Universität
Münster

University of Münster
School of Business & Economics

Nina Herrmann

Exploring High- and Low-Level Approaches for GPGPU Processing of Telescope Data.

Master Thesis

at the Chair for Practical Computer Science
(University of Münster)

Principal Supervisor: Prof. Dr. Herbert Kuchen

Presented by: Nina Herrmann [417708]
Dieckstraße 6
48145 Münster
+49 17661328318
nina.herrmann@uni-muenster.de

Submission: 16th September 2019

We stand at the threshold of a many core world. The hardware community is ready to cross this threshold. The parallel software community is not.

Tim Mattson, principal engineer at Intel.

Contents

Figures IV
Tables V
Listings VI
1 Introduction 1
2 Background 2
  2.1 Polyphase Filterbank 2
  2.2 Comparison of GPU and CPU 6
  2.3 Low-Level Approaches to GPGPU 8
  2.4 High-Level Approaches to GPGPU 12
3 Methodology 20
  3.1 Systematic Approach 20
  3.2 Related Work 22
  3.3 Research Objectives 22
  3.4 Requirements Analysis 23
4 Prototype Design 25
  4.1 Low-Level Implementation 25
  4.2 High-Level Implementation 35
5 Evaluation 39
  5.1 Runtime Measurements settings 39
  5.2 Comparison of CPU and GPU Programs 41
  5.3 Intra Program Comparison 43
  5.4 GPGPU Program Comparison 50
  5.5 Non-Functional Requirements 59
  5.6 Interpretation of Results 61
  5.7 Implications of Results 62
6 Outlook 64
7 Conclusion 66
Appendix 68
References 73

Figures

Figure 1 FIR-Filter Calculation 3
Figure 2 Butterfly Pattern 6
Figure 3 Comparison of the Architecture of GPUs and CPUs [cf. 22, 1. Introduction] 7
Figure 4 Grid of Thread Blocks cf. [22, 2.2. Thread Hierarchy] 9
Figure 5 Memory Hierachy cf. [23, 2.3. Memory Hierarchy] 11
Figure 6 Example for Applying a Map Skeleton 14
Figure 7 Example for Applying a Fold Skeleton 14
Figure 8 Process of DSRM [cf. 24] 21
Figure 9 Exemplary Process of PFB 24
Figure 10 FIR-Filter Data Division to with Fixed Number of Channels per Block 28
Figure 11 FIR-Filter Data Division Calculating Multiple Elements per Thread ... 31
Figure 12 CPU and GPU Programs: Complete Runtime 42
Figure 13 Low-Level Program: Grouped Runtime 43
Figure 14 Low-Level Program: Runtime of FIR-Filter 46
Figure 15 Low-Level Program: Runtime of FFT 47
Figure 16 Low-Level Program: Runtime of PFB Steps 48
Figure 17 High-Level Program: Grouped Runtime 49
Figure 18 High-Level Program: Runtime of PFB Steps 50
Figure 19 GPGPU Programs: Comparison of Runtime of FIR-Filter 51
Figure 20 GPGPU Programs: Comparison of Runtime of FFT 53
Figure 21 GPGPU Programs: Comparison of Runtime for Memory Allocations .. 54
Figure 22 GPGPU Programs: Comparison of Complete Runtime 57

Tables

Table 1 Settings for Runtime Measurements 41
Table 2 CPU and GPU Programs: Complete Runtime 42
Table 3 Low-Level Program: Runtime and Percentual Share of Grouped Operations 44
Table 4 Low-Level Program: FIR-Filter Runtime (ms) 45
Table 5 Low-Level Program: FFT Runtime (ms) 47
Table 6 High-Level Program: Runtime and Percentual Share of Grouped Operations 50
Table 7 GPGPU Programs: Runtime and Speed-up for FIR-Filter Calculations . . 51
Table 8 GPGPU Programs: Runtime and Speed-up of FFT 53
Table 9 GPGPU Programs: Runtime For Memory Management Operations (ms) 54
Table 10 GPGPU Programs: Runtime for Grouped Memory Management Operations (ms) 55
Table 11 GPGPU Programs: Allocated Memory Space 56
Table 12 GPGPU Programs: Runtime and Speed-up of Complete Programs (ms) . 57
Table 13 GPGPU Programs: Runtime of PFB Steps (ms) 58
Table 14 Low-Level Program: Runtime (ms) 71
Table 15 High-Level Program: Runtime (ms) 71
Table 16 CPU Program: Runtime (ms) 72

Listings

Listing 1 CUDA Example cf. [22] . . . 10
Listing 2 Example Program Musket: Configuration Block . . . 16
Listing 3 Example Program Musket: Data Structures . . . 16
Listing 4 Example Program Musket: User Functions . . . 17
Listing 5 Example Program Musket: Main Function . . . 18
Listing 6 CUDA FIR-Filter Implementation: Global Memory . . . 26
Listing 7 CUDA FIR-Filter Implementation: Shared Memory . . . 27
Listing 8 CUDA FIR-Filter Implementation: Data Reuse . . . 30
Listing 9 CUDA FIR-Filter Implementation: Multiple Elements per Thread . . . 32
Listing 10 CUDA DFT Implementation . . . 34
Listing 11 Musket Implementation: FIR-Filter . . . 36
Listing 12 Musket Implementation: FFT . . . 37
Listing 13 Musket Implementation: Complete PFB . . . 69

1 Introduction

General purpose computation on graphics processing units (GPGPU) is a powerful tool to accelerate programs which can be solved with the single instruction multiple thread (SIMT) execution model, and have a massive amount of data to process. However, writing efficient parallel programs for graphics processing units (GPUs) is an arduous task since bugs caused by deadlocks and race conditions are troublesome to detect and strenuous to fix. Moreover, expertise is required, especially for choosing the most suitable memory spaces, deciding on a number of threads, and allocating tasks to the threads. Those barriers are simplified or overcome by high-level approaches. High-level approaches provide an additional level of abstraction which hides hardware-specific details from the programmer and simplifies parallel programming. Examples for concepts for such approaches are skeletons [6] and the bulk synchronous parallel (BSP) bridging model [9]. Those concepts enable programmers with little knowledge about hardware-specific characteristics, and low-level frameworks to write parallel programs. Moreover, programs written with high-level approaches are often portable to different hardware architectures and require little effort to be maintained. The major disadvantage of high-level approaches is that the level of abstraction usually entails a loss in performance.

