

Institutionen för datavetenskap  
Department of Computer and Information Science

Master's Thesis

# Using OpenCL to Implement Median Filtering and RSA Algorithms: Two GPGPU Application Case Studies

Lukas Gillsjö

Reg Nr: LIU-IDA/LITH-EX-A-15/004-SE  
Linköping 2015-03-11



**Linköping University**  
**INSTITUTE OF TECHNOLOGY**

Department of Computer and Information Science  
Linköpings universitet  
SE-581 83 Linköping, Sweden



Master's Thesis

Using OpenCL to Implement Median  
Filtering and RSA Algorithms: Two  
GPGPU Application Case Studies

Lukas Gillsjö

Reg Nr: LIU-IDA/LITH-EX-A-15/004-SE  
Linköping 2015-03-11

Supervisor: **Maghazeh, Arian**  
IDA, Linköpings universitet  
**Seeger, Malin**  
Syntronic

Examiner: **Kessler, Christoph**  
IDA, Linköpings universitet



# Abstract

Graphics Processing Units (GPU) and their development tools have advanced recently, and industry has become more interested in using them. Among several development frameworks for GPU(s), OpenCL provides a programming environment to write portable code that can run in parallel. This report describes two case studies of algorithm implementations in OpenCL. The first algorithm is Median Filtering which is a widely used image processing algorithm. The other algorithm is RSA which is a popular algorithm used in encryption. The CPU and GPU implementations of these algorithms are compared in method and speed. The GPU implementations are also evaluated by efficiency, stability, scalability and portability. We find that the GPU implementations perform better overall with some exceptions. We see that a pure GPU solution is not always the best and that a hybrid solution with both CPU and GPU may be to prefer in some cases.

# Sammanfattning

Graphics Processing Units (GPU) har på senare tid utvecklats starkt. Det har kommit fler verktyg för utveckling och det ses mer och mer som ett allvarligt alternativ till att utveckla algoritmer för Central Processing Units (CPU). Ett populärt system för att skriva algoritmer till GPU är OpenCL. Det är ett system där man kan skriva kod som körs parallellt på många olika plattformar där GPU är den primära. I denna rapport har vi implementerat två olika algoritmer i OpenCL. En av dem är en algoritm för bildbehandling kallad Median Filtering. Den andra är RSA vilket är en populär krypteringsalgoritm. Vi jämför de seriella CPU implementationerna mot de parallella GPU implementationerna och analyserar även stabiliteten, portabiliteten och skalbarheten hos GPU implementationerna. Resultatet visar att GPU implementationerna överlag presterar bättre än de seriella med vissa undantag. Vi ser att en ren GPU lösning inte alltid är den bästa utan att framtiden nog ligger i ett hybridssystem där CPU:n hanterar vissa fall och GPU:n andra fall.



# Acknowledgments

*I want to thank my examiner Christoph Kessler for his help during this thesis. I also want to thank my supervisor Arian Maghazeh for all the comments, discussions and advice that I've received during this time. I would also like to thank my supervisor at Syntronic, Malin Seeger. I also want to thank all the other people at Syntronic for the help and good times, and Åsa Detterfelt for giving me the opportunity to do this thesis.*

*Special thanks to my family for supporting me during my whole education.*

*Lukas Gillsjö*



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Problem statement . . . . .	1
1.2	Algorithm choices . . . . .	1
1.3	Main challenge . . . . .	1
1.4	Outline of the thesis . . . . .	2
1.5	Abbreviations and definitions . . . . .	2
<b>2</b>	<b>Background</b>	<b>5</b>
2.1	Difference between CPU and GPU . . . . .	5
2.2	OpenCL . . . . .	5
2.2.1	Execution Model . . . . .	6
2.2.2	Memory Model . . . . .	7
2.3	SkePU . . . . .	9
<b>3</b>	<b>Median filtering</b>	<b>11</b>
3.1	Algorithm description . . . . .	11
3.1.1	Constant-Time Median Filtering . . . . .	12
3.2	General parallelizations done . . . . .	14
3.2.1	Data parallelization . . . . .	14
3.2.2	Histogram parallelization . . . . .	14
3.3	Other median filtering alternatives . . . . .	15
3.4	GPU specific parallelizations and optimizations . . . . .	15
3.4.1	Complementary Cumulative Distribution Functions . . . . .	15
3.4.2	CCDF example . . . . .	18
3.4.3	Tiling . . . . .	21
3.4.4	Memory . . . . .	22
3.5	SkePU . . . . .	23
3.5.1	Median filtering in SkePU . . . . .	23
3.5.2	Sequential calculation of Median kernel . . . . .	23
3.5.3	Histogram median method . . . . .	24
3.5.4	Choice of algorithm . . . . .	27
3.6	Experimental Evaluation . . . . .	28
3.6.1	Tiling . . . . .	28
3.6.2	Image2D or buffers . . . . .	29

3.6.3	CPU vs GPU implementation . . . . .	30
3.6.4	Stability and scalability of GPU implementation . . . . .	31
3.6.5	Portability . . . . .	31
3.6.6	Ease of programming . . . . .	32
3.7	Challenges and benefits of porting this algorithm to OpenCL . . . .	32
3.7.1	Challenges . . . . .	32
3.7.2	Benefits . . . . .	32
<b>4</b>	<b>RSA</b>	<b>33</b>
4.1	Algorithm description . . . . .	33
4.1.1	General description . . . . .	33
4.1.2	Algorithm usage . . . . .	33
4.1.3	Key generation . . . . .	34
4.2	General optimizations done . . . . .	35
4.2.1	Multiprecision system . . . . .	36
4.2.2	Chinese Remainder Theorem . . . . .	36
4.2.3	Square-and-multiply . . . . .	39
4.2.4	Montgomery reduction . . . . .	41
4.2.5	Barret reduction . . . . .	45
4.3	General parallelizations done . . . . .	46
4.3.1	Add carries and overflows . . . . .	47
4.3.2	Addition . . . . .	48
4.3.3	Subtraction . . . . .	49
4.3.4	Multiplication . . . . .	49
4.3.5	Division by $R$ . . . . .	51
4.3.6	Modulo by $R$ . . . . .	51
4.4	GPU specific parallelizations and optimizations . . . . .	53
4.4.1	Multiprecision operations . . . . .	53
4.4.2	Kernel handling . . . . .	53
4.4.3	Memory handling . . . . .	54
4.4.4	Workgroup optimization . . . . .	54
4.5	Experimental Evaluation . . . . .	55
4.5.1	CPU vs GPU implementation . . . . .	55
4.5.2	GPU implementation vs SSLShader . . . . .	59
4.5.3	Stability and scalability of the GPU implementation . . . .	60
4.5.4	Portability . . . . .	61
4.5.5	Ease of programming . . . . .	61
4.5.6	Test conclusions . . . . .	61
4.6	Challenges and benefits of porting this algorithm to OpenCL . . . .	61
4.6.1	Challenges . . . . .	61
4.6.2	Benefits . . . . .	62

---

<b>5</b>	<b>General insights about porting algorithms to GPU</b>	<b>63</b>
5.1	Memory handling . . . . .	63
5.2	Choice of algorithm . . . . .	64
5.3	Debugging . . . . .	64
5.3.1	Buffers . . . . .	64
5.3.2	Printing . . . . .	64
5.3.3	CodeXL . . . . .	64
5.3.4	Macros . . . . .	64
5.4	SkePU . . . . .	65
5.5	General OpenCL tips . . . . .	65
<b>6</b>	<b>Related Work</b>	<b>67</b>
6.1	GPGPU . . . . .	67
6.2	Median filtering . . . . .	68
6.3	RSA . . . . .	69
<b>7</b>	<b>Discussion</b>	<b>71</b>
<b>8</b>	<b>Future work</b>	<b>73</b>
	<b>Bibliography</b>	<b>75</b>
<b>A</b>	<b>Median filter code</b>	<b>79</b>
<b>B</b>	<b>RSA Kernel code</b>	<b>81</b>



# Chapter 1

## Introduction

### 1.1 Problem statement

Graphics Processing Units (GPU) and their development tools have advanced recently, and industry has become more interested in using them. Among several development frameworks for GPU(s), OpenCL provides a programming environment to write portable code that can run in parallel. At the moment, there is not much documentation about developing with this framework in non-scientific applications. Syntronic AB is an engineering design house which works with both electronics and software and they are interested in more documentation and experience in this area which prompted this master thesis. The goal of the thesis is to consider two popular algorithms from two different application domains, implement them on GPU and evaluate them for performance, scalability, stability and portability.

### 1.2 Algorithm choices

The two chosen algorithms are median filtering and RSA encryption and decryption. Median filtering was chosen because it is an often used image filter which has a high potential for massive parallelization. Syntronic has an interest in image processing and how its performance can be increased by parallelization which is why this algorithm was chosen. Another area that Syntronic has interests in is RSA cryptosystems and how they can be sped using parallelism and GPUs.

### 1.3 Main challenge

The main challenge of this thesis will be to implement the algorithms in an efficient manner using OpenCL on GPU. The algorithms need to be analyzed for opportunities to parallelize. We also need to realize what is not worth parallelizing. We also need to analyze the GPU hardware and OpenCL framework to see how we

can implement these parallelizations and what we can do to optimize them for speed.

## 1.4 Outline of the thesis

We will begin by giving background on the technology and frameworks used. We will follow up by describing the Median filter algorithm and how we parallelize it. The results of the parallelization and implementation will then be presented and discussed. After that the RSA cryptosystem will be presented and discussed in the same manner. The end of the thesis will be about general OpenCL and GPU. There will be an appendix in the end with the main GPU kernels. Note that all the code will not be in the appendix, only enough to get a general overview of how the kernels work.

## 1.5 Abbreviations and definitions

- CCDF - Complementary Cumulative Distribution Function.
- CPU - Central Processing Unit.
- CRT - Chinese Remainder Theorem, used in the RSA implementation.
- CTMF - Constant Time Median Filtering.
- CU - Compute Unit, consists of Processing Elements and memory. Executes work-groups.
- Global memory - Main memory. Visible from host side, and shared across all threads.
- GPU - Graphic Processing Unit.
- Local memory - Memory shared by threads in the same work-group.
- OpenCL - Framework for writing parallel code.
- OpenMP - Open Multi-Processing, an API for parallelizing operations on the CPU.
- PE - Processing Element, executes work-items.
- Private memory - Memory only available to a single thread.
- RSA - Asymmetric cryptosystem designed by Rivest, Shamir and Adleman.
- SIMD - Single Instruction Multiple Data, the same instruction is performed on multiple data.
- SIMT - Single Instruction Multiple Thread, the same instruction is performed by multiple threads in parallel.

- SkePU - A skeleton programming framework for multicore CPU and multi-GPU systems.
- Work-item - Instance of a kernel that is to be executed by a Processing Element.
- Work-group - A group of work items.



# Chapter 2

## Background

This chapter gives a brief overview of the main differences between a CPU and a GPU, and also some information about programming for a GPU using OpenCL and SkePU.

### 2.1 Difference between CPU and GPU

A multi-core CPU mainly uses the Multiple Instructions-Multiple Data (MIMD) strategy for parallelizing the work. This means that a given CPU can execute different threads with different instructions and different data on different cores. A GPU on the other hand uses the Single Instruction-Multiple Threads (SIMT) strategy, which means that all threads will run the same instructions. This approach is powerful when there is a large amount of data so that multiple threads can be executed on the data.

They also differ in hardware. A CPU has typically a few powerful cores (typically 4 on consumer CPUs right now), which means that the cores have a low instruction latency but the CPU has a lower theoretical throughput than a GPU. A GPU has many cores, hundreds, even thousands depending on the model. These cores are not as powerful as the CPU cores, which gives the GPU cores a higher latency. The winning factor of the GPU is the high throughput possible with all these cores.

Using all these cores in a SIMT manner puts some requirements on the algorithm used though. It needs to be highly parallelizable, and in a data parallel way. Also, computations with a high arithmetic intensity are more likely to get close to the maximum performance of a GPU, as the GPU can use this fact to hide the slow memory accesses.

### 2.2 OpenCL

OpenCL is a framework for portable programming for different kinds of devices. OpenCL code can execute across heterogeneous platforms consisting of many dif-

ferent types of devices, for example CPUs, GPUs, digital signal processors (DSPs), field-programmable gate arrays (FPGAs) and other types of processors. The framework defines different models but leaves the implementation of these models up to the device. The most interesting model for us, when running a program on one GPU, are the Execution Model and the Memory Model. See the OpenCL 1.2 specification [23] for more information.

### 2.2.1 Execution Model

OpenCL is executed by having a host send kernels and data to the devices, which then execute these kernels using the data provided. When a kernel is submitted for execution by the host, an index space is created. This index space defines how many instances of the kernel will be executed. Each instance is called a work-item, and these work-items can be grouped together into work-groups. Each work-item has a global ID and will run the same kernel as all the other work-items. The data provided may make the work-items differ in execution path, but they all run the same kernel. An illustration of the index space, work-items and work-groups is given by Figure 2.1. Each cell in the illustration represents a work-item, while each collection of cells is a work group. The global ID of each thread is given by its position in the grid which can be thought of as the index space. The work-items are executed in groups called wavefronts.

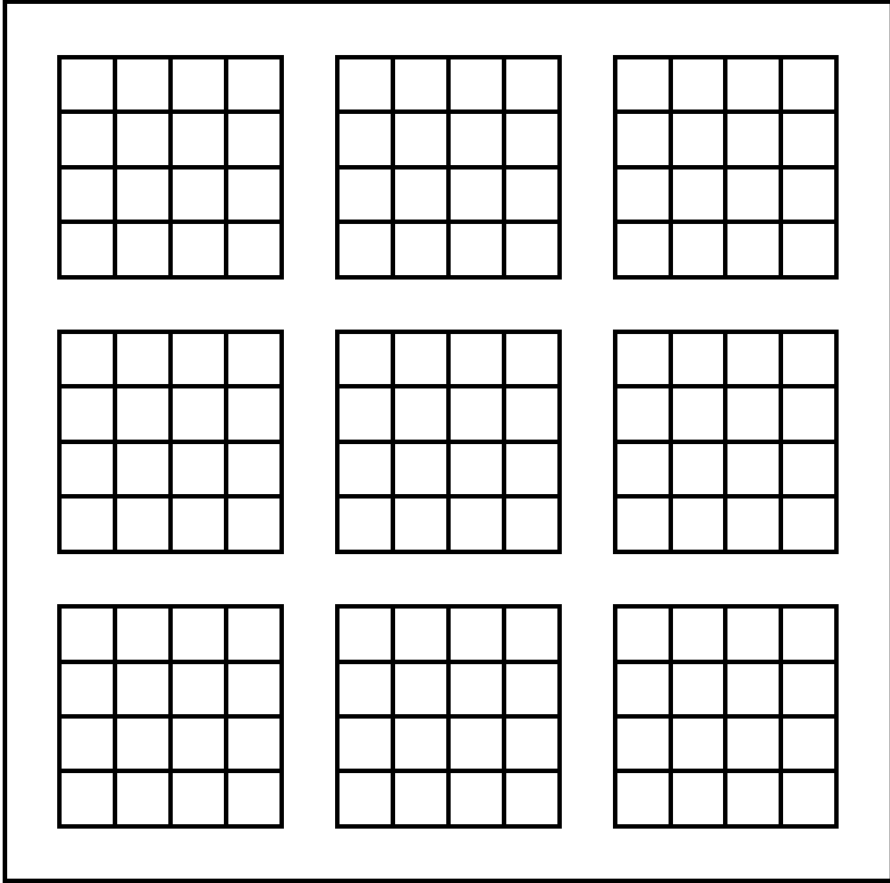


Figure 2.1: An illustration of the index space, work-items and work-groups in OpenCL. Adapted from the OpenCL specification [23].

The index space, which in OpenCL is called an NDRange, can be in 1, 2 or 3 dimensions. In Figure 2.1, both the NDRange and the work-group are in 2 dimensions. The kernel will run for each work-item in the index space and each work-item will use its global id, which is in 1, 2 or 3 dimensions depending on the dimensionality of the NDRange, to figure out which data it will work on.

### 2.2.2 Memory Model

The conceptual OpenCL device architecture is a part of the memory model, and it is how the architecture of the devices looks according to OpenCL. A visual overview is given by Figure 2.2. This model resembles the hardware architecture of the GPU, which is why OpenCL code typically runs faster on a GPU than a CPU.

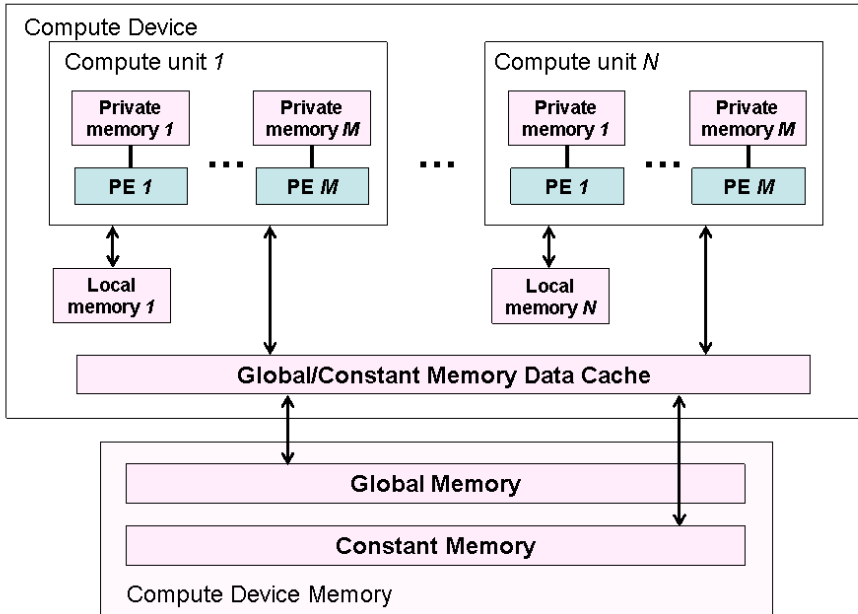


Figure 2.2: The memory model of OpenCL. Image reprinted with permission from the OpenCL specification [23]. © Khronos Group 2012

We will go through all the components in a bottom-up approach. First we have the Processing Element. This is a virtual scalar unit that is used for executing work-items. A specific work-item may execute on one or more processing elements. Private Memory is private to each work-item. Variables defined in Private Memory are not visible to other work-items.

A Compute Unit consists of Processing Elements and Local Memory. Work-groups execute on Compute Units. A work-group is guaranteed to only be executed on a single Compute Unit.

Each Compute Unit has Local Memory attached. This memory is shared between all the work-items in a work-group, and is a way of communicating between them. The Global/Constant Memory Cache is a cache between the Compute Device and the Global Memory and Constant Memory.

A Compute Device consists of Compute Units, and might be for example a GPU. There is also the Compute Device Memory which has two parts, Global Memory and Constant Memory.

The GPUs nowadays conform really well to this model, while CPUs are not as good a match.

### Coalesced memory access

An important concept when speeding up memory intensive kernels is memory coalescing. This concept is primarily used when accessing the global memory which is the slowest memory on the GPU. Coalesced memory access is done by combining multiple memory accesses into a single access reducing the total amount of memory accesses done which speeds up the execution time. This is achieved by moving data in the memory so that the data we know will be accessed at the same is continuous in memory. All memory accesses from the Processing Elements will be combined into as few accesses as possible by the Compute Unit. This is achieved by reading the memory in segments which are called cache lines. The size of these cache lines differ but an example would be 64 bytes.

Let us presume that we have a kernel which is being executed by one wavefront, 64 threads. Each thread is accessing 8 bytes of memory and no two threads are accessing the same data. If all blocks of 8 bytes are spread out in a way in memory such that the distance, in bytes, between each block is larger than 56 bytes we would need 64 memory accesses to read all the data. If the blocks instead are laid out in a continuous manner in memory only 8 memory accesses are needed. The reason is because of the cache-line. As each memory access will read 64 bytes, each memory access will read data for 8 threads.

So by making sure that the memory accessed by each Compute Unit is laid out in a continuous manner we can reduce the amount of memory accesses dramatically. Doing this also means that we might need to rearrange the data in memory before launching our kernels. Testing needs to be done to evaluate if the time saved by reducing the memory accesses is larger than the time spent rearranging the memory.

## 2.3 SkePU

SkePU is a skeleton programming framework for multicore CPUs and multi-GPU systems. A skeleton can be thought of as high-level program structure or algorithm. The skeletons in SkePU describes how an operation will be carried out on a set of data. Given an operation to apply, and data to apply it to, the skeleton will carry out the operation according to its specifications. Let us say that we have a large array. We want to add the number 2 to each element to this array. That is what is called a Map operation, an identical operation that is carried out on all elements. Doing this on a single-threaded CPU would mean that we just loop over all the elements and add 2 to each. We could also write an OpenCL kernel that adds 2 to a single element and launch this kernel using as many threads as there are elements. But this would mean that we need to setup the OpenCL environment, move the buffer to GPU memory, launch the kernel, and copy the resulting buffer back. It would be a lot of code just to do this simple task.

A simpler way to do this on the GPU would be to use SkePU. SkePU has a map skeleton, where the only thing that we need to provide is the function that is to be applied to each element and the input buffer. We would then send this to the Map skeleton, and SkePU would handle all the environment work and copying of

memory. This could be accomplished in less than 10 lines of code. It can also be run using OpenCL, CUDA, OpenMP or normal sequential CPU. It also has support for multiple GPUs.

# Chapter 3

## Median filtering

This chapter will describe the first case study, the median filtering algorithm, what optimizations are done, and what the results are.

