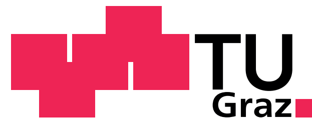

Stefan Brandstätter, BSc

Accelerating Peak Detection in MS Data with Graphics Processing Unit Programming

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: **Biomedical Engineering**

submitted to

Graz, University of Technology

Supervisor

Dr. Gerhard Thallinger

Institute of Computational Biotechnology

Institute of Neural Engineering

Graz, November 2017

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis.

Date	Signature

Acknowledgements

I would like to thank my supervisor, Dr. Gerhard Thallinger, for guiding me through my master thesis. He helped me out when things were not clear, gave input for new approaches and provided quick responses.
Dr. Jürgen Hartler was always very ambitious to help me in the beginning to understand the whole project and furthermore during the implementation.
Last but not least I would like to thank my family and friends for supporting me when I had struggles during the working process of my thesis but also lend my an ear when I was enthusiastic taking about new outcomes. And special thanks to my parents who made it possible to come this far.

Abstract

Objective Lipid Data Analyzer identifies lipids and their structure based on mass spectrometry. A bottle neck of the analysis was the Savitzky-Golay Filter which is used for smoothening the raw data of the chromatogram. This implementation is needed to detect the begin and the end of a peak in the chromatogram without changing the shape and position of the peak significantly. The aim was to speed-up the calculation by filtering in parallel.

Methods To reach the goal, the GPU (Graphic Processing Unit) was invoked to perform the calculation by using the programming model CUDA (Compute Unified Device Architecture). The existing Java code was ported to the C-like programing model. A native library was created which was included into the existing project and tested on three different platforms. If a CUDA capable device is installed, the library is called which uses the GPU.

Results The GPU version of the algorithm led to a speed-up between 7-170 depending on the input data and available graphics card. When smoothing a whole chromatogram file the CUDA version led to a speed-up of 6.5-43 compared to the Java version. Due to different floating point representations there are numerical deviations compared to the Java implementation. From 8084 lipids 28 differ in their result by more than 0.1 %. In average the deviation of the peak area is 0.05 % and the maximum 1.56 %.

Conclusion Enabling the GPU using CUDA leads to a noticeable speed-up of the analysis. The degree of performance enhancement is although very depending on the type of data which is processed and also on the used hardware.

Key Words: Savitzky-Golay; CUDA; Lipid Data Analyzer; Bioinformatics

Contents

1 Introduction **1**
1.1 Central Processing Unit vs. Graphics Processing Unit 2
1.2 Ways to Code for Acceleration 3
1.2.1 CUDA and OpenCL 4
1.2.2 Hardware Implementation of CUDA 5
1.3 Lipid Data Analyzer 7
1.4 Aims of the Thesis 10

2 Methods **11**
2.1 Savitzky–Golay Filter 11
2.2 Development Environment 12
2.3 Result Verification 13
2.4 Time Measurement 13

3 Results **15**
3.1 Implementation with Various Test Drivers 18
3.2 Speed Comparison within the Savitzky-Golay Filter 21
3.3 Speed Comparison within Different Settings 21
3.3.1 GPU Specific Code Changes 21
3.3.2 Variation of Parameters 23
3.4 Speed of Full Chromatogram Files 28
3.5 Deviations 30

4 Discussion **32**
4.1 GPU Accelerated Bioinformatics Algorithms 32
4.2 Filter Migration to the GPU 32
4.3 Hardware and Software Preparation 33
4.4 Test Driver 34
4.5 Speed-Up with Different Settings 36
4.5.1 Chromatogram Length 37
4.5.2 Number of Repetitions 37
4.5.3 Block Size 37
4.5.4 Multiple CPU Threads 38
4.5.5 Number of Chromatograms 38
4.5.6 Hardware Independent Comparison 39
4.6 Speed-Up with whole Chromatogram Files 40

4.7 Deviation of the Numeric Result of the Chromatogram 40
4.7.1 Single Chromatogram 40
4.7.2 Full Chromatogram 41
4.8 Conclusion 42

References **43**

Appendices **46**

Abbreviations

AMD	Advanced Micro Devices
AUR	Arch User Repository
BLAS	Basic Linear Algebra Subprograms
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DNA	Deoxyribonucleic Acid
DSP	Digital Signal Processor
FPGA	Field-Programmable Gate Array
GCC	GNU Compiler Collection
GEO	Gene Expression Omnibus
GFLOPS	Giga Floating Point Operations Per Second
GPGPU	General-Purpose Computing on Graphics Processing Units
GPU	Graphic Processing Unit
IDE	Integrated Development Environment
JNI	Java Native Interface
LC-MS	Liquid Chromatography–Mass Spectrometry
LDA	Lipid Data Analyzer
NCBI	National Center for Biotechnology Information
NGS	Next Generation Sequencing
OpenACC	Open Accelerators
OpenCL	Open Computing Language
RAM	Random-Access Memory
RNA	Ribonucleic Acid
SAXPY	Single-Precision A·X Plus Y
SIMD	Single Instruction, Multiple Data
STL	Standard Template Library
ULP	Unit in the Last Place
VM	Virtual Machine

1 Introduction

The *Human Genome Project* unraveled the DNA base pair sequence of the human genome. This laid the foundations for studying the human genome with computers to help understanding diseases, for identification of mutations, to design medication and simulating their effects and advancement in forensic, to name but a few. With the invention of *next generation sequencing (NGS)* [1] it was possible to sequence 20 megabases in a 5-hour run. About ten years later an Illumina Hiseq 4000 [2] sequences now up to one terabases in three days. Due to this fast improvement the amount of data is roughly doubling every six months. The NCBI Gene Expression Omnibus (GEO) database already contains more than a million samples of diseased and healthy tissue which is publicly available to researchers. This gene expression data is used for analyses and classifications which is done by bioinformatic applications. Not only the huge amount of data makes it a time consuming task but also the complexity of solving it too. For example the Nussinov [3] and also the Zuker [4] RNA folding algorithm have a time complexity of $O(n^3)$, where n is the length of the folded sequence. One way to shorten this time consuming tasks is to find better algorithms which is not trivial. A more common and easier way is to enhance the computational power.

Besides sequence alignment and prediction of molecular structures several other important fields which involve bioinformatics exist. Lipidomics with the goal to understand the physiological and pathological mechanics of lipids and their metabolites in single cells and in the whole organism. With molecular dynamics it is possible to investigate molecular processes to enable a rational design of new materials, test analytical theories or to get a insight into molecular mechanisms. For finding lead compounds in drug discovery or prediction of protein complex structures molecular docking is used. Among other things it is also necessary to simulate mRNA movement within the cytoplasm or the bacterial chemotaxis.

The improvement of the clock speed of CPUs (central processing unit) came nearly to a standstill in the last years. Power consumption and heating problems are the main issue. A workaround for that is to parallelize the computational tasks. A GPU (graphics processing unit) is well suited for this kind of operations since up to several hundred cores can work simultaneously at the same time.

Varré et al. [5] show how more efficient bioinformatics applications can be developed. They also give some insights into many-core processor architectures and parallelism.

Arenas et al. [6] provide a preview of the use of GPUs and also give an overview of software systems. Fang et al. [7] present a comprehensive performance comparison between CUDA and OpenCL.

1.1 Central Processing Unit vs. Graphics Processing Unit

Due to increasing power consumption and heating problems with higher CPU frequencies [8] the clock speed increase came nearly to a standstill. The speed up of modern processors is ensured by using multiple processors which work in parallel. This can be achieved by using multi-core or many-core processors on a normal workstation or by performing the calculations on a compute cluster.

Thanks to the gaming industry, which is using real-time, high-resolution and three-dimensional graphics, a solution with higher memory bandwidth and more computing power than a CPU is available at a low price in the form of GPUs. They are now standard in every personal computer and have up to hundreds of processor cores which provide high parallelism and hence a huge speed-up. CPUs are more suitable to perform operations with low latency (in short time). GPUs on the other hand provide a high throughput (no. of operations per time). With this approach the Moore's Law [9] can be maintained. Moore stated that complex integrated circuits with minimal components costs will double regularly every 12 to 24 mounts.

In the following text the CPU Intel® Core™ i7-6700K and the GPU NVIDIA GeForce GTX 970 are compared (Table 1 including NVIDIA GP102 Titan X). These are at the time of writing state of the art computer components. The CPU has 4 cores and can handle 8 threads whereas the GPU has 1500 cores. Even though the CPU has a clock rate of 4 GHz and the GPU of 1 GHz it is possible to achieve considerable accelerations when performing calculations on the GPU. Power consumption is important as well. With 90 W and 113 GFLOPS (Giga Floating Point Operations Per Second) the CPU has a performance per watt of 1.25 GFLOPS/W. The GPU, however, consumes 145 W and executes 3494 GFLOPS which leads to 24 GFLOPS/W.

Despite the advantages of the GPU there are some restrictions. GPUs use the *single introduction, multiple data (SIMD)* [10] programming model where all processors execute the same instruction with different data at the same time. Additionally, coding is different and algorithms have to be adapted.

Applying scientific computations on a GPU which are normally solved on CPUs is called GPGPU (general-purpose computing on graphics processing units). Early adoptions of computationally challenging scientific problems have been made for dynamic simulation in physics, signal and image processing and visualization techniques [11].

Table 1: Comparison of CPU Intel® Core™ i7-6700K, the GPU NVIDIA GeForce GTX 970 and NVIDIA GP102 Titan X

Feature	**CPU**	**GTX 970**	**GP102**
Cores	4	1,500	3,584
Clock rate (GHz)	4	1	1.4
GPLOPS	113	3,494	10,970
Power consumtion (W)	90	145	250
GFLOPS/W	1.25	24	40.6
DRAM (GB)	-	4	12

1.2 Ways to Code for Acceleration

To implement algorithms on the GPU several frameworks are provided. The most popular programming model is *Compute Unified Device Architecture (CUDA)* [12] (released in 2007) from NVIDIA which is however restricted to NVIDIA products. Apart from that the *Open Computing Language (OpenCL)* [13] (specified in 2008 by the Khronos Group) can be used across heterogeneous platforms consisting of central processing units (CPUs), graphics processing units (GPUs), digital signal processors (DSPs), field-programmable gate arrays (FPGAs) and other processors or hardware accelerators. Another programming standard is *OpenACC (Open Accelerators)* [14] where directive pragmas can be used to copy variables to/from the GPU and specify which *for-loops* should be accelerated. Thrust [15] is a C++ template library for CUDA based on the Standard Template Library (STL). Other available programming interfaces for GPGPU are C++ AMP [16], DirectCompute [17], Jacket [18], OpenHMPP [19] and WebCL [20].

For the most GPGPU implementations CUDA and OpenCL are used. Therefore further work in this study focuses on these two topics.

1.2.1 CUDA and OpenCL

CUDA and OpenCL have different programming interfaces, consisting of code executed on the GPU (the kernel) and code running on the CPU. Converting a CUDA kernel to an OpenCL kernel takes just minimal modifications. The remaining GPU-related code, like setting up the GPU and data transfer, requires some more modifications.

The CPU and GPU with the related storage are declared as *host* and *device* respectively. In both programming interfaces the host controls the device. Many independent *threads* which run the device code, are hierarchically grouped into *blocks*. The blocks are organized in *grids*. Not only computation has its hierarchy, but also memory. The host uses the RAM which is referred to as host memory. Before executing a kernel the required data has to be transferred from the host memory to the global device memory. All threads within a grid can access the global memory. For speed-up parts of the global memory can be copied to the faster shared memory where every thread within a block has access. The fastest memory is the local memory which can be just used by a single thread. A schematic hierarchy is shown in Figure 1. In OpenCL thread, block and grid are named work item, work group and index space respectively. OpenCL also defines the *platform* which are all processors in a heterogeneous system (host plus devices).

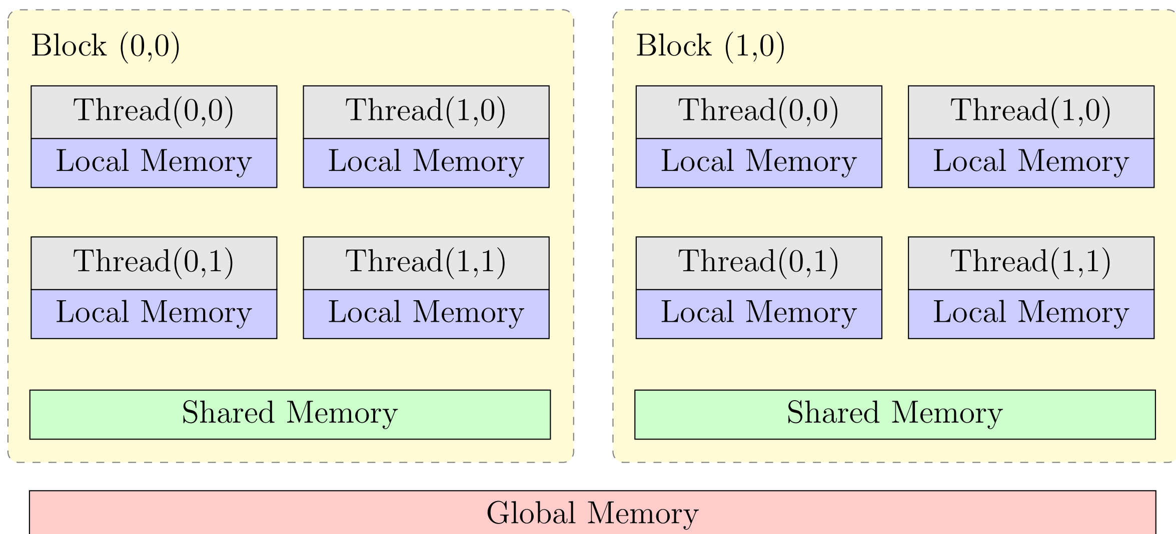

Figure 1: Schematic hierarchy of CUDA. In this example the grid holds two blocks with 2x2 threads each. Every thread has access to its own local memory, the shared memory in the related block and to the global memory (adapted from [21]).

