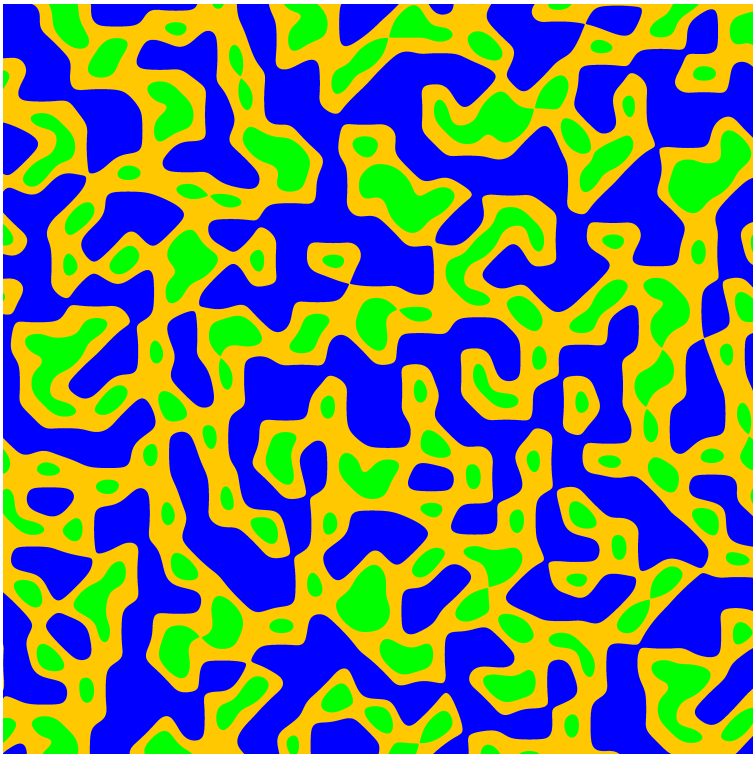

Accelerating geographic processing using GPUs

Implemented in OpenCL

Master's thesis in Computer Science-Algorithms, Languages and Logic

Alexander Jaballah
Rafael Mohlin

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY AND UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2017

Master's thesis 2017:06

Accelerating geographic processing using GPUs

Implemented in OpenCL

Alexander Jaballah
Rafael Mohlin

Department of Computer Science and Engineering
Division of Algorithms
Chalmers University of Technology And University Of Gothenburg
Gothenburg, Sweden 2017

Accelerating geographic processing using GPUs

Implemented in OpenCL

Alexander Jaballah
Rafael Mohlin

Supervisor: Johan Persson from Qlik, Andreas Abel from Department of Computer Science and Engineering
Examiner: Carlo Furia, Department of Computer Science and Engineering

Master's Thesis 2017:06
Department of Computer Science and Engineering
Division of Algorithms
Chalmers University of Technology And University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Visualisation of geography generated with Simplex noise.

Typeset in LaTeX
Gothenburg, Sweden 2016

Accelerating geographic processing using GPUs
Implemented in OpenCL
Alexander Jaballah
Rafael Mohlin
Department of Computer Science and Engineering
Division of Algorithms
Chalmers University of Technology And University Of Gothenburg

Abstract

In This thesis, we present how a geographical process can be increased in execution time and asymptotic complexity, by moving the processing algorithm from the Central Processing unit (CPU) to the Graphics Processing Unit (GPU). We also investigate different memory strategies on the GPU in order to further decrease the execution time for the algorithm. To improve the asymptotic complexity two new algorithms are investigated and implemented, the first algorithm is based on the concept of separability, and the second algorithm is a state-of-the-art algorithm called Gaussian filter kernel. The outcome of our work is an approach of how algorithms on the CPU that has great potential of parallelism can be moved to the GPU to improve the execution time. In order to evaluate the different algorithms, tests regarding the execution time and outcome accuracy were conducted. Lastly, we concluded the overall success of the improvement regarding the execution time and reduction for the asymptotic complexity.

Keywords: Image processing, CPU, GPU, MATLAB, OpenCL, RGBA packing, Local caching, SVD, Guassian filter, Second-order shit property of DCT-5.

Acknowledgements

This Master thesis has been carried out under the Department of Computer Science and Engineering at Chalmers University of Technology. We want to thank our academical supervisor Andreas Abel for his support, and feedback on writing. We want to express our deepest gratitude towards our examiner Carlo A. Furia for his support and feedback during the thesis. We want to thank Qlik for providing the thesis topic, a workplace and tools for testing. Furthermore, we would like to express our gratitude towards Johan Persson for his willingness to help, and enthusiasm towards the thesis topic.

Alexander Jaballah and Rafael Mohlin, Gothenburg, 06-2017

Contents

1

Introduction

Much of the processing taking place in today's datacenters is performed exclusively on Central Processing Units (CPUs). The CPU is well suited for everyday computing, and as a clear majority of algorithms can be implemented in a relatively efficient way for the CPU. Many of these processing tasks are very complex and involve huge amounts of data.

Powerful Graphical Processing Units (GPUs) are available for most computer configurations and provide huge processing power when properly exploited[1]. By moving some parts of the computations from the CPU to the GPU, one can reduce the total execution time significantly. This can play a great part when processing large amounts of data, where long execution times make the computation unfeasible. However, depending on the input parameters and the algorithm, in some cases it may be slower to run the algorithm on the GPU instead of the CPU. This is much dependent on the algorithm, what its purpose is and how it is implemented.

One type of processing algorithms that might be a good candidate for GPU's are geographic computations. These usually involve large amount of two-dimensional data, with several attributes associated with each point in data space. To enable easier management of this data simplification or aggregation can be done over the data, which can easily be partitioned into many small parts. This processing method suits a GPU very well, as it is especially designed to do many small jobs in parallel.

In this project, we will investigate a geographic algorithm currently implemented on the CPU, and how we can reduce its execution time. This will be done in three different phases: In phase one, a one-to-one replica of the algorithm will be translated onto the GPU, there will also be an investigation on different memory access strategies to reduce the execution time. The aim of the second phase is to experiment with different strategies to reduce the complexity of the algorithm. In the third phase, we will implement a state-of-the-art algorithm that is like the algorithm in question and known to be efficient by existing studies[2]. For each phase an evaluation will be done in which the speed up will be measured and discussed. Additionally, tests will be conducted to reveal possible differences in the outputs of the different phases. The three different phases are listed below.

- Phase 1 : One-to-one replica from CPU to GPU

- Phase 2 : Reduce the complexity of the algorithm
- Phase 3 : Implement a state-of-the-art algorithm

A co-operation has been established with Qlik who will be providing the original algorithm and the tools for the project. Qlik is a company that specializing in data visualization, Qlik also provides different geographical services. To provide these services Qlik has different data in different forms, where one form is data in vector format. This vector data is processed by Qlik in three different steps, rasterize and divide into sections, optimize to suit different services and Stitch it together, vectorise for later use. This thesis will be focusing on the second step. Today Qlik uses an algorithm to do the optimization. The focus will improve the algorithm and to decrease the processing time needed and thus increase the efficiency.

1.1 Problem

Qlik provides several map and geographical related services to their customers. These services all share a common source of map data, which the company also maintain. Raw map data is delivered to the company in a vector format which is not suitable for many of the business scenarios that the company wants to use it for. Thus, they process the raw data in three step: First the map data is rasterized and divided into sections. Then each section is optimized in different ways to better suit the different services. Lastly the sections are stitched together, vectorised, and then stored for later use.

This project will focus on the optimization step of this process, as currently most time spent is in this step. Processing all the available data the company has today requires more than 80 CPU days when run on their hardware. Some quick fixes have been set in place to try and limit this run time, one of them being reduced resolutions of both the rasterized map data and the optimization algorithms. As such if a significant speed up can be achieved, the tradeoff between performance and quality can be reduced.

1.2 Purpose and goals

In this project, we will investigate how the run time of a process can be improved by utilizing a GPU for some parts of the algorithm. We will also try to improve the complexity of the already existing algorithm, further increasing the efficiency. At last we will implement an already known low complexity algorithm to improve the processing time. To achieve this, we will develop test-cases to compare the accuracy between the original, the improved and the new suggested algorithm.

When evaluating the improved algorithms, the primary challenge will consist of evaluating it in a fair way. The aim is to reduce the run time of the processing by reducing the complexity. Because we will try to improve the already existing algorithm as well as implementing a new algorithm, another challenge will be to ensure possible differences in the output. This is thoroughly considered in relationship to the improved complexity, and will be ensured by measuring the different run times for the implementations. Several tests will be conducted to ensure that the algorithms produce the same outcome.

1.3 Limitations

We will primarily investigate the performance a GPU can deliver with respect to the company's geographical workload. The performance of the original CPU implemented algorithm will be the baseline for the measurement of speed up.

We will not implement the optimized algorithms on the CPU, but instead try and leverage the GPU's architecture in conjunction with the reduced complexity. A CPU implementation of these could very well be of value to the company, but with time restrictions in mind we believe that it is more important to focus on what we believe will be the best performing solution.

There exists research on topics relevant to the type of algorithm that the company was first inspired by when developing their own. We consider the three phases previously described to be of a large enough scope, and as such we will not implement or discuss other solutions or research that could be seen an obvious choice.

1.4 Related Work

This section is divided into the different phases that follows the structure of the thesis, and the relevant related work for each Phase is under respective Phase section below.

General-purpose computing on graphics processing units has become relevant for optimization processes [3]. There exist several studies regarding this topic, where the overall goal often is a comparison of the running time on a GPU contra on a CPU. "Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU" [4] is a study where they analyze the performance of an Intel i7-960 compared to a Nvidia GTX280. They conclude that the performance gap was not always is that wide, one factor for this is that they have focused on optimizing the CPU usage. This study shows that for some cases it is worth it to just optimize the CPU code instead of implementing new GPU code.

1.4.1 Phase 1 Local caching

Since the company is providing the algorithm we have not put any effort into finding related work for this algorithm. Instead we focused on finding ways to optimize the OpenCL C code to achieve a better performance. In a case study "Case study: High performance convolution using OpenCL ___local memory" done by CMSoft they show how much an efficient implementation of OPENCL C can affect the performance[5]. In the case study they compare a greedy convolution filter algorithm implementation with two improved filter algorithms using right memory spaces off the GPU. The first improved filter is based on using the constant space in the memory region, the work-items have faster access to the data placed here. The second improved filter is based on caching information in the local space. The local space is a fast memory that is shared between work-items that is within the same work-group. They concluded that with a kernel of a size up to $7 * 7$ the speed-up ability for the constant memory is 4%, while the speedup for local caching was at least is 10% for any filter size. This case study shows that the use of right memory space can contribute to better performance. This is a concept that will be adapted in this thesis for the first phase to improve the processing run-time.

1.4.2 Phase 1 RGBA Packing

Another concept by the same authors is Red Green Blue Alpha (RGBA)-packing. The authors have done a case study "OpenCL Image2D Variables" [6] where they explore the performance speed-ups when using images2d variables. The theory behind the case study is that GPUs are optimized by design to cache and sample textures quickly. In this study, they also listed some facts and reasons why images should be used to improve the performance. Their results are based on border filter with the kernel size up to $7 * 7$. They succeeded to achieve a speed-up that was up to 3 times better than the regular filtering algorithm. This concept is also tested in the article "RGBA Packing for Fast Cone Beam Reconstruction on the GPU" [7]. They introduce a method that enables the possibility to pack projections into RGBA data without rebinding, as well as reuse of data to save memory bandwidth. They succeed to achieve a speed up to 3 times compared with their GPU implementation without the RGBA packing. This concept will also be adapted in this thesis for the first phase to improve the run-time.

1.4.3 Phase 2

There is an article performed by Intel (An investigation of fast real-time GPU-based image blur algorithms) [8]. In this article they have a 2D Gaussian blur filter which has the complexity of $O(N^2)$ with $N * N$ kernel implemented in a Naïve way. To improve the complexity, they adapted the concept of kernel matrix separability, which means that the kernel matrix is divided into to one $nx1$ horizontal vector and

one $1xn$ vertical vector. Since the 2D Gaussian filter kernel is perfect separable it only requires one horizontal and one vertical vector to recreate the original kernel matrix. Both the horizontal and vertical vectors can then be applied on the image for processing in parallel and thus the complexity is reduced to $O(n)$. The concepts of separable will be used to implement Phase 2 in this thesis.

1.4.4 Phase 3

With the respect to Phase 3 and Gaussian filter there is many different studies made. For this thesis, we found an article evaluating and implementing the Gaussian filter, this article will be the foundation for this thesis Phase 3 implementation. The article Fast Gaussian Filter With Second-order Shift Property Of DCT-5 [2]. In the article, they provide a comparison with different state-of-the-art implementation of the Gaussian filter algorithm. They compare the implementations regarding complexity and the number of arithmetic operations. The implementation presented in the article is has a complexity of $O(1)$ and an overall minimal of arithmetic operations compared to other constant Gaussian filter implementations. The key for their implementation is that they use the Second-order shift property of Discrete Cosine Transform-5 to calculate DCT coefficient recursively.

1.5 Outline

Chapter 2 will present the algorithms for the different Phases to understand the implementation in Chapter 3. Chapter 2 also contains relevant GPU theory and mathematical theory. Chapter 3 will describe implementation choices for the chosen algorithms, encountered problems, and suggested solutions. Chapter 4 will contain results and evaluation of the different Phases. Chapter 4 will be followed by a discussion chapter 5 and a conclusion chapter 6.

2

Theory

This chapter contains relevant information about the algorithm that already exist on the CPU, as well as information about the algorithms that will be implemented on the GPU. There is also a brief theory about General-Purpose computing on Graphics Processing Units (GPGPU) programming and how to one could go about optimizing code for GPUs.

2.1 Data set

The source data set exists in the form of vector graphics, which the company rasterizes before further processing. The different vector shapes define areas that describes geographical properties, such as water bodies, buildings, forests etc. After rasterization, the data set can be thought of as an image, where each pixel has a single value, representing the geographical property beneath it. In this thesis, this concept is referred to as the pixel's class. The class is a value between 0 and 255, where a given value uniquely identifies the geographical property in the image. However, the classes' values are not strictly defined and is determined at run time for each image. This ensures that the class values can be packed together.

In this project, we will use computer generated test data during development. This provides better control over the data, and thus enables us to more easily investigate the algorithms, and their expected outcomes.

The test data is generated from Simplex noise, which creates gradient noise suitable for terrain generation, with evenly spaced thresholds to select the classes. An example image of the test data can be seen in Figure 2.1, as well a smaller example in Figure 2.2, showing a numerical representation of a data set. Simplex noise is commonly used to generate terrain, as it produces the same quality noise as Perlin noise, another well-established noise algorithm, but with better complexity [9].

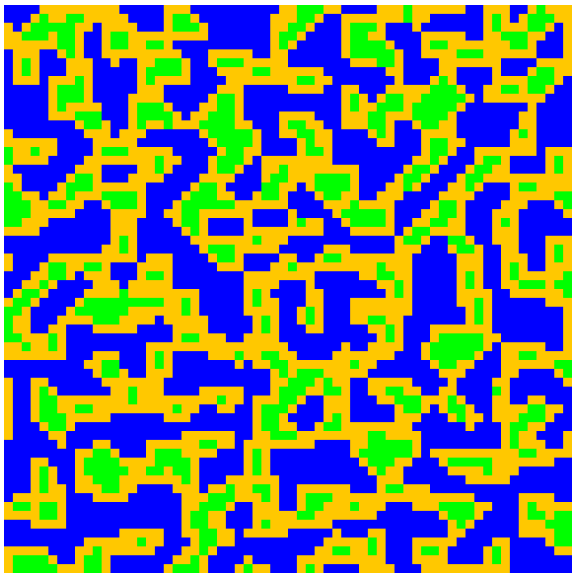

Figure 2.1: An example data set of a $64 * 64$ pixel image generated from Simplex noise.

$$\begin{bmatrix} c_1 & c_1 & c_1 & c_1 & c_3 \\ c_1 & c_2 & c_3 & c_2 & c_3 \\ c_1 & c_1 & c_3 & c_3 & c_3 \end{bmatrix}$$

Figure 2.2: A numerical representation of a smaller data set.