### 3.1 Algorithm description

Median filter is an often used filter for noise reduction or blurring of images. It is the foundation of many more advanced filters, which makes it a good candidate for optimizations using parallelization. Also, as images increase in amount of pixels a multicore solution is becoming more promising. A median filter is applied to each pixel and replaces it by the median of itself and its neighbours. The neighbours are chosen using a square kernel with radius  $r$ , where  $r$  usually is even to make the median calculation faster.

7	9	9	5	10
6	10	1	12	9
10	22	5	8	5
6	13	11	23	7
5	2	6	8	1

Figure 3.1: An example of a median filter with a radius of 2. In this case, the median of the shaded area is 11 which would become the new value for the pixel in the middle.

In Figure 3.1 we see a kernel with a radius of 2. The value of the pixel in the middle of the shaded area will be the median of the pixels in the  $3 \times 3$  area around it.

### 3.1.1 Constant-Time Median Filtering

The fastest serial implementation that we know of at the moment is the Constant-Time Median Filtering (CTMF) [17]. We will begin by defining what a histogram is as it is something that Constant-Time Median Filtering is built on. A histogram is another way to represent a collection of data where we count how many occurrences of each value that exists in the data. An example is the way Constant-Time Median Filtering uses histograms. Let us presume we want to create a histogram for an image. Each pixel in the image has a value in the range  $[0, 255]$ . For this reason the histogram we create will have 256 bins. Each bin will contain how many pixels there are with that specific value. So if the second bin, with index 1, had a value of 134 it would mean that there are 134 pixels with the value 1 in the image.

The Constant-Time Median Filtering algorithm is based on adding and subtracting such histograms. To see why that is effective, we first need to talk about some attributes of histograms.

One of these is that of distributivity. We will use the notation  $H(A)$  to denote the histogram for the region  $A$ . The operator  $\cup$  will be used to denote a union of regions. Adding histograms is done by adding each pair of bins together. For disjoint regions  $A$  and  $B$ , Equation 3.1 holds.

$$H(A \cup B) = H(A) + H(B) \quad (3.1)$$

So for example, if we create a histogram for each column of an image, we can calculate the histogram for 3 columns by adding together the 3 column-histograms.

The action of adding histograms to each other takes linear time in terms of the number of bins in the histogram. In our case the number of bins is the same as the bit depth of the image. The bit depth of an image is the number of bits used to indicate the color of a single pixel. The bit depth of the images we use is 256. The same can be said for subtracting histograms. That is, Equation 3.2 holds. Subtracting histograms is done by subtracting the bins pairwise.

$$H(A \cup B) = H(A) - H(B) \quad (3.2)$$

This means that adding or subtracting histograms from each other is a constant time operation in regards to the amount of elements in the histogram. In our case, the number of elements in a given histogram is the number of pixels that the histogram contains.

Adding and subtracting an element from a histogram is also a constant time operation.

Finding the median in a histogram is also constant with regards to the amount of elements in the histogram.

With these attributes in mind, we can construct an algorithm for median filtering which uses constant time in regards to filter size. In CTMF, there are two different types of histograms. One is the *kernel histogram*, which represents the median kernel. This kernel histogram will sweep over the image from left to right, and begin on the left end of the next row when the last row is done. The other type of histogram is the *column histogram*. These histograms are added and subtracted to the kernel histogram as it sweeps over the image. The amount of elements in a column that the column histogram includes is equal to  $2r + 1$ .

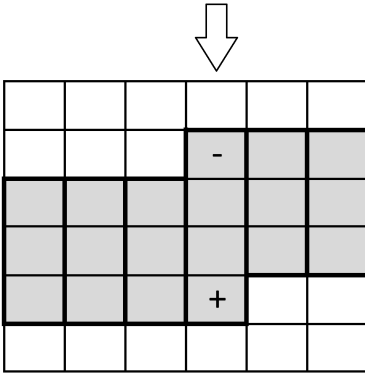


Figure 3.2

Downward movement of column histogram. Adapted from [17].

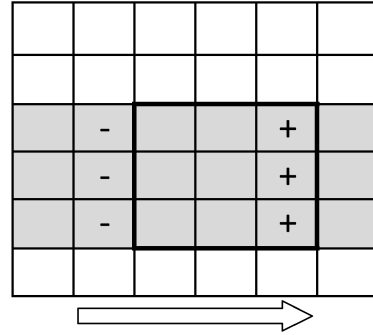


Figure 3.3

Kernel histogram sweeping right. Adapted from [17].

Figure 3.2 shows how the column histograms move down one row by removing the topmost pixel and adding the pixel from the row below the histogram, while Figure 3.3 shows how the kernel histogram sweeps over the column histograms by removing the leftmost column histogram and adding the rightmost column

histogram. Both these steps will be done in constant time with regard to the filter radius  $r$ .

There is still one step left, that sadly is not constant in time. It is the initialization. To initialize the column histograms, we need to create a new histogram from the  $r$  first elements in each column. This takes  $\mathcal{O}(w \cdot r)$  time, where  $w$  is the width of the image. This initialization is only done once per image. The kernel histogram also needs to be built in the initialization and each time we go down one row. This consists of adding the  $r$  first columns histograms together, and has a time complexity of  $\mathcal{O}(r)$ . This is only done once per row. Given a large enough image, the amortized time complexity of this algorithm will be  $\mathcal{O}(1)$  in terms of filter size. More about this can be read in the paper by Perreault and Hébert [17].

The amount of work done is linear in terms of the number of pixels in the image. For every row with  $w$  pixels in the image we need to move  $w$  histograms down one row. For every pixel in the row we need to move the kernel histogram one step to the right and calculate the median. All these operations are constant. Given that each row contains  $n$  pixels the work done is  $\mathcal{O}(w \cdot n)$  where  $w \cdot n$  will be the number of pixels in the image. We also have the initialization work of  $\mathcal{O}(w \cdot r)$ . Both of these scale linearly with the number of pixels in the image.

## 3.2 General parallelizations done

### 3.2.1 Data parallelization

One simple way to parallelize this algorithm is doing data parallelization. We divide the image into subimages and process different parts of the image in parallel using this algorithm. Here there are two things to consider when choosing how large each subimage will be. If the subimages are too large, the GPU will not be used to its full potential as the parallelization is too small. We will also have a problem where the amount of local memory used by each work-group will be large which means that the number of wavefronts in execution in each Compute Unit will decrease. If the subimages are too small, the initialization overhead will get too large, slowing down the algorithm even though we have a lot of parallelism.

### 3.2.2 Histogram parallelization

Another way is to parallelize the histogram operations. Adding and removing a pixel is not parallelizable on a single histogram, but if we remove a pixel from all histograms in parallel and then add the next rows pixel in parallel to all histograms we can speed it up. We can also parallelize the adding and subtracting of histograms. Each such operation consists of 256 independent scalar operations which means that we can parallelize them with a simple map operation. The problem is finding the median. This is sadly a sequential operation which is hard to parallelize. Therefore we do a small change to the algorithm by using Complementary Cumulative Distribution Functions (CCDFs) [19] instead of histograms. More information about this is in a later chapter.

### 3.3 Other median filtering alternatives

There are other methods for calculating the median in an image. The easiest would be by using stencil computations. For every pixel, read in the pixel in the neighbourhood with radius  $r$  and calculate the median. This is easily parallelizable and also scales linearly in terms of image size. This approach will be implemented in SkePU, see Section 3.5. The main reason we chose CTMF instead is because of the scaling with filter radius. The SkePU implementation scales linearly in terms of image size while the CTMF implementation is constant. This can also be seen in the results later where the SkePU implementation performs well with lower filter sizes but worse with larger filter sizes.

### 3.4 GPU specific parallelizations and optimizations

#### 3.4.1 Complementary Cumulative Distribution Functions

As said in Section 3.2.2, there is a problem of finding the median in histograms in a parallel fashion. By using CCDFs instead, we can do all the operations in parallel. This approach has been done before by Sánchez and Rodríguez [20] in their implementation of median filtering. We will show how this works by defining CCDF-sorting. To define the Complementary Cumulative Distribution Function (CCDF) we first define the Cumulative Distribution Function (CDF). The CDF is a function given by Equation 3.3 for a real-valued random variable  $X$ . The right-hand side of the equation is the probability that the random variable  $X$  takes on values less than or equal to  $x$ .

$$F_X(x) = P(X \leq x) \quad (3.3)$$

The CCDF is the complement of the function defined in Equation 3.3. The formal definition for this is given in Equation 3.4.

$$\bar{F}_X(x) = 1 - F_X(x) = 1 - P(X \leq x) = P(X > x) \quad (3.4)$$

Now, if we instead of working on random variable  $X$  want to work with a vector  $\mathbf{y}$  with  $n$  elements and where  $a \leq y_i \leq b \forall i \in [0, n-1]$ , and we want to sort this vector, we replace the probability function  $P(X > x)$  with the counting function  $C_x(\mathbf{y})$  given by Equation 3.5.

$$C_x(\mathbf{y}) = \sum_{i=0}^{n-1} I_{[y_i > x]} \quad (3.5)$$

The indicator function  $I$  is defined in Equation 3.6.

$$I_{[y_i > x]} = \begin{cases} 1 & \text{if } y_i > x \\ 0 & \text{otherwise} \end{cases} \quad (3.6)$$

We will use the  $ccdf(\mathbf{x})$  to denote the Complementary Cumulative Distribution Function of  $\mathbf{x}$  in the following definitions.

To sort the vector  $\mathbf{y}$  we need to create a temporary vector  $\mathbf{t} = ccdf(\mathbf{y})$  where  $t_j = C_j(\mathbf{y}) \in [0, n]$ ,  $j \in [a, b]$ . From this we can get the sorted vector of  $\mathbf{y}$ , called  $\mathbf{z}$ . This vector is given by  $\mathbf{z} = ccdf(\mathbf{t})$  where  $z_k = C_k(\mathbf{t}) \in [a, b]$ ,  $k \in [0, n-1]$ . So for example, assume that we have the vector  $\mathbf{y} = [5, 3, 8, 7, 2]$ ,  $a = 0$ ,  $b = 8$ . Calculating  $\mathbf{t}$  we get  $\mathbf{t} = [5, 5, 4, 3, 3, 2, 2, 1, 0]$ . We use this to calculate  $\mathbf{z} = [8, 7, 5, 3, 2]$  which is the sorted vector. We can see that, if we only wanted the maximum number in this vector, we only needed to calculate  $z_0 = C_0(\mathbf{t})$ , and if we only wanted the minimum, we could calculate  $z_{n-1} = C_{n-1}(\mathbf{t})$ . So if we wanted the  $k$ th number in the sorted sequence  $\mathbf{z}$  we only need to calculate  $z_k = C_k(\mathbf{t})$ . The median can be found in the middle of the sorted sequence. If we choose the vector  $\mathbf{y}$  so that  $n$  is odd, we will find the median at  $k = (n-1)/2$ . So to find the median of the vector  $\mathbf{y}$  we need to calculate  $z_{(n-1)/2} = C_{(n-1)/2}(\mathbf{t})$ .

CCDFs also share the same attribute as the histograms in that CCDFs can be added and subtracted to each other to create CCDFs of larger areas, see Equation 3.7.

$$ccdf(A \cup B) = ccdf(A) + ccdf(B) \quad (3.7)$$

This means that we can use CCDFs in the same way as histograms with a CCDF for each column which will be added and subtracted to create the CCDF for the kernel. So why did this change make the algorithm better suited for GPU? Let us compare some points.

### Initialization

Initialization of a histogram in parallel is possible. The initialization process only happens once for each column in each subimage. This can be parallelized by, for example, using one thread for each pixel that is added in the initialization and adding to the same histogram using atomic operations. So if we, for example, are going to create a histogram from 32 pixels in a column we launch 32 threads and let them add to the column histogram at the same time. The problem with this is that nearby pixels have a high chance of being similar, meaning that the memory access to the histogram might be sequentialized as the threads will attempt to write to the same bins in the histogram.

For CCDFs it is a little different. In the case of histograms, only one bin is updated for each pixel added to the histogram. In the worst-case scenario for the CCDF, all bins need to be updated. This means that using one thread per bin in the CCDF is more efficient. So all the threads read the same pixel and update the bin assigned to them. This is not as bad as it might seem, as all of the threads for a CCDF are in the same work-group meaning that the data can be cached. All writes to the CCDF will also be coalesced.

So for the initialization step histograms need fewer threads, but the bin accesses might be serialized if many threads accesses the same bin. For CCDFs more threads are needed, but the bin accesses will be parallelized as all threads access different bins. The initialization is written in pseudocode at Algorithm 1. Note that it assumes that all CCDFs are adjacent in memory.

---

**Algorithm 1** Initialization of CCDFs

---

**function** INITIALIZATION(columnCCDFs, startRow, startColumn, numRows, numColumns, image)

**Input:** The CCDFs for the columns, the starting row, the starting column, the number of rows and columns in the image and the image itself.

```

for CCDFId from 0 to numColumns - 1 do
    globalBinId = CCDFId * NUM_BINS + threadId
    columnsCCDFs[globalBinId] = 0
    for rowId from startRow to startRow + numRows - 1 do
        if image[startColumn + CCDFId][rowId] > threadId then
            columnsCCDFs[globalBinId] ++
        end if
    end for
end for
end function

```

---

**Adding and subtracting histograms and CCDFs**

Both the histograms and CCDFs will have the same bin size, and adding two of them together is just adding bins with the same index together. The same can be said for subtraction. This can be done by using one thread per bin, so these operations will be parallelized in the same way for both.

So for adding and subtracting histograms and CCDFs to/from each other the same parallelizations can be done. The code for adding and subtracting CCDFs and histograms is given below in Algorithm 2 and Algorithm 3.

---

**Algorithm 2** Add two CCDFs.

---

**function** ADD(CCDF1, CCDF2)

**Input:** The two CCDFs to be added.

**Output:** The result of the addition.

```

    CCDFResult[threadId] = CCDF1[threadId] + CCDF2[threadId]
    return CCDFResult
end function

```

---



---

**Algorithm 3** Subtract two CCDFs.

---

**function** SUBTRACT(CCDF1, CCDF2)

**Input:** The two CCDFs to be subtracted.

**Output:** The result of the subtraction.

```

    CCDFResult[threadId] = CCDF1[threadId] - CCDF2[threadId]
    return CCDFResult
end function

```

---

### Adding and removing elements

Adding or removing elements in a histogram only changes the value of one bin. In a CCDF, all the values in all of the bins might change. This means that more threads are needed for the CCDF.

So the histogram wins out in adding and removing elements. The pseudocode for adding and removing elements can be found at Algorithm 4 and Algorithm 5.

---

#### Algorithm 4 Add an element to the CCDF

---

```

function ADDELEMENT(CCDF, element)
  Input: The CCDF and the element to be added.
  if  $element > threadId$  then
     $CCDF[threadId] = CCDF[threadId] + 1$ 
  end if
end function

```

---



---

#### Algorithm 5 Remove an element from the CCDF

---

```

function ADDELEMENT(CCDF, element)
  Input: The CCDF and the element to be added.
  if  $element > threadId$  then
     $CCDF[threadId] = CCDF[threadId] - 1$ 
  end if
end function

```

---

### Finding the median

Finding the median in a histogram is a typically sequential operation. There are techniques for reducing the amount of operations needed which will be explained in Section 3.5.3, but it will still be sequential.

In a CCDF, the median is found by calculating the element at the  $(n - 1)/2$ 'th position in the sorted sequence that can be retrieved from the CCDF. This can be done by calculating  $C_{(n-1)/2}(\mathbf{t})$  where  $\mathbf{t}$  is the CCDF. This can be done in parallel using a parallel reduction algorithm. Further improvements were done inspired by Harris' presentation on optimizing parallel reductions on CUDA [13]. This is where the problem with parallel histograms is found. Finding the median is a sequential operation, and will be done once per pixel. By using a CCDF, we can parallelize this as seen in Algorithm 6.

#### 3.4.2 CCDF example

We will here use an example to show how the CCDF algorithm works. We will use an image where the only allowed values are in the range  $[0, 7]$  to simplify the process. We median-filter a small part of an image with 16 pixels using a  $3 \times 3$  filter.

**Algorithm 6** Calculate the median from a CCDF

---

**function** CALCULATEMEDIAN(CCDF, kernelRadius)

**Input:** The CCDF the radius of the kernel used.

 $numIndices = (2 * kernelRadius + 1)^2 - 1$ 
 $sumBuffer[threadId] = CCDF[threadId] > numIndices/2$ 
**for**  $stride = NUM\_BINS/2; stride > 0; stride >= 1$  **do**

    **if**  $threadId > stride$  **then**

         $sumBuffer[threadId] = sumBuffer[threadId] +$   
             $sumBuffer[threadId + stride]$ 

    **end if**
**end for**

    **return**  $sumBuffer[0]$ 
**end function**


---

A	B	C	D	Index	0	1	2	3	4	5	6	7
3	3	4	4	A	3	3	3	2	1	1	0	0
6	7	0	7	B	3	2	2	1	1	1	1	0
4	1	2	5	C	2	2	1	1	0	0	0	0
2	6	4	1	D	3	3	3	3	2	1	1	0

Figure 3.4: Calculation of column CCDFs.

What we see in Figure 3.4 is the starting column CCDFs. The left figure is the image where the lightly shaded area is the area covered by the CCDFs. The right figure shows the CCDFs, one for each column. Each CCDF consists of 3 elements because we know that the filter size is  $3 \times 3$ . We will begin calculating the median on the second row, second column. This is to make the example more clear. We initialize the kernel CCDF by adding the CCDFs *A*, *B* and *C* together. The darkly shaded area in Figure 3.5 is the kernel.

A	B	C	D	Index	0	1	2	3	4	5	6	7
3	3	4	4	Kernel	8	7	6	4	2	2	1	0
6	7	0	7									
4	1	2	5									
2	6	4	1									

Figure 3.5: Initializing of kernel CCDF.

We now have a CCDF of the kernel. We calculate the median by counting the

elements in the kernel CCDF which are larger than  $(2 \cdot (\text{kernelRadius} - 1))^2 / 2 = 4$ . The number of elements that are larger than 4 is 3, so 3 is the median of the pixel at location (1, 1) in the image using a  $3 \times 3$  median filter. We will now move the kernel CCDF one pixel to the right by removing CCDF A and adding CCDF D.

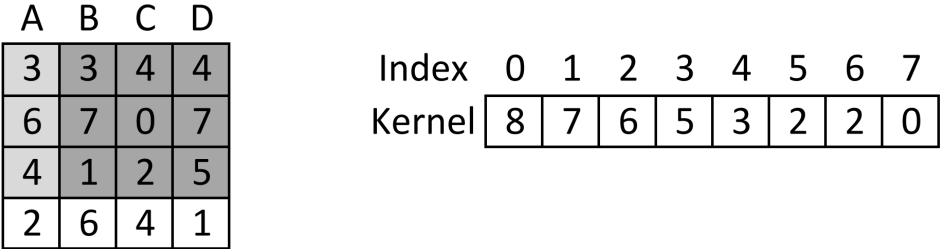


Figure 3.6: Moving the kernel CCDF one pixel to the right.

Figure 3.6 gives us the new kernel area and CCDF. We do as before and count the number of elements which are larger than 4. The answer is 4, which is the median of the pixel in position (2, 1). The process of moving the kernel to the right continues until we have calculated the median of all pixels on that row. Then we need to move the column CCDFs one step down.

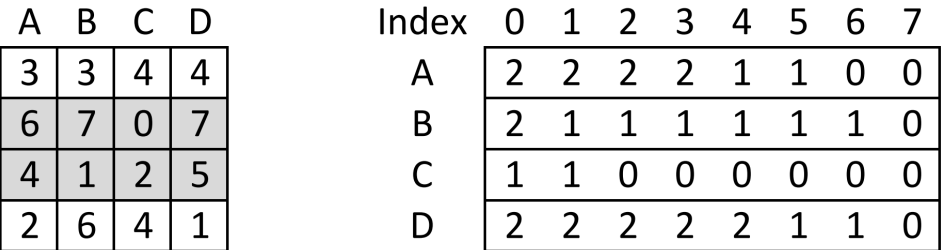


Figure 3.7: Removing the first row from the column CCDFs.

Figure 3.7 shows how the column CCDFs look after removing the first row from them. For example, in CCDF A, all elements with an index smaller than 3 were decremented.

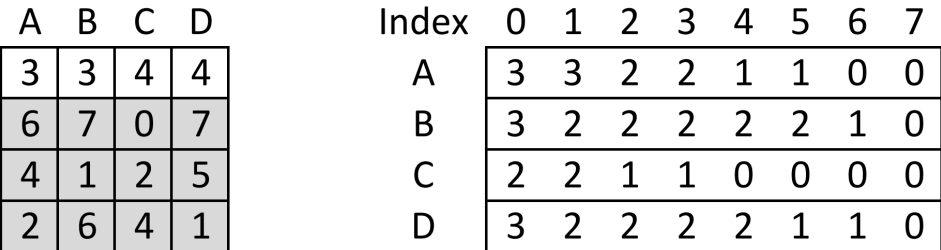


Figure 3.8: Adding the last row to the column CCDFs.

Figure 3.8 shows how the column CCDFs look after adding the last row to them. For example, in CCDF A, all elements with an index smaller than 2 were incremented. The process with initializing the kernel CCDF now begins again and the kernel CCDF will sweep from left to right over the column CCDFs.

