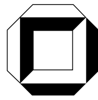


Universität Karlsruhe (TH)
Research University · founded 1825

Parallel programming with CUDA
Architecture, Analysis, Application

Student research project at the
Institute for Program Structures und Data Organization
Chair for Programming Systems
Prof. Dr. W. Tichy
Fakultät für Informatik
Universität Karlsruhe (TH)

and
Agilent Technologies Deutschland GmbH
Waldbronn

by

cand. inform.
David Münch

Advisor:
Prof. Dr. W. Tichy
Dr. Victor Pankratius

Date of Registration: 2008-02-02
Date of Submission: 2009-04-07

I declare that I have developed and written the enclosed Student research project completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, 2009-04-07

Nvidia's *Compute Unified Device Architecture* (CUDA) promises several teraflops of performance on GPUs. In this work, the CUDA programming model is investigated. Matrix multiplication, discrete convolution and a morphological filter operation - - the *Rolling Ball* algorithm - are implemented, compared, and their performance is evaluated. The results are speedups up to 190 and a variety of limitations on the GPU which have to be accepted. If the problem to solve with CUDA on the GPU has the right structure, meaning it is data-parallel, it has a high arithmetic intensity, it is not memory intensive and it is large enough, very good performance can be expected.

Contents

1 Introduction to NVIDIA's CUDA **1**

2 Hardware Structure & Programming Model **3**

2.1 Basic Hardware Structure 3

2.2 General Overview of CUDA Capable GPUs 4

2.3 Thread Hierarchy 5

2.4 Memory Hierarchy 6

2.5 Summary 8

3 Matrix Multiplication **9**

3.1 Approaches 9

3.1.1 Sequential CPU implementation 9

3.1.2 OpenMP Optimised CPU implementation 10

3.1.3 GPU implementation 10

3.1.4 CUBLASSGEMM Library Function 10

3.2 Environment for Performance Evaluations 10

3.2.1 Hardware 11

3.2.2 Software 11

3.2.3 Testing Process 12

3.3 Performance Evaluation 12

3.4 Summary 14

4 Discrete Convolution **15**

4.1 Sequential Algorithm 15

4.2 Designing the Parallel Algorithm 16

4.3 Transform the Parallel Algorithm to the GPU - First 17

4.4 Transform the Parallel Algorithm to the GPU - Second 18

4.5 Performance Evaluation 21

4.6 Summary 22

5 Rolling Ball **29**

5.1 Sequential Algorithm 29

5.2 Designing the Parallel Algorithm 31

5.3 Transform the Parallel Algorithm to the GPU 31

5.4 Performance Evaluation 33

5.5 Summary 35

6 Limitations of CUDA **41**

6.1 Kernel Call Overhead 41

6.2 Memory Copying Overhead 42

6.3 Upper Bound of Performance 43

6.4 IEEE-754 Precision 45

6.5 CUDA depends on NVIDIA 45

6.6 Other Problems 46

6.7 Summary 46

7 Discussion **47**

7.1 Comparison with Multi-core Processors 47

7.2 Consequences for Software Engineering 48

7.3 CUDA worth the effort 50

8 Conclusion & Future Work **51**

A Appendix - Source Code **53**

B Appendix - Additional Runtime Measurements **65**

C Appendix - Runtime Measurement Data **73**

Bibliography **89**

1. Introduction to NVIDIA's CUDA

Over the last few years Parallel Programming has turned into a major area in computer science. Theoretical basics of Parallel Programming have been developed since the 1950s[Gil58][Wil94], but no affordable, parallel hardware was available for the consumer market. Times changed in 2005[inta] when Intel released its first mainstream multi-core CPU, which was the advent of Parallel Programming. Considering that Graphics Processing Units (GPU) already are many-core processors, in 2007 Nvidia introduced their architecture *Compute Unified Device Architecture* (CUDA). There are three reasons why Parallel Programming with CUDA is getting more and more popular: the hardware is now available, it is comparably cheap and a great number of consumer computers have a CUDA-capable Nvidia GPU.

A modern GPU is no longer only a memory controller and display generator as it used to be in the 1990s. Instead, it is a highly parallel and multithreaded multiprocessor. Being both a programmable graphics and a scalable programming platform, a modern GPU is breaking the mould concerning the variety of capabilities. To take advantage, it was necessary to add some processor instructions and memory hardware to the GPU and provide a more general API. With these modifications the data-parallel GPU can be used as a general-purpose, programmable many-core processor with its own benefits and limitations. The modern GPU is characterised by its large amount of floating-point processing power, which can be used for nongraphical problems. This was the birth of the programming model CUDA, which bypasses the graphics API of the GPU and allows simple programs in C. Single-Program, Multiple Data (SPMD) is the underlying abstraction to achieve high parallelism on the thread level. In the SPMD style of parallel programming all the threads execute the same code on different portions of data, see [Ata99]. The coordination is done with a barrier synchronisation method. In summary, the three key abstractions of the CUDA programming model are:

- hierarchy of thread groups,
- shared memory and

- barrier synchronisation.

The two components of the programming system are the *host* (=CPU) and at least one *device* (=GPU).

$$\textit{host} \xrightarrow{\text{uses as a coprocessor}} \textit{device}$$

The host calls and controls functions running massively parallel on the device. The host code has a few extensions of a programming language or API (see Figure 1.1) to specify the execution parameters for device functions, to control the device, memory and context management and more. Currently, the functions callable on the device are limited by those provided by the high or low-level CUDA APIs. They comprise some mathematical, texture and memory functions as well as barrier synchronisation. Nvidia's marketing department is successfully advertising their CUDA-capable

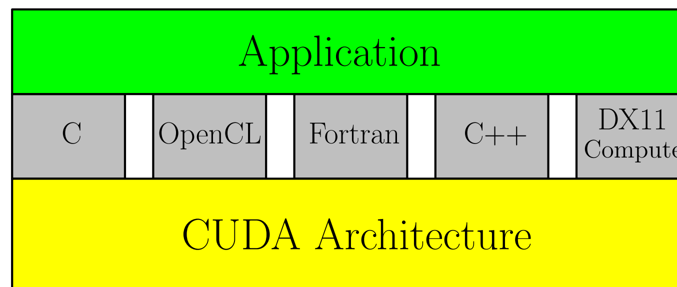


Figure 1.1: CUDA supports various languages and APIs

GPUs and promising an easy and instantly learnable programming model resulting in speedups of 10 to 200[nvic]. However, a more detailed and a closer inspection reveals a rudimentary programming model with many major limitations compared to standard programming, such as recursion and the IEEE 754 standard, see Chapter 6.

The following chapters will provide details on the GPU's architecture and the CUDA programming model, a presentation of our test configuration, investigation of an existing matrix multiplication and development of a discrete convolution algorithm to become familiar with CUDA. Finally, a morphological filter for a real life product will be developed, the limitations of CUDA will be evaluated and its programming model will be discussed.

2. Hardware Structure & Programming Model

This chapter takes a closer look at the CUDA programming model and the underlying hardware structure. The first part introduces the hardware implementation of the CUDA programming model, the second presents a CUDA-capable GPU and some CUDA basics and the third explains the thread and memory hierarchy.

2.1 Basic Hardware Structure

The Nvidia Quadro FX 3700 is a high-end workstation graphics solution for advanced graphics applications. It has a G92 core consisting of 14 Streaming Multiprocessors as seen in Figure 2.1, see [PH09]. Each Streaming Multiprocessor consists of eight Streaming Processors (SP), two Special Function Units, shared memory, multithreaded instruction unit, constant and instruction caches. A Streaming Multiprocessors is connected to the 512MB DRAM with an interconnection network with a theoretical peak bandwidth of 51.2GB/s. Nvidia uses a very scalable technique for their GPUs, as the running time scales linearly with the number of Streaming Multiprocessors. If there are more Streaming Multiprocessors, more work can be computed at the same time. CUDA provides automatic distribution to the different Streaming Multiprocessors.

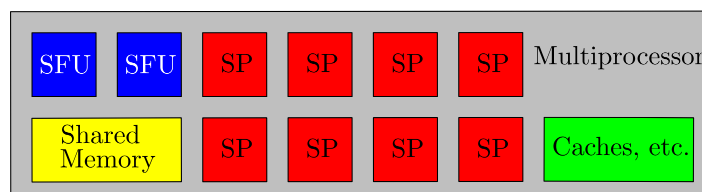


Figure 2.1: Multiprocessor

In CUDA, threads are executed in groups of 32 threads to hide memory latency. A group of 32 threads is called a **warp** which is distributed to the SPs in a Streaming

Multiprocessor, so that each SP gets four threads for four clock cycles. Such an architecture is called a Single-Instruction Multiple-Thread (SIMT) Architecture. The physical limits for the GPU are 24 warps per Streaming Multiprocessor, 768 threads per Streaming Multiprocessor, 8 threadblocks, see 2.3, per Streaming Multiprocessor and 16kB shared memory per Streaming Multiprocessor. The programmer defines the execution parameters with the size of the threadblock. E.g. dimBlock(256,1,1) implies three active threadblocks per Streaming Multiprocessor consisting of 24 active warps of 32 threads. CUDA maps the different threadblocks to the Streaming Multiprocessors. It is difficult to find the best execution parameters for a particular application. Most of the time it is better not to have 100% occupancy of each Streaming Multiprocessor, but more shared memory per threadblock. These parameters have to be tested with every application and cannot be predicted, at most estimated. The CUDA Toolkit contains the CUDA GPU Occupancy Calculator or the CUDA Visual Profiler to vary these parameters.

2.2 General Overview of CUDA Capable GPUs

A standard CPU comprises a top level control logic, a few Arithmetic Logic Units and a hierarchy of fast caches. However, a modern GPU has few control and cache units, but many Streaming Processors, similar to Arithmetic Logic Units. By nature, the GPU has to compute the visual output in a simple, but highly data parallel way, therefore the caches and complex control flow are not necessary. Figure 2.2 illustrates the described difference between CPU and GPU. Unlocking the potential power of Nvidia's GPUs is what CUDA has been developed for.

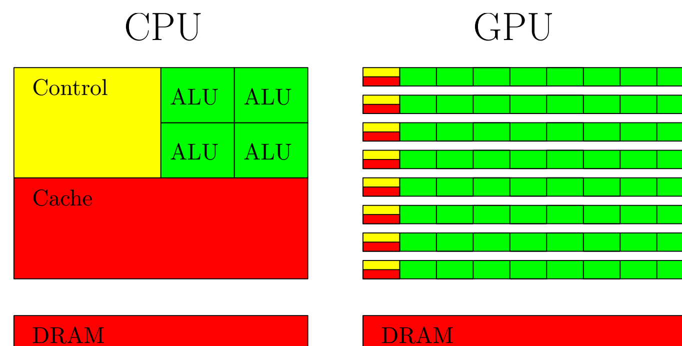


Figure 2.2: CPU compared to GPU

As seen in chapter 1, the CUDA programming model consists of a host and at least one device, working as the host's coprocessor running C code. The programmer can choose between the CUDA Runtime API and the CUDA Driver API. The Driver API is closer to the hardware of the GPU and more difficult to use than the simpler Runtime API. On top of the CUDA APIs, there are some libraries such as a CUDA-adopted BLAS library, CUBLAS[nvia]. The different layers of the APIs, Libraries and the Application are shown in Figure 2.3.

An example programming code using the Runtime API for the host can be seen in Listing 2.1. The truncated host function *main* calls in line 5 the device function *pMul*, also named *kernel* function. Each thread computes exactly one element of the array C with all the threads running parallel. The elements in line 13 are computed

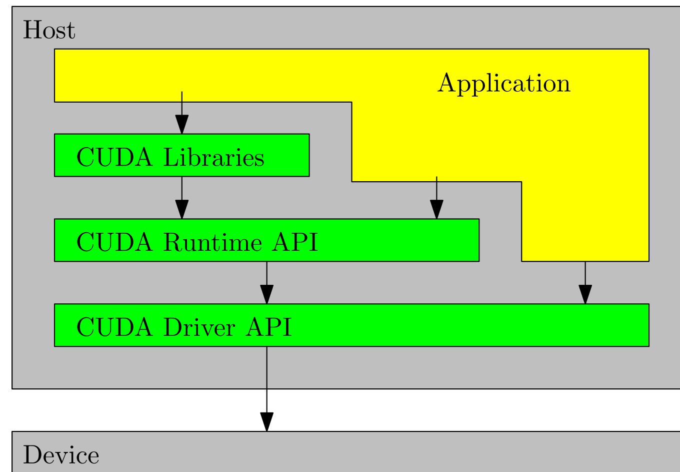


Figure 2.3: CUDA software stack

concurrently. Line 13 is equivalent to **for** $i \longleftarrow 0$ **to** $|A|-1$ **do** $C[i] \longleftarrow A[i]*B[i]$ in a sequential program.

```
//Host function, calling the kernel function pMul
int main()
{
  //Kernel invocation. |A|=|B|=|C|=N
  pMul<<<1,N>>>(A,B,C);
}

//Device function computing pair-wise the product of two
//vectors A and B and stores the result in C
__global__ void pMul(float* A, float* B, float* C)
{
  unsigned int i = threadIdx.x;
  C[i]=A[i] * B[i];
}
```

Listing 2.1: Simple host and device function

In the following section the execution parameters and the thread hierarchy of the device will be examined.

2.3 Thread Hierarchy

A thread on a GPU is a single path of execution. To organise the threads and map them to the hardware it is necessary to have a thread hierarchy. The CUDA programming model has a scalable thread hierarchy with the smallest entity being a single **thread**. A three-dimensional array of single threads is a **block**. Multiple blocks are organised in a two-dimensional **grid**. Each kernel on the device is invoked on a grid. The execution parameters in the arrow brackets seen in Listing 2.1 line 5 show one grid with one one-dimensional threadblock with N threads. In general, the grid parameter $dimGrid(x,y)$ is a two-dimensional vector, the threadblock parameter $dimBlock(x,y,z)$ is a three-dimensional vector.

Figure 2.4 shows an example of a two-dimensional grid consisting of 4×2 two-dimensional threadblocks each with 5×3 threads. Altogether this totals $1\cdot4\cdot2\cdot5\cdot3\cdot1 = 120$ threads.

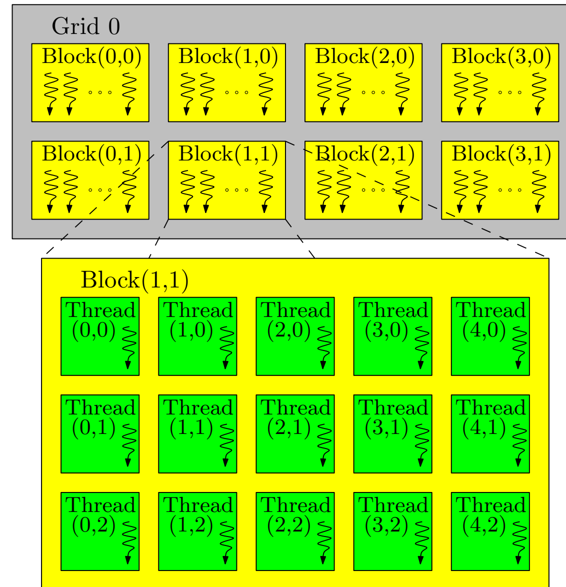


Figure 2.4: Example thread hierarchy

Typically, the programmer defines the execution parameters by the problem size and not by the number of multi-cores on the GPU.

2.4 Memory Hierarchy

There is no cache hierarchy on the GPU similar to the one you can find on a CPU. This is kind of a challenge and change, as by now the conventional programmer has nothing to do with the memory transfers in the cache hierarchy. He allocates, initialises, uses and frees memory, but he does not copy the data from DRAM to Level 3 Cache and so on. To the contrary, the requirements for fast memory transactions on the GPU are hardware related. Without knowledge of the exact hardware structure of the GPU, writing efficient programs will likely not be successful.

First of all, the GPU's *main* memory is a large global DRAM with a size between 512MB and 4GB. Its characteristics are summarised in Table 2.1 in [RRB+08]. *Global* memory is not on the cores but connected with an interconnection network. Thus, it has an unsuitable hit latency of about 200-300 cycles on the GPU. Unfortunately, the programmer is forced to use global memory as it is the only readable and writable memory for all grids.

Shared memory is a benefit but at the same time a hindrance to the programmer. Each threadblock has its own shared memory, currently only 16kB. The big advantage is it has really fast access. To benefit from this advantage, the programmer has to copy the data manually from global to shared memory. For example, using four byte numbers, 4096 numbers can be stored in shared memory. While computing, additional space for intermediate results is needed. In some cases, the compiler might require some additional demand for shared memory. As a result, the programmer is often limited by shared memory.

Each thread has a private *local* memory consisting of registers. If necessary, a thread-owned part of global memory can be allocated. Read-only *constant* memory might be interesting although it resides in global memory, because it is cached per multiprocessor. The 8kB cache of the multiprocessor is as fast as shared memory. If the programmer is limited by shared memory he can bypass the limitation with constant memory.

Texture memory can be useful in certain applications like video encoders etc.

Memory	Location	Size	Hit Latency	Read Only	Program Scope
Global	off-chip	512MB total	200-300 cycles	no	global
Local	off-chip	up to global	same as global	no	function
Shared	on-chip	16kB per SM	$\simeq$ register latency	no	function
Constant	on-chip	64kB total	same as shared	yes	global
Texture	on-chip	up to global	approx. 100 cycles	yes	global

Table 2.1: Memory on Nvidia Quadro FX 3700 GPU

Adding the memory management to functionality Listing 2.1, generates Listing 2.2. In the host function lines 5 - 10 and 16 are added to allocate memory on the host and on the device and to copy data from the host to the device and back. In the device function, it is necessary to allocate sufficient shared memory and copy the data from global to shared memory and back. Fortunately, this can be done in parallel. This is the reason why barrier synchronisation in Lines 29 and 37 is needed, to ensure that the copying action has finished.

```
1  //Host function, calling the kernel function pMul
2  //assert |A|=|B|=|C|=N < (shared memory capacity / 3)
3  int main()
4  {
5    initialize host_A, host_B and host_C;
6    initialize A, B and C on the device;
7
8    //Copy the two arrays host_A and host_B to device memory
9    cudaMemcpy(A, host_A, sizeof(host_A), cudaMemcpyHostToDevice);
10   cudaMemcpy(B, host_B, sizeof(host_B), cudaMemcpyHostToDevice);
11
12   //Kernel invocation. |A|=|B|=|C|=N
13   pMul<<<1,N>>>(A,B,C);
14
15   //Copy the result C back to the host
16   cudaMemcpy(C_host, C, sizeof(C), cudaMemcpyDeviceToHost);
17 }
18
19 //Device function computing pair-wise the product of two
20 //vectors A and B and stores the result in C
21 __global__ void pMul(float* A, float* B, float* C)
22 {
23   //initialise shared memory
24   __shared__ s_A[N]; __shared__ s_B[N]; __shared__ s_C[N];
```

```
25   unsigned int i = threadIdx.x;
26
27   //copy data from global to shared memory
28   s_A[i]=A[i];   s_B[i]=B[i];
29   __syncthreads();
30
31   //do some arithmetic operations
32   //in practice much more operations!
33   s_C[i]=s_A[i] * s_B[i];
34
35   //copy data from shared to global memory
36   C[i]=s_C[i];
37   __syncthreads();
38 }
```

Listing 2.2: Simple host and device function with memory management

2.5 Summary

In this chapter a general overview of the differences between a CPU and a GPU has been given. Currently there are major differences, especially in the rudimentary memory management. Additionally, the basic structure of the CUDA programming model is simple and not powerful. The whole programming model is very close to the hardware which means a considerable programming effort.

3. Matrix Multiplication

In the following chapter, matrix multiplication will be examined. First, the different approaches used, second, the evaluation and finally the comparison of the results will be shown.

3.1 Approaches

To evaluate matrix multiplication with CUDA, a CUDA implementation cannot only be studied alone, it also has to be compared with other results. Therefore, CUDA matrix multiplication will be compared to two simple CPU versions. Furthermore, the performance of the official CUBLAS library is examined. In the following subsections the four algorithms to be examined will be described.

3.1.1 Sequential CPU implementation

Algorithm 1 is a simple sequential matrix multiplication algorithm for the CPU. This ikj-algorithm is a cache-efficient modification of the standard brute-force ijk-algorithm.

Algorithmus 1: ikj matrix multiplication algorithm

Input: floating-point matrices A, B
Result: matrix $A \cdot B = C$

1 **for** *int* $i = 0$; $i < height_A$; i++ **do**
2   **for** *int* $k = 0$; $k < width_A$; k++ **do**
3         $inv = A[i][k]$;
4   **for** *int* $j = 0$; $j < width_B$; j++ **do**
5             $C[i][j] += inv * B[k][j]$;

3.1.2 OpenMP Optimised CPU implementation

Parallelisation of the outer loop of Algorithm 1 with an OpenMP command is performed and consequently Algorithm 2 is the result.

Algorithmus 2: ikj matrix multiplication algorithm

Input: floating-point matrices A, B
Result: matrix $A \cdot B = C$

```
1 #pragma omp parallel for private (inv)
2 for int i = 0; i < height_A; i++ do
3     for int k = 0; k < width_A; k++ do
4         inv = A[i][k];
5         for int j = 0; j < width_B; j++ do
6             C[i][j]+ = inv * B[k][j];
```

3.1.3 GPU implementation

Algorithm 3 from the CUDA SDK [nvif] is used to determine the performance of the GPU. Although Nvidia developed Algorithm 3 "not with the goal of providing the most efficient generic kernel for matrix multiplication" it is, however, very efficient and it shows various design principles of parallel computing on GPU. A detailed description of Algorithm 3 can be found in [nvib].

Algorithmus 3: Nvidia's CUDA SDK matrix multiplication algorithm

Input: floating-point matrices A, B
Result: matrix $A \cdot B = C$

1 Load the matrices A and B blockwise from global device memory to shared memory as needed.
2 **while** *not all C_Sub are computed* **do**
3   Let each block compute an submatrix *C_Sub* as seen in Figure 3.1. Each thread of an block computes one element of *C_Sub*.

3.1.4 CUBLASSGEMM Library Function

In order to compare Algorithm 3 to a high-performance matrix multiplication, the Single-Precision General Matrix Multiply (CUBLASSGEMM) subroutine from the CUBLAS Library is used, which is directly accessible with C language, so, a programmer does not need to know anything about programming with CUDA or about GPU architecture.

3.2 Environment for Performance Evaluations

This section describes the environment which accounts for all of our developed and evaluated algorithms below. First, the hardware and software being used will be reviewed, second, the testing process will be presented and finally, there are a few basic definitions to be given.

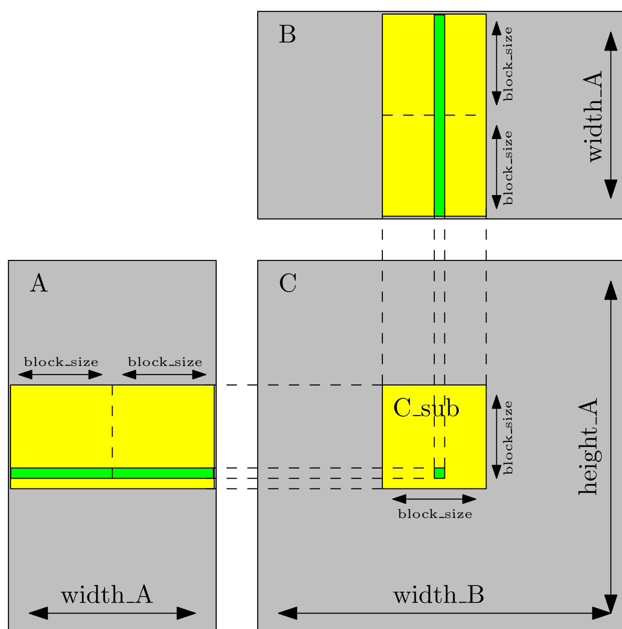


Figure 3.1: Visualization of Algorithm 3. Each thread block computes one submatrix C_{Sub} of C. Each thread within the block computes one element of C_{Sub}. See [nvib].

3.2.1 Hardware

All the following examinations of algorithms will be performed on a DELL Precision Workstation T5400, which is equipped with an Intel Xeon E5430 running at 2,66GHz, with 8GB Random Access Memory (RAM) and a Nvidia Quadro FX 3700 GPU. This kind of workstation is appropriate for the needs of the algorithms and it is comparable to those available for chemical engineers who will apply the examined algorithms.

3.2.2 Software

Microsoft Windows XP Professional 32Bit with Service Pack 3 is used as the operating system. As the x86 address size of 32bits cannot address the entire 8GB RAM, it is necessary to enable the Physical Address Extension. The decision for using the 32Bit operating system instead of 64Bit is due to of well-engineered software libraries instead of experimental beta releases. It has been a requirement to use the Microsoft Visual Studio 2008 Development Edition for the implementation of the algorithms. The applied key components of CUDA 2.1 are:

- NVIDIA Driver for Microsoft Windows XP with CUDA Support (181.20),
- the CUDA Toolkit version 2.1 for Windows XP,
- the CUDA SDK 2.1 for Windows XP
- and the CUDA Visual Profiler 1.1.

The Profiler is only compatible with Windows XP. The CUDA resources are free to download from the web[nvie].

3.2.3 Testing Process

All the following algorithms are evaluated as follows: Every configuration has to run three times under the same conditions. The resulting time $\overline{x}$ is the arithmetic mean of the three measured runtimes x_i. While running the tests, the computer does not perform anything else of significance.

The speedup S, in parallel programs is the ratio between the runtime of the best sequential program T_1 and the runtime of the parallel program T_P.

$$S = \frac{T_1}{T_P}$$

It is a wrong assumption that the speedup increases linearly with the number of cores. Instead, it is limited by the sequential part of every program as Amdahl describes in [Rod85].

3.3 Performance Evaluation

First, the running time of the CPU, OpenMP-optimized-CPU, GPU and CUBLASS-GEMM implementations with different dimensions of matrices have been determined. With CUDA, it was not possible to use matrices with dimensions larger than 4096, as memory allocating errors occurred. The runtime of the CPU versions is the pure computing time of the matrix multiplication ranging from calling the multiply function until the function returns. To compare the results between CPU and GPU in a suitable way, the memory overhead is added - consisting of allocating, copying the data to GPU and reading the result back - to the pure running time of the *kernel*-function on the GPU.

In Figure 3.2, the runtime of the four different implementations can be seen. While doubling the matrix' size, the runtime increases eight times.
With a poor result for small matrices, Algorithm 2 improves the performance for larger matrices asymptotically by a factor of four as shown in Figure 3.3. For small matrices the use of OpenMP is counterproductive because of its overhead, whereas a speedup of four on a quad-core confirms good parallel code.
To our surprise, Algorithm 3 is performing quite well for matrices smaller than $n = 512$ ($n \times n$ matrix). As Algorithm 3 is in $\mathcal{O}(n^3)$ for multiplications and additions and mostly operate on neighboring data elements, the $\mathcal{O}(n^2)$ memory overhead does not play an important role in this case.
The "high-performance matrix multiplication" of CUBLAS has poor results for matrices smaller than $n = 512$ and best results for huge matrices were obtained, resulting in a speedup of only 2.4 to Algorithm 3. For $n = 4096$, this is equivalent to a performance of $137 GFlop/s$. Unfortunately, CUBLAS with matrices larger than $n = 4096$ fails with a memory allocating error on the GPU. Compared to other routines of CUBLAS, CUBLASSGEMM is optimized good compared to e.g. CUBLASSSYRK, see [BCI+08]. However, the "standard" GPU implementation is not much slower than the CUBLAS routine, although it seems as if CUBLAS is not completely optimised.

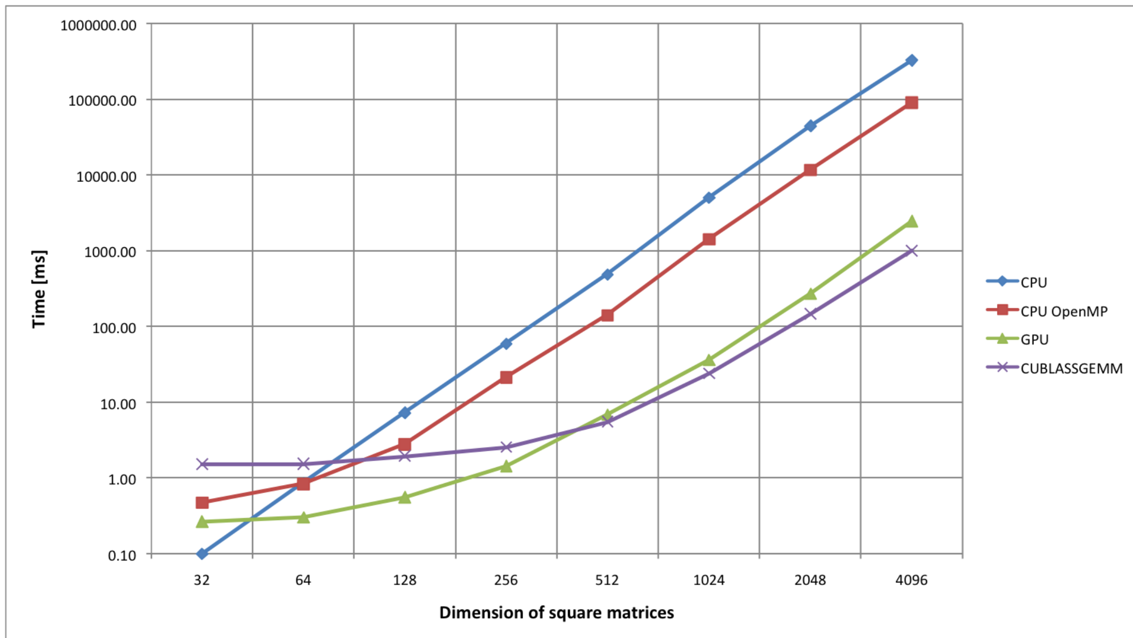


Figure 3.2: Runtime of several matrix multiplications on CPU and GPU

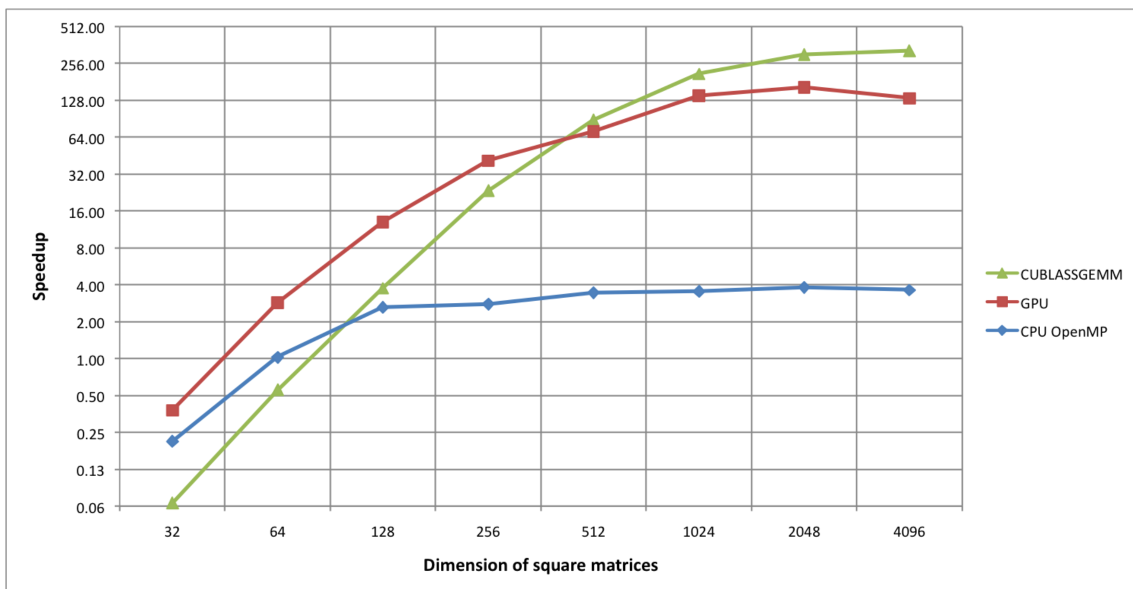


Figure 3.3: Speedup of matrix multiplication on GPU

3.4 Summary

The analysis of the different matrix multiplication approaches leads to the conclusion that one algorithm cannot be preferred over another in general. Each one has its field of application, depending on the problem size. Whereas small problems are solved faster on the CPU, bigger problems are solved faster on the GPU.