Commonly used benchmarks include matrix multiplication, the Frobenius norm, and Fish School Search [36]. Those benchmarks should be complemented with an practical application context, to prove the applicability of the high-level approach. Furthermore, it can be reasoned whether possible losses in performance are negligible since the practical application context defines an exact requirement for the execution time. Data rates in radio astronomy are infamously high. For example, at the Max Planck institute for radio astronomy (MPIfR), 2 billion samples are produced by one telescope each second with one sample being one float. It can be expected, that with advancing telescopes, the rates for the incoming data even increase. Ideally, the data should be processed in real-time in a streaming manner. This process includes multiple computations, such as filters and the discrete Fourier transform (DFT) which get computational intensive. For this purpose, fast processing of data is required, which might be solvable with GPGPU programming. Hence, it is a meaningful context for comparing high- and low-level approaches for GPU programming.

The given problem is approached by first, providing the necessary background knowledge for signal processing and GPGPU programming. Second, the methodology and the research objectives are defined. Subsequently, the programs created to solve the problem are described. Next, those programs are evaluated and compared. Last, an outlook to future research topics is given, and all findings are concluded.

2 Background

To understand the designed solution for the application context and the solution approach, fundamental knowledge of signal processing and GPGPU programming is essential. Regarding the application context the individual steps of the polyphase filterbanks (PFBs) need to be introduced. The explanation about the calculations performed within the steps of the PFB will be followed by a description of the GPGPU approach. This sequence enables to show the suitability of parallel GPU programming for accelerating the computation of a PFB. Moreover, distinct approaches to GPGPU programming are presented. Those are grouped into high- and low-level approaches.

2.1 Polyphase Filterbank

Filterbanks (FBs) are used to revise, analyse, and summarise data. They are commonly used in signal and image compression, and processing. Typical examples are the elimination of noise in audio signals and graphic equalisers for images. There exist multiple processes and mathematical approaches to realise FBs. PFBs are a special kind of FBs which combine multiple processing steps to adjust and summarise the incoming data. This thesis focuses on one specific PFB which combines a finite impulse response (FIR)-filter and the fast Fourier transform (FFT). This specific type of PFB was chosen since the consultation of the MPIfR showed that those steps are commonly used for signal processing. Hence, it fulfils the claim for a real application context. Supplementary computational steps, such as data oversampling, are not included in this thesis. The major focus is put on the comparison of the GPU approaches.

2.1.1 Finite Impulse Response Filter

FIR-filters are commonly used in digital signal processing and return an impulse response with finite length. This conditions that the application of the filter never turns unstable or causes autonomous waves [32, p.53]. There exist different types of FIR-filters, such as windowing filter, equiripple filter, or weighted least square filters [32, 2.3 Lowpass and Highpass Filter Design, p.53-60]. The filters differ in the numbers used for the coefficients. Independent from those filter coefficients the sequences of steps to perform the computations are similar for all filters and specified in Equation 2.1 [27]. For the computation of the filter, the sum of weighted values of the past is calculated as the new element. For each element, K products are summed up. One product consists of the element itself or a succeeding element multiplied by the filter coefficient. In total, $K - 1$ elements from the past are considered.

$$y(n) = \sum_{k=0}^{K-1} h(k)x(n-k) \tag{2.1}$$

This type of filter is used as the first step in the PFB to process the incoming signal of the telescopes. Specifically, three terms are used in radio astronomy to define the parameters for the FIR-filter and the DFT: *channels*, *spectra*, and *taps*. One *channel* contains one sub-sequence of data. *Taps* represent the number of values taken from the past, previously referenced as $K - 1$. Most commonly, fewer taps are used when the number of channels rises and vice versa. One *spectrum* contains one element from each channel.

For the ease of understanding, the data is organised in a matrix-like structure which is displayed in Figure 1. The first item in the figure is an array of the input data which should be processed. In the context of radio astronomy, the elements of the input are signals from radio telescopes. Therefore, the input is a sequence of floating-point numbers. However, the FIR filter could also be applied to other sources with more complex input types.

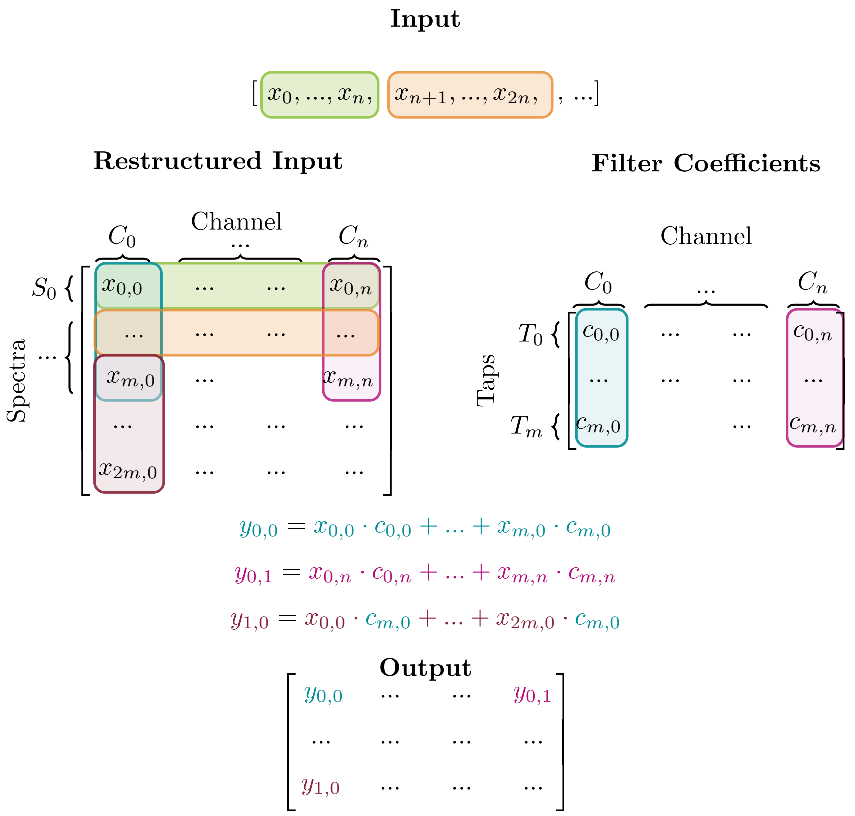
Input
$[x_0, ..., x_n, \; x_{n+1}, ..., x_{2n}, \; ...]$
Restructured Input
Filter Coefficients
Channel
C_0
C_n
S_0
Spectra
$x_{0,0}$
$x_{0,n}$
$x_{m,0}$
$x_{m,n}$
$x_{2m,0}$
Taps
T_0
T_m
$c_{0,0}$
$c_{0,n}$
$c_{m,0}$
$c_{m,n}$
$y_{0,0} = x_{0,0} \cdot c_{0,0} + ... + x_{m,0} \cdot c_{m,0}$
$y_{0,1} = x_{0,n} \cdot c_{0,n} + ... + x_{m,n} \cdot c_{m,n}$
$y_{1,0} = x_{0,0} \cdot c_{m,0} + ... + x_{2m,0} \cdot c_{m,0}$
Output
$y_{0,0}$
$y_{0,1}$
$y_{1,0}$

