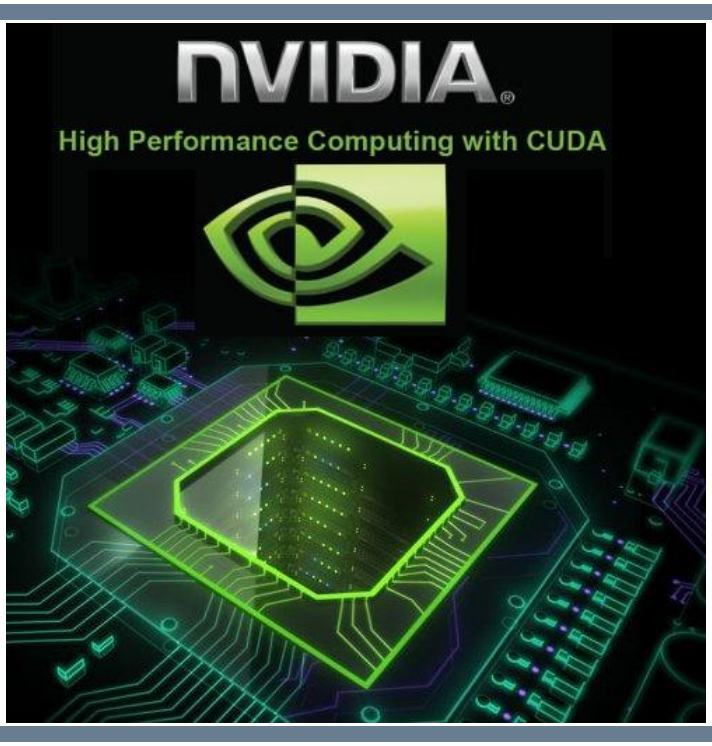

OCT-Processing Boost on GPU

GPU acceleration with CUDA

Division: HuCE - OptoLab
Author: Daniel Tschupp
Date: 07. July 2019

Bern University of Applied Sciences

Abstract

Optical Coherence Thomography (OCT) is an analysis method that physicians often use to identify diseases inside the human eye, especially on the retina. It generates high resolution 3D images of the retina . The modern OCT systems are built with either a swept-source laser (SS-OCT) or a spectrometer (SD-OCT). Both are Frequency Domain OCTs and therefore the SNR is better than with traditional Time Domain OCTs but at the cost of a more complex processing. Especially the new generation of commercially available swept-source lasers may be very fast. They can generate about 800'000 depth scans (A-Scans) in a second. Those scans consists usually of 1024 to 4096 axial measurements points. This makes a total of about 800MSamples/s - 3.2GSamples/s. Those have to be processed. To use a Graphical Processing Unit (GPU) is one way to process the measured data that fast. NVIDIA provides its own language called CUDA to implement such algorithms on GPUs. The original CUDA implementation of the HuCE-Optolab could process about 400'000 to 500'000 A-Scan per second with 1024 points per A-Scan on a NVIDIA GeForce 1080Ti graphics card. The goal of this paper was to identify the short-comings of the actual implementation, optimize it to process at least 1'000'000 A-Scans per second with a vector length of 2048 points. This goal was reached with a measured maximum of 1'350'000 A-Scans per second. This paper describes the new implementation.

Table of Content

1. Introduction

1.1. Overview

OCT is an analysis method that physicians often use to identify diseases inside the human eye, especially on the retina. The modern OCT systems are built with either a swept-source laser (SS-OCT) or a spectrometer (SD-OCT). Especially the new generation of commercially available swept-source lasers may be very fast. They can generate about 800'000 depth scans (A-Scans) in a second. Those scans consists usually of 1024 to 4096 axial measurements points. This makes a total of about 800MSamples/s - 3.2GSamples/s. Those have to be processed. To use a GPU is one way to process the measured data that fast. The original CUDA implementation of the HuCE-Optolab could process about 400'000 to 500'000 A-Scan per second with 1024 points per A-Scan on a NVIDIA GeForce 1080Ti graphics card. The goal of this paper was to identify the short-comings of the actual implementation, optimize it to process at least 1'000'000 A-Scan per second with a vector length of 2048 points.

1.2. Scope

This documentation shows the processing steps worked out by the HuCE-Optolab for OCT processing. It won't give a mathematical description of this processing though, nor will it describe why each of the processing steps are necessary. To work through the whole OCT theory would go far beyond the scope of this work but there are lots of publications and theory around for interested people.
A further topic of this document is the analysis of the original HuCE-Optolab algorithm. It will point out weaknesses thereof.
Finally the new and faster algorithm is described step by step. The important hints why certain implementations are better than others will be provided too.
A small documentation of general GPU hardware as well as some basic principle for general heterogeneous programming can be found in the appendix.

1.3. Audience

This work may be of interest for anyone who wants to accelerate OCT processing with GPU. Furthermore it is an example on how to write Compute Unified Device Architecture (CUDA) kernels and use streams. This could also guide a beginner who is looking for an example to start with the CUDA programming language or for people who are generally interested in GPU programming or OCT processing.

1.4. Situation and purpose

As the data acquisition hardware gets faster by the year the HuCE-Optolab of the Bern University of Applied Science gets the problem that their actual CUDA implementation for the processing of those data is to slow to do it in real-time anymore. The goal of this work is to improve this implementation. This has been done with an optimized CUDA software implementation described in this paper.

1.5. Results

In this work a successful analysis of the original HuCE-Optolab algorithm has been done. This resulted in a number of recognized weaknesses like serial data exchange and processing, lots of kernel calls preventing each to use the GPUs shared memory, lots of unnecessary Fast Fourier Transformation (FFT) calculations and lots of global memory loads which could be circumvented. A new algorithm has been implemented which corrects those flaws. For the DC-Removal some comparison measurements were made as the algorithm completely changed. Those measurements showed that the new algorithm produces similar results as the old one. With the new implementation speeds up to 2.7M A-Scans per second have been achieved with a 1024 element data vector.

2. OCT processing

This is the central chapter of this work. The first section describes the OCT processing as a whole to give an overview over the topic.
In the second section, the actual situation is analyzed. It will point out possible improvements.
The last section is about the new implementation. It describes step by step the whole implementation as well as the ideas behind it. As can be seen most of the mentioned points of the second section are covered in the new implementation.

2.1. Overview

The whole processing behind an OCT A-Scan is described in this section. The math behind it would show why each of the steps in fig. 2.1 is necessary but it is not part of this work and can be read in the corresponding literature. This work is only about accelerating the given algorithm.

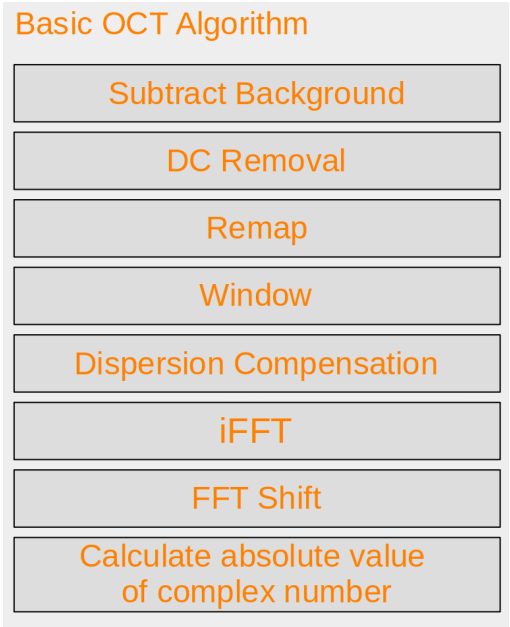

Fig. 2.1.: This black box diagram shows the OCT processing step by step for an A-Scan.

Step by step description of the algorithm:

Subtract Background:
An OCT system has multiple lenses in each, reference arm as well as sample arm. Those can cause auto-correlation and other effects which disturb the measurement but are independent of the measurement signal. These disturbances can be removed by recording a background vector and subtracting it from the measurements.

DC Removal:
When processing FD-OCT signals the measured signal will be a mix of the interference signal and a DC offset which results from asymmetries of the swept-source OCTs balanced detectors or the general light spectrum in the

spectrometer based OCTs.

Remap:
In general a spectrometer is not linear in k (Wave number). This is because of the optical grating inside it that disperse light with a sinus function. In swept-source OCT systems the non-linearity comes from the swept-source laser that doesn't sweep linearly. Therefore it is crucial to interpolate and re-sample the measured signal. The method chosen to do that is the NUFFT algorithm because it is convolution based which can be done by parallelism. The Remapping section will describe the NUFFT algorithm further. More about the math behind the NUFFT algorithm can be found in the corresponding literature.

Window:
To minimize artifacts, a Hann-Window [4] is used. This artifacts are caused by signals just a little above the Shanon-Sampling-Theorem [6] which then back-reflects inside the measurement signal.

$$FD(k) = FD(k) * Hann$$
$$Hann = \frac{1}{2} * \left[sin\left(\frac{2\pi k}{n-1} + \frac{3\pi}{2}\right) + 1 \right]$$

FD: Frequency Domain Signal
k: Wave number
Hann: Von Hann window
n: Number of samples per A-Scan

Dispersion Compensation:
As an OCT system consists of multiple lenses and other optical and opto-mechanical parts there will be dispersion in it. The more bandwidth the system has the bigger gets this impact. But it can be calibrated and removed by the following calculation.

$$FD(k) = FD(k) * e^{j\varphi}$$
$$x = -\frac{1}{2} * \frac{k}{n-1}$$
$$\varphi = C_1 x^2 + C_2 x^3 + C_3 x^4$$
$$\varphi = \sum_{i=1}^{N_{coef}} c_i \left(\frac{-k}{2(n-1)}\right)^{i+1}$$

FD: Frequency Domain Signal
k: Wave number
n: Number of samples per A-Scan
N_{coef}: Number of coefficients
c_i: Coefficients

iFFT:
The measured signal of a FD-OCT system is the interference signal of the reference arm and the back reflected light from the sample. This makes it a frequency domain signal. To extract the relative positions of the reflecting layers a inverse Fourier transformation must be applied.

$$SD(k) = \mathfrak{F}^{-1}\left[FD(k)\right]$$

FD: Frequency Domain Signal
SD: Spatial Domain Signal

FFT Shift:
The FFT-Shift step has no physical meaning. It's to present the signal so that DC is in the center and the high frequencies at the boundaries.

Absolute Value:
The inverse Fourier transformation of the FD-signal returns a complex signal that can not be displayed. To get the real signal out of it, one calculates the distance to the origin of the complex plane.

2.2. Analyse of the original Algorithm

As this document is about improving an existing algorithm, the first important step is to analyze which parts of the implementation can be optimized. This chapter shows the original one and points out the weaknesses. To give an overview fig. 2.2 displays the architecture of the original CUDA implementation of the OCT processing algorithm.