4. Discrete Convolution

In this chapter discrete convolution will be examined, a parallel algorithm will be developed from a sequential algorithm and then adapted to the CUDA programming model. Last, the different algorithms will be evaluated.

Discrete convolution is defined as follows: $f, g : D \to \mathbb{C}$, where $D \subseteq \mathbb{Z}$.

$$(f * g)(n) = \sum_{k \in D} f(k) \cdot g(n-k) \tag{4.1}$$

One function f is weighted by another function g. It is widely used in digital signal processing and many other fields of application. The convolution, for example, is part of the Savitzky-Golay smoothing filter [SG64], which is used as a signal smoothing algorithm to enhance the signal-to-noise ratio. The aim was not to use a Fast Fourier Transform, but rather to understand the CUDA programming model.

4.1 Sequential Algorithm

The brute force approach of the discrete convolution described in Algorithm 4 is simple, but inefficient, as its complexity is in $\mathcal{O}(|M| \cdot |N|)$, $|M|$ being the signal length and $|N|$ the filter width. A concrete C implementation can be found in Appendix A.1.

Algorithmus 4: Brute Force Discrete Convolution

Input: signal M, filter N w.l.o.g. $|N|$ is odd.
Output: discrete convolution $P = M * N$

1 **for** $n \longleftarrow 0$ **to** $|P|$ **do**
2   **for** $k \longleftarrow |N| - 1$ **downto** 0 **do**
3         $P[n] \longleftarrow P[n] + M[n + k - (|N| - 1)] \cdot N[k]$;
              `/* let` $M[i] \longleftarrow 0$`, if` $M[i]$ `is out of range. */`
4 **return** P

In Figure 4.1 the discrete convolution is visualised. The output value is the sum of the distinct signal data “weighted” with the filter. Thus, first $|N|$ multiplications

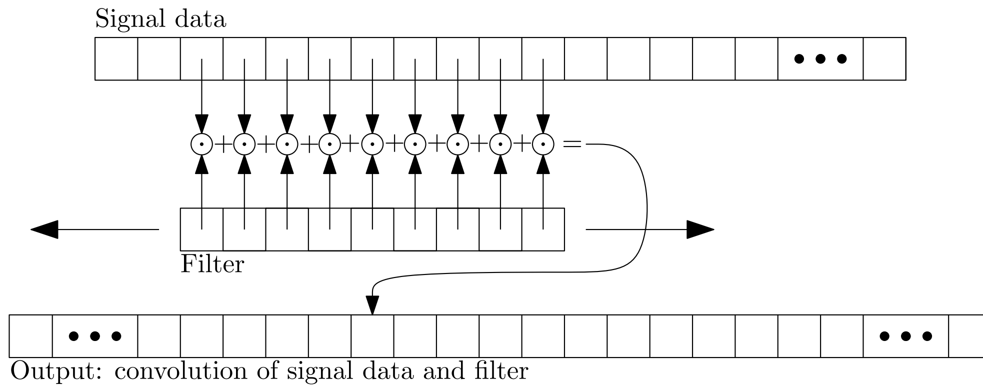


Figure 4.1: Visualisation of discrete convolution

and afterwards the $|N|-1$ summations have to be done. As expected, during these simple operations there were no problems occurring in the sequential Algorithm 4.

4.2 Designing the Parallel Algorithm

No concurrent memory transactions occur in the sequential Algorithm 4, however due to the parallelised Algorithm 5, concurrent writes could occur e.g. in line 5, causing a conflict. Therefore, the **reduction** and **private** clauses are used. The OpenMP API is used, as it is very simple to use and fits perfectly together with our easy sequential Algorithm 4. In various tests, the fastest optimisation technique for parallelising the two loops with OpenMP were figured out. The result is quite simple: create one thread per core, e.g. our implementation for four CPU-cores is shown in Algorithm 5: Parallelise the outer loop to minimize overhead caused by OpenMP. See also [TM08]. A concrete C implementation can be found in Appendix A.2.

Algorithmus 5: Parallelised Brute Force Discrete Convolution

Input: signal M, filter N w.l.o.g. $|N|$ is odd.
Output: discrete convolution $P = M * N$

```
1 omp_set_num_threads(4);
2 #pragma omp parallel for private(k, p) reduction(+ : sum) for n ⟵ 0 to |P|
  do
3     sum = 0;
4     for k ⟵ |N| − 1 downto 0 do
5         sum ⟵ sum + M[n + k − (|N| − 1)] · N[k];
                              /* let M[i] ⟵ 0, if M[i] is out of range.  */
6     P[i] = sum;
7 return P
```

The following paragraph will show that it is not as simple as that with a parallel algorithm on the GPU.

4.3 Transform the Parallel Algorithm to the GPU - First

The first attempt tries to parallelise the outer loop of Algorithm 5 line 3 and following lines in distributing parts to different blocks of the GPU. Each block computes the elements stepwise. Additionally, the whole block also parallelises the reduction part in line 5. See Figure 4.2. In an example configuration with 256 threads per block and 3 active threadblocks per multiprocessor $2 \cdot 3 \cdot 14 = 84$ elements of the result are computed at the same time. But the speedup was poor: 1.5 times faster then Algorithm 4. Looking at CUDA's architecture, three major reasons were identified: First, considering the load: A reduction with n threads takes $\log_2 n$ time. There are $2n - 1$ active and $n \log_2 n - n + 1$ idle units of time of the threads. With $n = 256$, a load of only 28.5% is achieved, thus wasting computing time. As already shown above, a high computational component is needed to amortise the expensive memory transfers. Second, the profiler revealed slow memory transactions and divergent branches, causing the entity of a warp to be serialised. Third, the fast, but small $16kB$ shared memory limits the execution time. Additionally, the windows watchdog causes a runtime problem resulting in big amounts of data, as it terminates the device function after having been run for ca 5 seconds.

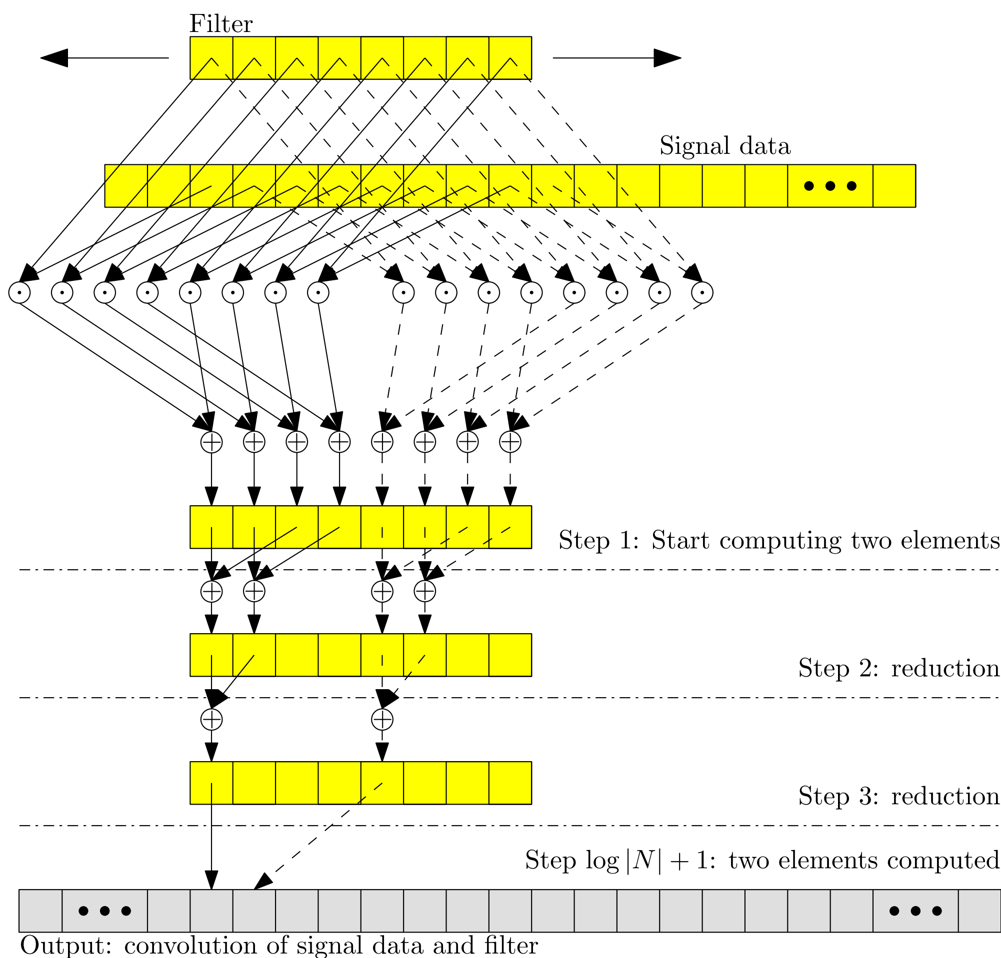


Figure 4.2: Visualisation of our first attempt of parallel discrete convolution

After disappointing optimisation results with the study of [Har07], it is interesting to see how many points have to be considered to write a fast and efficient algorithm with CUDA. In this first attempt the following optimisations were done:

- Avoid divergent branching and bank conflicts ⇒ sequential addressing.
- First add during global load, see Figure 4.2 Step 1.
- Unroll the last 32 threads -corresponds a warp- of the reduction part, because a warp is the smallest entity executing in parallel.
- Compute multiple elements per thread, see Figure 4.2 Step 1.

Afterwards a performance gain of 5.2 and an overall speedup of 7.8 were achieved, however this result was not satisfying and a second attempt to transform Algorithm 5 to the GPU was started.

4.4 Transform the Parallel Algorithm to the GPU - Second

In this second attempt the direction of parallelisation from the reduction of one or two elements orthogonal to the computation of many elements in parallel is changed. Now, knowing some problems with CUDA, every design decision was made very carefully.
One aim of Algorithm 6 and 7 is to be scalable, another to deal with the various problems of CUDA like the Windows watchdog and last not to overflow shared memory and results in slow global memory. To achieve this, the whole convolution is not computed with one kernel call, but the filter of size N is split up into parts of 384 each. Thus, Algorithm 7 calls the device function, Algorithm 6, several times, until the whole filter is completed. The runtime of one kernel call is only dependent on the input data M, resulting in a runtime of milliseconds. Our concrete implementation of the host and device function is shown in Appendix A.3 and A.4.

Algorithmus 6: Device Function of Discrete Convolution

Input: signal M, filter c_N w.l.o.g. $|c_N|$ is odd., filter offset fo
Output: discrete convolution $P = M * N$

```
// Initialize memory
1  initialise shared memory s_M for signal data with zero ;
2  initialise shared memory s_P for result with zero;
3  tid ⟵ threadIdx.x                         /* current thread identifier */
4  bid ⟵ blockIdx.x                           /* current block identifier */
5  dim ⟵ blockDim.x                            /* current block dimension */
   // Memory copy on device:  global → shared
6  s_M[tid] = M[bid * dim + tid + fo];
7  s_M[tid + dim] = M[(bid + 1) * dim + tid + fo];
                            /* every block gets its dedicated signal data */
8  __syncthreads();
        /* Barrier synchronisation to complete the copying operation */
   // loop in parallel over every computed output value
9  for i ⟵ 0 to dim do
10     s_P[tid] = s_P[tid] + (s_M[tid + i] * c_N[i]);
11     __syncthreads();
           /* Barrier synchronisation to complete the store operation */
   // write back the result from shared to global memory
12 P[bid * dim + tid]+ = s_P[tid];
13 __syncthreads();
        /* Barrier synchronisation to complete the copying operation */
```

Algorithmus 7: Host Function of Discrete Convolution

Input: signal M, filter N w.l.o.g. $|N|$ is odd.
Output: discrete convolution $P = M * N$

```
1  for fo ⟵ 0 to fo < |N| do
       // copy current needed part of the filter to constant memory
2      cudaMemcpyToSymbol("c_N", &N[fo], num_threads * sizeof(float));
       // call the device function
3      Algorithm 6 <<< gridsize, num_threads >>> (M, fo, P);
4      cudaThreadSynchronize();
5      fo ⟵ fo + num_threads;
6  return P
```

Thus, $\mathcal{O}(|N|)$ arithmetic operations have to be done per thread. As the filter is the same for all threadblocks and does not change, it is stored in the fast cached constant memory. Additionally this is done to save shared memory, which is limited to $16kB$ per multiprocessor and was a bottleneck in the first attempt. See Figure 4.3. In an example, the configuration is 384 threads per block, which computes 384 elements in parallel. The blocksize of 384 implies 8kB shared memory per thread block, two active thread blocks per SM and an occupancy of 100% of the SM.

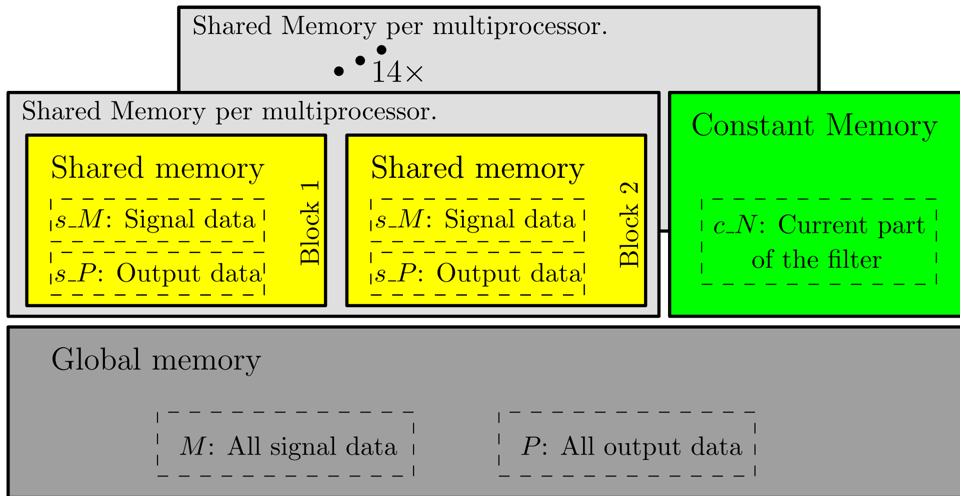


Figure 4.3: Visualisation of memory hierarchy of Algorithm 6 and 7.

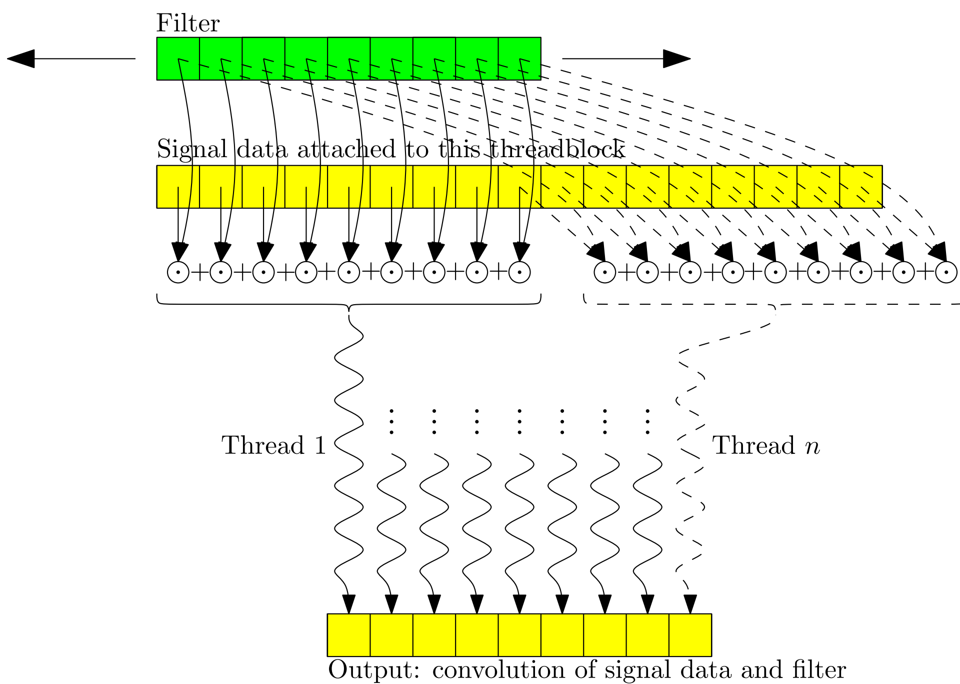


Figure 4.4: Visualisation of our second attempt of parallel discrete convolution.

The device function shown in Algorithm 6 is a result of consideration of almost every optimisation of CUDA. Later it can be seen that its efficiency is based on its simplicity and uncompromising application of CUDA-related design principles.

The device function of the discrete convolution is visualised in Figure 4.4. With each threadblock of size n, n elements are computed in parallel, thereby every thread works on one output element. The CUDA implementation of the device function can be seen in Algorithm 6. First, it is necessary to allocate shared memory for the input signal data and the result, see lines 1f. The filter remains in the fast cached constant memory. Second, in lines 6f the each threadblock dedicated data is copied from global to shared memory. The subsequent barrier synchronisation completes the copying operation. Third, in the computing part of this algorithm, lines 9-11 loop over every element of the currently considered part of the filter. Finally, the intermediate result is written back to global memory, ready for further use. The above described memory hierarchy can be seen in Figure 4.3.

4.5 Performance Evaluation

In this section the evaluation of Algorithms 4, 5 and 6 & 7 will be presented. Different filter widths from 1 to 999.999 and signal data sets with 10 to 10.000.000 elements have been used. In Appendix C detailed measuring values and other diagrams are attached.

In Figure 4.5, the runtime of the sequential Algorithm 4 on CPU is shown. The expected $\mathcal{O}(n^3)$ growth without irregularities can be seen.

Figure 4.6 shows the runtime of parallelised Algorithm 5 on CPU with the OpenMP library. For small instances, a great overhead compared to the sequential algorithm can be seen. That is as OpenMP needs some time to be loaded.

Figure 4.7 shows the relationship between the sequential and the parallel algorithm. For small instances it is counterproductive to use Algorithm 5 because of its overhead. However, for instances with $datasize \cdot filtersize > 10.000$ the speedup converges to four.

Figure 4.8 visualises the runtime of Algorithm 6 and 7 with all the memory transfers from and to the device. In contrast, Figure 4.9 visualises the same without the memory overhead. Small instances need much time.

Figure 4.10, explicitly shows the pure overhead which will never decrease below approx. 25ms. This is caused by memory management and the kernel initialisation.

Figure 4.11, elucidates Figure 4.9 for a filter size of 1, 9 and 99. It is remarkable that the larger datasize takes a shorter time. This effect will be reviewed in Chapter 6, see Figure 6.4.

In Figure 4.12, the speedup of the GPU with the overhead in relation to Algorithm 4 is visualised. This figure describes the expected speedup in a real application. Unfortunately, the instance has to be large enough, i.e. $datasize \cdot filtersize >$ 10.000.000, to gain a speedup up to 80. Bearing in mind Amdahl's law, a speedup of 80 with 114 cores is quite successful.

To prove that the implementations A.3 and A.4 are almost perfect, the CUDA profiler was used. For the profiler output, see Table 4.1. 100% occupancy, enough shared memory and registers, no uncoherent global stores and loads, no local stores and loads, no divergent branches and no divergent warps imply an well-thought implementation with CUDA.

4.6 Summary

In this chapter an algorithm for discrete convolution was transformed into a parallel algorithm, followed by the presentation and evaluation of two attempts to adapt it to the GPU. The result was a speedup of 80, but only for large datasizes.

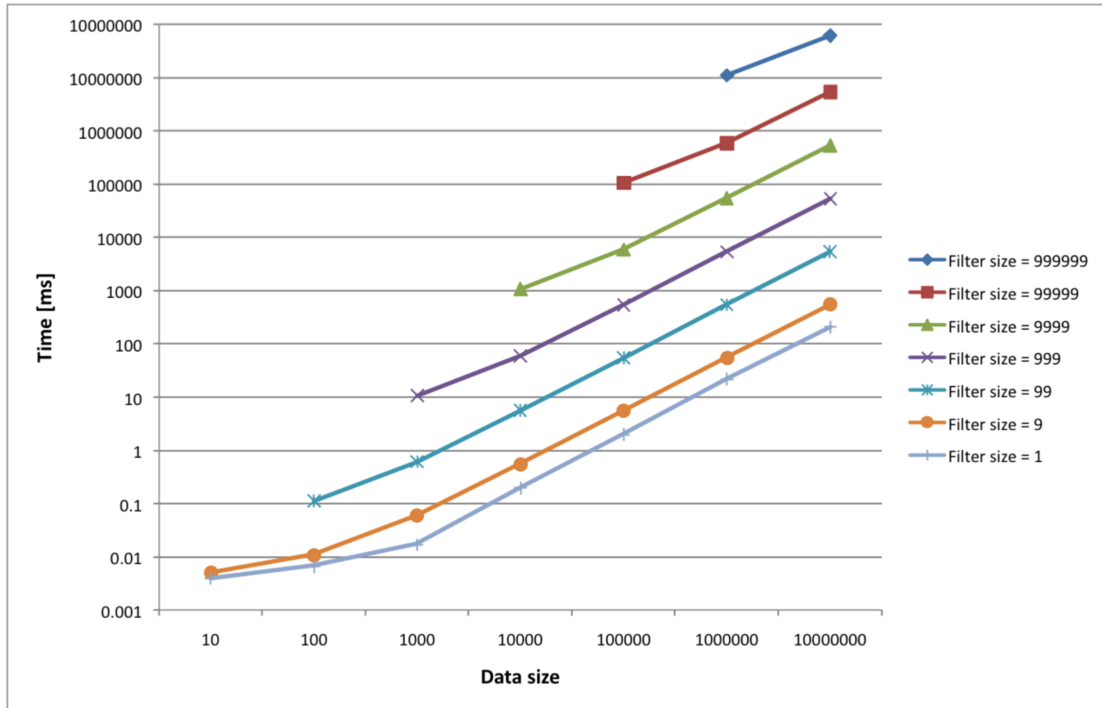


Figure 4.5: Runtime of sequential Algorithm 4 on CPU

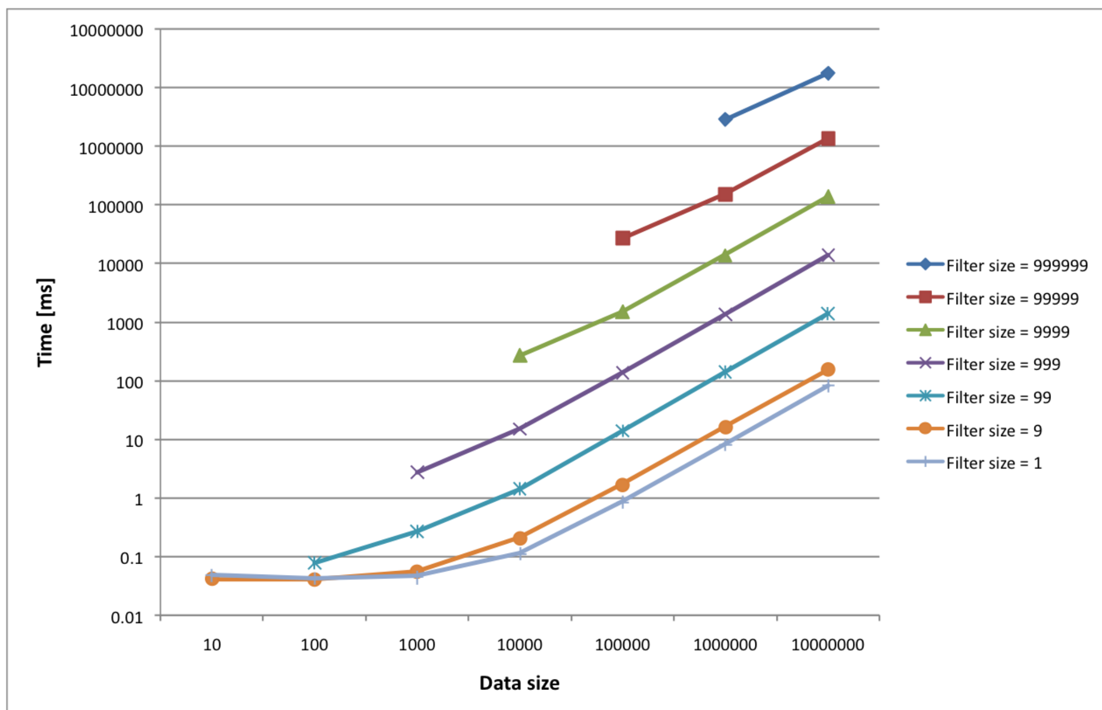


Figure 4.6: Runtime of parallelised Algorithm 5 on CPU

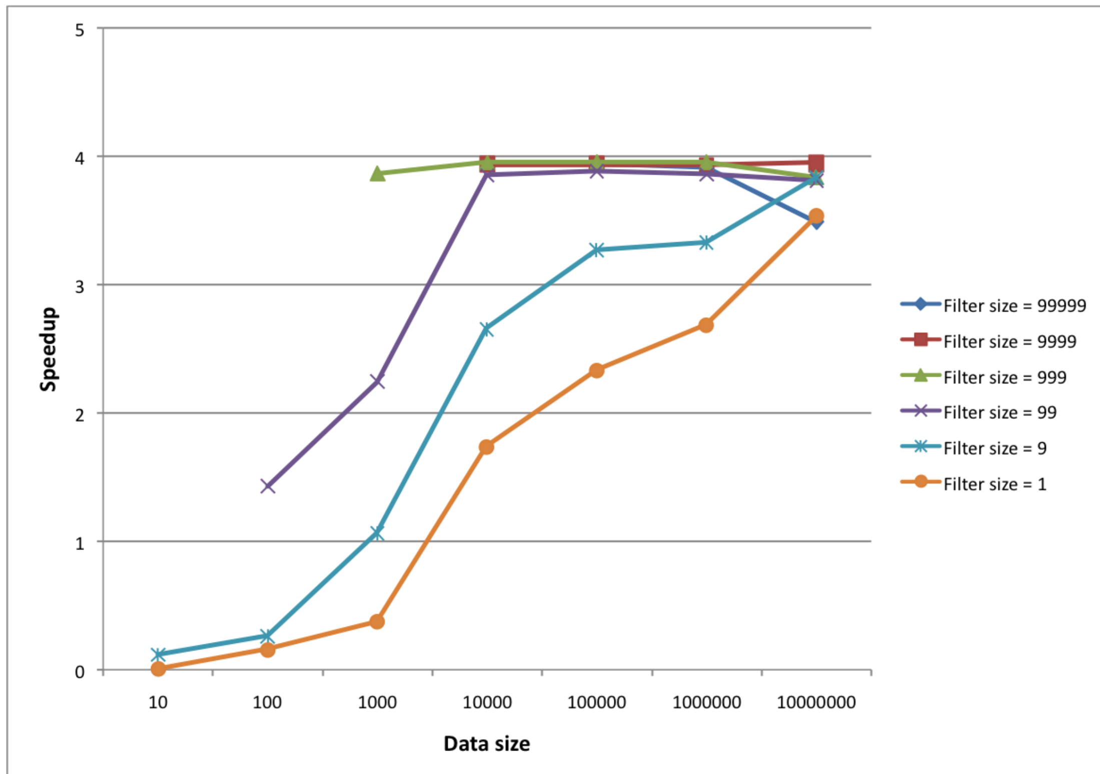


Figure 4.7: Speedup of OpenMP-parallelised Algorithm 5 on Quadcore vs. sequential Algorithm 4