2.2 The company's optimization algorithm

The optimization algorithm the company uses today is like a naive image filtering algorithm, with the key difference that the pixels in the data set, or image, is not composed of colors. Instead each pixel has a class, representing what kind of geography or object exist at that point [10].

The optimization algorithm takes two inputs; the kernel, and the image to be filtered. The kernel is a square matrix with width and height of N, where N may be any positive odd integer smaller than or equal to the image dimensions. The algorithm calculates new values for each pixel in the image, which is done by first overlaying the kernel centered on the current pixel. Each pixel within the bounds of the kernel matrix is multiplied by the corresponding weight in the kernel, and then the mean of all weighted pixels is saved to an output image at the current pixel's location. Not all pixels are valid for this process, since the pixels close to the borders of the input image cannot have the kernel centered over them, while still having valid values for the entire kernel. Thus, each output dimension is smaller than the input, $M - N + 1$ to be exact, where M is the input dimension and N is the kernel size. The output

of the smoothing algorithm is highly dependent on the kernel weights, which should therefore be chosen with patterns and relative scales such that a desirable result is achieved.

As each class is represented by an arbitrary integer, the naive smoothing algorithm would not create any sensible results. Rather the algorithm should choose the category with the greatest presence amongst the pixels in the kernel area.

Because the entire kernel is applied for all pixels in the image, the complexity of the algorithm is $O(N^2)$ per pixel.

In Figure 2.3 an example kernel is shown. Here $N = 3$ with a relatively heavy weight in the center cell. This is a small and very simple smoothing kernel. Processing is done by aligning the center cell of the kernel on each pixel, and then considering the weighted average of all pixels encapsulated by the kernel. In the company's optimization algorithm, the total accumulated weight for each class is kept in a histogram instead, with the dominant class being chosen as the result.

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 5 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Figure 2.3: An example of a $3 * 3$ kernel.

The result of processing the example data set in Matrix 2.2 with the kernel Matrix in Figure 2.3 is presented in Figure 2.4. As mentioned, the algorithm starts by finding the first valid pixel, in this case it is the pixel located at $(2, 2)$ in the data set. The new class for this index is then calculated as follows, where values from the kernel is annotated with a k: See Equation 2.1, 2.2, 2.3.

$$1^k_{(1,1)} + 1^k_{(1,2)} + 1^k_{(1,3)} + 1^k_{(2,1)} + 1^k_{(3,1)} + 1^k_{(3,2)} \rightarrow c_1 = 6 \tag{2.1}$$

$$5^k_{(2,2)} \rightarrow c_2 = 5 \tag{2.2}$$

$$1^k_{(2,3)} + 1^k_{(3,3)} \rightarrow c_3 = 2 \tag{2.3}$$

The new processed class will then be the class that has the highest value between the three classes c_1, c_2 and c_3. Here c_1 has the largest value, and thus the pixel at $(2, 2)$ of the output is chosen to be 1. If multiple classes have the same value, and the value is the largest among all classes, the new class can be either one of them.

The kernel is applied for every valid pixel within the data set. The output of this example is displayed in Matrix 2.4. Note the pixels which originally had a class of 2, as they have changed. Figure 2.5 displays the outcome when using the kernel in 2.3 on the simplex noise image shown in Figure 2.1.

$$\begin{bmatrix} x & x & x & x & x \\ x & c_1 & c_3 & c_3 & x \\ x & x & x & x & x \end{bmatrix}$$

Figure 2.4: An example data set processed by the companies algorithm with $3 * 3$ kernel. The symbol x denotes values in the output that is not valid and should be discarded.

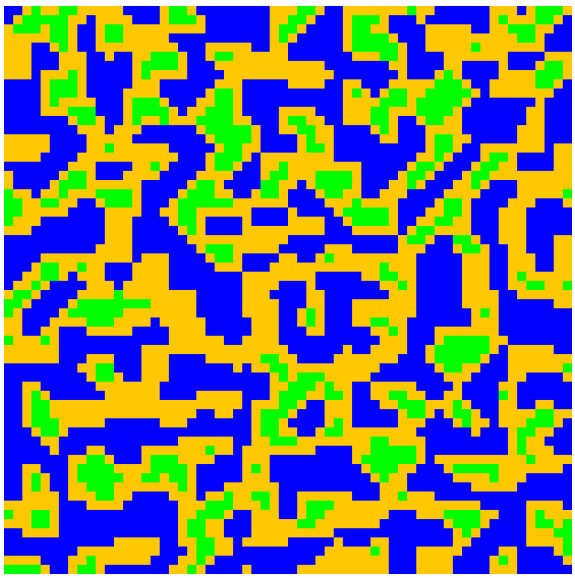

Figure 2.5: The data set shown in Firgue 2.1 processed by the example kernel in Matrix 2.3.

2.3 Parallelism

According to Moore's Law, **ref** the computing power of new microchips, such as CPUs, grows exponentially with time. This has been true since first introduced in 1965, with some minor reinterpretation needed along the years. First it was defined as the number of transistors available, but has since been modified to the amount of computing power per price unit. This has allowed Moore's law to stay relevant, even as the reducing of transistors slowed down [11].

Instead, modern microchips are becoming more capable of performing computations in parallel, whether it be by increasing the number of computing cores, or providing

operations that performs multiple computations at the same time. The latter is referred to as Single Instruction, Multiple Data (SIMD).

To utilize the parallel processing power available, both the algorithm and the data set can be considered. The algorithm can be split into independent parts, which can be executed on multiple processors in parallel, or multiple parts of the data set can be processed independently in parallel. In the former case, the algorithm has a high level of task parallelism, and in the latter, it has a high level of data parallelism. This thesis focuses on an application of data parallelism.

2.3.1 CPU vs GPU

Both the CPU and GPU are high performance microprocessors but with key design differences, which gives each advantage over the other for different cases. Thus, depending on the work to be done, choosing one over the other can greatly impact the processing throughput.

A modern CPU typically has at least two cores, where each core can execute an instruction in parallel with the other cores. Each core is designed to be powerful enough to execute a large and complex task. This design choice implies that each core has large memory resources available, and it has advanced instructions to accommodate different usage scenarios. In addition to this, each core can execute many instructions per second [12].

The GPU is designed for computing graphics, which includes matrix and vector math computations. These computations usually have a high level of data parallelism, which is what the GPU exploits. Thus, the GPU has thousands of cores, but each core has much less processing power in comparison to the cores that one might find on the CPU. The cores of a GPU also have much more limited memory resources available to them.

Instead of each core working independently on a complete task, the cores of a GPU are designed to cooperate on a task to come to a result. The cores are divided into compute units to achieve this cooperation, where each compute unit allows its cores to share more resources than between compute units. Figure 2.6 visualizes how a CPU could differ from a GPU.

Another important design aspect of the GPU is that each core is most efficient when working on small memory footprints. This allows the caching mechanisms in the GPU to reduce the latency introduced by memory access. On the contrary, the CPU should allow an algorithm to access different parts of the memory without introducing large latency. This requires larger and more complex caching mechanisms, which is not feasible to implement on a GPU without increasing the production cost.

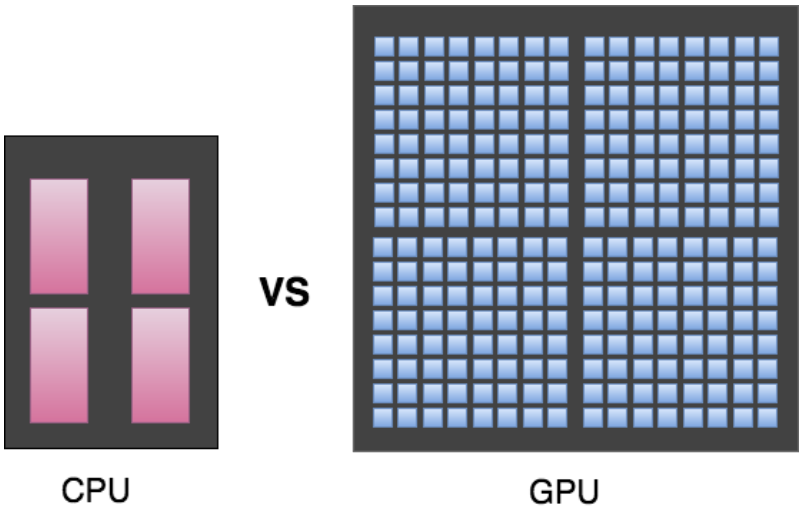

Figure 2.6: To the left, a CPU with multiple cores. To the right, a GPU with many more.

2.4 OpenCL

Open Computing Language (OpenCL) is a open source framework that allows developers to perform general purpose computing on devices that supports it. The support among modern GPUs is very good, but it is also available on CPUs and special made Field Programmable Gate Arrays (FPGA). OpenCL provides a common API through which developers can run code written in OpenCL C, a programming language based on the C99 version of the C programming language. [13]

In OpenCL, each core executes a work item, which in turn belongs to different work groups. The work groups execute its work items on a compute unit. To distinguish the work items from each other they have two identifiers. The first is the global identifier and is unique among all work items. The second is the local identifier, which allows one to uniquely identify the work item in its work group [13].

2.4.1 JogAmp

The OpenCL framework is accessed through C headers. These headers specify how a program should interact with the devices on which the OpenCL C code should execute. However, the company's software solution is written mainly in Java, and the optimization algorithm is implemented in Java. There exists different solution for Java to C interfacing, one such being the Java Native Interface (JNI) framework, which allows a Java program to call specially crafted C functions. Thus, all OpenCL headers can be accessed by the company's Java program.

JogAmp is a library that implements the Java to C interfacing while also applying an abstraction layer which presents the OpenCL framework in an object-oriented way.[14]

2.4.2 Kernels

The main entry point of OpenCL C code is a kernel function, which has some limitations imposed on it relative to a regular function. These limitations include the maximum number of arguments possible and the maximum size of these arguments, and are set in place to allow the framework to efficiently invoke and execute the kernel function. Only functions marked as kernels can be invoked by the OpenCL framework. An OpenCL kernel is not to be confused with a kernel matrix used by both the smoothing algorithm and the companies' algorithm.

A kernel can be applied to many inputs in parallel, where each planned application becomes a work item. The pool of work items is divided into work groups, and each work group will be executed by a computing unit (CU). The CU consists of several executing units, and each work item in the work group will ultimately be executed by one of these executing units[13].

2.4.3 Memory access

The OpenCL specification requires the device to present its memory and arithmetic resources in a specific way in order be compliant. These requirements define a key concept that the developer must understand to fully utilize the resources available, as different components of the device may experience starvation otherwise, Figure 2.7 displays a device and its different memory banks.

In some applications, different memory access patterns can have a large impact on performance. The device presents different memory banks with different capacities and bandwidths. To reach high performance it is then preferable to maximize memory utilization, which can be achieved by proper propagation of data between the memory banks. If the application is memory intensive, ensuring high utilization avoids starvation of the device.

The OpenCL memory architecture contains four different kinds of memory spaces that can be accessed, where each usually has a direct correlation to a memory bank on the device. The first one is called global memory, it is the memory which is the slowest but has the largest capacity. This memory is primary used to transfer data between host and device and is accessible by all work items[15].

The second memory space in the architecture is the constant memory space also called the cached memory and is accessible by workers in the same block. The data that is transferred to this memory space must be constant during the whole

process. The third memory in the architecture is local memory, this memory is only accessible by workers in the same working group. Both the constant memory and local memory are much faster than the global memory.

The last type is called private memory and is the smallest and fastest of them all. The private memory is a region of memory that is private for every worker and is not visible by other workers. Variables placed in the private memory will be assigned to available registers.

Figure 2.7 shows the memory architecture of OpenCL.

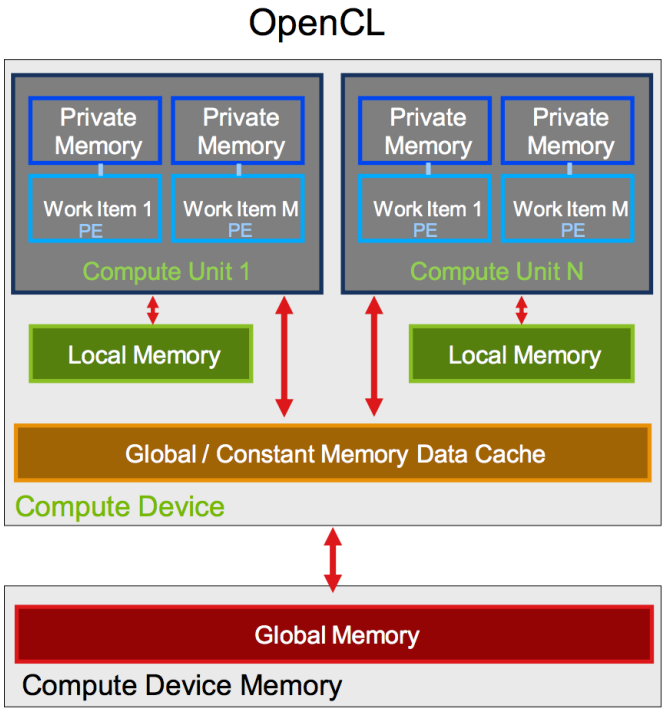

Figure 2.7: OpenCL memory architecture.
Source: `http://www.nvidia.com/content/GTC/documents/1068_GTC09.pdf`

RGBA Packing

OpenCL defines functions for reading from and writing to textures. Textures are single- or multidimensional images where each pixel has a combination of six channels; red, green, blue, alpha, intensity, and luminance. Only some combinations can be combined for usages.

Some devices, such as GPUs, have memory architectures which are especially designed to access textures, rather than simple buffers. This allows the device to more efficiently cache pixels that are close to each other. These devices also perform the pixel access as a single step, essentially reading or writing multiple values at once[6].

In regular image processing, RGBA packing can greatly improve performance as they are designed to read and write pixels composed of multiple channels and can take advantage of the SIMD features of the device. However, one can use the increased memory performance even if the pixel should not be considered. This can be done by packing a value into each channel, effectively converting a n-dimensional matrix into an n-dimensional image. The OpenCL C code can then read the pixel containing that channel, and manually extract the value from the channel again. To reduce the amount of read and writes needed the OpenCL C code can be modified to process all the channels in the pixel, on after the other [7].

As an example, consider a $M * N$ matrix that we will pack into an equivalent texture. We arbitrarily choose the RA (red and alpha) channel configuration, which allows us to store two values in each pixel. Thus, we take two adjacent values, and store the first into the red channel and the second into the alpha channel. Of course, there are many ways to define adjacency in this case, and any can be valid. In this example we will choose two horizontally adjacent values, and thus the equivalent texture will have the dimensions $(M/2) * N$. Padding is needed if M is not divisible by the number of channels. A visualization of this example can be seen in Figure 2.8.

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \end{bmatrix} \qquad \begin{bmatrix} ab & cd \\ ef & gh \\ ij & kl \end{bmatrix}$$

Figure 2.8: To the left, the original $M * N$ matrix, and to the right the equivalent texture. Here $M = 4$, $N = 3$ with the RA channel configuration.

Local caching

Some memory access patterns reuse the same data multiple times for different outputs. The naive smoothing algorithm and the company's optimization algorithm have these access patterns, as one pixel influences the output of all other pixels in an area around in it. This can reduce the total throughput when implemented in OpenCL if the data is read from global memory, as it is the slowest type of memory available in OpenCL. The device may be able to mitigate this if the access pattern allows all, or most all, read data to fit in the device's caches. If the exact device and algorithm parameters is known, one could rely on these caching mechanics to increase the throughput. If this is not the case, a better approach would be to consider the access pattern when writing the OpenCL C code.