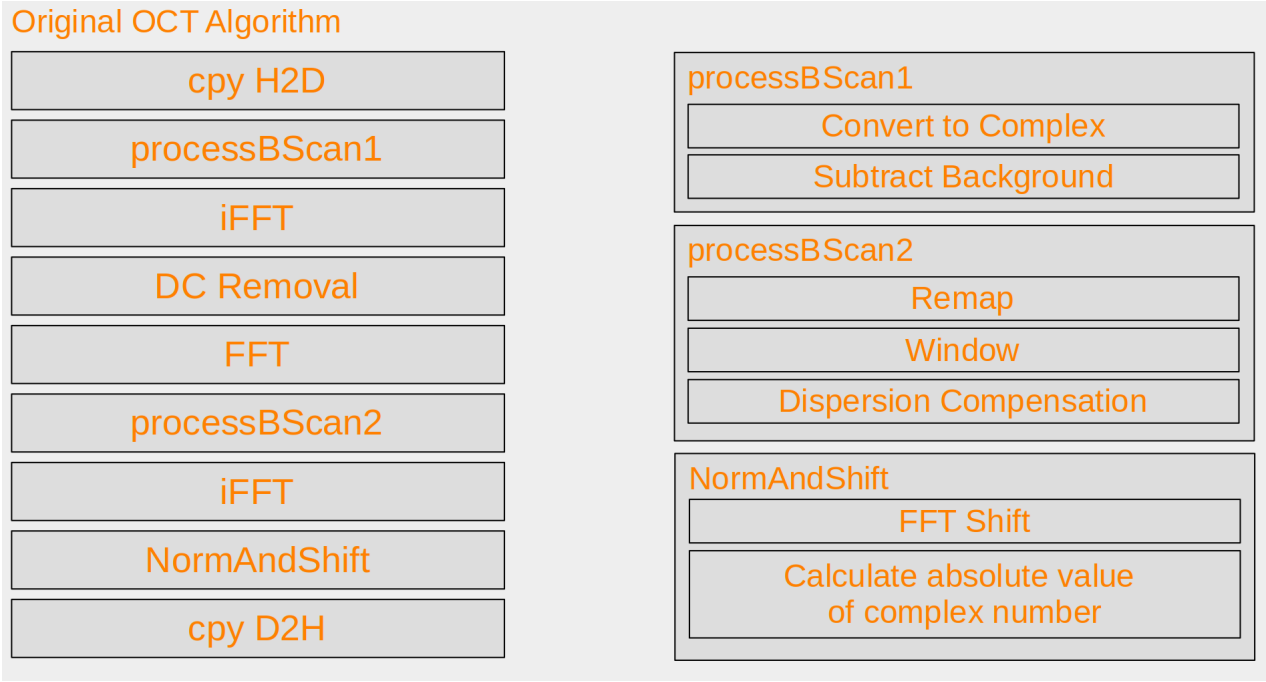

Fig. 2.2.: This black box diagram shows the original implementation of the OCT processing. On the left hand side are the kernels in the order the Central Processing Unit (CPU) calls them. The right hand side shows which kernel executes which step of the OCT processing.

One can see that most of the intermediate steps have to be initiated by the host computer, namely all the one on the left hand side of fig. 2.2. This means that all of those are separate kernels and therefore the data transfer from one to the other must utilize the global memory of the GPU which is quite slow as it takes about 100 cycles to load/store data on it. In addition, it is harder to pipeline the data processing with so many kernels. In fact most of the GPUs do not even allow different kernels to execute in parallel. So it will result in a serialization of the different kernels.

Eye-catching as well is the fact that the algorithm needs FFTs. There is an optimized algorithm provided by NVIDIA to do a FFT but it is still an algorithm that takes considerable time to execute ($nlog(n)$). The reason for two of those FFTs is only the DC removal. There may be a more efficient solution for that.

All the vectors to calculate Background removal, windowing and dispersion compensation are calculated at software start and stored to the global memory of the GPU from which they must be loaded every time. As the calculations of the window and the dispersion compensation are quite small ones. It may be more efficient to just store the coefficients to the constant memory and real-time calculate the values every time. As the background is a measurement it is not as simple to substitute it. But as the background consists of low frequencies only it is possible to condense the amount of data to about 100 points which can be store on the fast constant memory too.

What fig. 2.2 does not show is that in the actual implementation the host computer always sends a B-Scan to the GPU and waits for the result. With the help of streams, it is possible to pipeline B-Scans. This means that download the next B-Scan, upload the previous B-Scan and kernel execution of the actual B-Scan will mostly be done in parallel and not in sequence anymore and therefore result in a big performance gain.

At last, upload as well as download speed can be optimize by configuring the CPU Random Access Memory (RAM) to be not page-able. This means that the memory address of the B-Scan data will always remain at the same place. This means that the Operating System (OS) is not allowed to move this data around to another address inside the RAM.

2.3. New Implementation

This chapter is all about the new implementation of the GPU acceleration CUDA software. As the original implementation has already been a GPU software, this new implementation focuses on specially improve certain aspects. Below a short summary of those improvements:

- Minimize global memory loads and stores by usage of shared memory.
- Minimize global memory loads by calculating some processing steps on the fly instead of looking them up from pre-calculated values inside the global memory.
- Reducing the number of kernel calls.
- Implementing streams for asynchronous data transfer through the Peripheral Component Interconnect Express (PCIe) bus.
- Optimize Grid and Blocks to fully utilize the Hardware.
- Ensure burst transfers for global memory loads.

Fig. 2.3 shows the new kernel architecture which includes the above improvements. Each of those steps will be discussed below.

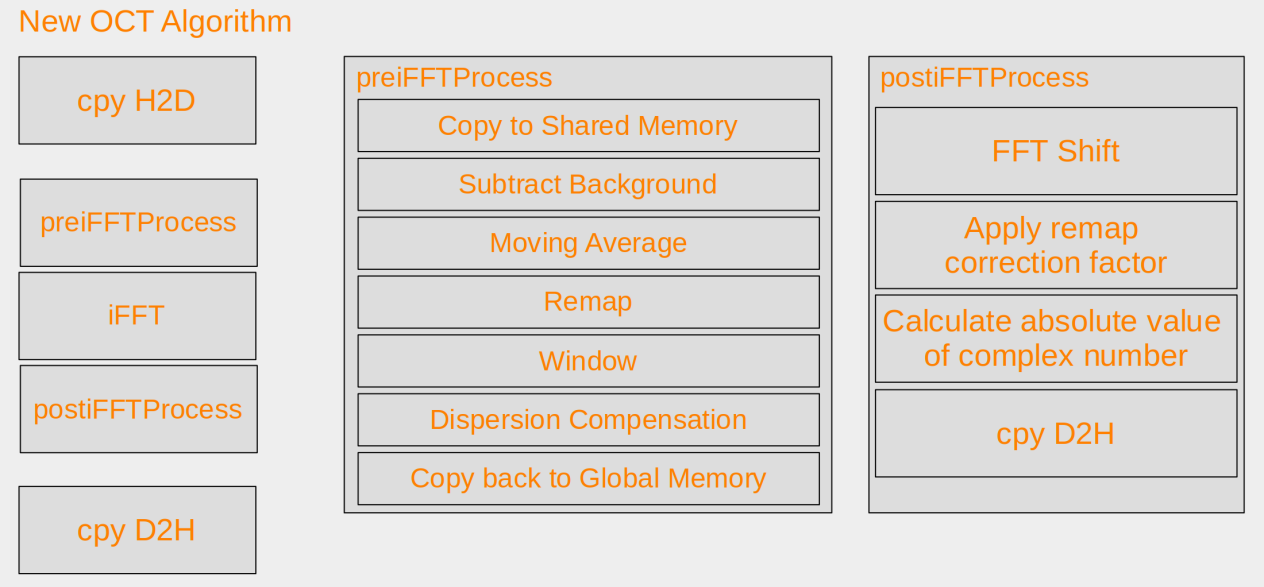

Fig. 2.3.: This black box diagram visualizes the new implementation of the OCT process. The leftmost column shows the kernels in order called by the CPU. The other columns show the tasks executed by each kernel.

2.3.1. Shared memory

To execute the different steps in the algorithm efficiently, a part of the input data is stored inside the shared memory. In the easiest case only one input value is used to calculate one output value which makes shared memory usage very easy. An example for this is the whole rendering process with shared programs. Unfortunately, in this OCT processing algorithm the chosen DC removal as well as the remaping need multiple neighboring input values to calculate an output value. Therefore the boundaries of the shared memory must be treated specially. The way it is done, is by padding the input data on both sides. A small size example shows this principle in the next figure (2.4).

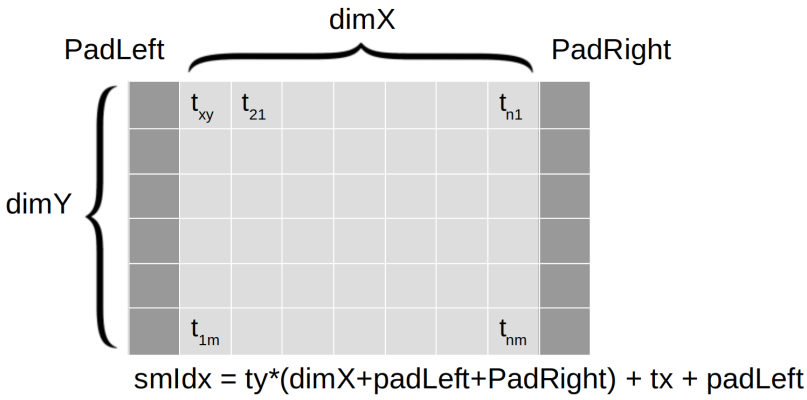

Fig. 2.4.: Principle of Padding in shared memory applications

In this example the data are in 2D matrix format. It turned out to be sufficient when used as a 1D array format, this corresponds to the first line of the example in fig. 2.4.

2.3.2. Background removal

The Background removal is a simple step of the processing. It is just a recording of a data set that is stored to the global memory of the GPU. This processing step simply builds the element wise difference between the new input data and the stored vector for each row.

2.3.3. DC removal

As the NUFFT remaping method delivers better result for a clean analog signal it is important to remove the DC part of the input signal. This is achieved with subtracting the moving-average filtered signal from itself. This results in a very simple kind of high-pass filter. The mathematical formulation of the moving-average filter can be seen in eq. 2.1.

$$p_{SM} = \frac{1}{n}\sum_{i=0}^{n-1} p_{M-1} \tag{2.1}$$

$$n = 25 \tag{2.2}$$

For the actual implementation a kernel size of 25 values has been chosen to build the moving average. [5]

2.3.4. Remaping

To understand the new implementation it is important to understand the NUFFT algorithm that is widely used for OCT remaping first. Fig. 2.5 visualizes the principle of NUFFT. It is basically a convolution and therefore predesignated for parallelism.

