Image Processing With CUDA

Final report

Submitted for the MEng in

Computer Science for Games Programming

April 2024

by

Dawid Kisielewski

Word count: XXXXX

Abstract

Image processing is a field with a wide use of GPU acceleration. GPU’s parallel nature makes them perfect for doing computation on images as each thread can represent one pixel of the image and most algorithms don’t need to manipulate shared data. Exploring the differences between image processing algorithms running sequentially on the CPU and parallel on the GPU allows us to develop an understanding of the characteristics of both GPU programming as well as GPU architecture.

Acknowledgements

Contents

[Abstract i](#_Toc165308837)

[Acknowledgements ii](#_Toc165308838)

[1 Introduction 3](#_Toc165308839)

[1.1 Background to the project 3](#_Toc165308840)

[1.2 Aims and objectives 3](#_Toc165308841)

[1.2.1 Objective 1 – Research Image Processing Algorithms 3](#_Toc165308842)

[1.2.2 Objective 2 – Creating CPU implementations 3](#_Toc165308843)

[1.2.3 Objective 3 – Research GPU Architecture 3](#_Toc165308844)

[1.2.4 Objective 4 – Creating GPU Implementation 3](#_Toc165308845)

[1.2.5 Objective 5 – Compare CPU and GPU Performance 3](#_Toc165308846)

[1.3 Research question 4](#_Toc165308847)

[2 Literature review 5](#_Toc165308848)

[2.1 Image Processing 5](#_Toc165308849)

[2.1.1 Edge Detection 6](#_Toc165308850)

[2.1.2 Image Segmentation 8](#_Toc165308851)

[2.1.3 Noise Reduction 10](#_Toc165308852)

[2.2 GPU 10](#_Toc165308853)

[2.2.1 GPU Architecture 11](#_Toc165308854)

[2.2.2 GPU Compute 11](#_Toc165308855)

[2.3 Language Options and Libraries 11](#_Toc165308856)

[2.3.1 Language 11](#_Toc165308857)

[2.3.2 Lrbraries 11](#_Toc165308858)

[2.4 Similar Studies 11](#_Toc165308859)

[3 Requirements 12](#_Toc165308860)

[3.1 Product requirements 12](#_Toc165308861)

[3.2 Functional requirements 12](#_Toc165308862)

[3.2.1 Interface 12](#_Toc165308863)

[3.2.2 Algorithms 12](#_Toc165308864)

[3.2.3 Benchmark 12](#_Toc165308865)

[3.3 Design constraints 12](#_Toc165308866)

[4 Project Management 13](#_Toc165308867)

[5 Design 14](#_Toc165308868)

[5.1 Software design 14](#_Toc165308869)

[5.2 Experimental design 14](#_Toc165308870)

[5.2.1 Repeatability 14](#_Toc165308871)

[5.2.2 Representability 14](#_Toc165308872)

[5.2.3 Timing 14](#_Toc165308873)

[5.2.4 Exporting Data 15](#_Toc165308874)

[6 Implementation and testing 16](#_Toc165308875)

[6.1 Implementation 16](#_Toc165308876)

[6.1.1 OpenCV 16](#_Toc165308877)

[6.1.2 Timing 17](#_Toc165308878)

[6.1.3 Convolution 17](#_Toc165308879)

[6.1.4 Sobel 18](#_Toc165308880)

[6.1.5 K-Means 20](#_Toc165308881)

[6.2 Testing 22](#_Toc165308882)

[6.3 CUDA Optimisation 22](#_Toc165308883)

[7 Evaluation and discussion of results 23](#_Toc165308884)

[8 Conclusion 24](#_Toc165308885)

[References 25](#_Toc165308886)

# Introduction

## Background to the project

Image processing is the process of computing an image to enhance it or modify the image to ease the extraction of data from it. These processes can vary and accomplish different things, anything from removing grain and noise from images, segmenting the image into different sections to detecting and highlighting edges in the image as well as many other tasks.

Graphics Processing Units (GPUs) were originally created to accelerate graphics pipelines such as DirectX and Vulkan using hardware. Over the years they have gained thousands of more cores making them excellent at computing many tasks other than graphics calculations. Image processing algorithms are one of these tasks that have benefited from using the GPUs parallel nature to decrease the time taken for the algorithms to complete.

The are a few different APIs that allow you to write programs for the GPU. A few examples are CUDA, OpenCL, compute shaders in graphics APIs and many others. CUDA is a closed source API made by Nvidia for Nvidia GPUs, which means that programs created using CUDA can only execute on Nvidia GPUs. Recently there has been a translation layer created called ZLUDA that allow non-Nvidia GPUs to run the programs.

## Aims and objectives

### Objective 1 – Research Image Processing Algorithms

To research different image processing techniques and algorithms. This objective will allow me to research commonly used image processing algorithms as well as how they are implemented. By knowing the available algorithms, I will be able to select appropriate algorithms to implement for the purposes of studying the performance differences between the CPU and GPU.

### Objective 2 – Creating CPU implementations

To implement the algorithms chosen in the first objective on the CPU. In order to compare performance differences between the CPU and GPU it requires the algorithms to exist on both CPU and GPU. This objective aims to create the selected algorithms on the CPU to be ready for comparison to the GPU implementations.

### Objective 3 – Research GPU Architecture

To research GPU architectures and how computation is executed on it. By researching GPU architectures, more specifically the architectures of the GPUs which are going to be used to run the image processing techniques, this will help develop an understanding of how to write performant code on the GPU and how GPU programming is different from CPU programming.

### Objective 4 – Creating GPU Implementation

To implement the algorithms on the GPU, which have already been implemented on the CPU. This objective will implement the algorithms chosen in the first objective on the GPU. To ensure that the GPU implementation is as performant as we can get it, we will be using Nsight compute.