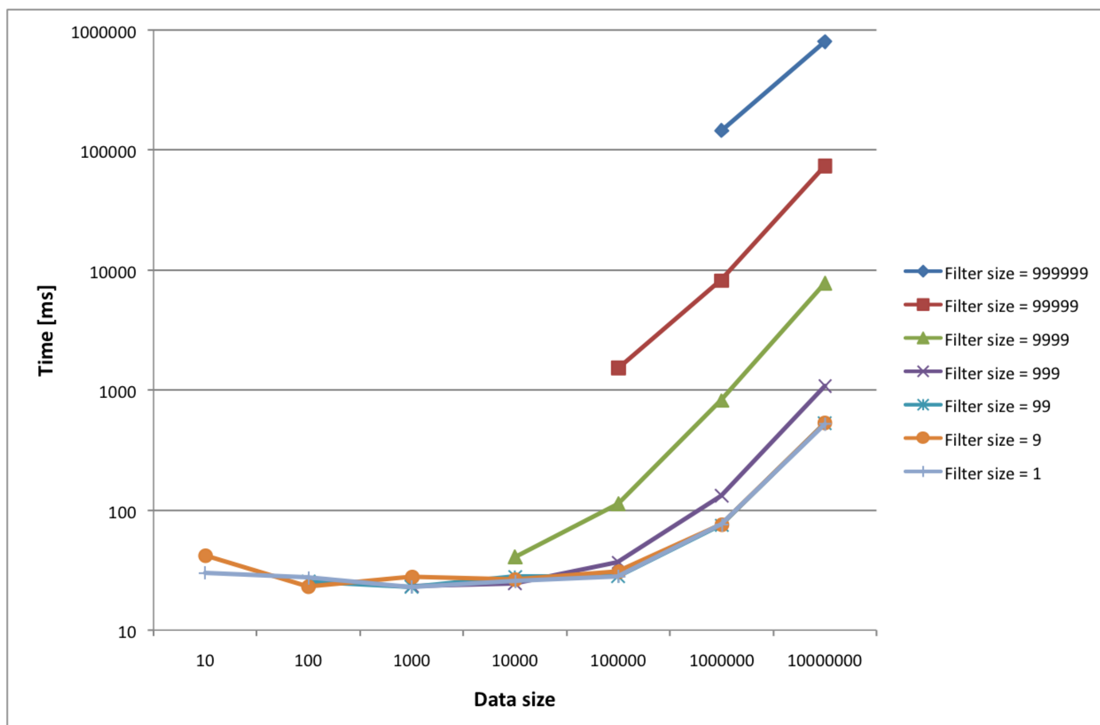


Figure 4.8: GPU Runtime of Algorithm 6 and 7

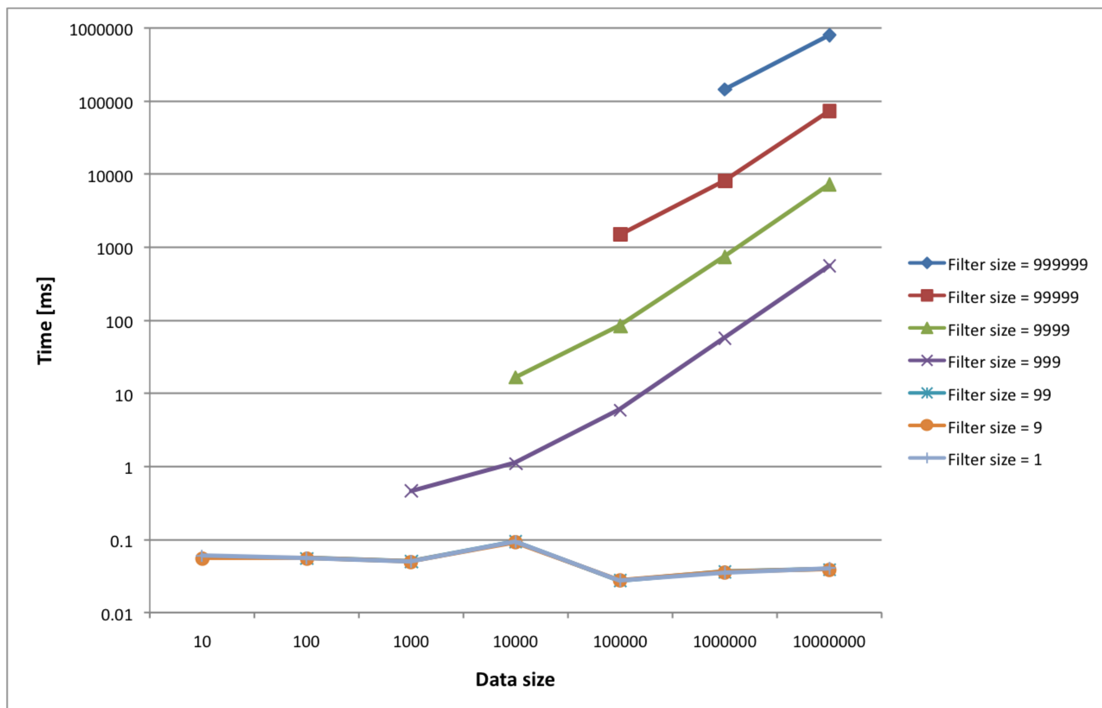


Figure 4.9: Pure GPU runtime of Algorithm 6 and 7

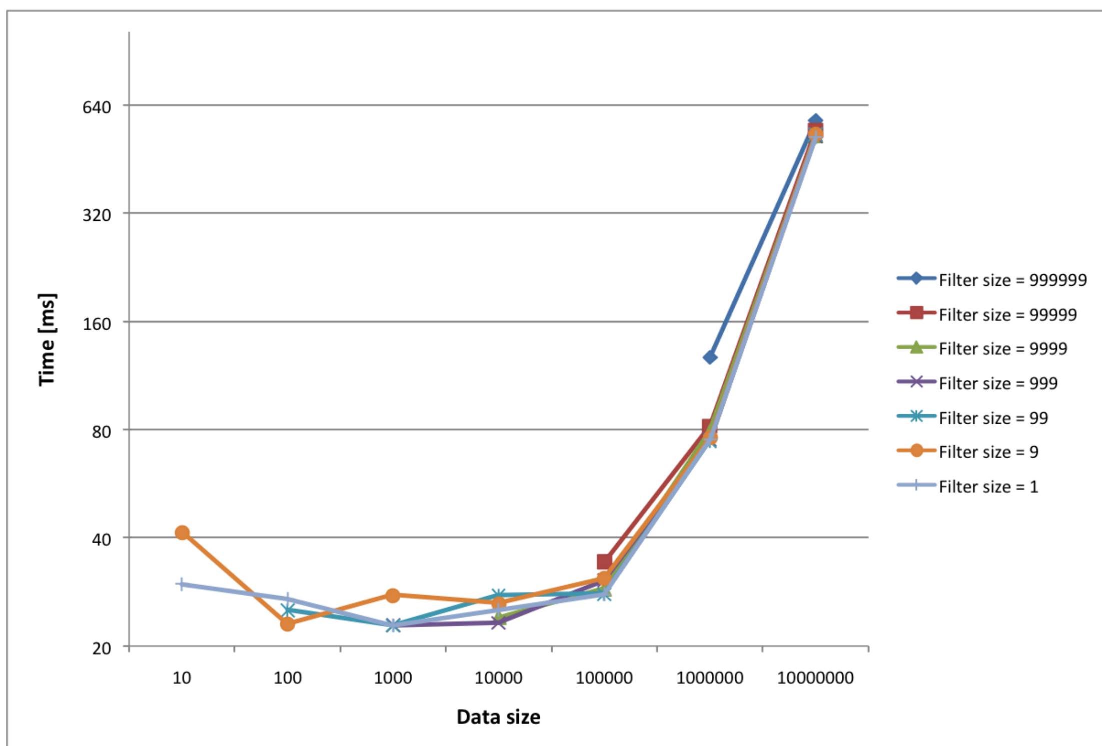


Figure 4.10: Overhead time of GPU, like memory transfer and allocation

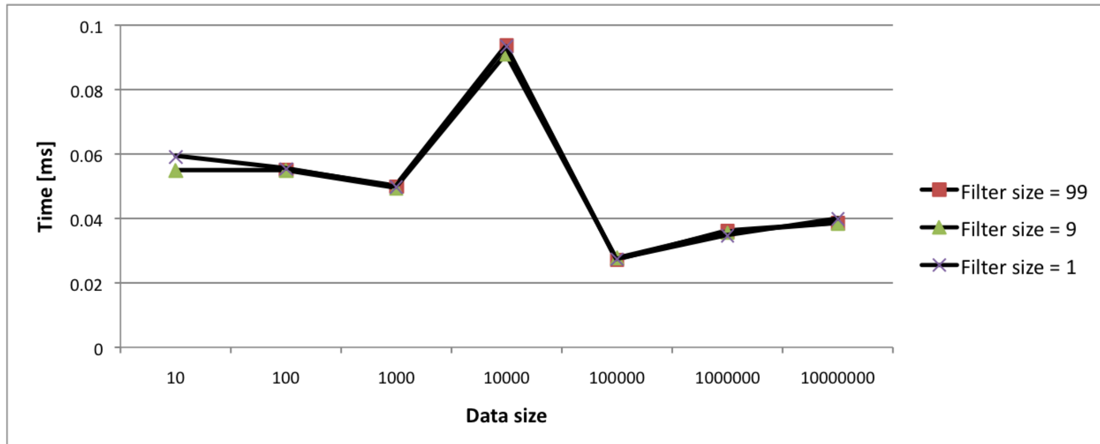


Figure 4.11: Pure GPU runtime, compare Figure 4.9

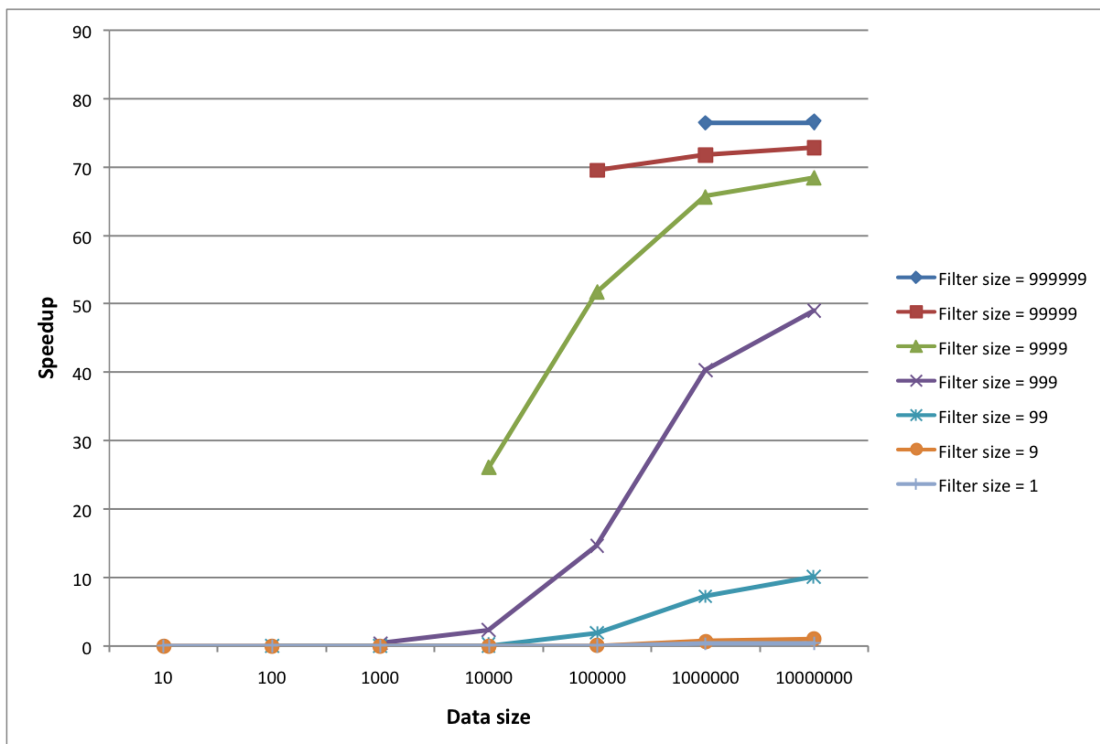


Figure 4.12: Speedup of GPU with overhead vs. CPU

CUDA_PROFILE_LOG_VERSION 1.3
CUDA_PROFILE_CSV 1
CUDA_DEVICE_NAME 0 Quadro FX 3700

timestamp	method	gputime	cputime	occupancy	gridSizeX	gridSizeY	blockSizeX	blockSizeY	blockSizeZ
545.495	memcopy	111.648	351.237						
981.933	memcopy	222.56	309.043						
1315.35	convolutionKernel	2877.34	2907.73	1	1	264	384	1	1
4303.63	convolutionKernel	2878.91	2903.92	1	1	264	384	1	1
7283.47	convolutionKernel	2873.86	2899.07	1	1	264	384	1	1
10254.4	memcopy	76.992	277.224						

dynSmemPerBlock	staSmemPerBlock	registerPerThread	streamID	memTransferSize	memTransferDir	gld_coherent	gst_coherent	branch	divergent_branch	instructions	warp_serialize	cta_launched
				411056	0							
				407064	0							
0	4636	6	0			2736	3648	175788	0	400499	0	38
0	4636	6	0			2736	3648	175788	0	400372	0	38
0	4636	6	0			2736	3648	175788	0	400827	0	38
				403992	1							

Table 4.1: Profiler output for $N = 100.000$, $M = 999$. The first two lines are the memory transfer from host to device and the last line from device to host. Line three to five are three kernel calls ($\lceil \frac{M}{DIM_BLOCK} \rceil = 3$). The numbers are the counters of the profiler.

5. Rolling Ball

In this chapter Rolling Ball (RB, see [DN00]) will be examined, a parallel algorithm will be developed from a sequential algorithm and then be adapted to the CUDA programming model. Finally, the different algorithms will be evaluated.

The RB is "a method for processing [...] measuring values, such as chromatograms" used in chemical laboratories. As "disturbed by an underlying drifting and noisy baseline" it is "difficult to localize the peaks in the chromatogram." RB is a preprocessing first step applying a morphological filter. Here the filter used is a structuring element. The second step, not part of the rolling ball algorithm, is analysis to detect "any peaks corresponding to peaks in said representation of measuring values."

RB is a binary morphological filter operation, called **opening**, which consists of **erosion** and **dilation**. The erosion with one-dimensional data M is definied as

$$M \ominus L = \min_{j \in L}(M(x+j) - L(j)) \qquad , (x+j) \in M$$

and dilation as

$$M \oplus L = \max_{j \in L}(M(x-j) + L(j)) \qquad , (x-j) \in M.$$

The opening operation with a spherical structuring element L

$$M \circ L = M \ominus L \oplus L$$

is called *rolling ball algorithm*. In Figure 5.1 the rolling ball algorithm is visualised.

5.1 Sequential Algorithm

The brute force approach of the RB described in Algorithm 8 is simple but inefficient as its complexity is in $\mathcal{O}(|M| \cdot |L|)$, $|M|$ being the signal length and $|L|$ being the filter width. A concrete C implementation can be found in Appendix A.5. The RB algorithm looks similar to the discrete convolution. It is possible to reduce the discrete convolution to RB with the following transformation:

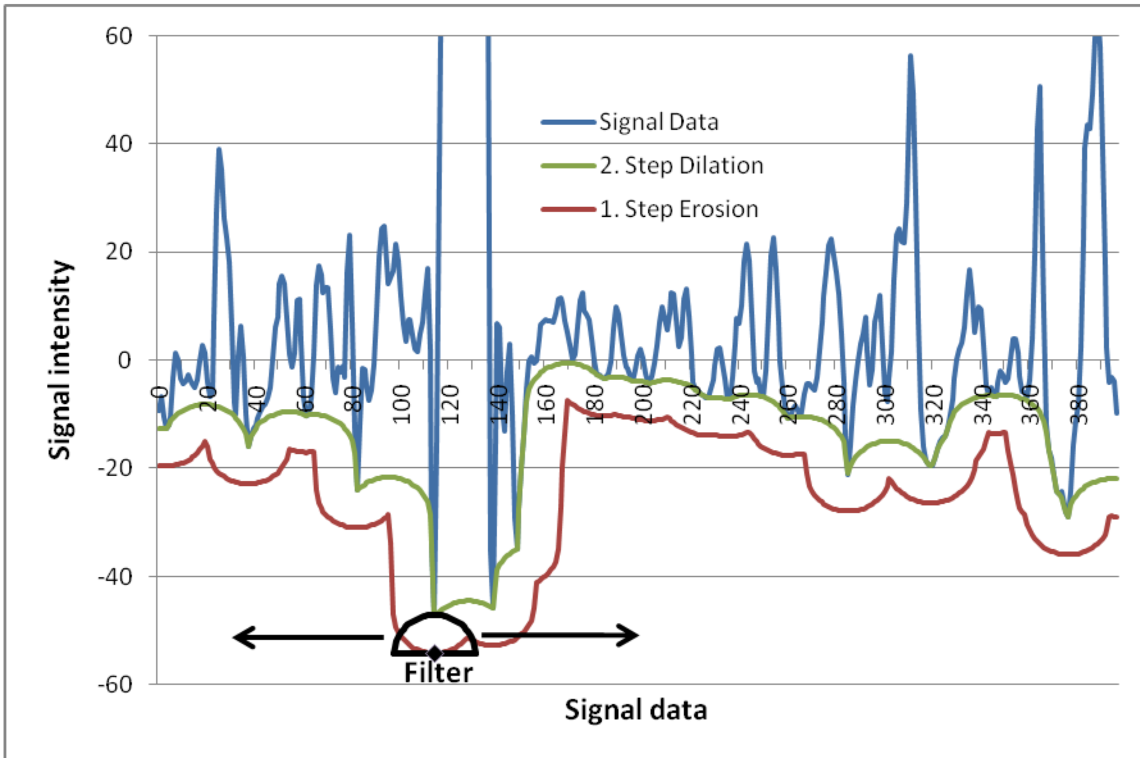


Figure 5.1: Application of Rolling Ball with intermediate results

- Replace “+” by “max”,
- replace “·” by “−” resp. “+” and
- slightly different initialisation and boundary conditions.

In skillfully transforming the transformation above, instead of just coding it, Algorithm 4 was adapted resulting in Algorithm 8, which is visualised in Figure 5.2.

Algorithmus 8: Brute Force Rolling Ball

Input: signal M, filter L w.l.o.g. $|L|$ is odd.
Output: opening $P = M \circ L = M \ominus L \oplus L$

```
// Erosion
1 for n ⟵ 0 to |P| do
2     for k ⟵ 0 to |L| − 1 do
3         P_temp[n] ⟵ min{P_temp[n], L[k] − M[n + k − (|L|−1)/2]};
                  /* let M[i] ⟵ undef., if M[i] is out of range.   */
4     P_temp[n] ⟵ −P_temp[n]
5 P = P_temp
// Dilation
6 for n ⟵ 0 to |P| do
7     for k ⟵ 0 to |L| − 1 do
8         P[n + k − (|L|−1)/2] ⟵ max{P[n + k − (|L|−1)/2], M[n] + L[k]};
                  /* let P[i] ⟵ undef., if P[i] is out of range.   */
9 return P
```

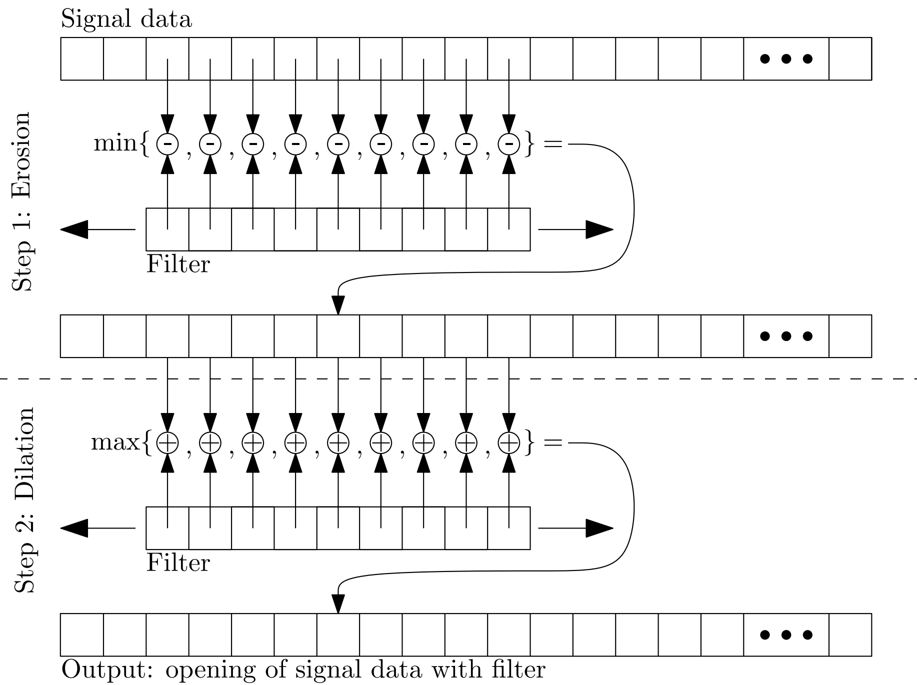


Figure 5.2: Visualisation of Rolling Ball

5.2 Designing the Parallel Algorithm

As mentioned above, it is possible to adapt discrete convolution to RB. The same parallelisation technique used in section 4.2 is applied now. The outer loops receive an OpenMP command to parallelise them, resulting in Algorithm 9. A concrete C implementation can be found in Appendix A.6.
The following section shows the transformation from CPU to the GPU similar to section 4.4.

5.3 Transform the Parallel Algorithm to the GPU

Considering the design principles of CUDA, the transformation from Algorithm 6 & 7 to Algorithm 10 & 11 is performed with precise understanding of the CUDA programming model and the underlying hardware to gain a maximum speedup. This is confirmed by the result of the profiler in Table 5.1. Shared memory for fast computations with the signal data and the result in each threadblock is initialised in Algorithm 10 in lines 1ff . In lines 6f, the signal data needed by each threadblock is copied from global to shared memory. The barrier synchronisation in line 8 assures that the copying operation has finished. Lines 9 - 11 embody the main part of the algorithm, with every thread computing one fo-width output value. To be scalable, the filter is divided into fo-width parts, the result cannot just be written back, but it has to be compared to an earlier intermediate result, see line 12.

This presented algorithm is visualised in Figure 5.3 Step 1a and the related memory hierarchy is visualised in Figure 5.4.

Algorithm 11 calls the device functions until the whole computation is finished. A threadblock consists of 384 threads, as set in line 3 & 9. Thus, a threadblock is fully occupied and enough shared memory is available, even though the filter is copied to constant memory.

Algorithmus 9: Parallelised Brute Force Rolling Ball

Input: signal M, filter L w.l.o.g. $|L|$ is odd.
Output: opening $P = M \circ L = M \ominus L \oplus L$

```
   // Erosion
1  #pragma omp parallel for private (k, sum)
2  for n ⟵ 0 to |P| do
3      sum ⟵ −∞
4      for k ⟵ 0 to |L| − 1 do
5          sum ⟵ min{sum, L[k] − M[n + k − (|L|−1)/2]};
                       /* let M[i] ⟵ undef., if M[i] is out of range. */
6      P_temp[n] ⟵ −sum
7  P = P_temp
   // Dilation
8  #pragma omp parallel for private (k, p)
9  for n ⟵ 0 to |P| do
10     for k ⟵ 0 to |L| − 1 do
11         P[n + k − (|L|−1)/2] ⟵ max{P[n + k − (|L|−1)/2], M[n] + L[k]};
                       /* let P[i] ⟵ undef., if P[i] is out of range. */
12 return P
```

Algorithmus 10: Device Function of Erosion of Rolling Ball

Input: signal M, filter c_N w.l.o.g. $|c_N|$ is odd., filter offset fo
Output: discrete convolution $P = M * N$

```
   // Initialize memory
1  initialise shared memory s_M for signal data with zero ;
2  initialise shared memory s_P for result with −∞;
3  tid ⟵ threadIdx.x                    /* current thread identifier */
4  bid ⟵ blockIdx.x                      /* current block identifier */
5  dim ⟵ blockDim.x                       /* current block dimension */
   // Memory copy on device: global → shared
6  s_M[tid] = M[bid * dim + tid + fo];
7  s_M[tid + dim] = M[(bid + 1) * dim + tid + fo];
                          /* every block gets its dedicated signal data */
8  __syncthreads();
       /* Barrier synchronisation to complete the copying operation */
   // loop in parallel over every computed output value
9  for i ⟵ 0 to dim do
10     s_P[tid] = max{s_P[tid], (c_N[i] − s_M[tid + i])};
11     __syncthreads();
           /* Barrier synchronisation to complete the store operation */
   // write back the result from shared to global memory
12 P[bid * dim + tid] = max{P[bid * dim + tid], s_P[tid]};
13 __syncthreads();
       /* Barrier synchronisation to complete the copying operation */
```

Algorithmus 11: Host Function of Rolling Ball

Input: signal M, filter L w.l.o.g. $|L|$ is odd.
Output: opening $P = M \circ L = M \ominus L \oplus L$

```
   // Erosion
1  for fo ⟵ 0 to fo < |L| do
       // copy current needed part of the filter to constant memory.
2      cudaMemcpyToSymbol("c_N", &N[fo], num_threads · sizeof(float));
       // call the device function for erosion
3      Algorithm 10 <<< gridsize, num_threads >>> (M, fo, P, R);
4      cudaThreadSynchronize();
5      fo ⟵ fo + num_threads;
6  call a device function, which negates P, see Figure 5.3 Step 1b;
   // Dilation
7  for fo ⟵ 0 to fo < |L| do
       // copy current needed part of the filter to constant memory.
8      cudaMemcpyToSymbol("c_N", &N[fo], num_threads · sizeof(float) · 2);
9      call a device function for dilation similar to Algorithm 10, see Figure 5.3 Step
       2.;
10     cudaThreadSynchronize();
11     fo ⟵ fo + num_threads;
12 return P
```

5.4 Performance Evaluation

In this section, the evaluation of Algorithm 8, 9 and 10 & 11 will be presented. Different filter widths from 9 to 999.999 and signal data sets with 10 to 10.000.000 elements have been used. In Appendix C detailed measuring values and other diagrams are attached.

In Figure 5.5 the runtime of the sequential Algorithm 8 on the CPU is shown. The expected $\mathcal{O}(n^3)$ growth without irregularities can be seen.

Figure 5.6 shows the runtime of the parallelised Algorithm 9 on the CPU with the OpenMP library. For small instances, a great overhead can be seen compared to the sequential algorithm. That is as OpenMP needs some time to be loaded.

Figure 5.7 shows the relation between the sequential and the parallel algorithm. For small instances, it is counterproductive to use Algorithm 9 because of its overhead. However, for instances with $datasize \cdot filtersize > 100.000$ the speedup converges to almost four.

Figure 5.8 visualises the runtime of Algorithm 10 and 11 with all the memory transfers from and to the device. In contrast, Figure 5.9 visualises the same without the memory overhead. Small instances need much time.

Figure 5.10 explicitly shows the pure overhead which is never going to deceed below approx. 25ms. This is caused by memory management and the kernel initialisation.

In Figure 5.11 the speedup of the GPU without the overhead in relation to sequential single threaded Algorithm 8 is visualised. Theoretically, a speedup up to 190 in

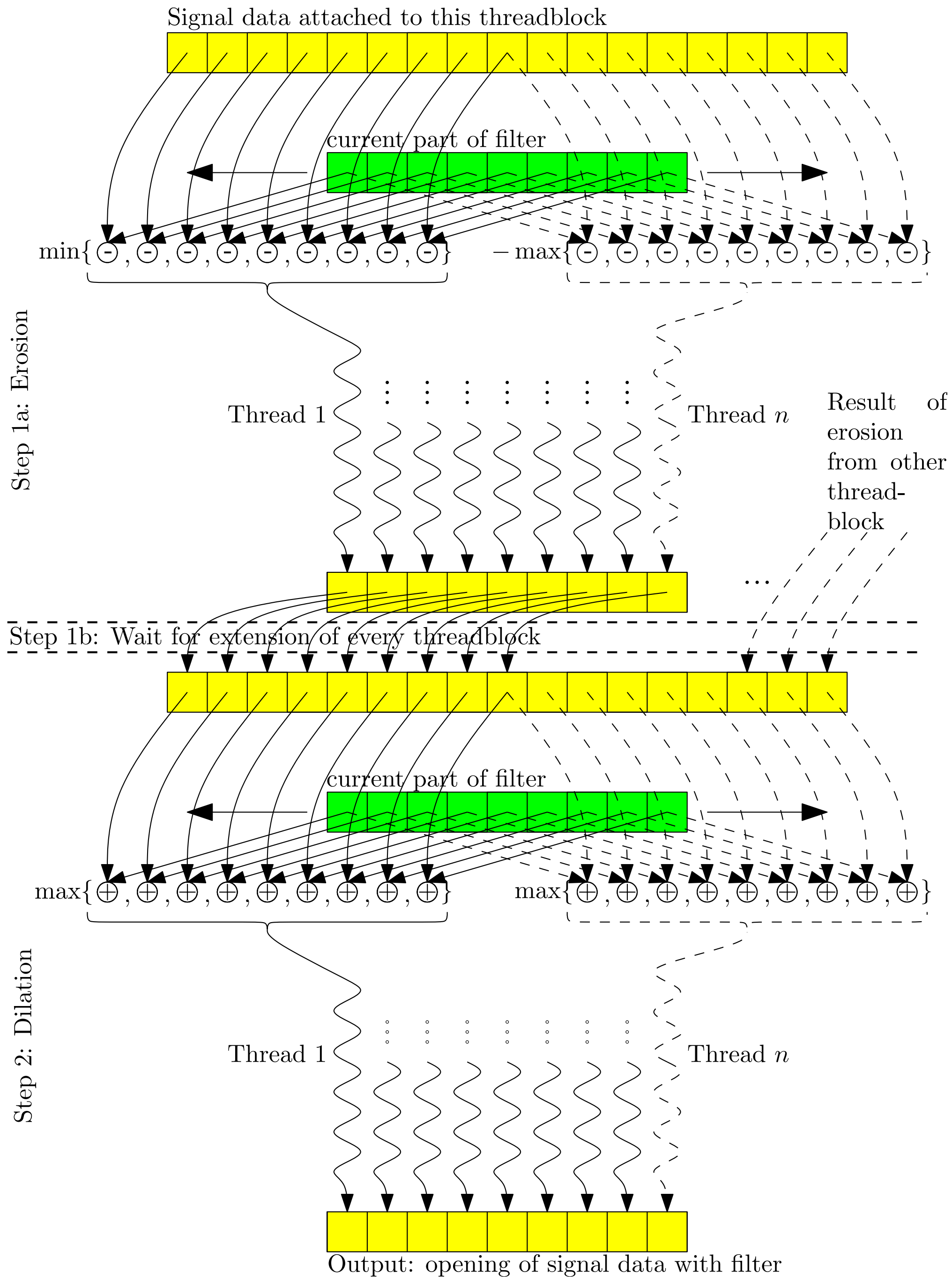


Figure 5.3: Visualisation of parallel Rolling Ball on GPU

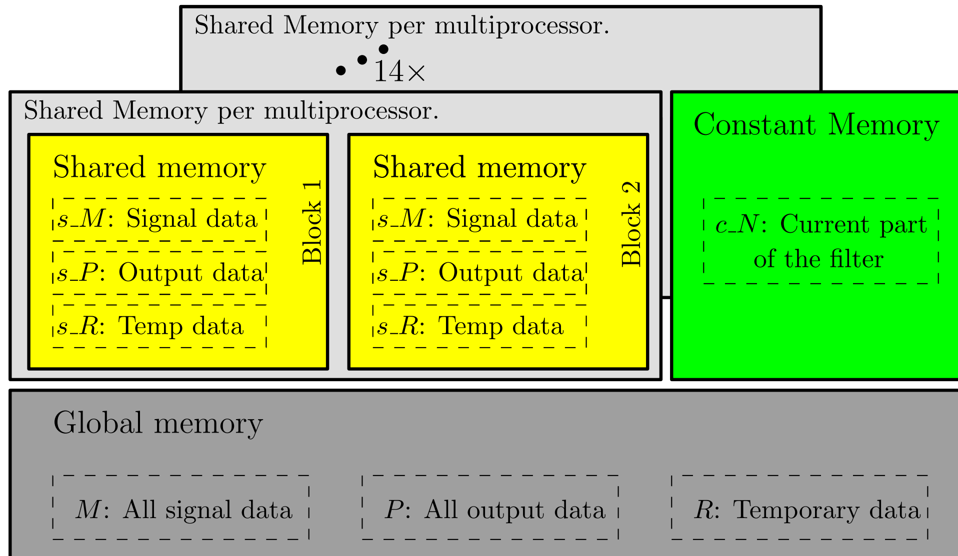


Figure 5.4: Visualisation of memory hierarchy of Algorithm 10 and 11

the main part is possible but not realistic as an increasing overhead relativises the speedup.

