interested objects or regions from their background, edge detection acts as a crucial cornerstone for higher-level image analysis, enabling tasks such as object recognition and scene understanding.

Its applications span across lots of domains, indicating its versatility and indispensability. In the realm of medical imaging, the precision of edge detection plays an important role in differentiating organs and tumors, aiding in the diagnosis of conditions and the planning of surgical interventions. Autonomous vehicles heavily rely on edge detection algorithms to interpret their surrounding environment, facilitating timely decisions for navigation and obstacle avoidance. Moreover, security and surveillance systems depend on edge detection to discern potential threats or unauthorized intrusions, hence enhancing the overall effectiveness of monitoring and response mechanisms. These various applications underscore the critical role of edge detection in shaping the baseline of modern technology and its profound impacts on multiple significant industries.

In this project, we delve into the optimization of the edge detection algorithm based on the commonly used Sobel operator, with a particular focus on adapting it for implementation on Field-Programmable Gate Arrays (FPGA). The primary goal is to enhance the efficiency of the algorithm by making them synthesizable on FPGA, thereby achieving a balance between minimizing latency and optimizing loop initiation intervals (II).

Basically, our tasks can be defined into two parts: firstly, to develop a Sobel-based edge detection project using C++ so that it can be further modified and optimized using HLS. Secondly, to modify the existing algorithm for seamless integration into FPGA architectures. By synthesizing the algorithms on FPGA, we aim to not only improve their performance but also ensure that the loops within the code operate with an initiation interval (II) of 1. This crucial modification is instrumental in achieving an effective balance between computational speed and FPGA resource utilization.

As we navigate through the details of our project, we will elucidate the underlying principles of edge detection, the challenges associated with FPGA synthesis, and the methodologies employed to realize a high-throughput, low-latency implementation. By the end of this report, we aim not only to explain the theories and functionalities of each step but also to demonstrate the tangible achievements in optimizing the edge detection algorithm for FPGA synthesis.

Section 2: Problem Description

In this section, we delve into a comprehensive description of the development process of the basic C++ edge detection project and the algorithm realized within our project, focusing on elucidating the operational framework without detailed information on the code.

Section 2.1: Pre-task Preparation

Our final design target is to have an optimized C++ edge detection project runnable on the Zybo Z7-20 FPGA board which has limited computational resources and is without image display capabilities. The better way to utilize the FPGA resources as an accelerator to speed up the computation process is to feed it with numerical data instead of the original image. Besides, most of the original images are colorful meaning that they have three channels: red, green and blue with each channel having numerous pixel data which can make our algorithm more intricate and resource-intensive.

Therefore, we decided to develop a simple C++ program using OpenCV to firstly convert the original colorful image into grayscale and then store each grayscale image pixel value as integers into a .dat file. Working with grayscale images simplifies the processing and reduces computational complexity. Meanwhile, edge detection primarily relies on variations in intensity or luminance, grayscale images represent intensity levels directly, providing a single

channel that simplifies the analysis of intensity changes across the image. **Figure 1** below demonstrates the original colorful image and its corresponding converted grayscale image.

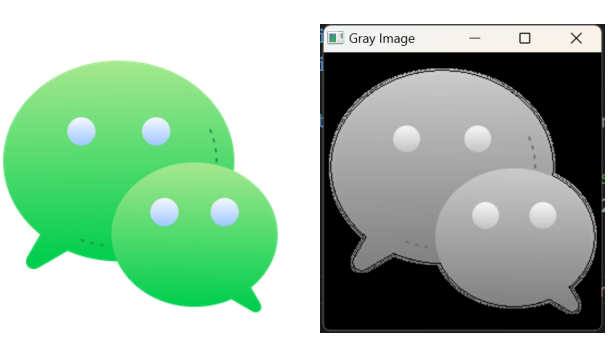

Figure 1. Example of RGB Image & Grayscale Image

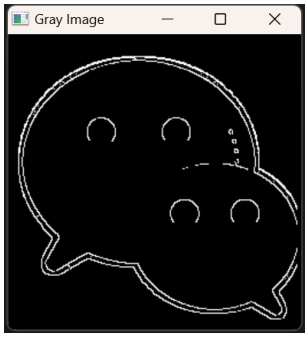

Figure 2. Example of Edge Detected Image

Now that we have obtained the grayscale image pixel values as numerical data stored in .dat file, the file can be uploaded to the FPGA as the input source to read data from. In this case, the edge detection algorithm can deal with the pixel values directly and hence output the processed pixel values indicating edges eventually. Since the FPGA has no display devices, the pixel values need to be converted back to an image to verify correctness of the edge detection algorithm. Thereby a simple C++ program using OpenCV is developed to restore the processed image by converting the pixel values back to an image. **Figure 2** demonstrates the image processed by the edge detection algorithm, where the edges of the picture are depicted precisely.