### 3.4.3 Tiling

We use a technique called tiling to subdivide the image into subimages. We will then let one work-group work on a subimage each. The benefit of this is that all the structures used can be stored in the local memory, making memory accesses much faster than if global memory was used. As the algorithm uses dependencies between the different rows to speed up the time taken, we will initially use as large subimages as possible. The size of a subimage is limited by the size of the local memory. As the images we work on have a bit-depth of 256, each CCDF will use 256 bins. If we make the assumption that the filter radius will never be larger than 127 then we can use one byte to represent each bin in the column CCDFs and two bytes to represent each bin in the kernel CCDF. There is also a temporary array used when calculating the median which also uses two bytes for each bin. Each column CCDF then has a size of 256 bytes, while the kernel CCDF and the temporary array has a size of 512 bytes each. To find out the maximum number of CCDFs that we can store in the local memory we need to find out the size of the local memory on the device. When that is known Equation 3.8 can be used to calculate the maximum amount of columns.

$$\text{Maximum number of columns} = (\text{localMemSize} - 2 * 512) / 256 \quad (3.8)$$

The value  $2 * 512$  comes from that the kernel CCDF and temporary array also needs to be on local memory and uses 2 bytes per value. On the device we use for developing, this is 32768 bytes. This means that the total amount of CCDFs we can store in the local memory is 125. This means that a work-group can handle at maximum 125 columns, as 1 CCDF is needed for the kernel. There are some more points we need to consider when choosing the amount of columns that each work-group should handle:

1. The local memory is shared for each compute unit. This means that all the work-groups on that compute unit share this local memory. They cannot access each other's local memory, but they need to share the memory space.
2. We want to maximize occupancy, that is, how many work-groups each compute unit is responsible for. We want this value as large as possible to enable context switching to hide the memory latency. This means as small tiles as possible, to minimize the local memory needed for each tile.
3. We want to do as few initializations as possible, which means as large tiles as possible.

The last two points go against each other. This is where we need to do testing and decide the optimal size of the tiles. These limitations mean that the largest

tile size possible is 125 columns, while the smallest is 1 column. If each tile is 125 columns then we would have 1 workgroup per compute unit which is a low occupancy. Testing will be done to establish a good tile size.

Another part is the amount of rows in a tile. By having many rows in a tile we will have few column CCDF initializations which is a good thing. But having fewer rows in a tile means that there will be more tiles per image which means higher parallelism.

### 3.4.4 Memory

In OpenCL there are two ways of accessing data from global memory. Either by reading it as normal data, or by using a sampler and accessing it as a texture. A sampler is an object that handles access to textures. The sampler can be initialized with different rules, for example what coordinates that should be used when accessing the sampler, how edges should be handled, how interpolation between pixels should be handled. The sampler can then be used by simply supplying it with a coordinate and it will return the pixel value given the rules it was initialized with. The operations of the sampler are also implemented closer to the hardware by the hardware vendor which makes it faster than if we would implement those operations in the kernel. Sampling in textures is achieved by using the Image2D [23] structure in OpenCL. The following sections will look at the benefits and problems of both.

#### Image2D

A large benefit of using Image2D is that accessing data outside the image will be handled by the sampler. So if we try to read from a coordinate outside the image the sampler used will interpolate that value by the rule that was decided on when creating the sampler. So if we for example initialize the sampler with the rule `CLK_ADDRESS_CLAMP_TO_EDGE`, all out-of-image accesses will be clamped to the edge instead. This means that there is no branching in the code for the edge cases as those are handled by the hardware. This means less thread divergence, which means faster code. The problem with Image2D is that there is a cache that always will be checked first. This means, that if one accesses each element only once, which is what this algorithm does, every access will be a cache miss. Another drawback is that that memory access for Image2D is not coalesced. Implementing memory access using Image2D was straightforward and no modifications were needed.

#### Buffers

Buffers are one-dimensional data containers that can be sent and read from the GPU. The large benefit of buffers is the ability to do coalesced reading and writing. This speeds up memory accesses if the reading is done in a coalesced manner.

Implementation of memory access using buffers needed some extra work. Firstly, the pixel information in the images needs to be altered. The pixels are stored in the following order in the image:  $r_0, g_0, b_0, r_1, g_1, b_1, \dots, r_{n-1}, g_{n-1}, b_{n-1}$  where  $n$  is the

number of pixels in the image. Median filtering is done one channel at a time, which means that this memory layout is not good for coalesced reading. For example, the cache line on the development GPU is 64 bytes wide. Each color value of a pixel is 8 bit. This means that a total of 8 color values can be read in one memory access. As we are only interested in one channel, we would at most get 3 values to use in each memory access. Assume we have an image with 65536 pixels. Using the above memory layout, 21846 memory accesses would be needed to read all the data in the red channel of the image. This can be optimized by rearranging the data in the following order before execution.  $r_0, r_1, \dots, r_{n-1}, g_0, g_1, \dots, g_{n-1}, b_0, b_1, \dots, b_{n-1}$ . Where  $n$  is the number of pixels in the image. Using this memory layout means that all the color values in the cache line will be used except when reading at the end of each color sequence. Using the above example with 65536 pixels, only 8192 memory accesses would be needed to read all the data in the red channel. This difference will scale with the size of the image, meaning that the speed gained will be higher on larger images.

So to use buffers to their full potential, we begin by rearranging the data in the image as shown before. Then we run the algorithm, and join the data together again before saving the resulting image. We also need to make sure that the data is read in a coalesced way in the algorithm.

One drawback is that we need to handle the edge cases ourselves, by clamping the indexing of the image.

## 3.5 SkePU

Median filtering is a kernel operation, which means that it is trivial to parallelize. The simplest parallelization is just to let each thread handle one channel in one pixel and launch  $numPixels \cdot numChannels$  threads. Let the thread sample the image around the pixel that it is responsible for and output a value. This approach was implemented in SkePU [28].

### 3.5.1 Median filtering in SkePU

SkePU has a MapOverlap2D [5] skeleton which gives us the possibility to write a simple function which will be launched for every element in a Matrix and which has access to the Matrix elements in its vicinity. That is exactly what we need for implementing Median filtering. What we need now is a sequential algorithm to implement in the kernel.

### 3.5.2 Sequential calculation of Median kernel

Note that the CTMF is an algorithm to median filter a whole image. What we need now is an algorithm to calculate a single median value given a median kernel and pixel position. Two different algorithms were considered. Histograms were chosen because of the results in the comparison of median filtering algorithms done by Juhola, Katajainen and Raita [12]. Median of medians was also considered given

the work by Dinkel and Zizzi on median finding on digital images [6]. We will go through both and motivate why we chose one of them.

### Median of medians

A selection algorithm is an algorithm for finding the  $k$ th smallest number in an array. It can be done in two ways. Either via sorting the array and just picking the number at position  $k$  or by partitioning the array. The algorithm we are interested in is partitioning-based which means that it will try to partition the array to reduce the amount of elements that needs to be sorted.

The most fitting for our problem is Quickselect. Quickselect works by partitioning the input into two using a pivot value. It is then decided which partition the wanted value is and that part is further partitioned using a pivot value. This continues until the needed value is found. Let us first define a function for partitioning an array using a pivot value, see Algorithm 7.

This function will be used in the selection function given at Algorithm 8.

Quickselect as given in Algorithm 8 has a best-case performance of  $\mathcal{O}(n)$ , but has a worst-case performance of  $\mathcal{O}(n^2)$ . The problem is in how to choose the pivot. The optimal value in our case would be the median. That would mean that only one partitioning would be needed. There is an algorithm called Median of medians that tries to do exactly this. It approximates a median which will then be used as a pivot value in the quickselect algorithm. It does this by dividing the data into groups of 5, calculating the median of each group and then calculating the median of these group-medians. This median is what is called the median of medians. This is not the true median though, only an approximation that can later be used as a pivot in the quickselect algorithm to find the true median faster. The number 5 is chosen because it is odd and small enough that the median calculation in each group will be fast. This algorithm uses a variant of the select algorithm at Algorithm 8: Instead of returning the value of the selected element, *selectIdx* returns the index of that element. Also note that *selectIdx* cannot use the median of medians as pivot selection function.

The median of medians algorithm at Algorithm 9 moves all the group-medians to the beginning of the array and then runs the Quickselect algorithm on these. By using the median of medians function at Algorithm 9 as a pivot selection strategy we can reduce the worst-case time of the select algorithm from  $\mathcal{O}(n^2)$  to  $\mathcal{O}(n)$  when using it to get the median. Further proof of this can be read in Dinkel and Zizzi's work on median filtering [6]. Reading the kernel values into an array also has a worst-case time of  $\mathcal{O}(n)$ . This means that using quickselect with median of medians as a pivot selection strategy has a worst and best case performance of  $\mathcal{O}(n)$ .

### 3.5.3 Histogram median method

Another method is to use histograms to calculate the median. This is a method that works great when we have domain knowledge about the possible values of the elements we are trying to find the median of. We know that the values are integers

---

**Algorithm 7** Partition an array using a pivot value

---

```

function PARTITION(array, left, right, pivotIndex)
Input: An array, its left and rightmost indices and the pivotIndex.
Output: The index of the pivot value after partitioning.
    pivotValue = array[pivotIndex]
    swap(array[pivotIndex], array[right])
    storeIndex = left
    for  $i$  from left to right - 1 do
        if array[i] < pivotValue then
            swap(array[storeIndex], array[i])
            storeIndex ++
        end if
    end for
    swap(array[right], array[storeIndex])
    return storeIndex
end function

```

---



---

**Algorithm 8** Select the k-th smallest element in the array and return its index.

---

```

function SELECT(array, left, right, k)
Input: An array, its left and rightmost indices and the index k of the element
        searched for.
Output: The k-th smallest element.
    if left = right then
        return array[left]
    end if
    pivotIndex = selectedPivot(array, left, right)
    partition(list, left, right, pivotIndex)
    if k = pivotIndex then
        return array[k]
    else if  $n < \text{pivotIndex}$  then
        return select(array, left, pivotIndex - 1, k)
    else
        return select(array, pivotIndex + 1, right, k)
    end if
end function

```

---

---

**Algorithm 9** Select and approximate median.

---

```

function MEDIANOFMEDIANS(array, left, right)
Input: An array, its left and rightmost indices.
Output: The approximate median.
    numMedians = ceil((right - left)/5)
    for  $i$  from 0 to numMedians do
        subLeft = left +  $i * 5$ 
        subRight = subLeft + 4
        if subRight > right then
            subRight = right
        end if
        medianIdx = selectIdx(array, subLeft, subRight,
                               (subRight - subLeft)/2)
        swap(array[left +  $i$ ], array[medianIdx])
    end for
    return selectIdx(array, left, left + numMedians - 1, numMedians/2)
end function

```

---

in the range  $[0, 2^b - 1]$ , where  $b$  is the number of bits in the input values. In the case of our images we know that  $b = 8$ . The algorithm given in Algorithm 10 shows how to find the median in a histogram.

In our case where  $b = 8$  this leads to an average of 128 comparisons and subtractions, and a worst case of 255 comparisons and subtractions. We can improve this further by using multi-layer histograms [2]. We will use two levels of histograms. One with  $2^{b/2}$  bins and one with  $2^b$  bins. The FindMedianNaive function is then rewritten as the FindMedian function given as Algorithm 11.

---

**Algorithm 10** Find the median in a histogram.

---

```

function FINDMEDIANNAIVE(histogram, N)
Input: A histogram and the number of elements  $N$  in the histogram.
Output: The median.
    count =  $N/2$ 
    for  $i$  from 0 to  $2^b - 1$  do
        count = count - histogram[ $i$ ]
        if count < 0 then
            return  $i$ 
        end if
    end for
end function

```

---

---

**Algorithm 11** Find the median in a histogram.

---

```

function FINDMEDIAN(coarseHistogram, fineHistogram, N)
Input: The coarse and fine histograms and the number of elements  $N$  in the
histograms.
Output: The median.
     $coarseIndex = 0$ 
     $count = N/2$ 
    for  $i$  from 0 to  $2^{b/2-1}$  do
         $count = count - coarseHistogram[i]$ 
        if  $count < 0$  then
             $coarseIndex = 2^{b/2} * i$ 
             $count = count + coarseHistogram[i]$ 
            break
        end if
    end for
    for  $i$  from 0 to  $2^{b/2-1}$  do
         $count = count - fineHistogram[coarseIndex + i]$ 
        if  $count < 0$  then
            return  $coarseIndex + i$ 
        end if
    end for
end function

```

---

Looking at our case with  $b = 8$  again we see that we have reduced the number of comparisons and subtractions to 16 in the average case and 32 in the worst case. We have sacrificed some memory in the process as we now need  $2^b + 2^{b/2}$  bytes of memory to store the histograms.

The worst case complexity of this algorithm is  $\mathcal{O}(2^b + n)$ . The  $2^b$  term will be constant while the  $n$  term will increase with larger filters.

### 3.5.4 Choice of algorithm

The algorithm that was finally chosen was the histogram approach. The reasons are as follows:

- The histogram approach is similar to our other approach which makes for an interesting comparison
- Both of these algorithms suffer from thread divergence when run on the GPU. But the operations in the divergent parts are much cheaper in the case of the histogram approach. Only comparisons and subtractions are done, while the median of medians approach would do calculation of pivot elements and partitioning of the input array in the divergent parts.
- The histogram approach is easier to implement. This is important as we cannot create functions in the kernel which we later can call.

One drawback of the histogram approach is the use of memory. The histogram needs  $2^b + 2^{b/2}$  bytes of memory for its histograms while the median of medians approach needs  $n$  bytes of memory for the array. As we have  $b = 8$ , this means that the histogram approach will use more memory as long as the radius of the kernel is smaller than 8 which is usually the case.

## 3.6 Experimental Evaluation

The AMD GPU used in these tests is an ASUS Radeon R9 280x-DC2T-3GD5 with 32 Compute Units running at 970 MHz. The CPU in this computer is an Intel i3570k and the amount of RAM is 8 GB. The OpenCL version used is 1.2.

The Nvidia GPU is a Tesla M2050 with 14 Compute Units running at 1150 MHz. This computer has 16 processors, each being an Intel Xeon E5520 running at 2.27 GHz, and the amount of RAM is 24 GB. The OpenCL version used is 1.1.

The CPU used in the testing is an Intel i3570k with 4 physical cores running at 3.4 GHz and the amount of RAM is 8 GB.

The compiler used was g++ 4.8.1 with the O2 and msse2 flags. No extra OpenCL flags were used when compiling the kernels.

### 3.6.1 Tiling

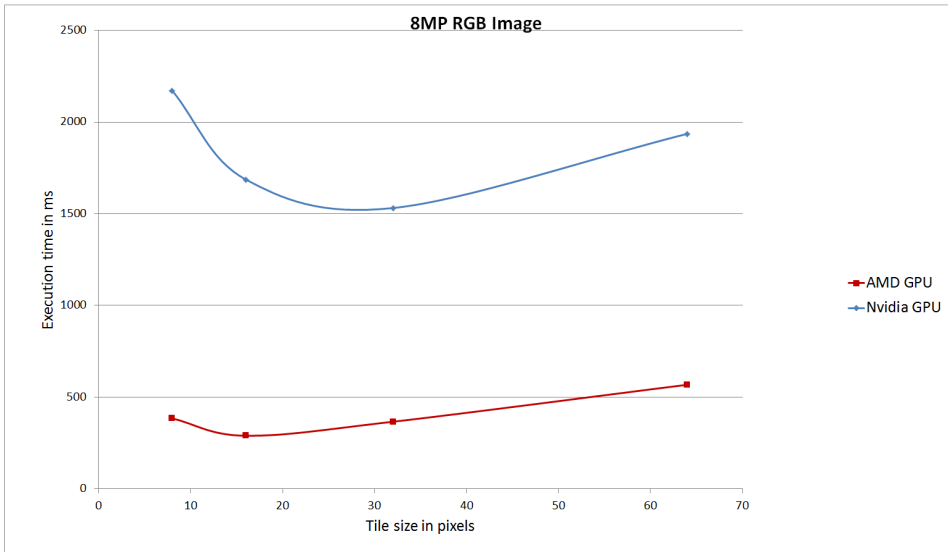


Figure 3.9: Comparison of execution time for different tile sizes using quadratic tiles

Tests were done with various tiling sizes. Figure 3.9 shows a comparison of different sizes using quadratic tiles.

Tests were also done using rectangular tiles, but they showed no performance gain over quadratic tiles. We see that  $16 \times 16$  is the best tile size for the AMD GPU, while  $32 \times 32$  is the most optimal for the Nvidia GPU.

### 3.6.2 Image2D or buffers

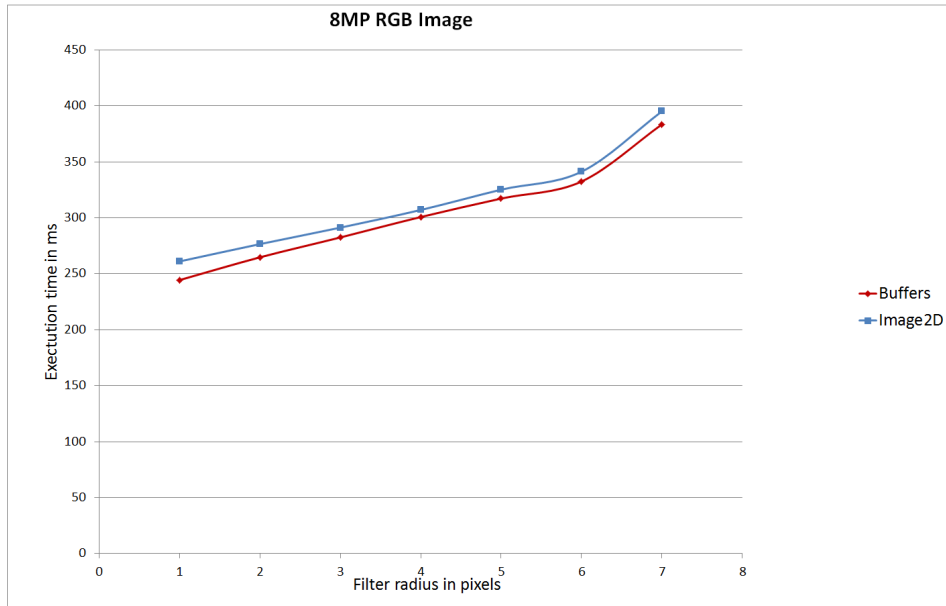


Figure 3.10: Comparison of buffer and Image2D implementation run on a 8 megapixel RGB image.

Both methods of memory access was implemented and tested. The results can be seen in Figure 3.10. This test was only done on the AMD GPU as OpenCL 1.2 is needed for Image2D. The Nvidia GPU only supports OpenCL 1.1.

The buffer implementation performs slightly better and is the one used in the rest of the thesis.

### 3.6.3 CPU vs GPU implementation

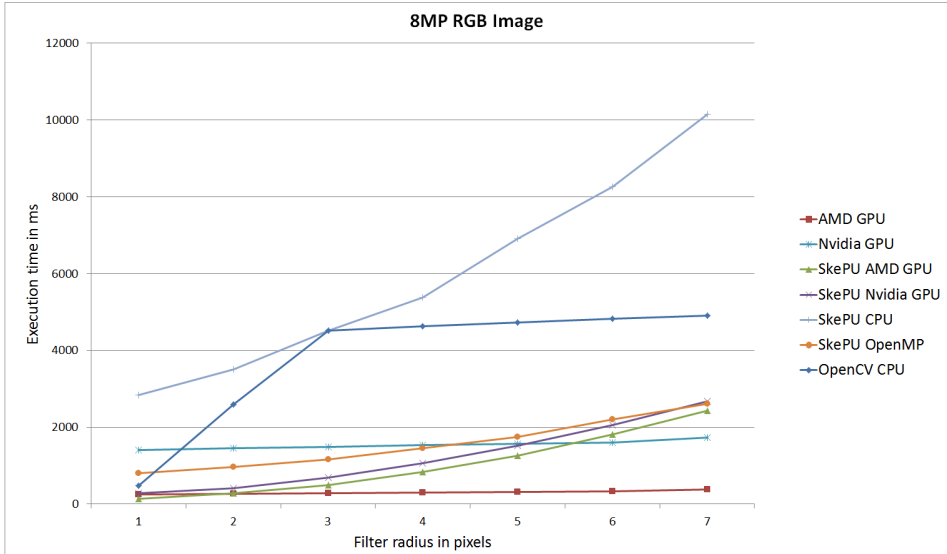


Figure 3.11: Comparison of the scaling with filter radius between CPU implementation and GPU implementation. Execution time is including operations to move the data to and from the GPU.

The CPU implementation is the Constant-Time Median Filtering implementation found in OpenCV 2.4.10. One thing to note about the OpenCV implementation is that sorting networks are used for filters where the filter radius is less than or equal to 2. That is the reason for the high slope of the curve in Figure 3.11 at those radius values. These were still included to show that the GPU implementation was faster than those too. We see that the AMD GPU implementation is the fastest while the single-threaded SkePU implementation is the slowest. The difference between the Nvidia GPU and the AMD GPU has many reasons. The AMD GPU has more processing power overall and has a newer version of OpenCL. AMD also focuses more on optimizing their OpenCL drivers while Nvidia has more focus on CUDA. We see that the SkePU implementations start out as the fastest but scales the worst with increasing filter size. The SkePU OpenMP implementation performs well, just slightly worse than the SkePU GPU implementations. It seems like the SkePU OpenMP implementation actually scales better.