Figure 1 FIR-Filter Calculation

The array is restructured to a matrix. Each row contains one sub-sequence of data; i.e. one spectrum. Exemplary two spectra are highlighted in orange and green. One spectrum contains one element for each channel, in this case, n channels are distinguished. Therefore, one column contains the data of one channel. For calculating the FIR-filter m elements are taken; hence, the number of taps is equal to m. Exemplary, subsets of elements which are processed to calculate one element of the output are highlighted in blue, pink, or red. For calculating a new value, they are multiplied and summed up with the corresponding elements from the filter, as shown in the equations in the figure. The filter elements are highlighted in blue or pink. In total $m \times n$ filter coefficients are required. Noticeably, the filter coefficients do not change for one channel. The blue elements are multiplied with the same filter elements as the red elements.

Three exemplarily calculation steps of the filter are displayed. To calculate the first output element of the matrix, the first element and m subsequent elements of the same channel are multiplied with the corresponding filter elements. Respectively, for calculating another element in the channel (red elements), the new element is calculated from the red input data, but the same filter coefficients are used. From this specification, it is deduced that the number of filter coefficients is always equal to $\#channels \times \#taps$, while the input can always be replaced with new data, and most commonly arrives in a streaming manner.

For calculating all element of the output, for each element m multiplications are performed and summed up. Regarding the complexity class for the calculation, the FIR-filter is in the class $O(m \times n)$ where m is the number of multiplications performed per element, and n is the number of input elements.

2.1.2 Discrete Fourier Transform

The DFT is a tool from the Fourier analysis which converts a finite and continuous signal into a discrete, periodic frequency spectrum. It is applied in signal processing to determine the amplitude of frequencies and to identify the dominating frequencies in a sampled signal [32, p.61]. This technique is used in audio processing to separate different sources of noise. In the context of electromagnetic radiation, it is used to separate distinct sources for electromagnetic waves.

More precisely, the complex vector $a_k = (a_0, ..., a_{N-1}) \in \mathbb{C}^N$ of input data is converted into the complex vector $\hat{a}_k = (\hat{a}_0, ..., \hat{a}_{N-1}) \in \mathbb{C}^N$ with the coefficients displayed in Equation 2.2.

$$\hat{a}_k = \sum_{n=0}^{N-1} e^{-2\pi i \times \frac{nk}{N}} \times a_n = \sum_{n=0}^{N-1} \left[\cos\left(\frac{2\pi kn}{N}\right) - i \times \sin\left(\frac{2\pi kn}{N}\right) \right] \times a_n, \quad k = 0, ..., n-1 \qquad (2.2)$$

One can imagine the sum, resulting in one element of the output vector, as the centre of mass of n points. Each point is through the multiplication with the e-function a number in a complex plane. In case the sum has a high x-value it is likely that the frequency for a source has been found.

For calculating the DFT, different algorithms exist to speed up the computation, most prominently the FFT. The idea for the algorithm emerged 1805 by Gauss. However, it did not receive significant attention until Cooley and Tukey developed an akin algorithm and published their solution in 1965. The FFT reduces the complexity of the naive calculations from $O(n^2)$ to $O(n \times \log(n))$ with n being the number of complex numbers to process.. For an input which has a size of the power of two, a radix-2 decimation-in-time (DIT) is the most commonly used approach. The speedup is achieved by splitting the calculations into multiple small calculations and storing the intermediate results. Essentially, the previously shown formula is split into the DFT of all even and all odd numbers as depicted in Equation 2.3. With multiple conversions this formulae can be written as the sum of the DFT of even numbers (E_k) and the multiplication of the complex exponential function and the DFT of the odd numbers (O_k).

$$\hat{a}_k = \sum_{j=0}^{N/2-1} (e^{-2\pi i \times \frac{2jk}{N}} \times a_{2j}) + \sum_{j=0}^{N/2-1} (e^{-2\pi i \times \frac{2j+1k}{N}} \times a_{2j+1}) = E_k + e^{\frac{2\pi ik}{N}} O_k \tag{2.3}$$

This split is repeatedly applied by the algorithm of Cooley and Tukey, by recursively calling the function `ditfft2` specified in Algorithm 1 [28, p.207]. First, an if-clause checks whether the size of the elements to process equals one. In case only one element is left the base case of the recursive call is reached, and the element itself is returned. With this bottom-up approach the intermediate result are calculated. One intermediate result is the e-function to a power of the respective sinus or cosine.

Algorithm 1 Pseudo Code Cooley Tukey Algorithm

```
$X_0, ...,_{N-1} \leftarrow$ ditfft2 $(x, N, s)$ : dotommentDFT of $(x_0, x_s, x_{2s}, ..., x_{(N-1)s})$ :
if $N = 1$ then
    $X_0 \leftarrow x_0$
else
    $X_0, ..., \frac{N}{2-1} \leftarrow$ditfft2$(x, \frac{N}{2}, 2s)$                ▷ DFT of $(x_0, x_{2s}, x_{4s}, ...)$
    $X_{\frac{N}{2}}, ...,_{N-1} \leftarrow$ditfft2$(x + s, \frac{N}{2}, 2s)$          ▷ DFT of $(x_s, x_{s+2s}, x_{s+4s}, ...)$
    for $k = 0$ to $\frac{N}{2-1}$ do
        $t \leftarrow X_k$
        $X_k \leftarrow t + exp(-2\pi ik/N)\, X_k + \frac{N}{2}$
        $X_{k+\frac{N}{2}} \leftarrow t - exp(-2\pi ik/N)\, X_k + \frac{N}{2}$
```

This algorithm implements the butterfly pattern depicted in Figure 2. The number of iterative calls is equal to $log(n)$ where n is the input size. After each iteration the respective result is returned and used for calculating the next result.