With the help of these helper programs to generate input files and display the edge detected image, the edge detection algorithm is simplified to fit the limited resources of the FPGA. The accuracy of the edge detection algorithm and the correctness of subsequently added HLS optimization commands can also be verified by restoring the processed image.

Section 2.2: Input and Output Specification

To simplify the process of dealing with input pixel value data files, although theoretically the edge detection algorithm can be used universally to process input data files of any grayscale image, in our project we decided to fix the original image size to 256×256 pixels.

For the edge detection project, initially, a function is defined to read the grayscale pixel values from .dat file generated before and store them in a 2D array whose first dimension indicates the index of an image while the second dimension is the number of pixel values, in our case is $256 \times 256 = 65536$. It needs to be noticed that the data type of the array is unsigned int8_t (8-bit integer) since the pixel values of a grayscale image only range from 0 to 255. Correspondingly, the processed pixel values are stored in a 2D array with the same size and same data type.

To transmit the pixel stored in the array to the FPGA, the HLS stream with the data type of bit32_t is created and hence the data in the array are further split into four values per group since each data only takes 8 bits while we have a 32 bits stream.

So, in summary, there is a loop iterating over segments of 32 bits (4-pixel values per time) from the array, packing these bits into a temporary bit32_t type variable, and then writing the resulting bit32_t variable to the HLS stream. This process allows the code to send the image data in chunks to the hardware block through the stream.

Similarly, to receive the output pixel values processed by the algorithm, which is accelerated by the FPGA, the corresponding HLS output stream with the data type of bit32_t and a temporary bit32_t type variable collecting groups of outputs are defined. The data are read from the output stream as 4 bytes per group (4-pixel values per time) and each individual byte (individual pixel) is extracted and written into a .dat output file, so that we can use

the “restore” helper function mentioned in the last section to analyze the correctness of our algorithm implemented on the FPGA by restoring the edge detected image from the output pixels.

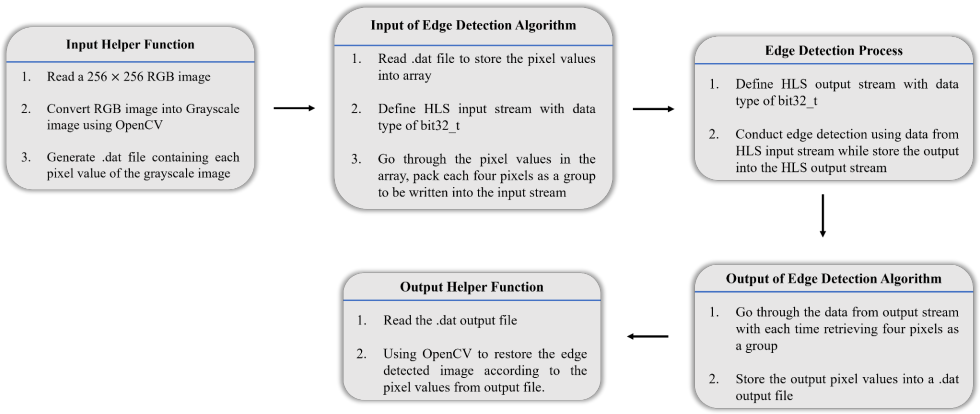

Figure 3. Application Flow Diagram

To summarize, the block diagram in **Figure 3** shows the process of generating the input file, transmitting data from the input file to the FPGA blocks, conducting edge detection using FPGA, receiving processed pixel values from FPGA to form an output file, and eventually restoring the output image based on the output file to check the edge detected result.