In Figure 5.12, the speedup of the GPU with the overhead in relation to Algorithm 9 on a Quadcore CPU is visualised. This figure describes the expected speedup in a real application. The instance has to be large enough, i.e. $datasize \cdot filtersize >$ 100.000.000, to gain a speedup up to 50. A instance in practice is about $datasize \sim$ 100.000 and $filtersize \sim$ 10.000. Bearing in mind Amdahl's law, a speedup of 50 with 114 cores is quite a success.

To prove that the implementations A.7 and A.8 are almost perfect, the CUDA profiler was used. For the profiler output see Tabel 5.1. 100% occupancy, enough shared memory and registers, no uncoherent global stores and loads, no local stores and loads, no divergent branches and no divergent warps imply a well-thought implementation with CUDA.

5.5 Summary

In this chapter the RB method has been introduced and a sequential algorithm has been developed in reducing the discrete convolution from chapter 4 to RB. Similar to discrete convolution a parallel and a GPU version of RB was developed. The evaluation states clearly that for large problems a realistic speedup on the GPU up to 50 can be reached. Even used in a C# application, the speedup can be measured.

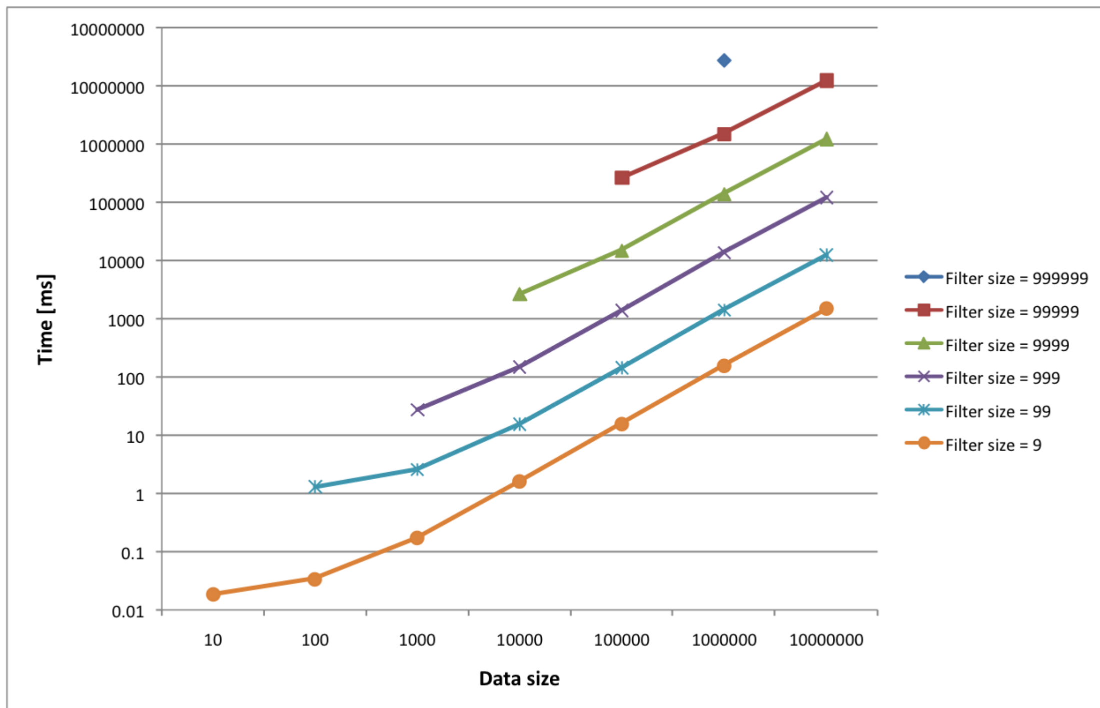


Figure 5.5: Runtime of sequential Algorithm 8 on CPU

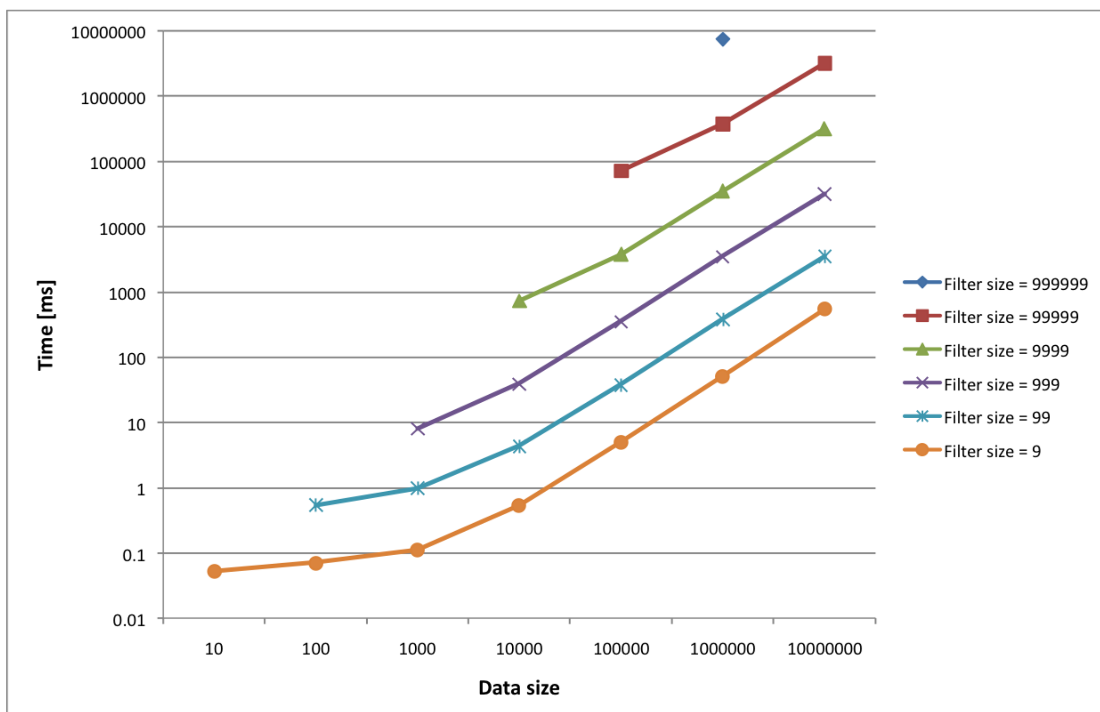


Figure 5.6: Runtime of parallelised Algorithm 9 on CPU

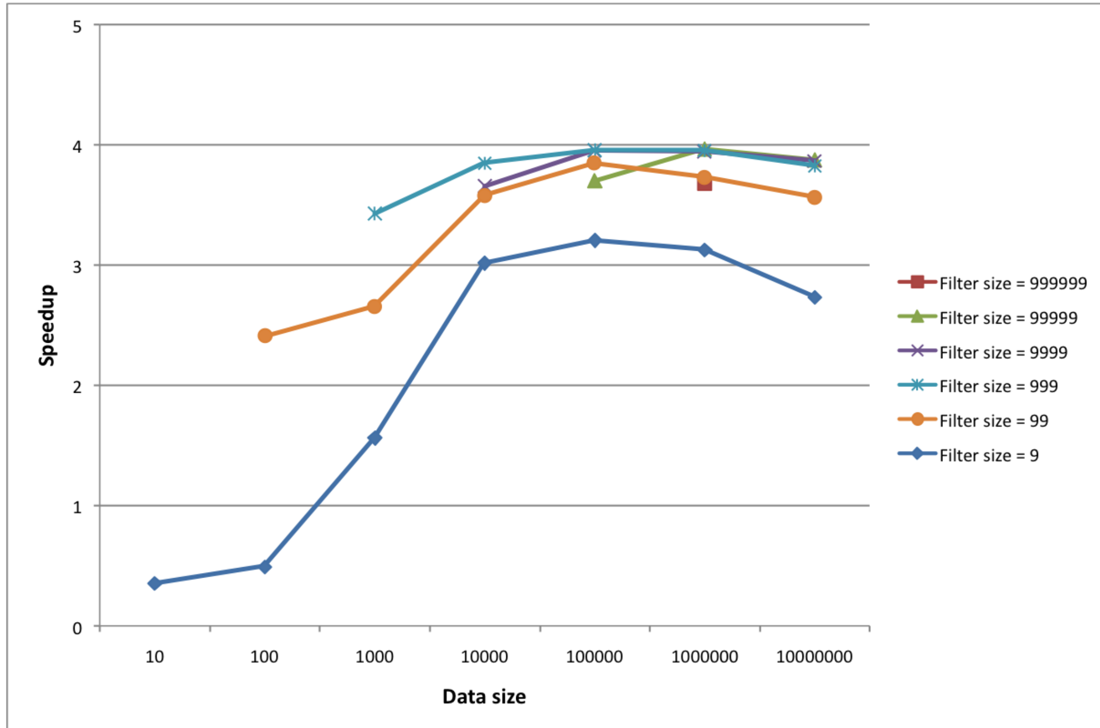


Figure 5.7: Speedup of OpenMP-parallelised Algorithm 9 on Quadcore vs. sequential Algorithm 8

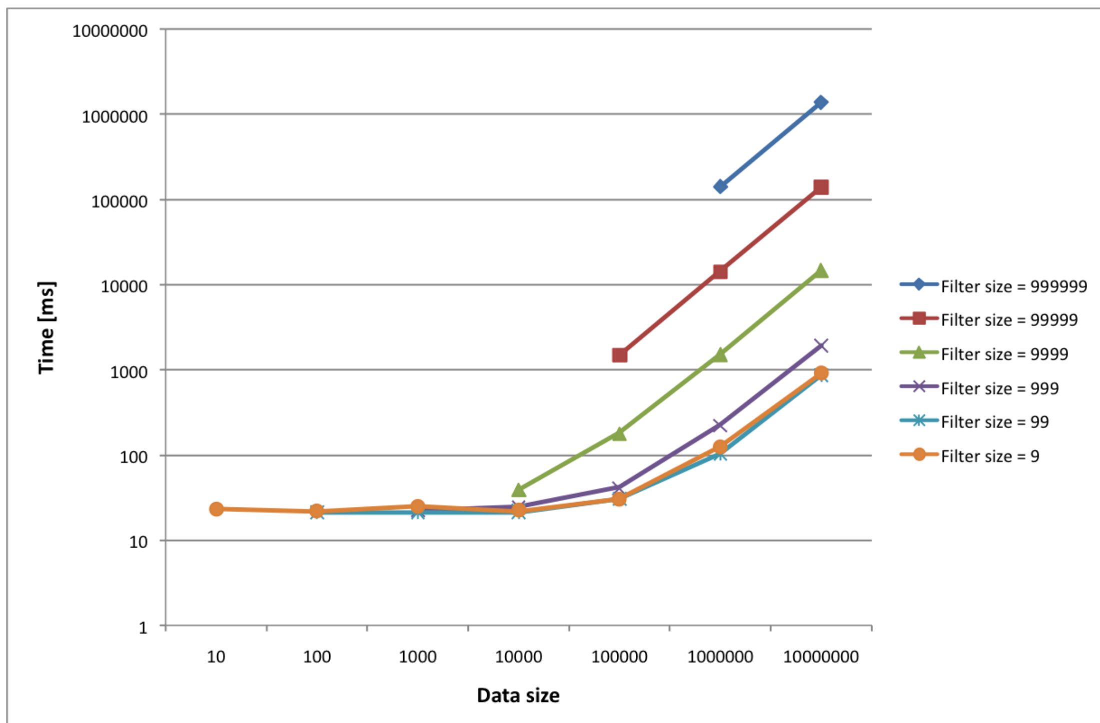


Figure 5.8: GPU Runtime of Algorithm A.7 and A.8

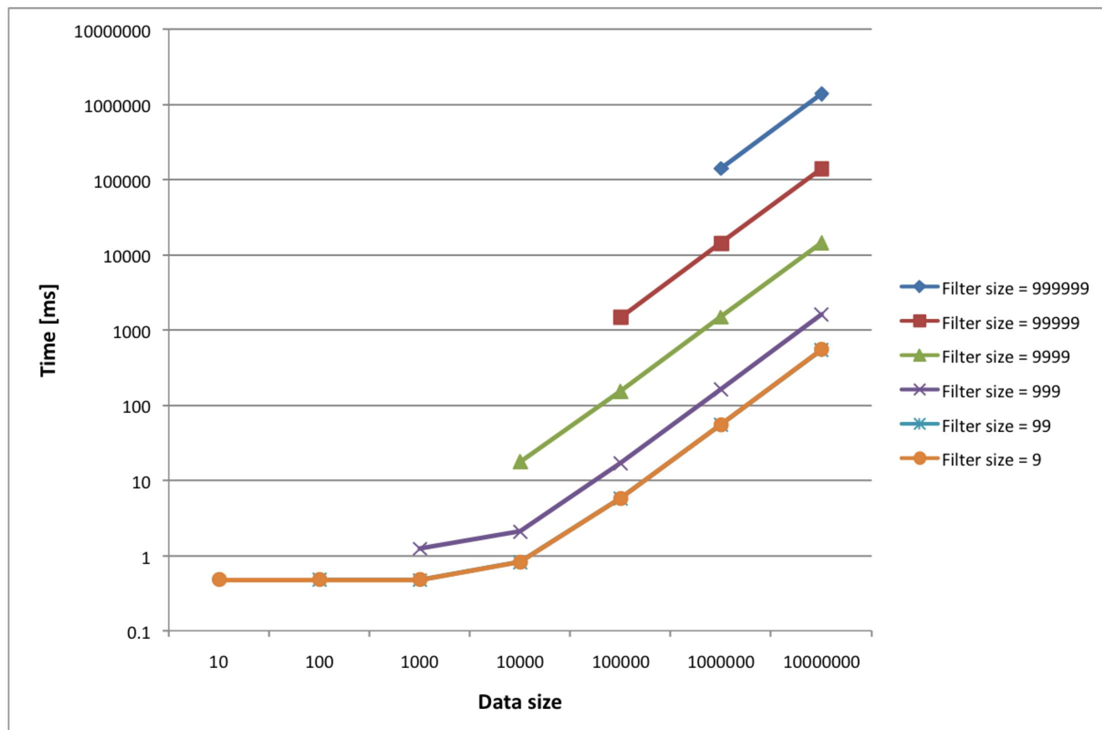


Figure 5.9: Pure GPU runtime of Algorithm A.7 and A.8

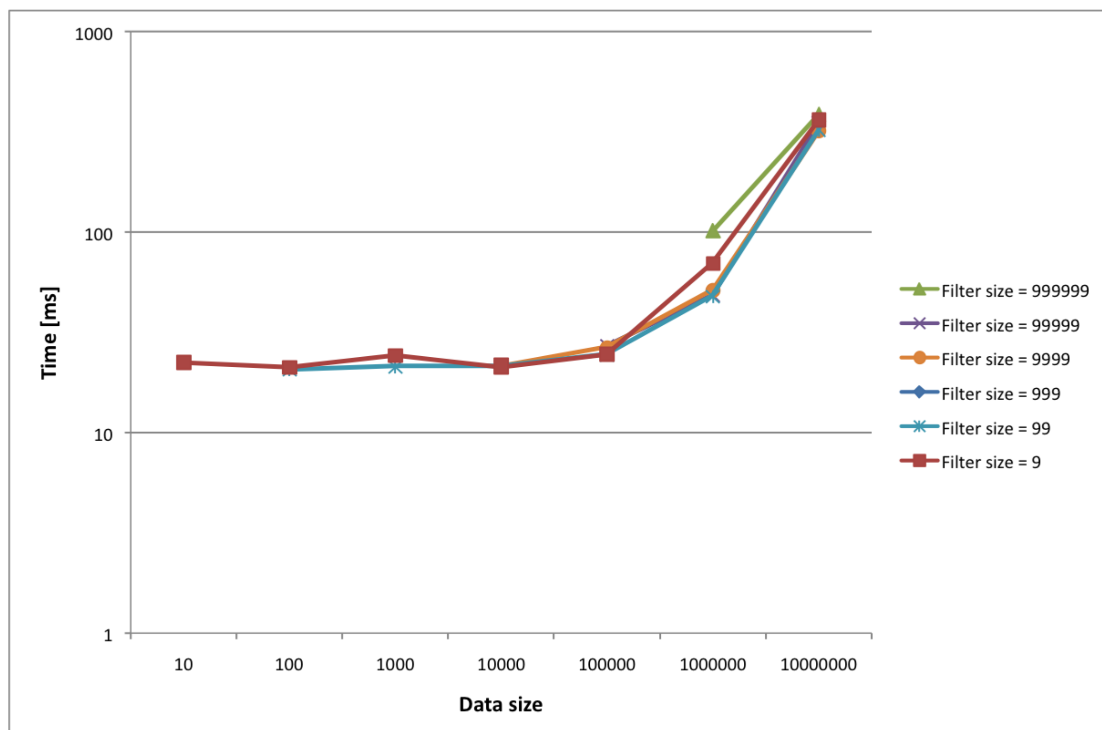


Figure 5.10: Overhead time of GPU, like memory transfer and allocation

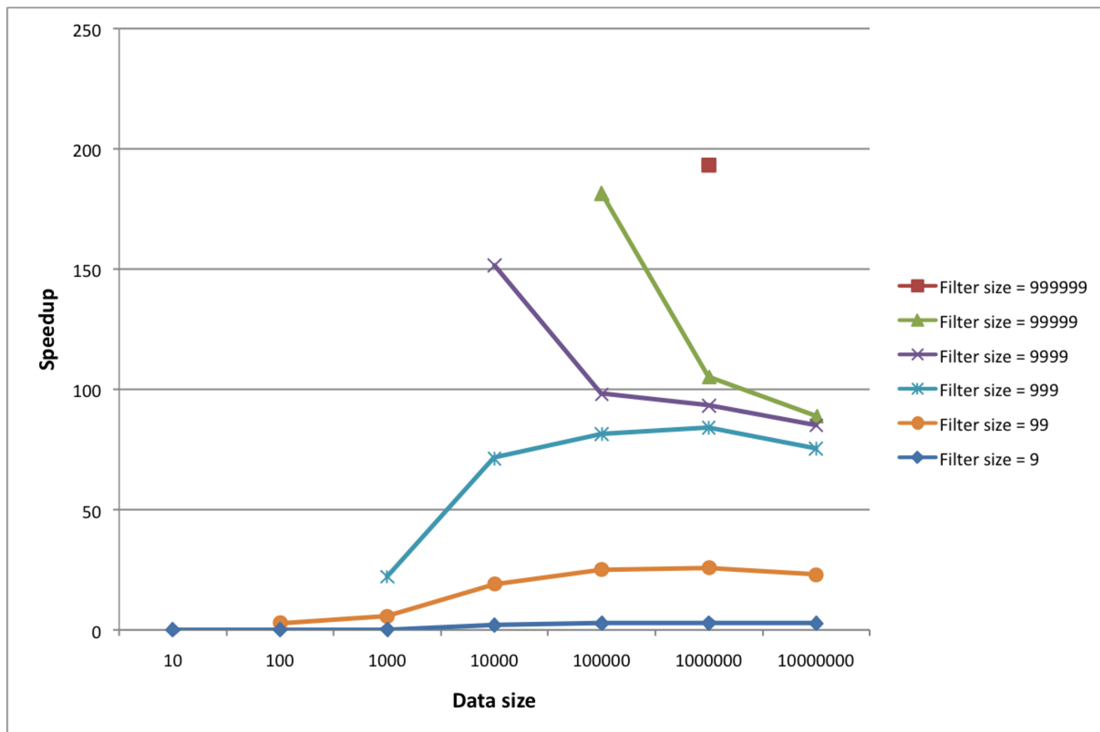


Figure 5.11: Speedup of GPU without overhead vs. CPU

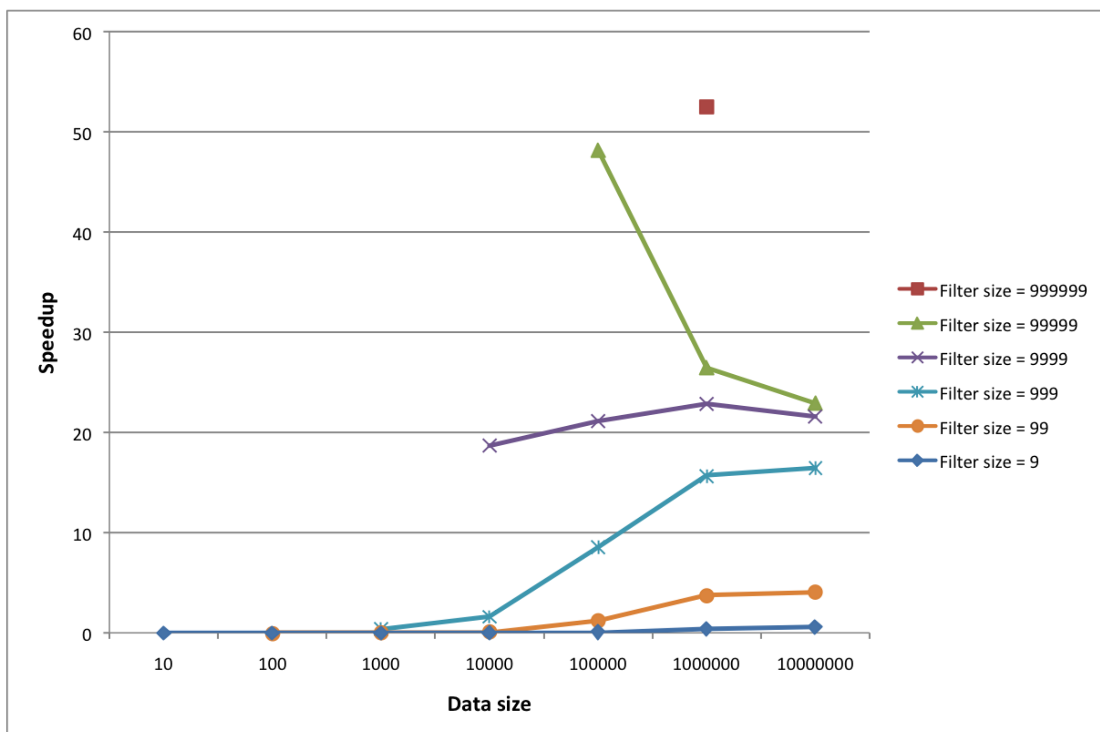


Figure 5.12: Speedup of GPU with overhead vs. Quadcore CPU

CUDA_PROFILE_LOG_VERSION 1.3
CUDA_PROFILE_CSV 1
CUDA_DEVICE_NAME 0 Quadro FX 3700

timestamp	method	gridSizeY	blockSizeX	staSmemPerBlock	registerPerThread	memTransferSize	gld_coherent	gst_coherent	branch	instructions
6320.42	memcopy					407064				
8638.56	memcopy					403072				
9031.92	memcopy					407064				
9716.72	rbKernel	261	384	4640	7		3648	3648	175788	388215
13005.2	rbKernel	261	384	4640	7		3648	3648	175788	386697
16706	rbKernel	261	384	4640	7		3648	3648	175788	386963
19815.2	rbKernel2	261	384	20	2		912	3648	228	1093
20465.9	memcopy					400000				
20613.3	rbKernel3	261	384	4636	7		2664	3552	175560	374566
23478.6	rbKernel3	261	384	4636	7		2664	3552	175560	369858
26338.3	rbKernel3	261	384	4636	7		2736	3648	175560	369299
29189.6	memcopy					400000				

Table 5.1: Profiler output for $N = 100.000$, $M = 999$

6. Limitations of CUDA

In this chapter, the various limitations of the CUDA programming model will be presented. First, the invocation time of kernel functions on the GPU will be determined, second, the bandwidth of memory transactions will be measured, third, the roofline model as a model of performance will be introduced and finally, the floating point issues and other major problems are to be presented.

6.1 Kernel Call Overhead

In the CUDA programming model the GPU, called the device, is used as a coprocessor of the host. As the host can only run distinct functions as kernels on the device, there is a kernel call overhead. To determine the importance of this additional time, the overhead is measured in various ways: Algorithm 12 was used to determine the kernel call overhead by measuring the runtime of Algorithm 12 with and without the kernel call in line 9. Thus, the difference between the two results shows the kernel invocation time without the loop's overhead.

In Figure 6.1 the dimension of the grid $dimGrid(16, 16)$ is fixed. However, the dimension of the threadblocks vary from 1×1 to 22×22. The $emptyKernel$ is called several times from $main$ to realise the initialisation time overhead, which is among others caused by the kernel binary sent to the GPU at the beginning. Depending on the rate the function is called, the overhead per kernel call is between $8\mu s$ for more than thousand consecutive kernel calls and $300\mu s$ for one single kernel call. One single kernel call takes more time as several consecutive kernel calls as the kernel binary has to be transfered onto the device. Thus, the lower bound is $300\mu s$ for an single empty kernel call. It can be seen that the number of threads does not play an important role as long as it is below 512, this beeing the maximum amount of threads per threadblock executed on a MP.

In contrast to Figure 6.1, the dimension of the threadblocks $dimBlock(16, 16)$ in Figure 6.2 is fixed and the dimension of the grid varies from 1×1 to 127×127. Except for a lower bound of $8\mu s$ the runtime grows almost linearly with the size of the grid.

The first time a program using CUDA is executed, there is a minimum initialisation overhead of about $40 - 90ms$, as CUDA has to be initialised and the program has to be loaded from disk. The overhead increases if some shared libraries have to be loaded, too.

Algorithmus 12: Kernel call overhead

```
   // Device function
1  __global__ void emptyKernel()
2  begin
3  |  do nothing;
4  end

   // Host function, calling the device function
5  int main()
6  begin
7  |  start timer;
8  |  for i ⟵ 0 to n do
9  |  |  emptyKernel <<< dimGrid, dimBlock >>> ();
10 |  |  cudaThreadSynchronize();
11 |  stop timer;
12 end
```

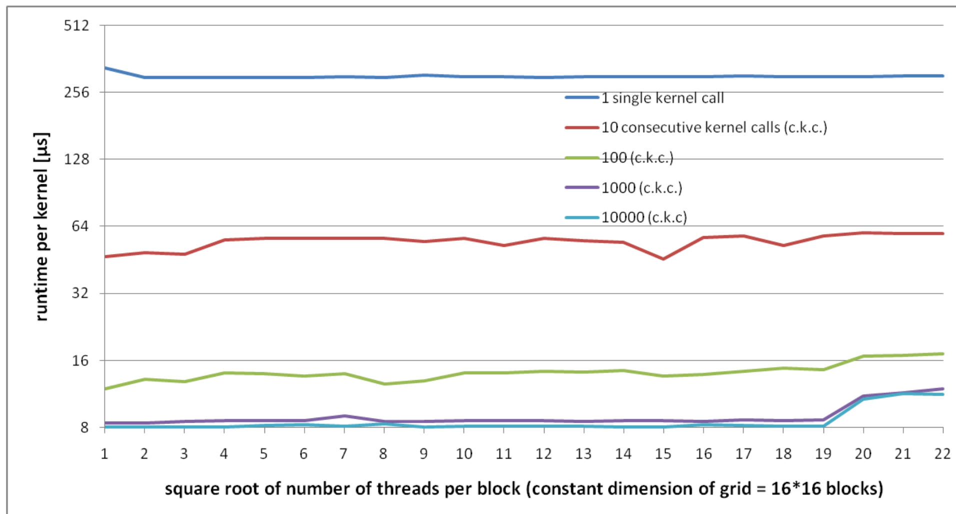


Figure 6.1: Kernel invocation time - constant gridsize.

6.2 Memory Copying Overhead

While running functions on the device, input data is needed and a result computed. It is essential to copy data from and to the GPU as well as to copy data within the GPU. In Figure 6.3 the bandwidth of the three possible transfers is visualised. The bandwidth is determined with the available bandwithTest.exe of the CUDA SDK 2.1 as follows: `bandwidthTest.exe -csv -memory=pinned -mode=range -dtod -start=a -end=b -increment=s`. The value a is the beginning of the measured range, constantly incremented by s until b is reached. The kinks in the plot are caused by an increasing increment s of data in the test. From approx. $1MB$ on, the

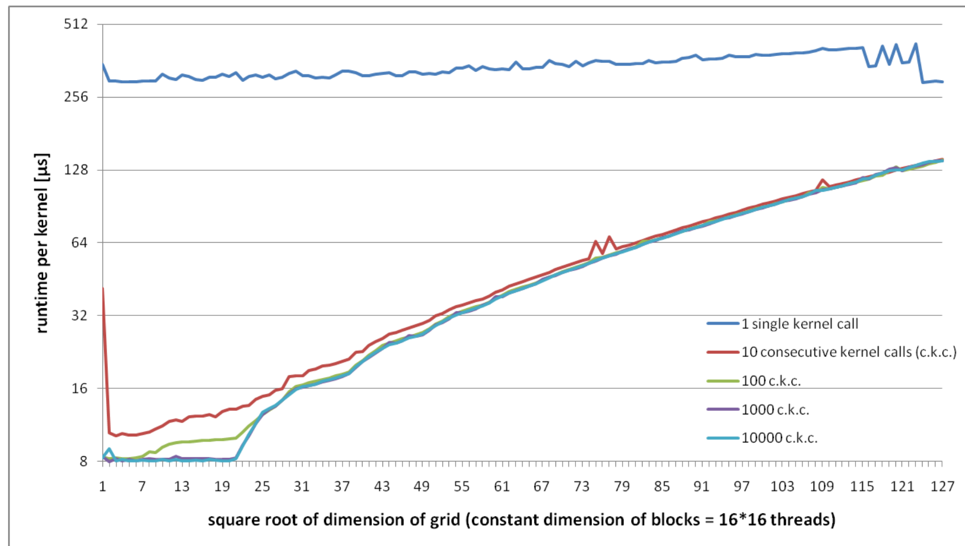


Figure 6.2: Kernel invocation time - constant blocksize.

upper bound bandwidth is reached. The upper bound of host to device and device to host bandwidth of about $4GB/s$ can be a bottleneck in an application.

The time of the memory transfers in Figure 6.4 is directly computed from Figure 6.3:

$$time[s] = \frac{transfered_data[B]}{bandwidth[\frac{MB}{s}] \cdot 1024^2}.$$

Despite irregularities for small sizes of data, the time increases linearly. According to Figure 6.4, it is even faster to copy $4kB - 1500kB$ than less. The Nvidia employee Tim Murray gives the answer to this surprising result, claiming that "it's almost certainly a BIOS issue." Others who did the bandwidth test have a strictly linear growth of time.