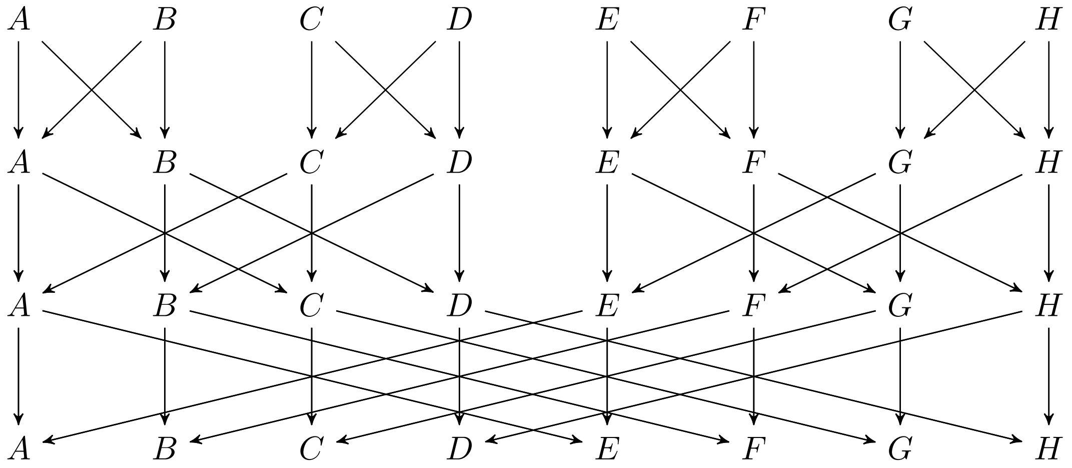

Figure 2 Butterfly Pattern

In addition, there exist improved versions of the algorithm, which optimise the sequence of action based on specific input sizes. Because the algorithm breaks the calculation of the DFT into smaller DFT it is also possible to include alternative calculations only at suitable subsequences. Since the input size can be chosen freely for the comparison of the GPGPU approaches, further algorithms are neglected, and it is focused on the implementation of the presented steps.

In the context of signal processing, the DFT is applied to each spectrum of data. This means for the computation of a fixed number of z spectra to process with j item per spectra and therefore j channels the computation of the corresponding FFT is in $O(z \times j \times \log(j))$. Since the number of spectra multiplied with the number of elements per spectra is equal to the number of overall elements, it can be concluded that for the PFB the complexity is$O(n \times \log(j))$ with n being the number of elements and j being the number of spectra.

2.2 Comparison of GPU and CPU

Originally, GPUs were used as processors to compute graphics operations for a display device. In addition, in recent years, GPUs are used in combination with central processing units (CPUs) or stand-alone to accelerate the execution of parallel programs, which is called GPGPU. To comprehend why GPUs are used to accelerate programs, the underlying hardware architectures of CPUs and GPUs are explained. Furthermore, from this explanation it is deduced which types of programs are especially suitable for GPGPU. On the basis of the introduced knowledge, it is argued why the steps presented in the previous section are suitable to be parallelised on GPUs.

The main differences between the architecture of a GPU and a CPU architecture are depicted in Figure 3. The precise number of components varies depending on the concrete hardware. However, the general idea is transferable. The colours in the illustration depict components serving the same purpose. CPUs usually have a limited number of caches and control units which are shared between all arithmetic logic units (ALUs). Each ALU executes rather complex computations. Multiple extensions for CPU programming such as SSE, MMX, and AVX exist to start multiple threads. However, they are less flexible in numbers and do not eliminate the disadvantage of inefficient memory accesses.

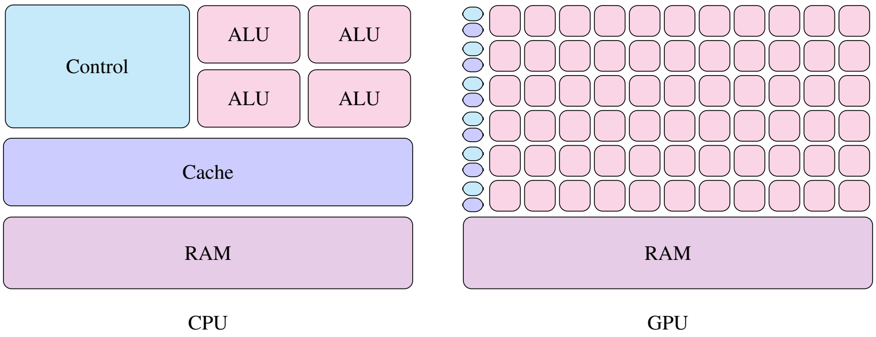

Figure 3 Comparison of the Architecture of GPUs and CPUs [cf. 22, 1. Introduction]

A GPU has multiple, small caches and control units shared between different computational units. Most importantly, a GPU has significantly more small ALUs. This architecture supports SIMT operations. All threads perform the same instruction but work on distinct subsets of the data. Ideally, all operations are independent from each other. Since the program does not change, the control flow is less important; i.e. the sequence in which the threads are executed is of minor relevance. As soon as the sequence of action requires a lot of attention the program requires communication and it is less suitable for the GPU. Memory access latency is a smaller problem on the GPU since it is hidden with other computations performed, instead of updating big data caches [22, 1. Introduction]. Conclusively, CPUs are rather developed for complex computations with a high amount of communication and a sophisticated control flow, while GPUs are ideal for solving a lot of smaller operations without communication between the different threads.

2.3 Low-Level Approaches to GPGPU

In recent years, the market for GPU-production has been dominated by Nvidia and AMD, with about 77,3% share of Nvidia and 22,7% AMD [1]. Unlike developing programs for CPUs, GPGPU programming often requires to consider the hardware-specific characteristics to develop efficient programs. However, there exist significant differences between the hardware of the GPUs, impeding the development of efficient programs. Two low-level approaches gained remarkably attention in practice: CUDA and open computing language (OpenCL). Alternatively, Microsoft developed a runtime library, C++ AMP, for parallel programming. However, the library is not commonly used, and therefore the description is neglected for this context.

2.3.1 CUDA

The technology company Nvidia developed a general-purpose parallel computing platform and programming model for GPGPU programming called CUDA. CUDA can only be used for Nvidia GPUs since it relies on the Nvidia architecture. CUDA has language extensions for the high-level languages C/C++ and Fortran, and recently supporting tools for Python have been published. Essentially, the existing languages are enriched with additional methods which enable, for example, memory transfers to GPU memory. Afterwards, the program can be compiled on the target platform. Due to the ability to allow fine-grained optimisation, it is considered a low-level approach to GPGPU programming. Moreover, CUDA provides the possibility to work with PTX, a low-level parallel thread execution virtual machine and instruction set architecture (ISA) [2]. This approach is rarely used due to the extreme overhead of very low-level operations such as explicitly managing memory addresses. In the following, it is focused on the C/C++ extension of CUDA.