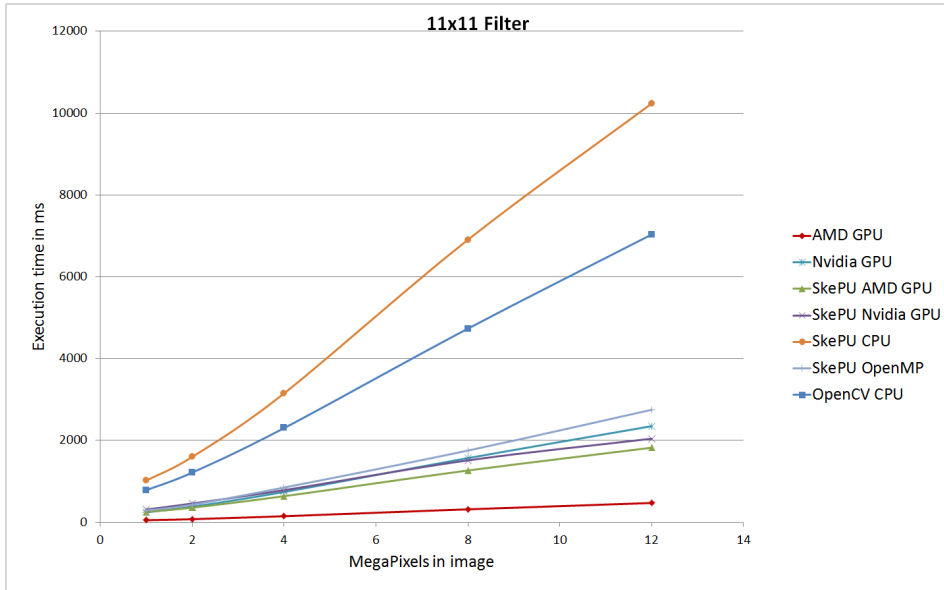


Figure 3.12: Comparison of the scaling with image size between CPU implementation and GPU implementation. Execution time is including operations to move the data to and from the GPU.

Figure 3.12 shows us the scaling with regard to image size. We see that the AMD GPU implementation is the fastest. The SkePU implementations, besides the singlethreaded CPU one, performs really well when considering how simple the implementation is. Overall, the AMD GPU implementation is the one that scales the best with both filter size and image size.

### 3.6.4 Stability and scalability of GPU implementation

As seen in Figure 3.11 and in Figure 3.12, the stability of the implementation at higher values is good. There are some things that limits the scalability of this implementation though.

There is a theoretical limit on the scalability of the algorithm, as the filter needs to fit inside one tile. This means that the limits calculated in Section 3.4.3 apply on the filter size. For example, on the developer machine, this sets a filter size limit of  $125 * 125$ .

### 3.6.5 Portability

There are some parameters that limit portability.

As mentioned in the previous section, the tile size will limit the size of the filter. So depending on the size of the local memory of the GPU, the maximum filter size will change. This can be calculated using Equation 3.8.

### 3.6.6 Ease of programming

The final kernel ended up being 162 lines of code. The only algorithmic changes between Constant-Time Median Filtering and the GPU implementation is that the GPU implementation is data parallel and that it uses CCDFs instead of histograms.

## 3.7 Challenges and benefits of porting this algorithm to OpenCL

### 3.7.1 Challenges

Not much was needed to change about the basic algorithm. CCDFs were used instead of histograms and data parallelizations were done both on the image and the CCDFs. The biggest challenges were in the implementation details.

One example is indexing. Indexing was easy using Image2D as it was accessed using a sampler. But using buffers made the indexing more challenging.

The buffer implementation also needed 2 additional kernels. One was for arranging the data from the  $r_0, g_0, b_0, r_1, g_1, b_1, \dots, r_{n-1}, g_{n-1}, b_{n-1}$  format into the  $r_0, r_1, \dots, r_{n-1}, g_0, g_1, \dots, g_{n-1}, b_0, b_1, b_{n-1}$  format. This was needed to be able to access the global memory in a more coalesced manner. Another kernel was needed to reverse the process after the filtering was done.

Debugging is also harder on the GPU. With so many threads running on so many different cores, running a normal debugger is not feasible. There are GPU debuggers but they are primitive in comparison with CPU debuggers. This means that debugging is mostly done by writing from the GPU into buffers and reading the buffers on the CPU after the kernels have run.

### 3.7.2 Benefits

The biggest benefit is the speed increase. Using hardware at similar costs the GPU implementation maintains a higher throughput of pixels. The kernels can also be run on all platforms supporting OpenCL which also means that it can be run on CPUs. The development of new hardware moves more and more into more parallel hardware which means that the GPU implementation will scale better with future hardware.

# Chapter 4

## RSA

### 4.1 Algorithm description

Following is a description of the general RSA algorithm and the usual optimizations done. Notice that these are also used in sequential implementations and are not specific for the parallel one.

#### 4.1.1 General description

RSA is an asymmetric cryptography algorithm that is often used in conjunction with a symmetric cryptography algorithm for secure data transmission. The name comes from the initials of the surnames of Rivest, Shamir and Adleman who published the algorithm 1977. RSA consists of a public key called  $e$ , a private key called  $d$  and a modulus called  $n$ . The modulus  $n$  is a product of two primes. Given a message transformed into an integer  $m$  which fulfills the constraint  $m < n$ , the encrypted message  $c$  is calculated using Equation 4.1.

$$c = m^e \bmod n \quad (4.1)$$

To decrypt the cipher text  $c$  an identical equation is used where the private key is used instead of the public key, see Equation 4.2:

$$m = c^d \bmod n \quad (4.2)$$

A big drawback of RSA is that modular exponentiation is an expensive operation as the  $m$ ,  $c$ ,  $e$  and  $d$  are  $k$ -bit integers where normal values for  $k$  are 1024, 2048 or 4096 bits. This leads to a large number of exponentiations if a naive implementation is used.

#### 4.1.2 Algorithm usage

RSA is mainly used to encrypt the keys which are used for symmetric encryption. Symmetric encryption provides much higher security while also being faster to

encrypt and decrypt. Asymmetric encryption is used to encrypt the key before sending it to the recipient. Let us say that Alice wants to send a message to Bob securely over the internet. The flow would look like this:

1. Alice initiates contact with Bob and informs him that she wants to communicate securely.
2. Bob generates a RSA key pair and sends his public key to Alice.
3. Alice encrypts her message using a symmetric algorithm.
4. Alice encrypts the symmetric key with Bob's public key.
5. Alice sends the encrypted data along with the encrypted symmetric key to Bob.
6. Bob decrypts the symmetric key using his private key.
7. Bob decrypts the message using the symmetric key.

We will call one symmetric key being encrypted and decrypted in this way a *block* of input. So the data being encrypted and decrypted using RSA will already be divided into blocks, where each symmetric key will be one block. That is why we will talk about blocks of input instead of bytes of input in the rest of this thesis. We will also focus more on decrypting than encrypting. Both are done in the same way, but the largest common exponent used for encryption is usually 65537, while the exponent used for decryption will have as many bits as the RSA keys have. This means that encryption is far cheaper than decryption. It also means that this optimization is most needed at computers which create and receive a lot of connections using small messages. A prime example of this would be servers using the SSL protocol.

### 4.1.3 Key generation

According to the PKCS # 1 v2.2 Cryptography Standard [25] the public key consists of the following parts:

- $n$  - the RSA modulus, a positive integer
- $e$  - the RSA public key, a positive integer

The private key can be given in two different forms, where the first form is:

- $n$  - the RSA modulus, a positive integer
- $d$  - the RSA private key, a positive integer

The second form includes more parts which are used to speed up the calculations. It also includes support for multi-prime keys, where the modulus is a product of more than two primes. This is not used often and will not be mentioned here, as the implementation done in this work does not support multi-prime keys. This form is as follows:

- $p$  - the first factor, a positive integer, prime
- $q$  - the second factor, a positive integer, prime
- $dP$  - the first factor's CRT exponent, a positive integer
- $dQ$  - the second factor's CRT exponent, a positive integer
- $qInv$  - the CRT coefficient, a positive integer

CRT stands for Chinese Remainder Theorem, and is one of the optimizations that will be explained in this paper. The process of generating a key is as follows.

1. Choose two different prime numbers  $p$  and  $q$ .
2. Compute  $n = p \cdot q$ . The number of bits in this number decides the length of the key.
3. Compute the totient  $\varphi(n) = (p - 1) \cdot (q - 1)$
4. Choose a number  $1 < e < \varphi(n)$  that is coprime to  $\varphi(n)$ . Choosing a prime makes it easier to check this.
5. Compute  $d \equiv e^{-1} \pmod{\varphi(n)}$

$e$  is often chosen to be 3, 17 or 65537 depending on key length. These are primes, have a low Hamming Weight and a short bit length which makes the encryption more efficient. In the binary case, the Hamming Weight of an integer is the number of bits that have the value 1 in the binary representation of the integer. This together with the short bit length are important factors as modular exponentiation often is done using the square-and-multiply algorithm which will be described later.

## 4.2 General optimizations done

Calculating the modular exponentiation is an expensive operation. One of the reasons is the modulo operation. We need to calculate the remainder after a division. This means that we will do a trial division to get the quotient which we will then use to calculate the remainder. The integers involved are also large, typically between 1024 and 4096 bits. Reducing the size of these numbers and increasing the speed of the modular exponentiation is necessary to use RSA in real time applications.

Decrypting is a more expensive process than encrypting. The reason for this is because the exponent  $e$  in the public key is usually not larger than 65537 while the exponent  $d$  in the private key can be as large as the key size. So for a key size of 1024 bits the maximum value of  $d$  is  $2^{1024} - 1$ . It is because of this that decrypting is much more expensive than encrypting. We will primarily discuss the decrypting aspect in the following chapters just because of this reason. All optimizations but the Chinese Remainder Theorem optimization will be applicable

for both encrypting and decrypting though.

There are a few optimizations that are done in almost all RSA implementations, both sequential and parallel ones. We will here go through the process of using these to calculate a modular exponentiation. We will later in the thesis go through the steps in a more detailed way.

Say that we want to decrypt the cipher  $c$  by calculating  $m = c^d \bmod n$ . The process is as follows:

1. We implement and use a multiprecision system. This is because the integers used in the modular exponentiation are too large for normal data types which usually only support a maximum of 64 bits.
2. We then reduce the size of  $d$  and  $n$  from  $k$  bits to  $k/2$  bits by using the Chinese Remainder Theorem together with Fermat's little theorem. This also splits the modular exponentiation into two modular exponentiations which gives us more parallelism.
3. We reduce the amount of modular multiplications to do by using the square-and-multiply method. With this we can decrease the maximum amount of modular multiplications from  $2^k - 1$  down to  $k$  where  $k$  is the number of bits in the integers.
4. We reduce the calculation cost of each modular multiplication. This is done by using Montgomery reduction. Montgomery reduction removes the need of doing a trial division when doing the modulo operation by converting the integers to montgomerized form where we can choose which modulus we want to use. We choose a modulus which is a power of two which means that all modulo and division operations will be just bitmasking and bitshifting.

We will now go through each of these steps in more detail.

#### 4.2.1 Multiprecision system

The integers used in these calculations are larger than 1024 bits which means that we cannot represent them using normal program integers. Instead we use a multiprecision system where the only limit is the memory of the GPU.

A multiprecision system is defined as a string of numbers  $d_1, d_2 \dots d_n$  that denotes the decimal number  $d_1b^{n-1} + d_2b^{n-2} \dots d_nb^0$  where  $0 \leq d_i < b$  and the only limit for  $n$  is the memory of the host system. The last part is the important one which makes it a multiprecision system.

We want to use the largest base possible in our implementation to use the capabilities of the GPU in the most efficient manner. The largest possible integer in OpenCL has a size of 64 bits so the largest base we can use is  $2^{64}$  and it is the base which we will use.

#### 4.2.2 Chinese Remainder Theorem

The terms  $d$  and  $n$  in Equation 4.2 contribute a lot to the complexity of the encryption and decryption. Reducing the size of these two variables would give us

a lot in terms of speed.

The Chinese Remainder Theorem [16] does this. It is a way to break up a modulo operation into multiple modulo operations with smaller numbers. The more factors of the modulus we have, the more parts we can break the operation into. In this thesis we only use two factors of the modulus which means that we will only split the modulo operations into two. Therefore the following explanation will only use the special case of having two factors. A more indepth description and a description of using more than two factors can be found in Johann Großschädl's paper on using the Chinese Remainder Theorem in RSA [10].

The general process to will be as follows:

1. We use the two factors  $p$  and  $q$  to split the modular exponentiation. Instead of doing a modular exponentiation using  $n$ , we will do two separate modular exponentiations using  $p$  and  $q$ .
2. We will also reduce the exponent  $d$  by applying Fermat's little theorem [16] on the two resulting modular exponentiations.
3. We will solve the two modular exponentiations separately.
4. We will then use the two results to calculate the final answer.

The character  $k$  will be used to denote the number of bits in one block of input, which is the same as the number of bits in the public and private keys.

This theorem assumes that we have the factors of the modulus  $n$  which are  $p$  and  $q$  in the case of the second form of the private key seen before. This also means that it can only be used while decrypting as we do not know the factors of the modulus when doing public encryption. The theorem is a way of solving equations of the form given in Equation 4.3.

$$\begin{cases} x \equiv a_1 \pmod{n_1} \\ x \equiv a_2 \pmod{n_2} \end{cases} \quad (4.3)$$

We will use this to rewrite Equation 4.2 into Equation 4.4.

$$\begin{cases} m \equiv c^d \pmod{p} \\ m \equiv c^d \pmod{q} \end{cases} \quad (4.4)$$

Notice that we now calculate  $c \pmod{p}$  and  $c \pmod{q}$  instead of  $c \pmod{n}$ , as  $n = p \cdot q$ . This means that we can do modulo calculations in  $k/2$  bits instead of  $k$  bits which will make the calculations a lot faster.

We are now going to create a running example, to illustrate the steps. Say that we want to calculate  $1569862^{1197377} \pmod{2639387}$ . We know the factors of the modulus, which are  $p = 1693$  and  $q = 1559$ . We can then rewrite our problem as:

$$\begin{cases} m \equiv 1569862^{1197377} \pmod{1693} \\ m \equiv 1569862^{1197377} \pmod{1559} \end{cases} \quad (4.5)$$

Before doing anything more here, let us look into what the three terms  $dP$ ,  $dQ$  and  $qInv$  in the second private key form are. The mathematical definition is given below.

- $dP = d \bmod (p - 1)$
- $dQ = d \bmod (q - 1)$
- $qInv = q^{-1} \bmod p$

For our running example this would mean:

- $dP = d \bmod (p - 1) = 1133$
- $dQ = d \bmod (q - 1) = 833$
- $qInv = q^{-1} \bmod p = 518$

These values will be used in the optimizations done below.

We have shown how to reduce the size of the modulus  $n$  but we also want to reduce the size of the exponent  $d$ . This can be done by also using Fermat's little theorem which states that given a prime  $z$  and an integer  $a$ , the number  $a^z - a$  is an integer multiple of  $z$ . This can also be written as  $a^z \equiv a \bmod z$ .

Something that follows from this given that the positive integers  $x$  and  $y$  fulfill  $x \equiv y \bmod (z - 1)$  is that for every integer  $a$  we have  $a^x \equiv a^y \bmod z$ .

So given that we have  $dP = d \bmod (p - 1)$ , we can rewrite  $(c^d \bmod p)$  to  $(c^{dP} \bmod p)$ . The same can be done for the  $q$  factor.

$$\begin{cases} m \equiv c^{dP} \bmod p \\ m \equiv c^{dQ} \bmod q \end{cases} \quad (4.6)$$

This is the biggest gain from using CRT. We managed to reduce the exponents from a maximum of  $k$  bits down to a maximum  $k/2$  bits. We can also do these two  $k/2$  modular exponentiations in parallel, which is especially good for our GPU implementation.

In our example this would mean:

$$\begin{cases} m \equiv 1569862^{1133} \bmod 1693 \\ m \equiv 1569862^{833} \bmod 1559 \end{cases} \quad (4.7)$$

And if we calculate the two expressions in the system above we get:

$$\begin{cases} m \equiv 826 \bmod 1693 \\ m \equiv 1217 \bmod 1559 \end{cases} \quad (4.8)$$

So after having calculated both modular exponentiations in Equation 4.6 we will need to obtain  $m$ .

Let us first show the formal definition. We want to solve for  $x$  in an equation in the form given in Equation 4.3. First let  $[a^{-1}]_b$  denote the multiplicative inverse

of  $a \bmod b$  given by the Extended Euclidian algorithm [16]. The solution to the system given in Equation 4.3 is then given in Equation 4.9.

$$x \equiv a_1 n_2 [n_2^{-1}]_{n_1} + a_2 n_1 [n_1^{-1}]_{n_2} \quad (4.9)$$

Another way to write this in the terms we have is as follows. Let us set  $m_p = c^{dP} \bmod p$  and  $m_q = c^{dQ} \bmod q$ . We can then obtain  $m$  by using Algorithm 12.

---

**Algorithm 12** Calculate  $m$  given  $m_1$  and  $m_2$ .

---

**Input:**  $m_p, m_q, p, q, qInv$ .

**Output:**  $m$

$h = (m_p - m_q) \cdot qInv \bmod p$

$m = m_q + q \cdot h$

**return**  $m$

---

In the case of our example, this gives us  $h = ((826 - 1217) \cdot 518) \bmod 1693 = 622$  which will give us  $m = 1217 + 1559 \cdot 622 = 970915$  which is the answer to  $1569862^{1197377} \bmod 2639387$ .

### 4.2.3 Square-and-multiply

Doing modular exponentiation is an expensive operation, and as  $d$  can be as large as the key length, for example 2048 bits, doing modular exponentiation the naive way is unfeasible. The standard way of doing it is by using the square-and-multiply algorithm instead. See Knuth [14] for a more in-depth description. The square-and-multiply algorithm uses two following mathematical properties to speed up the calculation:

1.  $x^m = \begin{cases} x \cdot (x^2)^{\frac{m-1}{2}}, & \text{if } m \text{ is odd} \\ (x^2)^{\frac{m}{2}}, & \text{if } m \text{ is even} \end{cases}$
2.  $(a \cdot b) \bmod c = ((a \bmod c) \cdot (b \bmod c)) \bmod c$

Let us say we want to calculate  $5^9 \bmod 11$ . We could do this the naive way which would lead to 8 modular multiplications. Or we could first rewrite it using the first attribute above:

$$(5^2 \cdot 5^2 \cdot 5^2 \cdot 5^2 \cdot 5^1) \bmod 11$$

This would mean that we only need five multiplications. One to calculate  $5^2$  and the other four to multiply the factors together. We would also only need one modulo operation. We can do even better by rewriting the expression to :

$$(5^4 \cdot 5^4 \cdot 5^1) \bmod 11$$

Now we only need four multiplications. Two to calculate  $5^4$  and two to multiply the factors together. We also need to do a modulo operation.

It can also be written as:

$$(5^8 \cdot 5^1) \bmod 11$$

This would give us the same amount of operations. The problem with the above is that we do modulo arithmetic. The multiprecision system has an upper limit and will overflow if the integers are too big. The multiplications would overflow before the modulo operation was done. Therefore we need to use the second attribute mentioned above to rewrite the expression a little more:

$$((5^8 \bmod 11) \cdot 5^1) \bmod 11$$

We would then need to precalculate:

$$5^8 \bmod 11 = ((5^4 \bmod 11) \cdot (5^4 \bmod 11) \bmod 11)$$

This means that we need to precalculate:

$$5^4 \bmod 11 = ((5^2 \bmod 11) \cdot (5^2 \bmod 11) \bmod 11)$$

This also means that we need to precalculate:

$$5^2 \bmod 11 = (5 \cdot 5 \bmod 11)$$

Each precalculation is one modular multiplication which means that we end up doing a total of four modular multiplications instead of the 8 that we mentioned in the beginning. Given in an algorithmic way it would be done as follows:

1.  $x = 5$
2. Calculate  $x_2 = (x \cdot x) \bmod 11$
3. Calculate  $x_4 = (x_2 \cdot x_2) \bmod 11$
4. Calculate  $x_8 = (x_4 \cdot x_4) \bmod 11$
5. Calculate  $5^9 \bmod 11 = (x \cdot x_8) \bmod 11$

Another way to look at this is to look at the binary form of 9 which is  $1001_2$ . Compare this to what we calculated above, but written in the following form:

$$((1 \cdot 5^8) \cdot (0 \cdot 5^4) \cdot (0 \cdot 5^2) \cdot (1 \cdot 5^1)) \bmod 11$$

Each bit in the binary form of 9 is combined with the exponent of the value of the position of the bit to form the above expression. An algorithmic way of writing this algorithm can be seen in Algorithm 13.

The expected number of modular multiplications needed using this method is given by  $\frac{3j}{2}$  where  $j$  is the amount of bits in the integers. This is a drastic reduction from the naive method. For example in the case of 2048 bits the worst case for the square-and-multiply algorithm is  $2 \cdot j = 4096$  modular multiplications. The worst case for the naive algorithm would be  $2^{4096} - 1$  modular multiplications. Also, as long as the exponents have the same amount of bits, the values of them do not change the execution time that much. This gives the algorithm stable time behaviour such that we can predict more accurately how long time it will take to encrypt and decrypt. The reason that we wanted a low amount of bits and a low Hamming Weight of the exponent  $e$  is because of this algorithm. A low number of bits mean that we will do less modular multiplications. And we do one modular multiplication for each bit set to 1 in the exponent, so a low Hamming Weight means less modular multiplications.