For launching a kernel on the GPU, depenent on the programming interface, several pre and post processing steps are needed. With CUDA memory needs to be allocated on the devics global memory. Before executing the kernel the data needs to be transferred to the global memory. After the calculation is done the result is copied back to the host. In addition to the steps that are needed for CUDA, OpenCL needs several more (Table 2). A more detailed comparison with a SAXPY code example can be found in the Appendix (List 1, 2). SAXPY stands for "Single-Precision A·X Plus Y" and is a specification of the Basic Linear Algebra Subprograms (BLAS). It is used as the "Hello World!" for parallel computing where a vector is multiplied with a scalar and another vector is added (1).

$$\vec{y} \leftarrow \alpha\vec{x} + \vec{y} \tag{1}$$

Table 2: Comparison of the needed operations for CUDA and OpenCL to prepare, load and run a kernel and moving the memory between host and device.

CUDA	OpenCL	Operation
	•	Get information about platform and devices available on system
	•	Select device to use
	•	Create an OpenCL command queue
•	•	Create memory buffers on device
•	•	Transfer data from host to device
	•	Create kernel program object
	•	Build (compile) kernel in-line (or load precompiled binary)
	•	Create OpenCL kernel object
	•	Set kernel arguments
•	•	Execute one or more kernels
•	•	Read kernel memory and copy to host memory

1.2.2 Hardware Implementation of CUDA

The CUDA C programming guide [22] provides a detailed explanation of GPGPU, the programming model, programming interface, hardware implementation, performance guidelines and other helpful information. For this section this guideline served as a reliable source.

When a kernel grid is launched, the blocks of the grid are enumerated and distributed to a scalable array of multithreaded *Streaming Multiprocessors* (SMs). A multiprocessor

consists of a different amount of CUDA cores depending on the compute capability (typically 32-192 CUDA cores). Threads within a thread block and multiple thread blocks (max. depending on the compute capability) execute concurrently on one multiprocessor. One thread block cannot be distributed onto two multiprocessors. After a thread block terminates, a new block is launched on the free multiprocessor.

Warps, which consists of 32 parallel threads, are created, managed, scheduled and executed by the multiprocessor. The multiprocessor partitions new thread blocks which are ready to execute into warps with their own warp scheduler. The scheduler schedules the executions of the thread. Depending on the compute capability there are 1, 2 or 4 warp schedulers for each multiprocessor. On a fully loaded GPU the number of threads can exceed the number of CUDA cores per multiprocessor. The multiprocessor will then have several threads "in flight". A warp with ready threads is selected by a warp scheduler to execute its next introduction. This selection occurs at every instruction issue time. At lower compute capability the number of CUDA cores in a multiprocessor is less than the size of a warp (8, 16 cores). Therefore it can take several clock cycles to issue a single instruction for the whole warp.

Latency is the number of clock cycles it takes to execute the next instruction within a warp. Full utilization is archived when at every clock cycle during a latency period all warp schedulers issue an instruduction on any warp. Latency increases when the input operands are not available yet e.g. input operands are written by some previous instruction(s) whose execution has not completed yet (11-22 clock cycles), or some input operand resides in off-chip memory (200-800 clock cycles).

A warp executes the same instruction for each thread at a time. When all 32 threads agree with their execution path full efficiency is realized. If the path of threads in a warp diverge through a data-dependent conditional branch the distinct executions are processed serially. Threads that are not on the path are disabled. After completion of each branch the threads converge back to a single one and all threads run parallel again. Divergence of a branch only occurs within a warp. Different warps are not effected by each other and act independently whether the code path is common or disjoint.

Threads of a warp are either called active or inactive (disabled). The active ones are on that warp's current execution path whereas the inactive ones are not.

During the whole lifetime of a warp the execution context (program counters, registers,

etc.) processed by a multiprocessor is maintained on-chip. That is why switching from one execution context to another has no cost.

1.3 Lipid Data Analyzer

Lipids are in general hydrophobic substances due to their long hydrocarbon residues. In living organisms lipids are responsible for the structure in cell membranes, serve as a energy storage and are used as signal molecules. Most biological lipids are amphiphilic which means that they consist of a lipophilic hydrocarbon radical and a polar hydrophobic headgroup. As a result they build in polar solvents like water lipisomes, micelles or a lipid bilayer. Fats represent a subset of lipids.

The research of lipidome gives insight in the physiological and pathological processes like metabolic diseases and neurocognitive diseases but also healing effects for multiple sclerosis. Lipids in tissue, organs and body fluids can be detected through mass spectrometry (MS, MS^2).

The Liquid chromatography-mass spectrometry (LC-MS) is an analytical chemistry technique which combines two approaches to quantify compounds. These two techniques and the intensity build the three dimensional data of LC-MS. Through liquid chromatography, compounds in mixtures get separated because of its physical separation capabilities. The mass spectrometry has mass analysis capabilities to get information about the structural identity, so this technique is used for biochemical to identify organic and inorganic components. Therefore this technique is used in biotechnology, environment monitoring, food processing, and pharmaceutical, agrochemical, cosmetic industries.

With the fast improvement of high-throughput measurement technologies, a high amount of data is collected which needs to be processed. LC-MS is capable of monitoring quantitative changes in hundred of lipids simultaneously. To process the data, there are currently two approaches for quantification commonly used: extracting m/z profiles or m/z spectra. For example SECD (Spectrum Extraction from Chromatographic Data) extracts the m/z spectra in a manually chosen area and LIMSA (LIpid Mass Spectrum Analysis) analyzes the output [23, 24]. The latter method extracts the information from the liquid chromatograms like mzMine2 [25, 26].

To use the full potential of LC-MS it is reasonable to use all three dimensions for analyzing

the data. The three dimensions consist of the retention time, m/z ratio and the intensity. Lipid Data Analyzer (LDA) [27] applies this approach. Using all three dimensions, it is easier to distinguish between overlapping peaks, which occurs where the lipids have double bounds.

A computational bottle neck in LDA is the Savitzky-Golay filter [28] which is used to smooth the profiles in m/z-direction and the chromatograms in retention time direction. With this method the signal-to-noise ratio can be increased without degrading the underlying information. This is needed for the peak border detection to work properly by keeping the peak almost at the same place and at the same hight. With a simple low-pass filter the peak would loose noticeably on height and may shift slightly.

To calculate the peak area, LDA performs several steps where some involve the Savitzky-Golay filter (marked with *). Figure 2 illustrates these points.

1. A chromatogram with a broad m/z-range gets extracted.*
2. For every peak in that chromatogram the time is determined.
3. At the time of a peak a m/z-profile with a narrow time-range is extracted.*
4. In this profile the peak and the m/z-peak borders are calculated.
5. Two chromatograms are extracted.* A broad one with the width of the previous calculated m/z-peak borders and a narrow one around the peak. From these two chromatograms the time borders are determined.
6. From the previous determined peak borders in m/z- and time-direction further points in the m/z-time plane are found out which are on the peak border as well.
7. Through these points an ellipse is fitted.
8. Every intensity within this ellipse is added and the background is subtracted. The result is the final peak area.

The Savitzky-Golay filter fits a polynomial to a set of points. These points are in a certain range around the point which should be smoothed. The value of the polynomial, at the position of the point which should be smoothed, is the smoothed value. If the points are evenly spaced a mask can be applied with precalculated weights. Otherwise, as it is in LDA, the polynomial fit has to be calculated for every point with a least square fit.

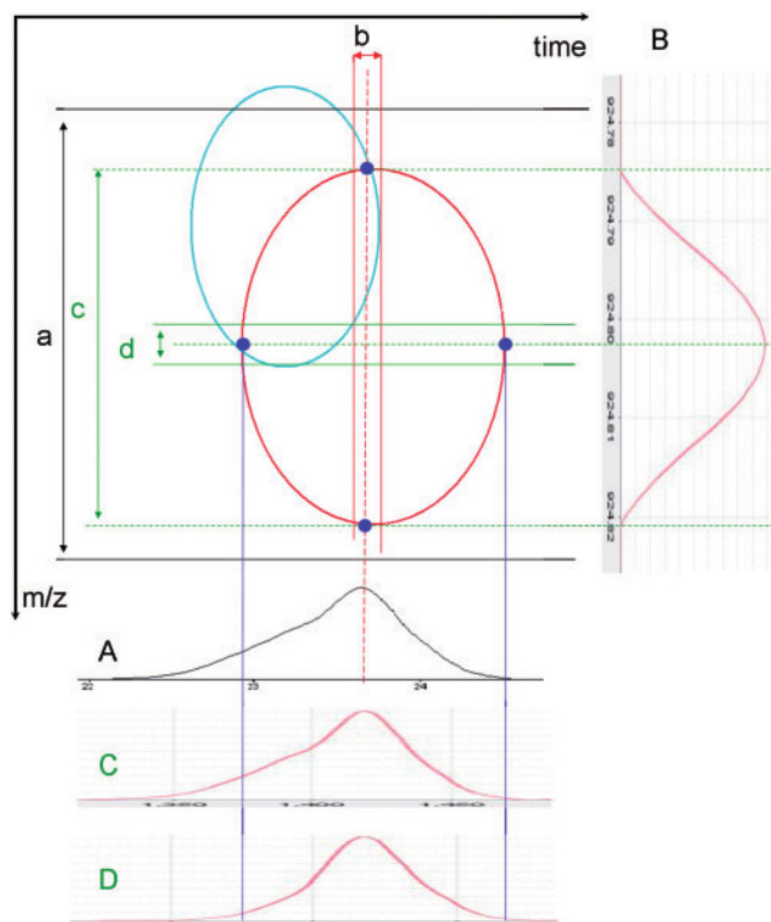

Figure 2: The graph illustrates the m/z-time plane of a LC-MS. The two ellipses represent two peaks where the bigger one is the one to quantify. With a broad m/z range the chromatogram A is extracted. At the time of the peak a narrow m/z profile B is extracted and the borders are defined. Around the peak of B a broad and a narrow chromatogram C, D with the width c, d are extracted. From these chromatograms the borders are defined. An ellipse is fit into the border points. The intensities within this ellipse are contributed to the total peak intensity. Image taken from [27].

1.4 Aims of the Thesis

The overall aim of the thesis is to speed-up the smoothing of chromatograms in LDA which are using the Savitzky-Golay filter, which is the bottleneck of the program, by transferring the computation to the GPU by using CUDA.

Specifically the following should be achieved:

- Get familiar with the Java code of LDA, especially with the filter implementation
- Implementing a C version of the filter including verification
- Implementation of the filter in CUDA including verification
- Create native library which can be loaded from Java for Windows, Mac OS X and Linux
- Integration of the library in LDA
- Determination of speed-up for a single chromatogram and for a whole data file
- Validation of the result

2 Methods

2.1 Savitzky–Golay Filter

The Savitzky-Golay filter [28] performs a polynomial regression to fit piecewise a polynomial function into a set of given points. An analytical solution to this least-squares calculation is available. For each data point a polynomial is fit to the data with the least square method. When the data points are equally spaced the convolution coefficients can be calculated in advance which are depending on the number of neighboring data points and the degree of the polynomial which should be used. These convolution coefficients can also be found in the publication of Savitzky et al. [28]. The data points have to be an odd number where the smoothed point is the central point of the sub-set. The coefficients can be applied with a moving window over the data points.

The general filter equation by Savitzky and Golay can be defined as follows (Equation 2).

$$g_i = \sum_{n=-\frac{m+1}{2}}^{\frac{m+1}{2}} C_n f_{i+n}, \qquad \frac{m+1}{2} \leq i \leq l - \frac{m+1}{2} \tag{2}$$

Where g_i is the smoothed data point at position i, m is the number of neighboring data points (need to be odd), n are the positions of the neighboring points, C are the filter coefficients, f are the raw data points and l the length of data set.

To calculate the solution the polynomial $a_0 + a_1 i + \cdots + a_M i^M$ should be fit to the data values f_{i+n}, where M is the degree of the polynomial. The smoothed data point will be the value of the polynomial at $n = 0$ which corresponds to a_0. To get the solution of the fitting the design matrix A is created (Equation 3).

$$A_{nj} = (t(i+n) - t(i))^j \quad n = -\frac{m+1}{2}, \ldots, \frac{m+1}{2} \quad j = 0, \ldots, M \tag{3}$$

Where $t(i)$ is the time at the position i. Taking the time into account a solution can be found even if the data points are not equally spaced. To get the vector a, the following normal equation with the design matrix A and the raw values f can be derived (Equation 4).

$$a = (A^T \cdot A)^{-1} \cdot A^T \cdot f \tag{4}$$

The filter coefficients C are elements of the matrix $(A^T \cdot A)^{-1} \cdot A^T$. The normal equation can be solved with LU decomposition and backsubstitution. With the calculated vector

a the smoothed value can be determined by getting the first element of the vector a_0.

Varying the window width and changing the smoothing coefficients the way of smoothing can be changed. The Savitzky-Golay filter can perform like a polynomial smoothing, moving average calculation as well as a smoothed differentiation.

The great advantage of the Savitzky-Golay filter is that it does not cut off high frequencies. Maxima, minima and the dispersion stay almost the same and the signal-to-noise ratio is increased as well.