The two most essential operations for GPGPU programming are the allocation of memory and the configuration of the number of threads started. To coordinate threads, they are structured in three physical levels, which are displayed in Figure 4. Each level has access to distinct memory spaces which will be explained subsequently. The smallest computational unit is a single thread depicted in the blue box. Threads are always started in one warp. Warps are not displayed in the figure since they are not managed by the user. Threads within one warps are always started together by the streaming multiprocessor (SM) and one warp always contains 32 threads. For this reason, programs working

---

[1] https://de.statista.com/statistik/daten/studie/197681/umfrage/marktanteile-am-absatz-von-grafikkarten-weltweit-seit-dem-3-quartal-2010/ access date: 15.07.2019

[2] https://docs.nvidia.com/cuda/parallel-thread-execution/index.html accessed:16/07/2019

on data structures which have a size divisible by 32 usually perform better, since every thread has a element in the data structure.

The next higher level are blocks depicted in rose. One thread always belongs to one block. The number of threads which can be started per block is limited to 1024 threads since compute capability 2.x [23, H.Compute Capabilities]. Previously, 512 threads per block could be started. In CUDA the compute capability is used to determine to which generation a GPU belongs and which features are supported. This information is also used by the compiler to optimise the program. Inside the block, threads can be arranged one-, two-, or three-dimensional. This does not physically change the structure but is helpful to for example, index matrices. In the example, threads are aligned two-dimensional. Due to the warps, it is in most cases reasonable to have a multiple of 32 started in the x-dimension. Otherwise, one warp has threads which do nothing, i.e. idling threads.

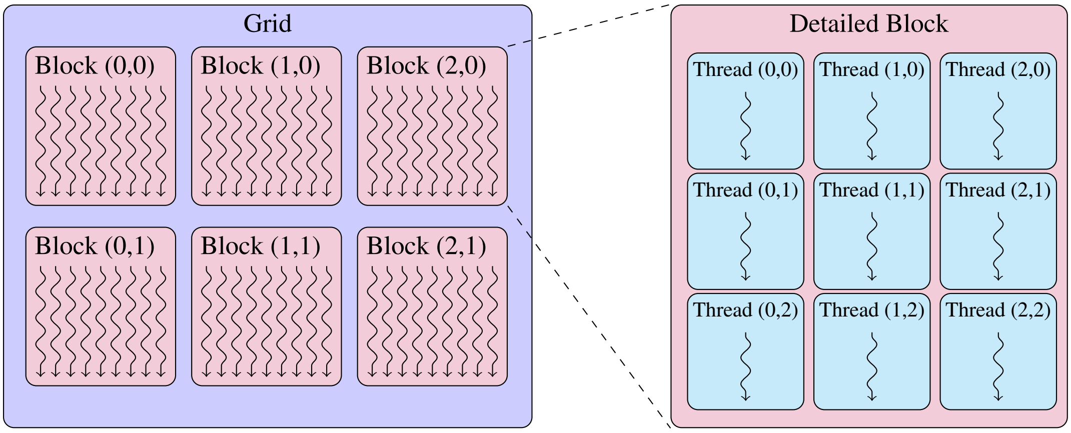

Figure 4 Grid of Thread Blocks cf. [22, 2.2. Thread Hierarchy]

The next higher level of structuring threads is a grid which groups multiple blocks. Blocks can again be aligned one-, two-, or three-dimensional. The depiction starts a two-dimensional grid with 3×2 blocks. Each block contains the same number and structure of threads. At most $2^{31} - 1$ block per x-dimension and 65535 blocks per y- and z-dimension can be started. However, depending on the architecture, not all blocks can be executed immediately. The number of blocks which can be started concurrently is limited by the number of SM and the generation of the GPU. A SM can start only a limited number of blocks, warps, and threads. The exact number depends on the compute capability. For example, for compute capability 3.7 a SM is restricted to 16 blocks, 64 warps, and 2048 threads. For example, in case 3 blocks with each 1024 threads are started on the SM one block is queued.

To executed code on the GPU so called kernel functions, usually referred to as kernels, are called. Those functions are flagged with the keyword `__global__`. The call of those functions requires the number of threads and blocks which should be started as arguments. One kernel call result in one grid which belongs to the execution. The kernel function should follow the SIMT execution model. This means the same function is executed by multiple threads, mostly performing the same operation, allowing parallelism by accessing different elements of the data structure. This is possible due to the usage of unique thread and block indices. For example, in Listing 1 a parallel function `VecAdd` is called with `N` threads and one block. Three floating pointers are given as arguments, `C` for writing the output and `A` and `B` as the input arrays. The input pointers are accessed with the thread index `threadIdx.x`. In the function the single elements of the array are added.

```
__global__ void VecAdd(float* A, float* B, float* C) {
  int i = threadIdx.x;
  C[i] = A[i] + B[i];
}
int main(){
  VecAdd<<<1, N>>>(A, B, C);
}
```

Listing 1 CUDA Example cf. [22]

Two further flags exist in CUDA: `__device__` and `__host__`. Host functions are executed on the CPU and device functions are executed on the GPU but can only be called from global functions. Functions executed on the GPU should always be designed to support the SIMT model. This means that the number of possible execution paths within the function should be minimised since distinct execution paths might be sequentially executed. For example, in case of an if-statement half of a warp might execute a different function than the other half. In this case the 32 threads cannot be started concurrently but are executed in two sequential groups which slow down the overall performance. This is called branch divergence.

The other major task in the development of parallel programs in CUDA is to choose and allocate memory spaces on the GPU. The memory hierarchy is displayed in Figure 5. Essentially, three layers of memory exist. Global memory is accessible for every thread on the GPU. Usually, at the beginning of a program the data to process is first copied from the CPU memory to global memory. It is possible to make parts of the global memory constant or texture memory. Constant memory is allocated by using the `__constant__` label and is read-only for the execution of one kernel. Texture memory is another type of read-only memory, which can reach a higher effective bandwidth since it is cached on the chip. Therefore, memory requests to off-chip global memory are reduced. However, this is only effective when the right values are cached, which depends on the overall program. Moreover, texture memory is designed to support 2D spatial locality. Spatial locality implies that threads are likely to access addresses which are close to the address a neighbouring thread accesses; with a 2D spatial locality, the addresses are checked for two dimensions. For using texture memory, special functions for allocation are used reserving space in the global memory. The function varies due to the number of dimensions that should be considered and the operations which should be performed on the texture memory.