**Algorithm 13** Calculate  $g^e \bmod m$ **Input:**  $g$ ,  $m$  and a positive integer  $e = (e_j, e_{j-1} \dots e_1, e_0)_2$ .

---

```

 $A = 1$ 
for  $i = j$  down to 0 do
     $A = A^2 \bmod m$ 
    if  $e_i = 1$  then
         $A = A \cdot g \bmod m$ 
    end if
end for
return  $A$ 

```

---

**4.2.4 Montgomery reduction**

Modular multiplication is a very expensive operation as described before. We have reduced the size of the exponent  $d$  and the modulus  $n$  by using the Chinese Remainder Theorem. We have also reduced the number of modular multiplications needed in the modular exponentiation by using the square-and-multiply method. We still have not solved the problem that the modular multiplication in itself is expensive. As said before, the modulo operations requires a trial division which we want to avoid.

Montgomery reduction [16] is a way to avoid this trial division. It is used to calculate large modular multiplications in an efficient manner. It uses a certain attribute of the modulo operation to accomplish this. Say that we have an integer in base 10, for example 7523. Then:

1.  $7523 \bmod 1000 = 523$
2.  $7523 \bmod 100 = 23$
3.  $7523 \bmod 10 = 3$

If the modulus is an integer of the form  $10^x$  where  $x$  is a positive integer, then the modulus operation becomes a mask operation. We can just mask away everything but the  $x$  rightmost digits.

The same can be done in base 2. Say that we have 214 which is  $[11010110]_2$ . Calculating  $214 \bmod 16 = 214 \bmod 2^4$  just means bitmasking away everything but the 4 rightmost bits. This is much cheaper than doing a trial division. It is also something that takes constant time on the GPU, where we can parallelize this operation using one thread for each word in the integer. The problem is how to rewrite a modular multiplication so that we can change the modulus to an integer in the form  $2^x$ . Note that we use  $2^{64}$  as a base in our implementation as mentioned in Section 4.2.1, so we actually want an integer in the form  $2^{64^x}$ . The modulus we choose can be expressed both in the form  $2^x$  and in the form  $2^{64^x}$ . The explanation will continue with the assumption that we use base 2.

The general process of applying Montgomery reduction is as follows:

1. Decide which modulus we want to use. In our case this is a modulus which can be written in the form  $2^x$ .

2. Convert the involved integers to montgomerized form. This is the form where we will carry out our montgomery reduction and Equation 4.10 details how the conversion is done.
3. Use a special algorithm to calculate the modular multiplication. Note that we can also do modular exponentiation here by repeating the modular multiplication.
4. Convert the result back to normal form.

Say we want to calculate  $c = a \cdot b \bmod m$ . First we need to pick an integer  $R$ . This will be our new modulus used when doing calculations in the montgomerized form, which means that we want it to be an integer of the form  $2^x$ . The only requirements are that  $R$  and  $m$  are coprime and that  $R$  is larger than our current modulus  $m$ . In other words,  $\gcd(R, m) = 1$  and  $R > m$ . In our case the first requirement is easy. We know that  $m$  is a prime. Also,  $R$  is a power of two and  $m$  will be uneven as it is a prime. The second requirement can also be fulfilled easily. If we assume that all calculations are done using integers that are  $k$  bit in size, then we just need to choose an integer which uses more than  $k$  bits.  $R = 2^k$  is a good choice. It fulfills the requirements and is an integer in the form  $2^x$  which we wanted. We will now convert our input  $a$  and  $b$  into montgomerized form. This can be done using Equation 4.10.

$$\bar{a} = a \cdot R \bmod m \text{ where } \gcd(R, m) = 1, m < R \quad (4.10)$$

We will call  $\bar{a}$  above the montgomerized form of  $a$ . So now we have our input  $a$  and  $b$  in the montgomerized forms  $\bar{a}$  and  $\bar{b}$ . Sadly the next step involves more steps than just a modulo operation. We will compute the modular multiplication by calculating  $\bar{a} \cdot \bar{b} \cdot R^{-1} \bmod m$ . We must first calculate two additional values,  $R^{-1}$  and  $m'$ .

- $R^{-1}$  is defined by the equation  $R \cdot R^{-1} \equiv 1 \bmod m$  and is calculated with the Extended Euclidian algorithm.
- $m'$  is defined by the equation  $R \cdot R^{-1} - m \cdot m' = 1$  and can easily be calculated as  $R$ ,  $m$  and  $R^{-1}$  are known.

Now we can calculate  $\bar{a} \cdot \bar{b} \cdot R^{-1} \bmod m$  by using Algorithm 14.

The good thing about this is that the most expensive operations in Algorithm 14, the division and modulo, is done by  $R$ . So for the modulo operation we will just mask away all but the  $k$  rightmost bits, while in the division we will just shift the integer  $k$  bits to the right. The rest is multiplications, additions and subtractions which are cheaper than the trial division that we would otherwise need.

Now we have the result in montgomerized form, let us call it  $\bar{c}$ . We need to turn this back into our normal form using Equation 4.11.

$$c = \bar{c} \cdot R^{-1} \bmod m \text{ where } \gcd(R, m) = 1, m < R \quad (4.11)$$

**Algorithm 14** Montgomery reduction

---

**Input:**  $\bar{a}, \bar{b}$  $R^{-1}$  such that  $R \cdot R^{-1} \equiv 1 \pmod{m}$  $m'$  such that  $R \cdot R^{-1} - m \cdot m' = 1$ **Output:**  $\bar{a} \cdot \bar{b} \cdot R^{-1} \pmod{m}$  $T = \bar{a} \cdot \bar{b}$  $M = T \cdot m' \pmod{R}$  $U = (T + M \cdot m)/R$ **if**  $U \geq m$  **then****return**  $U - m$ **else****return**  $U$ **end if**

---

We now have our result  $c = a \cdot b \pmod{m}$ . Note that we need a few values to use Algorithm 14 namely  $R^{-1}$  and  $m'$ . Both of them can be precomputed once for each private key, which means that it is a low overhead when using this technique. And as said before, if we choose  $R = 2^k$  where  $k$  is the number of bits of the private key parts  $p$  and  $q$ , then all divisions and modulo operations in Algorithm 14 can be done using bit masking and bit shifting.

We still have a problem though. We need to do modular multiplications with the modulus  $m$  when converting to and from the montgomerized form, see Equation 4.10 and Equation 4.11. These can also be done using Montgomery reduction. We write the Montgomery reduction as function  $Mont(\bar{a}, \bar{b}) = \bar{a} \cdot \bar{b} \cdot R^{-1} \pmod{m}$ . This function will be calculated using Algorithm 14 so that the modulo operation with regards to  $m$  will never be carried out. It then follows that:

$$Mont(a, R^2) = a \cdot R^2 \cdot R^{-1} \pmod{m} = a \cdot R \pmod{m} = \bar{a}$$

This is equal to Equation 4.10. So to calculate  $\bar{a}$  we just calculate  $\bar{a} = Mont(a, R^2)$ . In this way we avoid the trial division by  $m$ .  $R^2$  can be precalculated once for each private key. And because of the rules of modular multiplication we can actually calculate  $\bar{R} = R^2 \pmod{m}$  and use that instead of  $R^2$ , so  $\bar{a} = Mont(a, \bar{R})$ . Similarly, we can go from montgomerized form using Montgomery reduction by calculating:

$$Mont(\bar{c}, 1) = \bar{c} \cdot 1 \cdot R^{-1} \pmod{m} = \bar{c} \cdot R^{-1} \pmod{m} = c$$

This is equal to Equation 4.11. So  $c = Mont(\bar{c}, 1)$ .

Note that because we have used the Chinese Remainder Theorem to split our modular exponentiation into two modular exponentiations we have two modulus,  $p$  and  $q$ . This means that we get two  $\bar{R}$  values,  $\bar{R}_p$  and  $\bar{R}_q$ .

Say for example that we want to calculate  $372 * 385 \pmod{919}$  in base 10. We want to skip doing expensive modulo operations and would rather prefer to use Algorithm 14. The process would be as follows:

1. We need a new modulus  $R$ . We choose the number 1000 as that means all modulo and division operations will only be masking and shifting. It also fulfills the requirements  $R > m$  and  $\gcd(R, m) = 1$ .
2. We calculate  $\bar{R} = R^2 \bmod m = 1000^2 \bmod 919 = 128$ . We also calculate  $R^{-1} = 295$  and  $n' = 321$ .
3. We then convert the two multiplication factors  $a = 372$  and  $b = 385$  to montgomerized form.  $\bar{a} = \text{Mont}(a, \bar{R}) = 724$  and  $\bar{b} = \text{Mont}(b, \bar{R}) = 858$ .
4. We calculate  $\bar{c} = \text{Mont}(\bar{a}, \bar{b}) = 283$  using the Montgomery reduction in Algorithm 14.
5. We convert  $\bar{c} = 283$  back to normal form by calculating  $c = \text{Mont}(\bar{c}, 1) = 775$  which is the result of  $372 * 385 \bmod 919$ .

The next step is to incorporate the Montgomery reduction into the square-and-multiply algorithm where we are doing the modular multiplications. Let us declare the function  $\text{MontReduce}(\bar{a}, \bar{b}, m)$  which does Montgomery reduction with the inputs  $\bar{a}, \bar{b}$  and the modulus  $m$  as seen in Equation 14. We assume that the rest of the needed values are precomputed and known by the function. Incorporating Montgomery reduction into the square-and-multiply algorithm given in Section 4.2.3 above we get Algorithm 15.

---

**Algorithm 15** Calculate  $g^e \bmod m$

---

**Input:**  $g, m, R^2 \bmod m$  and a positive integer  $e = (e_j, e_{j-1} \dots e_1, e_0)_2$ .

$\bar{A} = \text{MontReduce}(1, R^2 \bmod m, m)$

$\bar{G} = \text{MontReduce}(g, R^2 \bmod m, m)$

**for**  $i = j$  **down to** 0 **do**

$\bar{A} = \text{MontReduce}(\bar{A}, \bar{A}, m)$

**if**  $e_i = 1$  **then**

$\bar{A} = \text{MontReduce}(\bar{A}, \bar{G}, m)$

**end if**

**end for**

$A = \text{MontReduce}(\bar{A}, 1, m)$

**return**  $A$

---

In general it is the same as the square-and-multiply defined at Algorithm 13. The differences are as follows:

- We begin by converting the number 1 and our input  $g$  to montgomerized form which gives us  $\bar{A}$  and  $\bar{G}$ . The rest of the calculations are done in the montgomerized form.
- We then do a Montgomery reduction to calculate  $A^2$  in montgomerized form.
- Every time we find a bit which is set to 1 we calculate the  $A \cdot g$  in the montgomerized form using Montgomery reduction.

- We end the algorithm by converting our montgomerized result  $\bar{A}$  back into the normal form which gives us the result  $A$ .

So first we transform the data to montgomerized form. We then run the algorithm but replace all modular multiplications with Montgomery reduction. Then we transform the data back to our normal form before returning it.

The main thing to note is that when calculating a modular exponentiation using Montgomery reduction we only need to convert the input to montgomerized form once. We also only need to convert the result back to normal form once.

### 4.2.5 Barret reduction

So can we use Montgomery reduction for all the modular operations we do? The answer is yes, but that it is not always worth it. The reason is that it is expensive to convert back and forth between normal form and the montgomerized form. So when it is only a simple modular multiplication and not a modular exponentiation it is usually not worth it.

Two examples can be found in the CRT calculations given in Section 4.2.2. In Equation 4.6,  $c$  is an integer of  $k$  bits, while  $p$  and  $dP$  have  $k/2$  bits. We want all the integers in those equations to be  $k/2$  bits in size.

We solve this by calculating  $c \bmod p$  and  $c \bmod q$  first. The result from these operations will be an integer of  $k/2$  bits. We then use the square-and-multiply algorithm when all values are of  $k/2$  bits. We could do these modulo operations using Montgomery reduction, but transforming the values to montgomerized form, calculating the value and transforming it back takes longer time than doing a normal modulo operation.

Another example is in Algorithm 12. We do a single modular multiplication with  $p$  which is more efficient to do using normal modulo computation because of the same reasons as given above.

An algorithm called Barret reduction[16] is implemented for this modulo operation, see Algorithm 16. It calculates  $x \bmod m$  where  $x$  has the double amount of bits as  $m$  which is exactly our situation in the case of  $c \bmod p$  and  $c \bmod q$ .

---

**Algorithm 16** Calculate  $x \bmod m$

---

**Input:** Positive integers  $x = (x_{2k-1}, \dots, x_1, x_0)_b, m = (m_{k-1} \dots m_1, m_0)$  and  $\mu = \lfloor b^{2k}/m \rfloor$ .  
 $q_1 = \lfloor x/b^{k-1} \rfloor, q_2 = q_1 \cdot \mu, q_3 = \lfloor q_2/b^{k+1} \rfloor$   
 $r_1 = x \bmod b^{k+1}, r_2 = q_3 \cdot m \bmod b^{k+1}, r = r_1 - r_2$   
**if**  $r < 0$  **then**  
     $r = r + b^{k+1}$   
**end if**  
**while**  $r \geq m$  **do**  
     $r = r - m$   
**end while**  
**return**  $r$

---

The while loop at the end runs a maximum of 2 iterations since  $0 \leq r < 3m$ .

### 4.3 General parallelizations done

Many of the algorithms below use a variable called *id*. This is the id of the current thread. This is something that is supplied by the OpenCL framework and does not need to be calculated or given as an argument.

We will also use the term *word*. One word is one part of the multiprecision system. So if we use a binary system each bit would be one word and if we use a system in base 10 each word would be one digit. In our case each word is a number in the range  $[0, 2^{64} - 1]$  as our system uses a base of  $2^{64}$ .

The term *block* will also be used. The input to RSA is usually symmetric keys. These come in different sizes such as 1024, 2048 or 4096 bits. These keys are usually referred to as blocks. So one block means one key in our case.

We will now describe the general parallelizations done.

1. The first parallelization is done across blocks. As described above, RSA is used to encrypt and decrypt symmetric keys mostly. This means that the input is already divided into blocks of 1024, 2048 or 4096 bits. So the first parallelization is simply launching these blocks at the same time using multiple threads. We will launch one thread for each word in each block.
2. The second parallelization is done per block. We will split each block into two sub-blocks using CRT. These sub-blocks will be decrypted in parallel using multiple threads. The threads will be split in half where each half decrypts one sub-block.
3. The third parallelization is done per sub-block. Each sub-block has multiple threads which can be used to decrypt the sub-block. We use these threads by implementing parallel versions of the operators used in the algorithms in Section 4.2. These parallel implementations usually works by having each thread be responsible for one word. So we have parallel implementations of addition, subtraction, multiplication and so on which do calculations using the multiprecision system that we have defined.

Let us go through an example with the parallelizations done so far. We will use the second form of the private key shown in Section 4.1.3. We have a group of  $l$  blocks that we want to decrypt, let us call them  $c_1, c_2 \dots c_l$ . Each of these are 4096 bits in size. This means that our private key components  $n$  and  $d$  are also 4096 bits in size. The two modulus factors  $p$  and  $q$  will then be 2048 bits in size. The two CRT exponents  $dP$  and  $dQ$  will also be 2048 bits as well as the CRT coefficient  $qInv$ .

1. We will first parallelize these  $l$  blocks on data level. All blocks will be run in parallel. So we calculate how many words we will need in base  $2^{64}$  for each block. The blocks are 4096 bits in size and each word is 64 bits, this means that we need 64 words per block. We want to use one thread per word so we launch a total of  $64l$  threads in parallel. Each block is handled similarly so we will only follow one of them through the process. The block we will follow will be  $c_1$ .

2. We will begin by splitting the block into sub blocks using CRT which will give us two different modular exponentiations to solve:  $m_{1p} = c_1^{dP} \bmod p$  and  $m_{1q} = c_1^{dQ} \bmod q$ . Let us also split the 64 threads we have for this block into two parts;  $t_{1p}$  and  $t_{1q}$ . The 32 threads in  $t_{1p}$  are responsible for calculating  $m_{1p}$  while the 32 threads in  $t_{1q}$  are responsible for calculating  $m_{1q}$ . All the integers in these equations are 2048 bits in size except  $c_1$  which is 4096 bits in size. We would like all of the integers to be 2048 bits. That would mean that we would only need to use a multiprecision system with 32 instead of 64 words in the rest of the calculations. Therefore we let the threads in  $t_{1p}$  use Barret reduction to calculate  $c_{1p} = c_1 \bmod p$ . The threads in  $t_{1q}$  will do the same to calculate  $c_{1q} = c_1 \bmod q$ . We can now rewrite the two modular exponentiations to  $m_{1p} = c_{1p}^{dP} \bmod p$  and  $m_{1q} = c_{1q}^{dQ} \bmod q$ . All the integers are now 2048 bits in size and we can continue.
3. The threads in  $t_{1p}$  will use the square-and-multiply algorithm together with Montgomery reduction as seen in Algorithm 15 to solve the equation  $m_{1p} = c_{1p}^{dP} \bmod p$ . The threads in  $t_{1q}$  will do the same to solve the expression  $m_{1q} = c_{1q}^{dQ} \bmod q$ .
4. We now have the two parts  $m_{1p}$  and  $m_{1q}$  of the final decrypted message  $m_1$ . The threads in  $t_p$  will use Algorithm 12 to join these two parts together to form the final decrypted message  $m$ .

Note that in the last step we only use the threads in  $t_p$ . This means that half the threads will be idle. This is not efficient. The way we solve it is to use a separate kernel for this last step which only uses half the amount of threads for each block. We will write more about this in a later chapter.

As seen in the example above we have multiple threads available when decrypting each sub block. We had 32 threads to use when calculating  $m_{1p}$  and 32 threads to use when calculating  $m_{1q}$ . We use these threads by parallelizing the operators used in the CRT, square-and-multiply, Barret reduction, and Montgomery reduction calculations.

The rest of the chapter will describe how we parallelize the most used operators in these calculation. Most of these implementations are parallelized versions of the operations found in the Handbook of Applied Cryptography [16].

### 4.3.1 Add carries and overflows

Most of the algorithms for arithmetic needs to be able to handle carries or overflows of words. For example the multiplication  $6 \cdot 8 = 48$ . But suppose that our multiprecision system only had one word in base 10 to use. The result of the multiplication would then be 8, with an overflow of 4. To get the correct result we need to calculate this overflow and add it to the next word in the system. These overflows are generally referred to as carries when used in this way.

There are OpenCL functions which calculate and return the overflow from additions and multiplications. These will be referred to as *carryAdd* and *carryMultiply* in the pseudocode. Both overflows and carries will henceforth be referred to as

carries to simplify the explanations.

The carries are used to build a carry array. The result of an addition or multiplication between two  $n$  word integers will maximally be  $2n$  words long. The carry array will therefore also be  $2n$  in size. This carry array will then be added to the result in the end of the operation, where the result will also be  $2n$  words long. This is done often, so the function will be presented here and then only referred to when it is used. Note that the adding of carries can overflow, this means that the multiprecision system was too small for the calculation and this situation needs to be detected. See Algorithm 17 for the function definition.

---

**Algorithm 17** Add carries  $c$  to  $x$  and output the result in  $result$ .

---

```

function ADDCARRIES( $c, x, result$ )
  Input: Positive integers  $c = (c_{2n}, \dots, c_1, c_0)_{2^{64}}$ ,  $x = (x_{2n-1}, \dots, x_1, x_0)_{2^{64}}$ 
    and  $id = \text{id of thread}$ 
  Output: True if the addition did not overflow, false if it did overflow.
     $successFlag = 0$ 
    while  $c! = 0$  do
       $overflowVar = \text{carryAdd}(c[id + n], x[id + n])$ 
       $result[id + n] = c[id + n] + x[id + n]$ 
       $c[id + n] = 0$ 
       $c[id + n - 1] = c[id + n - 1] + overflowVar$ 
       $overflowVar = \text{carryAdd}(c[id], x[id])$ 
       $result[id] = c[id] + x[id]$ 
       $c[id] = 0$ 
      if  $id == 0$  then
         $successFlag = 1 - overflowVar$ 
      else
         $c[id - 1] = c[id - 1] + overflowVar$ 
      end if
    end while
    return  $successFlag$ 
end function

```

---

### 4.3.2 Addition

Parallel addition in its core is to let every thread take care of the addition of one pair of words each. So the first thread adds the first word of both blocks together, the second adds the second word of both blocks together and so on. The carries generated when doing these word additions then needs to be added to the result. We implemented this as a three step process. First all words are added in parallel using the *carryAdd* operation which returns the carry from the addition. These are used to build the carry array. Then the words are added to each other ignoring carries. The carries are then added to the result using the *AddCarries* function defined at Algorithm 17. See Algorithm 18 for pseudocode of the addition.

---

**Algorithm 18** Calculate  $x + y$  and output the result in *result*.

---

**function** ADD( $x, y, result$ )

**Input:** Positive integers  $x = (x_{n-1}, \dots, x_1, x_0)_{2^{64}}$ ,

 $y = (y_{n-1}, \dots, y_1, y_0)_{2^{64}}$  and  $id = \text{id of thread}$ 
**Output:** True if the addition did not overflow, false if it did overflow.

 $overflow[id + n - 1] = carryAdd(x[id], y[id])$ 
 $tempResult[id] = x[id] + y[id]$ 
**return** AddCarries( $tempResult, overflow, \text{out } result$ )

**end function**


---

### 4.3.3 Subtraction

Subtraction is very similar to addition. The only difference is that we get borrows instead of carries. Subtraction is also done in a three-step process. First we calculate if any borrowing is needed by comparing the size of the words. We use this information to build a borrow array. We then borrow where needed and subtract accordingly. We then need to subtract the borrow array from our result. This algorithm only handles unsigned data, which means that it can underflow. This can be detected if the most significant word needs to borrow. See Algorithm 19 for pseudocode.

