COMPX553  
GPGPU Programming Cartoonify Photos

By Hannah Carino  
1575791  
University of Waikato

# Executive Summary

This report outlines the development process involved in enhancing GelAnim's photo processing application, Cartoonify, by implementing GPU acceleration in Java. The objective was to evaluate the performance improvement and enhance the overall efficiency of the application. Figure 1 illustrates the processing time of the image "img\_bucket.jpg" across 10 iterations, chosen to ensure reliable average observations and accommodate any anomalies. Meanwhile, Figure 2 demonstrates the processing time using the GPU implementation over the same 10 iterations. Throughout each iteration, debugging was enabled, with minimal observed impact based on experimentation. The findings reveal that the GPU implementation achieves approximately 6.1 times faster processing speeds compared to its original CPU counterpart. Figure 3 illustrates this drastic performance hike.

Figure 1 GPU Implementation Performance

Figure 2 Original Program Performance

Figure 3 Comparison of Original and GPU Performance

# Introduction

During the course of this assignment, a divide and conquer approach was adopted to effectively tackle the project at hand. By dissecting the large overarching task into smaller, more manageable segments, we successfully developed a GPU-accelerated version of Cartoonify. This process involved three primary steps:

1. Task Understanding: A thorough comprehension of the assignment's requirements and objectives.
2. Java Code Optimization: Fine-tuning the Java codebase for improved performance and efficiency.
3. GPU Implementation with Parallelization: Integrating GPU acceleration and parallel processing techniques into the application.

Each of these steps will be elaborated upon in the subsequent sections of this report.

# Process and Discussion

## Step One: Understanding the Task

Understanding the task involved reading the requirements in depth, writing down key steps of each process, debugging, and measuring performance of the original application as a baseline. These findings were recorded, producing Figure 1.

### Exploration

Initial output of debugging produced the information in Figure 4.

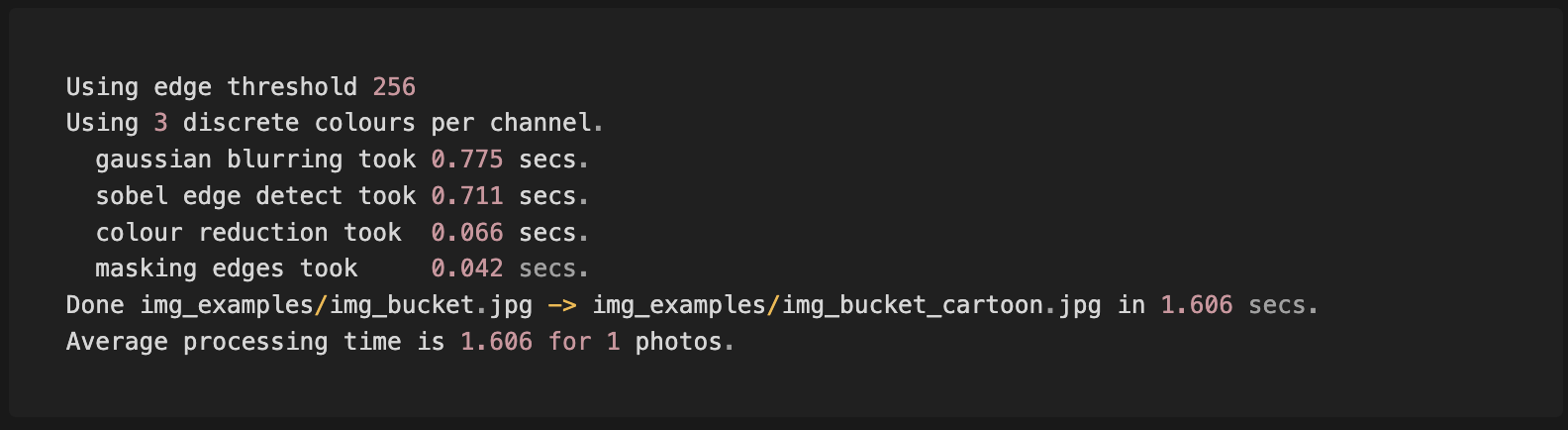


Figure 4 Initial debugging output of program

The output displays the bottleneck functions of this application are gaussian blurring and sobel edge detection. Further investigation revealed this is due to the multiple calls to the `convolution` function, the real culprit of the slow process.

### Conceptual Understanding

Leveraging MermaidMD graphs, I created the directed acyclic graph to conceptualise what functions can and cannot be parallelised. Figure 5 illustrates the process from the original image, the two parallel branches of execution, and how the output of each branch is merged in the final state of processing.