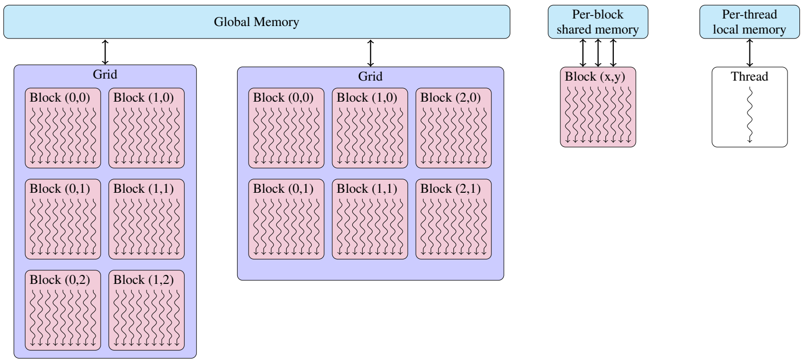

Figure 5 Memory Hierachy cf. [23, 2.3. Memory Hierarchy]

Even fast to access is the shared memory, which is approximately 100x lower in latency compared to global memory [3]. The access time can hardly be compared to texture and constant memory since the major deceleration occurs when cache misses occur. This depends on the specific program. Shared memory is available on chip. The amount of shared memory available is limited, depending on the architecture, and varies from 16 KB to 112 KB per SM and 48 KB to 96 KB per block. Shared memory can be reserved with a parameter in the kernel function call or within the kernel with the `__shared__` label. Within one block threads can access the data loaded from other threads into the shared memory. This enables efficient programs for, for example, parallel reduction where it is inevitable to consider other elements. However, shared memory is only efficient in case the data is sufficiently often reused since the copy step from global to shared memory and back also requires time. Moreover, using shared memory can be inefficient when bank conflict occur. The shared memory is structured in multiple banks which store subsequences of the data. In case threads access the same bank their request cannot be served sequentially and need to be serialized. The prevention of bank conflicts depends on the compute capability since banks are differently designed depending on the GPU

---

[3] https://devblogs.nvidia.com/using-shared-memory-cuda-cc/ accessed: 16/07/2019

architecture. The last type of memory is per-thread local memory. For devices with compute capability 2.x or higher, each thread has 512 KB of local memory. Local memory is not explicitly assigned but managed by the compiler.

Memory allocation and structuring of threads provide two powerful tools to create high performant programs in CUDA. Certainly, they are only a subset of all possible operations. CUDA provides much more features, for example, asynchronous memory transfers and a variety of libraries.

2.3.2 Open Computing Language

OpenCL is maintained by the companies AMD, IBM, Intel, and Nvidia. In contrast to CUDA, OpenCL supports CPUs, GPUs, and digital signal processorss (DSPs) of different manufacturers, including AMD, ARM, Creative, IBM, Imagination, Intel, Nvidia, Qualcomm, Samsung, Vivante, Xilinx, and ZiiLABS [4]. OpenCL is an application interface containing the programming language OpenCL C. This language enriches the C language with data-types and functions to enable parallel programming. However, also some restrictions to the C language are made. Recursion, function pointers, bit fields and variable-length arrays are not supported or explicitly forbidden. Apart from that, OpenCL is similarly constructed to CUDA. However, instead of working on the GPU as default, any device can be chosen. For this purpose the types `CL_DEVICE_TYPE_GPU`, `CL_DEVICE_TYPE_CPU`, and `CL_DEVICE_TYPE_ACCELERATOR` are available. After the respective device is chosen, a context for the device has to be created, the memory has to be allocated, the compute program has to be created, the executable for the program has to be built, the individual arguments for the device have to be set, and finally the program is enqueued for execution. Also the memory architecture is similar to the CUDA model. In general, the advantage of OpenCL is the support of other hardware types. Regarding the performance, different results can be found. SU ET AL. found CUDA to be 3.8%-5.4% faster when comparing 5 benchmarks [33] while more recently GIMENES ET AL. found no significant differences [17]. For the context of this thesis, CUDA will be used, since it is speciallised for GPUs and performed in some studies better.

2.4 High-Level Approaches to GPGPU

Parallel programming is an error-prone and strenuous task for inexperienced programmers. Bugs caused by deadlocks and race-conditions are difficult to detect and arduous to fix. This task gets increasingly difficult when using a complex hardware architecture.

---

[4] https://www.khronos.org/conformance/adopters/conformant-companies#opencl accessed: 15.07.2019

High performance cluster (HPC) typically include multiple nodes, CPUs, and accelerators such as GPUs or manycore processors. To exploit all hardware components, different frameworks such as OpenMP, OpenCL, and CUDA have to be used. Using multiple tools, in combination, makes the creation of programs even more challenging.

Low-level approaches have several disadvantages compared to high-level approaches: They require expertise in the domain of GPGPU programming, are often not easily transferable to different hardware architectures, and require effort to be maintained. High-level approaches attenuate those shortcomings by providing an extra level of abstraction, which most commonly takes care of memory management and hardware-specific configurations. This approach simplifies the maintenance of the program since the program itself is smaller, and with the continuous development of the high-level framework, new features are included. The development of the program requires less expertise since commonly known programming patterns are used. Lastly, the approach might provide configurations to generate code for different hardware architectures, increasing the portability of a program.

2.4.1 Skeletons

One used parallel programming model for developing a high-level approach are algorithmic skeletons. In this approach, a skeleton provides an additional level of abstraction by hiding the implementation details behind a common parallel programming pattern [5]. Commonly, the functions which wrap the skeletons are called higher-order functions. From multiple skeletons, a more complex program can be built. Examples for approaches include: Muesli[5], FastFlow[6], eSkel[7], and SkePU[8].

Skeletons are commonly categorized into two distinct groups: data-parallel and task-parallel skeletons. Data-parallel skeletons focus on computation and on one or more distributed data structures. During the execution, the data is manipulated by some operation. In contrast task-parallel skeletons focus on communication between different computational nodes [15]. Some exemplary skeletons will be introduced to increase the understanding of working with skeletons.

---

[5] http://muesli.uni-muenster.de/

[6] http://calvados.di.unipi.it/

[7] https://swmath.org/software/4423

[8] https://www.ida.liu.se/labs/pelab/skepu/

Map

The *Map* skeleton applies a function to each element and replaces the element with the result of the function. An example is to take the square of each element shown in Figure 6. From a parallelisation perspective, this operation is often ideal for a GPU. In the given example for each element, a thread is started, which squares the element. However, *Map* skeletons accessing multiple elements of the data structure are more complicated to implement. Moreover, to guarantee correct implementations, the operation needs to be scrutinised. In case elements from the data structure are reused, it has to be assured that the initial values of the data structure are used.