NUFFT algorithm:

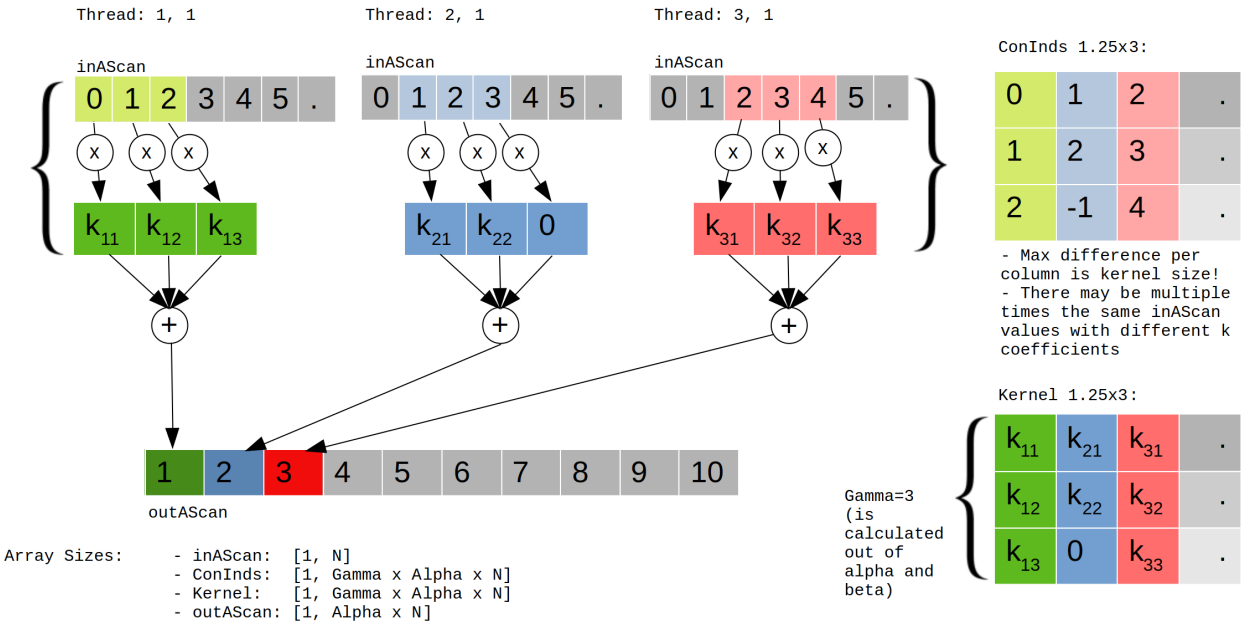

Fig. 2.5.: Principle of NUFFT Algorithm

After investigating this algorithm a little further, two problems are eye-catching. Firstly, due to the factor alpha the dimensions of the input and output data are different. Secondly, as the NUFFT may interpolate different signals, the values used to calculate the interpolation may be near together or not. An example for that can be seen in fig. 2.6. At the start and end of the remap vector the slope is flat. This means the needed values to interpolate are far apart, but in the middle, the slope is steep and therefore the values lay near together.

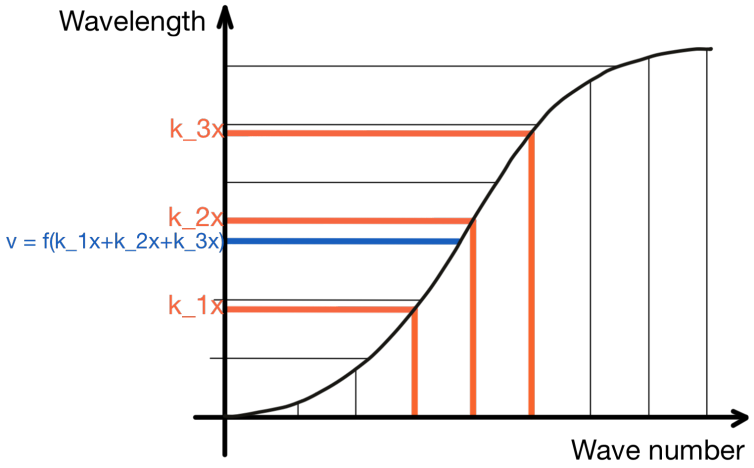

Fig. 2.6.: Example remap vector

The first of those problems will minimize the efficiency of the algorithm because it leaves two choices: Input partitioned data or output partitioned data. With input partitioned data some threads will have to do double the work,

which means all the rest of the threads have to wait for them. With output partitioned data some threads are half of the time of the algorithm without any work to do and are therefore wasted resources. For alphas between 1 and 1.5 the output partitioned data solution is more efficient and for alphas between 1.5 and 2 the input oriented data solution is better.The new implementation of the algorithm uses the output partitioned data solution and is therefore more efficient the smaller the alpha gets.

The second of those problems needs some more attention and is a further reason why to use an output partitioned data solution.
The result of the remap vector creation for NUFFT are two twodimensional arrays. In this document they are referred to ConInds and Kernel. The ConInds Array dimension are kernel size times the output vector length. In a row, the array holds all the incidences of all the values inside the input array that must be used to calculate the corresponding interpolated output value. The Kernel array is of the same size and contains a multiplication factor that is derived from the remap vector. Fig. 2.5 tries to visualize the procedure.

To calculate the NUFFT efficiently it is crucial to further utilize the shared memory. As this is a very limited resource it is not possible and not efficient to load the whole input data array into it. Therefore only the needed input data for the actual block is stored inside the shared memory. An additional array called smStartInds contains the starting indices for each block. It is defined once right after the creation of the ConInds array and stored to the constant memory to minimize the access time to it, as each thread will read it. Fig. 2.7 further illustrates this. It 's basically a convolution with variable input data.

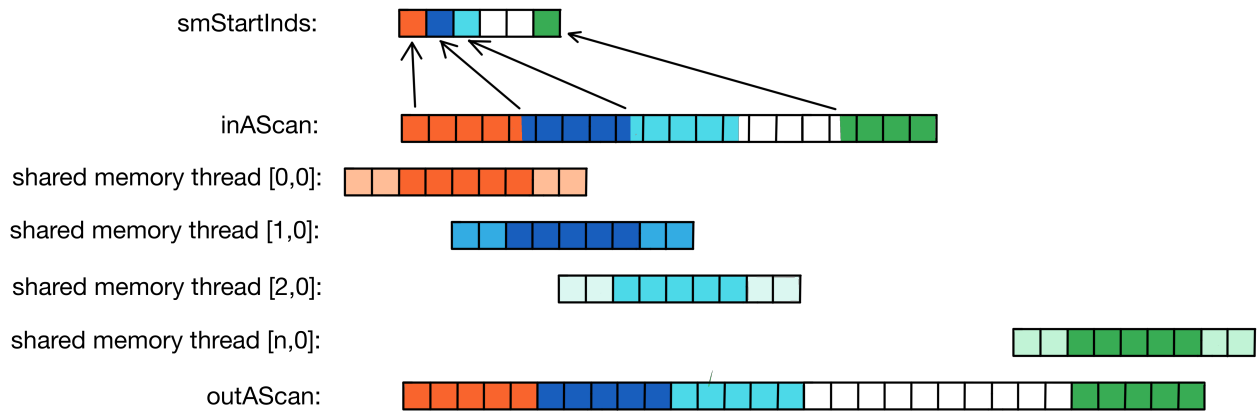

Fig. 2.7.: Example remap vector

2.3.5. Windowing

As the measured data consists of a finite number of data points for an A-Scan the FFT will introduce artifacts. These can be minimized by windowing the measured data. This means that the measurements near the edges are weighted less than the ones in the center. There exists lots of different windows. The old algorithm implemented three of them, namely the Turkey, the Hann and the Hemming window. In the new algorithm there is only the Hann window implemented yet.

2.3.6. Dispersion compensation

The materials used in the reference arm and the sample arm of the OCT system induces dispersion. Explaining this in detail would go beyond the scope of this documentation. But it can be compensated with a simple complex phase shift. This is called dispersion compensation. This calculation is now done real-time to reduce global memory loads.

2.3.7. Steams

The last big optimization is the pipelining of the processing. This is important because the GPU is able to do the uploading, downloading and processing of data in parallel. Only when pipelining the whole OCT processing this feature can be used. Fig. 2.9 shows a pipelined processing in comparison to the non pipelined one in fig. 2.8. As can be seen, the difference is quite large.

Fig. 2.8.: This is the not pipelined algorithm

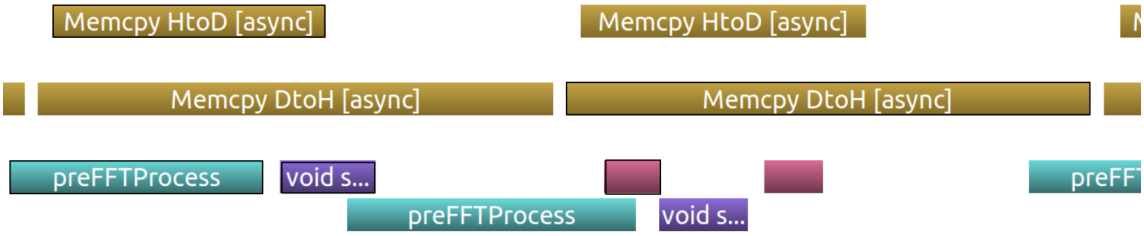

Fig. 2.9.: This is the pipelined algorithm

To implement this pipelined processing CUDA provides a tool called streams. Those streams are a kind of a queue that will be processed as soon as free hardware resources exist. More about streams can be read in the CUDA Programming Guide. [2]

3. Results

This chapter presents the achieved performance of the new algorithm. It compares performance improvements on different graphic cards. It can be seen that the improvements do have a bigger influence on the newer graphic cards. The reason for that is the PCIe link is improving slower than the calculation capabilities of the graphic cards.
The first section of this chapter shows that the new algorithm as good results as the original one. The rest of the sections show the improved performance.

3.1. Output comparison of algorithms

The method to prove that the new algorithm performs as well as the original one was to directly compare the results of the same data set, once generated with the new algorithm and once generated with the original one. The next diagram Fig. 3.1 displays the differences of the algorithms without the dc removal and fig. 3.2 with it .

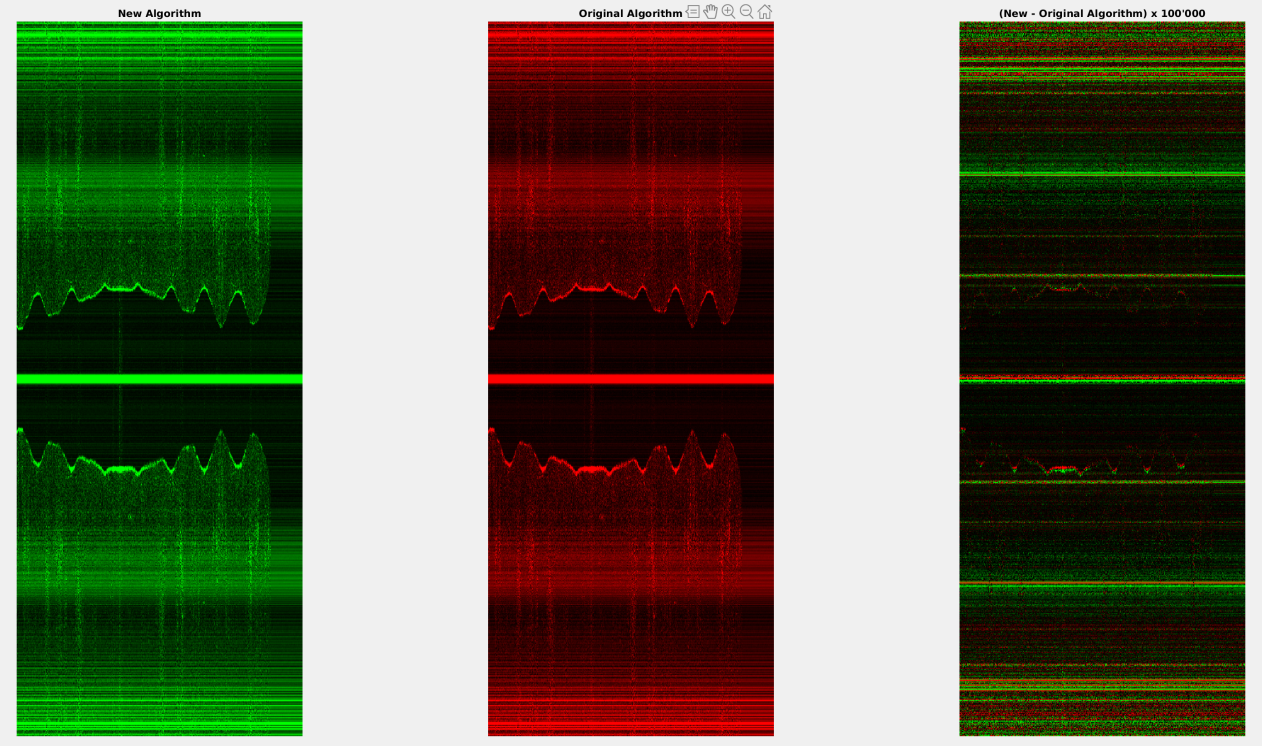

Fig. 3.1.: Algorithm comparison. In the difference image on the right hand side, red means that the original algorithm created a stronger signal and green means that the new one did.