2.2 Development Environment

The implementation should be done on the three major operating systems. LDA is designed to work on Windows, Mac OS X and Linux using Java. Therefore the development environment was prepared for each of these operating systems (Table 3).

For developing a library which invokes CUDA the appropriate software is needed. The NVIDA CUDA Compiler (NVCC) compiles the CUDA specific device code. The standard C code is delivered to a C compiler like the GNU Compiler Collection (GCC) on Linux and OS X or Microsoft Visual C on Windows. NVCC is part of the CUDA Toolkit which can be downloaded from the NVIDIA web page [29].

For writing the actual code an integrated development environment (IDE) is advantageous. On Linux and Mac OS X the development platform NVIDIA Nsight powered by the Eclipse platform [30] is part of the toolkit as well. On Windows it is necessary to install the version of Microsoft Visual Studio that fits the needed CUDA version in advance to work properly.

For the Linux library the code was compiled on a server since no desktop computer was available with Linux and a NVIDIA GPU. On both computers NVCC was installed and Nsight on the local computer. The local and the remote system need to have a matching operating system and the same CUDA version. A Windows remote needs a Windows local system as a Unix remote needs a Unix local system. It is possible to code local and transfer the files via git [31] to the remote system. This can be done automatically when building the library in Nsight.

The CUDA code is platform independent and can be copied between the operating systems without any changes. Changes are only needed if functions are used which need a recent compute capability of the GPU.

2.3 Result Verification

For single chromatograms the raw data gets printed in the console and compared between the CPU version and the GPU version to be sure, that the input is the same. After the smoothing with the Savitzky-Golay filter, the print out of the smoothed values can be compared again. With the result the relative deviation of the GPU implementation can be calculated.

The integration in LDA gets verified by comparing the output files. LDA saves the results in .xlsx files which includes detailed information about the found lipids. All peak areas of the test files get compared and the relative deviation can be calculated (Equation 5).

$$d = \frac{|A_{GPU} - A_{CPU}|}{A_{CPU}} \tag{5}$$

Where d refers to the relative deviation, A_{GPU} to the peak area calculated by the GPU and A_{CPU} to the peak area calculated by the CPU.

2.4 Time Measurement

For single chromatograms the time measurement wanted to be done in the C implementation. But the CUDA function clock(), which counts the elapsed clock cycles, resulted to 0. The measurement got moved to Java close to the JNI call and was done with System.nanoTime().

LDA has already a time measurement implemented and prints it out in the console at every run. This time measurement served as comparison for the CPU and GPU implementation.

Table 3: The used hardware and software for compiling and testing the Savitzky-Golay library. The used CUDA toolkits for all OSs which contain the compiler and IDE can be downloaded from the CUDA Toolkit Archive [29].

Specification	**Windows**	**macOS**	**Linux**	
			local	remote
OS	Win7 Enterprise SP1	MacOS verion 10.10.3	Arch Linux 4.8.6-1-ARCH	Debian GNU/Linux 8 (jessie)
CPU	Intel Core i5-4330M @ 2.8 GHz	Intel Core Duo E8435 @ 3.06 GHz	Intel Core i5 M 480 @ 2.667 GHz	Intel Xeon E5-2630 v2 @ 2.60 GHz
GPU	Quadro K610M	GeForce GT 130	Radeon HD 5400/6300 Series	Tesla K20Xm
CUDA cores	192	48		2688
GPU Clock	954 MHz	500 MHz		732 MHz
DRAM	1 GB	768 MB		2 GB
Compute capability	3.5	1.1		3.5
nvcc	V8.0.60	V6.5.12	V7.5.17	V7.5.17
IDE	Microsoft Visual Studio 15 V14.0.25431.01 Nsight Visual Studio Edition 5.2.0.16321	Nsight V6.5 (Eclipse Edition)	Nsight V7.5 (Eclipse Edition)	

3 Results

Reverse engineering of LDA revealed that the Savitzky-Golay is used in several steps in the peak detection and peak area calculation (Figure 2). After clarifying where the filter is used the next step was to check at which point it would be appropriate to transfer the calculation to the GPU. The function which is performing the smoothing of a single chromatogram (Listing 3) proofed to be the best choice.

The function performs the steps from raw data processing to the smoothed data point and uses the retention time, the raw data, the time range and the smoothing repetitions (Algorithm 1, Figure 4). The retention time is the time a specific compound spends in the column during a chromatography. The raw data corresponds to the intensity of a data point. The data points within the time range are used to fit the polynomial. The function also is able to smooth the spectrum several times which is designated as smoothing repetitions.

To calculate the smoothed value for a given point the points within the time range are determined. These points are used for the polynomial fitting (Figure 3). Therefore a design matrix is created from the time positions of these points. With the design matrix and the raw data a normalequation can be established which can be solved with a LU decomposition and backsubstitution.

In many cases there is no need to smooth the whole chromatogram, but just a short part of it, eg. just the region around a peak. Therefore it is necessary to calculate the indices of the start and stop point of the given time span.

In the serial implementation (Figure 4a) a loop iterates over every single raw data point which should be smoothed. In the parallel version (Figure 4b) each point is handled individually by different threads of the GPU.

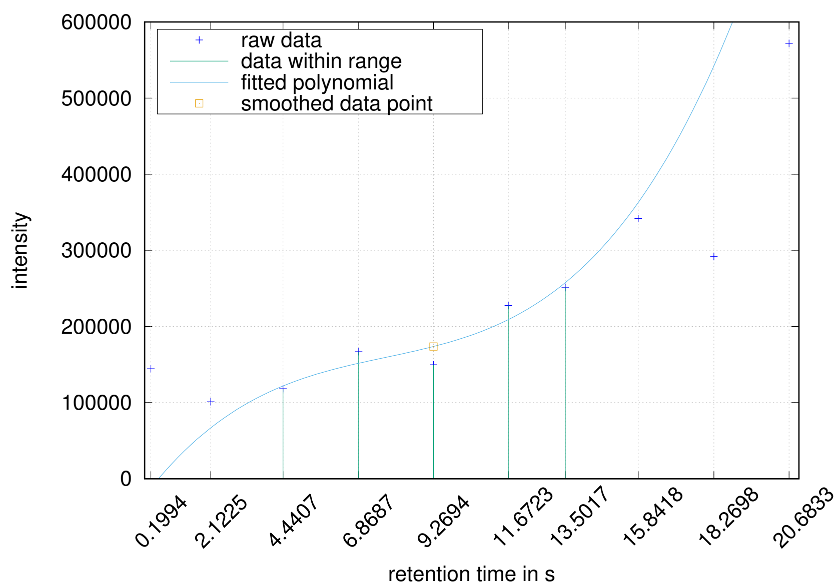

Figure 3: Smoothing of a single datapoint: All these steps are performed on the GPU with a single thread. The data point that should be smoothed is at 9.2694 s and the range is 3 s. The points at 6.8687 s and 11.6723 s are in this range. For fitting the polynomial five points are needed. Therefore the point at 4.4407 s at the lower end and 13.5017 s at the upper end are added. The polynomial is fit into these 5 points. The value of the polynomial at the position 9.2694 s is the smoothed value.

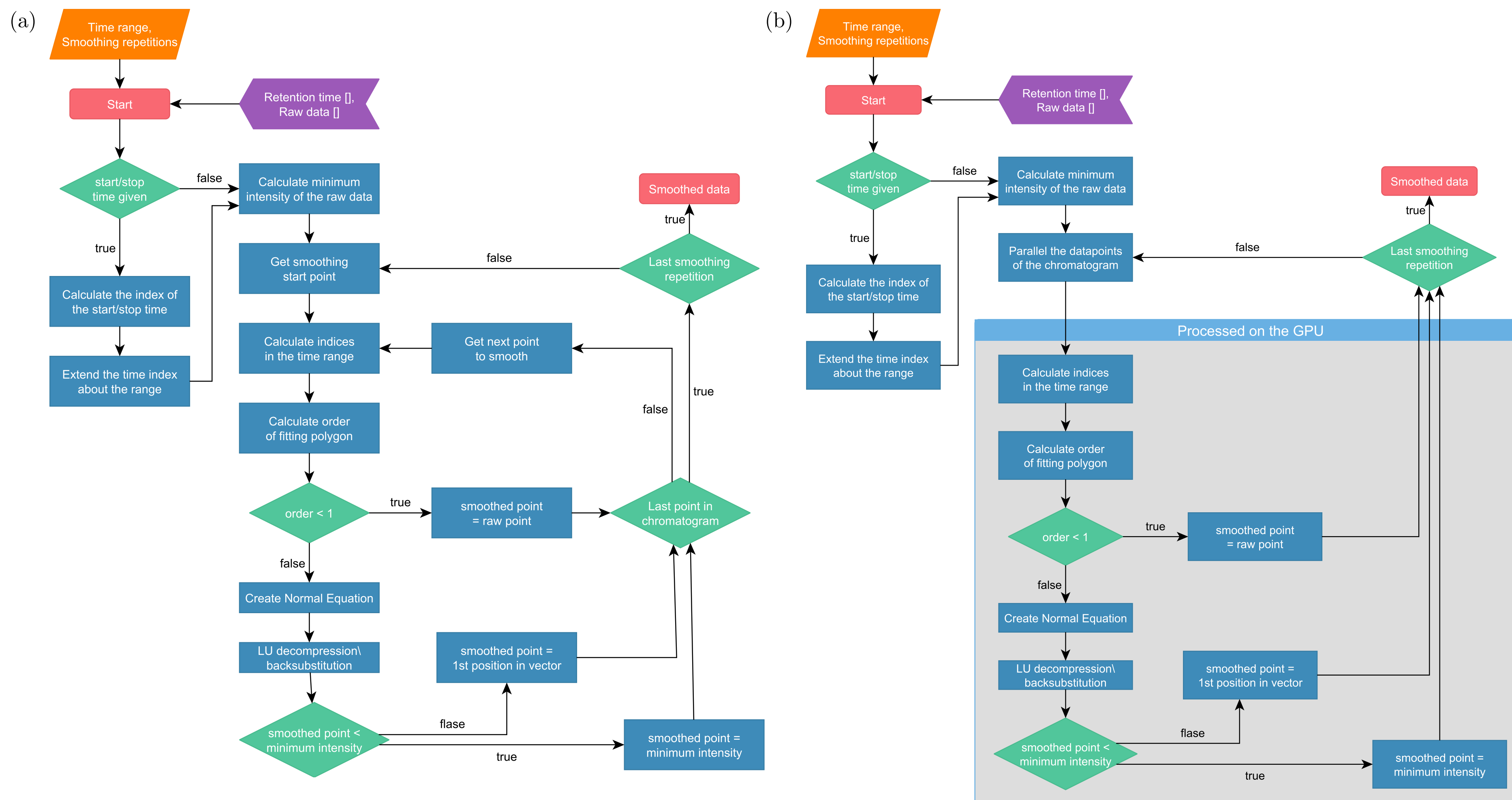

Figure 4: Flowchart of the function for the Savitzky-Golay filter. In both implementations the time range and smoothing repetitions are provided from the LDA at the start as well as the retention time and the raw data from a file. The preprocessing is handled in both implementations on the CPU. The serial polynomial fitting (a) is done with a loop and one data point after the other is smoothed. Whereas the parallel implementation (b) tries to fit as much points as possible at the same time on the GPU.

Algorithm 1: Savitzky-Golay filter running on the GPU with precalculation steps done on the CPU.

```
Input  : retention_time, raw_data, time_range, smoothing_borders
Output: smoothed_value
/* Precalculation steps on the CPU                                    */
threshold ← min(raw_data);
if smoothing_borders ≠ raw_data_borders then
    smoothing_borders ← smoothing_borders + time_rang + 10 indices;
end
calculate grid_dimension dependent on smooth_length;
calculate shared_memory_size;
copy retention_time and raw_data to device;
foreach repetition step do
    /* Kernel running on the GPU to precalculate the powers           */
    foreach raw_data_point per CUDA thread do
        if raw_data_point > 1 then
            precalculated_power ← raw_data_point^(1/4);
        else
            precalculated_power ← 1;
        end
    end
    /* Kernel running on the GPU to smooth a data point               */
    foreach raw_data_point per CUDA thread do
        smoothing_points ← raw_data_point+ points in time_range;
        copy retention_time and raw_data from global to shared memory;
        /* Creation of the normal equation and solving it             */
        create matrix from retention_time and raw_data;
        matrix decomposition;
        create vector from retention_times, smoothing_points and precalculated_power;
        smoothed_value ← matrix and vector backsubstitution;
    end
end
```

3.1 Implementation with Various Test Drivers

Porting Java Code to C Code

Several test drivers were created to get stepwise to the full implementation. Translating the Java function to a C function was essential since CUDA can use the C syntax rules. To test the translated code a single chromatogram was smoothed and compared with the result from Java code.

Porting from CPU to GPU

The for loop, which iterated over every point in the chromatogram for smoothing, was parallelized. At this point it was necessary to assign the appropriate memory (local, shared, global) on the GPU for each parameter and array. Memory was allocated in different hierarchical levels to see if it might give improvements regarding the speed. Other GPU specific changes have been made (Pages 21ff.). The filter parameter and array which were transferred to the device have been checked. The smoothed result was compared against the Java results. In this test driver the time of the different calculation steps were measured to see which part was the most time consuming one. The results of the calculation steps itself were examined as well to see if the algorithm itself can be adjusted to speed-up. The initialization time was measured as well. The Tesla GPU on Linux took between 3 and 5 seconds and the Quadro on Windows just around 500 ms.