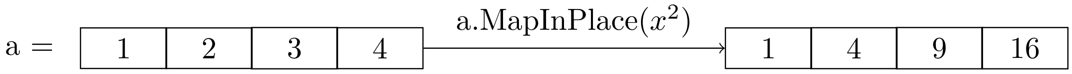

Figure 6 Example for Applying a Map Skeleton

Fold

Another data-parallel skeleton is *fold*. *Fold* reduces a data structure by successively applying a function. The result is always a singular element. This can be used, for example, to get the minimum of a list of element shown in Figure 7. However, the implementation on a GPU is not trivial since an efficient way to read data structures between different threads has to be implemented.

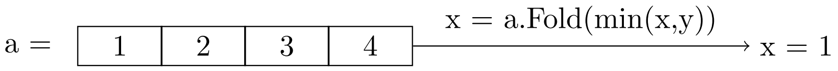

Figure 7 Example for Applying a Fold Skeleton

Farm

An example of a task-parallel skeleton is *farm*. Conceptually, a *farm* skeleton consists of a farmer and multiple workers. The farmer receives a sequence of tasks and distributes the task to the workers. Then the results are returned to the farmers who passes the result to the next process. As pointed out by Poldner and Kuchen the straightforward implementation might lead to the farmer being a bottleneck when an enormous amount of workers has to be managed [26]. For this purpose, the implementation of a farm skeleton might not be as trivial as expected. This becomes even more troublesome when it is taken into consideration that in HPC environments, not all computational nodes have the same computational power, i.e. the workers have different abilities to work on task.

Branch and Bound

However, not all skeletons can be clearly assigned. An example of a mixed skeleton is the implementation of branch and bound. Branch and bound is a mathematical method often utilised in operation research to systemically find the best of multiple solutions. The algorithm creates all possible solutions in a tree-structure and afterwards starts checking whether the solutions can become better than the current optimal solution. The problem is divided into sub-problems to accelerate the computation. For those problems, the lower- and upper-bound for the result is calculated. In this part the skeleton is data-parallel. After the calculation, the results of the skeletons have to be summarised, and depending on the result, the remaining possible solutions are rechecked. In this part, tasks have to be distributed again, like in task-parallel skeletons. For a concrete implementation proposal see the proposal by POLDNER and KUCHEN [25].

2.4.2 Muenster Skeleton Tool

To gain a concrete idea of how a skeleton tool might look like, the Muenster skeleton tool for high-performance code generation (Musket) is introduced. As the name implicates, it is a skeleton-based tool which defines a domain specific language (DSL) called Musket which is compiled into parallel code [36]. Musket was build and is maintained since 2015 at the University of Münster by the chair of practical computer science.

The structure of the DSL Musket can be divided into four parts: (1) a configuration block, (2) a part which specifies the required data structures, (3) user function(s) which are executed in parallel, and (4) a main function calling the skeletons with the suitable user functions. The following paragraphs will explain those parts at an exemplarily program.

Configuration Block

At the start of the program, the configuration parameters are defined. Five parameters have to be set: `PLATFORM`, `PROCESSES`, `CORES`, `GPUS`, and `MODE`. The first parameter defines the targeted hardware component and the desired language. Possible values for defining the hardware are `CPU` and `GPU`. Further options to specify the output are `SEQ` for sequential programs, and `CUDA` for creating CUDA programs. Otherwise, OpenACC programs are generated for the GPU, and OpenCL is used for the CPU. In case multiple options are listed, multiple programs are created; the combination of CPUs and GPUs components is not implemented. Three further configuration variables are used to specify the number of available computational units and their structure. For the `PROCESSES` variable a numerical value is expected determining how many computational nodes are available. The variable `CORES` defines the number of CPU processors, and `GPUS` the number of GPUs available per node. This means that the example program shown in Listing 2 creates two programs: one program for a four-node cluster with two processors (so eight processors in total) containing parallel CPU code, and one program for 4 nodes with 2 GPUs each. Last but not least, two modes can be set in the configuration block: release and debug, which most importantly influence the flags passed to the compiler in the build script. Thereafter, other musket files can be included.

```
#config PLATFORM GPU CUDA
#config PROCESSES 4
#config CORES 2
#config GPUS 2
#config MODE release
```

Listing 2 Example Program Musket: Configuration Block

Data Structures

For building a program, variables and data structures are essential. In this context data structures refer to collections of variables. The data structure block serves to define variables of primitive types and collections of variables. Primitive types supported by Musket are integer, double, float, and boolean. Collection types supported in Musket are matrices, arrays, and structs. Arrays and matrices can contain primitive types or structs. Structs can contain any nested structure of primitive and complex types. Arrays and matrices require to define the type (primitive or struct), the size, and the distribution mode within their declaration. It is assumed that the size is a multiple of the number of cores, processes, or GPUs used. This restriction was made to ease the distribution of data. The distribution mode determines where and how data structures are spread across different computational units. Available distribution modes are: `copy`, `dist`, `rowDist`, `columnDist`, and `loc`. Either the complete data structure is copied to all units (`copy`), or parts of the data structure are copied to each unit. In the example in Listing 3, an integer array of 16 elements is distributed with 4 elements on each of the 4 nodes. For matrices it can be distinguished between splitting the data structures into rows, columns, or, as default, submatrices are distributed. Last, `loc` is a specific case of `dist` where no global copy of the data structure is maintained. The array is filled with the integer one at all indices.

```
array<int,16,dist> a = {1};
```

Listing 3 Example Program Musket: Data Structures

User Functions

The syntax of Musket functions close to the C++ syntax. User functions must start with a return type which is not void. It can be any of the introduced data types or `auto`, which automatically evaluates which type is used. Afterwards, the name of the function and the parameters are listed. Parameters are typeset to their data type. In the example in Listing 4 one functions are listed: `init` and `add_index`. Both return an integer, but the usage of the main function will show how they can be differently used within skeleton calls.

```
int init(int i){
  return 42;
}
int add_index(int i, int j){
  return i + j;
}
```

Listing 4 Example Program Musket: User Functions

Main Function

The main function is the entry point for the program. In Listing 5, an exemplary program is depicted, which is build on the previous examples. The first instruction is a specific function call from the Musket language, belonging to the type *MusketFunctions*. *MusketFunctions* wrap commonly needed functions for writing parallel programs which do not need to be executed in parallel. This includes timers for CPUs and GPUs to measure the runtime, maximum and minimum values for primitive types, a random number generator, and a print function. In the aforementioned example, a timer is started to measure the runtime of the skeletons. As shown in line 11 variables do not necessarily need to be defined in the data structures block, but can be defined later. However, firstly initialising the data types improves the readability and clarity of the program.