Section 2.3: Edge detection

Edge detection is a popular image processing technique used to identify edges in an image. It was developed by John F. Canny in 1986 and is known for its robustness and accuracy. The Canny edge detection algorithm aims to identify the boundaries of objects within an image by detecting sudden changes in intensity or color. Generally, the edge detection algorithm consists of several steps as follows:

1. Gaussian Smoothing: The first step is to apply a Gaussian blur to the input image hence minimizing the noise in the input image because noise can lead to false detection of edges.
2. Gradient Calculation(Sobel): For calculating the gradient amplitude, the traditional Canny edge detection algorithm employs the central pixel in a small 2×2 neighborhood window to determine the finite difference mean value, representing the gradient amplitude. This approach, while straightforward, is notably susceptible to noise, leading to the potential detection of false edges and the omission of genuine edges. In contrast, it has become increasingly prevalent to utilize a 3×3 Sobel filter as a more effective alternative. The Sobel filter, owing to its larger kernel size, enhances the accuracy of both gradient magnitude and direction determination. The choice of edge detection operator, such as the Sobel filter, significantly impacts the quality of the edge detection results, striking a better balance between sensitivity to real edges and resistance to noise.
3. Non-Maximum Suppression: In this step, the algorithm identifies the local maxima in the gradient magnitude. This step examines each pixel and its neighbors in the direction perpendicular to the edge direction. If the pixel's intensity is not greater than its neighbors in this direction, it is suppressed (set to zero). This ensures that only the strongest edges (local maxima) are retained, and the edges are as thin as possible.
4. ZeroPadding: Zero padding is typically used in image processing to pad the border of the image with zeros. This is often a preparatory step for other operations like convolution, which requires a certain context for each pixel. In the case of edge detection, zero padding is used to maintain the original image size after applying filters like the Sobel operator.
5. Edge Tracing by Connectivity Analysis: The final step involves tracing the edges by connecting weak edges to strong edges. The weak edge pixels are analyzed to be connected to nearby strong edge pixels so that to form the connected edges.

Section 3: Implementation

Section 3.1: Baseline Design

Figure 4 illustrates the high-level diagram of the baseline design. In detail, we leveraged the two FIFOs with a width of 32 bits, and all pixels of one image will be organized into an array before transmitting to the processing modules. That is to say, a whole array where 256*256 pixels with the representation of an 8-bit unsigned integer will be communicated between submodules. Similarly, the array will be split into 32-bit data streams, that is, 4 pixels at a time and sent back to the processor.

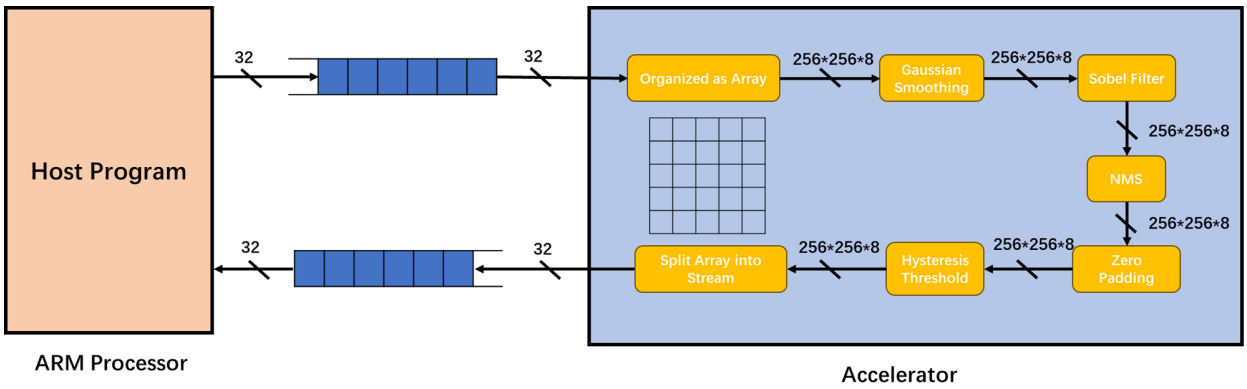

Figure 4. Baseline Design in High-Level

In our design, we employed three categories of optimization methods: leveraging abundant parallelism, distributed memory accesses, and data streams with custom numeric types, which are described below.

Section 3.2: Parallelism Exploitation

As we've done in the previous lab, the first optimization strategy is to exploit the parallelism of the implementation of our algorithm. Specifically, we mainly used array partitioning and array reshaping to reorganize the structure of some of the variables that are involved in the loops. In detail, we've tried to partition and reshape the pixel arrays, convolution kernels, line buffer, window buffer, etc.