Java Native Interface

To access the GPU from Java a Java Native Interface (JNI) [32] was created, to communicate over the C-library with CUDA. Therefore a class was written in Java (example in Listing 4) which contained the function names with a native declaration of the dynamic library which needed to be accessed from Java. The header file (example in Listing 5) for the C code was generated from the Java class by using the program javah [33]. The generated header file is derived from the native declared function names, the package and the class name and must not be changed. The C code with the CUDA calls was compiled as a dynamic library. When the class is created during the runtime of Java the library gets loaded and its functions can be used. The transfer of the function parameters and array and the returned smoothed array were controlled concerning their correctness.

Multithreading on the CPU

LDA performs the analysis using multiple threads. Each thread is handling one mass of interest. Multiple threads were created in this test driver where each thread was accessing the same instance of the native library. Therefore each thread got his own address pointing to its allocated area on the global memory on the graphics card. With this setup multiple tests were performed. The impact of speed influencing parameters have been observed. The block size, the length of the chromatogram, the number of smoothings for each chromatogram (repetitions), the total amount of chromatograms and the number of threads used (Pages 23ff.).

Whole Chrom File Processing

Until that point just specific chromatograms have been processed multiple times in one run for testing purpose. A chrom file was loaded and the chromatograms got smoothed. Therefore the already implemented classes from maspectras were used. LDA makes use of general MS-signal-analysis algorithms of MASPECTRAS [34, 35], and extends them by various peak border detection methods and by a 3D-algorithm for peak integration [27]. All needed classes were copied and adapted to load the file and process its data. First the classes where the smoothing is called were added. Step-by-step the missing classes were checked if they were needed for this test driver. If they were required they have been added. This was repeated until all classes that were needed for reading and processing a chromatogram were added. In this test environment the program was able to run with or without the assistance of the GPU. The speed-up and values of the smoothing results between the CPU and the GPU run have been compared.

LDA Integration

The native library was integrated into LDA where it is appropriate to work. Minor changes to the derived classes enables the usage of the graphics processing unit.

At the beginning of LDA a query is done to check if a CUDA capable device is installed. First the Savitzky-Golay class is created which tries to load the dynamic library. The library will first look after the CUDA runtime library cudart.dll/libcudart.so/libcudart.dylib (dependent on the operating system). This library tries to invoke the NVIDIA driver. If no driver is installed an unsatisfied linking error is thrown. This error is caught and the useCuda flag is set to false, indicating CUDA will not be used. If all necessary libraries can be loaded a function is called that counts the available devices. If zero devices are detected the flag is set to false as well. The user is informed as well that the following calculation is done without any assistance of a graphics card. Otherwise the library with the GPU implementation can be used and the calculations are done on the graphics card.

The functions of MASPECTRAS, that call the Savitzky-Golay filter, were re-implemented in LDA. At the position where the smoothing was called an if statement checks the useCuda flag. If no CUDA device is present the Java implementation was used. If there is a CUDA capable device the functions from the dynamic library are called.

3.2 Speed Comparison within the Savitzky-Golay Filter

The Savitzky-Golay filter is a sequence of different calculation steps. Each step was investigated, if a speed-up of the algorithm itself could be established. The creation of the design matrix A took 50% of the calculation time. All elements of the counter diagonal of this matrix had the same value (Equation 6). This is when the indizes of a_{ij} have the same value $k = i + j$. In the implementation every element of the main diagonal and the elements above have been calculated. So it happened that the same value was calculated several times. After code refactoring the value for each counter diagonal was calculated once (bold in equation 6) and stored at every associated diagonal element.

$$A = \begin{bmatrix} a_{00} & a_{01} & \dots & a_{0j} \\ a_{10} & a_{11} & \dots & a_{1j} \\ \vdots & \vdots & \ddots & \vdots \\ a_{i0} & a_{i1} & \dots & a_{ij} \end{bmatrix} = \begin{bmatrix} \boldsymbol{a_0} & \boldsymbol{a_1} & \dots & \boldsymbol{a_{\frac{k}{2}}} \\ a_1 & a_2 & \dots & \boldsymbol{a_{\frac{k}{2}+1}} \\ \vdots & \vdots & \ddots & \boldsymbol{\vdots} \\ a_{\frac{k}{2}} & a_{\frac{k}{2}+1} & \dots & \boldsymbol{a_k} \end{bmatrix} \tag{6}$$

Rewriting the creation of the design matrix A led to a speed-up of a factor of 1.3 (Table 4).

Table 4: Relative execution time of the different steps in the Savitzky-Golay filter before and after optimizing the creation of the matrix. A kernel call is 100%.

	before optimization	**after optimization**
Initiation of the variables	1.40%	1.87%
Calculation of the boundaries	3.86%	5.14%
Creation of the matrix	**50.35%**	**33.88%**
LU decomposition	8.85%	11.78%
Creation of the vector	29.69%	39.54%
LU back substitution	5.85%	7.79%

3.3 Speed Comparison within Different Settings

3.3.1 GPU Specific Code Changes

By plainly copying and adapting the Java code to CUDA computation time decreased by a factor of 2 which was not expected. To work against this issue the way a GPU works

should be considered (Table 5).

The size of some arrays depend on the order of the polynomial which is fitted to the data points. First the memory of these arrays was allocated on the device which led to a slow down of the whole implementation. The next step was allocating the memory from the host on the global memory which led to a speed-up of 7.81. Last but not least the arrays had a fixed size, so that calculations with the maximum order are possible, and stored on the local memory. The last step accelerated the code to a speed-up of 9.71.

In GPU programming if statements and resulting branching should be avoided. At different points in the algorithm the same if statement was executed multiple times which led to perform the same calculation multiple times as well. If the raw data value was greater than 1 the fourth root of the value was multiplied with another parameter. The else statement was empty and the parameter stayed unchanged. In the revised version the fourth root of the raw data was set to the value *power* if the raw data was greater than 1. Otherwise *power* was set to one. In the code, everywhere where needed, *power* was just multiplied instead of working off the if statement. Additionally the function $pow(x, 1/4)$, which calculates the fourth root, was replaced with $rsqrt(rsqrt(x))$, which calculates two times the reciprocal of the square root of a value which equals the fourth root. This change improved the speed to a factor of 10.85.

The last improvement was moving the values of the chromatogram and the precalculated power from the global memory to the shared memory. With all the previous changes this led to a speed-up of 13.44. All these tests have been examined with one thread, a block size of 128, a chromatogram with the length of 2688 and 8 repetitions.

Table 5: Speed-up of the Savitzky-Golay filter at different stages of the GPU optimization.

Change	**Speed-up**
Plain copy from CPU to GPU	0.58
+ Move dynamic memory allocation from kernel to host	7.81
+ Fixed memory size on local memory	9.71
+ Avoided branching	10.85
+ Moved global memory to shared memory	13.44

3.3.2 Variation of Parameters

The parameters that will change from calculation to calculation are the length of the chromatogram, the repetitions and the number of chromatograms which should be smoothed. Depending on the GPU the block size may vary. CUDA provides a function that determines, depending on the specific GPU and the used kernel function, which block size is optimal. This can be done with *cudaOccupancyMaxPotentialBlockSize()*. The CPU differ from the capability of handling multiple threads. Each thread is taking care of one mass of interest and invokes the GPU when the Savitzky-Golay filter is needed. All these parameters were changed in a certain range to see the effect. With longer chromatograms and more repetitions the speed-up increases linearly (Figure 5). Varying the block size and the number of chromatograms had nearly no effect (Figure 5, Figure 6). Increasing the number of CPU threads leads to exponential decay to a fixed speed-up (Figure 5).

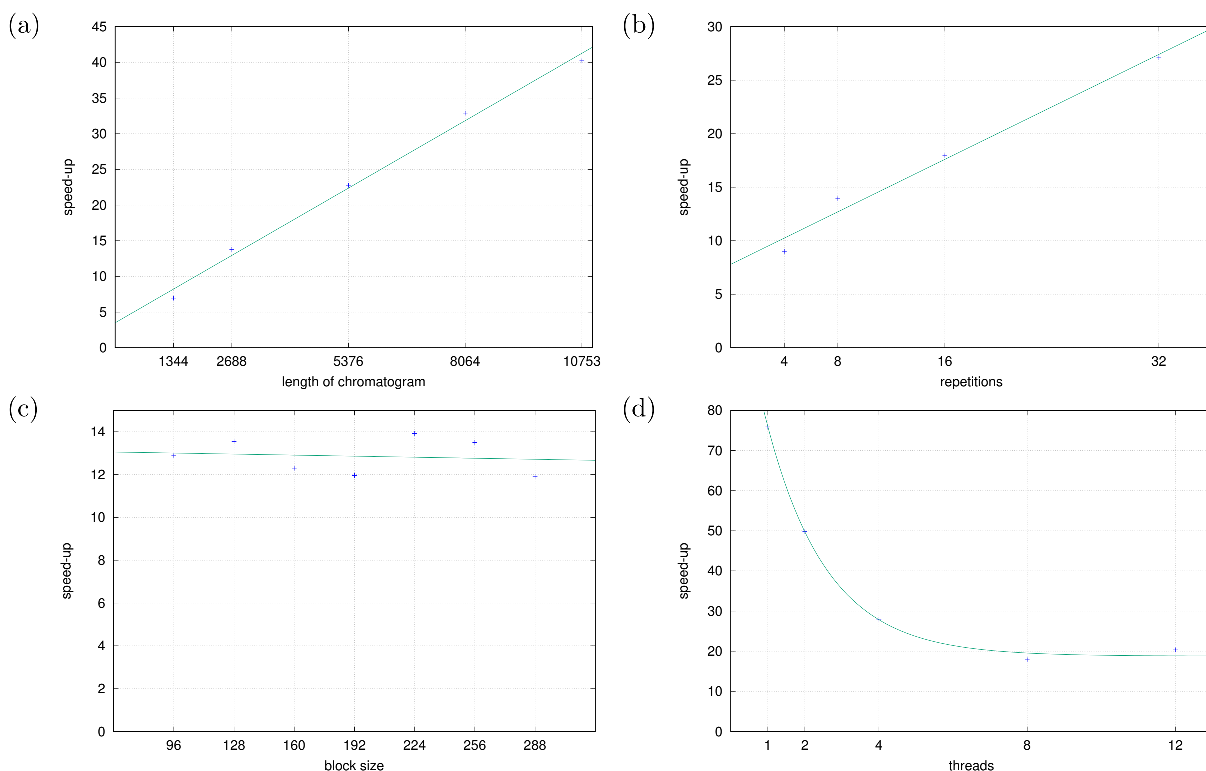

Figure 5: Speed-up changes by varying several parameters. The length of the smoothed chromatogram (a), the repeats (b), the block size (c) and the number of threads (d) have been increased. The default values have been a chromatogram with the length of 2688, 8 repetitions, a block size of 224 and a single thread.

The number of chromatograms was increased on all three systems to compare their GPUs (Figure 6, Table 6). The measured calculation times and the speed-ups of the three different GPUs stayed for each operating system nearly the same. The Tesla graphics

processor has the most CUDA cores and has been used to simulate the other two GPUs to get a hardware independent comparison with a different amount of CUDA cores. When launching the kernel the number of threads have been limited to the amount of CUDA cores of the comparing GPU. Windows is using a Quadro with 192 cores and the Mac uses a GeForce with 48 cores (hardware specifications in Table 3). The speed-ups of the simulated GPUs on the Tesla are lower than on the native GPUs (Figure 6, Table 7).

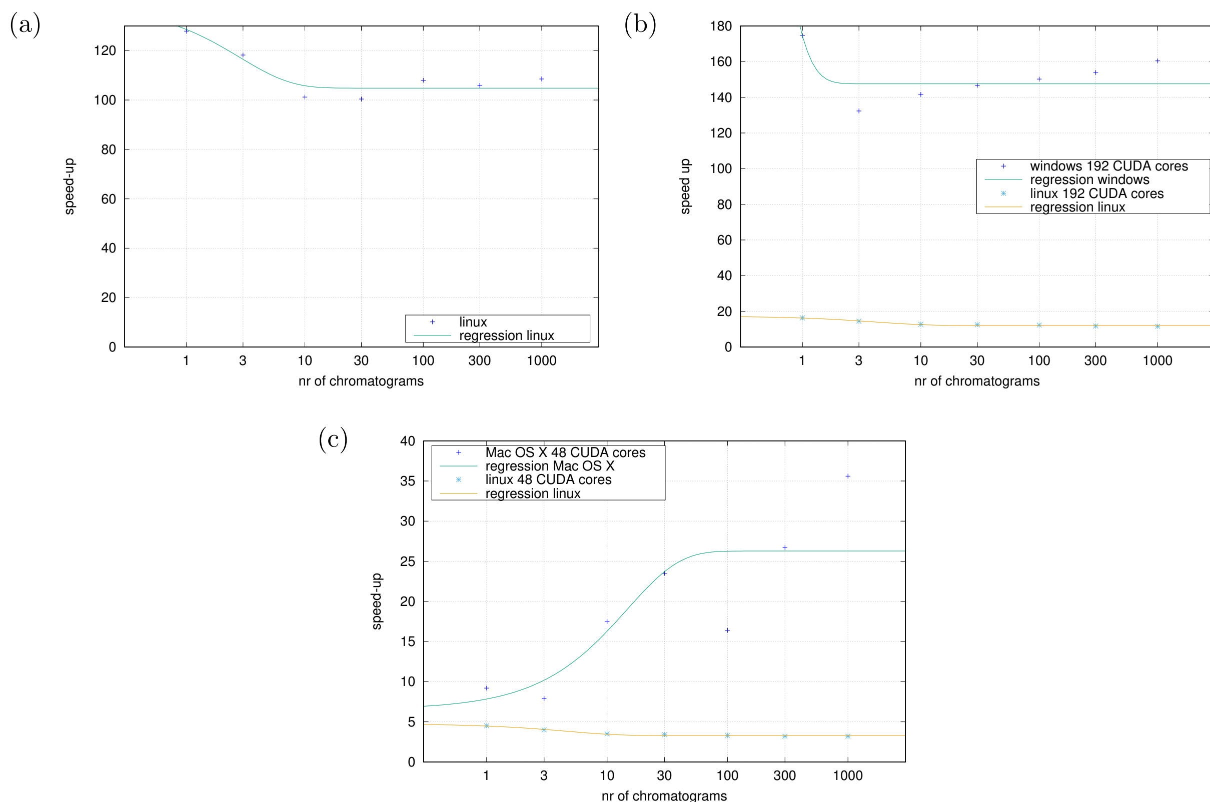

Figure 6: Speed-up during varying the number of single chromatograms on the operating systems Linux (a), Windows (b) and Mac OS X (c). The yellow line shows a run with the Tesla on Linux with restricting the GPU to just call as many threads as there are cores on the compared GPU. The number of threads are manually restricted to have a hardware independent comparison with identical conditions.