By constructing work groups in which the work items access the same data, the memory access can be moved from global memory to local memory. The host machine is not able to directly write to the device's multiple local memory regions, and

thus the work items must first ensure that the data they will use is copied from the global memory to their local memory. All subsequent data access can then be performed through local memory, which can significantly improve the total performance of the algorithm [5].

2.5 Singular Value Decomposition (SVD)

Singular Value Decomposition is a method of factoring matrices into singular values and singular vectors. Two types of vectors are produced, left-singular vectors and right-singular vectors. The original matrix can be reconstructed in three steps; convolute each left-singular vector with its corresponding right-singular vector, scale the resulting matrices with the corresponding singular value, and finally sum the matrices.

The number of singular values and vector pairs needed to fully represent the original matrix depends on the distribution in the matrix and its size. This number is referred to as the matrix's rank. A matrix can never have a rank that exceeds N in the case of an $N*N$ matrix. Figure 2.9 displays how matrix is separated into two vectors.

The naive image smoothing algorithm using a $N * N$ kernel has the complexity $O(N^2)$. This can be reduced by the use of the singular values and vector pairs and the convolution property to instead achieve a complexity of $(N * R)$, where R is the rank of the kernel. As the rank cannot exceed N, the average time needed to compute the Filter algorithm could be improved[8].

An example of how a matrix relates to its singular vectors is presented in Figure 2.9.

$$\begin{bmatrix} A \times a & A \times b & A \times c \\ B \times a & B \times b & B \times c \\ C \times a & C \times b & C \times c \end{bmatrix} = \begin{bmatrix} A \\ B \\ C \end{bmatrix} \times \begin{bmatrix} a & b & c \end{bmatrix} \tag{2.4}$$

Figure 2.9: To the left, a matrix kernel that is constructed from a pair of singular vectors.

2.5.1 Rayleigh quotient iteration

SVD is a generalization of eigendecomposition for certain types of matrices. Thus, the singular values are related to the eigenvalues and the singular vectors are related to the eigenvectors. The result of an eigendecomposition can therefore´ be used in much the same manner as the result from a SVD. One method of calculating the

eigendecomposition of a matrix is the Rayleigh quotient iteration, which is an iterative process that achieves increasingly accurate eigenvalues and eigenvectors. The process is given a matrix and an initial guess of any eigenvalue and its corresponding eigenvector of the matrix. The eigenvalue and eigenvector is then recalculated several times until the change is smaller than a threshold. The result is of the iteration is the eigenvalue and eigenvector that was closest to the initial guess. It follows that an educated guess can speed up the iteration process.

The process can be done again with a new initial guess to produce another eigenvalue and eigenvector. However, this can yield the same result again, even though there are more eigenvalues and eigenvectors to discover. Instead they found eigenvalue and eigenvector should be terminated from the input matrix using matrix deflation. Here we consider the Wielandt deflation method[16]. The deflation first subtracts the eigenvalue and the eigenvector from the matrix and then removes a column and a row from the matrix. By applying the Rayleigh quotient iteration method on the smaller matrix, one can be sure that it will not find the same eigenvalue and eigenvector again. However, the newly found eigenvalue and eigenvector is not correct for the original input matrix, and must be modified to essentially undo the subtraction and removal done by the deflation. Deflation can be performed once again on the smaller matrix with the newly found eigenvalue and eigenvector to find a third eigenvalue and eigenvector. Iteration and deflation is repeated until the eigenvalue is smaller than a threshold close to zero, or until the matrix is too small to be deflated[16].

2.5.2 One-sided Jacobi algorithm

Another method of computing the SVD of a matrix is the one-sided Jacobi algorithm. As with the Rayleigh method, the Jacobi method is an iterative process that produces increasingly accurate eigenvalues and eigenvectors. However, the Jacobi method is faster and produces more accurate results than many other algorithms, as well as being suitable for parallel computing [17].

The Jacobi method utilizes Jacobi rotations to diagonalize the input matrix, which can then be used to easily determine the singular values and vectors. Unfortunately, the Jacobi method does not work on all types of matrices, which could limit the application of this method in this project.

For proper decomposition using the Jacobi method, the input matrix must be symmetric, i.e. that it is square ($size(A) = N * N$) and that it is equal to its transpose ($A = A^T$). The kernel matrices used by the company today satisfies both criteria, but the optimization algorithm they use does not utilize these properties.

2.6 Complexity Reduction

The original optimization algorithm the company has today is implemented in a naive way. There are several different ways to improve the complexity of the original algorithm. The improvement on the algorithm will be the second phase in this thesis. To achieve a noticeable improvement on the complexity, the previously mentioned concepts Separable and SVD will be applied.

By applying the concept of SVD stated in Section 2.5 the complexity can be improved from $O(N^2)$ to $O(N * R)$ per pixel. However, since we need to consider the rank of a given kernel and what class each pixel, some modification is needed to get the same outcome as the original optimization algorithm. Therefore must C which is the number of different classes within the data set also be considered, and thus the new complexity becomes $O(N * R * C)$.

Even if these two new variables are introduced the complexity could still be improved. This is because the number of ranks R is bounded by the size of the kernel and thus R will never exceed N. However, if the kernel has a high rank and the data set has many classes the rank and class $C * R$ could exceed N and the overall complexity is then as bad, or worse, as the original algorithms complexity.

2.7 Gaussian kernels

Gaussian filtering uses a 2-d convolution operator for blurring and noise on images. The outcome of Gaussian filtering is a weighted average of a pixel and its neighbors, moving towards the most central pixels. The use of Gaussian kernel for smoothing has become extremely popular, this is because the Gaussian algorithm has certain helpful properties. For instance, the Gaussian filter kernel is always separable and has a rank of 1.

There are different ways to implement the Gaussian filter for different use, the easiest and most common way to implement the Gaussian filter is done by convolution. A Gaussian filter implementation using convolution will at best give the complexity $O(n)$ per pixel. This is not good enough for this thesis and thus a more complex implementation will be considered. To reduce the complexity as far as possible known, then there is only one distinct way to do the implementation. Instead a recursive implementation of the Gaussian filter will be implemented which has a complexity of $O(constant)$ per pixel. This is shown in the "Fast Gaussian Filter With Second-order Shift Property Of DCT-5" article, where a state-of-the-art implementation is described [2]. The proposed implementation in the article is for one-dimensional Gaussian filter while the one needed in this thesis is a 2-dimensional Gaussian filter. But because of the Gaussian separability, multiple-dimensional Gaussian kernels can be decomposed into products of multiple one dimensional Gaussian kernels.

The reason the one-dimensional Gaussian filter implementation is so successful is due to the Second-order shift property of Discrete-Cosine-Transform-5 (DCT-5) also described in the article. The Second-order shift property of DCT-5 enables to recursively calculate a certain DCT coefficient, which is a key aspect in order to have constant-time complexity $O(1)$. Another contributing factor for choosing this implementation is due too minimal arithmetic operating needed, where it also has an overall advantage over similar implementations. The total number of operations per element needed for 1-D filtering is $4K+2$ for multiplication and division, and $4K+1$ for addition and subtraction, where K is the number of DCT coefficient needed. In the article, they conducted tests which showed that when the standard deviation σ for the approximation is large, K should be equal to 3.

The implementation that will be performed in this thesis will have small differences from the one-dimensional Gaussian filter. Because to get an outcome with minimal differences from the original optimization algorithm we need to consider every class in the data set. To achieve this a trade of is needed and the complexity will increase from $O(1)$ to $O(C)$ where C is bounded by the number of different classes in a given data set.

To be able to fully understand the algorithm one need to read article. For this thesis, it is enough to know how to use the algorithm correctly. Two variables need to be defined, the first is $/sigma$ which is a scale parameter i.e. the standard deviation for the approximations and R which is the truncation location.

The Gaussian kernel $g_u(u = -R, ..., +R$ can then be defined as Equation 2.5.

$$g_u = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{u^2}{2\sigma^2}} = \sum_{k=0}^{R} G_k \cos(\phi k u) \tag{2.5}$$

G_k is the number of DCT coefficient of g_u and like mentioned before set to $k = 3$. ϕ is defined as $\phi = \frac{2\pi}{2R+1}$.

In order to calculate G_k a condition is needed shown in Equation 2.6.

$$G_k \simeq \frac{c_k}{2R+1} e^{-\frac{1}{2}\sigma^2\phi^2k^2}, c_k = \begin{cases} 1, & \text{if } k = 0 \\ 2, & \text{otherwise} \end{cases} \tag{2.6}$$

After doing the assumption that g_u can be approximated by low frequency components, the truncation can on frequency components K that is higher than $K < k$ which means G_k is 0. With approximation in mind an approximate kernel is derived $g_u^{\sim}$. The next step is a convolution with an arbitrary sequence $f_x(x = 0, 1..., N-1)$.

$$(f \cdot g_u)_x \simeq (f \cdot g_u^{\sim})_x = \sum_{u=-R}^{R} f_x + u g_u^{\sim} = \sum_{k=0}^{K} G_k F_k^{(x)} \tag{2.7}$$

In the Equation 2.7 $F_k^{(x)}$ is the know the k-DCT coefficient of the input sequence at the given location x. This is obtained by the same way as G_k is obtained.

$$F_k^{(x)} = \sum_{u=-R}^{R} f_{x+u} \cos(\phi k u) \tag{2.8}$$

G_k is already known to be pre-commutable before the filtering process it is crucial that $F_k^{(}x)$ is obtained at an R-independent and a lower cost than G_k.

The following steps are the key for the low complexity and the minimal amount of arithmetic operation. This is because $F_k^{(x)}$ is derived recursively by the second-order shift property of DCT-5. There is three different Equations 2.9, 2.10 and 2.11 that needs to be considered. C_u in the following equations $C_u = \cos(\phi k u)$.

$$F_k^{(x-1)} = \sum_{u=-R}^{R} f_{x+u} C_{u-1} + f_{x+R+1} C_R - f_{x-R} C_{-R-1} \tag{2.9}$$

$$F_k^{(x+1)} = \sum_{u=-R}^{R} f_{x+u} C_{u+1} - f_{x+R} C_{R+1} + f_{x-R-1} C_{-R} \tag{2.10}$$

The following third equation 2.11 is the reason why F_k^x can be recursively computed at an R-independent cost. The third equation is a summation of the two Equations 2.9 and 2.10. Since $C_R = C_{-R} = C_{R+} = C_{-R-1}$, then $F_k^{(x+1)} + F_k^{(x-1)}$ can be rewritten as follow.

$$F_k^{(x+1)} = 2C_1 F_k^{(x)} + F_k^{(x-1)} + C_R \delta^x \tag{2.11}$$

$\delta^x = f_{x+R+1} - f_{x+R} - f_{x-R} + f_{x-R-1}$. All the coefficient is pre-computed and can be all inserted into look-up tables. In order to do the implementation correctly the first $F_k^{(0)}$ and the second $F_k^{(1)}$ DCT-coefficient needs to be calculated explicitly from Equation 2.8 with $k = 0$.

2.7.1 Discrete Gaussian kernels

Given the standard deviation, σ, and a truncation location R, discrete kernel can be computed proposes $R = 3\sigma$ [2]. Without the truncation location, an infinitely large kernel would be needed. With the proposed truncation location, the equations for

converting between σ and the discrete kernel size, N, is presented in Equation 2.12 and Equation 2.13.

$$N = R * 2 + 1 = 3\sigma * 2 + 1 \tag{2.12}$$

$$\sigma = \frac{N - 1}{3 * 2} \tag{2.13}$$

The discrete kernel matrix is then populated per Equation 2.14 with $\sigma_x = \sigma_y = \sigma$. A is the amplitude of the Gaussian distribution, which in this thesis is defined such that the volume of the Gaussian distribution is normalized, i.e. equal to one. The volume equation is defined in Equation 2.15, and the normalized amplitude is defined in Equation 2.16

$$f(x, y) = A * \exp\left(-\left(\frac{x^2}{2\sigma_x^2} + \frac{y^2}{2\sigma_y^2}\right)\right) \tag{2.14}$$

$$V = 2\pi A\sigma_x\sigma_y \tag{2.15}$$

$$A = frac12\pi\sigma_x\sigma_y \tag{2.16}$$

3

Method

This chapter will describe how the implementation was done and the different steps towards the results. This chapter also contains an overall explanation of the translation between the different programming languages used.

Since this thesis contains three different phases, these phases, where each phase improves upon the run time and complexity of the optimization algorithm. The theoretical incremental increase in performance between each phase was considered to give the project a clear structure.

Every phase contains different steps that needed to be fulfilled before the phase could be fully completed and the next phase could start. Overall there are three different steps within a phase, the first being research step. The research step involved reading previous work and how to handle different problems that could occur. This step also included the planning for the current phase. After the ideas and topics for the current phase had been thoroughly researched, the implementation could begin. First a proof of concept implementation was done in MATLAB. The MATLAB code was then translated into OpenCL C code. The final piece of the implementation was the Java to OpenCL interface, which was written using JogAmp. The third and last step of the phase was evaluation, which aimed to examine the run time impact of the input parameters of the algorithm. In addition to this, the differences in output between the original algorithm and the new algorithms was measured. The overall description for every phase is listed in List below.

- Phase 1 : One-to-one replica from CPU to GPU
  - Implementing considering RGBA Packing
  - Implementing considering Local cashing
  - Implementing considering Arrays
- Phase 2 : Reduce the complexity of the algorithm
  - Implementing considering Separability Jacobi
- Phase 3 : Implement a state-of-the-art algorithm

– Implementing considering Gaussian filter kernel

3.1 MATLAB

MATLAB is an efficient programming language for mathematical computations. It is developed by Mathworks and its main intention is to do mathematical operation. MATLAB has a great advantage over other languages like Java, C, C# etc. when it comes to matrix computations. Due to MATLAB's predefined matrix operations it was easy to simulate the given data set and the different algorithms. We simulated the input map-data by defining matrix whose elements are pixels.

3.2 Hardware

The CPU we used to conduct the tests on was an Intel i5 4590 clocked at 3.3GHz, and the GPU was an Nvidia 1060 6GB which is an off-the-shelf, low- to mid-end GPU.

3.3 GPU implementation

OpenCL version 1.1 was used to execute the algorithm on the GPU, and during development on the CPU. JogAmp was used to interface the Java code with the OpenCL framework.

The OpenCL C implementation of the algorithms developed during the three phases was a close translation of the MATLAB implementations. However, the languages differ in many ways and thus the MATLAB code was pseudo code rather than a finished solution for use with a GPU. Throughout out the phases, there were recurrent translation steps which had to be considered. These include indexing of arrays, accessing multidimensional arrays, dynamically allocating memory, handling different data types, and extracting the core parts of the algorithms that can be parallelized.

3.3.1 Translating MATLAB to OpenCL C

An offset of one is used when indexing arrays in MATLAB, i.e. MATLAB considers the first item of an array to be in the one position. In OpenCL C, however, it is assumed that the first item is in the zero position.

MATLAB has very good support for multidimensional arrays, but this concept does not exist in OpenCL C. Instead, one could use an array of arrays to simulate a two-dimensional array for example. One could also allocate an array big enough to hold all items in all dimensions, and then calculate an offset into the array which holds the correct item. The latter solution was used in this project, which eliminates some pointer arithmetic operations while also ensuring that the entire multidimensional array exists in a continuous block of memory.

Much like multidimensional arrays, MATLAB has very good support for dynamically allocating memory. However, OpenCL C does not provide a simple way of doing this, and requires the programmer to implement their own dynamic allocator. In this project, the local memory space was used extensively to provide the work-items with a memory block in which a small part was to be used by each individual work-item.

The data types used in MATLAB usually does not require the programmer to consider how these data types interact. Thus, function written in MATLAB works with both integer values and float values, and with differently sized data types. The exact opposite is true in OpenCL C, requiring the programmer to precisely define the data types to be used, and to some extent define how to convert between them. Thus, some functions, when translated, required multiple definitions to accommodate both floating point and integer data types.