The reorganization of these arrays allows us to exploit the parallelism by employing loop pipelining, loop unrolling, and function pipelining optimization strategies to improve the performance of our design primarily. By using pipelines at every stage of design（including Gaussian Smoothing, Gradient Calculation(Sobel), Non-Maximum Suppression, ZeroPadding, and Edge Tracing), we successfully reduced the II of each function as much as possible. Pipelining can be applied to each step to improve throughput and reduce latency. The key is to manage data dependencies and ensure a continuous flow of data through the system, maximizing the utilization of available resources.

However, exploitation of parallelism will inevitably increase resource usage, indicating more hardware components like LUTs and BRAMs are consumed. For baseline design, we transmitted the data in the form of the whole array between each module, so many BRAMs will be consumed. The consumption is more evident when we increase the size of the input image, so we considered using distributed memory accesses, that is, line buffer and window buffer to further optimize our design.

Section 3.3: Line Buffer and Window Buffer

For the line and window buffer, instead of reading the entire array over and over again, we only read one data at a time, which helps to reduce the read penalty. As shown in **Figure 5,** the first step is to move the line buffer up by

one row; the second is to write the new data in the button row of the buffer; the third is to shift the window buffer left by one column; finally, replace the most right column of the window buffer with the newly created line buffer.

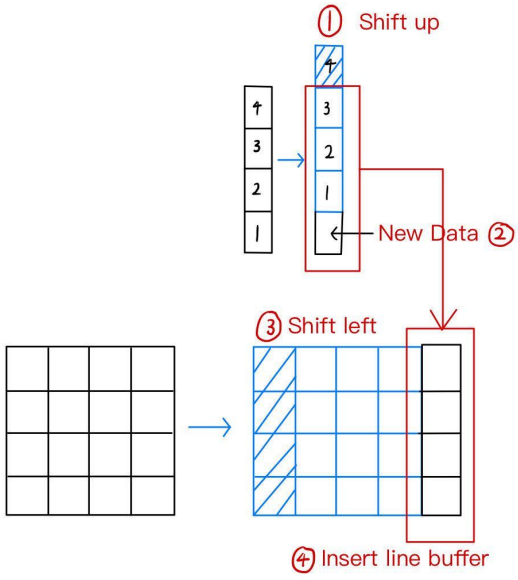

Figure 5. Demo of the Line Buffer & Window Buffer

Section 3.4: Data Stream

Apart from introducing line buffers and window buffers, we also implemented a data stream to further reduce resource usage and leverage pipelining to the maximum extent. As shown in **Figure 6**, instead of transmitting the whole array between submodules, we only transmit one pixel with its relevant information.

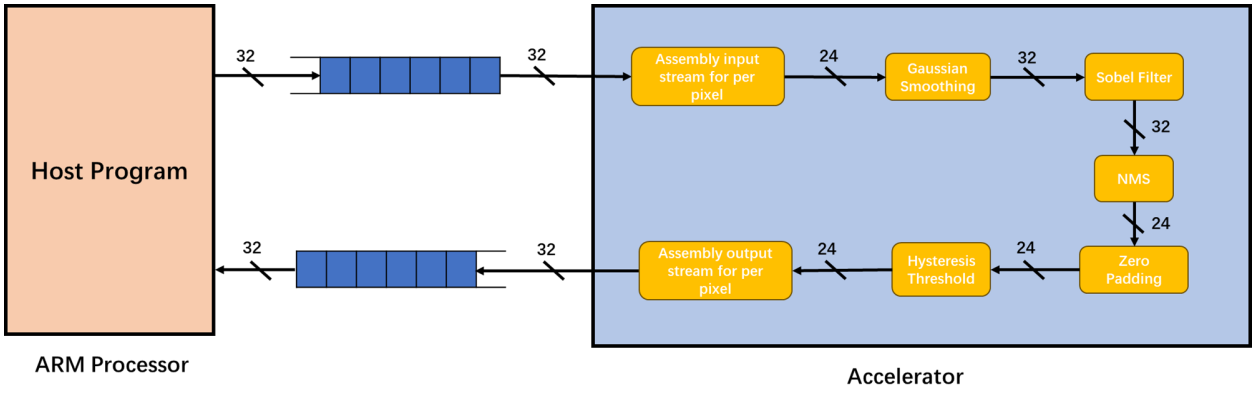

Figure 6. Data Stream Schematic Diagram

As shown in **Figure 7**, we assembled the data stream into a specific format with three to four fields before sending it to the next module. Specifically, the first field is the pixel value with a size of 8 bits; the second and third fields are the corresponding column and row, respectively, which are used to index the line buffer and window buffer; and the fourth field contains the gradient information of this pixel with a size of 8 bits, which performs as one of the output results of Sobel filter and the input data of NMS submodule.