### Objective 5 – Compare CPU and GPU Performance

To compare performance metrics produced by running the algorithms with different sized images and parameters, we will be able to create graphs to analyse the differences between the performances. Using the knowledge gained from researching GPU architecture and comparing the results gained to the results of preexisting studies, this will allow us to come to a clearer conclusion.

## Research question

*“How does the performance of image processing Algorithms differ between running on the CPU and GPU?”*

The aim of this project its to analyse the differences between image processing algorithms on CPUs and GPUs. This will involve analysing the differences in time to run, any variation in output result as well as the actual time it takes to develop and implement the algorithms.

# Literature review

## Image Processing

Image processing is the field of running computation on an image resulting in the image being modified to ease the process of extracting data from the image. There is a wide range of modifications that can occur to the image such as removing graininess and other noise, segmenting the image into different sections, detecting, and enhancing edges and many other modifications (Niblack, W. 1986)

Image processing is used in a wide range of applications. The most common application of image processing is noise reduction. All sensory equipment will have some sort of noise that will make it harder to use the data gathered, which also applies to digital image sensors. Images produced by these sensors will have some noise meaning noise reduction/suppression algorithms are used to remove some of the noise while reserving the actual image. (Alvi, F. 2023)

Another commonly used technique is edge detection which is the process of finding edges in an image. Usually the result of the algorithm is a black and white image where all the edges are shaded white and the rest of the image is black. Some algorithms display a gradient depending on how strong the edge issuch as the Sobel operator (see section 2.1.1.1). A smaller set of algorithms display the direction of the edge using two colours. Edge detection can be used in many applications such as image stylisation, parts of computer vision pipeline and many others. (Encyclopaedia of Mathematics. 2011)

Image segmentation is another example of an image processing technique which is a vast field that encompasses a lot of different other techniques. The general purpose of segmentation is to separate different parts of an image from one another. An example of this could be separating different structures in an organ or to separate different parts of an image according to colour.

The techniques we will be focusing on will be edge detection, image segmentation and noise reduction. We will be using the image in figure 1 to display the results of each image processing algorithm.

Fig 1: An image widely used test image processing algorithms. The image is of model Lenna Forsén

Figure 1: An image widely used to test image processing algorithms. The image is of model Lena Forsén (https://imageprocessingplace.com/root\_files\_V3/image\_databases.htm)

### Edge Detection

There is a variety of different edge detecting algorithms. We will be focusing on the classical approach for image processing as modern techniques involve neural networks and machine learning techniques that already leverage the power of the GPU. There are a handful of algorithms we can study which mainly use the method of convolution to produce a result. These algorithms are:

* Sobel
* Canny
* Roberts
* Prewitt

#### Sobel

The Sobel operator is one of the most common edge detection algorithms used, that runs on grayscale images using two separate convolutions. One convolution finds the horizontal edges and the other find the vertical edges. Both convolutions are 9x9 and are rotations of one another.

is the convolution of the kernel for the vertical edges and the is the convolution of the kernel for horizontal edges. These kernels are run for every pixel resulting with two separate images each containing separate edge directions. The magnitude is then found using the two directions calculated previously using the Pythagorean theorem followed by normalizing it. The resulting value is the edge intensity and can be set to the output pixel.

A person wearing a hat

Description automatically generated

Figure 2: The original image (Left) and the result of the Sobel (Right) (Generated using my implementation)

As seen from figure 2 the output result shows a gradient across an edge, where the brighter the pixel appears the stronger the edge is in that location.

A useful side effect of calculating the vertical and horizontal edges separately means that the angle of the edge can be calculated and displayed in colour. There are quite a few applications where this function is used, such as artistic uses of the difference of gaussians (Winnemoller H. 2012).

#### Canny

The Canny edge detector gets its name from John F. Canny who created it. The edge detector starts of by preforming the same steps as a Sobel i.e. it uses the two kernels to find horizontal and vertical edges, then it finds the direction of the edge as well as the gradient (the edge). After performing these steps it continues on to perform two additional steps that result in a cleaner and more usable result.

After the Sobel operator, it finds the centre of each edge changing the image from an image with a gradient over the edges, to an image where all the edges are a single pixel wide. This is possible due to the ability of the Sobel operator to calculate the direction of the edge meaning that we can look at the pixel values across the edge and find the strongest value across the gradient. The pixel with the greatest value will be set as the centre of the edge and the rest of the pixels across the edge will be set to black. The last step in the process is to find the edges that we are sure are actual edges in the image and not a slight change in the colour of an object or noise. To do this we perform hysteresis thresholding which requires two additional parameters to be entered to complete. These parameters are a lower bound and an upper bound. The first step in the threshold is to discard all the pixels under the lower bound entered. Next, it marks all the pixels at or above the upper bound. Finaly, it travels along the edges of each strong edge, as found from the previous step, and marks all the pixels in between the bounds as strong edges and all the pixels that are not connected to any strong edges are discarded. This process of steps ensures that only the strongest and most prominent edges in the image is displayed as well as the edges exact position.

A person wearing a hat

Description automatically generated

Figure 3: The original image (Left) and the result of the Canny edge detector (Right) (Generated using MATLAB)

As seen from Figure 3 the result of the Canny edge detector produces a more precise result compared to the Sobel operator shown in Figure 2. This is not true for most applications as each time a new image is used the lower and upper bound of the hysteresis threshold needs to be adjusted for the result to be usable.

#### Roberts

The Roberts edge detector (also known as the Roberts cross) is a convolution filter similar to the Sobel operator where it also uses two kernels and has the ability to find the direction of the edge. Unlike the Sobel operator, the Roberts does not find horizontal and vertical edges, instead it finds edges going diagonally.

The Robers cross uses two 2x2 kernels. Here we have that calculates the intensity of the edge in one diagonal and that does the same but in the opposite diagonal. After this step the magnitude is calculated using the Pythagorean theorem with the resulting image being similar to the result of a Sobel operator, except it’s a lot dimmer and it doesn’t capture as many edges as the Sobel does. It is also more liable to noise.