Parallelism was not considered when designing the MATLAB implementation as to not introduce more complexity, which may not have translated well into OpenCL C code. Instead, once a working implementation was found, independent parts of the code which could be parallelized was identified. These parts where then translated to OpenCL C code.

3.4 Phase 1

The first step in Phase 1 required a lot of research to establish how the optimization algorithm could be used in different ways without changing its core or functionality. The data set is created as a $N_{dataset} * M_{dataset}$ matrix. The kernel is created as a square matrix $N * N$ where $N_{dataset} >= N$ and $M_{dataset} \geq N$. Additionally, a vector of size N is also created, which contains a weight for each class. The purpose of the weight vector is to increase the priority of certain classes. During the development and testing in this project, all weights were equal to 1. It is only mentioned here as it is a part of the company's original algorithm. This section contains implementation of the MATLAB code with an example data set, kernel and a weights vector see Example 1.

Example 1. Assume that we want a data set that is $5 * 5$ and a kernel that is $3 * 3$, where the center cell should have the priority value 5 and the outer "ring" a value of 1. We also want to create a weight vector. This could then be done in MATLAB as follows: see Listing 3.1

```
dataset = [1 1 2 1 1; 1 1 2 1 1; 2 2 3 2 2; 1 1 2 1 1; 1
    1 2 1 1]
nClasses = 3
kernel = [1 1 1; 1 5 1; 1 1 1]
weights = [1 1 1]
\caption{Phase 1 MATLAB implementation}
```

Listing 3.1: Phase 1 MATLAB implementation, continued

In order to create the outcome i.e. the new processed data set, a new empty $(N_{dataset} - N + 1) * (M_{dataset} - N + 1)$ matrix has to be allocated. When the kernel is applied on a pixel in the data set every class within the kernel needs to be considered to calculate the new class, which requires a vector that keeps track of the accumulated class values. In Listing 3.2, this vector is referred to as a histogram. The new class is stored in the corresponding pixel of the output.

```
kernelSize = size(kernel,1);
% Dimensions of the output dataset
col = size(dataset, 2) - kernelSize + 1;
row = size(dataset, 1) - kernelSize + 1;
output = zeros(col, row);
% Histogram initialised with 0
histogram = zeros(nClasses, 1);
```

Listing 3.2: Phase 1 MATLAB implementation

Now, two loops are used to iterate over all valid pixels in the input image. Since the algorithm needs to consider every class within the kernel range, two more loops are needed. The first kernel loop considers all the rows while the second kernel loop consider all the columns within the kernel. In every iteration, a class from the input image is stored in a variable, referred to as *vin* in the MATLAB code. This *vin*-value will be the index for increasing the value for that class, with the right priority. After the two inner loops are done, the argument of the maxima (argmax) of the histogram vector is chosen as the output class, for the current pixel.

```
for j = 0:row-1
  for i = 0:col-1
    for y = 0:kernelSize-1
      for x = 0:kernelSize-1
        vin = dataset(j+y+1, i+x+1);
            histogram(vin+1) = histogram(vin+1) + weights
               (vin+1)*kernel(y, x);
        end
      end
    [~, argmax] = max(histogram)-1;
    output(j+1, i+1) = argmax;
    histogram = zeros(nClasses, 1);
  end
end
```

Listing 3.3: Phase 1 MATLAB implementation, continued

3.4.1 OpenCL implementation

The algorithm in Listing 3.3 then translated into OpenCL code to be used with our GPU. The initial translation did not consider much optimization regarding memory access or work-group sizes. Instead the code was kept close to the MATLAB code. A major difference is the absence of the two outermost loops, as these are used as work-dimensions for OpenCL instead. Two additional OpenCL versions was also written, implementing the use of RGBA packing and local caching respectively.

The initial version made use of simple one-dimensional arrays, where the input image and kernel was laid out row after row in memory. The output image was also laid out in this manner. The work-groups are constructed such that every compute unit would calculate the pixels of a small square in the output image.

When implementing RGBA packing the input image and kernel was instead represented as the *image2d_t* type. This was required to use image related functions which utilizes samplers and multidimensional caching algorithms. When converting the input image and kernel to the *image2d_t* type the red, green, blue, alpha (RGBA) channel configuration was used, which allowed the OpenCL algorithm to read or write four values at the same time. Thus, four adjacent values in a row could be read or written simultaneously, as the memory layout was consistent with the one used in the initial version. The work-groups was constructed as in the initial version.

Finally, local caching was implemented in the third version. Here each work-item starts of by reading a few values from the input image, kernel and weight vector and then storing these in arrays located in the local memory space. After this step the work-item enters a local memory barrier, which halts the execution of

the work-item to allow for other work-items to execute. When all work-items in the work-group has entered the barrier, execution continuous and all values that the work-group shares will be cached in local memory. Now the work-items performs the optimization algorithm, but unlike the previous versions where each item calculates a single output, here each item calculates a small square. This was done to maximize the amount of data cached.

3.5 Phase 2

When implementing Phase 2, the optimization algorithm considers the output as independent classes, rather than pixels dependent on all neighbors who lies in the range of the kernel. Thus, parts of the algorithm had to be modified to accommodate this.

The algorithms still require the input image, denoted data in Listing 3.4, and the weight vector containing weights for all classes. The kernel is represented as R singular vector pairs, where R is the rank of the kernel, denoted v and h in Listing 3.4. Additionally, the algorithm requires the rank of the kernel and the number of classes present in the input image.

The singular vectors are scaled per the corresponding singular values such that the calculated result is scaled properly. In our implementation, this was done by multiplying each singular vector with the square root of the corresponding singular value.

```
1 function output = phase2(v, h, data, weights, nRanks,
    nClasses)
2 kernelSize = size(v, 2);
```

Listing 3.4: Phase 2 MATLAB implementation

First, the algorithm convolutes the rows of the input with the horizontal singular vectors, as seen in Listing 3.5. However, it is not the classes that are convoluted, but their weights. This also means that the classes must have separate output images, creating the need for a four-dimensional output from the horizontal pass.

```
hPassCols = size(data, 2) - kernelSize + 1;
hPassRows = size(data, 1);
hPass = zeros(hPassRows, hPassCols, nRanks, nClasses);
for y = 0:hPassRows-1
  for x = 0:hPassCols-1
    for r = 0:nRanks-1
      for k = 0:kernelSize-1
        vin = data(y+1, x+k+1);
        hPass(y+1, x+1, r+1, vin+1) = ...
            hPass(y+1, x+1, r+1, vin+1) + ...
            weights(vin+1) * h(r+1, kernelSize-k-1+1);
      end
    end
  end
end
```

Listing 3.5: Phase 2 MATLAB implementation, horizontal pass

Next, the algorithm convolutes the columns of the output from the previous pass with the vertical singular vectors, as seen in Listing 3.6. Unlike the original input image, the values contained in *hPass* is not classes, but the result from the horizontal convolution. The algorithm also performs a summation of all ranks for the current pixel and class, which is then used to determine which class had the largest value.

```
outputCols = hPassCols;
outputRows = hPassRows - kernelSize + 1;
output = zeros(outputRows, outputCols);
for y = 0:outputRows-1
  for x = 0:outputCols-1
    argmax = 0;
    maxval = -realmax;
    for vin = 0:nClasses-1
      val = 0;
      for r = 0:nRanks-1
        for k = 0:kernelSize-1
          val = val + ...
              hPass(y+k+1, x+1, r+1, vin+1) * v(r+1,
                  kernelSize-k-1+1);
        end
      end
      if val > maxval
        argmax = vin;
        maxval = val;
      end
    end
    output(y+1, x+1) = argmax;
  end
end
```

Listing 3.6: Phase 2 MATLAB implementation, vertical pass

During initial testing and development, the singular vectors and values were computed using MATLAB's built in *svd* function, but was later replaced by a Jacobi SVD algorithm. The Jacobi algorithm was also implemented in Java, which was used by our program to compute the SVD as a preprocessing step to the OpenCL implementation of the convolution. Once the SVD for a matrix has been computed, it can be used for all subsequent convolutions for that matrix, which is why we decided to not implement the Jacobi algorithm in OpenCL.

3.5.1 OpenCL implementation

The OpenCL implementation of the MATLAB code for Phase 2 was divided into three steps; a horizontal pass, a vertical pass and lastly an argmax step. These steps were implemented as separate OpenCL kernels, which when chained together results in the desired output. A consistent difference between the MATLAB and OpenCL implementation is that the OpenCL implementation only considers a single rank at a time. Thus, multiple runs with different singular vectors is needed before finally choosing the argmax of each pixel.

The horizontal pass closely resembles the MATLAB code, with the two outermost loops used as work-dimensions for OpenCL. The output from the horizontal pass is a three-dimensional array, as the ranks are no longer considered.

The vertical pass uses all three of these dimensions as work-dimensions, leaving only a single for-loop for the actual convolution.

Much like the horizontal pass, the argmax step uses the X-axis and Y-axis as work-dimensions. Only a single loop over all classes is needed to choose the result for the pixel.

3.6 Phase 3

The implementation for Phase 3 was done with the respect of the Second-order shift property of DCT-5 found in the Fast Gaussian Filter With Second-order Shift Property Of DCT-5 article. The contributors of this article included a pseudo-code section in the article. This pseudo-code was to great help when implementing the 1-D Gaussian filter in MATLAB and later in OpenCL C code.

The pseudo code well as our implementation are both based on the equations explained in Theory 2. We did the same sort of implementation as the previous Phases, we first implement the pseudo-code with changes in MATLAB. Then after conducting MATLAB tests to get the right functionality we translated the MATLAB code into OpenCL C code. There were two major changes that was needed. The first change was to have a 2-dimensional filter while the pseudo code only is for 1-dimensional. The second and hardest change was to consider the different classes and their priority to get the right outcome. The pseudo code from the article is shown for the sake of this thesis. This is so the reader easily can follow what

changes that was done to achieve the desired outcome.

Algorithm 1: Constant-time Gaussian filter

1 Calculating the first and second terms from the article[2]
2 $F_0 = \sum_{u=-R}^{+R} f_u$
3 **for** *k=1 to K* **do**
4 $\quad Z_k^- = \sum_{u=-R}^{+R} \{\gamma_k cos(\phi uk)\} f_u$
5 $\quad Z_k = \sum_{u=-R}^{+R} \{\gamma_k cos(\phi uk)\} f_{u+1}$
6 **end**
7 Convoluting cosine terms slidingly
8 $(f * g_u^{\sim})_0 = \{G_0\}(F_0 + \sum_{k=1}^{K} Z_k$
9 $F_0 = F_0 - f_{-R} + f_{R+1}$
10 **for** *x=1 to N-1* **do**
11 $\quad$ Calculating the output value for location x
12 $\quad (f * g_u^{\sim})_0 = \{G_0\}(F_0 + \sum_{k=1}^{K} Z_k$
13 $\quad$ Updating the DCT Coefficients for the next
14 $\quad F_0 = F_0 - f_{x-R} + f_{x+R+1}$
15 $\quad \delta = f_{x-R-1} - f_{x-R} - f_{x+R} + f_{x+R+1}$
16 $\quad$ **for** *k=1 to K* **do**
17 $\quad\quad \zeta = \{2cos(\phi k)\} Z_k - Z_k^- + \{\gamma^{(k)} cos(\phi kR)\}\delta$
18 $\quad\quad Z_k^- = Z_k, Z_k = \zeta$
19 $\quad$ **end**
20 **end**

3.6.1 OpenCL implementation

The OpenCL implementation of pseudo code for Phase 3 was as Phase 2 divided into three steps; a horizontal pass, a vertical pass and lastly an argmax step. The horizontal pass is the pass with the biggest changes from the pseudo code 1. The first and most global change in the horizontal pass is that we need to consider every class and its priority in the data set. This is done by dividing the data set into c different data sets to consider every class within the original data set. This means that F_0 becomes an array containing the F_0-value for every class, therefore line number 2 needs to be changed. Due to this functionality change every appearance in the pseudo code containing the input data f_u needs translate a value into a class. The row numbers in the pseudo code that are direct effected by this is 2,4, and 5. The rest of the pseudo code encounter the functionality in the three lines and thus it means minor changes was required for the rest of the code. To do the correct functionality for every class an iteration is introduced within the last iteration in the pseudo code. The new iteration is inner-bounded by the number of classes and is outer-bounded by the K.

The vertical pass has the same functionality as the pseudo-code, and is a straight forward implementation from the pseudo-code. This is because the vertical pass

only consider 1 class as each work-item is given its own class to process.

Essentially the horizontal pass divides the input data set into different data set, where each data set corresponds to a class in the original data set. However, the algorithm exploits the fact that each pixel has a unique class to avoid iterate over all classes in the crucial steps in the algorithm. Instead our implementation only needs to iterate over classes when slidingly calculating the DCT-coefficients and when computing the output. This is the reason why our implementation has a complexity of $O(C)$ where C is classes and not $O(1)$.

4

Results

This chapter will present the strategies used when measuring run time and evaluating correctness of the algorithms. The results gathered from these measurements will also be presented. First, the original optimization algorithm results were gathered from measuring in terms of run time, and the output from this algorithm will be considered as the desired output of the algorithms developed during the different phases. Next, the three phases results gathered from measuring will be presented. Finally, a comparison between the phases is presented to show the accuracy between the phases and the original optimization algorithm.

4.1 Testing strategy

The run time measurements in this chapter was done such that the size of their common input parameters was recorded and compared. These parameters are input image size, kernel size, and the number of classes present in the input. In the following results, the input image size represents the size of both sides, such that an input image size of N represents a $N * N$ image. The same assumption is correct for the kernel size. In addition to this, the measurements from Phase 2 also depends on the rank of the kernel, which is bounded by the kernel size.

The run time measurements were run ten times per input parameter configuration and averaged. This was done to reduce the impact of temporary system resource starvation.

The ranges of the input parameters were chosen as to work for the original algorithm and the three phases. On the GPU used to run these tests, the maximum kernel size was limited by Phase 1. As the kernel was to reside in the constant memory space, the maximum kernel size was limited to 127, after considering other data in the same memory space.

The maximum input image size and number of classes were both limited by the available global memory on the GPU, which up to 6GB memory. However, the calculations of these sizes in the program are done with 32-bit signed integers, and can only address 2GB of memory. In addition to this, the data type used by the

classes are 8-bit unsigned integers, which restricts the number of classes to 256.

Phase 2 and Phase 3 requires a three-dimensional image to store the output from the horizontal pass. The image has the dimensions $X * Y * C$, where $X * Y$ is the input image size, C is the number of classes, and the data type of the image is 32-bit floating point numbers, requiring four bytes per floating point. Thus the maximum image size is calculated as $sqrt(2^31/4/256) \approx 1448$. A power of two number was desired, and thus the maximum was chosen to be 1024 instead. The minimum input image size was chosen to be 256.

All results will be presented as two dimensional graphs, where the Y-axis is the run time. Each input parameter will be presented in separate graphs on the X-axis. The other input parameters will be set to either its minimum or maximum, as stated by accompanying captions. The X-axis is presented in base two while the Y-axis is not scaled, which must be kept in mind when observing the results.

The input images were generated by assigning each pixel a randomized class in the range of the number of classes that should be present.

4.2 Original optimization algorithm

The original algorithm, implemented in Java, was executed on the CPU and the run time was measured as a function of its input parameters. These measurements can be seen below.

Figure 4.1 shows the impact of the input image size. For each step, the size is doubled, resulting in an image that is four times larger. This trend can be seen in the run time, as the run time is roughly multiplied by four as well as the size increases.