As can be seen is the difference between the algorithms without the DC removal minimal. It needs a amplification of a 100'000 to make it even visible.

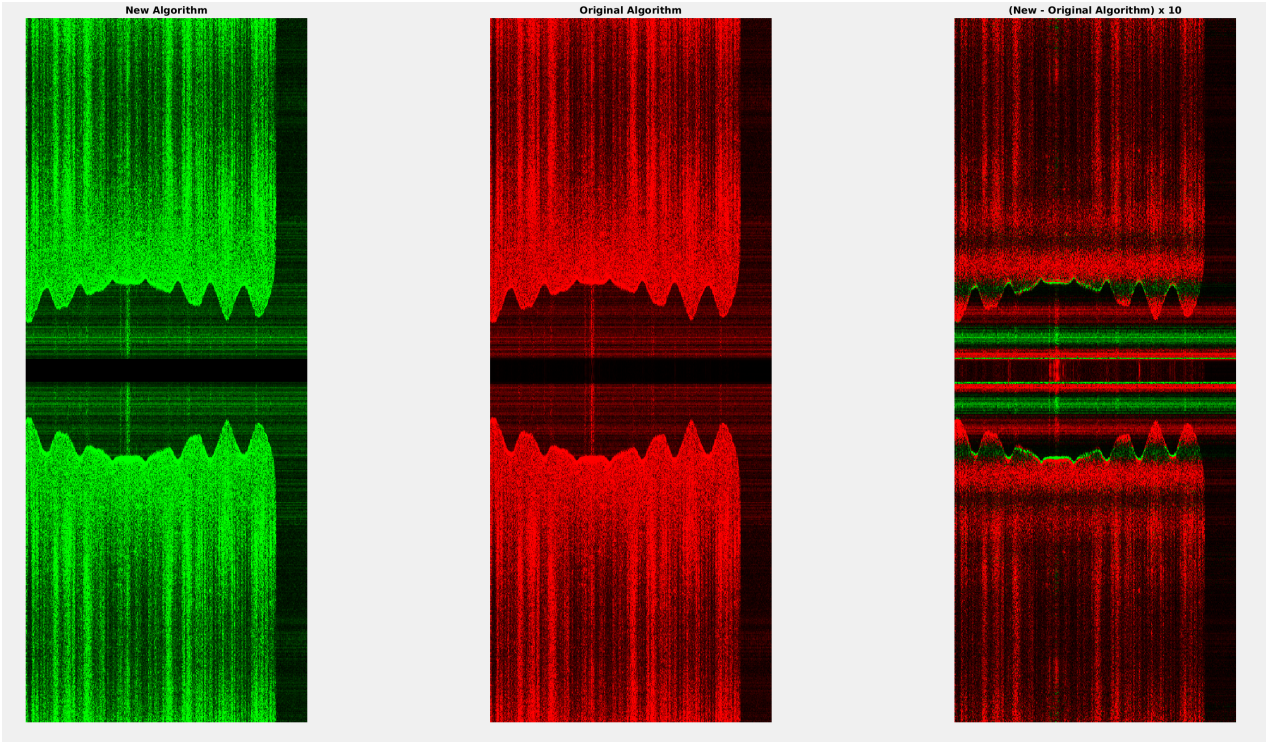

Fig. 3.2.: Algorithm comparison. In the difference image on the right hand side red means that the original algorithm created a stronger signal and green means that the new one did.

The differences between the algorithms with the DC removal is a bit bigger. But this has to be expected as the algorithm for DC removal changed from iFFT-removing dc-FFT to a 1 - moving average Finite Impluls Response (FIR)-filter.
In the image on the right hand side one can recognize a pattern of interleaving green and red sections that is mostly at low frequencies. This is due to the fact, that the side lopes of the two DC removal algorithms are not exactly in phase which results in this pattern. The side lopes of the old algorithm results from the fourier transformation of the recangular window of the deleted low frequencies. The side lopes of the new algorithm comes from the windowed moving average filter. On the next page it will be analyzed which of those effects is bigger.

To do so a mean SNR is calculated from the same raw data B-Scan for each algorithm and compared. The following tabular displays the results of this analysis.

Algorithm	SNR	SNR_{dB}
Original without DC Removal	26.20	32.66 dB
Original with DC Removal	157.24	50.58 dB
New without DC Removal	26.20	32.66 dB
New with DC Removal	190.29	52.49 dB

To Calculate those values the following parts of the B-Scan were used:

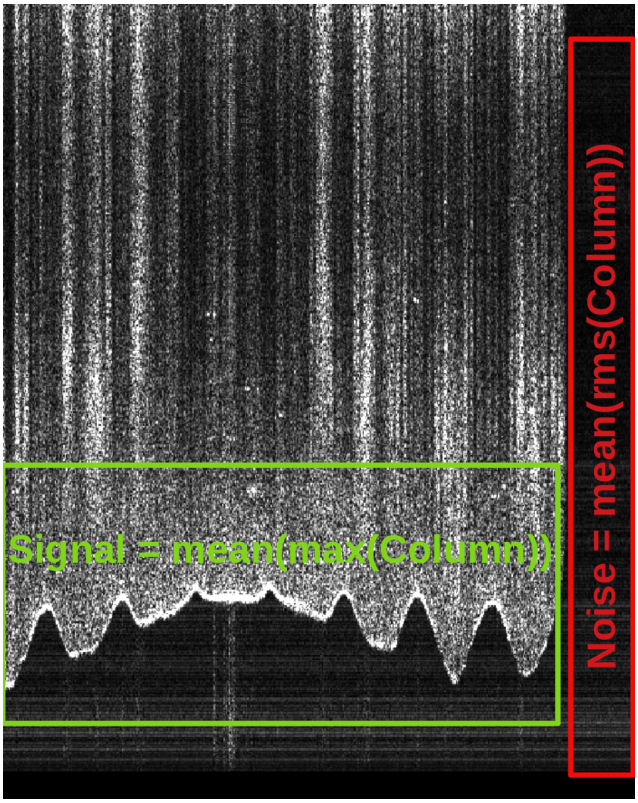

Fig. 3.3.: B-Scan partitioning to calculate SNR.

The comparison shows that the moving average algorithm to remove the DC component results in a SNR that is 1.9dB higher than the original one for the analyzed case. This is not a statistical analysis yet but an indication that the new algorithm will perform at least similarly good.

3.2. Benchmark on NVIDIA K420

3.2.1. Previous GPU algorithm

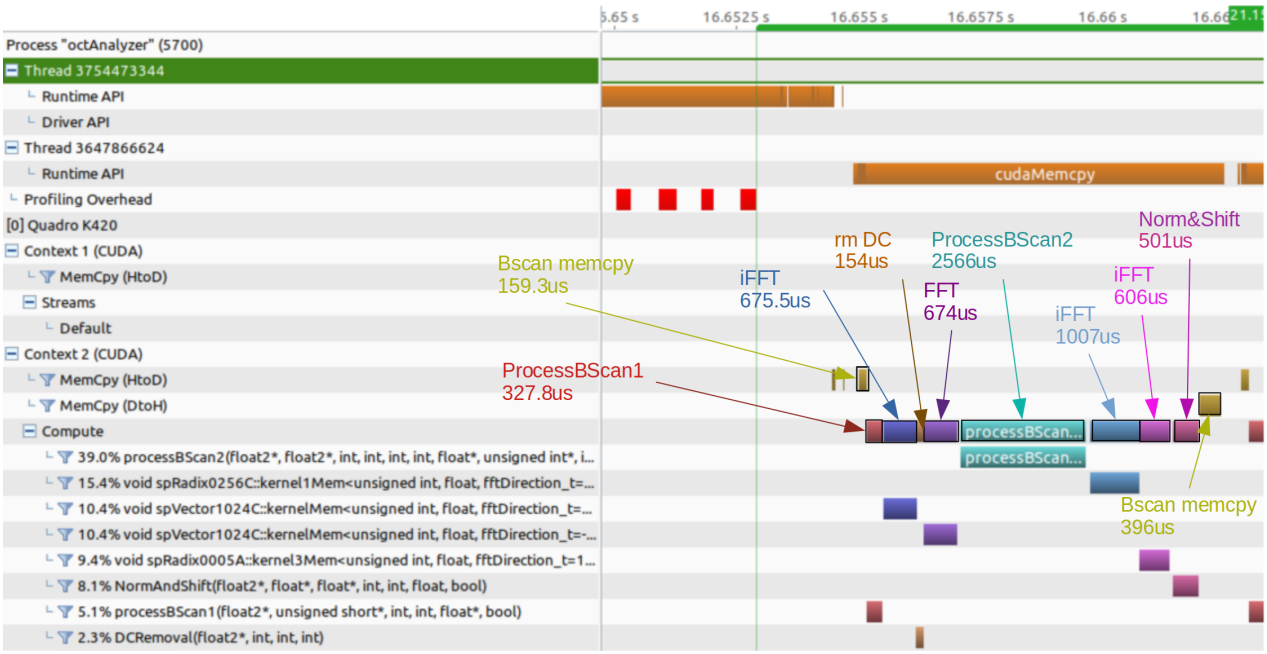

Fig. 3.4.: This profiler diagram shows the performance of the NVIDIA Quadro K420 GPU with the original algorithm implementation.

H2D:	159.3	μs
Processing:	6183	μs
D2H:	396	μs
Total:	6738	μs
Theoretical Throughput:	148	B-Scans/s
Number of A-Scan per B-Scan:	512	
Number of measurements per A-Scan:	1024	
Processing speed:	77.81	MSamples/s

3.2.2. New GPU algorithm

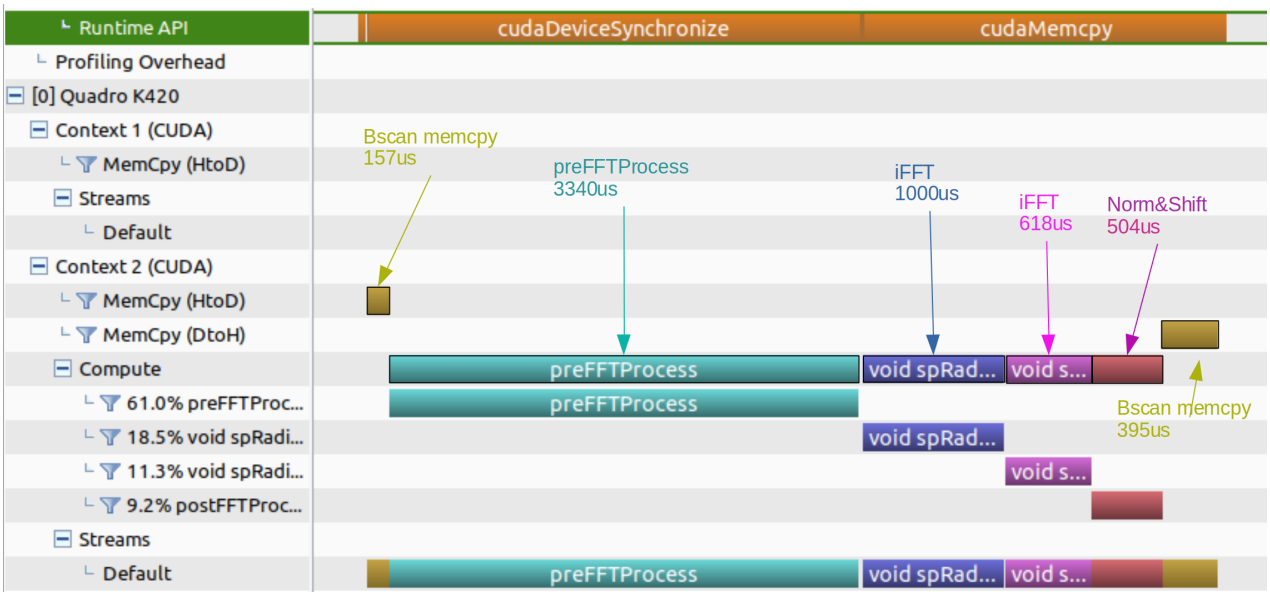

Fig. 3.5.: This profiler diagram shows the performance of the NVIDIA Quadro K420 GPU with the new algorithm implementation.