#### Prewitt

The Prewitt operator works exactly the same way as the Sobel operator. The only difference between the two operators is the kernels they use. The Prewitt operator use the following kernels:

As you can see the only difference between the Prewitt and the Sobel operators’ kernels are the waits from the closest neighbour across the edge. This means that the resulting image is extremely close to the Sobel result.

#### Edge Detector Conclusion

All edge detectors on display in this section are well documented and have uses in real life but for the purpose of this study only one will be implemented. The edge detector which is going to be implemented will be the Sobel operator. This is due to a few reasons, notably the Sobel operator is the most commonly used algorithm other than the Canny edge detector. Compared with the Canny edge detector, the Sobel operator is much easier to implement. With the time constraint this has been a large factor in this decision.

### Image Segmentation

“Image segmentation is the process of dividing an image into meaningful regions” (Niblack W. 1986). These sections depend on the algorithm used and desired result. For example, an image of an organ could be separated into different structures inside of it using colours. It could also be used to separate different objects from within an image. Segmentation algorithms use a lot of different techniques such as machine learning and neural networks others use typical logical processing techniques.

There are two types of image segmentation, semantic and instance. Semantic segmentation is the simplest as it only looks at individual pixels and determine if those specific pixels should be grouped together. Instance segmentation looks at the overall picture and finds the boundary of an object.

The segmentation techniques discussed will be:

* Thresholding
* K-Means
* Neural network models

#### Thresholding

Thresholding is one of the simplest and most commonly used image segmentation techniques. This technique always returns a binary image (an image containing only 2 colours), usually black and white. The simplest version of this algorithm takes a threshold parameter which is checked against every pixel in the image. If the pixel is below the threshold its value is set to back, otherwise the value of the pixel is set to white. This is the simplest form of image segmentation as it only requires one if statement to be run on every pixel value.

It is also possible to have a variable threshold. This involves this formula:

In this formula is the threshold value, is an arbitrary value (for the purposes of this we will be using 0.18 like show in Wayne Niblack’s book on image processing), is the standard deviation of the pixel and its neighbours, and is the mean value of the pixel and its neighbours.

A collage of white splatters

Description automatically generated

Figure 4: From top left going clock wise, The origrinal image of a cluster of stars, the image with a hight threshold, the image with a low threshold, the image with a varing threshold (Niblack W. 1986)

Figure 4 shows the difference between having a static threshold. The original image shows a cluster of many stars where the stars in the middle of the image merge together and appear brighter than the ones on the edge of the image. With a high threshold most of the stars around the edge of the image are culled and only the stars in the centre remain. With a low threshold all that remains from the stars in the centre is a blob meaning nothing can be comprehended from it. With the varying threshold all the stars on the edges are preserved and most of the stars in the centre of the image are preserved.

#### K-Means

K-means is an unsupervised machine learning clustering algorithm. Machine learning is the process of an application gradually improving on its accuracy over time learning. This being an unsupervised machine learning model means that human input isn’t required to teach the program for it to improve. Clustering is the act of grouping object together in a way where objects in one group have more in common with each other compared to objects in other groups. All this in combination means that k-means clustering is an algorithm that learns improves itself to eventually group objects.

K-means clustering can be used to segment an image. K-means is a semantic segmentation technique as it doesn’t understand/recognise any objects in the image but rather tries to group pixels into k number of colours. This process is also called pixel quantization which is the general process of reducing the number of colours in an image. The result of the k-means cluster is always results in a different result as the machine learning process.

K-means clustering requires 2 parameters, the number of clusters and the number of times the algorithm will iterate. When working with coloured images it is important to think of the image as a 3d graph where each axis represents red, green, and blue (this is reduced to 1d graph if grayscale and 2d if only 2 colours are taken into account) and every pixel is placed somewhere on the graph. There are 2 parts of the k-means algorithm assignment and update. First, we randomly place k number of “means” on the graph. Next, we iterate over the assignment step then the update step the amount of times specified by the parameter. In the assignment step the distance between all the pixels and the means and each pixel is assigned to the mean closes to them. In the update step each groups mean is moved to its centroid (the mean position of the group). By repeating these step the means will move towards large clusters and find pixels that are similarly coloured. At the end of the algorithm the colour of every pixel is set to the colour the group mean it is in. this results in a image with k number of colours.

A person wearing a hat

Description automatically generated

Figure 5: The original image (left), the image quantized with 5 colours using 10 iterations (generated from implementation)

As seen in figure 5, the k-means successfully selects 5 appropriate colours to quantize the image with only 10 iterations. This technique is quite useful to reduce the colours in an image to ever execute the features in an image or to compress the image by reducing the colours presents in the image.

#### Neural Network Models

### Noise Reduction

Noise Reduction and image restoration is a heavily used process that you see everywhere. Almost every image taken in the modern day from phone cameras and medical imagery machines. Especially in the field of medical imagery noise reduction and image restoration are very important as scans using radiation use low doses resulting in an image with a lesser resolution and fainter features causing the image to be more prone to artifacts and noise.

## GPU

Graphical processing Units (GPUs) are a specialized piece of hardware designed to make the rendering of 3D graphics faster and more streamline. Graphics rendering is well suited for parallel tasks meaning that the GPU is highly parallel. This means that the GPU is efficient for tasks that involve a lot of the same calculation performed multiple times on different sets of data but highly inefficient at complicated sequential logical processes that the CPU preforms.

GPUs can trace back to early 70s home video game console which contain specialised chips (often called Picture Processing Units or Video Interface Chips). These chips were designed to process video to be able to be displayed on screen. At the time home computers would not have been able to display images/videos and would only be able to display characters from the machines code page (a file containing all the characters the computer can display) (Singer G. 2023).