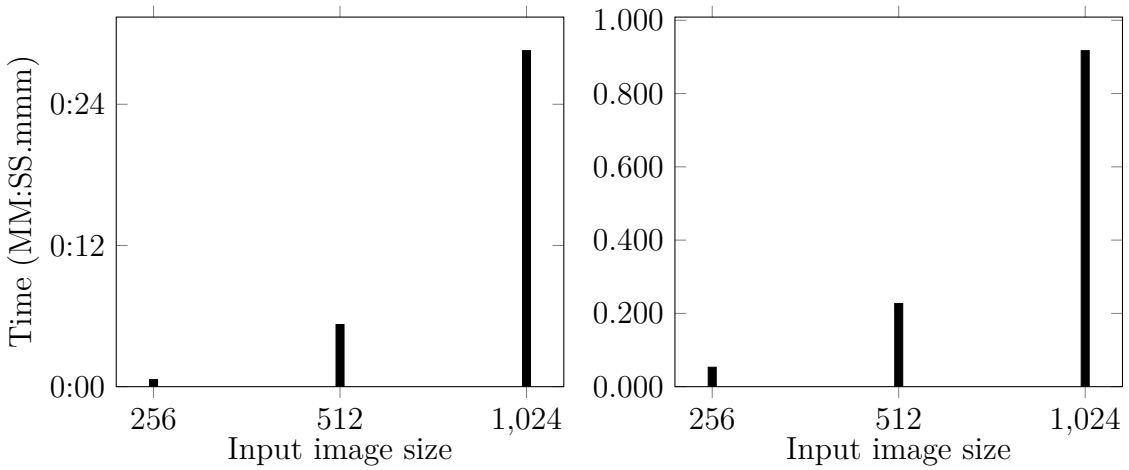

Figure 4.1: Run time as a function of input image size.
Left: Kernel size=127, Number of classes=256
Right: Kernel size=15, Number of classes=16

The impact of kernel size follows the same trend as the input image size, as can be seen in Figure 4.2. This was expected, as the algorithm loops over all items in the kernel.

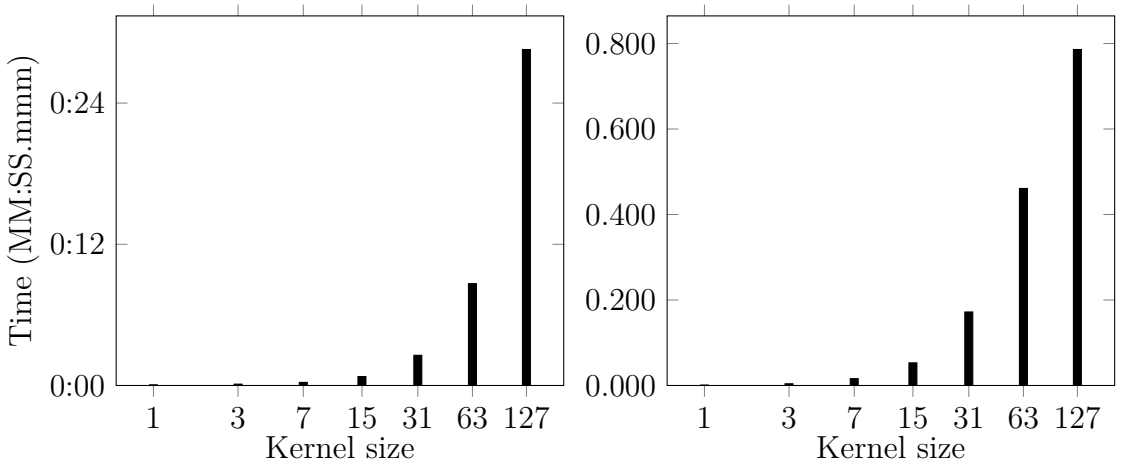

Figure 4.2: Run time as a function of kernel size.
Left: Input image size = 1024, Number of classes = 256
Right: Input image size = 256, Number of classes = 16

The relationship between run time and the number of classes is presented in Figure 4.3. As the algorithm, does not loop over this parameter, the relationship is expected to be constant. However, this is not the case as seen in Figure 4.3, instead a reverse relationship exists. In Section 2.2, the a complexity per pixel of $O(N^2)$, where N is the kernel size, was states. This is strengthened by the results presented in Figure 4.3.

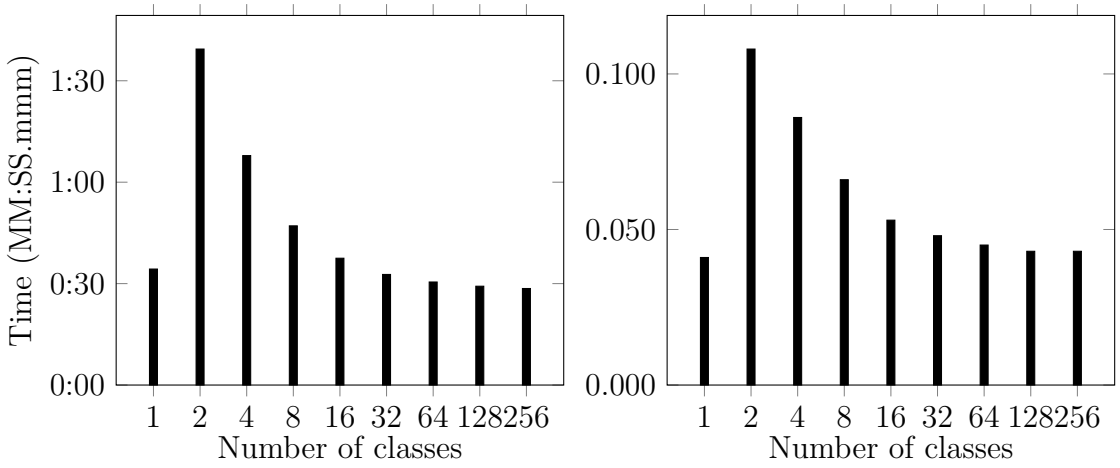

Figure 4.3: Run time as a function of the number of classes in the input image.
Left: Input image size = 1024, Kernel size = 127
Right: Input image size = 256, Kernel size = 15

4.3 Phase 1

The measurements from the three different implementations developed during Phase 2 is presented below. The initial implementation is denoted Array, while the RGBA and Local optimized implementations are denoted by their respective optimization strategy.

As with the original algorithm, the impact of the input image size induces a linear increase in run time. This can be seen in Figure 4.4.

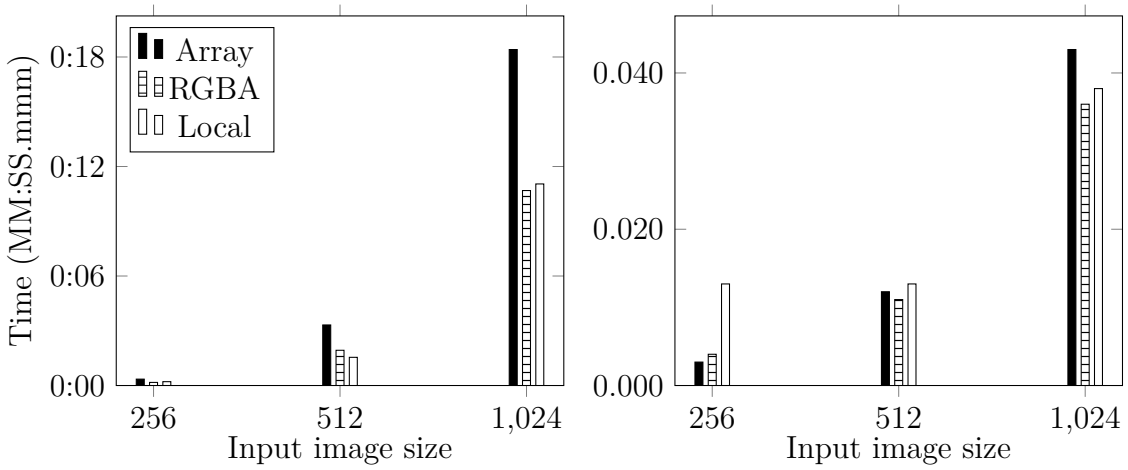

Figure 4.4: Run time as a function of input image size.
Left: Kernel size = 127, Number of classes = 256
Right: Kernel size = 15, Number of classes = 16

The size of the kernel used has a polynomial impact on the run time, as seen in Figure 4.5. In the left figure, where the input image size and number of classes is maximized, the impact is greater than N^2, where N is the kernel size. However, on the right, where the other parameters are minimized, the impact is close to N^2. This differences cannot be seen in Figure 4.4, but can be explained by Figure 4.6, where large number of classes has a much larger impact.

The optimizations introduced in the RGBA and Local implementations can be seen, as the outperform the Array implementation when the other parameters is maximized. However, the Local implementation has an overhead associated with the caching algorithm used, as can be seen when the parameters is minimized.

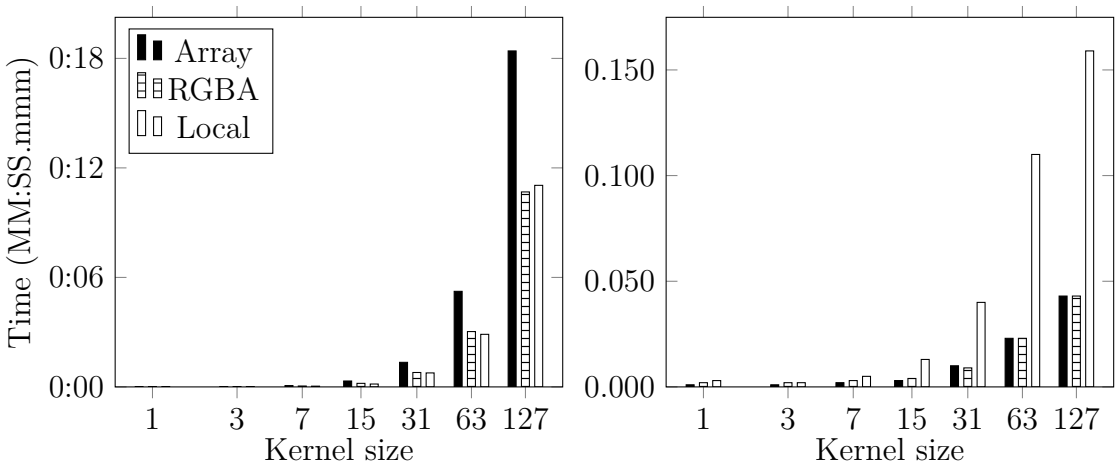

Figure 4.5: Run time as a function of kernel size.
Left: Input image size = 1024, Number of classes = 256
Right: Input image size = 256, Number of classes = 16

The impact of the number of classes in the input image is presented in Figure 4.6. Recalling the impact on the original algorithm, as seen in Figure 4.3, a difference can be seen. For Phase 1, a larger number of classes negatively impacts the run time, but it is only noticeable after 16 classes. This contrasts with the original algorithm, where a larger number reduced the run time, per a continuous relation.

As with the impact of the kernel size, seen in Figure 4.5, the Array implementation is affected to a larger degree by the number of classes. The overhead of the caching algorithm in the Local implementation is once again also noticeable.

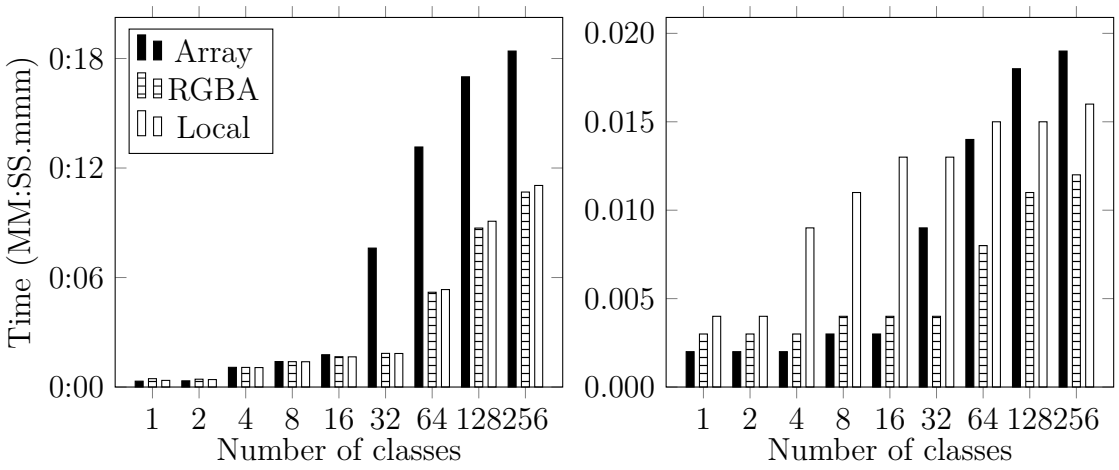

Figure 4.6: Run time as a function of number of classes in the input image.
Left: Input image size = 1024, Kernel size = 127
Right: Input image size = 256, Kernel size = 15

Many relations found in the original algorithm is also present in the OpenCL implementation of Phase 2, with the most prominent deviation being the impact of the number of classes. Disregarding this deviation, the stated $O(N^2)$ complexity of the algorithm holds true.

The overall run time of Phase 1 is reduced when compared to the original algorithm, where the RGBA implementation performed the best in most configurations.

4.4 Phase 2

In addition to the input image size, the kernel size and the number of classes, the algorithm for Phase 2 also depends on the rank of the kernel.

In Figure 4.7, the relationship between the input image size and the run time can be observed. As with the previous phases, this relationship is linear.

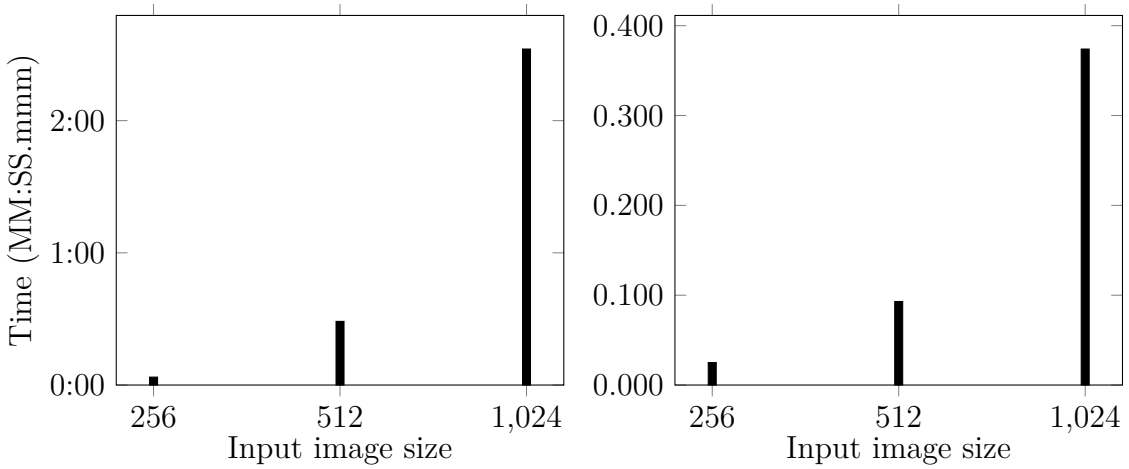

Figure 4.7: Run time as a function of input image size.
Left: Kernel size = 127, Rank of kernel = 127, Number of classes = 256
Right: Kernel size = 15, Rank of kernel = 15, Number of classes = 16

In order to show the impact of the kernel size, all other parameters must be constant. In previous figures, all parameters were independent. However, as stated in Section 2.5, the rank of a kernel is bounded by the kernel size. Thus, we must choose a rank which can be satisfied by all presented kernel sizes. In Figure 4.8, kernels with a rank of one was used.

In Figure 4.8, the relationship is masked by the fact that the algorithm is very fast when processing kernels with a rank of one. However, a linear increase in run time can be seen when the kernel size increases and other parameters were large, when one realizes that the offset of the values are caused by the overhead introduced by OpenCL. When the other parameters were small, the run time becomes noisy instead due to the small workload.