6.3 Upper Bound of Performance

In 2009, Patterson and Hennessy presented in [PH09] [WWP09] a new visualisation of a performance model for multi-cores. It is a two-dimensional plot consisting of floating-point performance, arithmetic intensity and memory performance as the diagram from [WP08] for the Nvidia G80 GPU shows in Figure 6.5. With this model, different multi-core architectures are comparable. They compute the performance in

$$Gflop/s = \begin{cases} Peak_Gflop/s \\ StreamingBW \cdot actualflop : DRAM_byte_ratio. \end{cases}$$

Thus, it embodies an upper bound of performance. It is the aim of every programmer to reach this peak performance which, according to Figure 6.5, is not as easy to accomplish as it seems:

- A low floating-point operational part and divergent warps limit the upper bound,
- inefficient memory transfers limit the diagonal stream bandwidth and

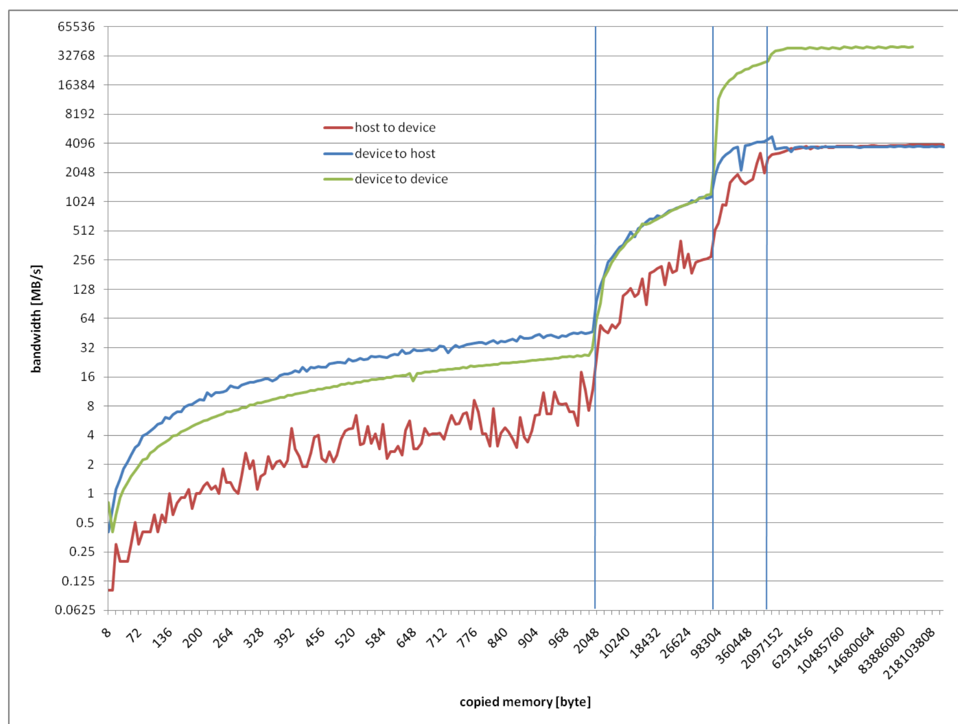


Figure 6.3: Bandwidth of memory transfers from host to device, device to host and device to device. The vertical lines show an increasing increment of transfered data.

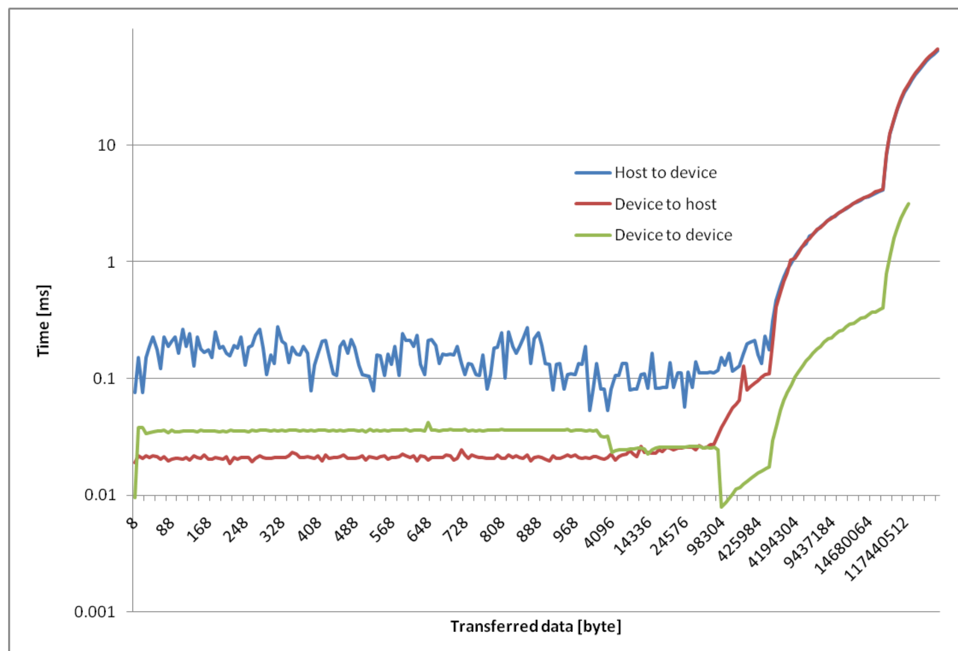


Figure 6.4: Time of memory transfers from host to device, device to host and device to device.

- a too small flop:DRAM byte ratio is the right bound.

Both hardware architecture and the program play the crucial part in the roofline model. An algorithm with a low arithmetic intensity will never unlock the peak bandwidth of the GPU as it is limited by bandwidth.

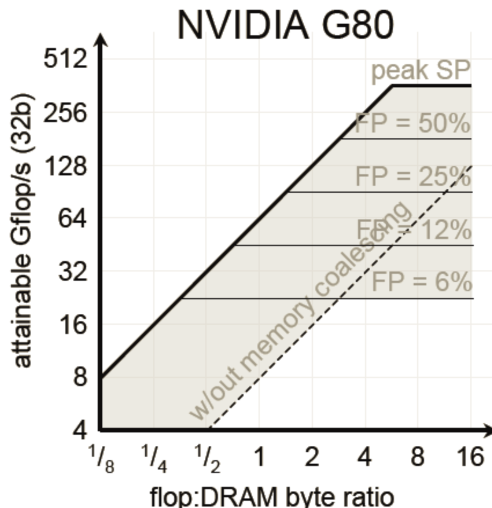


Figure 6.5: Roofline model of Nvidia G80 GPU.

6.4 IEEE-754 Precision

A major disadvantage of CUDA is the incorrectly supported IEEE-754 Floating point standard[iee]. All CUDA-capable GPUs support single precision floating point operations in hardware, but only the newest GPUs with a compute capability 1.3 and higher support double precision floating point operations in hardware. If the programmer wants to use double precision, he has to buy the latest GPU or use software simulated double precision with a bad performance. Additionally, the limitations of CUDA's single precision IEEE-754 are:

- Signalling non-numbers (NaN) and some of the rounding modes are not supported,
- denormals arbitrary get flushed to zero and
- precision of the division and the square root are below the standard.

Bearing in mind these limitations, it is almost impossible to get the same results on the GPU as on the CPU. In the worst case, these errors can lead to cancellation, perhaps when solving a problem hybridly on the CPU and GPU. Computing with CUDA can be useful if programs do not deal with high precision numbers.

6.5 CUDA depends on NVIDIA

CUDA-enabled GPUs are only available from NVIDIA. Thus applications will only run on NVIDIA GPUs. Even the quality of the CUDA programming model depends

on NVIDIA, if they do not develop further their software.
To provide a simple solution the Khronos Group works on the platform independent framework OpenCL [ope]. It can deal with CPUs, GPUs and other procssors from different brands. Similar to OpenGL and OpenAL, OpenCL tries to define an industrial standard for general purpose computing on GPUs. Having a closer look at OpenCL it is more or less the same as CUDA, with other words. But the applications using OpenCL are platform independent. In future releases CUDA will support the OpenCL framework.

6.6 Other Problems

Published in 2007, CUDA is in the early stages of its development and has some more limitations, the major ones being listed below:

- There are no recursion and function pointers in CUDA. Thus recursive algorithms have to be redesigned, if possible.
- Only one kernel a time can be run on the device, so the device functions have to be strictly modular.
- It is not possible to write directly in GPU's memory with DMA, therefore memory transfer time increases.
- The host code is C++ with the device code being subset of C.
- A mode switch of the screen can be critical and crash the GPU.
- Only Microsoft Windows XP, Microsoft Windows Vista, Mac OS X and some Linux operating systems are supported.
- A debugger is only available for Linux, this means an increasing implementation time.
- In Microsoft Windows Vista the profiler does not work properly, as counters are not supported.
- In Microsoft Windows the Timeout Detection and Recovery mechanism, a watchdog, kills kernels calls on GPUs with a display attached after $2 - 5s$. CUDA claims the GPU for its computations and the watchdog handles this as a graphics driver crash. The expensive solution is to buy a second GPU to attach the display there. The other challenging solution is to build scalable below two seconds running kernels.

6.7 Summary

As seen above, the CUDA programming model has major limitations and architecture-related time overhead. To amortise the time overhead, a program should have a high arithmetic intensity meaning much more arithmetic operations than memory operations. The non-standard floating-point implementation forces the program to waste time in software-simulated floating point operations or to accept the inaccuracies and limitations. E.g. the Microsoft Windows Timeout Detection and Recovery mechanism and other mostly software based limitations listed above are annoying as the programmer has to do a workaround, if even possible.

7. Discussion

The discussion following the examination in this work will extend on three fields. First, CUDA-capable GPUs related to the context of its classification in multi-core processors are to be discussed. Second, the consequences of using parallel CUDA programming languages in software engineering will be considered. Finally, we will be dealing with the effort of programming with the CUDA programming model.

7.1 Comparison with Multi-core Processors

On the one hand, a modern high end GPU consists of more than 100 multiprocessors which again consist of eight streaming processors. on the other hand the first modern CPUs like the Intel 80-core[intb] endeavour an increasing number of cores. As a matter of fact, the GPU's purpose is completely different to the CPU one's. This will be considered in the following.

The memory hierarchy of a GPU is completely different from the one of a CPU. The traditional CPU architecture comprises several cache levels and a fast DRAM bandwidth. The GPU does not have a complex cache hierarchy, but local and global memory and is restricted in bandwidth by the mainboard's chipset. Having to manage the memory transactions from global to local memory manually is a big disadvantage of CUDA as it hampers easy programming and may lead to less efficient programs. To support the programmer, future releases of CUDA should accomplish the memory management automatically.

The hardware resources such as the shared memory on the GPU are limited to currently $16kB$ per multiprocessor. Using shared memory for input data, temporary variables and results in order to profit from the fast memory access, only about 4096 floating point numbers can be stored within it. The implication on this limited fast storage is a limitation of the number of concurrently active threads. Only a low-level optimisation of code can improve this hardware limitation up to a certain degree. On CPUs, these problems do not really exist as the cache memory is much bigger and the cache hierarchy provides faster access to data.

The distribution of data on the CPU does not play an important role as it is loaded as needed in the cache hierarchy. Even the cache coherence protocol is fast on

multiprocessors. On the GPU, however, the programmer has to distribute data to the different devices and threadblocks sophisticatedly to gain a high bandwidth with coalesced memory transfers.

On a GPU, the programmer has to execute many threads in parallel to hide the data access latency as thread switches are very cheap compared to a CPU. On the CPU good performance can be often reached with as many threads as cores, whereas on the GPU much more threads than cores are needed. This being an optimistic design principle, *embarrassingly parallel algorithms* need to be considered which cannot provide massive parallelism.

Currently, recent CPUs adopt some features from GPUs such as increasing number of cores and hardware multithreading and at the same time the GPUs get more flexible e.g. more fast memory. Originally, there is a difference between multi-cores and GPUs, but the differences decrease as the two architectures converge more and more together.

7.2 Consequences for Software Engineering

Working on the CUDA programming model and consequently getting in contact with a new style of programming has implied consequences for software engineering which is going to be discussed in the following section.

As seen above, CUDA depends on Nvidia's GPUs. It is not an open standard and therefore not far from beeing supported by any other GPU vendor. One solution to the hardware dependency could be the platform-independent RapidMind[rap] development platform or the newly released OpenCL framework which was already mentioned earlier in this document. With these platform-independent solutions it is possible to build applications runnable on NVIDIA's, ATI's, Intel's, AMD's and many other multiprocessors. As a consequence, general purpose computing on the GPU is not any longer depending on one company which should be a primary goal.

The programming paradigm when using CUDA is very close to hardware which implies that the programmer has to be familiar with hardware internals to transcribe his program to the GPU in an optimal way. This, however, cannot be expected from a software engineer who is normally developing on a high abstraction level. With CUDA, the process of developing a parallel algorithm can be divided into four parts: First, the problem has to be partitioned; second, the programmer has to think about the interprocess communication of the problem; third, he has to agglomerate the tasks of the problem and finally, he has to map the tasks to the CPU resp. GPU. This is too much to be considered by a single programmer who needs to write a fast and correct program.

As seen above, CUDA is limited in many ways. It is, for example, not possible to use recursion, which means that the programmer has to redesign his algorithm as a whole. There are also several limitations to the GPU's hardware such as the slow division or non-standard implementations of the IEEE-754. This means a workaround for the programmer in case he wants to use one of them.

Many algorithms are inherently hard to parallelise. They are called *embarrassingly parallel algorithms*. This means that the software investigation and design phase

will take much longer since every algorithm has to be analysed in the view of data-parallelism. Sometimes, however, the runtime cannot even be improved.

What makes things worse is that there are only few libraries the programmer can use[nvid]. Unfortunately, they are difficult to use and require deep knowledge of the GPU's hardware. As seen above, the libraries are not applicable in all cases: With the amount of input data being too large, library calls have failed.

Apart from this, the CUDA approach is completely new, which means that the programmer has to rethink and restructure his algorithms. This is a great effort as the software engineering primitives of the ancient sixties need to be overcome. Additionally, CUDA does not have any object-oriented techniques, with the available wrappers for higher languages being not even able provide additional object orientation but only pass-through the CUDA commands. Some wrappers are jCUDA for Java, CUDA.NET for the .NET platform and FORTRAN CUDA[gas]. The programmer has to deal with parallelism in particular again.

If CUDA becomes object oriented, software engineers will claim patterns for massive parallel designs usable with CUDA. Perhaps master-worker patterns will gain more performance than fine-granulated in-code parallelisation. Additionally, there might be an asynchronous run-pattern where an event is fired once the computation on the GPU is done. It could also be a good idea to organize data exchange in a kind of parallel queue. Probably, the users of CUDA invent their own patterns, as CUDA has its very own programming paradigms.

The intelligence should move from the programmer to the system. Humans are prone to make errors again and again but a system can learn permanently. As an example, the class should decide if it is worth to compute on the GPU according to the current amount and type of input values. Even the existence, capability and amount of GPUs and the delegation of work to them should be done by the system itself. NVIDIA has not implemented anything in this direction, yet.

During the process of programming, a developer wants to be able to debug his written code. In the CUDA programming model he is faced with the problem as it is for the time beeing only possible to debug within a linux operating system environment. Without debugging, however, it is more difficult to find and identify errors.

Additionally, race conditions and synchronisation errors can occur in parallel programs. Thus, the question is who will use CUDA, as debugging is only possible with linux.

Many software developers are not able to program in parallel at all, as it was neither part of their education nor part of their job challenges. Thus, it is even more unlikely that they will not use a close-to-hardware parallel programming tool like CUDA in the near future. Furthermore, the automatic parallelisation of the code is not realistic at all, as we need the definition of a sequential algorithm and then generate a parallel algorithm. One solution to this dilemma would be adding parallel programming lectures and tutorials to young and old software developer's schedules. They should learn how to deal with each level of parallelisation.

Recent software engineering research claims that parallel programming cannot only be delegated to compilers and libraries which means that new programming tools are needed in the near future. They comprise new programming languages, parallel

design patterns, better search of concurrency and synchronisation errors and new methods of testing.

7.3 CUDA worth the effort

Throughout this work, programming and optimising with CUDA was a challenge to gain maximum speedup. If the problem is likely to be large, potentially data-parallel and not too complex such as discrete convolution in chapter 4 and rolling ball algorithm in chapter 5, it is a good candidate for the CUDA programming model as it fits to the GPU's architecture and the CUDA programming model. Thus, high speedups can be expected.

8. Conclusion & Future Work

In this work, the CUDA programming model has been investigated and the three sample algorithms matrix multiplication, discrete convolution and rolling ball have been implemented. The results are throughout comparable as a speedup of more than 100 is possible, but only for large instances. The CUBLAS library is not easy to use as the programmer has to allocate memory manually and as it is not completely optimised, as small-size problems are having a slow runtime. If the application to be transformed to the GPU is memory intensive, a low speedup is expected caused by memory latency. An advanced algorithm with a complex memory management is a challenge for every experienced programmer even on the CPU, thus, a big speedup is not really realistic.

A kernel with a high arithmetic intensity and low memory transactions is therefore the best candidate for impressive speedups. The problem to solve has to be large enough, to amortise the GPUs overhead and an accurate knowledge of the GPUs hardware architecture is a must to gain runtime benefits.

In all cases, better tools are necessary to specify the runtime structure of the kernels for best performance. Some research on automated optimisations for the GPU architecture should be worked on. A higher level API is needed to simplify programming with CUDA. This API should include high-level data structures managing concurrency, communication and synchronisation. The libraries for CUDA such as CUBLAS are to be analysed, negative aspects should be discovered and consequently performance should be improved. The bandwidth of one GPU may be sufficient but we have to think about big clusters of GPUs, where bandwidth probably appears to be a bottleneck. Finally, double precision and a standard implementation of IEEE-754 floting point numbers should be a short-term goal, using the GPU as a reliable numerical co-processor.

A. Appendix - Source Code

```
/////////////////////////////////////////////////////////////
// computes simple 1D discrete convolution on the CPU
//
// M2 signal data input array, type: float
// N filter data input array, type: float
// M_length length of array M2
// N_length length of array N
//
// P output result array, type: float
// P is of size M_length+N_length-2
/////////////////////////////////////////////////////////////
float* simple_convolution(float* M2, float* N, int M_length, int
    N_length)
{
  //output array
  float* P = (float *) malloc( (M_L+N_L-1)*sizeof(float));
  //initialise output array
  init_array_with_zero(P, M_length+N_length-1);
  float sum=0;
  for (int p=0; p<=M_length+N_length-2; p++)
  {
    sum=0;
    for (int k=0; k<N_length; k++)
    {
      sum+=M2[p+k]*N[N_length-k-1];
    }
    P[p]=sum;
  }
  return P;
}
```

Listing A.1: sequential discrete convolution algorithm

```
1  ////////////////////////////////////////////////////////////////
2  // computes simple 1D discrete convolution on the CPU,
3  // using the OpenMP library for parallel execution
4  //
5  // M2 signal data input array, type: float
6  // N filter data input array, type: float
7  // M_length length of array M2
8  // N_length length of array N
9  //
10 // P output result array, type: float
11 // P is of size M_length+N_length-1
12 ////////////////////////////////////////////////////////////////
13 float* simple_convolution_omp(float* M2, float* N, int M_length,
      int N_length)
14 {
15   //output array
16   float* P = (float *) malloc( (M_L+N_L-1)*sizeof(float));
17   //initialize output array
18   init_array_with_zero(P, M_length+N_length-1);
19   //set the number of threads
20   omp_set_num_threads(4);
21   float sum=0;
22   int k;
23   int p;
24   #pragma omp parallel for private(k,p) reduction(+:sum)
25   for (p=0; p<=M_length+N_length-2; p++)
26   {
27     sum=0;
28     for (k=0; k<N_length; k++)
29     {
30       sum+=M2[p+k]*N[N_length-k-1];
31     }
32     P[p]=sum;
33   }
34   return P;
35 }
```

Listing A.2: OpenMP-parallelised discrete convolution algorithm

```
1  ////////////////////////////////////////////////////////////////
2  // host programm to manage the kernel calls, which compute
3  // 1D discrete convolution on GPU with CUDA
4  //
5  // M signal data input array, type: float
6  // M_length  length of array M
7  // N filter data input array, type: float
8  // N_length  length of array N
9  // P output result array, type: float
10 // P is of size M_length+N_length-1
11 // timer_pure time in ms for the kernel call
12 ////////////////////////////////////////////////////////////////
```

```
13 __host__ void runConvolutionGPU( float* M, int M_length, float* N
     , int N_length, float* P, unsigned int* timer_pure)
14 {
15   //to consider boundary conditions and avoid if-branches use a
        new array M_apron, see below
16   int M_apron_length=M_length+2*(N_length-1);
17   float* M_apron = (float *) malloc( (M_apron_length+
        last_loop_offset)*sizeof(float));
18
19   init_array_with_zero(M_apron, M_apron_length+last_loop_offset);
20   init_array_with_zero(P, M_length+N_length-1+last_loop_offset);
21
22   //initialize signal data with zeros on the right and on the
        left, see above
23   for(int i=0; i<M_length; i++)
24     M_apron[i+N_length-1]=M[i];
25
26   //allocate device memory
27   float* d_M_apron;
28   cutilSafeCall( cudaMalloc( (void**) &d_M_apron, (M_apron_length
        +last_loop_offset)*sizeof(float)));
29
30   //copy host memory to device
31   cutilSafeCall( cudaMemcpy( d_M_apron, M_apron , (M_apron_length
        +last_loop_offset)*sizeof(float), cudaMemcpyHostToDevice) );
32
33   //allocate device memory for result
34   float* d_P;
35   cutilSafeCall( cudaMalloc( (void**) &d_P, (M_length+N_length-1+
        last_loop_offset)*sizeof(float)));
36
37   //copy host memory to device
38   cutilSafeCall( cudaMemcpy( d_P, P , (M_length+N_length-1+
        last_loop_offset)*sizeof(float), cudaMemcpyHostToDevice) );
39
40   //compute execution parameters
41   unsigned int num_blocks = ((M_length+N_length-1)/num_threads)
        +1;
42   //grid configuration
43   dim3 grid( num_blocks, 1, 1);
44   //block configuration
45   dim3 threads( num_threads, 1, 1);
46
47   //start the timer for the pure kernel execution time
48   cutilCheckError( cutStartTimer( *timer_pure));
49
50   //execute the kernel stepwise, as it is divided into parts
51   for(int i=0; i<N_length; i+=num_threads)
52   {
53     //copy current needed part of the filter to fast cached
          constant memory as shared memory is limited and needed for
           other data
```

```
54     cutilSafeCall(cudaMemcpyToSymbol("c_d_N", &N[i], num_threads*
         sizeof(float),0,cudaMemcpyHostToDevice));
55     convolutionKernel<<< grid, threads>>>(d_M_apron, i ,d_P);
56     cutilSafeCall(cudaThreadSynchronize());
57   }
58 
59   //stop the timer for the pure kernel execution time
60   cutilCheckError( cutStopTimer( *timer_pure));
61 
62   //copy result from device to host
63   cutilSafeCall( cudaMemcpy( P, d_P, sizeof( float) * (M_length+
       N_length-1), cudaMemcpyDeviceToHost) );
64 
65   //free the allocated and not anymore needed memory
66   free(M_apron);
67   cutilSafeCall(cudaFree(d_M_apron));
68   cutilSafeCall(cudaFree(d_P));
69 }
```

Listing A.3: CUDA hostcode, discrete convolution algorithm

```
1 ///////////////////////////////////////////////////////////////
2 // kernel, which computes 1D discrete convolution on GPU
3 // with CUDA. Each kernel can handle max. 384 filterelements.
4 //
5 // d_M_apron global signal data input array, type: float
6 // fo is the current offset of the filter beeing used
7 // d_P output data array in global memory
8 // c_d_N part of the filter available in constant memory
9 ///////////////////////////////////////////////////////////////
10 __global__ void convolutionKernel( float* d_M_apron, int fo,
     float* d_P )
11 {
12   //Initialize memory
13   //for signal data in shared memory
14   __shared__ float s_d_M_apron[384*2];
15   //for result in shared memory
16   __shared__ float s_d_P[384];
17   //current thread identifier
18   unsigned int tid=threadIdx.x;
19   //initialise with zero
20   s_d_M_apron[tid]=0;
21   s_d_M_apron[tid+blockDim.x]=0;
22   s_d_P[tid]=0;
23 
24   //memory copy on device: global -> shared
25   //[commented out] using the bank checker makro to detect bank
       conflicts in shared memory
26   /*cutilBankChecker(s_d_M_apron, tid) = d_M_apron[blockIdx.x*
       blockDim.x+tid+fo];
27   /cutilBankChecker(s_d_M_apron, tid+blockDim.x) = d_M_apron[(
       blockIdx.x+1)*blockDim.x+tid+fo];*/
```

```
28   s_d_M_apron[tid]=d_M_apron[blockIdx.x*blockDim.x+tid+fo];
29   s_d_M_apron[tid+blockDim.x]=d_M_apron[(blockIdx.x+1)*blockDim.x
         +tid+fo];
30   __syncthreads();
31 
32   //loop in parallel over every computed output value
33   for(int i=0; i<blockDim.x; i++)
34   {
35     /*cutilBankChecker(s_d_P, tid) = s_d_P[tid]+(s_d_M_apron[tid+
          i]*c_d_N[i]);*/
36     s_d_P[tid]=s_d_P[tid]+(s_d_M_apron[tid+i]*c_d_N[i]);
37     __syncthreads();
38   }
39 
40   //write back the result from shared to global memory
41   /*d_P[blockIdx.x*blockDim.x+tid]+=cutilBankChecker(s_d_P, tid);
         */
42   d_P[blockIdx.x*blockDim.x+tid]+=s_d_P[tid];
43   __syncthreads();
44 }
```

Listing A.4: CUDA devicecode, discrete convolution algorithm

```
1  ////////////////////////////////////////////////////////////
2  // computes rolling ball algorithm on the CPU
3  // rolling ball consists of two steps:
4  // 1. Erosion
5  // 2. Dilation
6  //
7  // M2 signal data input array, type: float
8  // N filter data input array, type: float
9  // M_length length of array M2
10 // N_length length of array N
11 //
12 // R output result array, type: float
13 ////////////////////////////////////////////////////////////
14 float* simple_rolling_ball(float* M2, float* N, int M_length, int
       N_length)
15 {
16   //intermediate result array
17   float* P = (float *) malloc( (M_L+N_L-1)*sizeof(float));
18   //output array
19   float* R = (float *) malloc( (M_L+N_L-1)*sizeof(float));
20   //initialise intermediate result array with infinity
21   init_array_with_inf(P, M_length+N_length-1);
22   //temporary variables
23   float sum=0, temp=0;
24   int p=0, k=0;
25   //minus infinity
26   float infi=log((float)0);
27 
28   //Erosion
```

```
29   for (p=0; p<=M_length+N_length-2; p++)
30   {
31     sum=infi;
32     for (k=0; k<N_length; k++)
33     {
34       //optimisation of: sum=max(sum,N[N_length-k-1]-M2[p+k]);
35       temp=N[N_length-k-1]-M2[p+k];
36       if (temp>sum) sum=temp;
37     }
38     P[p]=-sum;
39   }
40
41   //intermediate step to copy the erosions result in a second
        array
42   for (p=0; p<M_length+N_length-1; p++)
43     R[p]=P[p];
44
45   //Dilation
46   for (p=0; p<=M_length+N_length-2; p++)
47   {
48
49     for (k=0; k<N_length; k++)
50     {
51       //optimisation of: R[p+k]=max(R[p+k],N[N_length-k-1]+P[p]);
52       temp=N[N_length-k-1]+P[p];
53       if (temp>R[p+k]) R[p+k]=temp;
54     }
55   }
56
57   //free the allocated and not anymore needed memory
58   free(P);
59   //return the result and cut off the margin
60   return &R[(N_length-1)];
61 }
```

Listing A.5: sequential rolling ball algorithm

```
1  //////////////////////////////////////////////////////////////
2  // computes rolling ball algorithm on the CPU,
3  // using the OpenMP library for parallel execution.
4  // rolling ball consists of two steps:
5  // 1. Erosion
6  // 2. Dilation
7  //
8  // M2 signal data input array, type: float
9  // N filter data input array, type: float
10 // M_length length of array M2
11 // N_length length of array N
12 //
13 // R output result array, type: float
14 //////////////////////////////////////////////////////////////
```

```
15 float* simple_rolling_ball_omp(float* M2, float* N, int M_length,
      int N_length)
16 {
17   //intermediate result array
18   float* P = (float *) malloc( (M_L+N_L-1)*sizeof(float));
19   //output array
20   float* R = (float *) malloc( (M_L+N_L-1)*sizeof(float));
21   //initialise intermediate result array with infinity
22   init_array_with_inf(P, M_length+N_length-1);
23   //temporary variables
24   float sum;
25   int p=0, k=0;
26   //minus infinity
27   float infi=log((float)0);
28
29   //Erosion
30   #pragma omp parallel for private(k,sum)
31   for (p=0; p<=M_length+N_length-2; p++)
32   {
33     sum=infi;
34     float temp;
35     for (k=0; k<N_length; k++)
36     {
37       //optimisation of: sum=max(sum,N[N_length-k-1]-M2[p+k]);
38       temp=N[N_length-k-1]-M2[p+k];
39       if (temp>sum) sum=temp;
40     }
41     P[p]=-sum;
42   }
43
44   //intermediate step to copy the erosions result in a second
       array
45   //more expensive with an OpenMP Parallel For
46   for (p=0; p<M_length+N_length-1; p++)
47     R[p]=P[p];
48
49   //Dilation
50   #pragma omp parallel for private(k,p)
51   for (p=0; p<=M_length+N_length-2; p++)
52   {
53     float temp;
54     for (k=0; k<N_length; k++)
55     {
56       //optimisation of: R[p+k]=max(R[p+k],N[N_length-k-1]+P[p]);
57       temp=N[N_length-k-1]+P[p];
58       if (temp>R[p+k]) R[p+k]=temp;
59     }
60   }
61
62   //free the allocated and not anymore needed memory
63   free(P);
64   //return the result and cut off the margin
```

```
65   return &R[(N_length-1)];
66 }
```