### GPU Architecture

### GPU Compute

Before the 2000s GPUs could not be programmed on, with the only way to use the hardware was with the graphics API that were available. This changed with the advent of programable shaders which are programs written to be run on the GPU to calculate positions of vertices’, the lighting and the colour of every pixel. The first GPUs with programable shaders were the GeForce 3 series of GPUs from Nvidia in 2001. These shaders are written in a graphics language such as High Level Shader Language (HLSL) and OpenGL Shader Language (GLSL). These languages were designed to make graphics programming easy and not to do general tasks like compute. They do not have the same types and data structures that are present in typical programming languages such as C++ and C#.

Tasks such as image processing running on the GPU have a special name; General Purpose compute on the Graphics Processing Unit (GPGPU). These sorts of programs running on the GPU would have been written using HLSL or GLSL and use their respective graphics API. This changed when Nvidia released Compute Unified Device Architecture (CUDA), which was the first API that allowed programming of the GPU using already existing programming languages. CUDA requires the GPU running the code to have CUDA cores which only Nvidia GPUs contain. This means that all CUDA programs can only run on Nvidia GPUs. Soon after the release of CUDA both Microsoft (creators of DirectX) and Khronos Group (the organisation in charge of OpenGL) made similar APIs called DirectCompute and OpenCL respectively (Ghorpade J. 2012).

For this project we had to decide on which API/Language to use. Due to only having access to Nvidia GPUs we have decided to use CUDA. Nvidia GPUS allows us to set up and use CUDA with ease making it easy to set up. Unlike DirectCompute, OpenCL and shader languages, CUDA allows us to have native support allowing the development of the algorithms to be both simpler and efficient without the need of downloading and installing third party SDKs.

Recently there has been a development from a software developer in partnership with AMD on ZLUDA. ZLUDA is a translation layer that allows AMD GPUs to run unmodified CUDA applications at near-native performance (Janik A. 2020).

## Language Options and Libraries

### Language

### Lrbraries

## Similar Studies

# Requirements

## Product requirements

When completed the program will be able to run image processing algorithms on images using both the CPU and GPU. The application should be able to display and save the images after the algorithm is run. It should also be able to run an automated test and save performance metric to a file that can be used to turn the data into a graph.

## Functional requirements

### Interface

The program will have a command line interface as it is not designed to be used often and by members outside of the research process. The interface should allow the user to select which algorithm they would like to run and any parameters needed for that algorithm. It should also allow the user to select an image to run the algorithm on. In addition to this, the program should also allow the user to run automated benchmarks with customisable parameters to allow targeted tests.

### Algorithms

The program will have the ability to run three different image processing algorithms.

* Sobel Operator
* K-Means Clustering
* Gaussian Blur

These algorithms will be implemented using both C++ and CUDA for the CPU and GPU respectively.

### Benchmark

The program will have the ability to benchmark algorithms and save the results of the benchmark into a file. The benchmark will have parameters that determine the size of the images tested as well as the type of algorithm to be used. The file that contains all the data gathered should be able to be used by graphing software and be turned into a graph.

## Design constraints

Due to time constraints as well as it being out of scope, there will not be an implementation for reading, saving and displaying of images. A library will be used to handle this part of the program.

# Project Management

# Design

## Software design

For the implementation that the program, that will run the image processing algorithms, OOP (object-oriented programming) techniques will not be used. OOP is great for programs were abstracting a large and complex problem into smaller and more manageable chunks and bundling everything into an object. For the purposes of processing an image there is no abstraction needed. Instead, 2 container classes will be made, one for the C++ implementation and the other for the CUDA implementation. This will allow easier distinction between the 2 versions of the algorithm.

All the CUDA methods will be in the CUDA class implementation file. This will mean that when calling a method from that class the method will be able to dispatch CUDA kernels. This way all CUDA related code will not be visible from outside the class and the interface will not need to be aware of anything happening in the methods.

## Experimental design

### Repeatability

Consistent and reliable performance benchmarking of the implementations is one of the most important steps in the process of developing the program. Due to not having complete control over the CPUs scheduler, its not guaranteed that the processor will run the application consistently and reliably. Because of this fact the program will be required to run the algorithms multiple times and then average the durations. This process is not as important for the GPU implementations as the GPU will not be running any other program unless any other applications, which are open, are being hardware accelerated. Despite this the program will still run the GPU implementation of the algorithms multiple times and average the resulting times.

### Representability

To produce a range of data to be able to analyse, the benchmark will run the CPU and GPU implementations one after the other on increasingly larger images. For the convolution algorithms the program will generate black images as they do vairy in time complexity depending on the image. This means that no matter what the image contains the time to complete the filter stays the same if the size of the image also stays the same. This differs to an algorithm such as k-means as its possible to stop the algorithm midway and still obtain the same result. This can be done by checking if the mean of the clusters moves from one iteration to another. If the means do not move this means that the algorithm already found the best possible solution and no additional work is needed to be done. This means that if the whole image is black no matter where the cluster means are placed, they will all tend towards the same point and finish the algorithm after the first two iterations. This means that the image will need a variety of values to successfully benchmark the algorithm. Luckily in OpenCV it is possible to generate images with randomly generated noise. This ensures that the algorithm will run for more than two iterations and give a better view on performance.

### Timing

Timing the algorithms will be done by the actual method that contains the algorithm. This is due to two reasons, first, this will mean that setting up the stack for the method to run for will not be recorded as part of the algorithms time to run. Secondly, on the GPU implementation the image is required to be copied from system memory to the GPUs memory. This is a very inefficient process meaning that the majority of the time measured will be copying the data from one place to the other.

### Exporting Data

At the end of the Benchmark the results will be exported to a document that will be able to be read by graphic software and turned into a graph. For this purpose, the CSV (comer separated value) file format will be used. The first row will contain the title of each column, Pixels, CPU time, GPU time. Following this the next rows will contain the pixel count of the image followed by the average time for the CPU algorithm followed by the GPU’s algorithm’s time. This means that any file will be able be read by programs such as excel and easily be turned into a graph.