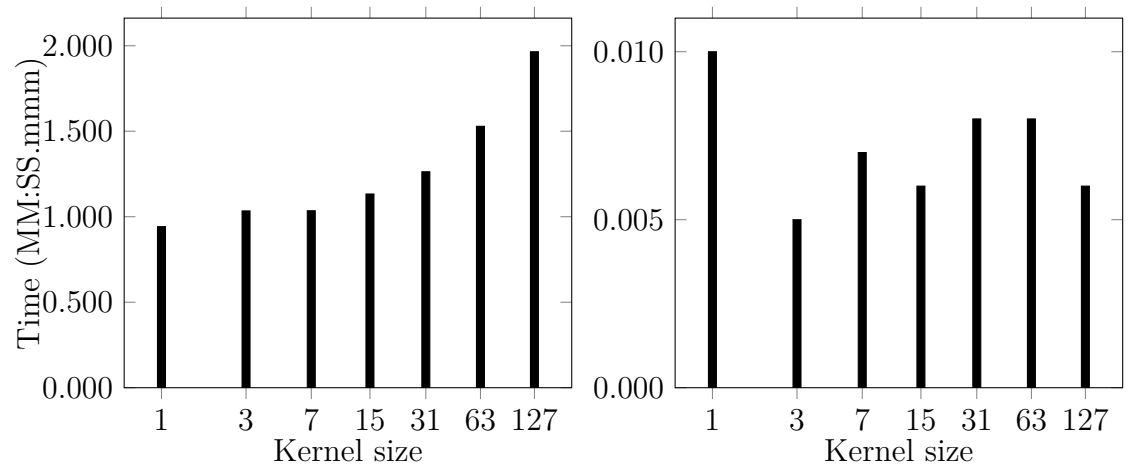

Figure 4.8: Run time as a function of kernel size.
Right: Input image size = 1024, Rank of kernel = 1, Number of classes = 256
Right: Input image size = 256, Rank of kernel = 1, Number of classes = 16

In Figure 4.9, kernels with a size of 127 was chosen to satisfy the relationship between the rank and kernel size, while keeping the kernel size constant.

Unlike the results seen in Figure 4.8, the range of the run times in Figure 4.9 is large. This makes the overhead induces by OpenCL unnoticeable, and a linear relationship between run time and kernel rank is clear.

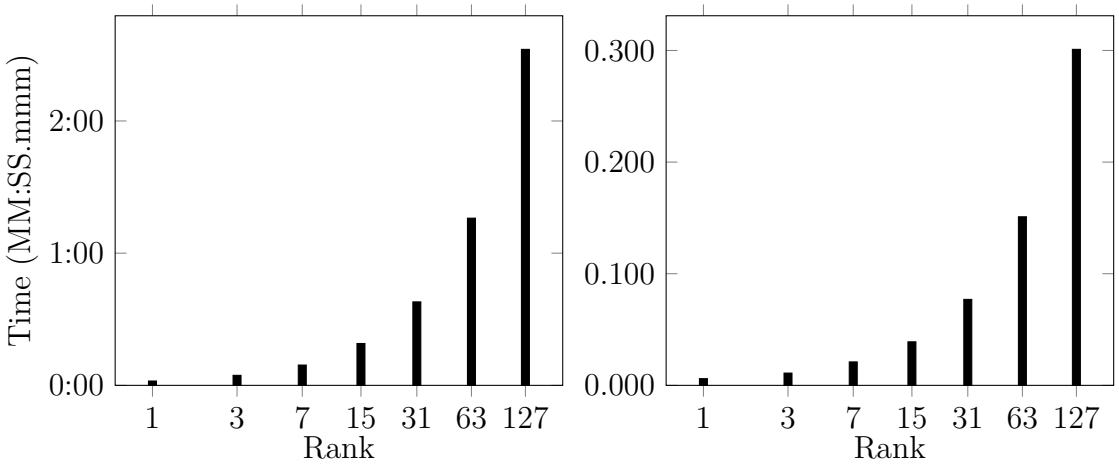

Figure 4.9: Run time as a function of kernel rank.
Left: Input image size = 1024, Kernel size = 127, Number of classes = 256
Right: Input image size = 256, Kernel size = 127, Number of classes = 16

Measurements of the run time as the kernel size and kernel rank increased together is presented in Figure 4.10. This combines the observations of both Figure 4.8 and Figure 4.9, creating a polynomial relationship. This relationship closely resembles the kernel size relationships which can be seen in the original algorithm and Phase 1. However, the longest run time measured in the original algorithm is shorter than the one observed in Figure 4.10. This showcases the increased overhead introduced by the much more complex algorithm.

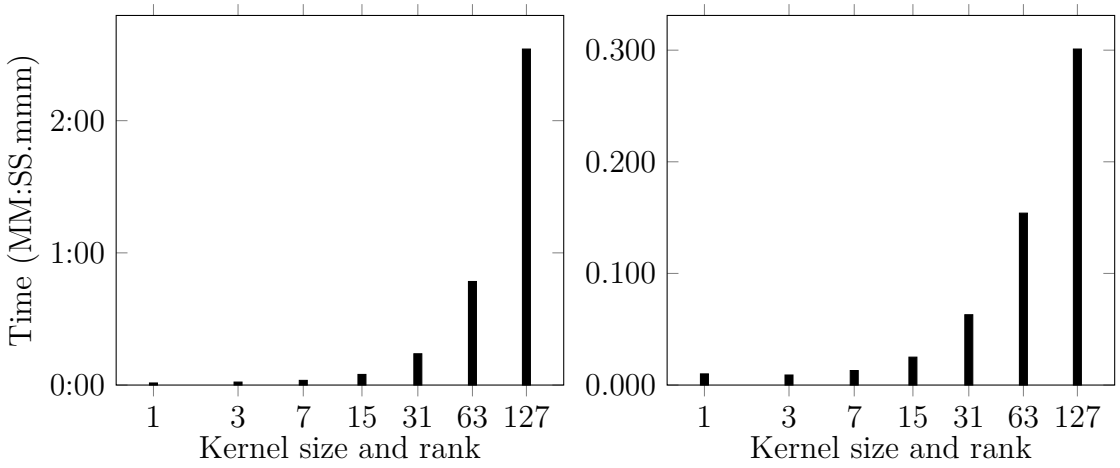

Figure 4.10: Run time as a function of kernel size and rank.
Left: Input image size = 1024, Number of classes = 256
Right: Input image size = 256, Number of classes = 16

In Figure 4.11, the relationship between run time and the number of classes can be observed to be linear.

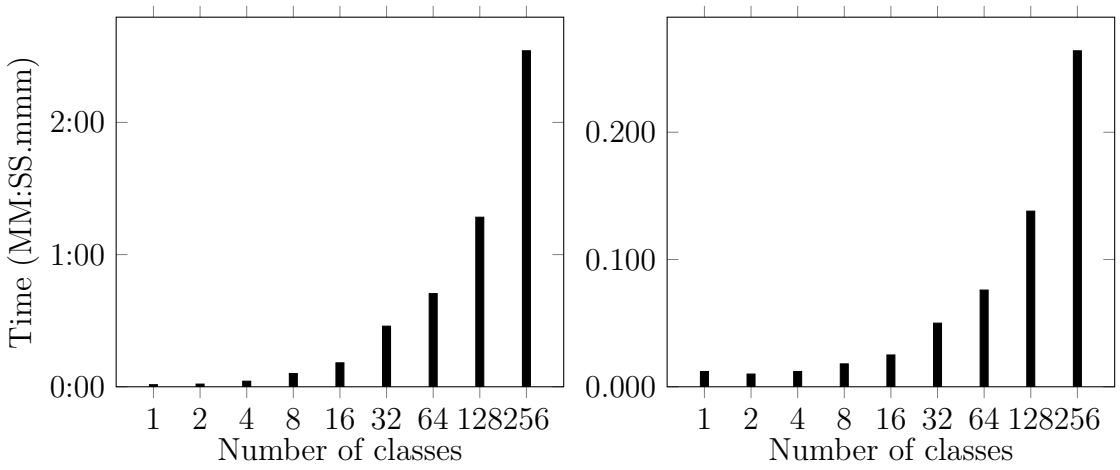

Figure 4.11: Run time as a function of the number of classes in the input image.
Left: Input image size = 1024, Kernel size = 127, Rank of kernel = 127
Left: Input image size = 256, Kernel size = 15, Rank of kernel = 15

The relationships between the input parameters and run time presented above is concise with the complexity $O(N * R * C)$ per pixel, states in Section 2.6, for Phase 2.

4.5 Phase 3

Unlike previous phases, Phase 3 does not need a discrete kernel. Instead, only the standard deviation of the Gaussian kernel used is needed. However, for the reader to make comparisons with previous phases, an equivalent kernel size is calculated as stated in Section 2.7.1. For reference, a kernel size of $N = 127$ is equivalent to a Gaussian distribution with a standard deviation of $\sigma = 21$, and $N = 15$ is equivalent to $\sigma = 2.5$.

The linear relationship between run time and input image size is present in Phase 3 as well, as can be seen in Figure 4.12. The graph to the right in Figure 4.12 does not show this relationship very well, but is a result of the very small work load, in addition to the measurement resolution of a millisecond.

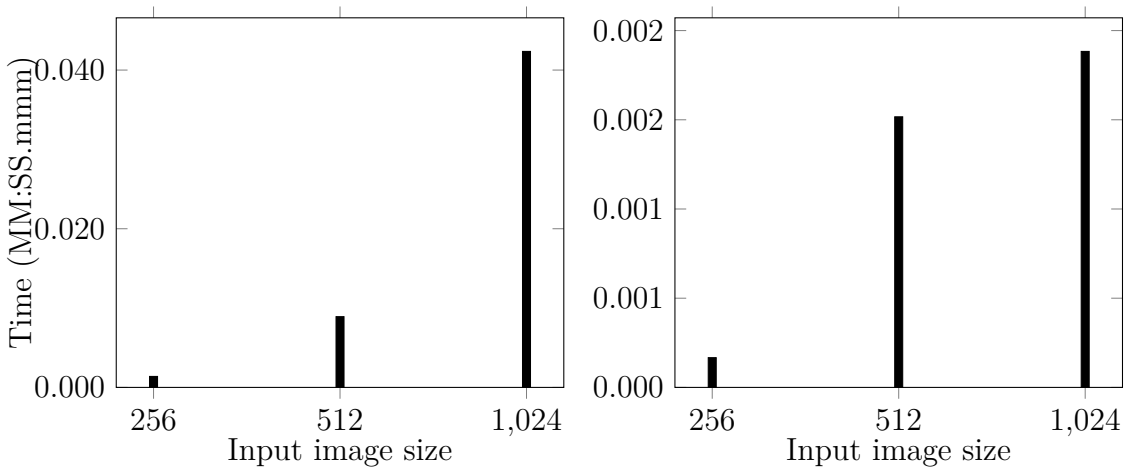

Figure 4.12: Run time as a function of input image size.
Left: Equivalent kernel size = 127, Number of classes = 256
Right: Equivalent kernel size = 15, Number of classes = 16

In Figure 4.13, a major speedup, much greater than the one obtained in the other phases, can be observed. The figure shows the impact of the equivalent kernel size on run time, which seems to be that they are independent. This is in line with the theory stated in Section 2.7, as the algorithm uses the single value of standard deviation to describe the Gaussian distribution.

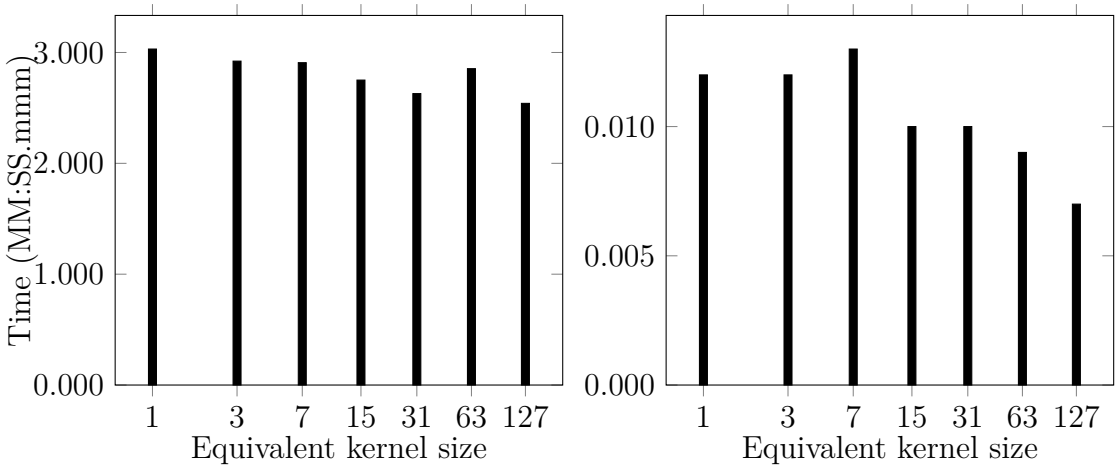

Figure 4.13: Run time as a function of equivalent kernel size.
Left: Input image size = 1024, Number of classes = 256
Right: Input image size = 256, Number of classes = 16

As with Phase 2, the modifications introduced in order to properly consider the use of classes is shown to have an linear impact on run time. This can be seen in Figure 4.14.

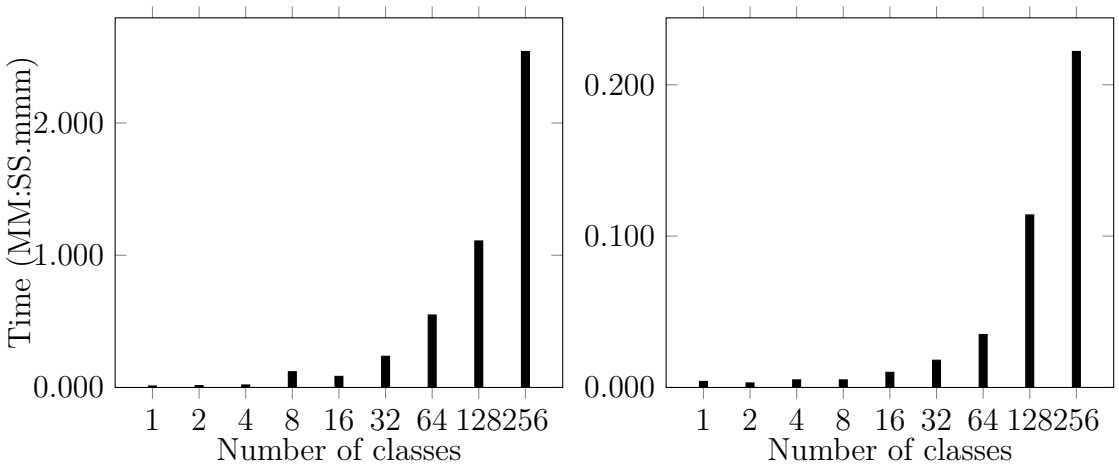

Figure 4.14: Run time as a function of number of classes in the input image.
Left: Input image size = 1024, Equivalent kernel size = 127
Right: Input image size = 256, Equivalent kernel size = 15

4.6 Comparing the phases

In this section, the results from all three phases is presented. The RGBA implementation of Phase 1 was chosen as it was shown that it had the best performance among the other implementations. As discussed, Phase 2 is dependent on the rank of the kernel as well, which for these results was chosen to be 15. The reason for

this is that a greater rank would create too long run times, and a much smaller rank does not represent realistic applications. Thus 15 was decided as it was a plausible average rank, while showing the flaws in the algorithm. For kernel sizes, smaller than 15, a rank equal to the kernel size is chosen instead.

Figure 4.15 shows that Phase 3 out performs the other phases when all parameters, including input image size, are large. Phase 2 is the least performant, but given a lesser rank it would outperform Phase 1, as seen in Figure 4.5 and Figure 4.8.