Listing A.6: OpenMP-parallelised rolling ball algorithm

```
1  //////////////////////////////////////////////////////////////
2  // host programm to manage the kernel calls, which compute
3  // rolling ball algorithm on the CPU.
4  // rolling ball consists of two steps:
5  // 1. Erosion
6  // 2. Dilation
7  //
8  // M signal data input array, type: float
9  // N filter data input array, type: float
10 // M_length length of array M
11 // N_length length of array N
12 // P output result array, type: float
13 // length of array P is of course M_length+offset
14 // timer_pure time in ms for the kernel call
15 //////////////////////////////////////////////////////////////
16 __host__ void runConvolutionGPU( float* M, int M_length, float* N
     , int N_length, float* P, unsigned int* timer_pure)
17 {
18   //to consider boundary conditions and avoid if-branches use a
       new array M_apron, see below
19   int M_apron_length=M_length+(N_length-1);
20   float* M_apron = (float *) malloc( (M_apron_length+
       last_loop_offset)*sizeof(float));
21
22   init_array_with_inf(M_apron, M_apron_length+last_loop_offset);
23   init_array_with_minf(P, M_length+last_loop_offset);
24
25   //initialize signal data with infinity on the right and on the
       left, see above
26   for(int i=0; i<M_length; i++)
27     M_apron[i+(N_length-1)/2]=M[i];
28
29   //allocate device memory
30   float* d_M_apron;
31   cutilSafeCall( cudaMalloc( (void**) &d_M_apron, (M_apron_length
       +last_loop_offset)*sizeof(float)));
32
33   //copy host memory to device
34   cutilSafeCall( cudaMemcpy( d_M_apron, M_apron , (M_apron_length
       +last_loop_offset)*sizeof(float), cudaMemcpyHostToDevice) );
35
36   //temporary array for the dilation method. initialised with -
       ininity.
37   float* R = (float *) malloc((M_length+N_length-1+
       last_loop_offset)*sizeof(float));
38   init_array_with_minf(R, (M_length+N_length-1+last_loop_offset))
       ;
```

```
39
40  //allocate device memory for result
41  float* d_P;
42  float* d_R;
43  cutilSafeCall( cudaMalloc( (void**) &d_P, (M_length+
       last_loop_offset)*sizeof(float)));
44  cutilSafeCall( cudaMalloc( (void**) &d_R, (M_length+N_length-1+
       last_loop_offset)*sizeof(float)));
45
46  //copy host memory to device
47  cutilSafeCall( cudaMemcpy( d_P, P , (M_length+last_loop_offset)
       *sizeof(float), cudaMemcpyHostToDevice) );
48  cutilSafeCall( cudaMemcpy( d_R, R , (M_length+N_length-1+
       last_loop_offset)*sizeof(float), cudaMemcpyHostToDevice) );
49
50  //compute execution parameters
51  unsigned int num_blocks = (M_length/num_threads)+1;
52  //grid configuration
53  dim3  grid( num_blocks, 1, 1);
54  //block configuration
55  dim3  threads( num_threads, 1, 1);
56
57  //start the timer for the pure kernel execution time
58  cutilCheckError( cutStartTimer( *timer_pure));
59
60  //execute the kernel stepwise, as it is divided into parts
61  for(int i=0; i<N_length; i+=num_threads)
62  {
63    //copy current needed part of the filter to fast cached
         constant memory as shared memory is limited and needed for
          other data
64    cutilSafeCall(cudaMemcpyToSymbol("c_d_N", &N[i], num_threads*
         sizeof(float),0,cudaMemcpyHostToDevice));
65    rbKernel<<< grid, threads>>>(d_M_apron, i ,d_P, d_R);
66    cutilSafeCall(cudaThreadSynchronize());
67  }
68  cutilSafeCall(cudaThreadSynchronize());
69
70  //execute a helper kernel, to copy data
71  rbKernel2<<< grid, threads>>>(d_P);
72  cutilSafeCall(cudaThreadSynchronize());
73  //copy d_P in the middle of d_R. d_R is a helper array
74  cutilSafeCall( cudaMemcpy( &d_R[(N_length-1)/2], d_P , M_length
       *sizeof(float), cudaMemcpyDeviceToDevice) );
75  cutilSafeCall(cudaThreadSynchronize());
76
77  //execute the kernel stepwise, as it is divided into parts
78  for(int i=0; i<N_length; i+=num_threads)
79  {
80    //copy current needed part of the filter to fast cached
         constant memory as shared memory is limited and needed for
          other data
```

```
81      cutilSafeCall(cudaMemcpyToSymbol( "c_d_N", &N[i], 2*
            num_threads*sizeof(float),0,cudaMemcpyHostToDevice));
82      rbKernel3<<< grid, threads>>>(d_R, i ,d_P);
83      cutilSafeCall(cudaThreadSynchronize());
84    }
85
86    //stop the timer for the pure kernel execution time
87    cutilCheckError( cutStopTimer( *timer_pure));
88
89    //copy result from device to host
90    cutilSafeCall( cudaMemcpy( P, d_P, sizeof( float) * (M_length),
          cudaMemcpyDeviceToHost) );
91
92    //free the allocated and not anymore needed memory
93    free(M_apron);
94    free(R);
95    cutilSafeCall(cudaFree(d_M_apron));
96    cutilSafeCall(cudaFree(d_P));
97    cutilSafeCall(cudaFree(d_R));
98  }
```

Listing A.7: CUDA hostcode, rolling ball algorithm

```
1   /////////////////////////////////////////////////////////////
2   // Erosion kernel, which computes erosion of the rolling
3   // ball algorithm on GPU with CUDA.
4   // Each kernel can handle max. 384 filterelements.
5   //
6   // d_M_apron global signal data input array, type: float
7   // fo is the current offset of the filter beeing used
8   // d_P output data array in global memory
9   // c_d_N part of the filter available in constant memory
10  /////////////////////////////////////////////////////////////
11  __global__ void rbKernel( float* d_M_apron, int fo, float* d_P,
       float* d_R )
12  {
13    //Initialize memory
14    //for signal data in shared memory
15    __shared__ float s_d_M_apron[384*2];
16    //for result in shared memory
17    __shared__ float s_d_P[384];
18    //current thread identifier
19    unsigned int tid=threadIdx.x;
20
21    //memory copy on device global -> shared
22    s_d_P[tid]=d_R[tid];  //inititlize with minus infinity
23    s_d_M_apron[tid]=d_M_apron[blockIdx.x*blockDim.x+tid+fo];
24    s_d_M_apron[tid+blockDim.x]=d_M_apron[(blockIdx.x+1)*blockDim.x
         +tid+fo];
25    __syncthreads();
26
27    //loop in parallel over every computed output value
```

```
28   for(int i=0; i<blockDim.x; i++)
29   {
30     s_d_P[tid]=max(s_d_P[tid],(c_d_N[i]-s_d_M_apron[tid+i]));
31     __syncthreads();
32   }
33
34   //write back the result from shared to global memory
35   d_P[blockIdx.x*blockDim.x+tid]=max(d_P[blockIdx.x*blockDim.x+
        tid],s_d_P[tid]);
36   __syncthreads();
37 }
38
39 ////////////////////////////////////////////////////////////
40 // Helper kernel. Inverts an array of type float
41 // d_P is pointer to the data of type float in device memory
42 ////////////////////////////////////////////////////////////
43 __global__ void rbKernel2(float* d_P)
44 {
45   unsigned int id;
46   //current global thread identifier
47   id=blockIdx.x*blockDim.x+threadIdx.x;
48   d_P[id]=-d_P[id];
49   __syncthreads();
50 }
51
52 ////////////////////////////////////////////////////////////
53 // Dilation kernel, which computes dilation of the rolling
54 // ball algorithm on GPU with CUDA.
55 // Each kernel can handle max. 384 filterelements.
56 //
57 // d_R global signal data array, type: float
58 // fo is the current offset of the filter beeing used
59 // d_P output data array of type float in global memory
60 // c_d_N part of the filter available in constant memory
61 ////////////////////////////////////////////////////////////
62 __global__ void rbKernel3( float* d_R, int fo, float* d_P)
63 {
64   //current thread identifier
65   unsigned int tid=threadIdx.x;
66   //for signal data in shared memory
67   __shared__ float s_d_P[384];
68   //for temp signal data in shared memory
69   __shared__ float s_d_R[384*2];
70
71   //Memory copy on device global -> shared
72   s_d_P[tid]=d_P[blockIdx.x*blockDim.x+tid];
73   s_d_R[tid]=d_R[blockIdx.x*blockDim.x+tid+fo];
74   s_d_R[tid+blockDim.x]=d_R[(blockIdx.x+1)*blockDim.x+tid+fo];
75   __syncthreads();
76
77   //loop in parallel over every computed output value
78   for (int i=0; i<384; i++)
```

```
79    {
80      s_d_P[tid]=max(s_d_P[tid],(c_d_N[i]+s_d_R[i+tid]));
81      __syncthreads();
82    }
83
84    //write back the result from shared to global memory
85    d_P[blockIdx.x*blockDim.x+tid]=s_d_P[tid];
86    __syncthreads();
87  }
```

Listing A.8: CUDA devicecode, rolling ball algorithm

B. Appendix - Additional Runtime Measurements

The following examinations of algorithms will be performed on a HP xw4600 Workstation, which is equipped with an Intel Core 2 Duo E6850 running at 3,00GHz, with 4GB Random Access Memory (RAM) and a Nvidia Quadro FX 1700 GPU with 512MB RAM. The GPU has 4 SMs, that implies 32 SPs. Microsoft Windows Vista Business 64Bit with Service Pack 3 is used as the operating system.

Performance Evaluation

In this appendix, the evaluation of Algorithm 8, 9 and 10 & 11 will be presented. Different filter widths from 9 to 99.999 and signal data sets with 10 to 1.000.000 elements have been used.

In Figure B.1 the runtime of the sequential Algorithm 8 on the CPU is shown. $\mathcal{O}(n^3)$ growth without irregularities.

Figure B.2 shows the runtime of the parallelised Algorithm 9 on the CPU with the OpenMP library.

Figure B.3 shows the relation between the sequential and the parallel algorithm. For small instances, it is counterproductive to use Algorithm 9 because of its overhead. However, for instances with $datasize \cdot filtersize > 100.000$ the speedup is about 1.75.

Figure B.4 visualises the runtime of Algorithm 10 and 11 with all the memory transfers from and to the device.

In contrast, Figure B.5 visualises the same without the memory overhead.

Figure B.6 explicitly shows the overhead which is never going to deceed below approx. 20ms.

In Figure B.7 the speedup of the GPU without the overhead in relation to sequential single threaded Algorithm 8 is visualised. Theoretically, a speedup up to 35 in

the main part is possible but not realistic as an increasing overhead relativises the speedup.

In Figure B.8, the speedup of the GPU with the overhead in relation to Algorithm 9 on a Dualcore CPU is visualised. This figure describes the expected speedup in a real application. The instance has to be large enough, i.e. $datasize \cdot filtersize >$ 100.000.000, to gain a speedup up to 20. A instance in practice is about $datasize \sim$ 100.000 and $filtersize \sim 10.000$. Bearing in mind Amdahl's law, a speedup of 20 with 32 cores is quite a success.

In Figure B.9 the speedup of Nvidia FX 3700 vs. FX 1700 according to Algorithm A.7 and A.8 is visualised. As the clock rate of the FX 3700 is $1.24GHz$ and it has 14 SMs and the FX 1700 has a clock rate of $0.92GHz$ and 4 SMs the expected speedup is 4.7. The reached speedup is 4.6. Thus the CUDA program scales on both GPUs linearly.

According to Section 6.2 the bandwidth test results are visualised in Figure B.10. From approx. $1MB$ on, the upper bound bandwidth is reached. The upper bound of host to device bandwidth of about $2.5GB/s$ and device to host bandwidth of about $2.9GB/s$ can be a bottleneck in an application. The upper bound for device intern bandwidth is about $9.5GB/s$. In Figure B.11 the time of copying the data according to Figure B.10 is visualised.

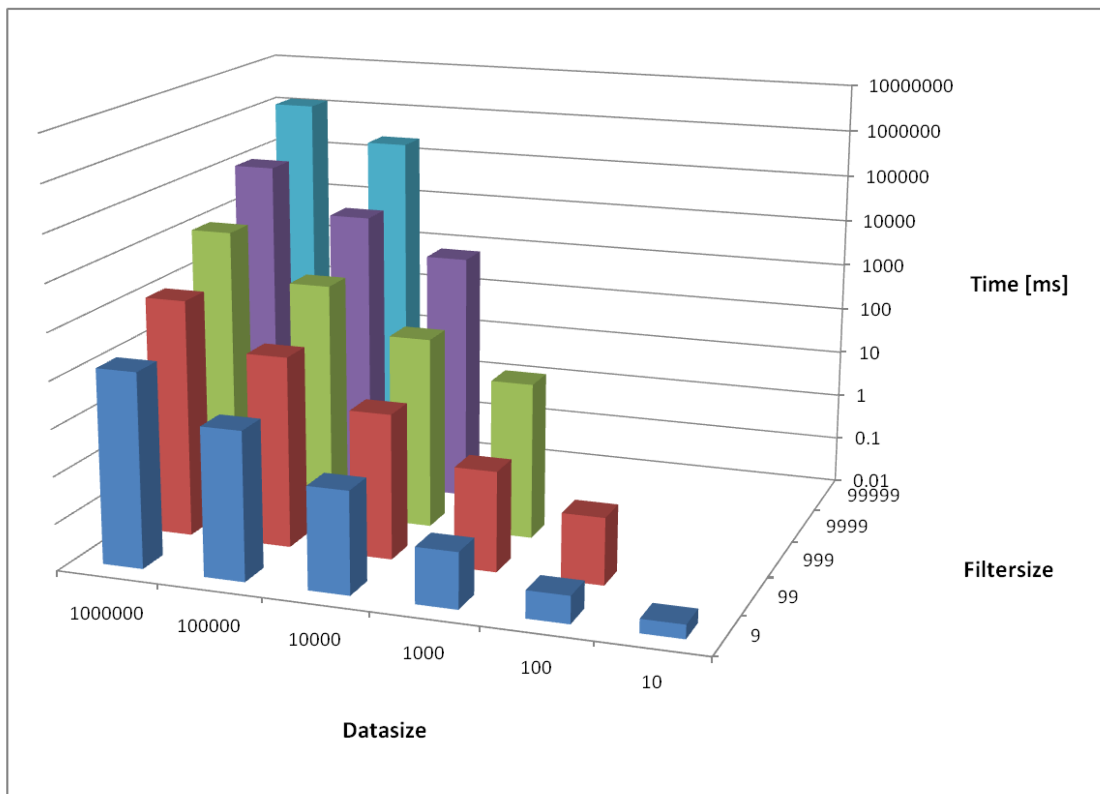


Figure B.1: Runtime of sequential Algorithm 8 on CPU

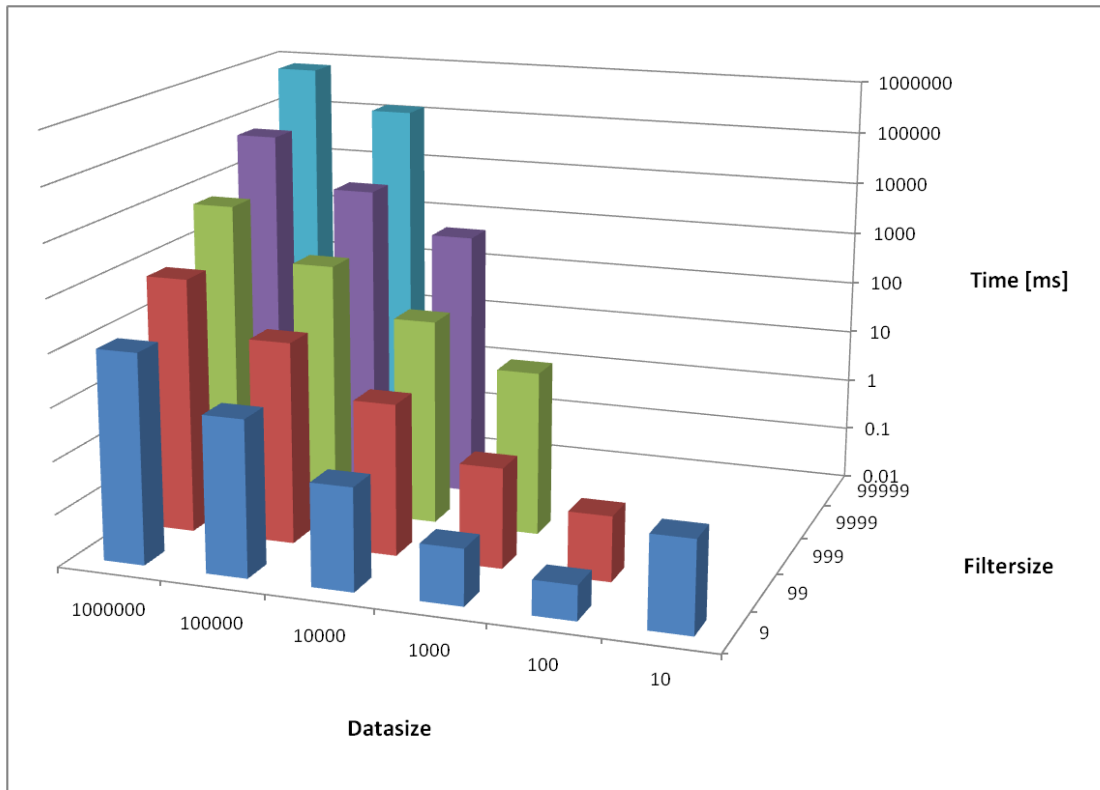


Figure B.2: Runtime of parallelised Algorithm 9 on CPU

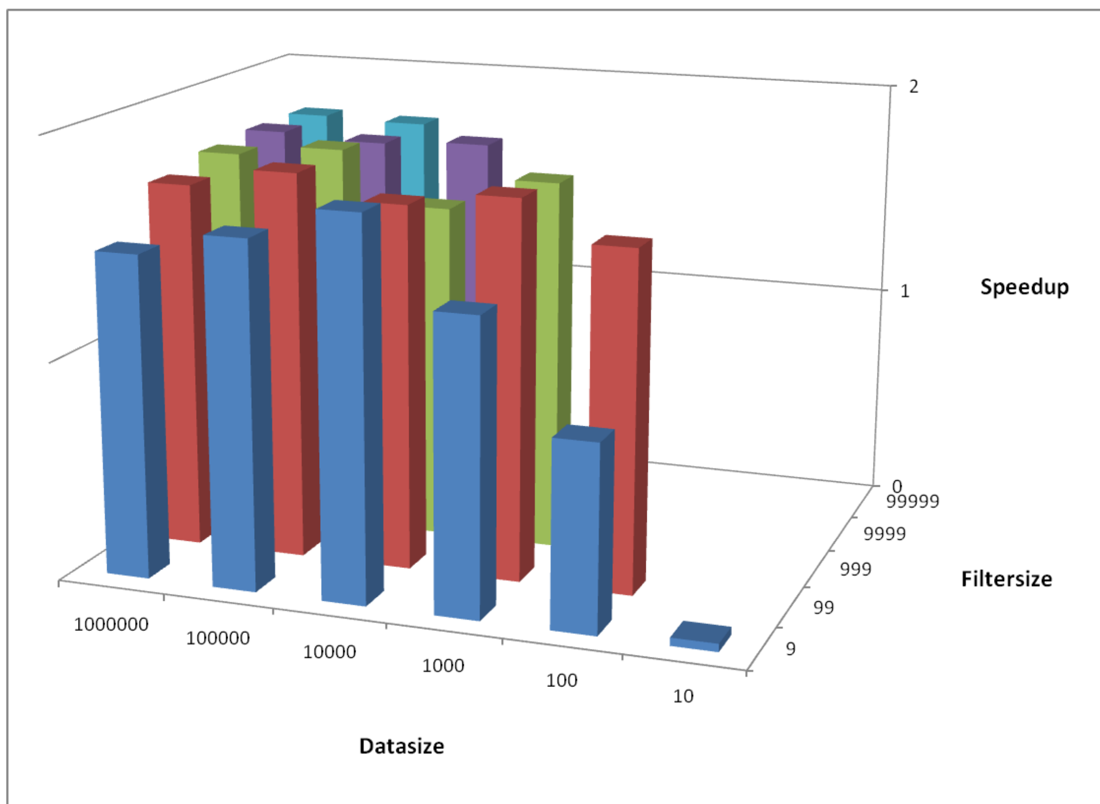


Figure B.3: Speedup of OpenMP-parallelised Algorithm 9 on Dualcore vs. sequential Algorithm 8