# Implementation and testing

## Implementation

For the implementation of Visual Studio 2022 was used. The CUDA Runtime temple project is used as the startup project. All the auto generated code was removed and 3 extra files were maid ‘*main.cpp*’, ‘*ImageProcessing.h*’ and ‘*CppImageProcessing.cpp*’. The existing .cu file was renamed to ‘*CUDAImageProcessing.cu*’. The header file will contain the class definition for both image processing container classes. The respective image processing files will contain the implementation for the C++ and CUDA versions of the algorithms. The ‘*main.cpp*’ will contain all the code to allow the user to run the algorithms and benchmark them.

### OpenCV

The OpenCV library for C++ can be downloaded from OpenCV website. The .zip file downloaded from the website contains all the required files for all the languages OpenCV supports. For C++ the files required are, the include folder from withing the source folder, and both the lib file and the bin file from within the build/x64 folders. These files need too selected in the project properties inside Visual Studio. In addition to the already mentioned folders, ‘*opencv\_world480.dll*’ is required in the executable folder. This file only works with the release mode of the executable, to be able to run the executable with debug mode the ‘*oepncv\_world480d.dll*’ is needed. Both these files can be found in the bin folder.

OpenCV provides simple methods to allow the developer to read, write and display images. For the purpose of this project the only OpenCV methods/classes used are:

cv::mat

cv::imread()

cv::imwrite()

cv::imshow()

cv::waitKey()

cv::hconcat()

cv::mat is the class that contains images. It can contain a variety of images, from grayscale images to images with full colour. It can also represent each pixel/subpixel as different types such as an unsigned char (8 bits) to a float (32 bits). This makes Mat a very versatile class that can contain basically any image.

Both cv::imread and cv::imwrite are two sides of the same coin. cv::imread returns a Mat of the image in the file entered as a parameter. The cv::imwrite method doesn’t return any values. It takes 2 peramiter an image and a file name. it saves the image in the file entered.

cv::imshow opens a new window to display the a image. It takes 2 parameters, the name of the window as a string and an image. cv::waitKey is an extension of cv::imshow as it is necessary if the developer wants to stop the program while the window is open. It takes one parameter which the delay between the user pressing a button/key and the program continuing.

cv::hconcat is a method that return an image. it takes 3 parameters, the first image, the second image and then an output image. it joins two images together horisontaly and puts the resulting image into the output image.

### Timing

Timing was done using the std::chrono namespace. This namespace contains all the methods and classes required to do anything with time in C++. Each image processing algorithm’s method returns a microsecond which contains the number of microseconds the algorithm took to run.

To measure the time it takes to run an algorithm we first take a the time at the start of the of the algorithm using high\_resolution\_clock.now() which gets the current time. For the CUDA this is after all the data is copied from system memory to the GPUs memory and just before the CUDA kernel is initiated. For the C++ implementation this is usually the start of the method as all the data required for the algorithm is already on system memory, meaning no moving of data is required. Next, we wait for the algorithm finishes and use the hight\_resolution\_clock.now() to get the time at what the algorithm finishes. After getting the end time we can take away the 2 time\_points from each other and use duration\_cast() to turn the resulting time\_point into a microsecond that can be interpreted into a number.

Doing this provides a high resolution of measurement and an accurate measurement. CUDA does have its own way to measure the time a kernel ran for. However using std::chrono for both C++ and CUDA implementations means that even if using the hight\_resolution\_clock adds some time algorithms, it will add similar amounts to both implementations making it less impactful.

### Convolution

A large part of the image filters in convolution. This means that both C++ and CUDA implementations will need to have the ability to convolute images with kernels. The implementation of both C++ and CUDA methods are the same as both will be running in serial. This is because the CUDA thread will run its own convolution on the pixel its responsible for.

The pseudocode is as follows:

fn Convolution (imageIn, kernel, x, y, width, height, kernelSize)

sum = 0

halfKernelSize = (kernelSize - 1) / 2

for (kernelY in range -halfKernelSize too halfKernelSize)

currentY = kernelY + y

if (currentY > height) then

continue loop

end if

for (kernelX in range -halfKernelSize too halfKernelSize)

currentX = kernelX + x

if (current > width) then

continue loop

end if

sum = sum + imageIn[currentX + currentY \* width] \* kernel[kernelX + halfKernelSize + (kernelY \* halfKernelSize) \* kernelSize]

end for

end for

return sum

This function takes in seven parameters, imageIn which is the image being processed, kernel which is the kernel that is being used to filter the image, x and y which are the coordinates of the current pixel, width and hight which are the dimensions of the image, and kernelSize which is the size of the kernel. It is important to note that only square kernels of odd size can be used using this function.

First we set a sum variable to 0, this will store the sum of the kernel as we go along. Next we find the distance it looks away from the current pixel. For example if we have a kernel size of 5 the convolution will take values from 2 pixels away from the current pixel. We calculate this value by taking 1 away from the kernelSize and dividing it by 2. After calculating this value and storing it in halfKernelSize we can iterate over the kernel in both x and y direction using it. We add this the current x and y iteration to the pixels x and y position to check if the current iteration has a valid pixel position. If there is no pixel in this position it continues to the next iteration. If the current x and y positions are valid it multiplies the pixel at the current position with the kernel value corresponding to that pixel and add the value to the sum. At the end function returns the sum to the method that called it.

### Sobel

For the Sobel operator the implementation will work as expected in part 2.1.1.1. Both C++ and CUDA implementations take the same parameters. These parameters are, imageIn which is the picture being processed, imageOut which is the output image that will contain the result of the function, and the width and height which contain the dimensions of the image. Both implementations return the time it took to execute the Sobel operator.