H2D :	159.3	μs
Processing:	5462	μs
D2H:	396	μs
Total:	6014	μs
Theoretical Throughput:	166	B-Scans/s
Number of A-Scan per B-Scan:	512	
Number of measurements per A-Scan:	1024	
Processing speed:	87.18	MSamples/s
Speedup:	1.12	

The K420 graphics card is not capable of parallel data transfer and data processing. Therefore no big improvement could be made.

3.3. Benchmark on NVIDIA Quadro P1000

3.3.1. Previous GPU algorithm

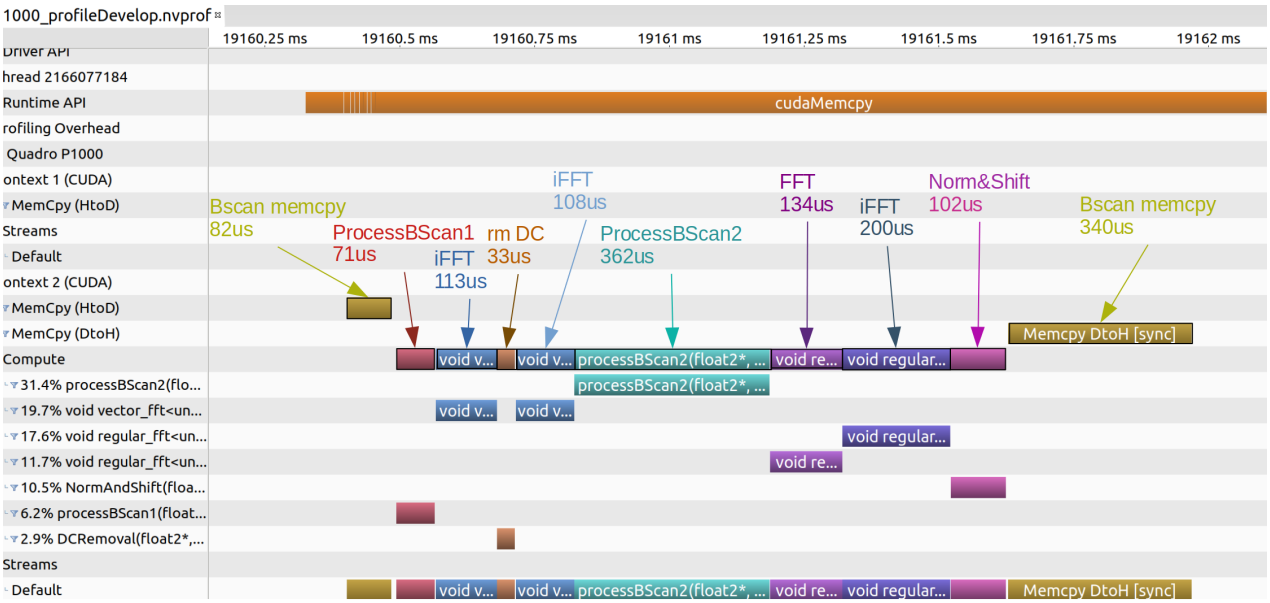

Fig. 3.6.: This profiler diagram shows the performance of the NVIDIA Quadro P1000 GPU with the original algorithm implementation.

H2D:	82	μs
Processing:	1123	μs
D2H:	340	μs
Total:	1545	μs
Theoretical Throughput:	647	B-Scans/s
Number of A-Scan per B-Scan:	512	
Number of measurements per A-Scan:	1024	
Processing speed:	339.3	MSamples/s

3.3.2. New GPU algorithm

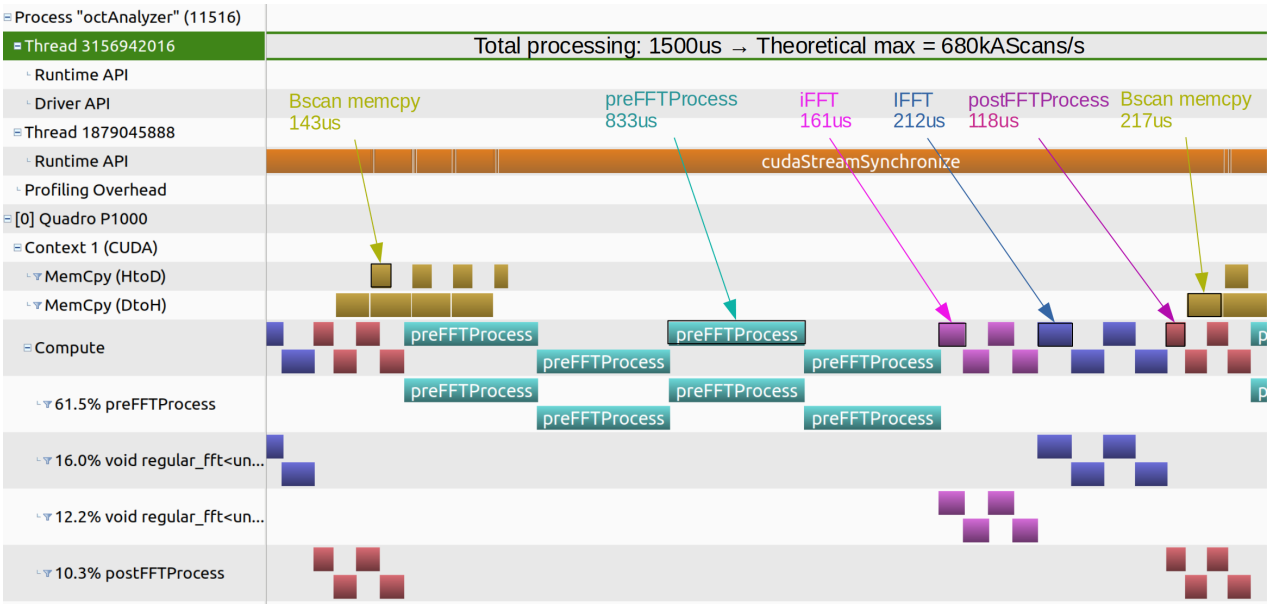

Fig. 3.7.: This profiler diagram shows the performance of the NVIDIA Quadro P1000 GPU with the new algorithm implementation.

H2D :	142	μs
Processing:	1324	μs
D2H:	217	μs
Total:	1324	μs
Theoretical Throughput:	755	B-Scans/s
Number of A-Scan per B-Scan:	512	
Number of measurements per A-Scan:	1024	
Processing speed:	396	MSamples/s
Speedup:	1.17	

3.4. Benchmark on NVIDIA GeForce 1080Ti

3.4.1. Previous GPU algorithm

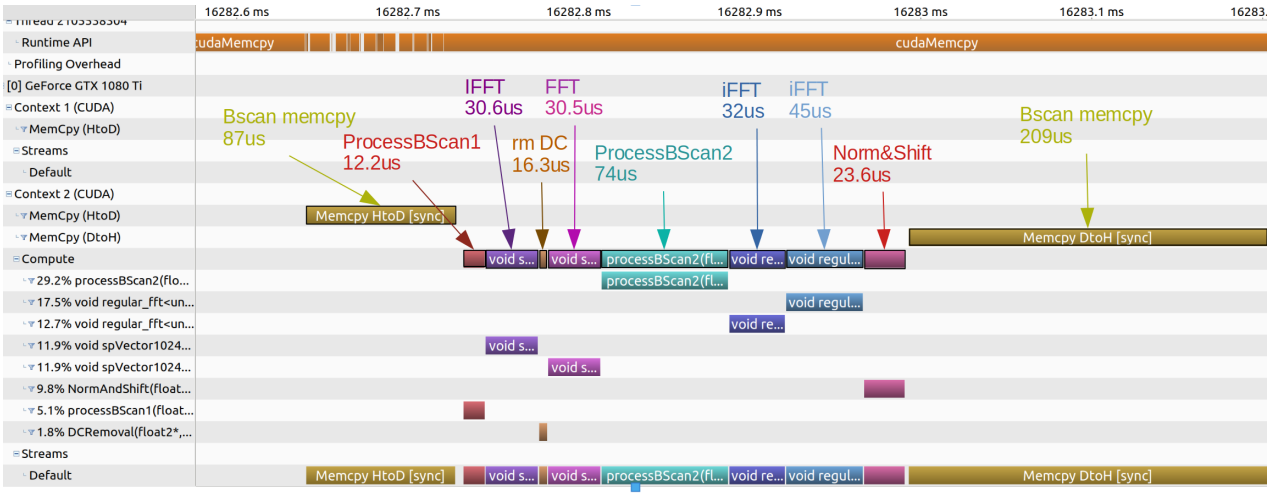

Fig. 3.8.: This profiler diagram shows the performance of the NVIDIA GTX 1080TI GPU with the original algorithm implementation.

H2D:	87	μs
Processing:	264	μs
D2H:	209	μs
Total:	560	μs
Theoretical Throughput:	1785	B-Scans/s
Number of A-Scan per B-Scan:	512	
Number of measurements per A-Scan:	1024	
Processing speed:	936.2	MSamples/s

3.4.2. New GPU algorithm

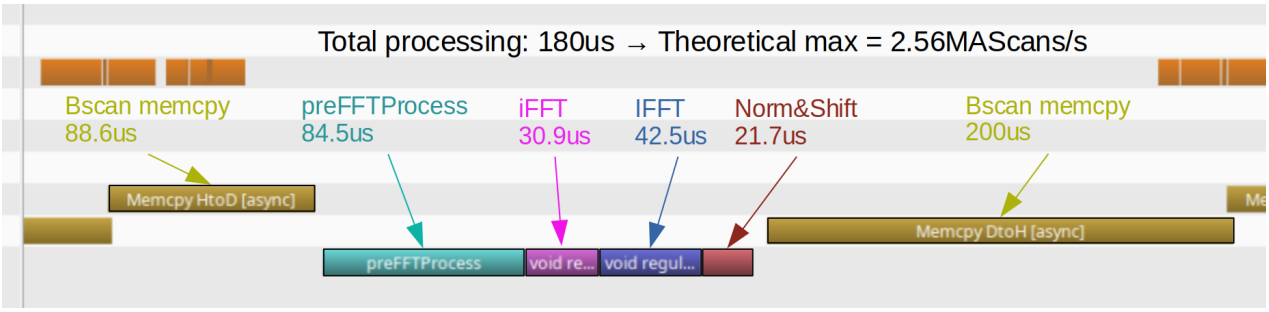

Fig. 3.9.: This profiler diagram shows the performance of the Nvidia GTX 1080ti GPU with the new algorithm implementation.