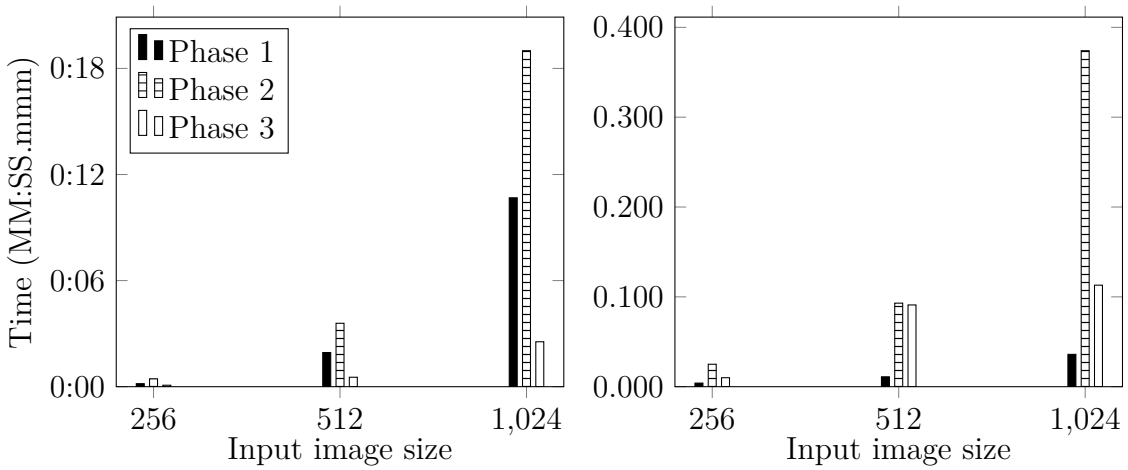

Figure 4.15: Run time as a function of input image size.
Left: Kernel size = 127, Number of classes = 256
Right: Kernel size = 15, Number of classes = 16

The same trend can be seen when the kernel size increases, as shown in Figure 4.16. The overhead present in Phase 3, introduced by the computation of F_0, Z^- and Z, is quickly overwhelmed by the polynomial and linear relationships in Phase 1 and Phase 2 respectively.

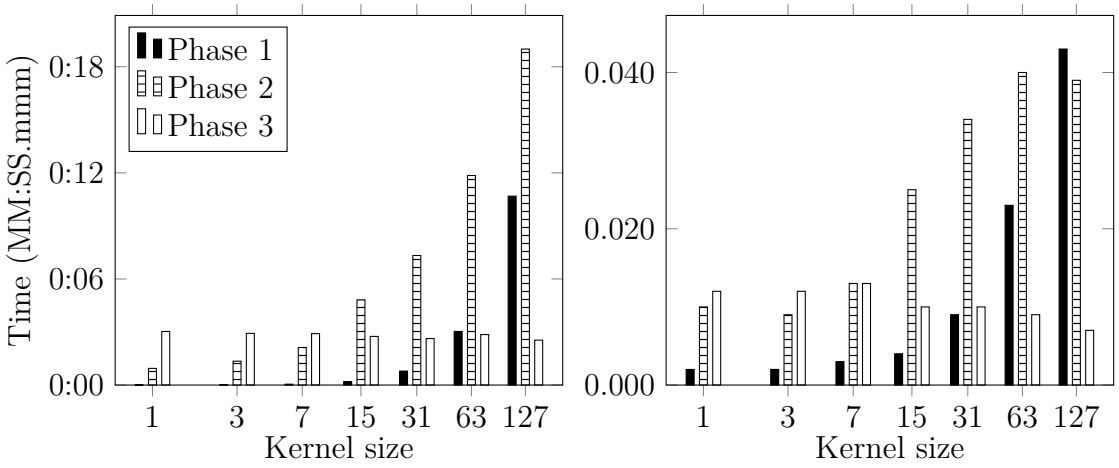

Figure 4.16: Run time as a function of kernel size.
Left: Input image size = 1024, Number of classes = 256
Right: Input image size = 256, Number of classes = 16

Some interesting observations can be done in Figure 4.17, showing the different impacts of the number of classes in the input image. As stated it should not have an impact on Phase 1, but the relationship does not seem to linear or polynomial, as can be seen in Figure 4.6. Both Phase 2 and Phase 3 has a linear relationship, but with the choice of a kernel with a rank equal to 15, it almost seems as if Phase 1 and Phase 2 performs in the same way. However, this is only a coincidence, which is evident if one considers what happens to the different phases when the number of classes is equal to 16, 32, and 256.

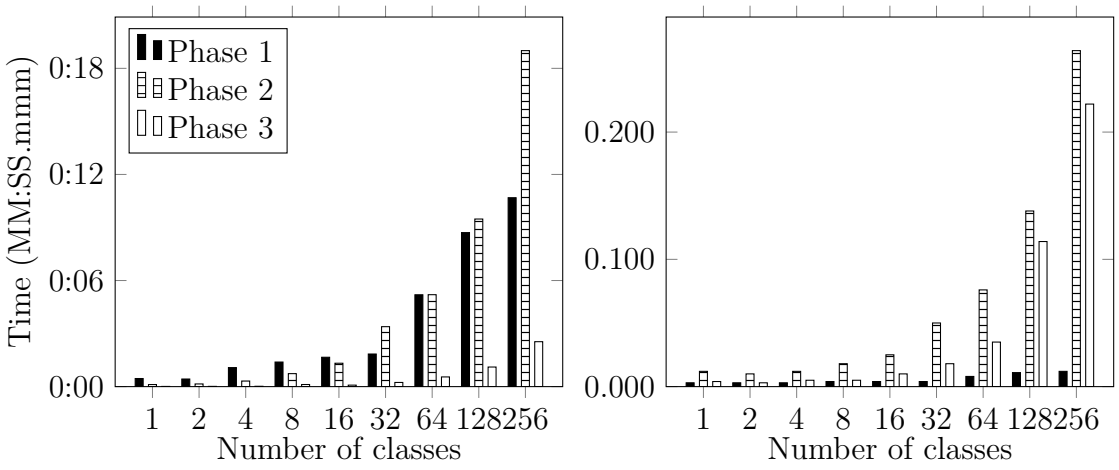

Figure 4.17: Run time as a function of number of classes in the input image.
Left: Input image size = 1024, Kernel size = 127
Right: Input image size = 256, Kernel size = 15

The above figures show that Phase 3 has a very large run time improvement when the kernel grows large. If the input images used contains very few classes, and the

kernel is very small, it is better to choose the original algorithm executed on the GPU. However, this is negligible as the algorithm improvements sought to reduce the impact of kernel size.

4.7 Accuracy evaluation

To evaluate the correctness of the new algorithms, the differences in output from the original algorithm was measured. The results of these measurements are presented in this section. An error is considered as pixels of the output that does not match that of the original algorithm, with the same parameters. Thus, Phase 1 is not considered as it produces the same outputs as the original algorithm. The average error and standard deviation over 100 iterations is presented in the figures below as percentages. The exact amount of errors can be seen in Appendix A.

The graphs shown below follows a similar theme to the ones presented above with a logarithmic X-axis. However, here the Y-axis represents the accuracy rather than run time.

In order to give a fair comparison, all kernels used are based on Gaussian distributions, as this is the only type of kernel compatible with all phases.

Figure 4.18 shows the error of Phase 2 and Phase 3 as the input image size changes. It is notable that Phase 2 produces a more accurate output, when compared to Phase 3. However, both phases' accuracy decreases as the amount of input pixels increases. The standard deviation decreases as well.

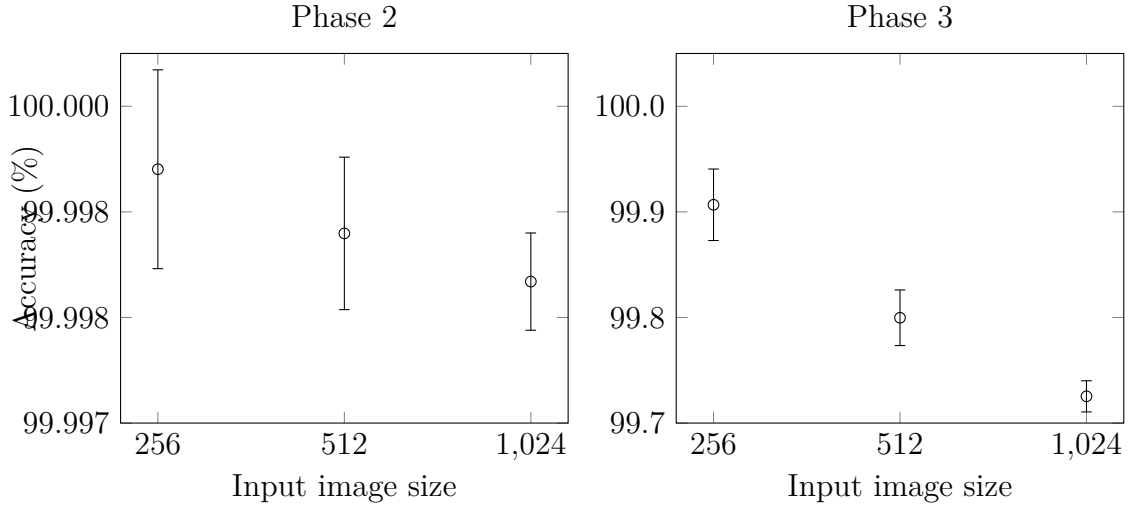

Figure 4.18: Output accuracy as a function of input image size.
Kernel size = 127, Number of classes = 4

Kernel size had a negative impact on the accuracy, as can be seen in Figure 4.19.

However, the relationship does not seem to linear or polynomial, but more complex.

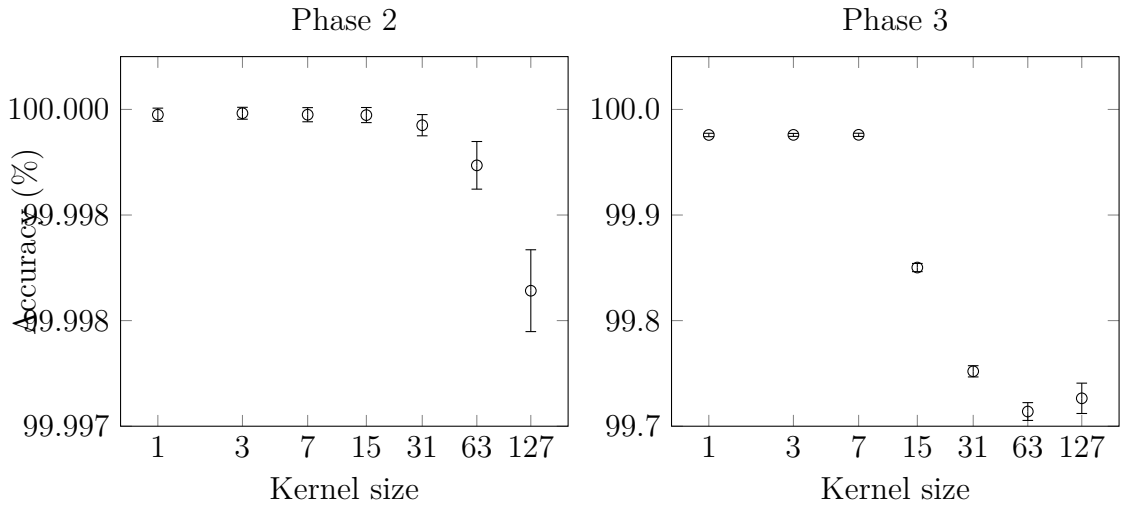

Figure 4.19: Output accuracy as a function of kernel size.
Input image size = 1024, Number of classes = 4

Unlike the accuracy presented above, the number of classes did not have the same impact on Phase 2 and Phase 3, as seen in Figure 4.20. An increased number of classes present has a positive effect on the accuracy of Phase 2, while it has a negative effect on Phase 3. The accuracy is 100% when there is only one class present, as the input image is completely uniform, and there is only one possible class to choose for the output.

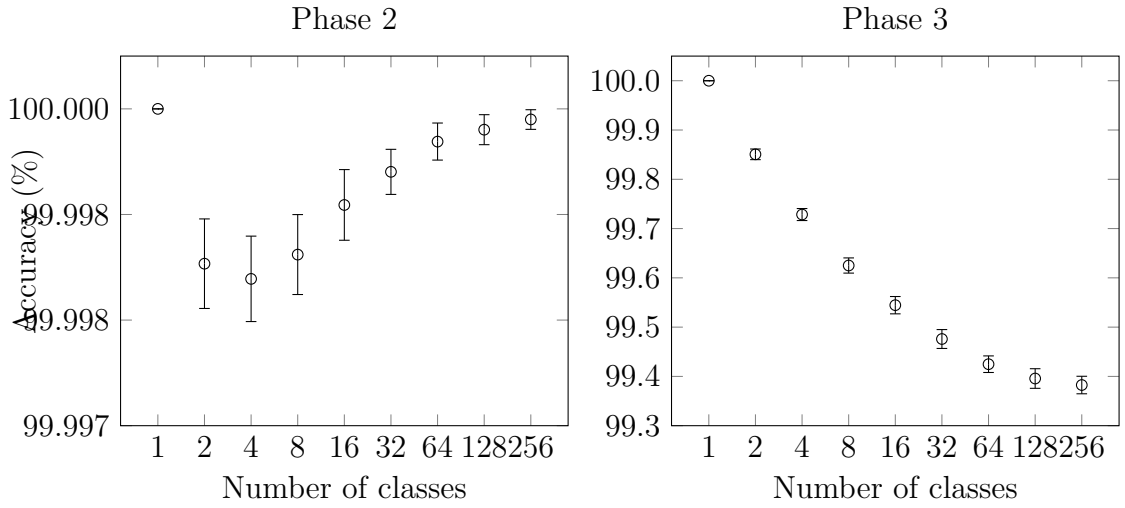

Figure 4.20: Output accuracy as a function of the number of classes in the input image.
Input image size = 1024, Kernel size = 127

5

Discussion

This chapter contains the discussions about how well the results fulfilled the purpose and goals. Problems and how this thesis could have been done differently is also discussed in this chapter. There will also be a discussion about interesting questions that arose during this thesis work.

5.1 MATLAB

The MATLAB implementations served a great purpose throughout this thesis. Inexperience regarding image processing and GPGPU programming was a problem during the start of the project, and learning both at the same would have been very hard. By separating the development of proof-of-concept code and OpenCL code the process was made easier. Thus, MATLAB provided clarity and comprehension at an increased rate. However, we believe that the required time for each Phase would been less if we decided to not use MATLAB as an intermediate step. Nevertheless, we also believe that without the use of MATLAB our understanding of the algorithms and implementations would be lacking, and thus this thesis would not have had the same quality in both theory and results.

As stated in Section 3.1, MATLAB has already defined mathematical methods which allowed us to efficiently develop our understanding of the algorithms. However, the MATLAB and OpenCL C implementations and functionality is different in many ways, as mentioned in Section 3.5.1, the translation between these two require more time and effort at certain points in this thesis.

5.2 OpenCL

OpenCL was chosen as it was preferred by the company over other solutions, and it is open source. Since neither one of us had previous experience of OpenCL, a learning period was needed. The main challenge with OpenCL was that we could not debug the code other than looking at the compilation errors, due to the lack of experience.

This caused further complications when implementing the OpenCL, and a better knowledge beforehand would have improved the overall learning experience.

An alternative for OpenCl could be Compute Unified Device Architecture (Cuda). Cuda is created by Nvidia and dose also only work with Nvia graphical cards. We decided together with Qlik to not restrict the thesis just to Nvida products and thus we choose OpenCl.

5.3 Hardware

A more expensive GPU would obviously have brought better performance. However, as shown in the Results chapter 4, it is enough to have an off-the-shelf, low- to mid-end GPU to experience an improved performance.

The Nvidia 1060 6GB is a consumer GPU with rendering as its primary purpose. This introduced an unexpected limitation, as Nvidia implements a failsafe mechanism that is triggered when the GPU is considered unresponsive, even when no monitor is connected to the GPU. To try to make the GPU responsive again, the failsafe essentially restarts the GPU. In the general use case, the applications try to handle this error and rendering is resumed.