#### C++

The first thing the function does is take the time it started at and stores it in a variable named start.

At the start of the CPU implementation, we declare two arrays, Gx and Gy. These arrays represent the Gx and Gy kernels shown in the Sobel section earlier. The Gx and Gy arrays are one dimensional version the kernels.

After declaring the two kernels the function goes into two nested for loops. The outer loop iterates over the y of the image and the inner loops iterates over the x. Doing the x and y in this order means that the from one pixel to another it will decrease the chances for a cache-miss as the image is stored sequentially from left to right, top to bottom in memory. Inside the loops we declare two variables sobelX and sobelY which store the results from two convolutions, one for Gx and the other for Gy. Next the magnitude of sobelX and sobely are calculated using the Pythagorean theorem. Finaly the magnitude is clapped between 255 and 0 and set as the pixel value of the output image.

After this process is done the end time is taken and stored in its own variable named end. The start time is taken away from the end time and the result is returned from the function.

The pseudocode for this looks like this:

fn Sobel (imageIn, imageOut, width, height)

start = time.now()

Gx = [1, 0, -1, 2, 0, -2, 1, 0, -1]

Gy = [1, 2, 1, 0, 0, 0, -1, -2, -1]

for (y in range height)

for (x in range width)

sobelX = Convolution (imageIn, Gx, x, y, width, height, 3)

sobelY = Convolution (imageIn, Gy, x, y, width, height, 3)

magnitude = squrt((sobelX \* sobelX) + (sobelY \* sobelY))

imageOut[x + y \* width] = clamp(magniture, 0, 255)

end for

end for

end = time.now()

return end – start

When first programming the algorithm the Sobel convolutions were saved into a separate image and then a second loop would calculate the magnitude of the Sobel using the 2 images created. This increased the time to perform the operations as it involved storing and reading more information which is a time expensive operation as well as going over the image data twice which would in of itself double the time it takes to execute the algorithm.

#### CUDA

For the CUDA implementation we need to first assign memory on the GPU where we will save the image data as well as copy the data from system memory to the GPU. First the function allocates memory for both the input image and the output image. After this the function copies the input image onto the GPUs memory. It does not need to copy data for the output image as it does not store any data yet.

Next the function sets two variables to store the bock and grid dimensions for the CUDA kernel named blockDim and gridDim respectively. blockDim is always set to be 8\*8 this means that we need to calculate the grid dimensions required to have one thread per pixel. We do this by dividing the width of the image by 8 and then rounding up. The same is done for the hight. This results in the number blocks we need in the x and y dimension of the grid to cover the whole image.

After this we can start out timing process by measuring the time and storing it in a variable named start.

After measuring the time, we can dispatch out CUDA kernel with the block and grid size calculated earlier and with the parameters required. Strate after dispatching the kernel the function needs to synchronise with the GPU meaning that the function will wait until the GPU is finished running the kernel.

Lastly the function takes the end time and saves it into a variable named end and cleans up the GPU memory used after copying the imageOut data from the GPU back to system memory. It returns the value resulting from taking the end time taken away from the start.

The pseudocode for this would look something like:

fn Sobel (imageIn, imageOut, width, height)

cudaImageIn = CUDAMemoryAllocation (width \* height \* size of uchar)

cudaImageOut = CUDAMemoryAllocation (width \* height \* size of uchar)

CUDAMemoryCopy (cudaImageIn, imageIn, host to device)

blockDim = (8,8)

gridDim = (width / 8, height / 8)

start = time.now()

cudaSobel <<<gridDim, blockDim>>> (cudaImageIn, cudaImageOut, width, height)

cudaDeviceSynchronize()

end = time.now()

CUDAMemoryCopy (imageOut, cudaImageOut, device to host)

CUDAFreeMemory (cudaImageIn)

CUDAFreeMemory (cudaImageOut)

Return start – end

The Sobel algorithm in the CUDA implementation is not located in the C++ method but instead located in the CUDA kernel. In C++, CUDA kernels which can be dispatched from the CPU are declared using \_\_global\_\_. This means that the Sobel kernel must be declared using this notation.

Inside the kernel the program first checks if the current thread corresponds to a valid pixel location. This is done by finding the position it is in charge. After doing this it checks if the position is within the image boundaries where it will exit the kernel if its outside the boundary.

Next it declares the Gx and Gy arrays similarly to the C++ implementation of the algorithm. After this it preforms the convolution with both Gx and Gy storing the results into separate variables named sobelX and sobelY. Then these variables are used to calculate the magnitude which is then proceeded to be clamped between 255 and 0. This clamped value is then set to the value of imageOut at the apropriate location.

The pseudocode looks like this:

\_\_global\_\_ fn SobelCUDA (imageIn, imageOut, width, height)

x = threadIdx.x + blockIdx.x \* blockDim.x

y = threadIdx.y + blockIdx.y \* blockDim.y

if (x > width or x < 0 or y > height or y < 0) then

return

end if

Gx = [1, 0, -1, 2, 0, -2, 1, 0, -1]

Gy = [1, 2, 1, 0, 0, 0, -1, -2, -1]

sobelX = Covolution (imageIn, x, y, Gx, width, height, 3)

sobelY = Covolution (imageIn, x, y, Gy, width, height, 3)

magnitude = sqrt((sobelX \* sobelX) + (sobelY \* sobelY))

imageOut[x + y \* width] = clamp(magnitude, 255, 0)

The process of initializing CUDA kernels takes a long time but the actual CUDA kernel is a lot shorter than the C++ implementation. The main reason for this is the lack of nested for loops. These loops are not required as each pixel is run independently from every other pixel meaning no looping is required.