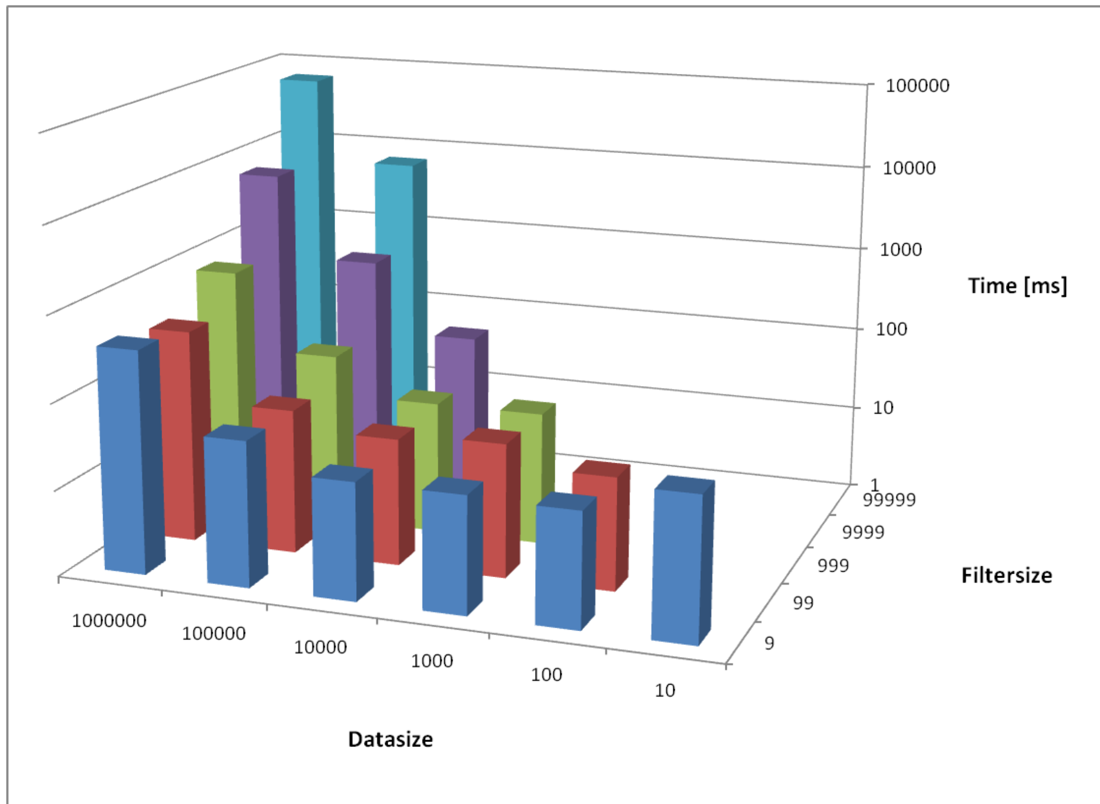


Figure B.4: GPU Runtime of Algorithm A.7 and A.8

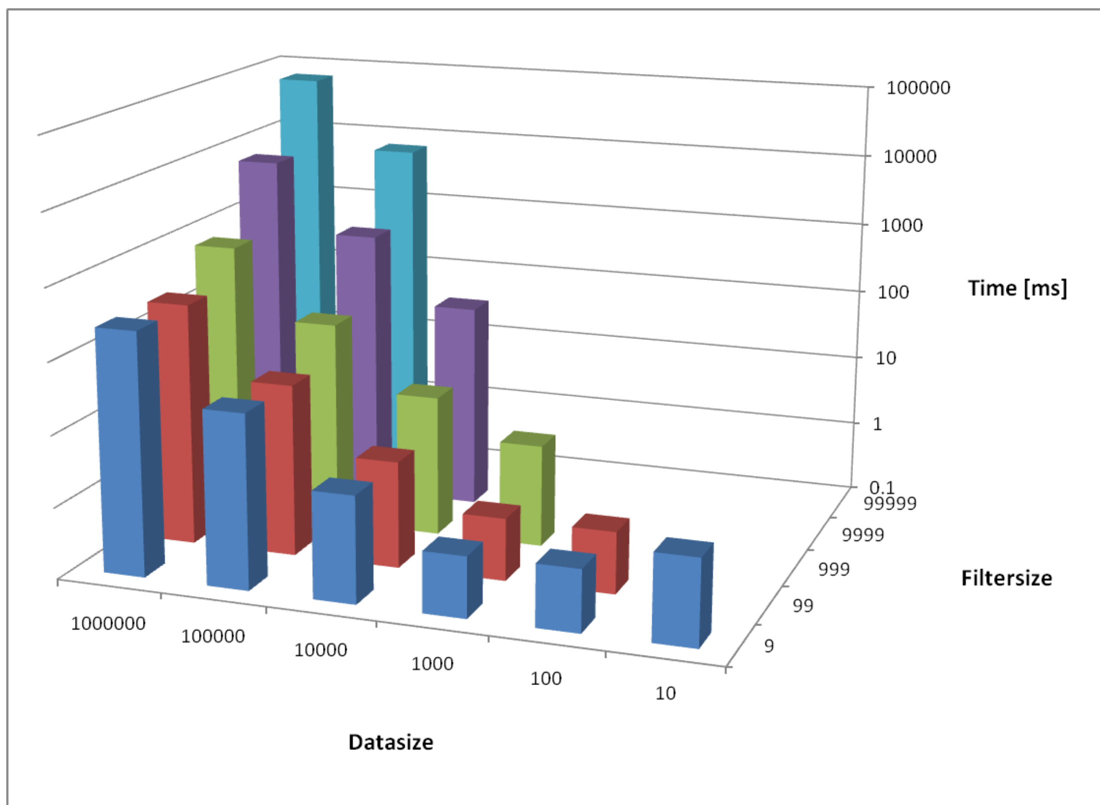


Figure B.5: Pure GPU runtime of Algorithm A.7 and A.8

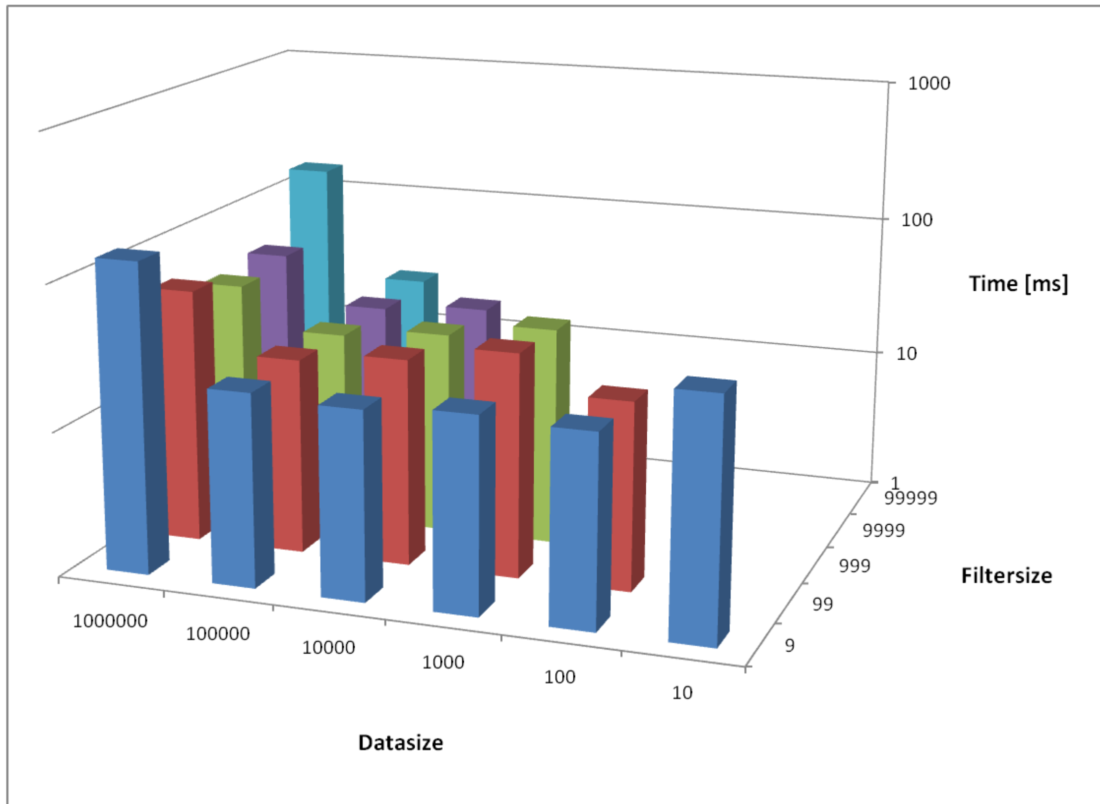


Figure B.6: Overhead time of GPU, like memory transfer and allocation

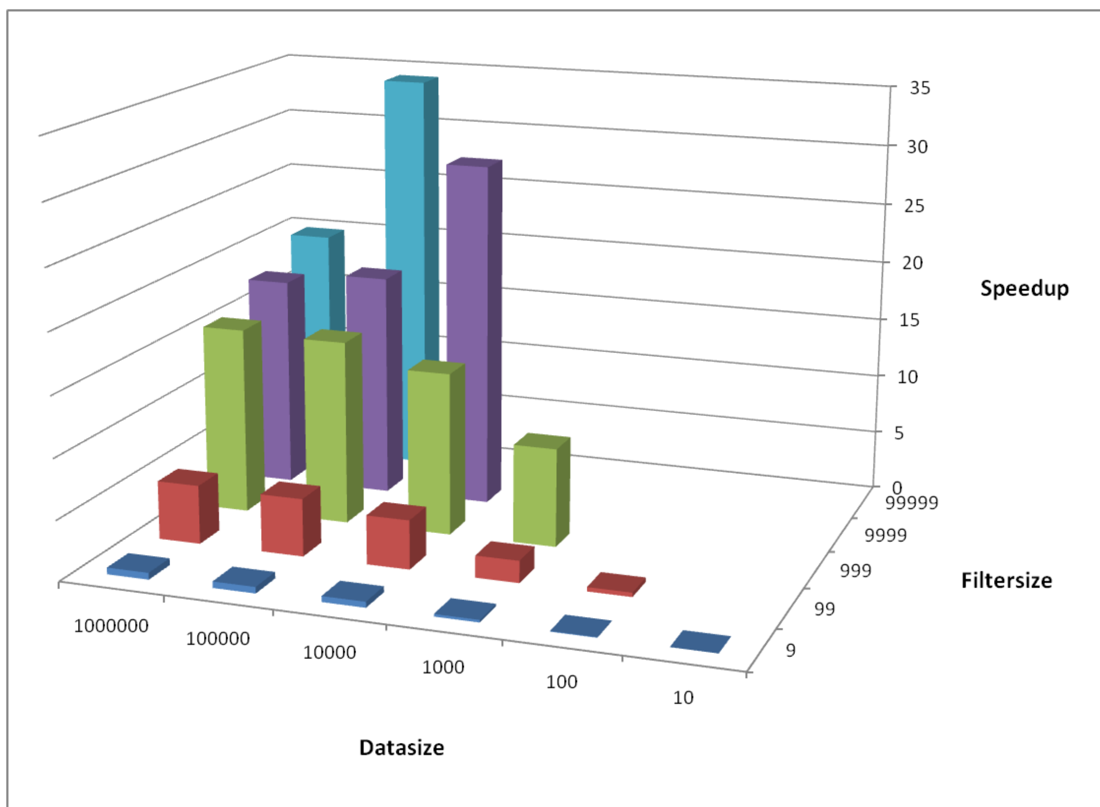


Figure B.7: Speedup of GPU without overhead vs. CPU

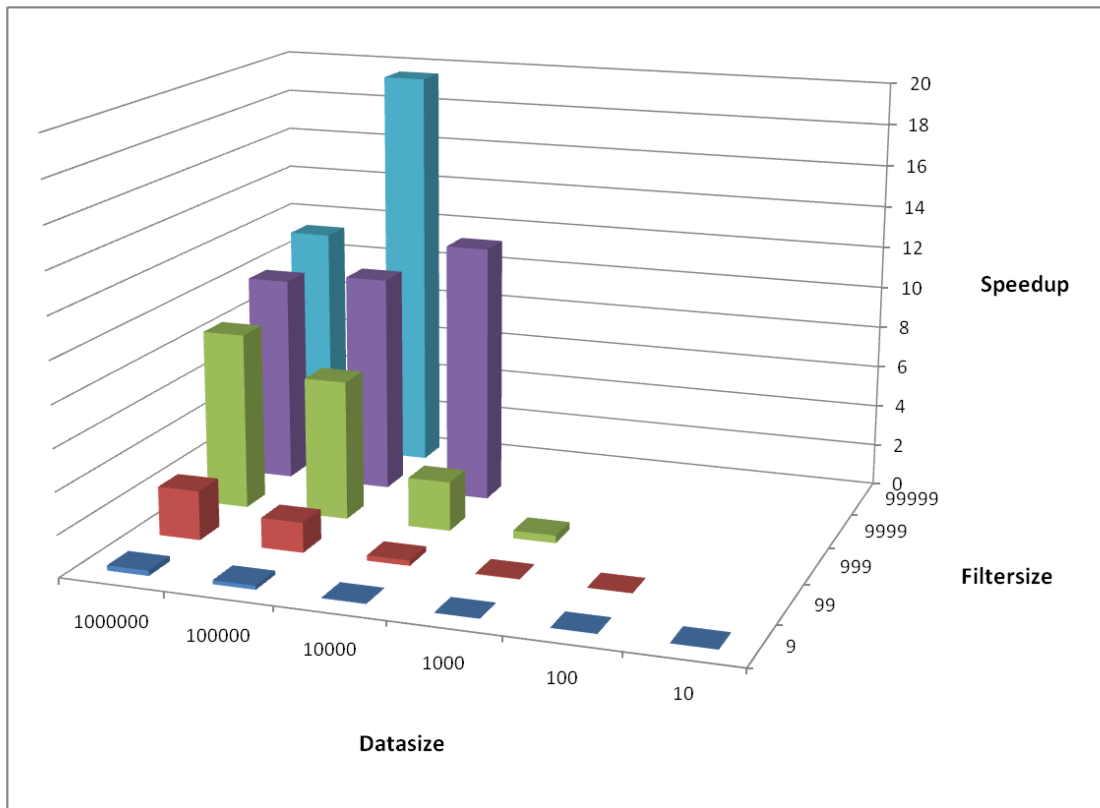


Figure B.8: Speedup of GPU with overhead vs. Dualcore CPU

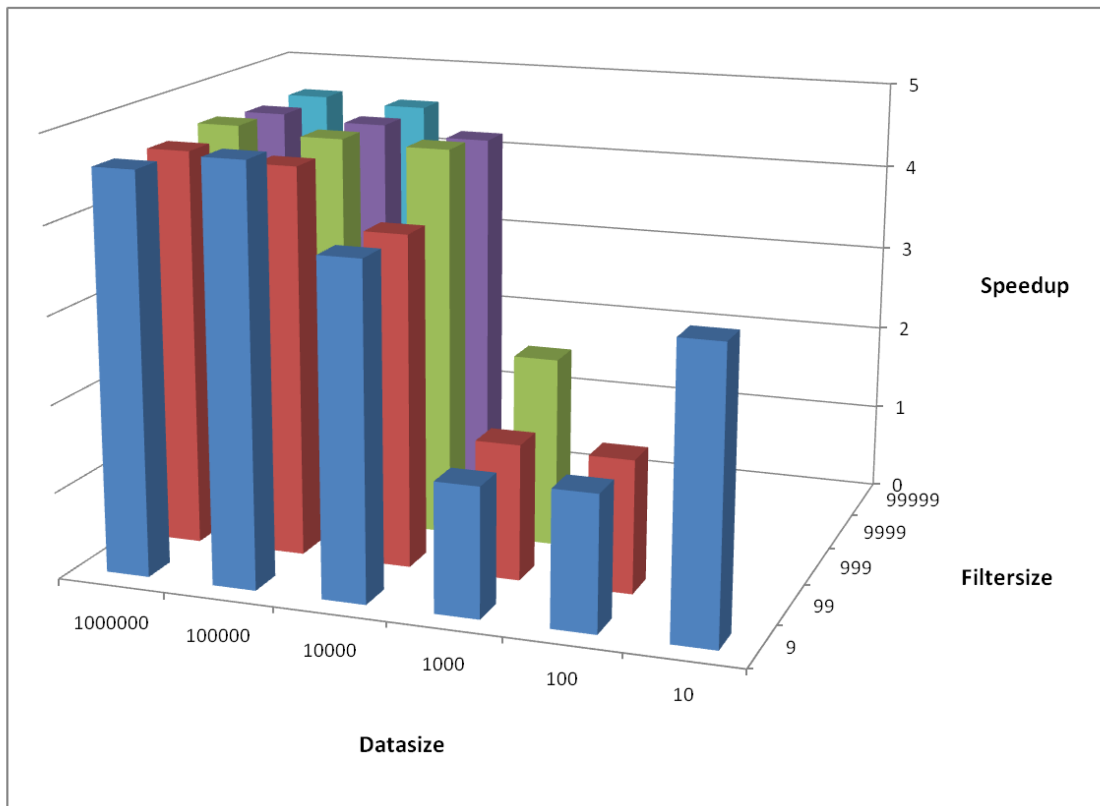


Figure B.9: Speedup of Nvidia FX 3700 vs. FX 1700 according to Algorithm A.7 and A.8

Figure B.10: Bandwidth of memory transfers from host to device, device to host and device to device. The vertical lines show an increasing increment of transfered data.

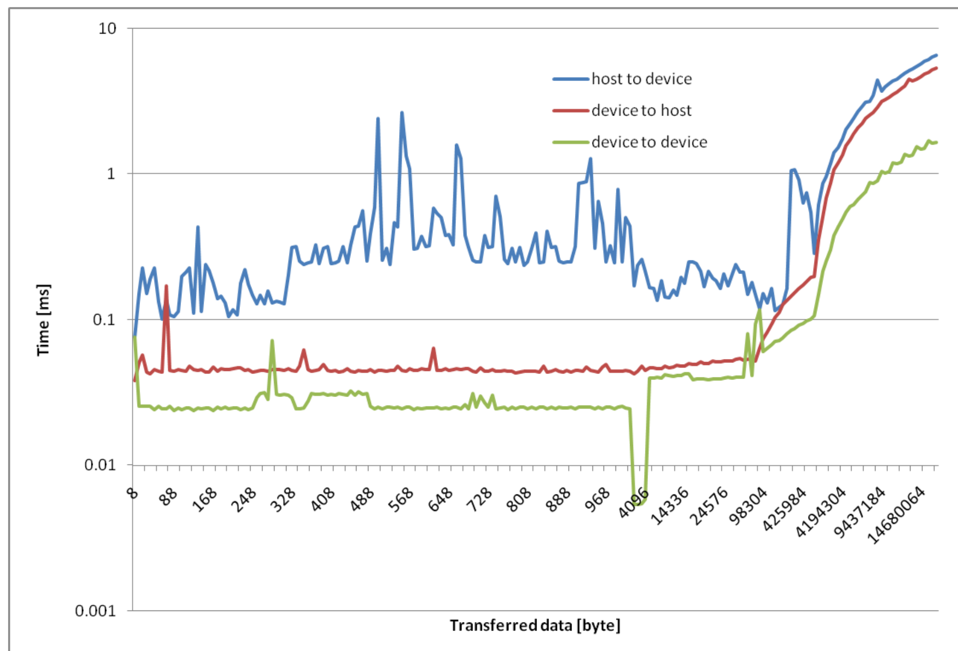
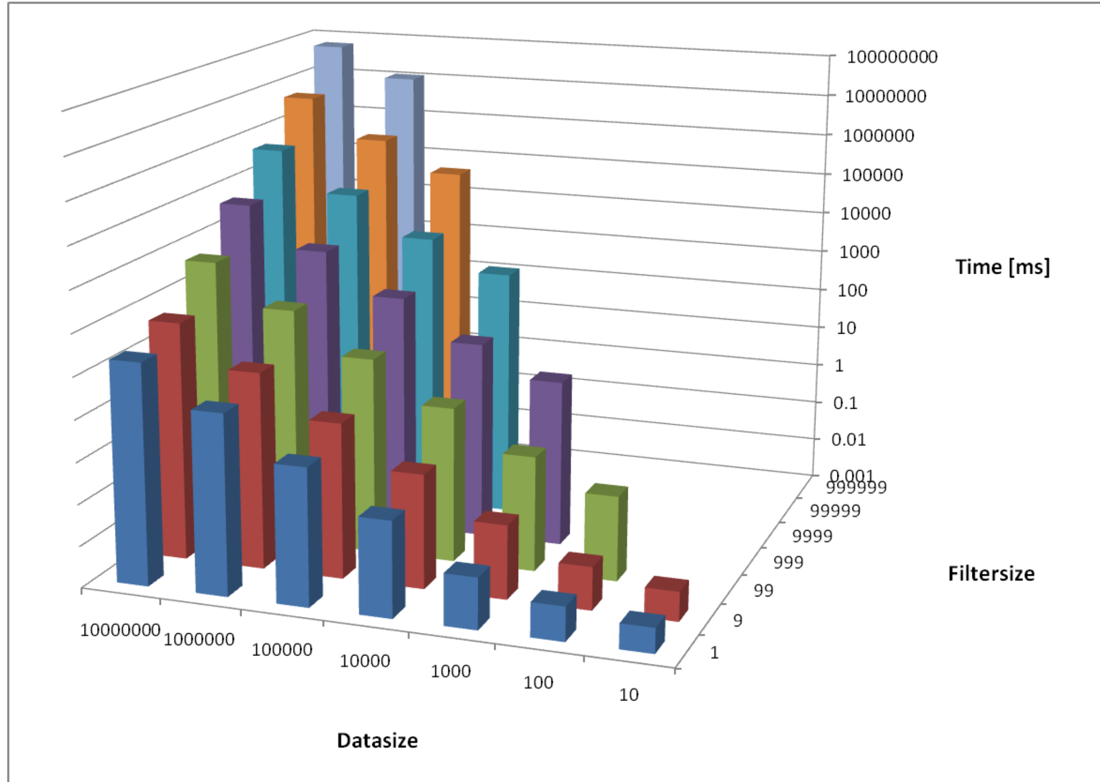


Figure B.11: Time of memory transfers from host to device, device to host and device to device.

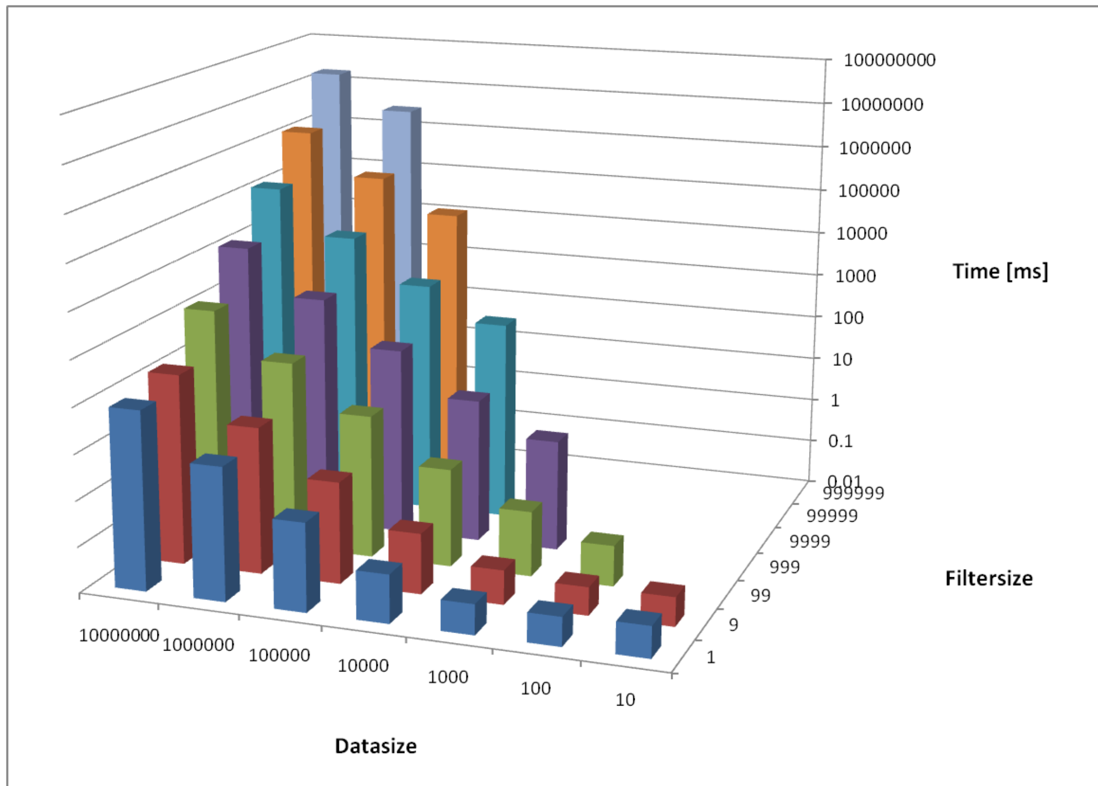
C. Appendix - Runtime Measurement Data

Performance Evaluation of Discrete Convolution Section 4.5



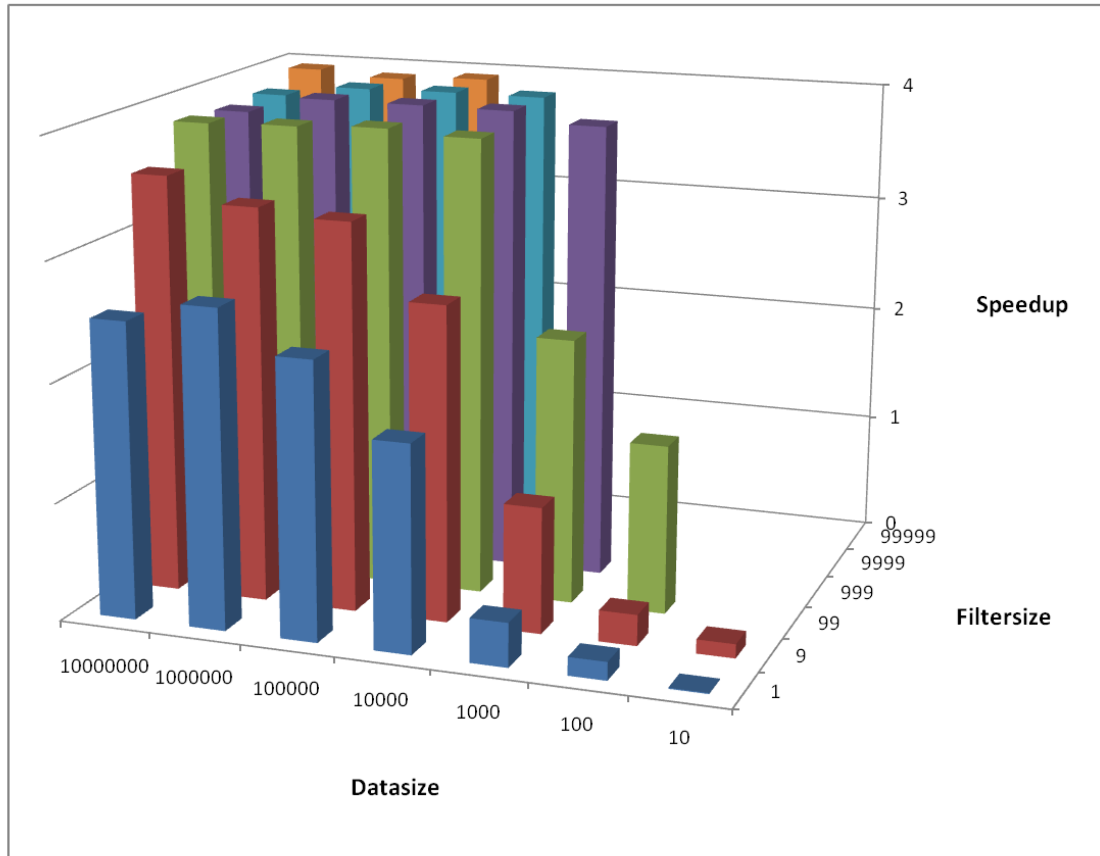
CPU	10	100	1000	10000	100000	1000000	10000000
1	0,003944	0,00655	0,016858	0,195667	1,976616	21,67899	206,2069
9	0,005072	0,01072	0,059200	0,549955	5,458774	54,66730	544,9637
99		0,11114	0,594699	5,466415	53,82201	537,8445	5390,250
999			10,55926	58,07612	533,3958	5287,000	53117,90
9999				1060,968	5832,031	53536,54	533482,6
99999					106047,8	584344,9	5383151
999999						11117224	61411612

Figure C.1: Runtime of sequential Algorithm 4 on CPU



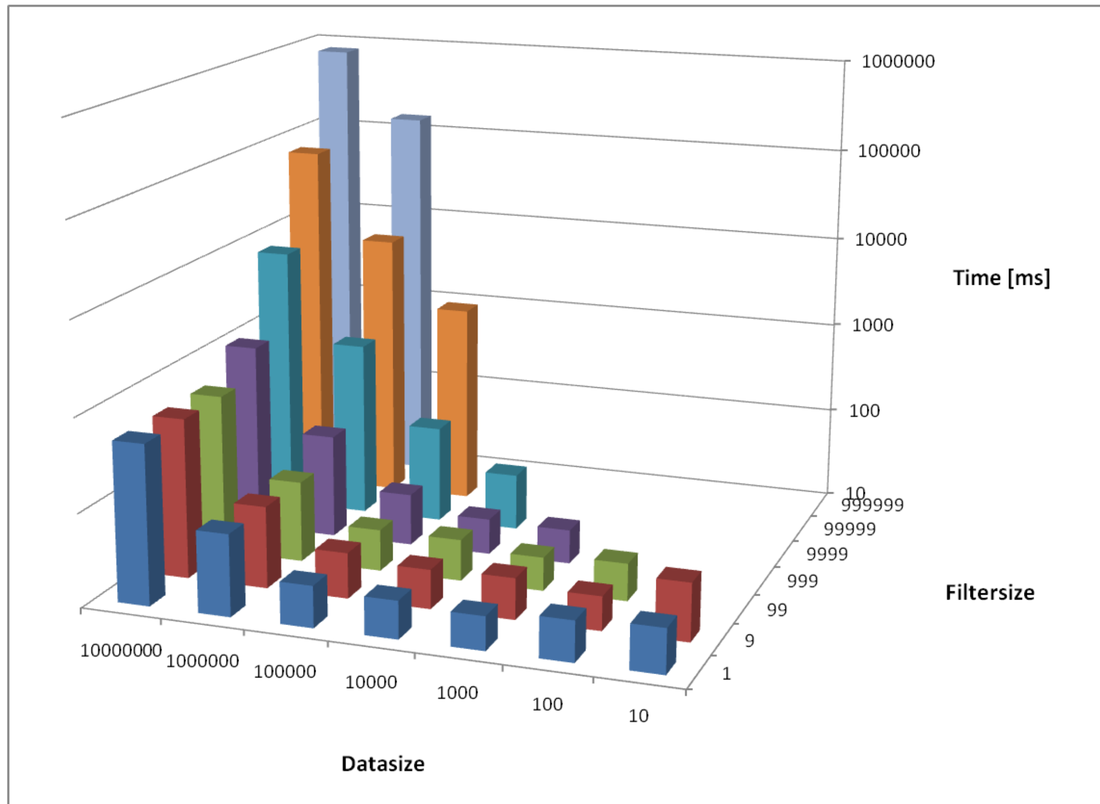
CPUmp	10	100	1000	10000	100000	1000000	10000000
1	0,047893	0,04254	0,045419	0,11264	0,847098	8,084098	82,33173
9	0,042716	0,04091	0,055748	0,20730	1,670591	16,42558	154,1975
99		0,07776	0,265266	1,41997	13,85371	139,4564	1404,817
999			2,732492	14,7002	134,8782	1336,662	13944,28
9999				269,021	1478,688	13611,37	139088,2
99999					26893,86	149522,9	1361733
999999						2845048	17615140

Figure C.2: Runtime of parallelised Algorithm 5 on CPU



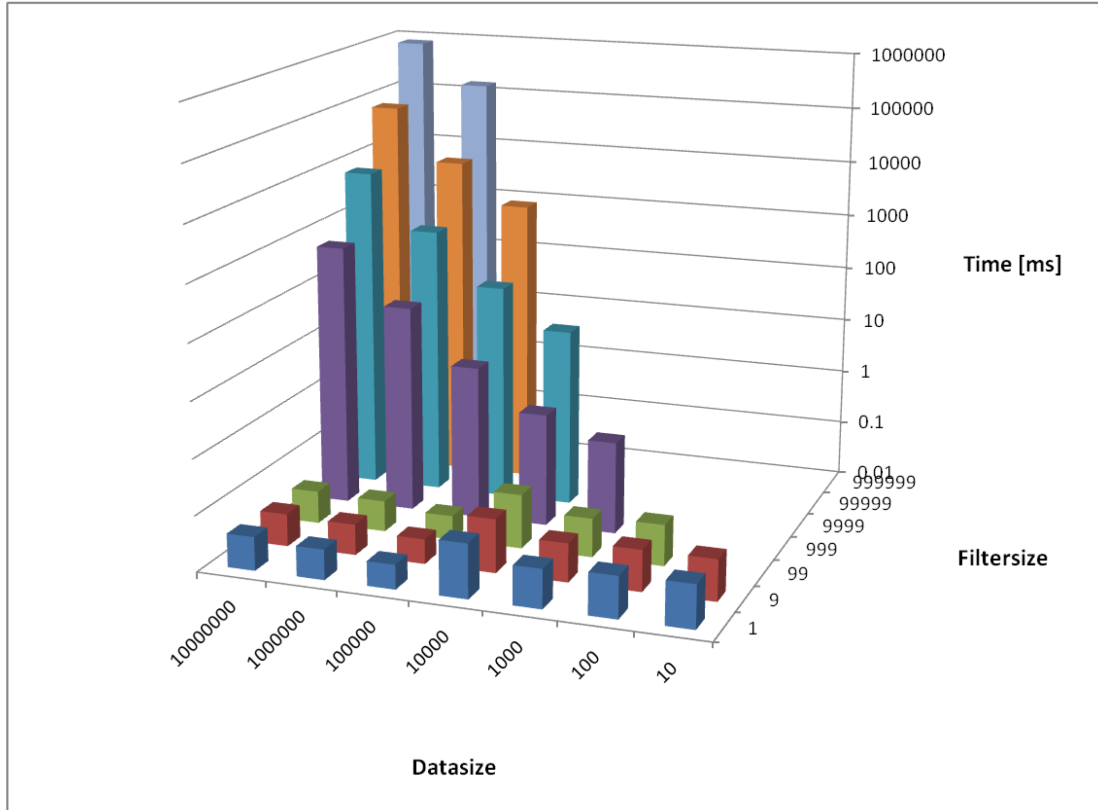
	10	100	1000	10000	100000	1000000	10000000
1	0,008234	0,154149	0,371177	1,736984	2,333391	2,681681	2,504586
9	0,118736	0,262146	1,061930	2,652904	3,267569	3,328178	3,534192
99		1,429126	2,241898	3,849661	3,885024	3,856719	3,836974
999			3,864334	3,950700	3,954647	3,955374	3,809295
9999				3,943807	3,944056	3,933219	3,835568
99999					3,943196	3,908061	3,953162

Figure C.3: Speedup of OpenMP-parallelised Algorithm 5 on Quadcore vs. sequential Algorithm 4



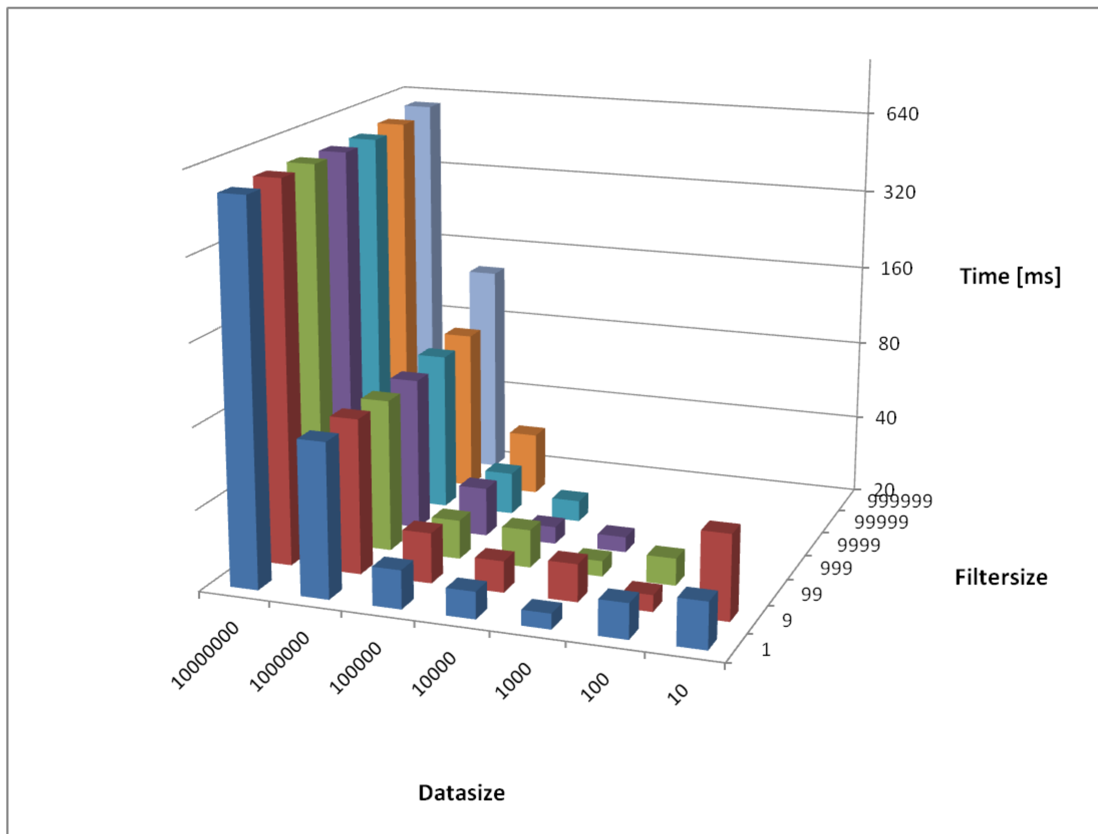
GPU	10	100	1000	10000	100000	1000000	10000000
1	29,76279	27,07262	22,86813	25,28900	27,83302	75,06784	520,7985
9	41,45894	23,11635	27,63391	26,24540	30,78648	75,79846	530,1592
99		25,30040	22,90259	27,80579	28,00226	74,35559	531,4260
999			23,29391	24,27217	36,29768	131,1501	1083,299
9999				40,65932	112,7492	815,4518	7793,458
99999					1524,556	8140,373	73878,28
999999						145391,4	800412,3

Figure C.4: GPU Runtime of Algorithm 6 and 7



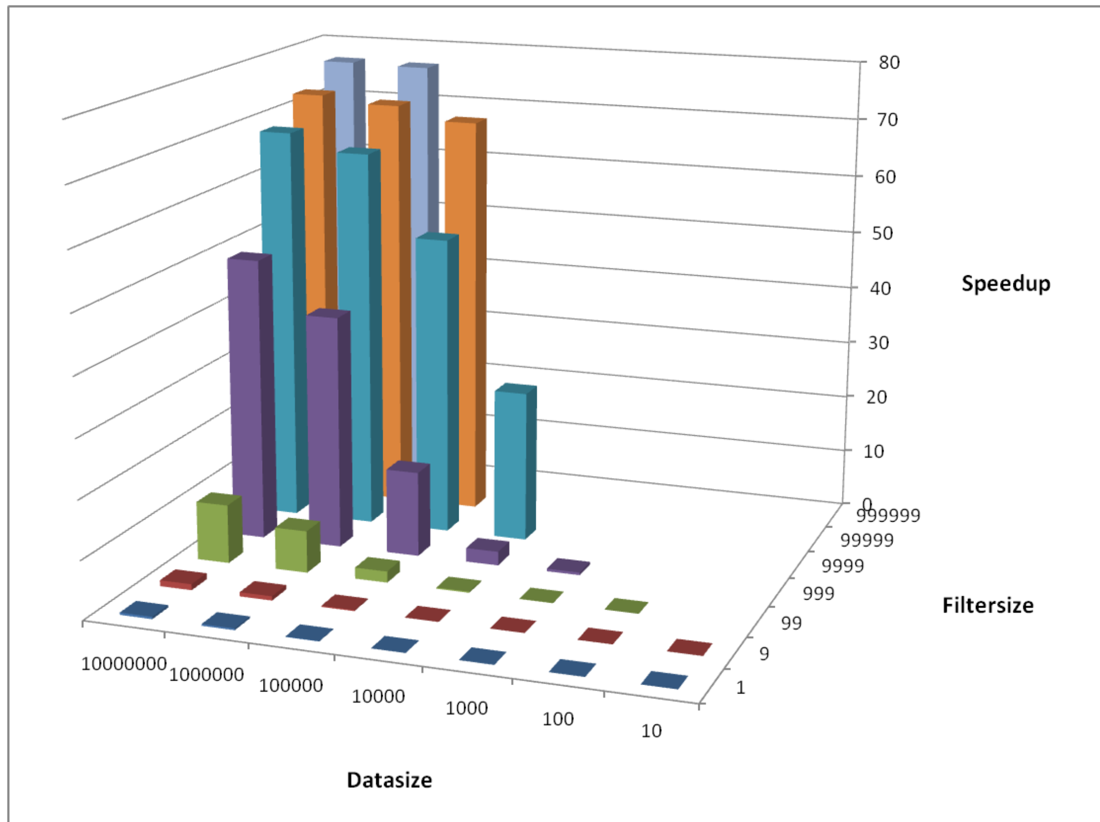
pGPU	10	100	1000	10000	100000	1000000	10000000
1	0,059196	0,055456	0,049786	0,093432	0,027436	0,034739	0,039993
9	0,054833	0,054772	0,049280	0,091075	0,027698	0,035549	0,038357
99		0,055127	0,049913	0,093707	0,027205	0,036039	0,038514
999			0,459692	1,085559	5,848817	56,17114	557,7921
9999				16,64027	83,93383	736,6070	7266,145
99999					1490,281	8058,598	73333,84
999999						145264,1	799832,9