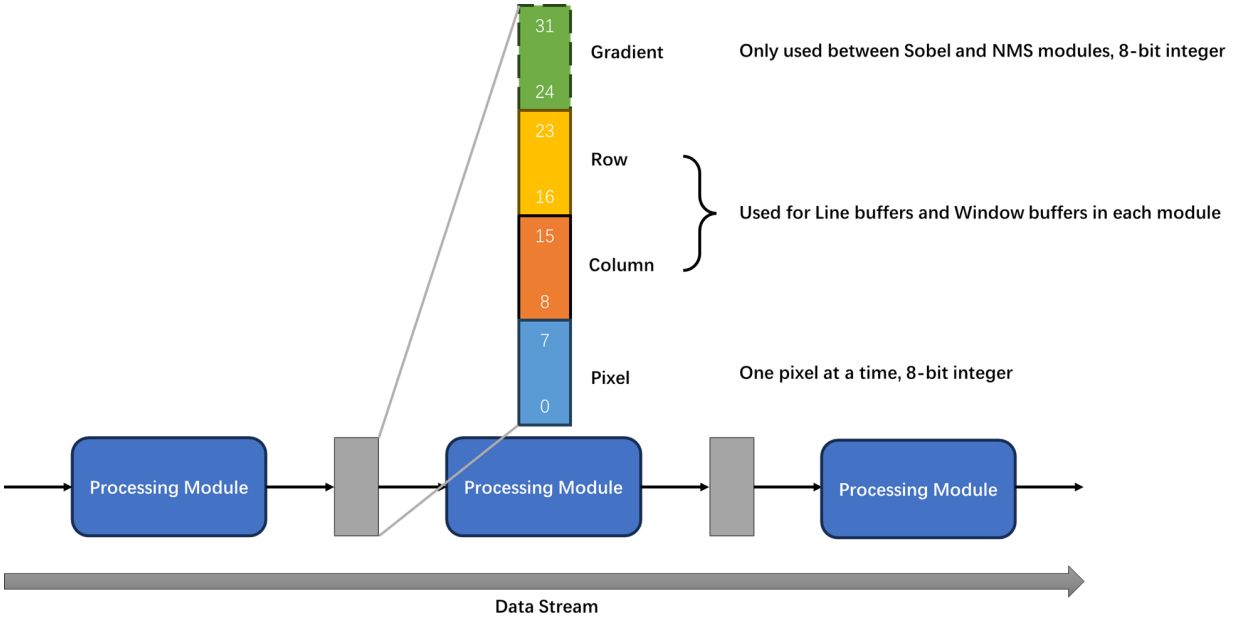

Figure 7. Data Stream Format

In each module where adjacent pixels are needed to conduct processing, we first updated the line buffer and window buffer of each module based on the new pixel as described in **Section 3.2**. Then, based on the updated window buffer and the new pixel and its corresponding row, column, and gradient, different processing algorithms will be executed.

Section 4: Evaluation

In this final lab, we have optimized three versions of implementations. The first version is directly based on the open-source code. (Baseline Design). In the second version of the code, we used the HLS tool to add window buffer and line buffer to the design to further optimize the performance of our design. In the third version implementation, we implemented data flow based on the second implementation and used the HLS tool to accelerate the design as well.

After optimizing the above three codes, we used the TCL file to generate the Verilog files of these three implementations and evaluated the three implementations separately through the information in the report file as shown in **Table 1**. The indicators evaluated mainly include latency and resource usage. The detailed data we collected are shown as follows. In order to make the comparison results more obvious and facilitate us seeing the advantages and disadvantages of the three designs, we classified the relevant data and integrated them into charts to display below, and analyzed and interpreted the charts.