### K-Means

The K-means implementation will only work for grayscale images as the process of making the algorithm work for coloured images is similar in result but involves more complexity into the implementation. Both C++ and CUDA functions take the same parameters which are, imageIn, imageOut, width, height, k, iterationNum. ImageIn is the image being processed, imageOut is the result of the K-Means algorithm, width and height are the width and hight of the image, k is the number of clusters/colours the use wants the result to contain and iterationNum is the number of iterations the K-Means algorithm should run for to get the desired result. Both versions of the function also return the time it took to run.

#### C++

The first step in the function is to create 2 arrays, one to store the cluster values and one to store the cluster that every pixel belongs too. The first array will be of length equal to the number of clusters and be called cluster. The second arrays length will be equal to the number of pixels in the image. Next the function assigns as random value to each value in the cluster array. These values will represent the cluster centres. After assigning all these values we store the current time into a variable named start.

This is where the actual K-Means algorithm starts. As discussed in section 2.1.2.2 the algorithm needs to repeat the assignment and update steps to get the best results. For this we start a for loop that will repeat the number of times specified by the iterationNum variable. The first of the two stages is the assignment stage where each pixel gets assigned a cluster. For this the function uses two nested for loops. The outer loop, loops through all the pixels in the image. The inner loop, loops over every cluster in the cluster array. Inside the two loops the function finds the cluster that has the closest value to its own value. When it find the closest cluster it assigns the clusters position in its array to the position in the pixel array that corresponds to the current pixel.

Next, still inside the iteration loop, the update stage begins. This stage begins with a variable called change being set to false. This variable will track if any of the clusters changed values. After, two nested for loops are started. The outer loop iterates over the cluster array and the inner loop iterates over every value in the pixel array. Inside the two loops the function checks the number of pixels that are part of the cluster as well as summing the value of those pixels. Next outside the inner loop and inside the outer loop, the function averages the sum of the pixel values and sets the result as the cluster’s value. The last part of the Update section uses the change variable to check if the cluster positions changed. If no cluster changed the function would exit the iteration loop to finish algorithm.

When the function exits the iteration loop, the function does its last loop to set the pixel values in imageOut equal to the values in of the clusters each pixel is a member of.

This is the end of the K-Means algorithm meaning the end time can be taken and saved into a variable. The last step in the function is to return the value of the start time minus the end time.

The pseudocode for the function looks like this:

fn K-Means (imageIn, imageOut, width, height, k, iterationNum)

clusters[k]

pixles[width \* height]

for each (cluster in clusters)

cluster = rand(0,255)

start = time.now ()

for (iteration in range iterationNum)

for (pixel in range width\*height)

minDistance = 255

for (cluster in range k)

distance = abs(clusters[cluster] – imageIn[pixel])

if (255 < distance) then

minDistance = distance

pixels[pixel] = cluster

end if

end for

end for

change = false

for (cluster in range k)

sum = 0

amount = 0

for (pixel in range width\*height)

if (pixels[pixxel] == cluster) then

sum = sum + imageIn[pixel]

amount = amount + 1

end if

end for

if (amount != 0) then

newValue = sum / amount

if (newValue != clusters[cluster]) then

clusters[cluster] = newValue

change = true

end if

end if

end for

if (!change) then

brake loop

end if

end for

for (pixel in range width\*height)

imageOut[pixel] = clusters[pixels[pixel]]

end for

end = time.now ()

return start – end

#### CUDA

The CUDA version of the K-Means algorithm varies drastically from the C++ implementation. The main reason for this is the problem of soring the cluster information. In the C++ implementation the cluster values and each pixels cluster were stored in arrays. In the CUDA implementation there are three arrays instead. All the arrays are of length k. The first array is called clusterValue which stores the values of the clusters. The second array is called clusterSum which will hold the sum of all the pixels in the clusters. The last array is called clusterAmount which will hold the amount of pixels contained in the cluster.

Just like in the C++ implementation the function generates random numbers to store in the clusterValue array. The rest of the arrays are all populated with the value zero. This these arrays as well as the imageIn, iamgeOut and a new array to contain the each pixels cluster are then allocated space on the GPU’s memory and copy the data of from the arrays to the allocated areas.

Next the function sets up the block and grid dimensions for the CUDA kernels. The size of the block for the per pixel kernel will be 8\*8 where as the size of the block for the per cluster kernel will be the number of clusters. This means that the per pixel kernel needs to have a dynamically sized grid. This is done by dividing both image hight and width by 8 and rounding up the result.

After these steps the function starts timing the algorithm meaning that it makes a variable and stores the current time. Proceeding this is the K-Means algorithm starts. The algorithm starts of with a for loop looping over the number of times specified by the iterationNum parameter. The reason why this part of the algorithm appears in the C++ part of CUDA is because there is no way to iterate multiple times over 2 separate CUDA kernels in a way that ensures ones completion before the other starts in CUDA. This means that the iteration part of the unsupervised machine learning algorithm needs to be written in the C++ that initiates the CUDA kernels. Inside the loop 2 kernels are initiated, first the kernel responsible for the assignment section of k-means, next the kernel responsible for the update section. It is important to note that the host is synchronised with the device after both kernels are initiated meaning that the CPU waits until the kernels have finished before continuing.

After the loop is finished the a third CUDA kernel gets initiated which sets the output pixel to the value corresponding to the cluster the pixel is part of. This is the last part of the algorithm meaning the next step is to take the end time. The function then return the end time minus the start time.