Figure C.5: Pure GPU runtime of Algorithm 6 and 7



oGPU	10	100	1000	10000	100000	1000000	10000000
1	29,70364	27,01716	22,81835	25,19557	27,80558	75,03311	520,7585
9	41,40411	23,06158	27,58463	26,15432	30,75878	75,76291	530,1208
99		25,24527	22,85268	27,71208	27,97506	74,31955	531,3875
999			22,83421	23,18661	30,44886	74,97900	525,5072
9999				24,01904	28,81544	78,84484	527,3129
99999					34,27548	81,77470	544,4453
999999						127,3281	579,3750

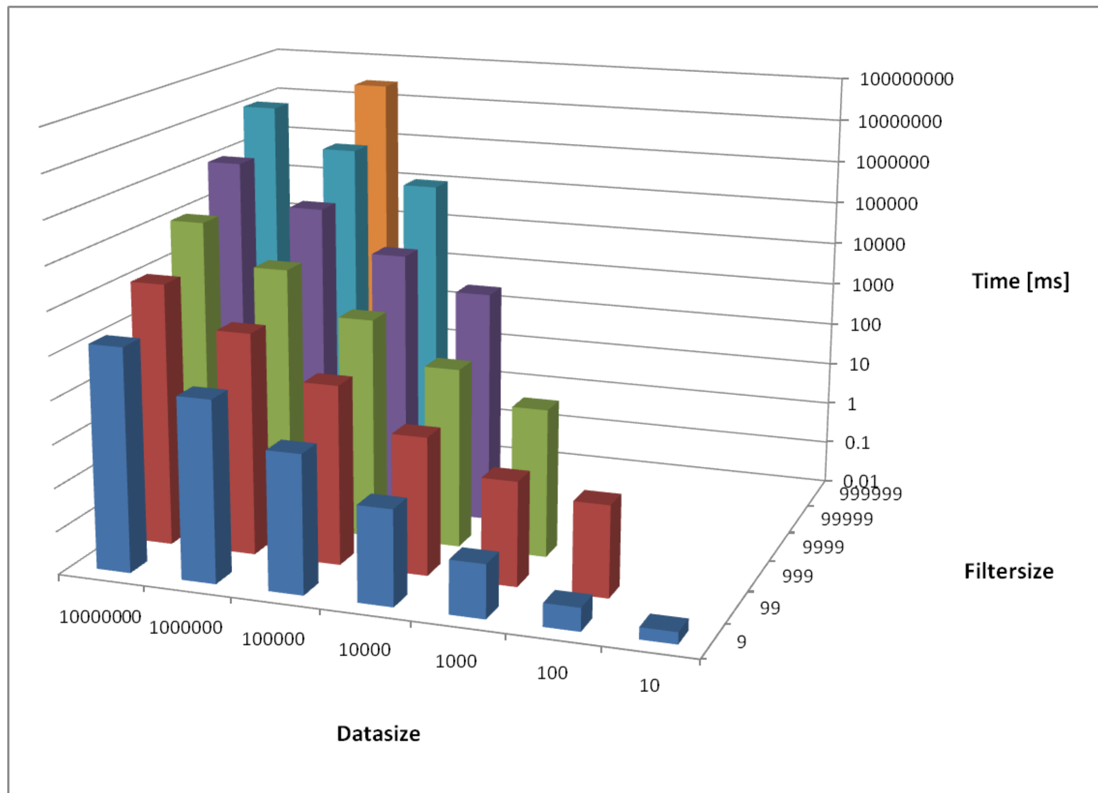
Figure C.6: Overhead time of GPU, like memory transfer and allocation



	10	100	1000	10000	100000	1000000	10000000
1	0,000132	0,000242	0,000737	0,007737	0,071016	0,288791	0,395943
9	0,000122	0,000464	0,002142	0,020954	0,177310	0,721219	1,027924
99		0,004392	0,025966	0,196592	1,922059	7,233410	10,14299
999			0,453305	2,392703	14,69503	40,31258	49,03344
9999				26,09411	51,72566	65,65260	68,45262
99999					69,55977	71,78355	72,86513
999999						76,46408	76,72497

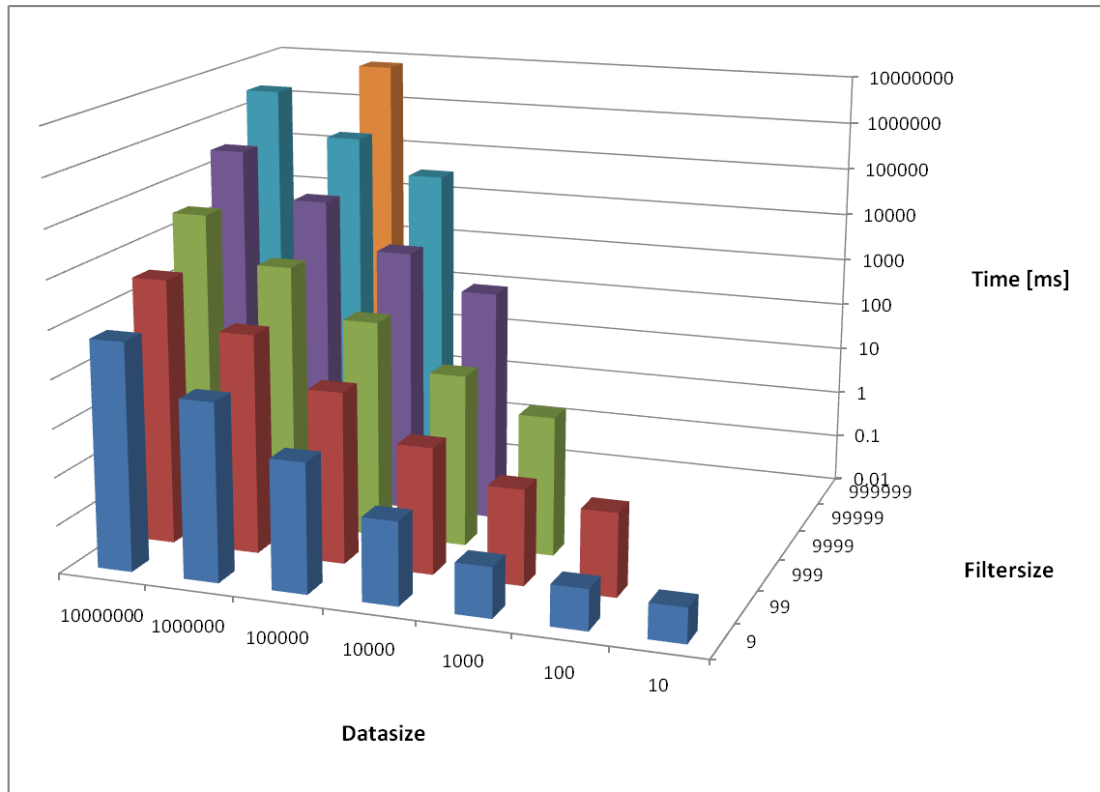
Figure C.7: Speedup of GPU with overhead vs. CPU

Performance Evaluation of Rolling Ball Section 5.4



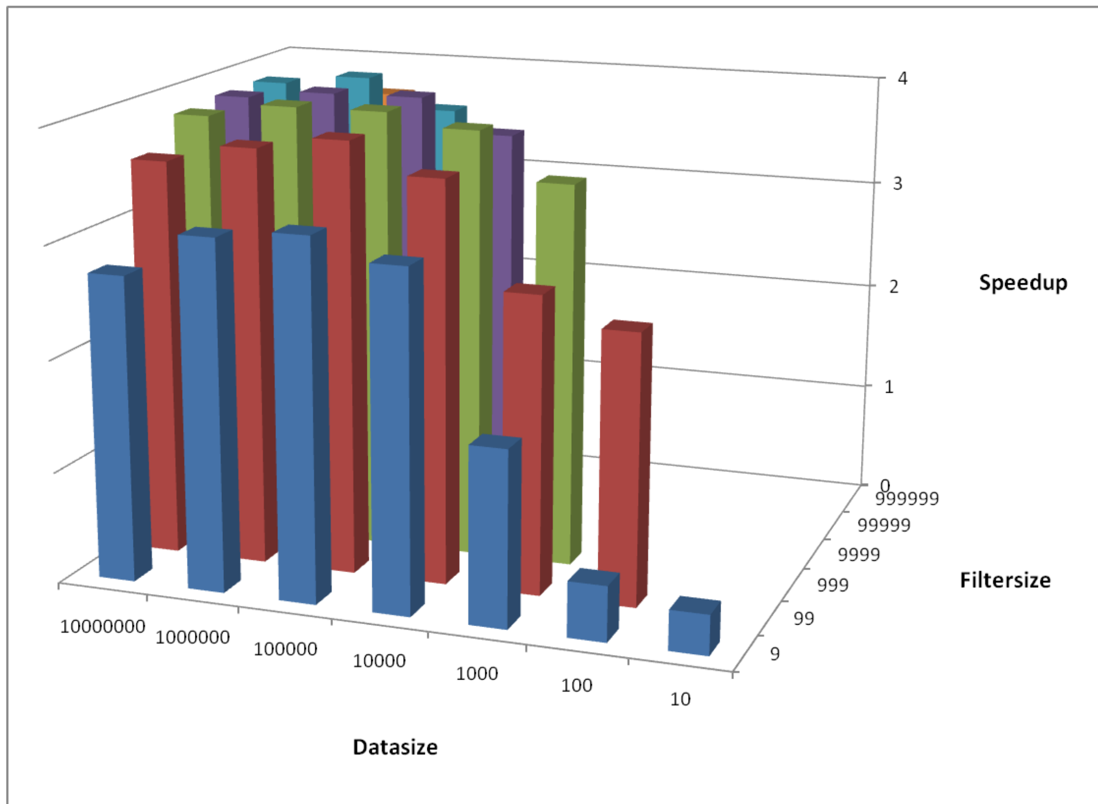
CPU	10	100	1000	10000	100000	1000000	10000000
9	0,018655	0,034032	0,170986	1,593155	15,81792	157,9421	1482,302
99		1,294785	2,558505	15,28338	143,1098	1415,178	12509,58
999			27,58110	149,4791	1376,885	13661,11	121401,9
9999				2672,916	14904,39	137380,8	1227938
99999					266197,7	1491503	12399998
999999						27264780	

Figure C.8: Runtime of sequential Algorithm 8 on CPU



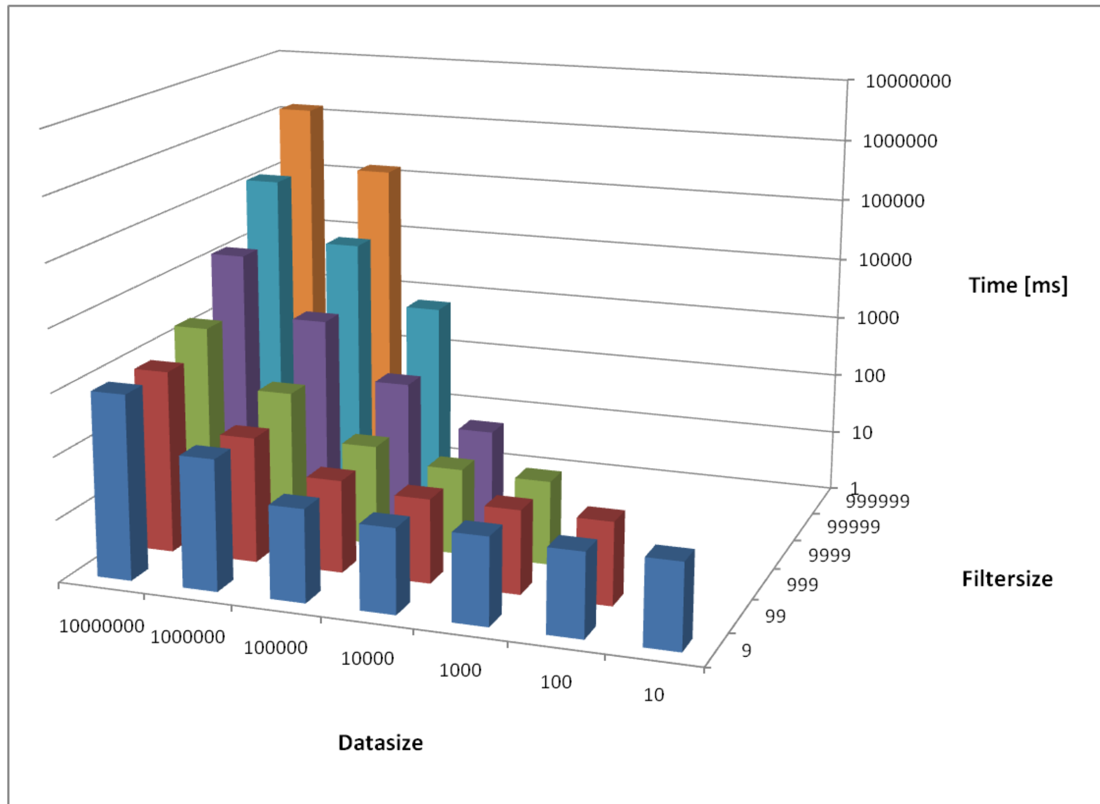
CPUmp	10	100	1000	10000	100000	1000000	10000000
9	0,052575	0,069283	0,109448	0,528207	4,933441	50,53851	542,8168
99		0,537710	0,964069	4,270786	37,19361	379,5483	3513,463
999			8,046522	38,87788	347,9736	3457,041	31744,21
9999				731,1175	3769,775	34833,54	317704,3
99999					71946,89	376097,8	3201256
999999						7411186	

Figure C.9: Runtime of parallelised Algorithm 9 on CPU



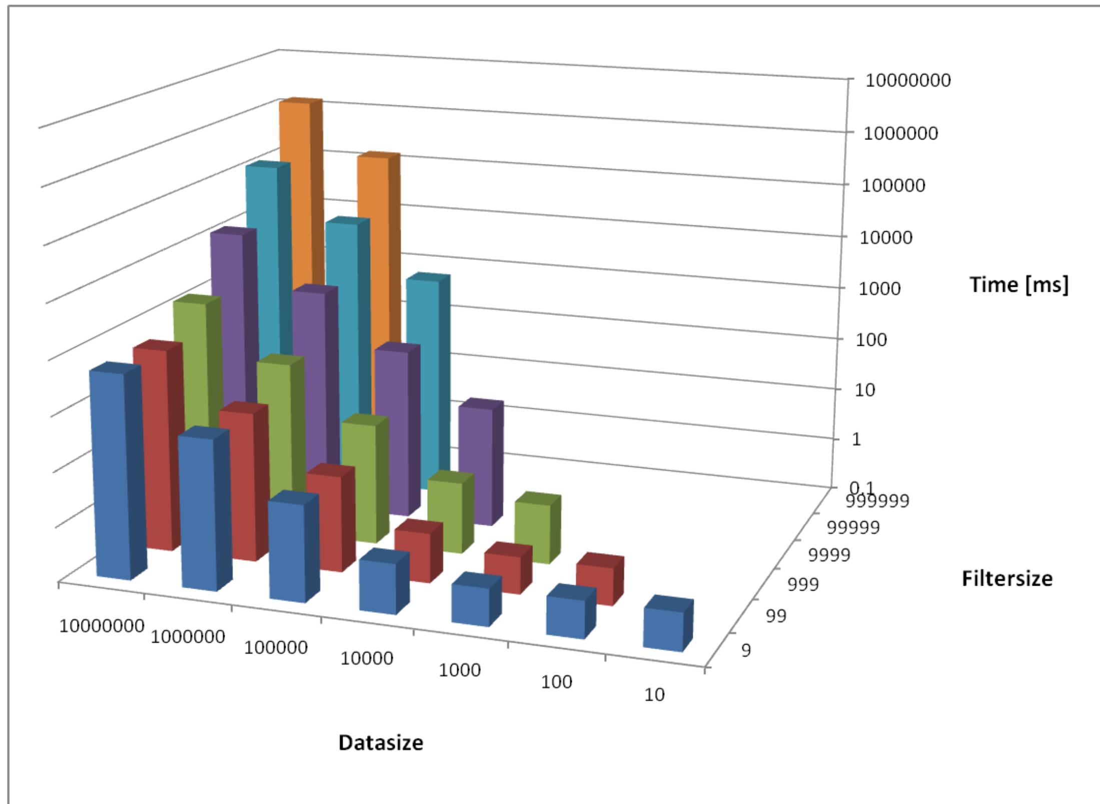
	10	100	1000	10000	100000	1000000	10000000
9	0,354825	0,491202	1,562257	3,016156	3,206266	3,125184	2,730760
99		2,407961	2,653860	3,578587	3,847698	3,728584	3,560472
999			3,427704	3,844837	3,956867	3,951676	3,824381
9999				3,655932	3,953655	3,943923	3,865033
99999					3,699920	3,965731	3,873478
999999						3,678868	

Figure C.10: Speedup of OpenMP-parallelised Algorithm 9 on Quadcore vs. sequential Algorithm 8



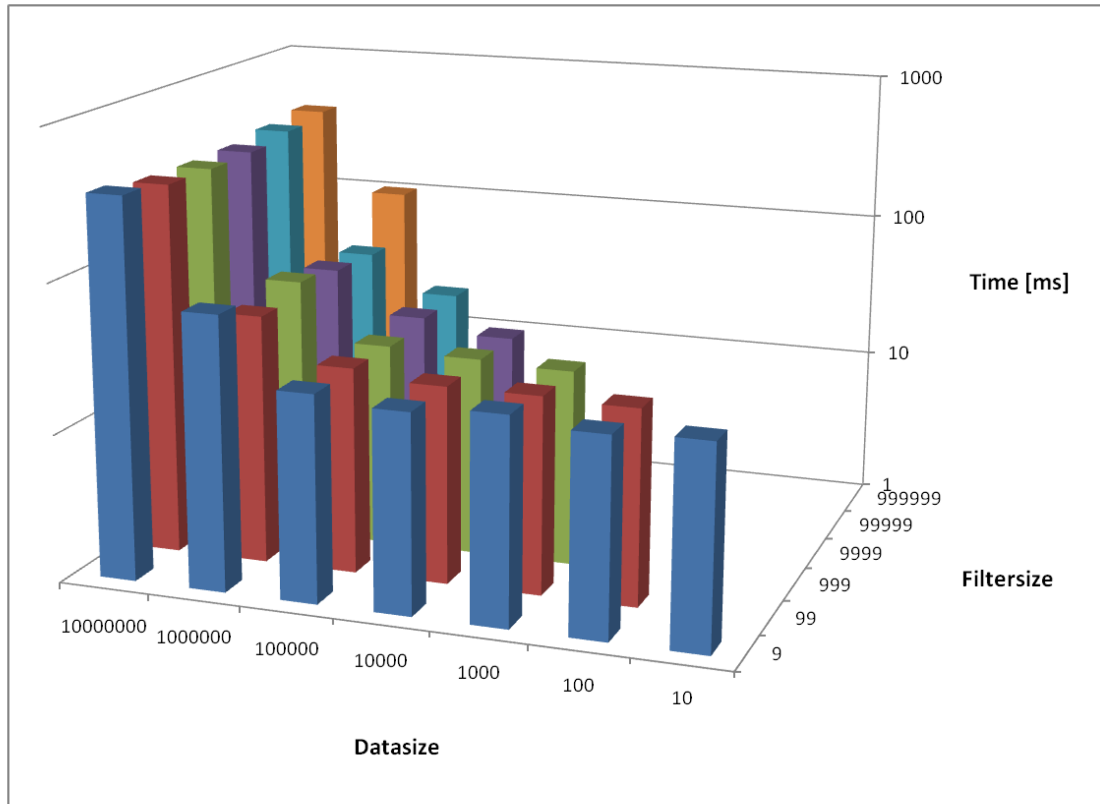
GPU	10	100	1000	10000	100000	1000000	10000000
9	22,93725	21,66378	24,77025	22,56176	30,30037	124,9578	910,6806
99		21,11122	21,87842	22,25383	30,32320	102,9182	868,0106
999			22,39855	24,35691	40,83733	220,9435	1928,824
9999				39,14475	178,5734	1523,813	14743,69
99999					1493,544	14236,70	139755,5
999999						141205,3	1380765

Figure C.11: GPU Runtime of Algorithm A.7 and A.8



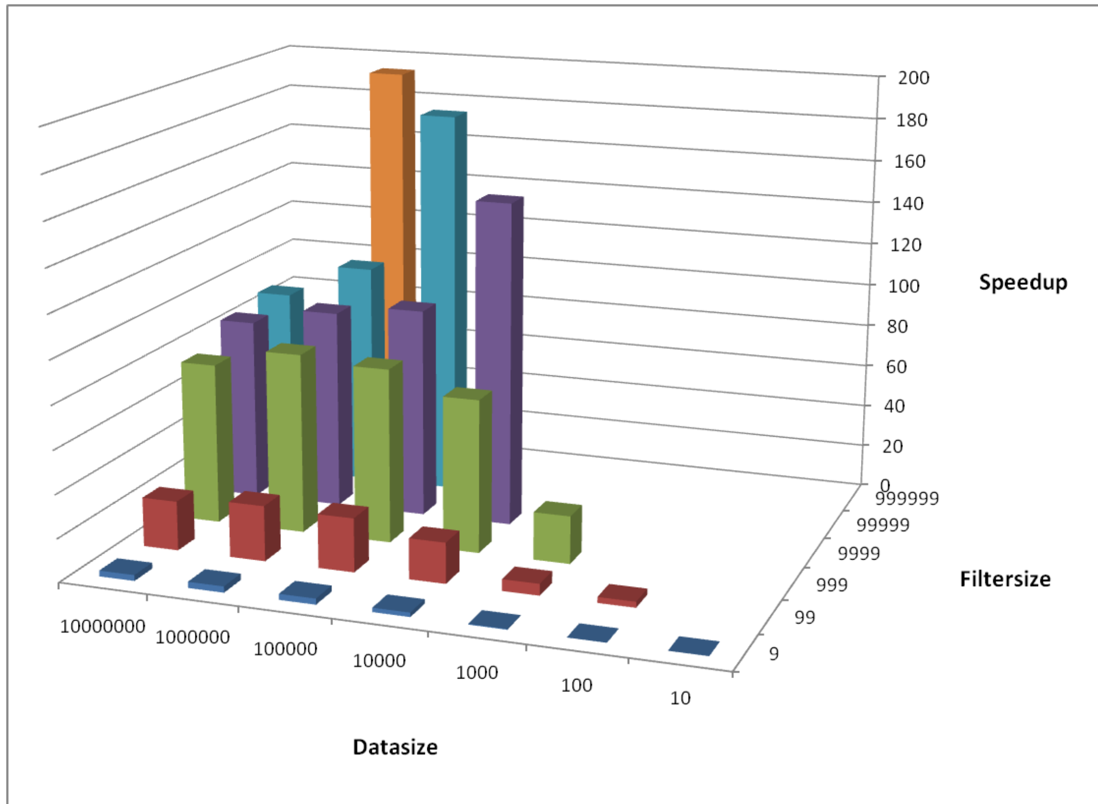
pGPU	10	100	1000	10000	100000	1000000	10000000
9	0,481257	0,472054	0,475937	0,806210	5,732386	55,13901	547,8827
99		0,476917	0,475555	0,810000	5,731862	55,00223	544,4168
999			1,242467	2,096950	16,93275	162,3682	1613,261
9999				17,62998	152,0972	1472,639	14425,08
99999					1466,365	14188,13	139404,8
999999						141103,9	1380378

Figure C.12: Pure GPU runtime of Algorithm A.7 and A.8



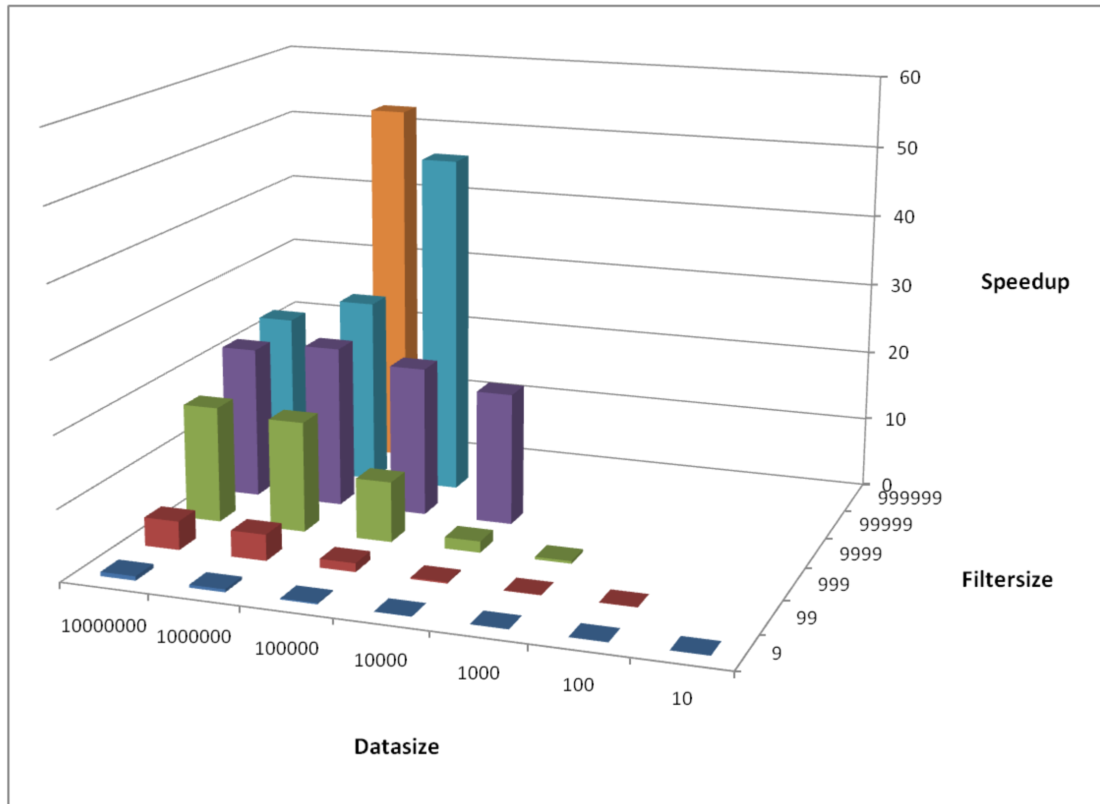
oGPU	10	100	1000	10000	100000	1000000	10000000
9	22,45599	21,19173	24,29431	21,75555	24,56798	69,81879	362,7979
99		20,63430	21,40286	21,44383	24,59134	47,91601	323,5937
999			21,15608	22,25996	23,90457	58,57530	315,5627
9999				21,51477	26,47618	51,17419	318,6123
99999					27,17944	48,57031	350,7500
999999						101,3906	386,7500

Figure C.13: Overhead time of GPU, like memory transfer and allocation



	10	100	1000	10000	100000	1000000	10000000
9	0,038763	0,072093	0,359261	1,976104	2,759396	2,864435	2,705511
99		2,714906	5,380040	18,86837	24,96742	25,72946	22,97796
999			22,19865	71,28407	81,31489	84,13659	75,25250
9999				151,6119	97,99249	93,28883	85,12520
99999					181,5357	105,1232	88,94956
999999						193,2247	

Figure C.14: Speedup of GPU without overhead vs. CPU



	10	100	1000	10000	100000	1000000	10000000
9	0,002292	0,003198	0,004418	0,023411	0,162817	0,404444	0,596056
99		0,025470	0,044064	0,191912	1,226572	3,687862	4,047719
999			0,359242	1,596174	8,520968	15,64672	16,45780
9999				18,67727	21,11050	22,85944	21,54848
99999					48,17190	26,41748	22,90611
999999						52,48515	

Figure C.15: Speedup of GPU with overhead vs. Quadcore CPU

Bibliography

[Ata99] Mikhail J. Atallah. *Algorithms and theory of computation handbook*. CRC Press, 1999.

[BCI+08] S. Barrachina, M. Castillo, F.D. Igual, R. Mayo, and E.S. Quintana-Orti. Evaluation and tuning of the level 3 cublas for graphics processors. *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8, April 2008.

[DN00] Helene Desmartis and Bernd Nawracala. A method for processing measuring values, 01 2000.

[FSH04] K. Fatahalian, J. Sugerman, and P. Hanrahan. Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 133–137, New York, NY, USA, 2004. ACM.

[gas] http://www.gass-ltd.co.il/en/products/default.aspx.

[Gil58] S. Gill. Parallel programming, 1958.

[Har07] Mark Harris. Optimizing parallel reduction in cuda. page 38, 2007.

[iee] http://754r.ucbtest.org/standards/754.pdf.

[inta] http://www.intel.com/pressroom/archive/releases/20050418comp.htm.

[intb] http://www.intel.com/pressroom/archive/releases/20070204comp.htm.

[MWHL06] Michael D. McCool, Kevin Wadleigh, Brent Henderson, and Hsin-Ying Lin. Performance evaluation of gpus using the rapidmind development platform. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 181, New York, NY, USA, 2006. ACM.

[nvia] http://developer.download.nvidia.com/compute/cuda/1_1/cublas_library_1.1.pdf.

[nvib] http://developer.download.nvidia.com/compute/cuda/2_0/docs/nvidia_cuda_programming_guide_2.0.pdf.

[nvic] http://forums.nvidia.com/index.php?showtopic=84440&view=findpost&p=478583.

[nvid] http://www.nvidia.com/object/cuda_develop.html.

[nvie] http://www.nvidia.com/object/cuda_get.html.

[nvif] http://www.nvidia.com/object/cuda_sdks.html.

[ope] http://www.khronos.org/news/press/releases/the_khronos_group_releases_opencl_1.0_specification.

[PH09] David A. Patterson and John L. Hennessy. *Computer organization and design*. Elsevier Morgan Kaufmann, 4. ed. edition, 2009.

[rap] http://www.rapidmind.net.

[Rod85] David P. Rodgers. Improvements in multiprocessor system design. volume 13, pages 225–231, New York, NY, USA, 1985. ACM.

[RRB+08] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82, New York, NY, USA, 2008. ACM.

[SG64] Abraham. Savitzky and M. J. E. Golay. Smoothing and differentiation of data by simplified least squares procedures. *Analytical Chemistry*, 36(8):1627–1639, 1964.

[TM08] Prof. Dr. Walter F. Tichy and David Meder. Matrizenmultiplikation vergleich verschiedener algorithmen. 2008.

[Wil94] Gregory Wilson. http://ei.cs.vt.edu/~history/parallel.html, 1994.

[WP08] Samuel Williams and David Patterson. The roofline model: A pedagogical tool for program analysis and optimization, 2008.

[WWP09] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, 2009.