Table 1. Resource Usage and Latency of Different Implementations

	Latency(cycles)	FF	LUT	DSP	BRAM
Baseline	14599614	4	16	4	82
HLS Buffer	17370126	4	15	4	87
HLS Buffer(Data Flow)	15171585	2	11	5	4
Baseline optimized	2210652	5	20	0	72
HLS Buffer optimized	1391886	33	73	0	72
HLS Buffer(Data Flow) optimized	2146305	9	51	0	5

Figure 8 and **Figure 9** below intuitively demonstrate the latency and resource usage versus different implementations of optimization methods.

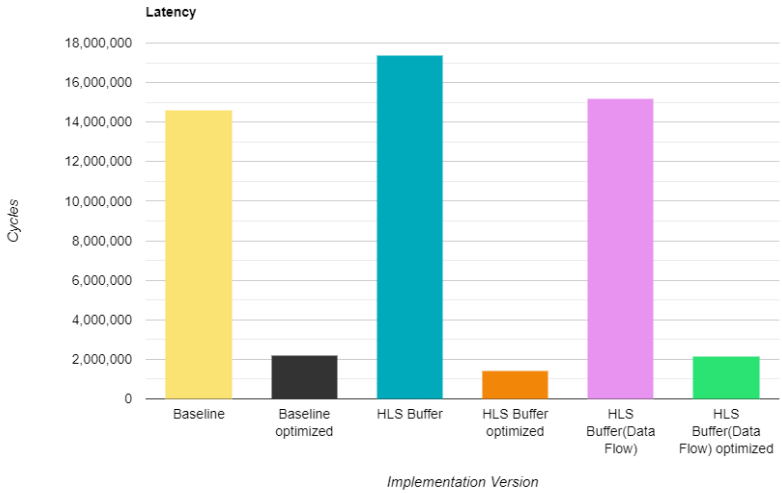

Figure 8. Latency versus Implementation of Optimization Methods

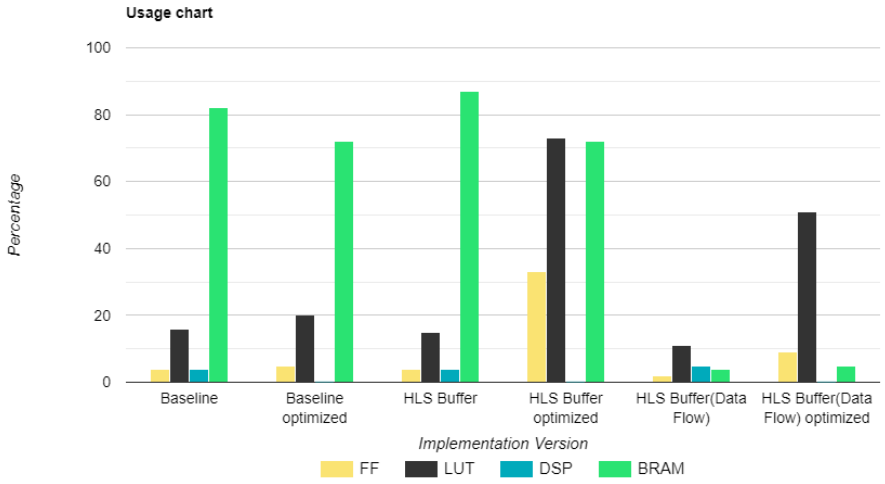

Figure 9. Resource Usage versus Implementation of Optimization Methods

In terms of resource utilization, this baseline design uses a lot of BRAM and LUT, and almost no DSP. The resource usage after optimization did not increase significantly for the baseline design. However, in the last two designs, after using HLS to optimize the design, the number of FF and LUT used increased significantly. This is mainly because we implement pipelining in our design, which can increase the usage of the area. The unoptimized third implementation has the least resource usage due to the data flow.

In terms of execution time (latency), after the code of these three versions is optimized using HLS, the processing time is greatly reduced. Among them, the second implementation (HLS buffers) with the most obvious effect is the design using the HLS line buffer. It has the shortest processing time (only 1391886 cycles) and the most obvious acceleration effect (about 12.5X) among the three versions of the code. The reason why the third design doesn't get the shortest execution time and the greatest acceleration effect is that in that design we get rid of a bunch of for loops

using data flow. So there are just a few for loops we can pipeline. However, the advantage of it is that this implementation can use far fewer resources than others.