Table 6: Speed-up when invoking the Savitzky-Golay filter with a different number of chromatograms and with different GPUs and operating systems. The chromatogram length has been 2688, 8 repetitions, a blocksize of 224 and a single thread.

Number of Chromatograms		**Linux** / Tesla K20Xm			**Windows** / Quadro K610M			**Mac OS X** / GeForce GT 130		
		Java	*CUDA*	*speed-up*	*Java*	*CUDA*	*speed-up*	*Java*	*CUDA*	*speed-up*
1	$mean, ms$	389.9	3.3	**120.0**	855.1	4.9	**174.5**	331.2	36.0	**9.2**
	SD, ms	5.5	0.4		36.7	1.8		28.9	1.1	
3	$mean, ms$	1036.5	9.7	**107.4**	2435.2	18.4	**141.6**	863.1	109.9	**7.9**
	SD, ms	8.8	0.7		30.0	8.3		20.5	1.2	
10	$mean, ms$	2967.7	29.2	**101.8**	8084.6	57.1	**141.6**	2780.0	158.8	**17.5**
	SD, ms	74.8	1.6		81.6	9.5		27.5	96.5	
30	$mean, ms$	8732.0	86.8	**100.6**	23759.0	162.0	**146.7**	8246.5	350.5	**23.5**
	SD, ms	200.7	2.7		458.3	10.9		25.9	264.0	
100	$mean, ms$	28390.8	254.9	**111.4**	74617.5	496.7	**150.2**	27650.4	1687.7	**16.4**
	SD, ms	325.4	4.2		13791.1	6.0		518.5	218.8	
300	$mean, ms$	82372.4	767.7	**107.3**	225194.7	1463.5	**153.9**	81977.2	3069.3	**26.7**
	SD, ms	553.5	10.9		47509.3	19.4		202.4	533.6	
1000	$mean, ms$	270689.2	2491.9	**108.6**	7772769.9	4816.7	**160.4**	274093.9	7709.5	**35.6**
	SD, ms	1299.2	21.8		7434.6	79.4		3481.1	344.5	

Table 7: Calculation done with the Tesla with restricting the GPU to just call as many threads as there are cores on the compared GPU. The restriction has been the number of CUDA cores on Windows (Quadro, 192 cores) and on Mac (GeForce, 48 cores). The threads have been manually restricted to have a hardware independent comparison with identical conditions. Mean gives the time spent for the calculation with its standard deviation (SD). Speed-up between the Java and CUDA version is listed. The ratio to the speed-up of the comparing GPU is listed as well.

Number of Chromatograms		*Number of Threads* 48	192	2688
1	*mean, ms*	85.7	24.0	3.3
	SD, ms	0.7	0.4	0.4
	speed-up	**4.5**	**16.3**	**120.0**
	ratio	2.4	4.9	1.0
3	*mean, ms*	256.3	71.4	9.7
	SD, ms	1.1	0.6	0.7
	speed-up	**4.0**	**14.5**	**107.4**
	ratio	2.3	3.9	1.0
10	*mean, ms*	852.7	234.1	29.2
	SD, ms	3.7	1.7	1.6
	speed-up	**3.5**	**12.7**	**101.8**
	ratio	5.4	4.1	1.0
30	*mean, ms*	2554.1	700.6	86.8
	SD, ms	8.3	5.4	2.7
	speed-up	**3.4**	**12.5**	**100.6**
	ratio	7.3	4.3	1.0
100	*mean, ms*	8507.1	2305.9	254.9
	SD, ms	48.2	15.7	4.2
	speed-up	**3.3**	**12.3**	**111.4**
	ratio	5.0	4.6	1.0
300	*mean, ms*	25511.9	6959.2	767.7
	SD, ms	116.3	46.7	10.9
	speed-up	**3.2**	**11.8**	**107.3**
	ratio	8.3	4.8	1.0
1000	*mean, ms*	85101.3	23296.5	2491.9
	SD, ms	398.0	101.2	21.8
	speed-up	**3.2**	**11.6**	**108.6**
	ratio	11.0	4.8	1.0

Multiple CPU Threads

GPU specific tasks, like memory copying between the devices and kernel invocations, are executed sequentially when using multiple CPU threads [22] (Table 8). With a single thread the graphics card specific tasks can be processed as soon as they needed to. When using several CPU threads they start to block each other. So it happens that a thread is occupying the GPU while another one also wants to access it. Therefore the new thread has to wait, until the other thread has finished using the GPU. Each CPU thread wanting to run kernels on the GPU has its own CUDA context. Different contexts cannot run simultaneously on the GPU, therefore one kernel is executed after the other.

Table 8: The GPU functions can just be executed sequentially. When using multiple CPU threads they start to block each other. H2D is the copy process from host to device, K is the kernel and D2H is the copying back the data from device to host.

Thread 1	CPU	H2D	K	D2H	CPU			
Thread 2	CPU				H2D	K	D2H	CPU

To run CUDA operations simultaneous streams can be used [22]. Asynchronous CUDA commands can be called which lead to an more efficient use of the graphics card. In the current implementation one stream is used which leads to the above described behavior. When the CUDA commands are called from the same context multiple streams can be used (Table 9). Depending on the compute capability the number of concurrent kernel executions can range from 4 up to 128. GPUs using the fermi [36] and more recent architectures are able to do this. These are the compute capabilities 2.0 to the most recent (Table 14 in [22]).

Table 9: Multiple streams can execute simultaneously several CUDA operations to get even a higher parallelism. H2D is the copy process from host to device, K1 and K2 refer to the called kernels and D2H is the copying back the data from the device to the host.

Stream 1	H2D	K1	K2	D2H	
Stream 2		H2D	K1	K2	D2H

3.4 Speed of Full Chromatogram Files

In the final implementation not just single chromatograms are calculated but a whole chromatogram file in which lipids were to be identified. Since all the steps to the finished implementation are done and it is known how the implementation reacts to various factors it is now interesting to know how the speed-up is during a whole run. Files were used like in an every day work environment.

With the full integration of the GPU version into LDA 12 chrom files were analyzed. Six measured with an OrbiTrap and six with a QTrap from a previously performed wet-lab experiment were used for the evaluation. The speed-up of the different types of files ranged from around 7 for the Orbitrap files to 30-40 for the QTrap files (Table 11). The distribution of the chromatogram lengths have been different for Orbitrap and QTrap (Table 10). In general Orbitrap has more short chromatograms than long chromatograms whereas the distribution for QTrap is the other way round.

Table 10: Distribution of smoothed chromatograms according to their length by measuring full chrom files. The calculation was done with a OrbiTrap negative, OrbiTrap positive, QTrap negative and QTrap positive file.

File	**Smoothing Length**	**Smoothed Chromatograms**	**Amount of Smoothed Chromatograms**
OrbiTrap negative	200	103,656	91.1 %
	1279	10,093	8.9 %
OrbiTrap positive	200	335,066	95.3 %
	1469	16,431	4.7 %
QTrap negative	137	50,978	38.2 %
	475	4,964	3.7 %
	1200	77,544	58.1 %
QTrap positive	193	49,676	39.5 %
	1200	76,104	60.5 %

Table 11: LDA was fed with 12 chrom files of 4 categories and run with the Java and the CUDA implementation. The tests were performed on Windows with the Quadro K610M.

File	**Java** (min:sec)	**CUDA** (min:sec)	**Speed-up**
Orbitrap negative / 113,749 chromatograms in 002[..]			
002_liver2-1_Orbitrap_CID_neg.chrom	30:18	4:11	**7.24**
003_liver2-1_Orbitrap_CID_neg.chrom	30:12	4:05	**7.34**
004_liver2-1_Orbitrap_CID_neg.chrom	29:57	4:07	**7.28**
Orbitrap positive / 351,497 chromatograms in 002[..]			
002_liver2-1_Orbitrap_CID_pos.chrom	72:04	10:41	**6.75**
003_liver2-1_Orbitrap_CID_pos.chrom	72:32	11:01	**6.58**
004_liver2-1_Orbitrap_CID_pos.chrom	73:02	11:17	**6.47**
QTrap negative / 133,486 chromatograms in [..]021[..]			
Data20151002_QTrap_Liver-021_QTrap_Liver1-1_neg.chrom	458:15	10:39	**43.03**
Data20151002_QTrap_Liver-022_QTrap_Liver1-1_neg.chrom	446:04	10:17	**43.38**
Data20151002_QTrap_Liver-023_QTrap_Liver1-1_neg.chrom	432:25	10:06	**42.81**
QTrap positive / 125,780 chromatograms in [..]002[..]			
Data20151002_QTrap_Liver-002_QTrap_Liver1-1_pos.chrom	419:04	13:13	**31.71**
Data20151002_QTrap_Liver-003_QTrap_Liver1-1_pos.chrom	434:55	13:40	**31.82**
Data20151002_QTrap_Liver-004_QTrap_Liver1-1_pos.chrom	449:32	13:45	**32.69**

3.5 Deviations

The smoothed chromatograms of the GPU implementation are numerically not identical to ones from the Java implementation. In median the deviation was 10^{-6} where the drift went up at the borders of a full chromatogram smooth. The first and the last values of a completely smoothed chromatogram could in rare cases differ by up to 10%. The largest differences were observed when a zero-crossing happened at the first or last element of the smoothing array. For example, when after a smoothing with the Java implementation the value of the first element is greater than zero and the one with the CUDA implementation has at that position a zero. The values following are greater than zero. When after the next smoothing repetition the values at the beginning are all greater than zero it is very likely that the deviation is high.

From every lipid the area and the result of isotope 0, 1 and sometimes 2 was calculated. In total this leads to different peak areas of the found lipids. From 8084 found lipids 28 differ more than 0.1% at the area under the curve. All lipids in the negative mode OrbiTrap samples were quantified correctly, with a slightly deviation of the calculation (Table 12).

Table 12: The numeric results of the Java implementation and the CUDA implementation have been compared. From the four different groups the number of input lipids, the number of peak area deviations and the number of lipids in which this occurred compared to the total amount of found lipids is listed. The maximum and average deviation is listed as well. The median has been 0 for all areas.

Group File	**Peak Area Deviation/** **in # Lipids**	**Total** **Lipids**	**Maximum** **Deviation**	**Average** **Deviation**
Orbitrap negative (2280 input lipids)				
002_liver2-1_Orbitrap_CID_neg.chrom	0/0	520	$3.05E-5$	$1.44E-7$
003_liver2-1_Orbitrap_CID_neg.chrom	0/0	480	$5.32E-5$	$1.78E-7$
004_liver2-1_Orbitrap_CID_neg.chrom	0/0	547	$9.63E-5$	$2.65E-7$
Orbitrap positive (2947 input lipids)				
002_liver2-1_Orbitrap_CID_pos.chrom	5/2	1201	$1.81E-1$	$4.30E-5$
003_liver2-1_Orbitrap_CID_pos.chrom	0/0	1187	$1.04E-5$	$1.16E-7$
004_liver2-1_Orbitrap_CID_pos.chrom	1/1	1062	$7.91E-3$	$1.81E-6$
QTrap negative (2280 input lipids)				
Data20151002_QTrap_Liver-021_QTrap_Liver1-1_neg.chrom	9/5	143	$1.50E-1$	$6.80E-4$
Data20151002_QTrap_Liver-022_QTrap_Liver1-1_neg.chrom	0/0	309	$5.91E-6$	$1.64E-7$
Data20151002_QTrap_Liver-023_QTrap_Liver1-1_neg.chrom	2/1	280	$4.79E-3$	$6.51E-6$
QTrap positive (2570 input lipids)				
Data20151002_QTrap_Liver-002_QTrap_Liver1-1_pos.chrom	17/7	672	$5.26E-1$	$4.90E-4$
Data20151002_QTrap_Liver-003_QTrap_Liver1-1_pos.chrom	20/7	1155	$1.56E-0$	$2.45E-3$
Data20151002_QTrap_Liver-004_QTrap_Liver1-1_pos.chrom	14/5	528	$1.53E-0$	$1.79E-3$

4 Discussion

By moving the Savitzky-Golay filter to the GPU, LDA could achieve a considerable speed-up. Dependent on the testing input files the acceleration reached from 6 to 43. Since several parameters change from one run to the next, the impact has been determined. Numerical deviations occurred from the CUDA version compared to the Java version.

The current implementation of LDA is written in Java to be available for the three most used desktop operating systems Windows, Mac OS X and Linux. CUDA can be accessed from C, C++, Fortran and Python. At the time of writing CUDA is officially not supporting Java. A Java native interface is needed to access the library and the CUDA functions to smooth the chromatogram. To invoke the GPU and being able to use CUDA a native library was implemented that can be called from Java.