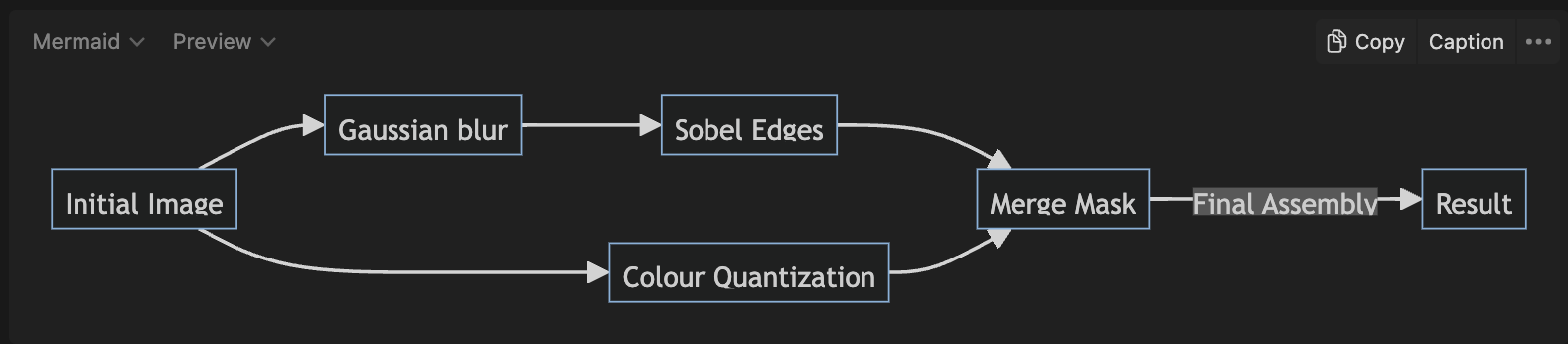


Figure 5 Directed Acyclic Graph for Parallel and Sequential Components

I kept note of a key hint in this state, where `cl\_events` can be used for synchronisation – important for later stages.

## Step Two: Java Optimisation

Upon analysis conducted in the discovery phase, it was determined that the `convolution` method posed the highest computational overhead. The issue stemmed from the while loop responsible for computing the size of the filterArray parameter passed to the `convolution` method. Subsequent modifications yielded an average time reduction of approximately 0.2 seconds, as depicted in Figure 6. While this optimization showcased discernible improvements in processing time for individual images, its impact is expected to be more pronounced in scenarios involving heavier computational loads, such as processing multiple images concurrently.

Figure 6 Performance of Original Program vs Java Optimised Version

Modifications for this section included:

* Converting the while loop calculation of filterSize to utilise Math.sqrt() – square rooting the original filter size.
* Reducing arithmetic operations in the wrap() method used widelt throughout the application.

## Step Three: GPU Implementation

This phase was further subdivided into several segments. Initially, each procedure depicted in Figure 5 was re-implemented within the kernel.cl file, including helper functions such as `convolution`. Next, validation was conducted to confirm that the outputs corresponded to the anticipated results utilizing the `md5sum` function within the Linux terminal. Subsequently, parallelization techniques were implemented with `cl\_events` to ensure synchronised execution for sequential events.

Four core kernel functions were created; corresponding to the functions defined in the Java implementation, and illustrated in Figure 5.

Figure 7 Core Kernel Functions

The helper functions implemented were:

Figure 8 Helper Kernel Functions

Additionally, constants were created in global scope of the kernel.cl file. Originally, I opted to use constant memory for this implementation with the `\_\_constant` modifier, which resulted in unexpected behaviour in the Linux Labs. Namely, I was unable to pass GAUSSIAN\_FILTER as an argument to the convolution function due to it being labelled a `\_\_constant`. The constants declared include the following:

Figure 9 Helper Kernel Constants

The constants in Figure 9 mimic their exact Java implementation counterparts.

The implementation of parallelization leads to the most substantial optimization gains. In this context, two methods were evaluated: employing a single command queue and utilizing multiple command queues. Initially, I intended to utilize three command queues, as outlined in Figure 10’s pseudocode.