There are some cases where the subtraction will underflow. Those cases only appear in situations where we are doing modular arithmetic though. So if we calculate  $(x - y) \bmod p$  and the subtraction underflows we can handle it by instead calculating  $(p - (0 - (x - y))) \bmod p$ .

### 4.3.4 Multiplication

Multiplication is the most complicated of the operations. It is best explained by looking at Figure 4.1 while reading.

It is a variant of long multiplication. We multiply all words of the first integer with all words of the second to create two matrices. One with the preliminary result, called loword, and one with the overflow, called highword. Each thread is responsible for one column in the loword matrix and one column in the highword matrix. These threads are responsible for adding their respective columns to the final result.

A carry array is calculated while adding. This will be sent together with the intermediate result to the AddCarries function described in Algorithm 17. The result of that operation is then returned. Note that the multiplication cannot overflow, the maximum size of the result when multiplying two  $n$  word integers is  $2n$ . Pseudocode is given at Algorithm 20. The operation called *carryMultiply* is used to retrieve the carry, also known as the highword result of each multiplication. This implementation is inspired by the multiplication description that can be found in the SSLShader [11].

---

**Algorithm 19** Calculate  $x - y$  and output the result in *result*.

---

```

function SUBTRACT( $x, y, result$ )
Input: Positive integers  $x = (x_{n-1}, \dots, x_1, x_0)_{2^{64}}$ ,
 $y = (y_{n-1}, \dots, y_1, y_0)_{2^{64}}$  and  $id$  = id of thread
Output: True if the subtraction did not underflow, false if it did underflow.
   $tempResult = |x[id] - y[id]|$ 
  if  $y[id] > x[id]$  then
     $borrows[id + n - 1] = 1$ 
     $tempResult = 2^{64} - tempResult$ 
  end if
   $result[id] = tempResult$ 
  while  $borrows \neq 0$  do
    if  $result[id] == 0$  and  $borrows[id] == 1$  then
       $didBorrow = 1$ 
       $result[id] = 2^{64} - 1$ 
    else
       $didBorrow = 0$ 
       $result[id] = result[id] - borrows[id]$ 
    end if
    if  $id == 0$  and  $didBorrow \neq 0$  then
      return false
    end if
  end while
  return true
end function

```

---

Lowword		Intermediate result	Carries
3 2 9		0 0 0 0 0 0	0 0 0 0 0 0
5 5 0 5	→	0 0 0 5 0 5	0 0 0 0 0 0
3 9 6 7	→	0 0 9 1 7 5	0 0 1 0 0 0
4 2 8 6	→	0 2 7 7 7 5	0 1 1 0 0 0
Highword			
3 2 9		0 2 7 7 7 5	0 1 1 0 0 0
5 1 1 4	→	0 2 8 8 1 5	0 1 1 1 0 0
3 0 0 2	→	0 2 8 0 1 5	0 1 2 1 0 0
4 1 0 3	→	1 2 1 0 1 5	0 2 2 1 0 0

Figure 4.1: Calculation of  $329 * 435 = 143115$  in base 10. Adding of carries is not included.

Note that the algorithm described at Algorithm 20 requires a lot of memory to store the lowword and highword matrices. The optimized implementation uses a single loop and interleaves all calculations to avoid this memory cost.

#### 4.3.5 Division by $R$

The only division we need to do is to divide by  $R$  as we are using Montgomery reduction. We know that the value of  $R$  is  $2^k$  where  $k$  is the number of bits in the CRT factors. This can be done by simply shifting the integer  $n$  words to the right. This can be seen in the pseudocode given at Algorithm 21. Note that the input is an integer with  $2k$  bits.

#### 4.3.6 Modulo by $R$

Another operation that is always done by  $R$ , if we are not using Barret reduction, is the modulo operation. It is simply done by masking the input so that the  $n$  highest words are set to 0. See pseudocode at Algorithm 22. Note that the input is an integer with  $2n$  words.

---

**Algorithm 20** Calculate  $x \cdot y$ .

---

```

function MULTIPLY( $x, y$ )
Input: Positive integers  $x = (x_{n-1}, \dots, x_1, x_0)_{2^{64}}$ ,
 $y = (y_{n-1}, \dots, y_1, y_0)_{2^{64}}$  and  $id = \text{id of thread}$ 
Output: The result  $result = (result_{2k-1}, \dots, result_1, result_0)_{2^{64}}$ 
  for  $i = n - 1$  down to 0 do
     $lowWord[i] = x[n - 1 - i] * y[id]$ 
     $highWord[i] = carryMultiply([n - 1 - i], y[id])$ 
  end for
  for  $i = n$  down to 0 do
     $carries[i + id - 1] = carries[i + id - 1] +$ 
       $carryAdd(result[i + id], lowWord[n - i - 1])$ 
     $tempResult[i + id] = tempResult[i + id] + lowWord[n - i]$ 
  end for
  for  $i = n - 1$  down to 0 do
     $carries[i + id - 1] = carries[i + id - 1] +$ 
       $carryAdd(result[i + id], highWord[n - i - 1])$ 
     $tempResult[i + id] = tempResult[i + id] + highWord[n - i]$ 
  end for
  if  $id \neq 0$  then
     $carries[id - 1] = carries[id - 1] +$ 
       $carryAdd(result[id], lowWord[n - 1])$ 
  end if
   $tempResult[id] = tempResult[id] + highWord[n - 1]$ 
  AddCarries( $tempResult, carries, \text{out } result$ )
  return  $result$ 
end function

```

---



---

**Algorithm 21** Calculate  $x/R$ .

---

```

function DIVIDEBYR( $x$ )
Input: positive integers  $x = (x_{2n-1}, \dots, x_1, x_0)_{2^{64}}$  and  $id = \text{id of thread}$ 
Output: The result  $result = (result_{n-1}, \dots, result_1, result_0)_{2^{64}}$ 
   $result[id] = 0$ 
   $result[id + n] = x[id]$ 
  return  $result$ 
end function

```

---



---

**Algorithm 22** Calculate  $x \bmod R$ .

---

```

function MODULOBYR( $x$ )
Input: positive integers  $x = (x_{2n-1}, \dots, x_1, x_0)_{2^{64}}$  and  $id = \text{id of thread}$ 
Output: The result  $result = (result_{n-1}, \dots, result_1, result_0)_{2^{64}}$ 
   $result[id] = 0$ 
   $result[id + n] = x[id + n]$ 
  return  $result$ 
end function

```

---

## 4.4 GPU specific parallelizations and optimizations

### 4.4.1 Multiprecision operations

All multiprecision operations are done in parallel where each thread is usually responsible for one word. This means that the time taken does not increase that much with the number of words in the integers. Instead the number of needed threads will increase. More threads per block means that less blocks will be calculated at the same time in the GPU, as each block needs more resources. The time needed for each block will only increase slightly because of the additional thread synchronizing needed.

### 4.4.2 Kernel handling

There are two different kernels being used. The first one is responsible for the modular exponentiation. It will do the following:

1. Transfer all needed data from global to local memory.
2. Split the block using CRT.
3. Do modular exponentiation on the sub-blocks.
4. Write the split result to global memory.

The second kernel is responsible for joining the split result together again:

1. Transfer the split result to local memory.
2. Join it together using Algorithm 12.
3. Write the result to global memory.

The calculations are now done, and the result is stored in the GPU global memory. There are multiple reasons for using two kernels instead of one:

- The modular exponentiation kernel uses one thread for each word in the block. This means that the number of threads used, given number of blocks  $i$  and number of words per block  $n$ , is  $i \cdot n$ . The second kernel uses half the amount of threads per block. So the number of threads used by the second kernel is  $i \cdot n/2$ .
- The kernel would need to read more data into local memory, which is bad when we want to maximize the kernel occupancy.
- The thread divergence would be higher. In the square-and-multiply algorithm the thread divergence is decided by the exponent used. If the whole workgroup uses the same modulus and exponent the thread divergence will be non-existent. This is optimal and would not be the case if we only used one kernel.

- Less kernel complexity. Each kernel has less input and less code, which makes it easier to test and optimize.

One drawback of the multiple kernel approach is that there will be more global memory accesses. First the first kernel reads and writes to global memory. Then the second kernel also needs to read and write to global memory. It will still be faster though, mainly because of the different amount of threads launched and thread divergence.

#### 4.4.3 Memory handling

As much calculations as possible are carried out on private memory. The intermediate results are stored in the private memory for as long as possible before they are transferred to the local memory. Global memory is only accessed at the start and end of each kernel and always in a coalesced manner.

#### 4.4.4 Workgroup optimization

OpenCL executes threads in wavefronts of size 64. This means that we always want the workgroups to be multiples of 64. This is achieved by adjusting the number of blocks that each workgroup will process.

We also want to minimize local memory storage and thread divergence. This is achieved by letting all threads in a workgroup work with the same modulus. The input block will be divided into two factors using the Chinese Remainder Theorem, one part which will be calculated modulo  $p$ , and one which will be calculated modulo  $q$ . Each workgroup will only use one of these modulus in their calculations. An example of these principles is as follows:

We want to decrypt 4 blocks of input which have a key size of 2048 bits. The number of threads launched using the modular exponentiation kernel will be 128. These will be divided into two workgroups of 64 threads. Let us call these workgroup 0 and workgroup 1. Each workgroup calculates 4 modular exponentiation in parallel using 16 threads for each one. Workgroup 0 will calculate the first CRT factor, while workgroup 1 will calculate the second. So the calculations will be carried out as follows:

- Thread 0-15 in workgroup 0 will calculate the first CRT factor of block 0.
- Thread 0-15 in workgroup 1 will calculate the second CRT factor of block 0.
- Thread 16-31 in workgroup 0 will calculate the first CRT factor of block 1.
- Thread 16-31 in workgroup 1 will calculate the second CRT factor of block 1.
- Thread 32-47 in workgroup 0 will calculate the first CRT factor of block 2.

- Thread 32-47 in workgroup 1 will calculate the second CRT factor of block 2.
- Thread 48-63 in workgroup 0 will calculate the first CRT factor of block 3.
- Thread 48-63 in workgroup 1 will calculate the second CRT factor of block 3.

Using this layout means that the threads in each workgroup will have as little thread divergence as possible.

The workgroup layout for the second kernel is different. This will also be padded so that each workgroup uses at least 64 threads. But we only need half the amount of threads as the modular exponentiation kernel. We will launch 64 threads in only one workgroup where this workgroup is responsible for all 4 input blocks. It will look as follows:

- Thread 0-15 will join together the CRT factors of block 0.
- Thread 16-31 will join together the CRT factors of block 1.
- Thread 32-47 will join together the CRT factors of block 2.
- Thread 48-63 will join together the CRT factors of block 3.

## 4.5 Experimental Evaluation

### 4.5.1 CPU vs GPU implementation

The most interesting values for key length today are 1024, 2048 and 3072 bit. As of NIST Recommendations [24], a key size of 1024 is not safe after 2010. A key size of 2048 is recommended and deemed secure until 2030 where a key size of 3072 bits is needed to be secure. Larger key sizes are projected to be needed after that. The following data will focus on 2048 bit and 4096 bit encryption. 2048 bit because it is the key size deemed secure right now, and 4096 bit to show how it may hold up in the future. There are still implementations that use 1024 bit as the key size though, so we will include data on that too.

The CPU implementation tested is from the OpenSSL [27] library. It is a widely used library that is deemed to be among the fastest and also placed overall fastest in a comparison done by Bingmann [3].

The AMD GPU used in these tests is an ASUS Radeon R9 280x-DC2T-3GD5 with 32 Compute Units running at 970 MHz. The CPU in this computer is an Intel i3570k and the amount of RAM is 8 GB. The OpenCL version used is 1.2.

The Nvidia GPU is a Tesla M2050 with 14 Compute Units running at 1150 MHz. This computer has 16 processors, each being an Intel Xeon E5520 running at 2.27 GHz, and the amount of RAM is 24 GB. The OpenCL version used is 1.1.

The CPU used in the testing is an Intel i3570k with 4 physical cores running at 3.4 GHz and the amount of RAM is 8 GB. Two implementations were tested using

the OpenSSL library, one running on a single core, and one running on all 4 cores using OpenMP.

The compiler used was g++ 4.8.1 with the O2 and msse2 flags. No extra OpenCL flags were used when compiling the kernels.

All values include both the time taken to send the data to the GPU and the time taken to retrieve the result.

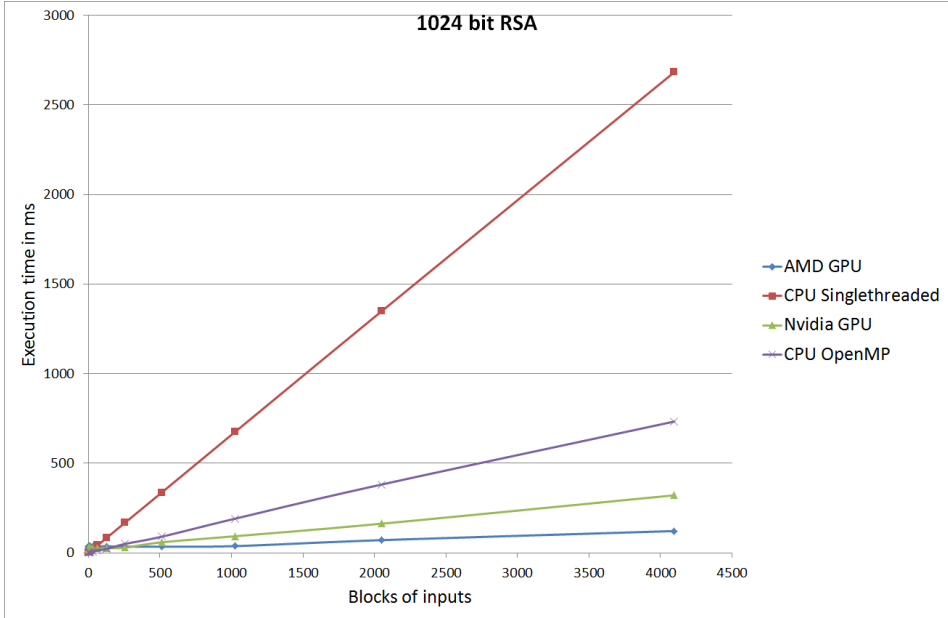


Figure 4.2: Decryption using a 1024 bit key.

We see in Figure 4.2 that the single threaded CPU solution is the slowest, which is no real surprise. The OpenMP implementation scales much better, but not as good as the GPU solutions. The Nvidia GPU is faster than the AMD GPU at input sizes which are lower than 512 which can be attributed to its higher clock speed while the AMD GPU is faster at larger input sizes which can be attributed to its higher amount of Compute Units. The CPU implementations are faster at lower inputs. For input sizes below 256, the CPU OpenMP implementation is the fastest one. Above that the GPUs are faster. Another important part is the latency. The latency is the time it takes to get the first result back, which is the same as the amount of time it takes to calculate one input.

We can see the latencies in Table 4.1a. These were measured by decrypting a single block of input. We see that the CPU implementations are far ahead on this part. On the other hand, the GPU implementations have higher throughput as seen in Table 4.1b. This suggests that a combined solution is the best, where different implementations are used based on the size of the input.

Latency in ms	
AMD GPU	36,01 ms
Nvidia GPU	19,76 ms
CPU Singlethreaded	0,8 ms
CPU OpenMP	0,8 ms

(a) Latency of 1024 bit decryption

Throughput in blocks/sec	
AMD GPU	33700,28 blocks/sec
Nvidia GPU	12758,26 blocks/sec
CPU Singlethreaded	1526,59 blocks/sec
CPU OpenMP	5588,757 blocks/sec

(b) Throughput of 1024 bit decryption

Table 4.1: Latency and throughput of 1024 bit decryption

Latency in ms	
AMD GPU	135,43 ms
Nvidia GPU	65,73 ms
CPU Singlethreaded	4,3 ms
CPU OpenMP	4,3 ms

(a) Latency of 2048 bit decryption

Throughput in blocks/sec	
AMD GPU	4473,54 blocks/sec
Nvidia GPU	2450,51 blocks/sec
CPU Singlethreaded	238,99 blocks/sec
CPU OpenMP	826,74 blocks/sec

(b) Throughput of 2048 bit decryption

Table 4.2: Latency and throughput of 2048 bit decryption

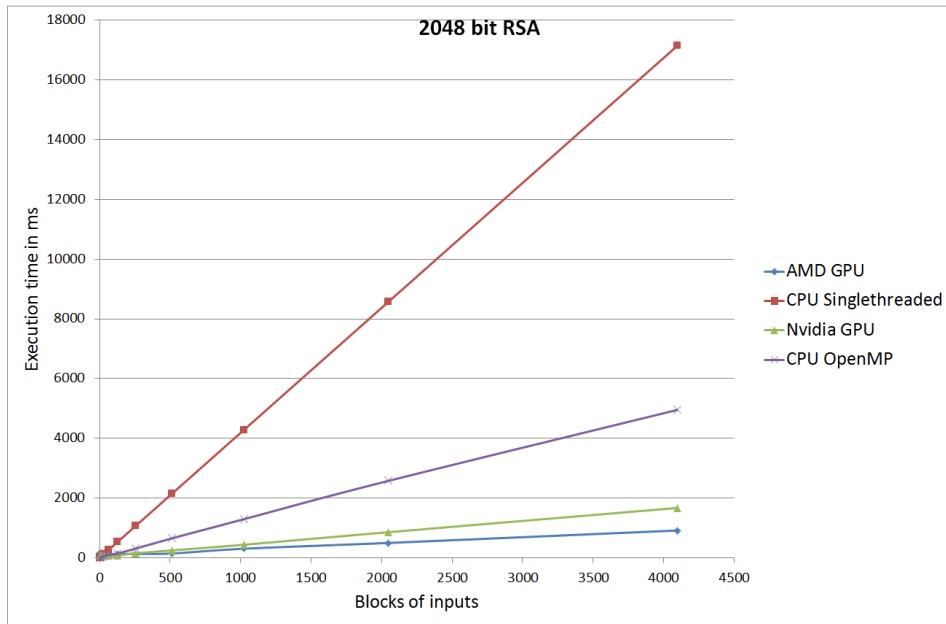


Figure 4.3: Decryption using a 2048 bit key.

We see the same pattern in Figure 4.3 which uses a key size of 2048 bits. The breakpoints are different though. The Nvidia GPU will be faster than the AMD GPU for input sizes below 256, and the CPU OpenMP implementation will be the fastest one for input sizes below 128.

Latency in ms		Throughput in blocks/sec	
AMD GPU	481,16 ms	AMD GPU	528,12 blocks/sec
Nvidia GPU	270,27 ms	Nvidia GPU	305,994 blocks/sec
CPU Singlethreaded	31,3 ms	CPU Singlethreaded	31,71 blocks/sec
CPU OpenMP	31,3 ms	CPU OpenMP	116,35 blocks/sec

(a) Latency of 4096 bit decryption

(b) Throughput of 4096 bit decryption

Table 4.3: Latency and throughput of 4096 bit decryption

The CPUs still have the lowest latency while the GPUs have the higher throughput, which still points to a solution using both platforms.

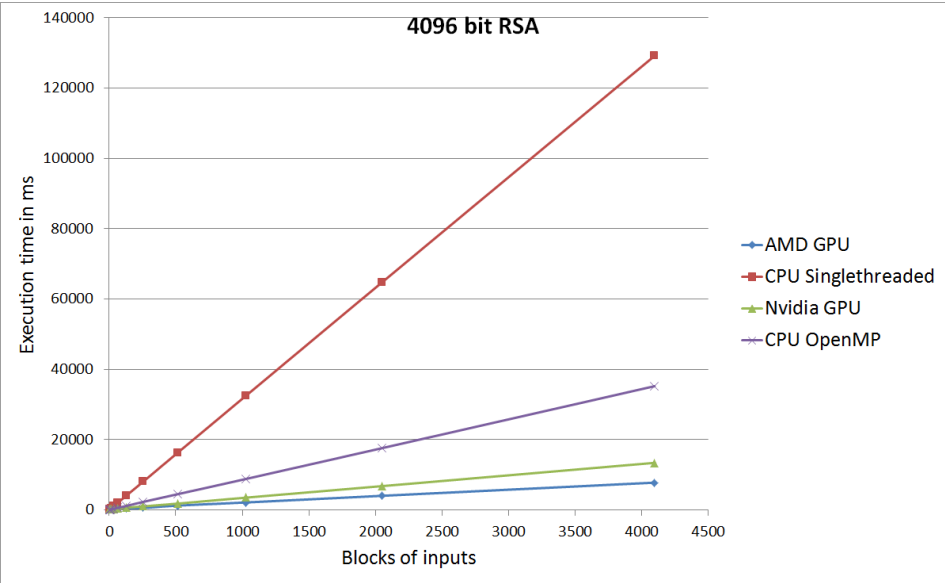


Figure 4.4: Decryption using a 4096 bit key.

Using a key size of 4096 bit gives us the result that we see in Figure 4.4. The Nvidia GPU will be faster than the AMD GPU for input sizes below 128, and the CPU OpenMP is the fastest one when the input size is below 64.

The latencies and throughput follows the same patterns as before. The reasons for the AMD GPU beating the Nvidia GPU in throughput are many. The major reason is probably that the AMD GPU is more powerful when comparing the number of compute units and the frequencies of these compute units. Another reason is that AMD takes OpenCL more into account when developing their drivers while Nvidia is more concerned with CUDA. A third is that CUDA uses a wavefront size of 32 threads instead of 64, which might mean that the Nvidia card is not as specialized in handling wavefronts of size 64 as the AMD is.

