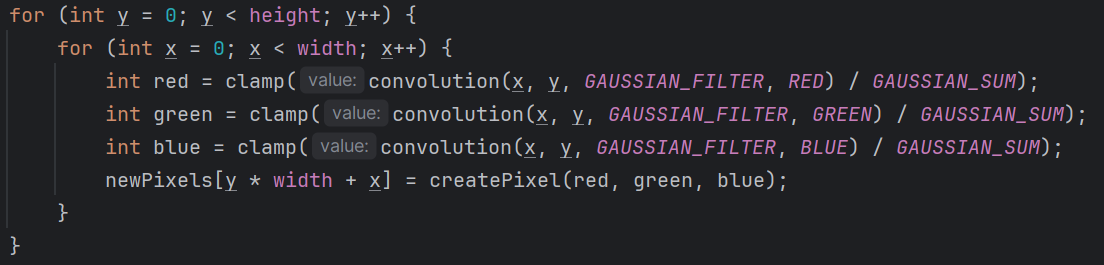
# Assignment 3 Report

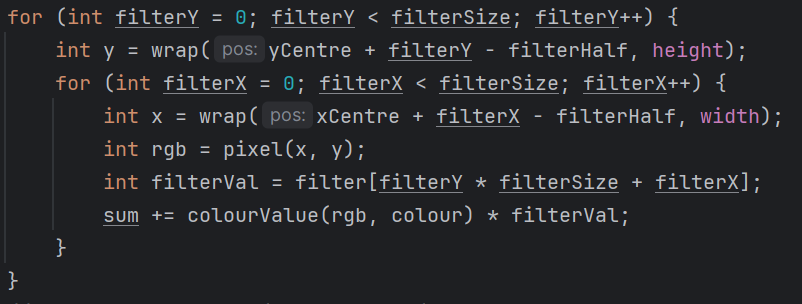
## Executive Summary

The graph displays the operation times in seconds for various image processing tasks on the GPU and the CPU both in their original and modified states. On average, the GPU code achieves a speedup of approximately 10.32x compared to the original CPU code. Likewise, the CPU also achieves a speedup a approximately 2.69x compared to the original CPU code. The average operation time for the GPU-modified code is 0.452 seconds, significantly faster than the average time of 1.74 seconds for the CPU-modified code and the average time of 4.96 seconds for the CPU-original code. In summary, these results demonstrate a substantial improvement in performance by optimizing the code for GPU dan CPU execution, indicating the effectiveness of leveraging GPU capabilities for image processing tasks.

## Code

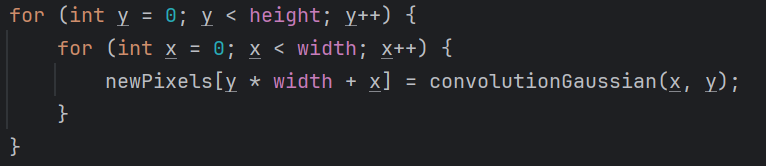
The original code takes approximately 5 seconds to process each image. The two most time-consuming components are the Gaussian filter, which takes about 2 seconds, and the Sobel filter, which takes about 1.5 seconds to complete. The codes calculates the colour one by one and combined it later after all colour calculated can be seen in Code 1 and Code 2.

Code 1 Originial Gaussian Filter

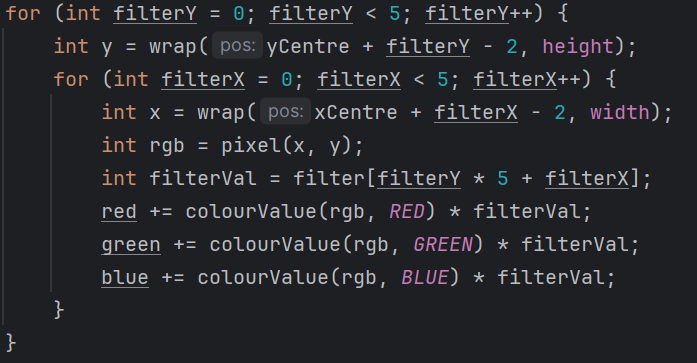


Code 2 Original Convolution

In the updated code, significant optimizations have been made. The Gaussian filter now runs in 1.4 seconds, and the Sobel filter in 0.5 seconds. These improvements were achieved by breaking down the convolution function into one part, allowing for more efficient processing as shown in the Code 3.