The line buffer is a specific type of buffer that stores only a few rows of the image (usually the ones currently being processed). This approach is more memory-efficient compared to storing the entire image, especially important in FPGA designs where memory resources are limited. This approach can in this way reduce the usage of BRAM. With a line buffer, an FPGA can process data in a streaming fashion. It reads, processes, and writes data continuously without the need for the entire image to be stored in memory first. This increases the throughput of the system. The use of a line buffer reduces the latency between reading an image row and processing it. The line buffers and window buffers are suited for the data flow. This is because the data is readily available in the buffer as soon as it's needed, rather than requiring a read from a larger memory structure. Implementing a line buffer might require more FPGA resources (like LUTs, and FFs), but it significantly improves the performance in terms of processing speed and latency.

Section 5: Project Management

Table 2 below illustrates the suggested timeline versus the actual timeline of our project, including some great progress and each group member's responsibilities regarding different tasks.

Table 2. Project Timeline

Date	Suggested Tasks	Actual Progress	Group Member
11/07 - 11/13	- Complete baseline design of edge detection algorithm with Sobel operator. - Perform comprehensive software simulation to evaluate the algorithm results.	- Finished development of OpenCV based helper programs to generate inputs for project and retore outputs from project to image.	Yiqi Sun Zhongqi Tao
11/14 - 11/20	- Perform design optimization using HLS tools. - Simulate the baseline design on the Zybo Z7-20 hardware.	- Debugged the errors encountered for baseline development.	Junze Zhou Yibin Xu
11/21 - 11/27	- Implement the optimization algorithm of baseline design on board. - Integrate the camera module into the system.	- Finished baseline design. - Instead of using camera captured images, we used 256 x 256 images.	Yiqi Sun Zhongqi Tao
11/28 - 12/04	- Further optimize the algorithm for processing multiple images.	- Added line buffer. - Achieved II = 1 for most of loops.	Junze Zhou Yibin Xu
12/05 - 12/14	- Prepare for final presentation. - Write final project report.	- Finished final presentation and final project report.	Junze Zhou Yibin Xu Yiqi Sun Zhongqi Tao

We successfully completed the project within the expected timeframe, managing to deliver both the presentation and the report on schedule. Despite encountering disparities and delays compared to our original plan, we navigated through challenges and adjusted our approach during the project's execution due to constraints related to hardware and time.

One notable obstacle was the limited memory capacity of the ARM on the FPGA, particularly concerning the resource-intensive nature of the edge detection algorithm we were implementing. The algorithm's high memory consumption became a significant concern, leading to the exhaustion of the ARM memory on the FPGA. However, we persisted, successfully completing the baseline design and subsequently optimizing it using High-Level Synthesis (HLS). This optimization resulted in a satisfactory outcome, indicating our ability to overcome obstacles and refine our approach for optimal project execution.

Section 6: Conclusion and Acknowledgements

In conclusion, our exploration into edge detection algorithms and FPGA synthesis has illuminated the fundamental role of edge detection in computer vision and image processing. This foundational technique serves as a bedrock for extracting crucial features from images, with applications spanning diverse domains such as medical imaging, autonomous vehicles, and security systems. The adaptability and indispensability of edge detection underscore its transformative impact on modern technology across various industries.

Our project specifically delved into the optimization of a Sobel-based edge detection algorithm, emphasizing its adaptation for FPGA implementation. The primary objective was to enhance the algorithm's efficiency by making it synthesizable on FPGA, achieving a delicate balance between minimizing latency and optimizing loop initiation intervals (II). Through multiple tasks, we successfully developed a C++ edge detection project, tailored it for FPGA synthesis, and achieved a seamless integration with FPGA architectures.

In acknowledgment, we express our gratitude to the course staff particularly Prof. Zhiru Zhang who provided invaluable assistance and support during the project. Their insights, guidance, and collaborative efforts significantly contributed to the successful execution of our edge detection optimization project. This journey has not only enhanced our technical skills but has also instilled a deeper understanding of algorithm optimization and FPGA synthesis.

Section 7: Reference

Medalotte. (n.d.). *Medalotte/HLS-Canny-edge-detection: FPGA implementation of canny edge detection by using Vivado HLS*. GitHub. https://github.com/medalotte/HLS-canny-edge-detection