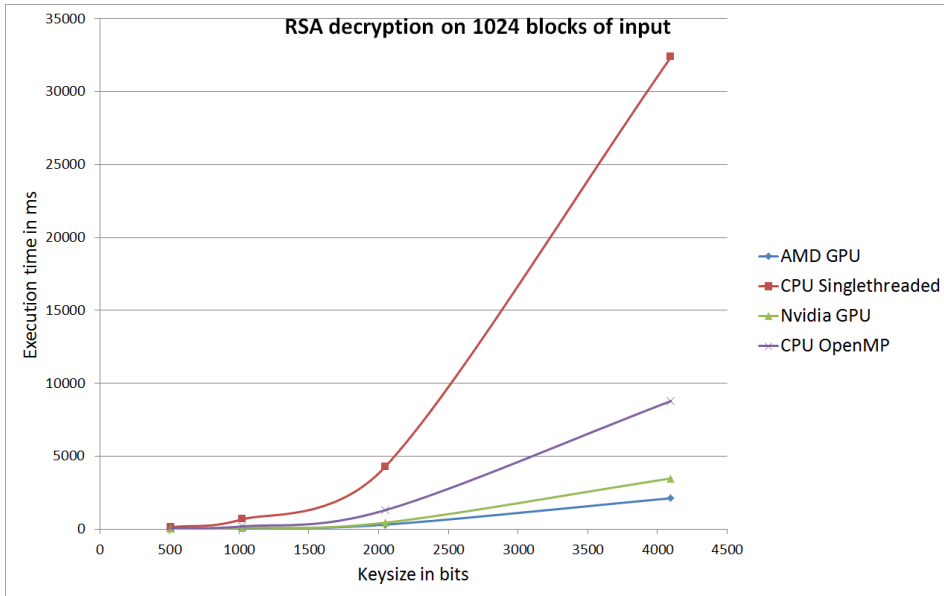


Figure 4.5: Decryption on 1024 blocks of input with different key sizes.

Figure 4.5 shows us the scaling of the implementations when the key size varies. Decryption was done on 1024 blocks of input, and the key size was varied between 512 and 4096 bits. We see that the GPU implementations have the best scaling, something we could also see in the values above.

### 4.5.2 GPU implementation vs SSLShader

As mentioned before, the fastest GPU implementation we could find of the RSA algorithm is in SSLShader [11]. The multiplication algorithm used in this thesis is made after the design described in the SSLShader paper. Their implementation is written in CUDA and the GPU they use in their testing is a Nvidia GTX580 with 16 Compute Units running at 1544 MHz. Their results will be compared against our implementation running on the AMD GPU used in this thesis.

The results in Table 4.4 and Table 4.5 show how much more optimization and

Key size	512	1024	2048	4096
SSLShader	1,1	3,8	13,83	52,46
Our impl.	13,06	36,01	135,43	481,17

Table 4.4: Latencies of decryption using different key sizes in ms.

work can be done on our implementation. The SSLShader is highly optimized,

Key size	512	1024	2048	4096
SSLShader	322167	74732	12044	1661
Our impl.	146664	33700	4473	528

Table 4.5: Throughput of decryption using different key sizes in blocks/sec.

down to instruction-level. There is a lot more that can be done, such as using a Constant Length Nonzero Window in the square-and-multiply algorithm which will reduce the amount of modular multiplications by taking advantage of subsequent zeros in the exponent, interleaving calculations based on domain knowledge, unrolling loops, minimizing divergence, minimizing local memory use, avoiding bank conflicts and so on. These optimizations are not as major as the ones done in this thesis, but they all add up to a faster implementation.

The graphs shown here focuses on the scaling with the number of input blocks and the scaling with key size. There is also a small variation in performance based on the individual keys. The most important factor is the Hamming Weight of the exponents in the modular exponentiation. A low Hamming Weight means that the square and multiply algorithm will do less modular multiplications. We did not see a large change in runtime when testing with different randomized keys, and all the comparisons between implementations in this chapter used the same keys.

### 4.5.3 Stability and scalability of the GPU implementation

As seen in the figures in Section 4.5.1 the scaling with regards to key size is good on the GPU. It scales better than the OpenMP CPU implementation does. There are some limits to this scaling though. The GPU has a global memory which is often smaller than the RAM of the CPU. This means that the GPU cannot work on as much data at the same time as the CPU does. This will probably not be a problem though. Using 4096 blocks of data, each being 4096 bits large, is about 2 megabyte. This is nothing compared to the memory size of GPUs today which is in the order of gigabytes.

This GPU implementation assumes that it is possible to use one thread for each word of the input, and that each CRT factor can be done by one workgroup each. The maximum workgroup size in OpenCL is 256 threads, which means that the largest block that one workgroup can process is 16384 bits.

There is also a limit in local memory. If we assume that we hit the upper limit of 256 threads in a workgroup the size of the local memory needed for the kernel using the most local memory would be around 26 kilobytes. This would be enough even for older GPUs which only have 36 kilobytes of local memory. So the hard limit would be at 256 threads meaning that the maximum key size this implementation can handle is 16384 bits.

#### 4.5.4 Portability

As mentioned in the previous section, the largest limit on portability would be the workgroup size of the GPU. The second limit would be the local memory that each Compute Unit has. A third limit would be the global memory of the GPU, but that will probably not be a problem.

#### 4.5.5 Ease of programming

This was more challenging than median filtering. The kernel was larger and the algorithms involved were more complicated. The final kernel size was 971 lines. This includes both kernels used and the functions they use.

One thing we did that helped a lot in the testing was to put the base used by the multiprecision system as a C macro in the kernel. This means that we could change from a base of  $2^8$  to a base of  $2^{64}$  easily. This is done by the preprocessor in C and therefore does not interfere with the optimizations the compiler can do. So the lowest base possible could be used for debugging while the base could be increased after the debugging was done.

It was also easy to debug in the sense that we could send in single values and see what output we got. We also implemented the RSA algorithm in C++ using the Boost library. This meant that we could test run single parts of the algorithm easily.

#### 4.5.6 Test conclusions

All data suggests that a hybrid solution would be the best. We mentioned earlier that servers communicating via the SSL protocol are a prime target for optimization. One way to do it would be to let the CPU handle RSA encryptions and decryptions until the queue of jobs has grown above a certain limit and then process the queue in parallel using a GPU solution. This would also offload the CPU, making it possible for the CPU to do other tasks while the GPU does the encryption or decryption.

### 4.6 Challenges and benefits of porting this algorithm to OpenCL

#### 4.6.1 Challenges

The biggest challenge was getting all the local memory management correct. Barriers are used to synchronize threads in the same workgroup when accessing local memory, but they will also slow down execution. So we want to use as few as possible without getting race problems. These problems can be hard to debug as they do not always appear and often give different results every time they are run. Another challenge was in understanding the algorithms used, and understanding

what assumptions could be done about the data to implement them in a smart way.

#### 4.6.2 Benefits

The benefits of this implementation is that it is faster when doing RSA operations on multiple blocks at the same time and that it offloads the CPU. The GPU implementation maintains a higher throughput of input blocks and scales better than the CPU implementation. The OpenCL kernel can also be run on all hardware that supports OpenCL.

# Chapter 5

## General insights about porting algorithms to GPU

In this chapter we will go through some general tips on parallelization and GPU programming based on our experiences during this thesis.

### 5.1 Memory handling

A lot of performance can be gained by thinking about which memory that should be used. Often the best choice is to use private memory where possible, then local memory, and lastly global memory. Global memory should be avoided as much as possible as the cost of reading and writing to it is large. The fastest memory is the private memory, so storing the variables in private memory before doing calculations is generally a good idea. Sharing data between threads within the same workgroup can be done using local memory which is a good way to avoid unnecessary global memory accesses. CodeXL [26] can be used to analyze what limits the kernel occupancy when the implementation is done. The report generated by CodeXL can be used to optimize the memory usage. CodeXL might also show that the most limiting factor is the registers. In that case it might make sense to actually move some calculations from private memory to local memory, but this needs to be decided by testing.

Global memory accesses can be sped up by reading and writing in a coalesced manner. It might be necessary to run additional kernels to rearrange the data for coalesced memory access. If this is beneficial or not also needs to be decided by testing.

The amount of local memory is limited and shared across all work-group running on the same Compute Unit. One of the most limiting factors for kernel occupancy in our case was this local memory limit. We managed to reduce this by analyzing the usage of local memory, removing and reusing as much as possible.

## 5.2 Choice of algorithm

The first question to ask before porting an algorithm is if it is parallelizable. Most algorithms are data parallelizable which can be enough to make GPU a good choice. Some algorithms are not worth parallelizing and works better on the CPU.

## 5.3 Debugging

Debugging on OpenCL is harder than debugging on a single-threaded CPU. There are a few ways to do it though.

### 5.3.1 Buffers

The easiest way is just to output an intermediary result to a buffer and read it on the CPU side. This can be tricky depending on how the indexing of the wanted elements is. It is easy to write the wrong value to the buffer and in that way get the wrong value on the CPU side, which is hard to detect. This is generally the best way to read a lot of output from the GPU, and it is also one of the easiest ways to debug a kernel.

### 5.3.2 Printing

It is possible to print from the kernel. The device that is running the code needs to have a *printf* extension which then needs to be enabled in the kernel. This will print for each thread which makes it hard to read in the case of a lot of threads, but it is a viable way to debug when using a low amount of threads.

### 5.3.3 CodeXL

CodeXL is development suite from AMD which includes a GPU debugger, a GPU profiler, a CPU profiler and a static OpenCL kernel analyzer. The debugger allows us to step through the kernel one instruction at a time while being able to see the private memory. Note that this only works on AMD GPUs. This is the most informative way of debugging an OpenCL kernel but it is also the slowest. The instructions will be slower when running the debugger and it can be hard to find the thread we want to observe when there are a large amount of threads. One drawback is that it is not possible to see the local memory when debugging using a GPU. It is possible if debugging with an AMD CPU though.

### 5.3.4 Macros

OpenCL C supports the `#DEFINE` command. This is great for debugging in some cases. An example is the RSA implementation done in this thesis. We could define the base of the multi-precision system to be of the type *unsigned char* while debugging. This made it a lot easier to read the output and follow along in the

calculations using pen and paper where needed. We then changed the define of the base to be *unsigned long* when running the final tests. This is also what the final implementation uses.

## 5.4 SkePU

As with most skeleton frameworks, it is really useful when there is a skeleton for the task you need but hard to use if there is no skeleton that fits well. An example is median filtering using the MapOverlap2D skeleton. It fits well at a glance, but it is missing edge handling. There is no way to get automatic edge handling and the kernel does not know its index which means that we cannot clamp the index either. The way this is solved is by padding. But there was no skeleton that suited padding in the way it needed to be done for median filtering which meant that the padding was implemented on CPU in the end. SkePU is also meant for simpler kernels which means that there is no way to create functions and call them from the kernel. This leads to big code if something more complex is to be done but works well for smaller implementations.

SkePU is great when implementing smaller kernels or combinations of kernels using different skeletons. The implementation is easy and it will run on OpenCL, CUDA, OpenMP and single-threaded CPUs.

## 5.5 General OpenCL tips

There is often a need to share local memory between different subgroups of threads within a work-group. This is solved using offsets. A problem is that local memory can only be allocated either in the host code before calling the kernel or in the kernel. It cannot be allocated in other functions in the device code. Sending both the local memory and the offset to each function will take a lot of arguments if there is a lot of local memory. But we can send pointers to local memory as arguments for functions. This gives us a way to remove the offsets from most of the function calls. The way to do this is to calculate the offsets and allocate the local memory in the kernel, and then call another function, with all the local memory pointers offset by the offsets, which then does all the calculation. This function will then have no idea about the offsets and can use the local memory pointers without any offsets.

Checking the error codes returned by the OpenCL API greatly helps debugging. Almost all calls to the OpenCL API return an error code. This can be read and converted to readable error. Missing to read one of these might make another API call later fail with a strange error, when the real error happened before that call but was not read. By checking the error codes from all OpenCL API calls we can avoid some strange bugs.

There are some cases where it is just not worth the time to parallelize, some tasks are faster on the CPU. Some typical examples are when the data size is small, when the algorithm is strictly sequential or when the overhead of transferring data to and from the GPU might be larger than the execution time. Some problems are

better solved in a sequential manner, while some are better solved in a parallel manner.

# Chapter 6

## Related Work

### 6.1 GPGPU

K. Komatsu and others made an evaluation of OpenCL in terms of performance and portability by comparing it to CUDA [15]. They raise some interesting points in what the differences in performance between OpenCL and CUDA could be caused by. They found that CUDA is in general faster given essentially identical kernels. They found that a large part is the difference in optimizations applied by the compilers when generating code, the CUDA C compiler generated more optimized code than the OpenCL C compiler even though the kernels were essentially identical.

While testing the portability they conclude that the work group size and optimization options needs to be tuned for each device which is similar to what we found when deciding the tile size on different devices for our Median filtering.

For a more in depth comparison of CUDA vs OpenCL the paper written by Fang and others [8] is a good read. They chose 16 benchmarks and found that CUDA was faster for most applications. By investigating further into the problem they find that this was mostly caused by the differences in the programming model, the architecture, optimizations done on the kernels and in the compiler. Their analysis shows that OpenCL can achieve similar performance as CUDA when taking these into account.

We made a comparison between our GPU implementation and an OpenMP implementation of RSA using OpenSSL. The code run was not the same, so the comparison was not the best. Shen and others made a comparison between OpenMP and OpenCL when run on the CPU which shows that OpenCL can also be made CPU friendly and is a good alternative to OpenMP [21]. By tuning the code for multi-core CPU usage and investigate the compiler flags OpenCL outperforms or achieve similar performance in 80% of their cases. Just as Komatsu and others found earlier, they also find that the OpenCL compiler is one of the main reasons for poor performance in OpenCL.

Enmyren and Kessler [7] is a recommended read for more information about SkePU. It contains more detailed information, benchmarks and examples.

## 6.2 Median filtering

The main related work in median filtering is the CUDA implementation of the CCDF median filter made by Sánchez and Rodríguez [20]. It shows a higher throughput than the implementation done in this thesis, 35-57 megapixels per second in comparison to the 20-33 megapixels per second our AMD implementation computes. The main implementation differences are in the threads used per tile and tile size. Their platform supports up to 512 threads per block, while ours only supports 256. This means that they can run 512 threads per tile instead of 256 which means that they can process two CCDFs in parallel in the same tile. They also use rectangular tiles instead which we also tried, but we got better performance using square tiles. It might also be because their implementation is more polished than ours.

Another interesting median filtering implementation on GPU is the one made by Perrot, Domas and Couturier [18]. The general principle in that implementation is to let each thread handle one pixel each. The important part of this algorithm is that the data is read into registers where all the calculation is done. This makes the calculations and memory accesses fast but limits the kernel size as the register count per thread is limited. For example, the maximum register count per thread for the Fermi architecture is 63. They use a special algorithm to reduce the amount of registers needed to improve the parallelism and make it possible to use larger filter sizes. They still have a lower limit on filter size than the CCDF implementation though.

Using a formula from their paper we can see that the largest filter that can be used on the Fermi architecture is  $11 \times 11$ . The throughput is impressive for filter sizes up to  $7 \times 7$  and it is by far the fastest for those sizes. It does not scale as good as the CCDF implementation with increasing filter size though and they only include data with filter sizes up to  $7 \times 7$ .

Median filtering is often used for de-noising in image processing which is usually done inside a camera's hardware. For this purpose  $3 \times 3$  filters are usually used to avoid the blurring effect of median filtering. Cheap, power efficient and fast hardware implementations of median filtering are therefore of interest. Most of the hardware implementations we have found uses sorting networks to do their filtering. This is the same method used by OpenCV when doing median filtering with a filter radius up to 7.

The fastest hardware implementation we could find was the one done by Abadi and others [1]. They use sorting networks specialized for the  $3 \times 3$  filter. They change the sorting network to first sort column wise, then row wise and lastly using these result to get the final value. The idea of sorting the columns first is somewhat similar to our way of doing the sorting with column CCDFs. They also read the image in a row by row structure, using as many hardware filters as there are pixels in a row. This is also similar to how we read the image by moving our CCDFs down one row at a time, using as many CCDFs as there are pixels in a row. They simulate their implementation using a Field Programmable Gate Array (FPGA) as the target device and median filters an image with 500 pixels per row in 6 clock cycles per row.

## 6.3 RSA

The work done by Jang and others on the SSLShader [11] was important while doing this RSA implementation. They have, as far as we know, the fastest RSA implementation on the GPU. Their implementation is done using CUDA instead of OpenCL and they have made more optimizations as mentioned in Section 4.5.2. A comparison between the SSLShader and our implementation can be read in Section 4.5.2.

Another way to speed up RSA calculations is to implement the RSA operations closer to the hardware. An example of this is the algorithm made by Song and others which uses FPGAs [22]. They attain high speeds using cheaper hardware and less power which can be important especially in mobile devices. It is not the fastest RSA implementation on hardware that we have found, but it is the most scalable and efficient when considering the hardware used. It can also be executed in parallel conveniently. The hardware implementation used in their chapter only uses Montgomery reduction and the square and multiply algorithm. They do not do any splitting of the input using CRT. They split the input integers into blocks of 17 bits before calculation. This can be seen as doing calculations using base  $2^{17}$ . In comparison, our GPU implementation uses a base of  $2^{64}$ .

Another important aspect of RSA is security of the hardware. There are attacks on RSA cryptosystems that attack the hardware structure instead of the algorithm itself. One type of attack is the Side Channel Attack (SDA). This is an attack where the architecture's hardware characteristics leakage is analyzed. By analyzing factors such as the power dissipation, computation time, electromagnetic emission information about the processed data can be extracted. This can be used to deduce the keys and messages. Fournaris and Koufopavlou [9] designed a hardware architecture and made modifications to the RSA algorithm to protect against such attacks by disassociating the leaked information from the secret data and minimizing the information leakage. Their algorithm uses CRT, Montgomery reduction and the square and multiply algorithm to speed up the encryption and decryption of messages. One modification they do is to introduce random numbers into the calculations. This randomness is later removed by using a special CRT reconstruction method. The hardware architecture they propose has protection against SDA attacks while maintaining efficiency both in speed and hardware size.



# Chapter 7

## Discussion

Multi-core GPUs have a lot of power that can be used for algorithms which can be easily parallelized, such as the median filter, and also for some algorithms which are not as easy as the RSA encryption. Data parallelization is usually possible if the input data are not dependent on each other but parallelizing the internal operations of the algorithm can be tricky. We can see that the median filtering benefits greatly from the massive parallelization of the GPU implementation. This is something that is often seen in filtering operations as they are easily parallelizable. In this case we would opt for a pure GPU solution to calculating the median and let the CPU handle other tasks.

In the case of the RSA algorithm we see that a pure GPU solution might not be the best. Latency is really important so that the CPU can start using the key to decrypt the symmetrical data. So the best approach in this case would probably be a hybrid solution where the CPU handles the decryption of the keys until the queue of keys is too large or the CPU is too busy to decrypt keys. Then the whole queue could be decrypted in parallel on a GPU.

SkePU is a good framework which works really well if the problem suits the skeletons that exists. It is particularly suited for solutions where the kernel is small and does not include any external functions. These types of frameworks make it easier to implement a parallel solution across multiple platforms but have the drawback of being harder to optimize. They offer good speedups considering the low adaptation overhead but we recommend using OpenCL or another such framework if it is important to achieve the highest speed possible.

Overall GPU solutions seem to be a good path forward when looking for more computing power. There are some things that need to be taken into consideration when choosing which algorithms to parallelize, but in many cases the algorithms can be parallelized on data level. In that case a GPU solution could be faster than CPU solutions given large datasets.



# Chapter 8

## Future work

There are still optimizations to do both on the median filter algorithm and the RSA algorithm. It would also be interesting to implement these in CUDA and compare the performance between them. There is also a new OpenCL specification which was released in November 2013. It has a lot of changes and it would be interesting to see if these change the implemented algorithms in some way. Solutions using multiple GPUs which run in parallel is also an interesting area which could speed up calculations where large datasets are used. SkePU already supports this, but not the OpenCL implementations that we did. It would be interesting to see if the datasets used for median filtering and RSA encryption are large enough to benefit from multiple GPUs.

It would also be useful to develop a formula to calculate the throughput of pixels and cryptographic blocks on a device given its specifications. This would mean that it would be easier to compare CPUs and GPUs without benchmarking them. Dastgeer and Kessler [5] discusses metrics which can be used to optimize the tile size and also calculate the tile size dynamically for different filter sizes. The properties of the GPU are also included in this calculation. This would be interesting to apply to our Median filtering implementation.