H2D :	88	μs	
Processing:	179	μs	
D2H:	200	μs	
Total:	200	μs	
Theoretical Throughput:	5000	B-Scans/s	The main difference why the speed up here is big-
Number of A-Scan per B-Scan:	512		
Number of measurements per A-Scan:	1024		
Processing speed:	2.62	GSamples/s	
Speedup:	2.8		

ger as for the Quadro P1000 graphic card is because the processing time is about the same as the memory copy time due to the higher calculation capabilities of the newer GPU. This makes the interleaving with streams way more efficient than for the Quadro P1000.

Fig. 3.10.: This profile shows the pipelining of the algorithm for 4 streams.

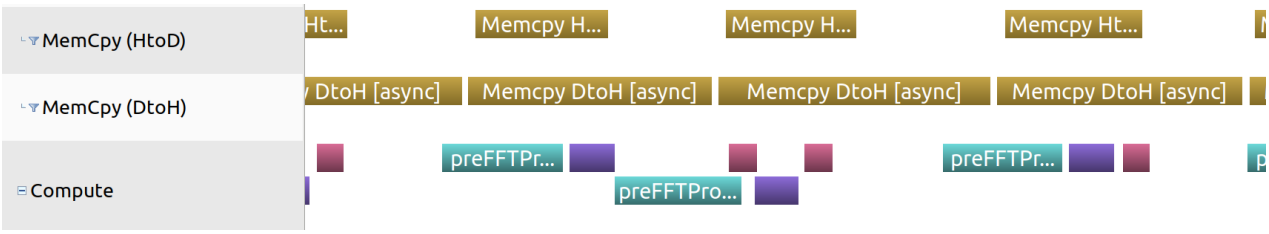

Fig. 3.11.: This profile shows a condensed view which shows that the PCIe bus is the remaining bottleneck.

3.5. Summary

The algorithm has been improved from 560 μs to 200 μs execution time to calculate an A-Scan vector of the size 1024. This is a performance gain factor of 2.8. This results in a processing speed of about 2.5M A-Scans/s. Peak measurements reached even a speed of 2.7M A-Scans/s. The NVIDIA Profiler analysis shows now the new bottleneck of the algorithm is the PCIe connection between CPU and GPU. In the next chapter some further improvement ideas will be discussed.

4. Outlook

4.1. Further algorithm optimization

4.1.1. Smaller background vector

At the moment the background vector is of the same size as the A-Scan raw data. As the background only consist of very small frequencies, which are lots of unnecessary data that prohibits it, to be stored inside the way faster constant memory. This background vector could be condensed to some few numbers that could be interpolated.

4.1.2. Implement configurable states

On each kernel call lots of parameters are transfered from the CPU to the GPU. This could easily be reduced by storing states inside the constant memory.

4.1.3. Direct rendering from GPU global memory

In the actual implementation the bottleneck is the upload via the PCIe connection from the GPU to the CPU. But this data transfer is only of interest when the processed data shall be stored on the computers hard drive. Otherwise one could do the rendering directly from the GPUs memory.

4.2. Framework integration

The actual implementation of the new algorithm was developed in a branch of the HuCE-Optolab cpp framework. All the unnecessary parts were striped away for simplicity. One of the next jobs is to integrate the new CUDA implementation into the original framework.

4.2.1. Factory pattern to set up processing

To be able to choose different processing methods like CPU processing, CUDA processing etc. a factory pattern would be a nice software structure.

4.2.2. Windows compatible interface

As the OCT processing should be a library and not part of the framework one has to generate and link this library. On a Linux system this is straight forward. On a Windows machine one has to create and link the correct dlls.

4.2.3. LabView compatible interface

To be able to use the new algorithm in the HuCE-Optolab LabView framework too, one has to work out a suitable interface. Especially the pipelining part of the software.

4.3. Extend device compatibility with OpenCL

This CUDA implementation of the OCT processing limits its usage to NVIDIA graphic cards. A more general aproach would be OpenCL. There, the integrated Intel graphic chip of most laptops could be utilized. As kernels for CUDA and for OpenCL are similar, this would be a minor effort.

4.4. Direct PCI linking

For very fast data acquisition it may be of interest to directly read the B-Scan raw data from the PCIe connector of the data acquisition device. This is possible for the newer generation of GPUs where CUDA can create a unified memory space. This exists for GPUs with Compute Capability 6 and upwards. Naturally the acquisition device must be able to do such direct transfers of the data too.

Declaration of Authorship

I hereby certify that I composed this work completely unaided and without the use of any sources or resources other than those specified in the bibliography. All text sections not of my authorship are cited as quotations, and accompanied by an exact reference to their origin.

Place, Date: Biel/Bienne, 07. July 2019

Name, Prename: Daniel Tschupp

Signature: 

References

[1] D. B. Kirk and W. mei W. Hwu, *Programming Massively Parallel Processors*. MPS Limited, Chennai, India: Katey Birtcher, 2017.

[2] NVIDIA-Corporation, "Cuda c programming guide." [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[3] S. Rennich, "Cuda c/c++ streams and cuncurrency." [Online]. Available: https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf

[4] Wikipedia, "Hann window." [Online]. Available: https://en.wikipedia.org/wiki/Hann_function

[5] Wikipedia, "Moving average." [Online]. Available: https://en.wikipedia.org/wiki/Moving_average

[6] Wikipedia, "Shannon theorem." [Online]. Available: https://de.wikipedia.org/wiki/Nyquist-Shannon-Abtasttheorem

Glossary

boundary conditions	If the data is not a multiple of the warp size there will be some pading threads that would access the wrong data. Boundary conditions prohibits that.
coalesces	Coalesces means that memory is ordered sequencially (N, N+1, N+2...).
control flow	The control flow is the program execution path of control structures like loops and branches.
corner turning	Technique used to make efficient memory accesses.
D2H	This referes to a device to host transfer which means a data transfer from the GPU to the CPU over the PCIe connection.
device	Device always relates to the GPU that executes the kernels.
DRAM bursts	DRAM bursts are multiple data transfers from DRAM grouped together.
H2D	This referes to a host to device transfer which means a data transfer from the CPU to the GPU over the PCIe connection.
host	Host always relates to the CPU that launches the kernels.
interleaved data distribution	Storing data in separate groups in the memory to use differen banks when accesing the data.
row major	Row major means that the different rows of a matrix is stored sequencial in a 1d array.
Thread Divergence	Threads of a warp may diverge when they follow different control flows.
warp padding	If the data isn't a multiple of the warp size the last warp will be filled regardlessly with threads. That's called padding of warps.

Acronyms

CPU	Central Processing Unit.
CUDA	Compute Unified Device Architecture.
DC	Direct Current.
DRAM	Dynamic Random Access Memory.
FFT	Fast Fourier Transformation.
FIR	Finite Impluls Response.
FPU	Floating Point Unit.
GDDRSDRAM	Graphical Double Data Rate - Synchronous Dynamic Random Access Memory.
GPU	Graphical Processing Unit.
OCT	Optical Coherence Thomography.
OS	Operating System.
PCIe	Peripheral Component Interconnect Express.
PU	Processing Unit.
RAM	Random Access Memory.
SIMD	Single Instruction Multiple Data.
SM	Streaming Multiprocessor.

A. GPU Architecture

In this chapter the architecture of a modern GPU is explained. It is crucial to understand how the GPU works to write and execute fast kernels.

The GPU architecture looks like this:

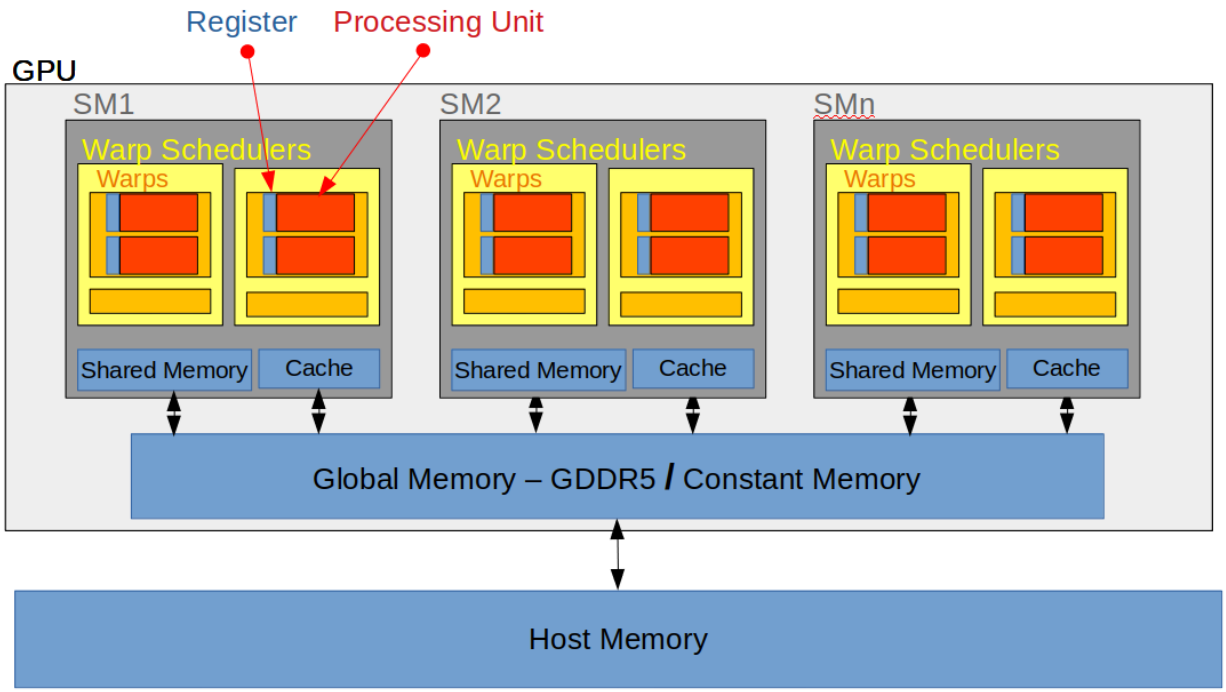

Fig. A.1.: Architecture of a GPU

[1, Parallel Computing Book]

A.1. Cores

Figure A.1 shows the architecture of a GPU. A modern GPU contains multiple so called streaming multiprocessors (SM). These work after the Single Instruction Multiple Data (SIMD) scheme. This means that there is only one instruction line that will be carried out by each processing unit. Special hardware indexers can identify the data on which this instruction operates for each processing unit individually. That is how it is possible to minimize chip-space by only having one instruction core but tons of data cores working on different junks of data.
Each processing unit contains at least 1 FPU, 1 int PU. It may also contain special function units, tensor calculation units, texture filtering unit etc. The floating point unit can calculate 1 float operation per cycle. It needs multiple cycles to calculate doubles and can calculate even 2 16bit floats per cycle. Integer processing unit is build up similarly.
The streaming processor can work on multiple blocks. Those will be split into junks of 32 threads called warp. Those warps are put into a queue and wait for execution. The SM consists of multiple warp schedulers each consists of 32 processing units that will work through the queue. Here it is important to mention that they do not work

linearly through the actual set of threads. They just process the thread until a time consuming instruction like a load/store/branch starts. Then they switch to another set of 32 threads and continue at their actual instruction location. This is done because processing units are build up simple to minimize chip-space so they do not include stuff like branch prediction, pipe-lining etc. to work efficiently through time consuming instructions. Instead they rely on a whole bunch of threads to compute in parallel to bypass such time consuming instruction. Load/store to global memory are the longest instructions. They need about 100 Cycles to complete. This means we would need about 32*100=3200 threads to fully utilize all the processing units while loading / storing data.