4.1 GPU Accelerated Bioinformatics Algorithms

A recent literature review reports that the speed-up achieved with a GPU implementation ranges from 0.3 to 20,000 and has a median of 23 (Figure 7) [37]. The acceleration depends on the on hand on the complexity of the algorithm. On the other hand it depends on the amount of data which has to be processed and transferred between host and device memories. Moving an algorithm to a GPU can even result in a slow-down.

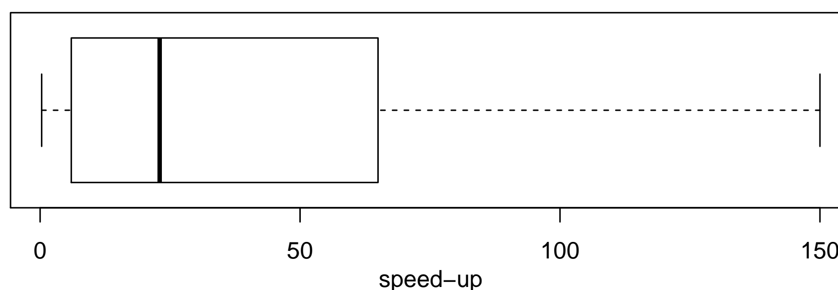

Figure 7: Speed-up of bioinformatics algorithms implemented on the GPU compared to the CPU implementation. Figure taken from [37].

4.2 Filter Migration to the GPU

Two options have been considered for migrating the current implementation of the Savitzky-Golay filter to the graphics card. One was moving the whole chromatogram into the GPU storage and let each thread calculate a single lipid. Another option was moving just the

calculation of a single chromatogram to the GPU.

Moving the whole chromatogram into the GPU storage would have been too complex since multiple classes would have been moved to the GPU. Although it is possible to construct classes on the GPU another problem could have been the size of the memory. Older graphics cards can run out of memory faster with bigger chrom files. Another problem is the divergence again. The whole code has many parts where the calculation path can differ heavily from one lipid to the next. This is a situation that should be strongly avoided on GPUs.

Migration just the smoothing of a single chromatogram to the GPU focuses on the bottleneck of the analysis. The little changes to the existing code made it also less prone to produce implementation errors from porting the code to the graphics card. With the less code moved to the GPU the path divergence of threads within a warp can also be decreased since less branching occurs.

4.3 Hardware and Software Preparation

For Linux the programming was done remotely because no Linux desktop computer with a CUDA capable device was present. The same CUDA version needs to be installed on the local and remote computer as well as the same type of operating system needed to be installed. Since the server was running with Debian Linux the local system needed to be Unix based as well. Because Ubuntu 16.04 LTS had troubles during several installation attempts with the AMD graphics driver, the Arch Linux operating system was installed. The Arch User Repository (AUR) provided the required CUDA version 7.0. Nsight and nvcc was part of the installation of the toolkit and after configuring git and Nsight the remote programming was ready.

For the installation of CUDA on Windows a compiler and the IDE Microsoft Visual Studio needed to be installed. The very important step in this process is to check if the versions of Windows, Microsoft Visual Studio and CUDA are compatible and which CUDA version is able to run on the desired GPU. It is also important to follow the installation order by installing first Microsoft Visual Studio and second the CUDA toolkit.

On the used Mac OS X system the current version 8.0 of CUDA was not compatible with the graphics card. So CUDA 6.5 was downloaded from the CUDA archive [29].

This archive provides every released CUDA version, so it is even possible to develop and compile CUDA code for older NVIDIA graphics cards. For Mac OS X no major pre installation steps are required.

4.4 Test Driver

Porting Java Code to C Code

Java version of the filter code was reimplemented in C because CUDA uses the C syntax rules. Since the programming languages are not that different the translation was straightforward. In addition to the syntax, the attributes used in Java, had to be changed to function parameters for functions that needed these values.

Porting from CPU to GPU

In general porting the functions from the C host code to the GPU with CUDA was easier than expected. However, just plain porting without considering the special needs of a GPU can result in a slow down. The spatial TAU-leaping in crowded compartments simulator of Pasquale et al. [38] had a similar problem. In their implementation the speed-up reached from 0.3-24. For the Savitzky-Golay library the slow-down was caused by the dynamic allocation of arrays on the local memory during the runtime of the kernel. The size of the arrays were dependent on the order of the polynomial which got to be fit into the raw data points. These arrays are used to solve the normal equation. Even tough the arrays are very small they had a comparative big impact. In total maximal 34 floats and 5 ints were dynamically allocated per thread. After identifying that the allocation was the problem it was moved from the device to the host. Like the raw data and the retention time these arrays are now allocated on the global memory of the device by the host. This led to a significant speed-up. But there was still a better solution. Since the arrays need nearly no memory space the maximum array size got hard coded even when the whole arrays are not used at every calculation. With this solution the fastest memory on the GPU, the local memory, is used.

One has to pay attention to the different memories of the GPU and how to use them. To take full advantage of the graphics card the way of operation has to be kept in mind. Dynamic allocation on the local memory in the kernel is not the most efficient way of using the graphics card. Every thread has to allocate the memory dynamically during every calculation. A better solution is to allocate the memory dynamically from the host on the devices global memory. Doing this the memory is allocated once for the whole calculation.

Otherwise the allocation has to be performed as often as points are smoothed. If the array size does not exceed the size of the local memory, then the array can be created with a fixed size, even when only parts of the whole array is used. The advantage of the local memory is that it is the fastest memory on the GPU.

It has been noticed that the first call of a CUDA function takes a lot longer than the subsequent calls. The reason for this behavior is that the CUDA runtime initialization is done when the first time a runtime function is called. All functions calls afterwards need no further initialization. During the first call a CUDA context is created and if necessary the device code is just-in-time compiled and loaded into device memory. When comparing the run time of the CPU and GPU implementation the initialization time was not measured.

Using the CUDA function *clock()* for the time measurement in the CUDA library led to a run time of 0 clock cycles which is very unlikely. The reason for that is that the compiler and assembler do perform instruction re-ordering. So it happened that the start and stop calls ended up next to each other resulting in 0 clock cycles.

By investigating the results of the different calculation steps the design matrix seemed to make the same calculation several times. After rewriting the code the calculation is done once and the rest of the matrix gets filled with the calculated values.

Some if statements led to calculating the same calculation multiple times. This was avoided by making the if statement once and replacing the original once.

Changing the function $pow(x, 1/4)$ to $rsqrt(rsqrt(x))$ is suggested in the *CUDA C Best Practice Guide* [39] since the performance is higher as well as the accuracy.

Java Native Interface

A Java Native Interface JNI was needed to access the GPU from Java, since Java is not officially supported by CUDA. The created function names from *javah* have to be used in the library so that Java can communicate with the library. To work with the parameters and arrays which are passed to the library they have to be converted to the corresponding C data types. Here it is important to know, that some functions which convert these arrays need their own free functions, otherwise a memory leak will arise. In this certain case every $GetObjectArrayElement()$ needs a $DeleteLocalRef()$ and every $GetFloatArrayElements()$ needs a $ReleaseFloatArrayElements()$. The rest can be the

same as it would be a standard C-program.

Multithreading on the CPU

Using multiple threads on the CPU was already an enhancement of the old implementation. Now every thread on the CPU is moving the calculation of the Savitzky-Golay filter to the library and furthermore to the GPU. Since multiple threads are accessing the same library at the same time the result of the smoothed chromatograms was non predictable. The simultaneous library access from the CPU threads was overwriting each others memory on the GPU. To avoid this, every CPU thread allocates its own memory on the global GPU storage. The address of the global memory gets saved in the Savitzky-Golay JNI Java class.

Whole Chrom File Processing

The previous test driver just used a set of a few chromatograms, which could be extend and shortened to test the different accelerations of different chromatogram lengths. It was also possible to change the number of chromatograms itself which should be smoothed. To apply the GPU implementation in a more realistic test case, a testing environment was created that loads a chrom file, handles the preprocessing, smooths multiple chromatograms and detects the peaks like in the LDA. But in the real environment the chromatograms have to be extracted from a chrom file. For each peak several chromatograms have to be extracted in m/z-direction and retention time direction.

LDA Integration

The last step was integrating the dynamic library into the LDA. With the previous test drivers it was no big effort to do this task. The difference was that classes from MASPECTRAS had to be extended and to check at the beginning of LDA if a CUDA capable device is present.

4.5 Speed-Up with Different Settings

Certain parameters change the speed of the calculation significantly and others have nearly no effect. With every new chromatogram file, the program is fed with, the parameters may change and lead to different processing times. The combination of the varying parameters also effect the speed-up of the new implementation. Korpar et al. [40] implemented a sequence alignment algorithm with CUDA and observed a similar behavior. Depending on the sequence sizes the calculation reached a speed-up between 440-960.

4.5.1 Chromatogram Length

One of these changing parameters is the chromatogram length. The length does not only change from chrom file to chrom file but also within a single run of a file. Not only the full length of chromatograms in time direction are smoothed but also shorter ones in time direction and profiles in m/z direction, to detect a peak and calculate the final peak area (Pages 7ff.). With longer chromatograms the speed-up is increasing nearly linear (Figure 5). This can be explained with the fact that with increasing length of the chromatogram the time that is spent on the CPU is proportionally getting less. The pure calculation time is higher which is the reason of the performance enhancement. Also for a higher amount of smoothed data points for each chromatogram the interactions between the CPU and GPU are getting proportionally less. For example storing data on the GPU, calling a kernel and copying the data back from the GPU.

4.5.2 Number of Repetitions

Another parameter that changes with nearly every call of the Savitzky-Golay function is the number of repetitions. The raw data is not smoothed just once but multiple times. Again it has a similar effect as before (Figure 5). With an increasing number of repetitions the speed-up is increasing nearly linear as well. The reasons are also similar. In relation to a single repetition a higher number of repetitions increases relatively the time on the GPU. Since the GPU performs a smoothing calculation more efficiently, the relatively longer time on the GPU leads to a higher speed-up. Another reason is that less data is copied between host and device. With a single repetition the raw data has to be copied from the host to the device. After one smoothing round the smoothed data has to be copied back to the host. With higher smoothing repetitions proportionally less time is spent with copying. Once the data is uploaded to the GPU the graphics device can perform as many smoothing repetitions as wanted. Only after all the calculations have been done the smoothed data has to be copied back.

4.5.3 Block Size

The block size is a hardware specific parameter, that does not change from one calculation to the next. It just varies between different computer setups. One way to choose a good block size is to aim for high GPU occupancy. Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of warps that can be active on the multiprocessor at once. The function *cudaOccupancyMaxPotentialBlockSize()* just

needs to know the kernel that is going to be launched and it can calculate the block size that fits the function best regarding the occupancy. Nevertheless a high occupancy does not guarantee the best performance. Even though the block size is fixed for a certain computer in this test the block size was changed to see which effects do occur. The speed-up does not change significantly when using different block sizes (Figure 5). During the tests the block sizes have been chosen to be a multiple of 32 threads. The reason for this is that the GPU manages the calculation in warps which are groups of 32 threads.

4.5.4 Multiple CPU Threads

Most state of the art desktop computers are capable of CPU multi threading. LDA is making use of this to already enhance the speed. Therefore it was also of interest how the implementation reacts to a different amount of CPU threads. The speed-up is going down when increasing the number of CPU threads (Figure 5). After a certain amount of threads the speed-up stays on a steady level. The decreasing speed-up does not mean, that a higher number of CPU threads leads to a longer processing time. With more CPU threads the whole calculation is still done in a shorter time.

4.5.5 Number of Chromatograms

A parameter that changes very likely at every run is the number of chromatograms which need to be smoothed. It always depends on how many lipids are examined in an analysis run. The speed-up on three different computer setups was tested (Figure 6, Table 6). The speed-up depends strongly on the system the calculation is performed on. It is not just the hardware components but also the software which is running on the operating system. After a certain number of chromatograms the speed-up stayed nearly on a same level. On the Mac OS X computer the speed-up is increasing whereas on the other two systems the speed-up is decreasing.

Regarding the hardware it can be said that the performance of the CPU and GPU of the particular systems have to be compared. When taking a look at the hardware specifications (Table 3) it can be guessed that the performance of the CPU to the GPU is relatively higher on the Mac than on the other computers. CPUs have a comparable clock speed between 2.6 to 3 GHz but the GPU of the Mac has very few CUDA cores and a slow clock rate. For the comparably weak GPU, the least speed-up is observed. But even though the graphics card is underpowered it provides a improvement which accelerates

the implementation by roughly a factor of 25 for a single chromatogram. When comparing the MS Windows based computer with the Linux computer the initial guess would be that Linux could perform a higher speed-up due its more powerful graphics card. But the reality looks different. In the tests windows could reach a higher speed-up. The reason for that is that different Java HotSpot VM (virtual machine) are installed on these two systems. There are two options for the HotSpot VM which are the client VM and the server VM [41]. Both use the same code base of the HotSpot runtime environment but use different compilers. The client one is suited for desktop computers. It is designed to optimize the start-up of the program to make it fast, has less performance when used on very long calculations. The server implementation whereas is optimized the other way around. It does not matter if the start-up is taking a little bit longer as long as the long time calculation is faster. In the tests windows is running the Java HotSpot Client VM whereas Linux is running the Java HotSpot Server VM.