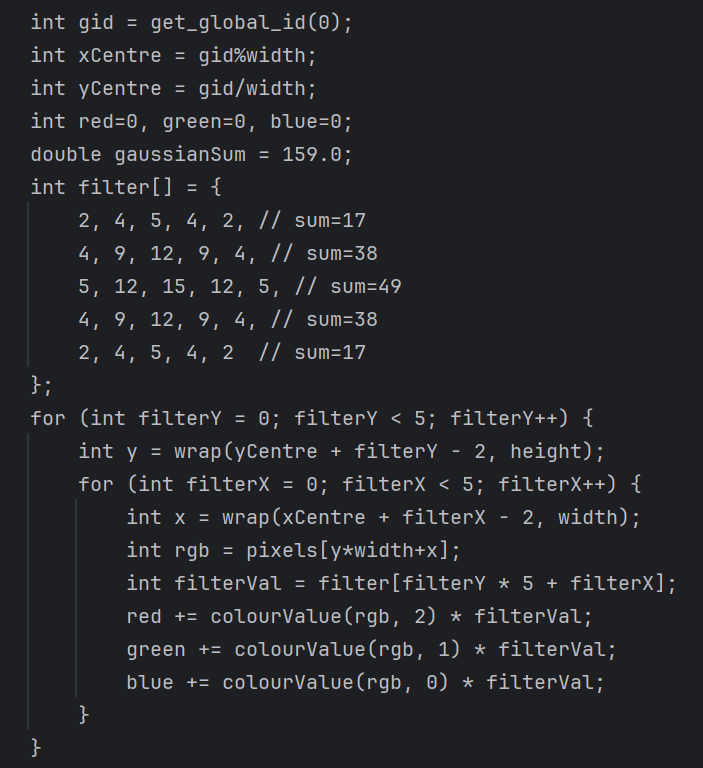
Code 3 Modified Gaussian Filter



Code 4 Modified Convolution Gaussian

One key optimization in the updated code is the reduction in memory access. Previously, the code accessed the required pixel’s memory for each color channel separately. Now, the code accesses the pixel's memory only once for each pixel, significantly reducing the overall memory access overhead and improving the processing speed.

The GPU code efficiently allocates each pixel to its processing unit as shown in the Code 5, a strategy that enhances performance significantly. By dedicating one processing unit per pixel, the GPU can exploit parallel processing capabilities. This approach maximizes the GPU's efficiency, outperforming traditional CPU processing by a wide margin.



Code 5 GPU Gaussian Code

Optimizing the Sobel convolution in the same manner as the Gaussian function sounds like a promising approach to further improve the efficiency of your program. Reducing the processing time from 1.5 seconds to 0.5 seconds can significantly enhance the overall performance, addressing another bottleneck in the program. Meanwhile, the color reduction and masking edge functions didn't show much improvement despite removing one loop from each function.

In GPU programming, there are several steps involved in setting up the environment, such as obtaining the platform and device information, creating the context, queue, and pointers for data transfer. The main idea behind GPU programming is to parallelize the work of each function, but some functions may still have dependencies on each other.

After examining the code, it can be concluded that the Sobel function requires the output from the Gaussian function. However, the Sobel function and the color reduction function can run independently. The merge mask function, on the other hand, requires the output from both the Sobel function and the color reduction function. Therefore, it is important to ensure that these dependencies are properly managed to achieve efficient parallelization. As the results, the code can be seen in Code 6

Code 6 GPU Enqueue Code

## Summary

The original code achieved a significant speedup of 10 times by utilizing GPU code, leveraging the parallel processing power of the graphics card to accelerate computation. Additionally, a 2.5 times speedup was achieved using a modified version of the original code. The primary bottleneck in the original implementation was identified as memory access, which was addressed by optimizing the code to minimize redundant memory access. The key challenge faced during the optimization process was the concept of reducing memory access by combining all color operations into a single function.

For future projects, it is recommended to explore alternative approaches to parallelization. Rather than having each processing unit calculate each pixel individually, a more efficient strategy could involve assigning several processing units to calculate a single pixel. This approach can help to further optimize memory access and improve overall performance.