A.2. Memory

There are 5 types of memory inside a GPU: Global Memory, Constant Memory, Shared Memory, Caches and Registers. Those can be seen in figure A.1. The first two of those can be written to by the host CPU with an API instruction (cudaMemcpy, cudaMemcpyToSymbol). To the later two there is only access from within the kernel. As Shared memory as well as caches are on chip memories they are very fast but cost also a lot of chip-space. Therefore it is a very limited resource. Registers are even faster than shared memory and cache because there directly embedded in the computing units. The constant memory can only be written to by the host controller but not by a kernel. The kernel has only read access to those. That is because everything that is stored inside the constant memory will be copied to the cache when the kernel launches which makes it very fast, but also limited in space.
The last memory is the global memory which is the biggest and by far the slowest one.

A.2.1. Memory Parallelism

As the access times for DDR is very long (multiple nanoseconds) it is built highly parallel to reach the bandwidth of modern CPUs/GPUs demand. Figure A.2 shows a simplified DDR system. It is basic cell stores one bit of information via a capacitance. Multiple Cells are combined into a block and multiple blocks serves a channel. Each channel has it is own channel controller that is capable of issuing multiple data requests. Those requests are concurring IF they do not request data from the same block. This makes it possible to interleave data requests like it is shown in figure A.3.

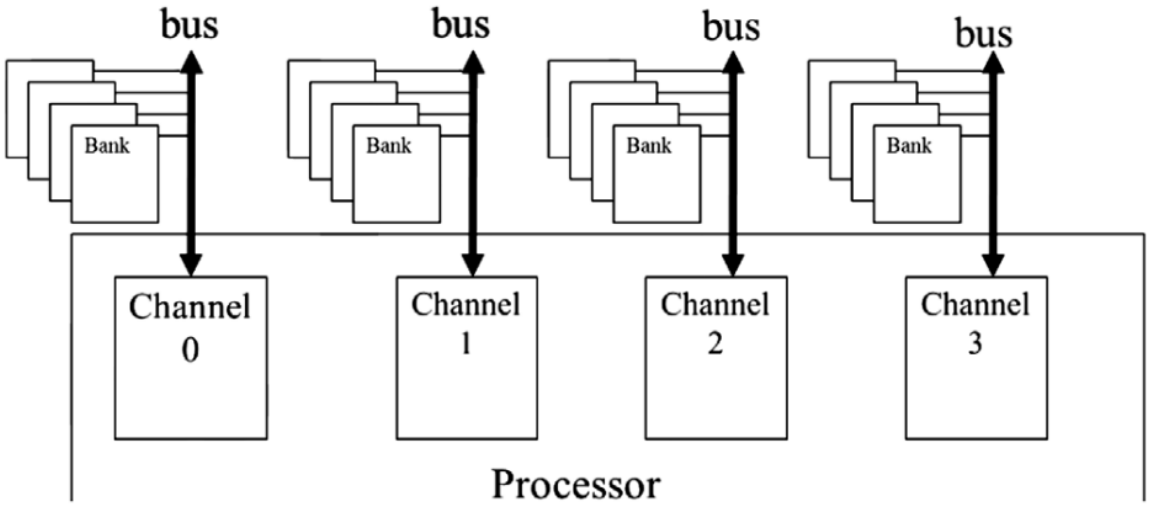

Fig. A.2.: Architecture of GDDR5. [1, Parallel Computing Book, p. 112]

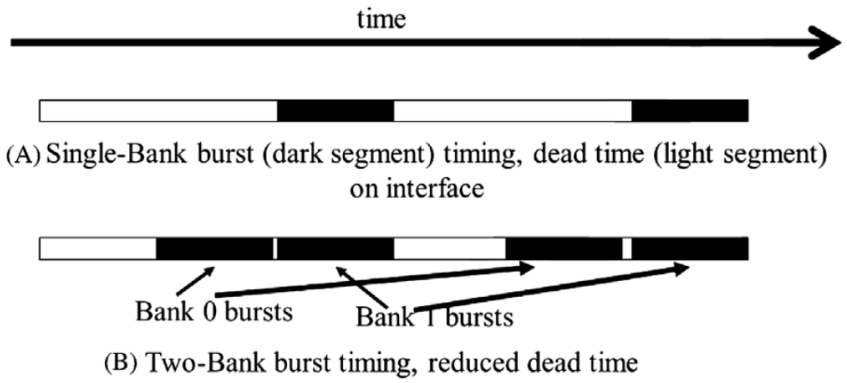

Fig. A.3.: Shows interleaved memory access.[1, Parallel Computing Book, p. 113]

As this is done automatically the programmer must only ensure that enough thread simultaneously access the memory to utilize the whole bandwidth available. As the granularity of those blanks is very fine (128byts = 32floats) the maximal performance will be reached if the thread slots inside the streaming multiprocessors are utilized and a whole warp accesses data in a coalesced way. Coalesced data access is described in the "GPU Programming Techniques"chapter.

B. GPU Programming Techniques

B.1. Data Partitioning

In a SIMD system data partitioning is a mayor topic. It is the basic principle that makes SIMD even possible. There are many ways for data partitioning. Some of them are specialized. For example those for graph searches but this chapter will only cover the three most basic partitioning schemes.

B.1.1. Input Data Partitioning

Whenever one has to apply some kind of mathematical function that depends only on one input element the input data partitioning is a good choice. A thread will be created for each input value. If multiple threads want to write to the same output element proper synchronization is necessary.

Example:

$$\begin{pmatrix} C_{11} & C_{12} & C_{13} \\ C_{12} & C_{22} & C_{23} \\ C_{13} & C_{23} & C_{33} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{12} & A_{22} & A_{23} \\ A_{13} & A_{23} & A_{33} \end{pmatrix} * f(x)$$

As there is only one input element needed to calculate an output element the input data partitioning would be appropriate. In this case no synchronization would be needed for the output writing as each element calculates another output element. This means one could also choose the output data partitioning.

B.1.2. Output Data Partitioning

In the output data partitioning scheme there is one big advantage. Synchronization between different threads is not necessary as each output element gets it's own thread and reading similar input elements is not critical.

Example:

$$\begin{pmatrix} C_{11} & C_{12} & C_{13} \\ C_{12} & C_{22} & C_{23} \\ C_{13} & C_{23} & C_{33} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{12} & A_{22} & A_{23} \\ A_{13} & A_{23} & A_{33} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} & B_{13} \\ B_{12} & B_{22} & B_{23} \\ B_{13} & B_{23} & B_{33} \end{pmatrix}$$

Multiple input elements are needed to calculate one output element. That is why output data partitioning is a good choice.

B.1.3. Intermediate Data Partitioning

The intermediate data partitioning is only possible if the calculation of the output element can be separated into two parts. For example a matrix multiplication. First part: Multiplying each row element with the corresponding column element. Second part: Sum - Reduction to one value.

Example:

$$\begin{pmatrix} C_{11} & C_{12} & C_{13} \\ C_{12} & C_{22} & C_{23} \\ C_{13} & C_{23} & C_{33} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{12} & A_{22} & A_{23} \\ A_{13} & A_{23} & A_{33} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} & B_{13} \\ B_{12} & B_{22} & B_{23} \\ B_{13} & B_{23} & B_{33} \end{pmatrix}$$

$$\begin{pmatrix} D_{11} & D_{12} & D_{13} \\ D_{12} & D_{22} & D_{23} \\ D_{13} & D_{23} & D_{33} \end{pmatrix} = \begin{bmatrix} A_{11} \\ A_{12} \\ A_{13} \end{bmatrix} * \begin{bmatrix} B_{11} & B_{12} & B_{13} \end{bmatrix}$$

$$\begin{pmatrix} E_{11} & E_{12} & E_{13} \\ E_{12} & E_{22} & E_{23} \\ E_{13} & E_{23} & E_{33} \end{pmatrix} = \begin{bmatrix} A_{21} \\ A_{22} \\ A_{23} \end{bmatrix} * \begin{bmatrix} B_{11} & B_{12} & B_{13} \end{bmatrix}$$

$$\dots$$

$$\Rightarrow C = D + E + \dots$$

As displayed in the calculations above a matrix calculation is also possible with intermediate data partitioning. This allows the creation of way more threads and means it is more parallel-able. But it is only an advantage if enough processing units are available and the communication overhead is minimal. With intermediate data partitioning it is possible to use n^3 threads (n: matrix width). In comparison to the output data partitioning with n^2 threads.

B.2. Launching Kernel

A CUDA kernel launch uses some parameters that are described in this chapter. The general syntax is the following:

```
kernelName<<<gridConfig, blockConfig, sharedMemoryByteSize, StreamID>>>(args);
```

B.2.1. Grid, Blocks, Threads

This section describes the general build up of a kernel. As a kernel is the way CUDA implements the SIMD scheme it is crucial to somehow differentiate the input data for each thread. The GPU architecture allows threads to be grouped into blocks and those blocks build a grid. It is the programmers choice how the shape of those blocks and the grid looks like. The architecture only provides some hardware signals to address the x,y,z coordinates of cubic oriented threads as well as x, y, z coordinates for cubic oriented blocks. With the parameter gridConfig/blockConfig one can define how the grid/blocks looks like.

For example:

```
dim3 gridConfig1(3,1,1);
dim3 gridConfig2(3,3,1);
dim3 gridConfig3(3,3,3);
```

GridConfig1 is a sequence of 3 blocks. This is often used for signal processing where the input data is one dimensional. GridConfig2 represents an area of blocks like can be used for image processing. And finally the gridConfig3 represents a volume. Such a structure can be used for example for physical volume simulations. The blockConfig is exactly the same as the gridConfig. It is best practice to orient the grid as well as the block structure on the input data. When

deciding for a structure it is important to bare in mind that threads inside a block can exchange data efficiently via the shared memory but different blocks cannot. But there is a limitation on how many threads a block can handle which is about 1024 to 2048 for the actual GPUs. The number of blocks a grid can handle is in contrary practically unlimited.

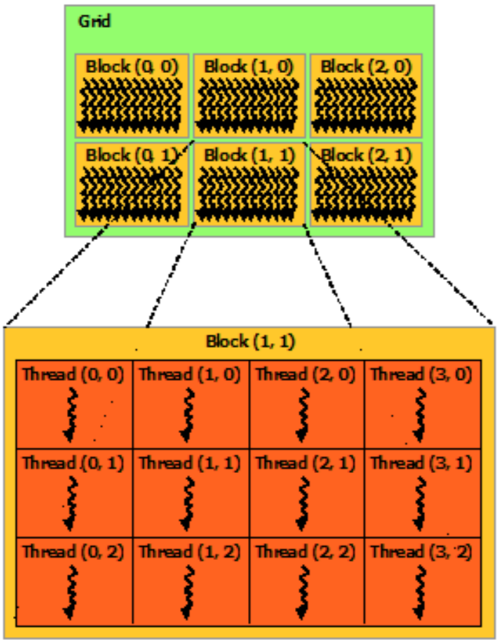

Fig. B.1.: Visualization of the grid and blocks. [2, Cuda Programming Guide]

To identify the data of the actual thread the GPU delivers hardware signals. The following code snippet shows how to use them for a 2d grid architecture:

Kernel Code:

```
__global__ smKernel(int imageWidth){
        int column = blockIdx.x * blockDim.x + threadIdx.x;
        int row = blockIdx.y * blockDim.y + threadIdx.y;

        int actIdx = row * imageWidth + column
}
```

BlockIdx and threadIdx deliver the index in each dimension of the actual thread. BlockDim delivers the block size in each dimension.