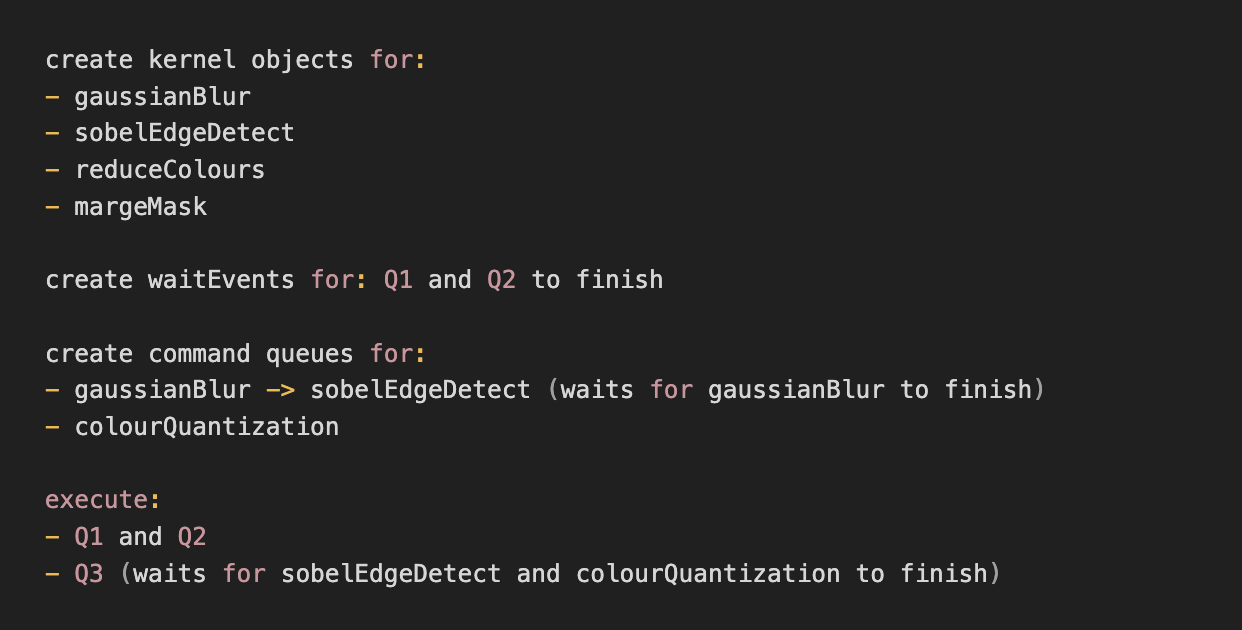


Figure 10 Pseudocode for multi-queue implementation

However, upon further consideration, I concluded that I could enhance the implementation by maximising the utilisation of a single command queue. Using a single command queue for a program that is not highly intensive offers several advantages over employing multiple queues. Primarily, it requires less overhead in terms of both memory and computational resources. By consolidating commands within a single queue, unnecessary duplication of resources is avoided, leading to more efficient execution and resource management. This streamlined approach can result in improved performance and simplified maintenance compared to the use of multiple queues.

The single queue approach was documented in pseudocode prior to implementation (Figure 11).

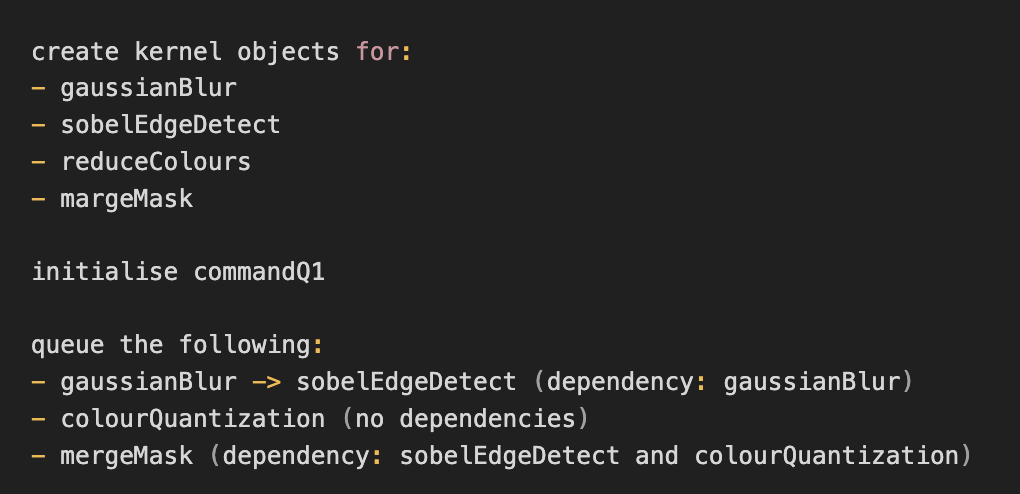


Figure 11 Pseudocode for single queue method

Through debugging, it was observed that the resultant images of each stage in the directed acyclic graph, as delineated in Figure 11, exhibit identical outcomes for both CPU (Original and Java Optimised) and GPU implementations. The corresponding md5sums are presented in Figure 12.

|  |  |
| --- | --- |
| Image Version | `md5Sum` Output |
| Gaussian Blur | f409aca991ec772a735aefd65f37e1a5 |
| Sobel Edge Detect | adf55df7839c4fccc30090d6278da1c7 |
| Colour Quantisation | 547db739f6b3edeac763b5f545f6eef1 |
| Cartoon | cf2be2945692744b323b7b47724b43aa |

Figure 12 Table displaying md5sum outputs

# Results

Experimentation and implementation underscore the benefits of GPU acceleration. The integration of GPU acceleration facilitated a significant enhancement in the speed of the Cartoonify program, achieving approximately a 6.1x acceleration compared to the original sequential Java application. This acceleration can be attributed to the concurrent execution of smaller tasks within the kernels, leveraging a divide-and-conquer approach to tackle computationally intensive processes such as image processing. While not all programs are amenable to parallelisation, this assignment has highlighted the pivotal role of GPU acceleration across various industries.

To further refine this assignment, enhancements could be made to handle more than one pixel per kernel. Currently, there may be underutilisation of each kernel's potential, leading to potential wasted resources. Additional experimentation is necessary to determine the optimal work item size, thereby establishing the ideal number of pixels to process within a work item. Figure 13 presents a visual comparison of the three stages the program underwent, transitioning from the original version to the Java-optimized version, and finally, to the GPU implementation.

Figure 13 Performance Comparison of Three Stages