# Bibliography

- [1] H.Z.H. Abadi, S. Samavi and N. Karimi, Image Noise Reduction by Low Complexity Hardware Median Filter, 21st Iranian Conference on Electrical Engineering, pp 1-5, 2013.
- [2] L. Alparone, V. Cappellini, and A. Garzelli, A Coarse-to-Fine Algorithm for Fast Median Filtering of Image Data With a Huge Number of Levels, Signal Processing, vol. 39, no. 1-2, pp. 33-41, 1994.
- [3] T. Bingmann, Speedtest and Comparison of Open-Source Cryptography Libraries and Compiler Flags, <http://panthema.net/2008/0714-cryptography-speedtest-comparison/>, 2008.
- [4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, Third Edition, 2009
- [5] U. Dastgeer and C. Kessler, A performance-portable generic component for 2D convolution computations on GPU-based systems, Proc. MULTIPROG-2012 Workshop at HiPEAC-2012, 2012.
- [6] K. Dinkel and A. Zizzi, Fast Median Finding on Digital Images, University of Colorado at Boulder, Department of Aerospace Engineering Sciences, <http://kevindinkel.com/files/FastMedianFindingOnDigitalImages.pdf>, 2012.
- [7] J. Enmyren and C. Kessler, SkePU: A multi-backend skeleton programming library for multi-GPU systems, Proceedings of the fourth International Workshop on High-level Parallel Programming and Applications, pp 5-14, 2010.
- [8] J. Fang, A.L Varbanescu and H. Sips, A Comprehensive Performance Comparison of CUDA and OpenCL, International Conference on Parallel Processing, pp 216-225, 2011.
- [9] A.P Fournaris and O. Koufopavlou, CRT RSA Hardware Architecture with Fault and Simple Power Attack Countermeasures, 15th Euromicro Conference on Digital System Design, pp 661-667, 2012.
- [10] J. Großschädl, The Chinese Remainder Theorem and its Application in a High-speed RSA Crypto Chip, 16th Annual Conference, Computer Security Applications ACSAC '00, pp 384-393, 2000.

- [11] K. Jang, S.Han, S. Han, S. Moon and K. Park, SSLShader: cheap SSL acceleration with commodity processors, NSDI'11 Proceedings of the 8th USENIX conference on Networked systems design and implementation:1-1, 2011.
- [12] M. Juhola, J. Katajainen, and T. Raita , Comparison of algorithms for standard median filtering, Signal Processing, vol. 39, no. 1, pp. 204-208, 1991.
- [13] M. Harris, Optimizing Parallel Reduction in CUDA, [http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86/\\_website/projects/reduction/doc/reduction.pdf](http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86/_website/projects/reduction/doc/reduction.pdf), NVIDIA Developer Technology, 2007.
- [14] D. E. Knuth. The Art of Computer Programming, volume 2. Addison-Wesley, 3rd edition, 1997.
- [15] K. Komatsu, K. Sato, Y. Arai, K. Koyama, H. Takizawa and H. Kobayashi, Evaluating Performance and Portability of OpenCL Programs, In The Fifth International Workshop on Automatic Performance Tuning.
- [16] A. Menezes, P. van Oorschot, S. Vanstone, Handbook of Applied Cryptography, CRC Press, 1997.
- [17] S. Perreault and P. Hébert, Median Filtering in Constant Time, IEEE Transactions on Image Processing 16(9):2389-2394, 2007.
- [18] G. Perrot, S. Domas and R. Couturier, Fine-tuned High-speed Implementation of a GPU-based Median Filter, Journal of Signal Processing Systems 75(3):185-190, 2014.
- [19] T. Ryan, Modern Engineering Statistics, chapter 14, p. 468, Wiley-Interscience, 2007.
- [20] R. M. Sánchez and P. A. Rodríguez, Bidimensional median filter for parallel computing architectures, IEEE International Conference on Acoustics, Speech and Signal Processing, pp 1549-1552, 2012.
- [21] J. Shen, J. Fang, H. Sips A.L. Varbanescu, Performance Gaps between OpenMP and OpenCL for Multi-core CPUs, 41st International Conference on Parallel Processing Workshops, pp 116-125, 2012.
- [22] B. Song, K. Kawakami, K. Nakano and Y. Ito, An RSA Encryption Hardware Algorithm Using a Single DSP Block and a Single Block RAM on the FPGAB, Networking and Computing (ICNC), pp 140-147, 2010.
- [23] The OpenCL Specification Version 1.2, <https://www.khronos.org/registry/cl/specs/openc1-1.2.pdf>, Khronos OpenCL Working Group, 2012.
- [24] Recommendation for Key Management, Special Publication 800-57 Part 1 Rev. 3, NIST, 2012.

- [25] PKCS #1 v2.2: RSA Cryptography Standard, <http://www.emc.com/emc-plus/rsa-labs/pkcs/files/h11300-wp-pkcs-1v2-2-rsa-cryptography-standard.pdf>, RSA Laboratories, 2012.
- [26] <http://developer.amd.com/tools-and-sdks/opencl-zone/codex1/>, 2014.
- [27] <https://www.openssl.org/>, 2014.
- [28] <http://www.ida.liu.se/~chrke/skepu/>, 2014.



# Appendix A

## Median filter code

```
// Kernel responsible for rearranging the data from RGBRGBRG to
// RRRGGGBBB to give coalesced reads when filtering each channel.
__kernel void SplitChannels(__global uchar* inputBuffer, __global
uchar* outputBuffer, __private int numPixels){
    int globalIndex = get_global_id(0);
    int colorRead = globalIndex%3;
    int globalIndexOfSpecificColor = globalIndex/3;

    int outputIndex = colorRead*numPixels +
        globalIndexOfSpecificColor;

    outputBuffer[outputIndex] = inputBuffer[globalIndex];
}

// Kernel responsible for rearranging the data from RRRGGGBB to
// RGBRGBRGB after the filtering is done.
__kernel void JoinChannels(__global uchar* inputBuffer, __global
uchar* outputBuffer, __private int numPixels){
    int globalIndex = get_global_id(0);

    int colorRead = globalIndex/numPixels;
    int globalIndexOfSpecificColor = globalIndex%numPixels;

    int outputIndex = globalIndexOfSpecificColor*3 + colorRead;
    outputBuffer[outputIndex] = inputBuffer[globalIndex];
}

// Main median filter.
// NUM_BINS is a constant with the number of bins needed, 256
// normally.
__kernel void medianFilter(__global uchar* inputBuffer, __global
uchar* outputBuffer, __local uchar * columnsCCDFs, __global uchar
* resultBuffer, __private uchar kernelRadius, __private ushort
numColumns, __private ushort numRows, __private ushort numBlocksX
, __private ushort numBlocksY, __private int pictureWidth,
__private int pictureHeight, __private int bufferOffset){
```

```

// Create local memory. The local memory for the column CCDFs is
// created before starting the kernel as it differs with filter
// radius.
__local ushort kernelCCDF[NUM_BINS];
__local uchar tempArray[NUM_BINS];
uchar kernelSize = 2*kernelRadius +1;

// Calculate indices and some constants.
const size_t groupId = get_group_id(0);
const size_t blockIdY = groupId/numBlocksX;
const size_t blockIdX = groupId - blockIdY*numBlocksX;
const size_t startPixelX = blockIdX*numColumns;
const size_t startPixelY = blockIdY*numRows;
const size_t startRow = startPixelY -(kernelSize-1)/2;
const size_t numColsTimeNumBlocksX = numColumns*numBlocksX;
const size_t numCols2KernelRad = numColumns + 2*kernelRadius;

// Create the column CCDFs.
CreateColumnCCDFs(columnsCCDFs, inputBuffer, startRow, kernelSize
, startPixelX - kernelRadius, numCols2KernelRad, pictureWidth
, pictureHeight, tempArray, bufferOffset);
for (uint rowId = 0; rowId < numRows; ++rowId){
    // Initialize the kernel CCDF for every row.
    CreateKernelCCDF(kernelCCDF, columnsCCDFs, 0, kernelSize);
    for (uint columnId = 0; columnId < numColumns; ++columnId){
        barrier(CLK_LOCAL_MEM_FENCE);
        // Calculate the median for this pixel
        CalculateMedian(kernelCCDF, kernelSize, tempArray);
        if (get_local_id(0) == 0){
            uint coordX = startPixelX + columnId;
            uint coordY = startPixelY + rowId;
            outputBuffer[bufferOffset + coordX + coordY*
                numColsTimeNumBlocksX] = tempArray[0];
        }

        // Move to the next pixel by adding and removing CCDFs
        // from the kernel CCDF
        if (columnId < numColumns-1){
            MoveKernel(kernelCCDF, columnsCCDFs, columnId,
                kernelSize);
        }
    }

    // Move the column CCDFs one step down when done with this
    // row.
    if (rowId < numRows-1){
        MoveColumnsCCDFDown(columnsCCDFs, inputBuffer,
            startRow+rowId, kernelSize, startPixelX -
            kernelRadius, numCols2KernelRad, pictureWidth,
            pictureHeight, tempLocal, bufferOffset);
    }
}
}

```

## Appendix B

# RSA Kernel code

```
// NUM_WORDS is a constant which is how many words each block is.
// Note that this is the number of words in the block after applying
// CRT, so k/2.
// BITMODE is the bitmode which is used as a base. Can be uchar,
// unshort, uint or ulong. ulong is used in the final implementation
// .
// NUMBERS_PER_GROUP = 64/NUM_WORDS, which is how many blocks each
// workgroup will handle.
//
// The kernel which splits the input using CRT and then does Modular
// exponentiation on the result which then is written to the
// outputbuffer.
// The names single, singlePlusOne and double refers to the amount of
// words in the buffer. Single = NUM_WORDS, singlePlusOne =
// NUM_WORDS+1, double = 2*NUM_WORDS.
__kernel void ModularExponentiation(__global BITMODE* inputBuffer,
__global BITMODE* pqBuffer, __global BITMODE* pqDashBuffer,
__global BITMODE* pqPrimeBuffer, __global BITMODE* pqMyBuffer,
__global BITMODE* pqLoopExponentBuffer, __global BITMODE*
outputBuffer){
// Note that all local memory must be declared in the kernel, and
// can't be declared in functions. This is why there is a lot
// of declarations here which are then sent into the functions.
__local BITMODE input[NUMBERS_PER_GROUP*2*NUM_WORDS];
__local BITMODE pq[NUM_WORDS];
__local BITMODE pqPrime[NUM_WORDS];
__local BITMODE pqDash[NUM_WORDS];
__local BITMODE pqLoopExponent[NUM_WORDS];
__local BITMODE pqMy[NUM_WORDS+1];

// As noted above, all temporary buffers used must be declared in
// the kernel and not in any functions, so everything is
// declared here.
__local BITMODE singleResult[NUMBERS_PER_GROUP*NUM_WORDS];
__local BITMODE overflow[NUMBERS_PER_GROUP*(2*NUM_WORDS + 1)];
__local BITMODE singleWordPlusOneTemp1[NUMBERS_PER_GROUP*(
NUM_WORDS+1)];
```

```

__local BITMODE singleWordPlusOneTemp2[NUMBERS_PER_GROUP*(
    NUM_WORDS+1)];
__local BITMODE doubleWordTemp1[NUMBERS_PER_GROUP*2*NUM_WORDS];
__local bool controlBit;

size_t localId = get_local_id(0);
size_t globalId = get_global_id(0);
// This index is the one that will be used in the internal
// functions.
// This is to simplify functions, as all threads will now have
// indices between 0 and NUM_WORDS
size_t modThreadIndex = localId % NUM_WORDS;

// Indexing calculations. Both for the input and for the
// temporary buffers.
size_t groupId = get_group_id(0);
size_t inputNumberInGroup = localId / NUM_WORDS;
size_t groupOffset = inputNumberInGroup*2*NUM_WORDS;
size_t singleOffset = inputNumberInGroup*NUM_WORDS;
size_t singlePlusOneOffset = inputNumberInGroup*(NUM_WORDS+1);
size_t doubleOffset = inputNumberInGroup*2*NUM_WORDS;
size_t overflowOffset = inputNumberInGroup*(2*NUM_WORDS+1);
size_t globalOffset;
size_t offsetMultiplier;
if (groupId % 2 == 0){
    globalOffset = groupId*64;
    offsetMultiplier = 0;
}
else{
    globalOffset = (groupId-1)*64;
    offsetMultiplier = 1;
}

// Reading of input to our local inputbuffer
input[groupOffset + modThreadIndex] = inputBuffer[globalOffset +
    groupOffset + modThreadIndex];
input[groupOffset + modThreadIndex + NUM_WORDS] = inputBuffer[
    globalOffset + groupOffset + modThreadIndex + NUM_WORDS];

// Reading of the variables which are not unique to each group of
// threads in the workgroup. All threads in the workgroup will
// use these variables.
if (localId < NUM_WORDS){
    pqMy[localId] = pqMyBuffer[localId + offsetMultiplier*(
        NUM_WORDS+1)];
    pqMy[NUM_WORDS] = pqMyBuffer[NUM_WORDS + offsetMultiplier*(
        NUM_WORDS+1)];
    pq[localId] = pqBuffer[localId + offsetMultiplier*NUM_WORDS];
    pqPrime[localId] = pqPrimeBuffer[localId + offsetMultiplier*
        NUM_WORDS];
    pqDash[localId] = pqDashBuffer[localId + offsetMultiplier*
        NUM_WORDS];
    pqLoopExponent[localId] = pqLoopExponentBuffer[localId +
        offsetMultiplier*NUM_WORDS];
}
barrier(CLK_LOCAL_MEM_FENCE);

```

```

// Call to an internal function where the real work is done. This
// is to simplify that function, as it doesn't need to care
// about bufferoffsets, as those are handled here.
ModularExponentiationInternal(input + doubleOffset, singleResult
+ singleOffset, pq, pqPrime, pqDash, pqLoopExponent, pqMy,
overflow + overflowOffset, singleWordPlusOneTemp1 +
singlePlusOneOffset, singleWordPlusOneTemp2 +
singlePlusOneOffset, doubleWordTemp1 + doubleOffset, &
controlBit, modThreadIndex);

// Write the result to global memory
outputBuffer[globalOffset + groupOffset + modThreadIndex +
offsetMultiplier*NUM_WORDS] = singleResult[singleOffset +
modThreadIndex];
}

// Internal function for ModularExponentiation. This function doesn't
// know about offsets or such, which simplifies design.
void ModularExponentiationInternal(__local BITMODE* input, __local
BITMODE* singleResult, __local BITMODE* n, __local BITMODE*
nPrime, __local BITMODE* nDash, __local BITMODE* nLoopExponent,
__local BITMODE* nMy, __local BITMODE* overflow, __local BITMODE*
singleWordPlusOneTemp1, __local BITMODE* singleWordPlusOneTemp2,
__local BITMODE* doubleWordTemp1, __local bool* controlBit,
size_t modThreadIndex){
// singleWordsPlusOneTemp3 <- input % p
// Calculate input % n, where n is p or q, this function isn't
// aware of that. This is to reduce the input to a bitsize of k
// /2.
BarretReduction(input, n, singleResult, nMy,
singleWordPlusOneTemp1, singleWordPlusOneTemp2,
overflow, doubleWordTemp1,
controlBit, modThreadIndex);

barrier(CLK_LOCAL_MEM_FENCE);
// Move the result to a temporary buffer.
singleWordPlusOneTemp1[modThreadIndex] = singleResult[
modThreadIndex];
barrier(CLK_LOCAL_MEM_FENCE);
// Do modular exponentiation using square and multiply and
// montgomery reduction.
MontModMult(singleWordPlusOneTemp1, singleResult, nLoopExponent,
n, nPrime, nDash, overflow, controlBit,
singleWordPlusOneTemp2, doubleWordTemp1, input,
modThreadIndex);
barrier(CLK_LOCAL_MEM_FENCE);
// The result will be in singleResult.
}

// This kernel reads the two calculated CRT factors and joins them
// together to the final result.
__kernel void chineseRemainderJoiner(__global BITMODE* inputBuffer,
__global BITMODE* pqBuffer, __global BITMODE* qInvBuffer,
__global BITMODE* pqMyBuffer, __global BITMODE* outputBuffer){
// Same as in the previous kernel, all local memory needs to be
// declared and allocated in the kernel.
// Input variables.
__local BITMODE messagePartP[NUMBERS_PER_GROUP*NUM_WORDS];

```

```

__local BITMODE messagePartQ [NUMBERS_PER_GROUP*NUM_WORDS];
__local BITMODE p [NUM_WORDS];
__local BITMODE q [NUM_WORDS];
__local BITMODE qInv [NUM_WORDS];
__local BITMODE pMy [NUM_WORDS+1];

// Temporary variables.
__local BITMODE singleResult [NUM_WORDS];
__local BITMODE overflow [NUMBERS_PER_GROUP*(2*NUM_WORDS + 1)];
__local BITMODE singleWordPlusOneTemp1 [NUMBERS_PER_GROUP*(
    NUM_WORDS + 1)];
__local BITMODE singleWordPlusOneTemp2 [NUMBERS_PER_GROUP*(
    NUM_WORDS + 1)];
__local BITMODE doubleWordTemp1 [NUMBERS_PER_GROUP*2*NUM_WORDS];
__local BITMODE doubleWordTemp2 [NUMBERS_PER_GROUP*2*NUM_WORDS];
__local bool controlBit;

// Calculation of the threadindex which will be used in the inner
    functions.
size_t localId = get_local_id(0);
size_t modThreadIndex = localId % NUM_WORDS;

// Indexing calculations.
size_t groupId = get_group_id(0);
size_t inputNumberInGroup = localId / NUM_WORDS;
size_t globalOffset = groupId*128;
size_t groupOffset = inputNumberInGroup*2*NUM_WORDS;
size_t singleOffset = inputNumberInGroup*NUM_WORDS;
size_t singlePlusOneOffset = inputNumberInGroup*(NUM_WORDS+1);
size_t doubleOffset = inputNumberInGroup*2*NUM_WORDS;
size_t overflowOffset = inputNumberInGroup*(2*NUM_WORDS+1);

// Reading of input.
messagePartP[singleOffset + modThreadIndex] = inputBuffer[
    globalOffset + groupOffset + modThreadIndex];
messagePartQ[singleOffset + modThreadIndex] = inputBuffer[
    globalOffset + groupOffset + NUM_WORDS + modThreadIndex];

// Reading of data which is not unique to the groups in a
    workgroup.
if (localId < NUM_WORDS){
    p[localId] = pqBuffer[localId];
    q[localId] = pqBuffer[NUM_WORDS + localId];
    qInv[localId] = qInvBuffer[localId];
    pMy[localId] = pqMyBuffer[localId];
    pMy[NUM_WORDS] = pqMyBuffer[NUM_WORDS];
}
barrier(CLK_LOCAL_MEM_FENCE);

// Call to internal function to simplify indexing.
ChineseRemainderJoinerInternal(messagePartP + singleOffset,
    messagePartQ + singleOffset, singleResult + singleOffset, p,
    q, qInv, pMy, overflow + overflowOffset,
    singleWordPlusOneTemp1 + singlePlusOneOffset,
    singleWordPlusOneTemp2 + singlePlusOneOffset, doubleWordTemp1
    + doubleOffset, doubleWordTemp2 + doubleOffset, &controlBit,
    modThreadIndex);

```

```

// Write result to outputbuffer.
outputBuffer[globalOffset + groupOffset + modThreadIndex] =
    doubleWordTemp1[doubleOffset + modThreadIndex];
outputBuffer[globalOffset + groupOffset + modThreadIndex +
    NUM_WORDS] = doubleWordTemp1[doubleOffset + modThreadIndex +
    NUM_WORDS];
}

// Internal helper function to simplify indexing.
ChineseRemainderJoinerInternal(__local BITMODE* messagePartP, __local
    BITMODE* messagePartQ, __local BITMODE* singleResult, __local
    BITMODE* p, __local BITMODE* q, __local BITMODE* qInv, __local
    BITMODE* pMy, __local BITMODE* overflow, __local BITMODE*
    singleWordPlusOneTemp1, __local BITMODE* singleWordPlusOneTemp2,
    __local BITMODE* doubleWordTemp1, __local BITMODE*
    doubleWordTemp2, __local bool* controlBit, size_t modThreadIndex
){
    // Calculate  $a = m_p - m^q \% n$ .
    // Descriptive functionnames are helpful because there is no
    // IntelliSense or such.
    SingleArg1MinusSingleArg2AddArg3IfNegAndReturnTrue(messagePartP,
        messagePartQ, p, singleResult, overflow,
        singleWordPlusOneTemp1, controlBit, modThreadIndex);
    barrier(CLK_LOCAL_MEM_FENCE);

    // Calculate  $b = a * qInv$ 
    Multiply(singleResult, qInv, doubleWordTemp1, overflow,
        controlBit, modThreadIndex);
    barrier(CLK_LOCAL_MEM_FENCE);

    // Calculate  $b \% n$ 
    BarretReduction(doubleWordTemp1, p, singleResult, pMy,
        singleWordPlusOneTemp1, singleWordPlusOneTemp2, overflow,
        doubleWordTemp2, controlBit, modThreadIndex);
    barrier(CLK_LOCAL_MEM_FENCE);

    // Calculate  $c = q * h$ 
    Multiply(singleResult, q, doubleWordTemp1, overflow,
        controlBit, modThreadIndex);
    barrier(CLK_LOCAL_MEM_FENCE);

    // Move q to a buffer with 2*NUM_WORDS in size, to prepare for
    // the next function which adds two 2*NUM_WORDS buffers.
    doubleWordTemp2[modThreadIndex] = 0;
    doubleWordTemp2[NUM_WORDS + modThreadIndex] = messagePartQ[
        modThreadIndex];
    barrier(CLK_LOCAL_MEM_FENCE);

    // Add  $m_q$  to c, calculating the final result m which will be put
    // into doubleWordTemp1.
    AddArg2ToArg1DoubleWordsWillReturnTrueIfOverflow(doubleWordTemp1,
        doubleWordTemp2, overflow, controlBit, modThreadIndex);
    barrier(CLK_LOCAL_MEM_FENCE);
}

```

## På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida <http://www.ep.liu.se/>

## In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: <http://www.ep.liu.se/>