B.2.2. Shared Memory Allocation

The third configuration argument of a kernel launch defines the shared memory size. It is an optional parameter. It is only a simple integer value that defines the total size of the shared memory in bytes.

Example: Two Matrices inside the shared memory: **Host Code:**

```
int smSize = ( matrix1Size + matrix2Size ) * sizeof(float);
smKernel<<<gridConfig, blockConfig, smSize>>>(matrix1Size, matrix2Size);
```

Kernel Code:

```
__global__ smKernel(int matrix1Size, int matrix2Size){
        extern __shared__ char sharedMem[];
        float* matrix1 = (float*)sharedMem;
        float* matrix2 = (float*)(sharedMem + matrix1Size*sizeof(float));

        // copy data from global memory into the shared memory
        // do kernel task
}
```

B.2.3. Stream assignment

The last kernel configuration parameter is optional as well and defines which stream should execute the kernel. If left out the default stream 0 will execute it. The stream 0 will always execute kernels in sequence. If they must be computed in parallel one must define streams other than the default one.
Fig. B.2 shows the different options for streams to work together. If just the default stream is used the execution will be a Serial one.

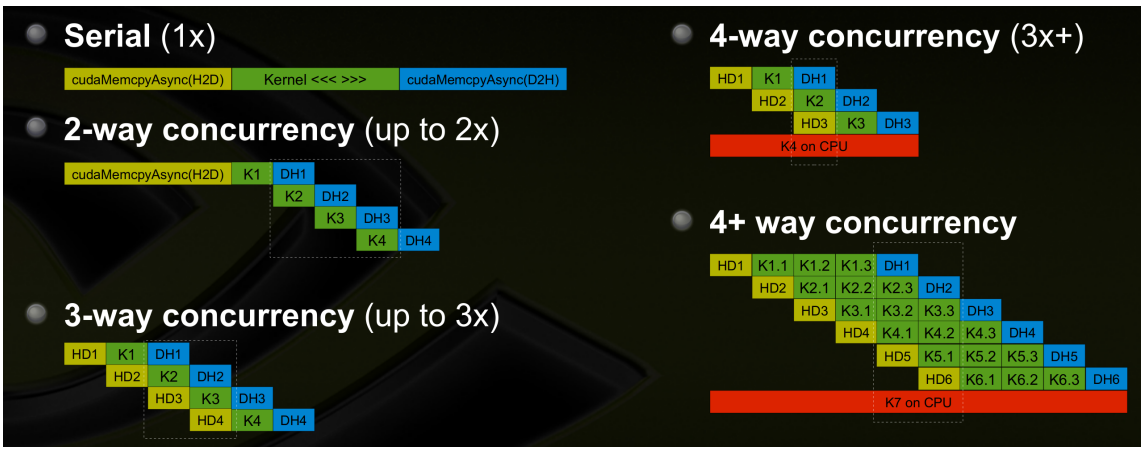

Fig. B.2.: Different possibilities for concurrency. [3, Streams and Cuncurreny Webinar, p. 5]

B.3. Boundary checks

When implementing a kernel that for example makes a convolution with an image gauss kernel it may happen that the boundary output element needs a not existing input element for calculation. This has to be implemented safely and that is called boundary checks. Fig. B.3 illustrates this.

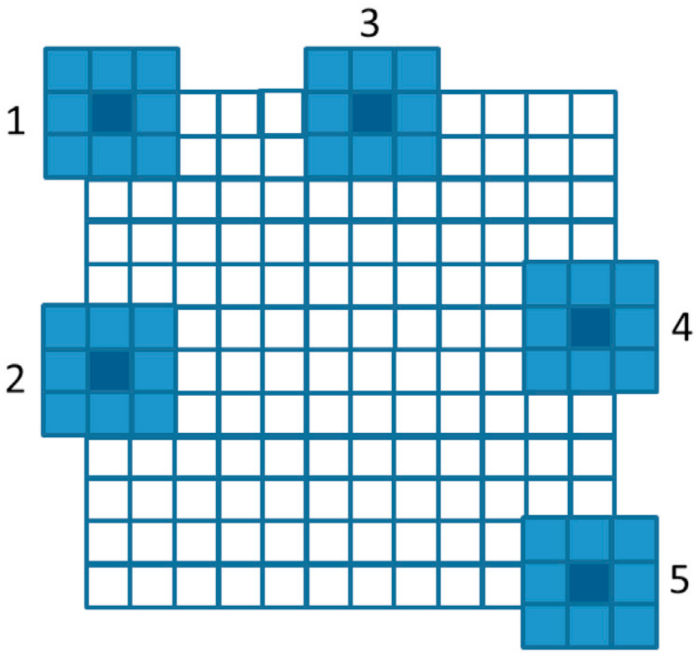

Fig. B.3.: Illustration of boundary check use case.

The implementation of such a boundary check is straight forward:
Kernel Code:

```
 if(actual_column < image_width && actual_row < image_height)
{
        //do stuff with indexes
}
```

B.4. Shared Memory Utilization

The shared memory is a on-chip-memory and therefore way faster than the global GDDR memory of the GPU. But keep in mind that at kernel execution start the shared memory has to be filled with the data out of the global memory. This means shared memory utilization is just rewarding if that data is accessed multiple times!
There may be other benefits from using shared memory as well when it comes to global memory access like the section "Corner Tuning"describes.
The shared memory resource is also a very limited one. If 1000 threads share a 48k shared memory then each thread may utilize 48bytes which corresponds to 12 floating point numbers.

B.5. Memory Coalescing

An important thing for accessing the global memory is to access grouped data as the device can access the global memory only in transactions of 32-, 64- or 128-bytes. Those bytes of one transaction are of sequencial memory locations (Example: N, N+1, N+2...N+64). Therefore if only one float value (4 bytes) is needed 3/4 of the transaction is wasted. Figure B.4 shows a good example for storing a matrix in the global memory. When accessing the data from the kernel in the order the data is stored, the maximum amount of coalescing is used.

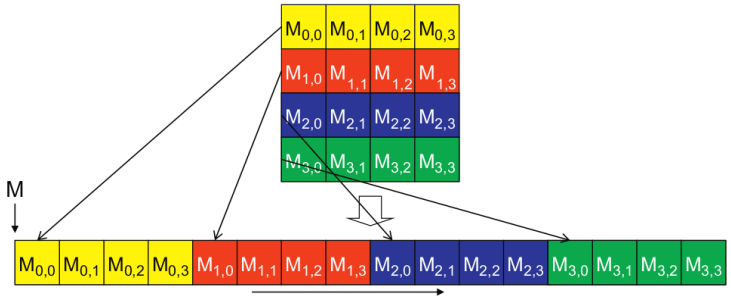

Fig. B.4.: Coalescing memory. [1, Parallel Computing Book, p. 106]

B.6. Corner Tuning

Corner Tuning is a method to access uncoalesced data in a coalesced way. This is done with using the shared memory. As access times do not differ in the shared memory for coalesced and uncoalesced data the solution is simply to copy the uncoalesced data in a coalesced way from the global memory to the shared memory and do the uncoalesced access there like it is shown in figure B.5

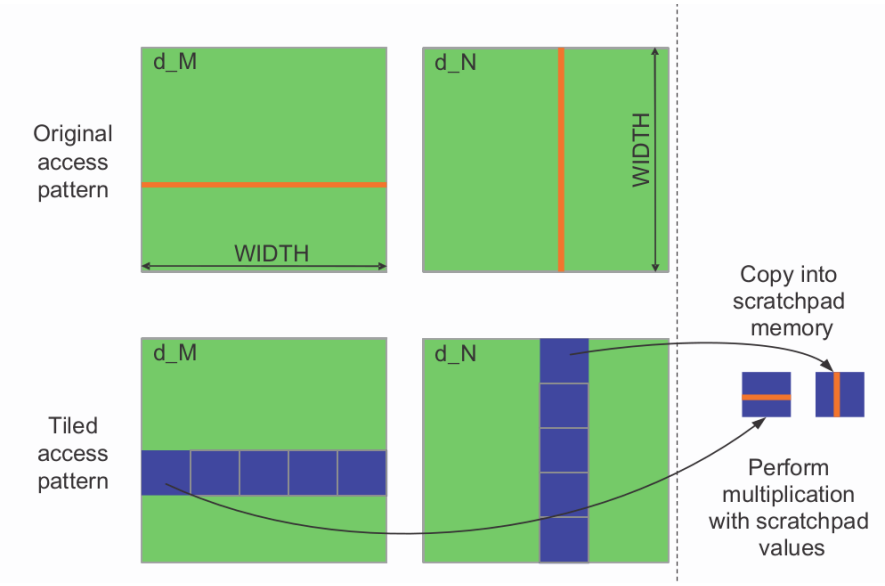

Fig. B.5.: Image shows corner tuning of uncoalesced data. [1, Parallel Computing Book, p. 110]

B.7. Cache utilization

The GPU contains L1 as well as L2 caches like each other processing unit. But the pre-fetching and pipe-lining hardware are nonexistent because they cost lots of chip space. To reduce long waiting time for load/store operations the GPU simply switches to another set of 32 threads(another warp) and continues there. So the cache on the GPU simple acts as a buffer for data read from the global memory and can therefore reduce multiple global memory loads of the same data to just one. This works as long as they are not overwritten which happens very quickly as the cache size is very limited.

B.8. Constant Memory utilization

The trick of the constant memory is that it is written directly to the cache which allows very fast access to that data, similar to shared memory. But it must not be loaded at the beginning of a kernel as it is loaded when transferred to the constant memory. The disadvantage is that it is not allowed to change the values out of the kernel and it is size is very limited (4kB) too.

Example host code for writing to constant memory:

```
#include "device_launch_parameters.h"
#include <cuda_runtime_api.h>

extern float constFactor;

void main(void){
        float factor = 2.5f;
        float* data = getData();
        int dataSize = sizeof(data) / sizeof(float);
        cudaMemcpyToSymbol(&constsFactor, &factor, sizeof(float)));
        kernel<<<1,dataSize>>>(data);
}
```

Example kernel code using constant memory:

```
__constant__ float constFactor;
__global__ kernel(float* data){
        // calculating actual idx
        data[idx] *= constFactor;
}
```

C. Simple Profiling

An easy way to analyze cuda kernel software for performance analysis is the combination of the cmd line tool nvprof to gather the information about the kernel execution and the nvvp tool to visualize them. A short intro into those two tools is given in this chapter.

C.1. Nvidia Proviler NVProf

For simple time line analysis the following command will be sufficient:

```
root@user:Path# nvprof -fo profileName.nvprof ./app
```

The -fo writes the gathered data to the file named profileName.nvprof and overwrites it if it already exists. The ./app executes the application named app at the current location and analyses it.
For a more in-depth analysis of a kernel one could use the –analysis-metrics option but it will slow down the execution extremely!

```
root@user:Path# nvprof -fo profileExtended.nvprof --analysis-metrics ./octAnalyzer
```

C.2. Nvidia Visual Profiler NVVP

Once an application has been profiled with the nvprof tool the exported file can be imported into the nvvp tool. To do so click on: File->Import->Next->Multiple Processes->Browse->Finish.

Then the timeline analysis will be opened. Depending on the application it may be necessary to zoom in quite a bit to see some interesting stuff.
If the –analysis-metrics option was chosen when the app was profiled it is now possible to go to Examine Individual Kernels, choose one and display all the hardware utilization for this kernel on the current graphics card.