The second circumstance that attracts attention is that the speed-up varies with changing the number of chromatograms. This could be explained because micro benchmarks have been performed which should not be done [42]. Java HotSpot starts by interpreting and running the program. During runtime it detects hot spots which are pieces of code that are looped over a lot. This code is then compiled. Either the compiled code is replaced after calling the method/loop another time or during the running loop which is called "on stack replacement". Therefore the program delivers a different speed-up for a different number of chromatograms. For one chromatogram no hot spot is identified but with higher repetitions of calculations the interpretation can be replaced with a compiled version.

4.5.6 Hardware Independent Comparison

To deal with the previous discussed differences of the hardware and software and just see the differences with a different amount of CUDA cores, the number of threads have been restricted (Figure 6, Table 7). As many threads with a kernel as CUDA cores available on the graphics card were launched. For the GPU of the Mac the kernel was launched with 48 threads since the related GPU has 48 CUDA cores. For the GPU of the windows computer the kernel was launched with 192 threads. The simulation of the graphics cards shows little acceleration. In the actual implementation for every smoothing point a thread is started. This is possible even when there are less CUDA cores than threads. The threads are scheduled by CUDA and it takes care that all calculations are done as fast

as possible. By restricting the number of threads a single thread has to smooth multiple points. Since the scheduling was performed manually and not by CUDA the speed-up was considerable worse.

4.6 Speed-Up with whole Chromatogram Files

The speed-up was measured for various chrom files measuerd on OrbiTrap and QTrap platforms in positive and negative ion mode. The speed-up of the OrbiTrap files was in general lower than the speed-up of the QTrap files (Table 11). Not just the times were measured but also the length of the smoothed chromatograms (Table 10). Since the length of the chromatograms influence the speed-up of a single chromatogram (Figure 5) the different speed-ups of the whole file can be explained. The amount of short chromatograms (around 200 data points) compared to longer chromatograms (around 1300 data points) are on OrbiTrap way much higher (around 90%) than on the QTrap (around 40%). As seen as in the comparison between different chromatogram lengths, short chromatograms have a lower acceleration whereas longer chromatograms lead to a higher speed-up. Since the amount of short chromatograms is relatively higher in the OrbiTrap files the calculation have a speed-up that is less than the one from QTrap. The longer chromatograms originate from smoothing of the whole chromatogram to find initial peaks. The shorter smooth lengths come from the calculation of the peak area.

4.7 Deviation of the Numeric Result of the Chromatogram

Numerical deviation between a CUDA and the CPU implementation have also been observed by Oliveira et al. [43]. They made computer simulations of cardiac electrophysiology, computed the error between the implementations and observed that CUDA derived less than a factor of $5 \cdot 10^{-5}$ compared to the CPU.

4.7.1 Single Chromatogram

The reason for deviations is that the operands and functions from CUDA have an equal or lower accuracy than the IEEE-compliant calculation. In the CUDA Programming Guide appendix D [22] the error bounds for all mathematical functions are listed. The "+"-operand was just used with integers for indexing and has no influence in the result. Few "-" were used for indexing and actual calculation of values where the error is a maximum

of 0.5 ULP (Unit in the last place). "*" is used for indexing as well but has already more invocations for multiplying values than the previous operands and has the same maximum error of 0.5 ULP as "+". Approximately as often as "*" is called "/" is called as well with the difference that the error bounds go up to a maximum of 2 ULP. The function $rsqrt()$ is called twice for each smoothing point which is very little. Since it has a maximum error of 1 ULP it is of nearly no consequence compared to the next function. The critical calculation that is responsible for the different calculation results is the calculation of the power which is used in the solving of the normal equation. The function used is $powf()$ and has a maximum of 8 ULP. The calculation is also done very often. 100 times for each smoothing point is not unusual. The calculation of $powf(x, 1/4)$ was already omitted by using $rsqrt(rsqrt(x))$ instead. There are workarounds to get more precise results which have the drawback that they lead to a performance drop.

Points with a higher deviation are usually the points at the beginning and end of a fully smoothed chromatogram. Compared to the rest of the chromatogram the relative position of the window, for choosing the raw parameters, changes to the point which should be smoothed. In the middle of the chromatogram the window is symmetrically around the smoothing point. Towards the end this structure change. Since the window has no points left it just keeps the same position for the last calculations. Just the smoothing point is going forward. Respectively the window is not symmetrical around the smoothing point anymore. At the very end the window is just at one side, but still has the same width. One step of establishing the normal equation is calculating distances between each point in the window and the smoothing point. These distances get multiplied with each other. Since the window is not symmetrical anymore around the smoothing point but moved to just one side the distances are roughly doubled which leads to the multiplication in way higher numbers. These large numbers get also fed to the $powf()$ function with a high error-proneness. This interaction of high numbers and using them in an inexact function lead to a higher deviation at the beginning and end of the smoothed chromatogram.

4.7.2 Full Chromatogram

The errors of the previous section can accumulate and lead to an error of the detection of the lipids when processing the whole chromatogram (Table 12). 8,084 lipids have been found where from 28 lipids at least one value differs more than 0.1%. In total these are 68 values. There was no correlation whether the deviations occur in high or low peaks.

4.8 Conclusion

With the very fast growing amount of data that is generated for lipidomics it is necessary to find a way to keep up and process the data faster. The data processing is necessary to evaluate the measurements and get out more information. The CPU was the main processing unit when it comes to calculation of data. Since the CPU speed-up is physically coming to its limits other options have to be considered.

The GPU provides a high throughput of data which is very convenient for processing a huge amount of data. The restriction that the same command is applied to multiple data points is in this kind of data processing not a big drawback since this is happening in many calculations. But the way how a GPU works have to be kept in mind.

CUDA enables a way that code can simply be ported to the GPU. It will run on the dedicated device with minor code changes. But achieving the best performance is not straight forward. At first the position of the storage has to be decided for every used variable. Some are obvious to find a place to store them, but others may need some different approaches and tests to find the best fitting position.

An additional way to accelerate the code even more is using streams. Streams enable the GPU to perform several CUDA commands at the same time. Therefore new data can be uploaded to the GPU when an old kernel is still running. It is even possible to run several kernels at the same time. But this is just possible when a single CPU thread is managing the streams. In the LDA the calculations of the single CPU threads had to be rewritten to calculate several masses of interest at the same time to sent multiple calculations to the GPU.

LDA with the new implemented library has a significant speed-up depending on the input. For specific data sets, it was possible to reduce calculation time from over 7 hours to just 10 minutes. Even tough there are minor numerical deviations of the results the implementation can be seen as an improvement of the software.

References

[1] Margulies M, Egholm M *et al.*: **Genome Sequencing in Open Microfabricated High Density Picoliter Reactors.** *Nature* 2005. **437**(7057): 376–80.

[2] Illumina. http://www.illumina.com/, Date accessed: 2016-11-11.

[3] Nussinov R, Pieczenik G *et al.*: **Algorithms for Loop Matchings**. *SIAM J Appl Math* 1978. **35**(1): 68–82.

[4] Zuker M and Stiegler P: **Optimal Computer Folding of Large RNA Sequences Using Thermodynamics and Auxiliary Information.** *Nucleic Acids Res* 1981. **9**(1): 133–148.

[5] Varré JS, Schmidt B *et al.*: **Manycore High-Performance Computing in Bioinformatics**. In *Advances in Genomic Sequence Analysis and Pattern Discovery* Edited by Elnitski L, Piontkivska H *et al.*, Singapore. World Scientific, 2011 , 1–18.

[6] Arenas MG, Mora AM *et al.*: **GPU Computation in Bioinspired Algorithms: A Review**. In *Advances in Computational Intelligence* Edited by Cabestany J, Rojas I *et al.*, Berlin, Heidelberg. Springer, 2011. 433–440.

[7] Fang J, Varbanescu AL *et al.*: **A Comprehensive Performance Comparison of CUDA and OpenCL**. In *International Conference on Parallel Processing (ICPP)* . IEEE, 2011: 216–225.

[8] Gelsinger P: **Microprocessors for the New Millennium: Challenges, Opportunities, and New Frontiers**. In *International Solid-State Circuits Conference (ISSCC)* , San Francisco, CA, USA. IEEE, 2001: 22–25.

[9] Moore GE: **Cramming More Components onto Integrated Circuits**. *Electronics* 1965. 114–117.

[10] Flynn MJ: **Some Computer Organizations and Their Effectiveness**. *IEEE Transactions on Computers* 1972. **C-21**(9): 948–960.

[11] Owens JD, Luebke D *et al.*: **A Survey of General-Purpose Computation on Graphics Hardware**. *Comput Graph Forum* 2007. **26**(1): 80–113.

[12] CUDA Zone. https://developer.nvidia.com/cuda-zone, Date accessed: 2016-10-18.

[13] OpenCL. https://www.khronos.org/opencl/, Date accessed: 2016-10-18.

[14] OpenACC Home. http://www.openacc.org/, Date accessed: 2016-10-18.

[15] Thrust. https://thrust.github.io/, Date accessed: 2016-10-18.

[16] C++ AMP. https://msdn.microsoft.com/en-us/library/hh265137.aspx, Date accessed: 2016-10-18.

[17] DirectCompute. https://developer.nvidia.com/directcompute, Date accessed: 2016-10-18.

[18] Jacket. http://www.omatrix.com/jacket.html, Date accessed: 2016-10-18.

[19] Dolbeau R, Bihan S *et al.*: **HMPP(TM): A Hybrid Multi-core Parallel Programming Environment**. In *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU)* , Washington, DC, USA. ACM, 2007: 1–5.

[20] WebCL. https://www.khronos.org/webcl/, Date accessed: 2016-10-18.

[21] Payne JL, Sinnott-Armstrong NA *et al.*: **Exploiting Graphics Processing Units for Computational Biology and Bioinformatics**. *Interdiscip Sci* 2010. **2**(3): 213–220.

[22] CUDA C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, Date accessed: 2017-08-16.

[23] Haimi P, Uphoff A *et al.*: **Software Tools for Analysis of Mass Spectrometric Lipidome Data**. *Anal Chem* 2006. **78**(24): 8324–8331.

[24] Haimi P, Chaithanya K *et al.*: **Instrument-Independent Software Tools for the Analysis of MS–MS and LC–MS Lipidomics Data**. *Methods Mol Biol* 2009. **580**: 285–294.

[25] Katajamaa M, Miettinen J *et al.*: **MZmine: Toolbox for Processing and Visualization of Mass Spectrometry Based Molecular Profile Data**. *Bioinformatics* 2006. **22**(5): 634–636.

[26] Pluskal T, Castillo S *et al.*: **MZmine 2: Modular Framework for Processing, Visualizing, and Analyzing Mass Spectrometry-Based Molecular Profile Data**. *BMC Bioinformatics* 2010. **11**(1): 395.

[27] Hartler J, Trötzmüller M *et al.*: **Lipid Data Analyzer: Unattended Identification and Quantitation of Lipids in LC-MS Data**. *Bioinformatics* 2011. **27**(4): 572–577.

[28] Savitzky A and Golay MJE: **Smoothing and Differentiation of Data by Simplified Least Squares Procedures**. *Analytical Chemistry* 1964. **36**(8): 1627–1639.

[29] CUDA Toolkit Archive. https://developer.nvidia.com/cuda-toolkit-archive, Date accessed: 2017-10-24.

[30] Nsight Eclipse Edition. http://docs.nvidia.com/cuda/nsight-eclipse-edition-getting-started-guide/index.html, Date accessed: 2017-10-27.

[31] Git. https://git-scm.com/, Date accessed: 2017-10-27.

[32] Java Native Interface Specification Contents. https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html, Date accessed: 2017-10-30.

[33] javah. https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javah.html, Date accessed: 2017-10-31.

[34] Li Xj, Zhang H *et al.*: **Automated Statistical Analysis of Protein Abundance Ratios from Data Generated by Stable-Isotope Dilution and Tandem Mass Spectrometry**. *Analytical Chemistry* 2003. **75**(23): 6648–6657.

[35] Hartler J, Thallinger GG *et al.*: **MASPECTRAS: A Platform for Management and Analysis of Proteomics LC-MS/MS Data**. *BMC Bioinformatics* 2007. **8**: 197.

[36] NVIDIA's Next Generation CUDA Compute Architecture: Fermi. https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, Date accessed: 2017-11-28.

[37] Brandstätter S: **Bioinformatics Algorithms and GPUs**. *Preject Report, Institute of Computational Biotechnology, Graz University of Technology* 2016.

[38] Pasquale G, Maj C *et al.*: **A CUDA Implementation of the Spatial TAU-Leaping in Crowded Compartments (STAUCC) Simulator**. In *International Conference on Parallel, Distributed and Network-Based Processing (PDP)* , Torino, Italy. IEEE, 2014: 609–616.

[39] CUDA C Best Practices Guide. http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html, Date accessed: 2017-11-28.

[40] Korpar M and Sikic M: **SW#-GPU-Enabled Exact Alignments on Genome Scale**. *Bioinformatics* 2013. **29**(19): 2494–2495.

[41] The Java HotSpot Performance Engine Architecture. http://www.oracle.com/technetwork/java/whitepaper-135217.html, Date accessed: 2017-09-13.

[42] Frequently Asked Questions About the Java HotSpot VM. http://www.oracle.com/technetwork/java/hotspotfaq-138619.html, Date accessed: 2017-09-13.

[43] Oliveira RS, Rocha BM *et al.*: **Comparing CUDA, OpenCL and OpenGL Implementations of the Cardiac Monodomain Equations**. In *Proceedings of the 9th International Conference on Parallel Processing and Applied Mathematics - Volume Part II* , PPAM'11, Berlin, Heidelberg. Springer-Verlag, 2012: 111–120.

Appendices

The comparison of the SAXPY code example are listed for CUDA (Listing 1) and OpenCL (Listing 2).