The pseudocode for this looks like this:

fn K-Means (imageIn, imageOut, with, height, k, iterationNum)

clusterValues[k]

clusterSums[k] = {0}

clusterAmount[k] = {0}

for (index in range k)

clusterValues[index] = rand(0, 255)

end for

cudaImageIn = CUDAMemoryAllocation (width \* height \* size of uchar)

cudaImageOut = CUDAMemoryAllocation (width \* height \* size of uchar)

cudaClusterValue = CUDAMemoryAllocation (k \* uchar)

cudaPixelCluster = CUDAMemoryAllocation (width \* height \* size of int)

cudaClusterSum = CUDAMemoryAllocation (k \* size of int)

cudaClusterAmount = CUDAMemoryAllocation (k \* size of int)

CUDAMemoryCopy (cudaImageIn, imageIn, host to device)

CUDAMemoryCopy (cudaClusterValues, clusterValues, host to device)

CUDAMemoryCopy (cudaClusterSums, clusterSums, host to device)

CUDAMemoryCopy (cudaClutserAmount, clusterAmount, host to device)

pixelBlockDim = (8, 8)

clusterBlockDim = (k)

pixelGridDim = (width / 8, height / 8)

clusterGridDim = (1)

start = time.now ()

for (iteration in range iterationNum)

pixelClusterSelect <<< pixelGridDim, pixelBlockDim >>> (cudaImageIn, width, height, k, cudaPixelClusters, cudaClusterValues, cudaClusterSums, cudaClusterAmount)

cudaDeviceSynchronize()

clusterUpdate <<< clusterGridDim, clusterBlockDim >>> (cudaClusterValues, cudaClusterSums, cudaClusterAmount, k);

cudaDeviceSynchronize()

end for

setPixels <<< pixelGridDim, pixelBlockDim >>> (cudaImageOut, width, height, cudaPixelCluster, cudaClusterValues)

cudaDeviceSynchronize()

end = time.now ()

CUDAMemoryCopy (imageOut, cudaImageOut, device to host)

return end – start

The first kernel in the function is responsible for the assignment part of the k-means algorithm. The kernel starts off by finding the pixel it is responsible for by finding its x and y position. Next checks if the x and y are valid. Then it initiates two variables, one that saves the current minimum distance to a cluster and the other which cluster is closest. It then checks each cluster and de distance to them. If the cluster is closer than the current closest cluster, it sets the two previously mentioned variable appropriately.

After checking all the clusters, it preforms two atomic adds. Atomics are a special type of operator that only take one operation to complete. This means that between the operation starting and it finishing no other thread can read/write to the data. It can also mean that the current thread that peace of data so that no other thread can access it. These operators are used to update the clusterValues and clusterAmount arrays.

Lastly the kernel sets the pixelsClusters array to the cluster its closest to. The pseudocode for this looks like:

\_\_global\_\_ fn pixelClusterSelect (imageIn, imageOut, width, height, k, pixelClusters, clusterValues, clusterSums, clusterAmount)

x = threadIdx.x + blockIdx.x \* blockDim.x

y = threadIdx.y + blockIdx.y \* blockDim.y

if (x > width or x < 0 or y > height or y < 0) then

return

end if

minimumDistance = 255

closestCluster = 0;

for index in range k

distance = abs (imageIn[x + y \* width] – clusterValues[index])

if (distance < minimumDistance) then

minimumDistancec = distance

closestCluster = index

end if

end for

clusterSums[closestClutser] atomicAdd imageIn[x + y \* width]

clusterAmounts[closestCluster] atomicAdd 1

The next kernel called is responsible for updating the cluster positions. This kernel starts by getting the threads x position. After this it checks if the thread is in range of k. next it preforms three operations. First it finds the average of clusters pixels using the clusterSums and clusterAmout arrays. Next it sets both clusterSums and clusterAmount to zero. The pseudocode for this kernel looks as follows:

\_\_global\_\_ fn clusterUpdate (clusterValues, clusterSums, clusterAmount, k)

x = threadIdx.x + blockIdx.x \* blockDim.x

if (x > k) then

return

end if

clusterValues[x] = clusterSums[x] / clusterAmount[x]

clusterSums[x] = 0

clusterAmount[x] = 0

The last kernel in this function is the kernel that finalises the result of the k-means. It does this by setting the values in imageOut to the value of the cluster each pixel belongs to. The pseudo code for this looks like this:

\_\_global\_\_ fn setPixels (imageOut, width, height, pixelCluster, clusterValues)

x = threadIdx.x + blockIdx.x \* blockDim.x

y = threadIdx.y + blockIdx.y \* blockDim.y

if (x > width or x < 0 or y > height or y < 0) then

return

end if

imageOut[x + y \* width] = clusterValues[pixelCluster[x + y \* width]]

These set of kernels in this configuration allows the function to parallelise the k-means algorithm on every step of the process.

## Testing

## CUDA Optimisation

# Evaluation and discussion of results

# Conclusion

References

Singer, G. (2023). The History of the Modern Graphics Processor. *Techspot*. Available online: <https://www.techspot.com/article/650-history-of-the-gpu/> (accessed 10/04/2024).

Niblack, W. (1986). *An introduction to digital image processing,* Englewood Cliffs, N.J: Prentice-Hall International.

Alvi, F. (2023). Computer Vision and Image Processing: Understanding the Distinction and Interconnection [Blog post]. OpenCV. Dec 13. <https://opencv.org/blog/computer-vision-and-image-processing/> (accessed 13/04/2024).

Encyclopaedia of Mathematics. (2011). Edge Detection. Available Online: <https://encyclopediaofmath.org/index.php?title=Edge_detection&oldid=17883> (accessed 13/04/2024).

Bhardwaj, S., Mittal, A., (2012). A Survey on Various Edge Detection Techniques. *Procedia Technology.* 4. 220-226.

Winnemoller, H. & Kyprianidis, J. E., & Olsen, S., C. (2012). XDoG: An eXtended difference-of-Gaussians compendium including advanced image stylization. *Computer &Graphics. 36(6). 740-753.*

Ghorpade, J. & Parande, J. & Kulkarni, M. & Bawaskar, A. (2012). GPGPU Processing in CUDA Architecture. *Advanced Computing: an International Journal*. 3(1), 105-120.

Janik, A. (2020). ZLUDA. Available online: <https://github.com/vosen/ZLUDA> (accessed 17/04/2024).