The timeout for the failsafe was in the range of five to ten seconds or our system, resulting in forcefully cancelled OpenCL computations. The failsafe can be disabled, but not on the consumer grade Nvidia GPU directly, only through the operating system. This has the drawback that all rending devices has the failsafe disabled. Since our computations required up to several minutes on the GPU, we needed to disable this fail safe.

5.4 Phase 1

The results from the original algorithm was for the most part in line with our expectations. The impact of the number of classes in the input image shows a strange relationship to run time, as seen in Figure 4.3. The reasons for this is unknown to us, but the code was examined for errors and multiple test runs was made, and the relationship was found throughout all.

A similar anomaly can be seen in the results from Phase 1, presented in Figure 4.6. The algorithm should not behave much different when the number of classes in changed, but nevertheless a large jump in run time can be observed when it exceeds 16 classes. We believe that this depends on the built-in caching algorithms of the GPU used. As the number of classes is small, all classes can be kept in the cache at all time, but as the number increases cache misses occurs. These cache misses are

very expensive, and could explain the observed increase in run time.

We also believe that this is the case for all results in this thesis that was produced on the GPU. However, the jump is not as noticeable, as both Phase 2's and Phase 3's complexity is dependent on the number of classes. These phases also try to consider one class at a time, reducing the amount of cache misses.

Implementing the RGBA and Local memory optimization were proven to be very successful, reducing the run time needed to almost 50% in comparison to the initial implementation. However, the time needed for these implementations could not be accompanied for Phase 2 and Phase 3, thus none were implemented.

5.5 Phase 2

The implementation of Phase 2 was accompanied by many difficulties, as it tested our newly gained knowledge of OpenCL to produce a new algorithm, without a working reference. In addition to our decision to not implement memory access optimization, as was done in Phase 1, the quality of the implementation ultimately suffered.

Many parts of the results from Phase 2 did not meet our expectations, especially when the rank or number of classes increased. We did not expect that Phase 2 could be slower than even the original algorithm run on the CPU, as shown in Figure 4.2 and Figure 4.9. However, when the rank is very small, and to some extent when the number of classes are small, the algorithm outperforms both the original algorithm and Phase 1, as seen in Figure 4.2, Figure 4.5, and Figure 4.8. Our expectation was that the use of convolution would allow outperform the original algorithm in all cases.

In addition to not implementing memory access optimization, the OpenCL implementation for Phase 2 is largely restricted by the fact that each rank must be calculated in separate steps. Thus, many OpenCL kernels are required to fully calculate the output, which introduces large amounts of overhead as OpenCL moves data back and forth from the host machine to the GPU. This can be seen in the GPU utilization during execution of Phase 2, when compared to either the original algorithm, Phase 1, or Phase 3.

In Figure 5.1, this utilization is shown. A kernel size of 15 was used, with a rank of 15 for Phase 2. GPU load measures the utilization of the GPU cores, memory controller load measures the memory bandwidth usage on the GPU, and bus interface load is the bandwidth usage between the host machine and the GPU.

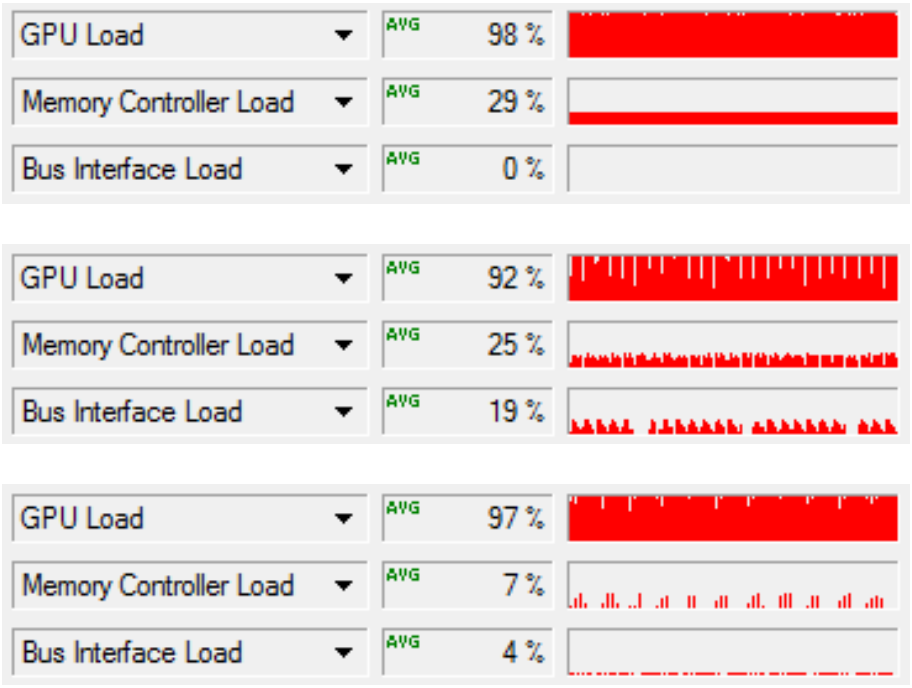

Figure 5.1: Statistics of average load over 30 seconds of GPU, memory controller, and bus interface.
From the top: Phase 1, Phase 2, Phase 3
Input image size = 1024, Kernel size and rank = 15, Number of classes = 256

As seen in Figure 5.1, the load during the execution of Phase 1 is constant and high, while it is very uneven during Phase 2. Phase 3 is also somewhat uneven, but this is due to the algorithm being executed many times, creating a small pause in between each execution. The bus interface is very busy during Phase 2, in comparison to Phase 1 and Phase 3, as each processed singular vector pair requires data to be moved between the host and the GPU.

We believe that a better implementation which tries to minimize the interfacing needed between the host machine and the GPU would greatly improve the performance as the rank increases.

We presented two different theories for SVD calculation, as seen in Section 2.5. We decided to only implement the Jacobi method, as the company uses symmetric kernels in all their optimizations. Additionally, the Jacobi improves the condition number over other approaches, reducing the extent of floating point errors, while being fast [17].

The Rayleigh method was only considered during research, as it was unclear whether a more general solution was needed. However, the specific implementation of SVD was not an important aspect of this thesis, and can easily be replaced by a different implementation if desired. It is also notable that the Jacobi method produced very accurate results, as seen in Section 4.7.

5.6 Phase 3

As with Phase 2, no memory access optimization was considered for the OpenCL implementation of Phase 3. However, the results are very good, and the promised constant complexity held true in all our results. Of course, the number of classes still has an impact on the run time. However, it cannot exceed 256, unlike the kernel size.

We believe that the usage of Gaussian kernels in the company's program would greatly improve the rate at which they can process their geographical data.

5.7 Accuracy evaluation

The accuracy is very high for the improved algorithms, and thus the error is most likely negligible in most applications when considering the speed up gained. Additionally, today the company chooses their kernels in a subjective way, where a good enough outcome is decided by manual inspection.

One explanation for the errors introduced could be due to the way floating point numbers handles precision. A part of the available bits is used to store what is called the mantissa, which represents the digits of the floating-point numbers. An exponent is then used to define where the decimal point lies. Thus, the precision of the floating point is limited by the mantissa. When adding two floating points, the exponents must be modified such that they match. This modification essentially moves the decimal point, and may result in some of the digits to lie outside of the range of the mantissa. This happens when a small number is added to a large number, resulting in some information from the small number to be lost [18].

Thus, one can imagine that when more floating points are to be summed, the larger the error will become. As the kernel size increases, the amount of arithmetic operations done by the convolution algorithm also increases in Phase 2. This is also true for Phase 3, as a larger kernel size is equivalent to a larger standard deviation for the Gaussian distribution. This in turn widens the truncation location, $R = 3\sigma$, which increases the amount of arithmetic operations done when calculating the first and second cosine terms.

The accuracy impact from the number of classes, presented in Figure 4.20, is more interesting. Given a constant input image size, a larger number of classes present reduces the number of occurrences of any class in the input image, given that the input image is uniformly distributed. Thus, the amount of non-zero additions and multiplications should also be less. This is most likely what is happening in Phase 2, incurring an increase of accuracy as the number of classes increase. However, this statement is not true for Phase 3, as can be seen in the figure.

6

Conclusion

In this thesis, we have shown that geographical processing can be a good candidate for GPU computations, even with low- or mid-end hardware. The aim was to investigate how the run time of this process could be improved by utilizing a GPU, as well as improving the process regarding run time and complexity.

Detailed algorithms of all implementations are presented in this thesis, with both MATLAB and pseudo code available, as well as an OpenCL translation description. Both the MATLAB and the pseudo code allows the reader to gain an understanding of the implementations, while OpenCL allowed us to interface with the GPU.

In addition to moving a CPU algorithm to the GPU, two different optimization approaches for this algorithm were successfully implemented. The first was founded on well-known image filter theory. The second utilized a state-of-the-art constant time Gaussian filtering technique.

Run time and accuracy measurements have been conducted. Results of individual implementations have been evaluated, as well as been compared with each other. An improvement of run time can be observed when moving the algorithm to the GPU, with further improvements available by exploiting the GPU architecture. The usage of Gaussian filtering kernels reduces the run time by orders of magnitude, while still producing accurate results. This is due to complexity being improved from $O(N^2)$ to $O(C)$ where C is classes, where $0 < N$ and $0 < C < 256$.

Bibliography

[1] S. N. Sinha, J.-M. Frahm, M. Pollefeys, and Y. Genc, "Gpu-based video feature tracking and matching," in *EDGE, Workshop on Edge Computing Using New Commodity Architectures*, vol. 278, 2006, p. 4321.

[2] K. Sugimoto and S.-i. Kamata, "Fast gaussian filter with second-order shift property of dct-5," in *Image Processing (ICIP), 2013 20th IEEE International Conference on*. IEEE, 2013, pp. 514–518.

[3] S. Ryoo, C. I. Rodrigues, S. S. Stone, J. A. Stratton, S.-Z. Ueng, S. S. Baghsorkhi, and W. H. Wen-mei, "Program optimization carving for gpu computing," *Journal of Parallel and Distributed Computing*, vol. 68, no. 10, pp. 1389–1401, 2008.

[4] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund *et al.*, "Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, pp. 451–460, 2010.

[5] D. C. de Andrade, "Case study: High performance convolution using opencl ___local memory," http://www.cmsoft.com.br/opencl-tutorial/case-study-high-performance-convolution-using-opencl-___local-memory/, 2011, (Accessed on 06/07/2017).

[6] ——, "Opencl image2d variables," http://www.cmsoft.com.br/opencl-tutorial/opencl-image2d-variables/, 2010, (Accessed on 06/07/2017).

[7] F. Ino, S. Yoshida, and K. Hagihara, "Rgba packing for fast cone beam reconstruction on the gpu," in *SPIE Medical Imaging*. International Society for Optics and Photonics, 2009, pp. 725 858–725 858.

[8] F. Strugar, "An investigation of fast real-time gpu-based image blur algorithms | intel® software," https://software.intel.com/en-us/blogs/2014/07/15/an-investigation-of-fast-real-time-gpu-based-image-blur-algorithms, July 2014.

[9] T. Archer, "Procedurally generating terrain," in *44th annual midwest instruction and computing symposium, Duluth*, 2011, pp. 378–393.

[10] I. T. Young, J. J. Gerbrands, and L. J. Van Vliet, *Fundamentals of image processing*. Delft University of Technology Delft, 1998.

[11] J. Fingas, "Roadmap says transistors will stop shrinking in 5 years," https://www.engadget.com/2016/07/25/roadmap-says-transistors-will-stop-shrinking/, 07 2016, (Accessed on 06/07/2017).

[12] C. Gregg and K. Hazelwood, "Where is the data? why you cannot debate cpu vs. gpu performance without the answer," in *(IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software*, April 2011, pp. 134–144.

[13] Khronos, "The opencl specification," https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf, 11 2012, (Accessed on 06/07/2017).

[14] JogAmp.org, "Jogamp.org - java graphics, audio, media and processing libraries exposing opengl, opencl, openal and openmax," https://jogamp.org/, 2017, (Accessed on 06/07/2017).

[15] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[16] S. Batterson and J. Smillie, "The dynamics of rayleigh quotient iteration," *SIAM journal on numerical analysis*, vol. 26, no. 3, pp. 624–636, 1989.

[17] J. Demmel and K. Veselić, "Jacobi's method is more accurate than qr," *SIAM Journal on Matrix Analysis and Applications*, vol. 13, no. 4, pp. 1204–1245, 1992.

[18] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys (CSUR)*, vol. 23, no. 1, pp. 5–48, 1991.

A

Appendix 1

Table A.1: Error dependent on the input size

inSize	inArea	p2Mean	p2Sd	p2Mean%	p2Sd%	p3Mean	p3Sd	p3Mean%	p3Sd%
256	65536	65535.61	0.61783266	0.99999404	9.427378E-6	65474.87	22.190022	0.99906725	3.3859286E-4
512	262144	262140.84	1.8929695	0.99998796	7.2211055E-6	261619.02	69.03446	0.99799734	2.6334557E-4
1024	1048576	1048558.56	4.845856	0.9999834	4.6213686E-6	1045695.75	154.78839	0.9972532	1.4761771E-4

Table A.2: Error dependent on the Kernel size

kernelSize	inArea	p2Mean	p2Sd	p2Mean%	p2Sd%	p3Mean	p3Sd	p3Mean%	p3Sd%
1	1048576	1048575.5	0.65874076	0.9999995	6.2822414E-7	1048322.5	16.897465	0.99975824	1.6114678E-5
3	1048576	1048575.6	0.5972158	0.99999964	5.6954934E-7	1048323.2	14.684352	0.9997589	1.4004089E-5
7	1048576	1048575.5	0.717107	0.9999995	6.838865E-7	1048323.8	15.883019	0.9997595	1.5147228E-5
15	1048576	1048575.44	0.75638694	0.99999946	7.213468E-7	1047005.94	40.551456	0.9985027	3.8672883E-5
31	1048576	1048574.44	1.0671874	0.9999985	1.0177492E-6	1045975.56	55.807457	0.99752	5.322214E-5
63	1048576	1048570.44	2.375889	0.9999947	2.2658244E-6	1045576.25	88.42398	0.9971392	8.432768E-5
127	1048576	1048558.0	4.0687404	0.99998283	3.880253E-6	1045707.4	150.678	0.99726427	1.4369773E-4

Table A.3: Error dependent on number of Classes

nClasses	inArea	p2Mean	p2Sd	p2Mean%	p2Sd%	p3Mean	p3Sd	p3Mean%	p3Sd%
1	1048576	1048576.0	0.0	1.0	0.0	1048576.0	0.0	1.0	0.0
2	1048576	1048560.6	4.4489927	0.99998534	4.24289E-6	1047010.2	113.22229	0.9985067	1.0797719E-4
4	1048576	1048559.1	4.2513337	0.9999839	4.054388E-6	1045728.94	127.8238	0.9972848	1.2190227 4E-4
8	1048576	1048561.5	3.9733963	0.9999862	3.789326E-6	1044644.3	161.60544	0.99625045	1.5411896E-4
16	1048576	1048566.44	3.5107195	0.9999909	3.348083E-6	1043800.2	184.35141	0.99544543	1.758112E-4
32	1048576	1048569.75	2.2511725	0.99999404	2.1468854E-6	1043080.44	200.84253	0.994759	1.9153836E-4
64	1048576	1048572.75	1.845661	0.9999969	1.7601595E-6	1042544.44	176.5083	0.99424785	1.6833143E-4
128	1048576	1048573.94	1.4924389	0.99999803	1.4233007E-6	1042239.7	208.60017	0.9939572	1.9893663E-4
256	1048576	1048574.94	0.98734415	0.999999	9.4160475E-7	1042101.1	186.34837	0.9938251	1.7771566E-4