Listing 1: SAXPY example for CUDA

```
/* SAXPY code example from  https://devblogs.nvidia.com/parallelforall/
    easy-introduction-cuda-c-and-c/ */

#include <stdio.h>

// The declaration specifier __global__ defines a kernel. This code
// will be copied to the device and will be executed there in parallel
__global__
void saxpy(int n, float a, float *x, float *y)
{
  // The indexing of the single threads is done with the following
  // code line
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  // Each thread is executing just one position of the arrays
  if (i < n) y[i] = a*x[i] + y[i];
}

int main(void)
{
  // Creating a huge number
  int N = 1<<20;
  float *x, *y, *d_x, *d_y;
  // Allocate an array on the *host* of the size of N
  x = (float*)malloc(N*sizeof(float));
  y = (float*)malloc(N*sizeof(float));

  // Allocate an array on the *device* of the size of N
  cudaMalloc(&d_x, N*sizeof(float));
  cudaMalloc(&d_y, N*sizeof(float));

  // Filling the array of the host
  for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  // Copy the host array to the device array
```

```
  cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);

  // Perform SAXPY on 1M elements. The triple chevrons dedicates how
  // the threads are grouped on the device
  saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);

  // Copy the result from the device to the host
  cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);

  // Display the result to the screen
  for(i = 0; i < N; i++)
    printf("%f\n", y[i]);

  // Free the memory on the host and device
  free(x);
  free(y);
  cudaFree(d_x);
  cudaFree(d_y);
}
```

Listing 2: SAXPY example for OpenCL (main)

```
// SAXPY code example from
// https://www.packtpub.com/books/content/hello-opencl

#include <stdio.h>
#include <stdlib.h>
#ifdef __APPLE__
 #include <OpenCL/cl.h>
#else
 #include <CL/cl.h>
#endif
#define VECTOR_SIZE 1024

//OpenCL kernel which is run for every work item created.
const char *saxpy_kernel =
"__kernel                                   \n"
"void saxpy_kernel(float alpha,      \n"
"                  __global float *A,       \n"
"                  __global float *B,       \n"
"                  __global float *C)       \n"
"{                                          \n"
"    //Get the index of the work-item       \n"
```

```
"    int index = get_global_id(0);          \n"
"    C[index] = alpha* A[index] + B[index]; \n"
"}                                          \n";

int main(void) {
  int i;
  // Allocate space for vectors A, B and C
  float alpha = 2.0;
  float *A = (float*)malloc(sizeof(float)*VECTOR_SIZE);
  float *B = (float*)malloc(sizeof(float)*VECTOR_SIZE);
  float *C = (float*)malloc(sizeof(float)*VECTOR_SIZE);
  for(i = 0; i < VECTOR_SIZE; i++)
  {
    A[i] = i;
    B[i] = VECTOR_SIZE - i;
    C[i] = 0;
  }

  // Get platform and device information
  cl_platform_id * platforms = NULL;
  cl_uint     num_platforms;
  //Set up the Platform
  cl_int clStatus = clGetPlatformIDs(0, NULL, &num_platforms);
  platforms = (cl_platform_id *)
  malloc(sizeof(cl_platform_id)*num_platforms);
  clStatus = clGetPlatformIDs(num_platforms, platforms, NULL);

  //Get the devices list and choose the device you want to run on
  cl_device_id     *device_list = NULL;
  cl_uint           num_devices;

  clStatus = clGetDeviceIDs( platforms[0], CL_DEVICE_TYPE_GPU,
                             0,NULL, &num_devices);
  device_list = (cl_device_id *)
  malloc(sizeof(cl_device_id)*num_devices);
  clStatus = clGetDeviceIDs( platforms[0],CL_DEVICE_TYPE_GPU,
                             num_devices, device_list, NULL);

  // Create one OpenCL context for each device in the platform
  cl_context context;
  context = clCreateContext( NULL, num_devices,
                             device_list, NULL, NULL, &clStatus);

  // Create a command queue
```

```
cl_command_queue command_queue = clCreateCommandQueue
                                 (context, device_list[0], 0,
                                  &clStatus);

// Create memory buffers on the device for each vector
cl_mem A_clmem = clCreateBuffer(context, CL_MEM_READ_ONLY,
                                VECTOR_SIZE * sizeof(float),
                                NULL, &clStatus);
cl_mem B_clmem = clCreateBuffer(context, CL_MEM_READ_ONLY,
                                VECTOR_SIZE * sizeof(float),
                                NULL, &clStatus);
cl_mem C_clmem = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                                VECTOR_SIZE * sizeof(float),
                                NULL, &clStatus);

// Copy the Buffer A and B to the device
clStatus = clEnqueueWriteBuffer(command_queue, A_clmem, CL_TRUE, 0,
                                VECTOR_SIZE * sizeof(float), A, 0,
                                NULL, NULL);
clStatus = clEnqueueWriteBuffer(command_queue, B_clmem, CL_TRUE, 0,
                                VECTOR_SIZE * sizeof(float), B, 0,
                                NULL, NULL);

// Create a program from the kernel source
cl_program program = clCreateProgramWithSource(context, 1,
                       (const char **)&saxpy_kernel, NULL, &clStatus);

// Build the program
clStatus = clBuildProgram(program, 1, device_list, NULL, NULL, NULL);

// Create the OpenCL kernel
cl_kernel kernel = clCreateKernel(program, "saxpy_kernel", &clStatus);

// Set the arguments of the kernel
clStatus = clSetKernelArg(kernel, 0, sizeof(float), (void *)&alpha);
clStatus = clSetKernelArg(kernel, 1, sizeof(cl_mem),
                          (void *)&A_clmem);
clStatus = clSetKernelArg(kernel, 2, sizeof(cl_mem),
                          (void *)&B_clmem);
clStatus = clSetKernelArg(kernel, 3, sizeof(cl_mem),
                          (void *)&C_clmem);

// Execute the OpenCL kernel on the list
size_t global_size = VECTOR_SIZE; // Process the entire lists
```

```
  size_t local_size = 64;            // Process one item at a time
  clStatus = clEnqueueNDRangeKernel(command_queue, kernel, 1,
                                    NULL, &global_size, &local_size,
                                    0, NULL, NULL);

  // Read the cl memory C_clmem on device to the host variable C
  clStatus = clEnqueueReadBuffer(command_queue, C_clmem, CL_TRUE, 0,
                                 VECTOR_SIZE * sizeof(float), C, 0,
                                 NULL, NULL);

  // Clean up and wait for all the comands to complete.
  clStatus = clFlush(command_queue);
  clStatus = clFinish(command_queue);

  // Display the result to the screen
  for(i = 0; i < VECTOR_SIZE; i++)
    printf("%f * %f + %f = %f\n", alpha, A[i], B[i], C[i]);

  // Finally release all OpenCL allocated objects and host buffers.
  clStatus = clReleaseKernel(kernel);
  clStatus = clReleaseProgram(program);
  clStatus = clReleaseMemObject(A_clmem);
  clStatus = clReleaseMemObject(B_clmem);
  clStatus = clReleaseMemObject(C_clmem);
  clStatus = clReleaseCommandQueue(command_queue);
  clStatus = clReleaseContext(context);
  free(A);
  free(B);
  free(C);
  free(platforms);
  free(device_list);
  return 0;
}
```

The Java code (Lisitng 3) which got ported to a library invocing the GPU without the code for LU decomposition and LU backsubstitution.

Listing 3: Java code, which got ported to the library. The function Smooth() and calcBoundIndex() stayed on the CPU and the rest was moved to the GPU. Another implementation of calcBoundIndex() was needed on the GPU which can be used by the GPU.

```
/**
 * Smoothes a rough chromatogram within a time range.
 * @param range
 *      number of seconds around given points
 * @param repeats
 *      number of smooth runs for each point
 */
public void Smooth(float range, int repeats, boolean copyRawDataFirst)
{
    float threshold;
    int i, j;

    // ============================================================
    // Calculate the minimum y value and
    // store it into threshold:
    // ============================================================

    threshold = Value[0][1];
    for (i=0; i<ScanCount; i++){
        if(Value[i][1]<threshold) threshold = Value[i][1];
    }
    if (copyRawDataFirst) copyRawData();

    // ============================================================
    // Smooth each point around the given time range
    // ============================================================

    for (j=0; j<repeats; j++)
    {
      int startScan = 0;
      int stopScan = ScanCount;
      if (startSmoothScan_>-1)
        startScan = startSmoothScan_;
      if (stopSmoothScan_>-1)
        stopScan = stopSmoothScan_;
      precalcPows_ = new float[Value.length];
```

```
      int preCalcStart = calcBoundIndex(startScan,range,false)-10;
      if (preCalcStart<0) preCalcStart = 0;
      int preCalcStop = calcBoundIndex(stopScan,range,true)+10;
      if (preCalcStop>ScanCount) preCalcStop =ScanCount;

      for (i=preCalcStart; i<preCalcStop; i++)
        precalcPows_[i] = (float)Math.pow(Value[i][2], 0.25);
      for (i=startScan; i<stopScan; i++)
        Value[i][3] = SmoothDataPoint(i, range, threshold);
      for (i=startScan; i<stopScan; i++)
        Value[i][2] = Value[i][3];
    }
}


private int calcBoundIndex(int dtIndx, float range, boolean posDirection
   ){
  int boundIndex = dtIndx;
  if (posDirection){
    while (boundIndex<(ScanCount-1)
        && (Value[boundIndex][0] - Value[dtIndx][0])<range)
      ++boundIndex;
  }else{
    while (boundIndex>0
        && (Value[dtIndx][0] - Value[boundIndex][0])<range)
      --boundIndex;
  }
  return boundIndex;
}

/**
 * smoothes rough spectrum at a specific point.
 *
 * @param dtIndx
 *           Index of point to be smoothed
 * @param range
 *           number of seconds around given points
 * @param threshold
 *           minumum value to pass back
 * @return
 *           smoothed value
 */
private float SmoothDataPoint(int dtIndx, float range, float threshold)
```

```
{
    int      order = 4;
    int      lower, upper;
    float    val;

    // ==============================================================
    // get boundary
    // ==============================================================

    lower = calcBoundIndex(dtIndx,range,false);
    upper = calcBoundIndex(dtIndx,range,true);

    while(upper-lower<10)
    {
        if (lower>0)                          --lower;
        if (upper<ScanCount-1)                ++upper;
        if( lower<=0 && upper>=ScanCount-1)   break;
    }

    // ==============================================================
    // get filter value
    // ==============================================================

    order = order < upper-lower-1 ? order : upper-lower-1;

    if(order < 1)         return Value[dtIndx][2];
    else                  val = SavGolFilter(dtIndx, lower, upper, order);

    if(val > threshold) return val;
    else                return threshold;
}

/**
 * Does a Savitzky Golay Filter around a given point
 *
 * @param dtIndx
 *          Index of point
 * @param lower
 *          lower range border
 * @param upper
 *          upper range border
 * @param order
 *          order of polynome
 * @return
```

```
 *          smoothed value
 */
private float SavGolFilter(int dtIndx, int lower, int upper, int order)
{
    float   mtrx[][];
    float   vec[];
    float   x = Value[dtIndx][0];
    int     indx[];
    float   sum;
    float adding = 0;
    int     i, j, k;

    indx = new int[order+1];
    vec = new float[order+1];
    mtrx = new float[order+1][order+1];

    // ================================================================
    // get "mtrx"
    // ================================================================

    for (i=0; i<=order; i++)
    {
        for (j=i; j<=order; j++)
        {
            sum = 0;

            for (k=lower; k<=upper; k++)
            {
              adding = 0f;
              for (int l=0; l<(i+j); l++){
                if (l==0)
                  adding = (Value[k][0] - x);
                else
                  adding = adding*(Value[k][0] - x);
              }
              if ((i+j) ==0) adding = 1f;

              if (Value[k][2] > 1)
                adding = adding * precalcPows_[k];
              sum += adding;

            }
            mtrx[i][j] = sum;
            mtrx[j][i] = sum;
```

```
        }
    }

    // ===============================================================
    // LU decomposition
    // ===============================================================

    myLUDcmp(mtrx, order+1, indx);

    // ===============================================================
    // get "vec"
    // ===============================================================

    for (i=0; i<=order; i++)
    {
        sum = 0;
        for (k=lower; k<=upper; k++)
        {
            if (Value[k][2]>1)
                sum += (float)(Math.pow(Value[k][0]-x, i)
                    * Value[k][2]
                    * Math.pow(Value[k][2], 0.25f));
            else
                sum += (float)Math.pow(Value[k][0]-x, i)
                    * Value[k][2];
        }
        vec[i] = sum;
    }

    // ===============================================================
    // LU backsubstition
    // ===============================================================

    myLUBksb(mtrx, order+1, indx, vec);
    x = vec[0];
    return x;
}
```

For the JNI a Java class needs to be created and functions called in the library need to be declared as native. After compiling with javac the header file can be created with javah which includes a machine generated header file. A minimal working example is provided in Listing 4 and 5

Listing 4: An example Java file for JNI. The Java file needs to be compiled (javac) before it is translated to a .h file (javah). The functions for the native library need to be declared as native.

```
public class HelloJNI {
   static {
      System.loadLibrary("hello"); // Load native library at runtime
   }
   // Declare a native method
   private native void sayHello();

   public static void main(String[] args) {
      new HelloJNI().sayHello();  // invoke the native method
   }
}
```

Listing 5: The machine generated .h file from the Java example file.

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class HelloJNI */

#ifndef _Included_HelloJNI
#define _Included_HelloJNI
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     HelloJNI
 * Method:    sayHello
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_HelloJNI_sayHello
  (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```