In the first exemplary skeleton usage in line 12, all elements of the array `a` are filled with the integer 42. In Musket, skeleton calls start with the data structure on which the skeleton should be called and continue with the function and the corresponding parameters. Depending on the skeleton, the function has more parameters than passed by the function call. This can be seen when the two skeleton calls are compared. Since the call in line 13 also includes the index the function which it is called on has two parameters one for the index one for the current value. In contrast, the `init` function only had one parameter, which is the current value of the element. Noticeably, index variants require different functions for arrays and matrices since matrices require two parameters for index calculation. In the second skeleton call to each value, the index is added. This results in an array containing the numbers 42-58.

```
main{
  mkt::roi_start();
  int i = 42;
  a.mapInPlace(init());
  a.mapIndexInPlace(add_index());
  mkt::roi_end();
}
```

Listing 5 Example Program Musket: Main Function

Musket supports the skeletons Map, Fold, Reduce, Zip, Gather, Scatter, and Shift. All skeletons have distinct individual variants. For example, skeletons can be combined such as *MapReduce*, and index-variants of skeletons allow to include the individual index in the function as seen previously with the Map skeleton. Since the implementation of the PFB can be built solely with Map skeletons, the explanation of other skeletons is neglected. The full DSL specification can be found in a code repository [35].

2.4.3 Bulk Synchronous Programming Model

Another approach to simplify creating parallel programs is the BSP model. Originally, this model was developed by Leslie G. Valiant around 1984. The model consist of three parts:

(1) a number of processors with a local memory referenced as one component (*p*),
(2) a router which can send messages between the components,
(3) and the possibility to synchronise all or a subset of components [9].

As additional parameters L refers to the minimal time between two successive synchronisation operations and g the total throughput of the system (the number of operations) to the throughput of the router (number of word of information delivered).

A program is structured in so-called supersteps. One superstep can be executed by a component using data available in its own memory. After one superstep all or the defined processors are synchronised. Depending on the context, it is allowed to mix computations and communication within one superstep. The synchronizer of the system checks every L time units whether all processors finished one superstep. Within this model, two modes of programming were suggested: First an automatic mode, where memory operations are hidden from the user and second, a direct mode where the programmer retains control over the memory allocation.

On the one hand, the abstract design of this approach increases the transferability to different hardware architectures and programming languages. On the other hand, the definition impedes to implement communication-based parallelism. For example, to pipeline operations, it must be allowed to start executing operations before a superstep is finished.

2.4.4 Open Accelerator

Another approach to simplify parallel programming are directives. Open accelerators (OpenACC) is to the best of the authors knowledge the most prominent programming standard for writing parallel GPU code with directives. It can be used in combination with in C, C++, and Fortran programs. Most importantly, it uses directives to parallelize loops and optimise data locality.

Parallelize Loops

To parallelize loops it is differentiated between the directives `#pragma acc parallel loop` and `#pragma acc kernels`. The kernel variant is used in case the compiler should check whether it is efficient to parallelize the loop. Additionally, in this variant the compiler checks that the result is correct, and does not depend on other executions of the loop. In some cases the compiler might not be able to check whether it is save to parallelize the loop. When using the parallel pragma it is assumed that the programmer knows that the for-loop can be efficiently parallelized. To make the parallelization of the loop more efficient additional information can be given to the pragma. For example, the keyword `private(var)` indicates that a local copy of the variable is required for every executing thread, or `collapse` can be used to subsume nested loops into one loop.

Data Locality

Unnecessary data copies have a major impact on the runtime of the program. Hence, OpenACC provides a directive `#pragma acc data` which warp a region where copies of data are not necessary although multiple parallel functions are executed. This directive can be specified by multiple options. The `Copy` option allocates memory space on the device and transfers the data to the device and back to the host. The option `Copyin` is equal to `copy` beside that the data is not copied at the end of the execution from the device to the host. In contrast, `Copyout` does not initialize the space created but copies the result back to the host. For all operations, the size of the variables copied can be explicitly defined or the identifier of the variable can be passed as an argument to the specifications. Moreover, there exist `update` directives to synchronize data.

The listed features are only a subset of features available for OpenACC [19]. OpenACC simplifies parallel programming but concurrently targets programmers with different levels of expertise.

3 Methodology

The previous chapters identified a research problem and provided the background knowledge to reason how the problem can be solved. In this chapter, the solution is systematically structured and circumvented from previous work. Moreover, the requirements for the practical application context of processing telescope data are collected from the MPIfR.

3.1 Systematic Approach

Every problem to solve in research requires a systematic approach. A variety of different methodologies have been developed which have distinct focal points. For the context of information systems, design science (DS) oriented methods are often used. Design science "creates and evaluates IT artifacts intended to solve identified organizational problems." [13, p.77]. Approaches within the DS-field enable to structure the research around the design of a system or artefact. The goal of this research is to provide a comparison between a high-level and a low-level program in the context of a practical application. Since the programs for the different approaches are developed as part of this thesis, the design of an artefact is considered an important aspect. For this purpose, the design science research methodology (DSRM) is used to structure this work. This method was developed by PFEFFER ET AL. with the purpose to structure the research around the design of artefacts within a system. The methodology specifies principles, practices, and procedures to assure the systematic solution of the problem

Practices and Principles

The most important practical rules for DS research were taken from HEVNER ET AL. [13]. The first requirement is that artefacts must be created during the research. Those artefacts should contribute to the solution of the defined business problem. The stated problem has to be meaningful for practice. The "utility, quality, and efficacy" [13, p. 85] of the business problem must be proven during the research. The solution is designed with the help of existing theories and knowledge. In this concise, sophisticated process, an efficient and suitable solution to the problem is the major objective. Those principles are put into action with the help of a structured process. This thesis is structured according to the process of the DSRM shown in Figure 8.

The first step *Identify Problem and Motivate* is realized in Chapter 1 and 2. Concretely, it is motivated why fast programs are desirable, and why they are necessary in the context of processing telescope data. Moreover, the chapters highlight why parallel programming is relevant for creating fast programs and provides an efficient and powerful tool for developing high-performant programs for GPUs. Current approaches to high- and low-level GPGPU programming are explained, and the need for a practical application for creating a meaningful comparison is reasoned. Moreover, the practical application context proves that the identified problem is relevant for practice.

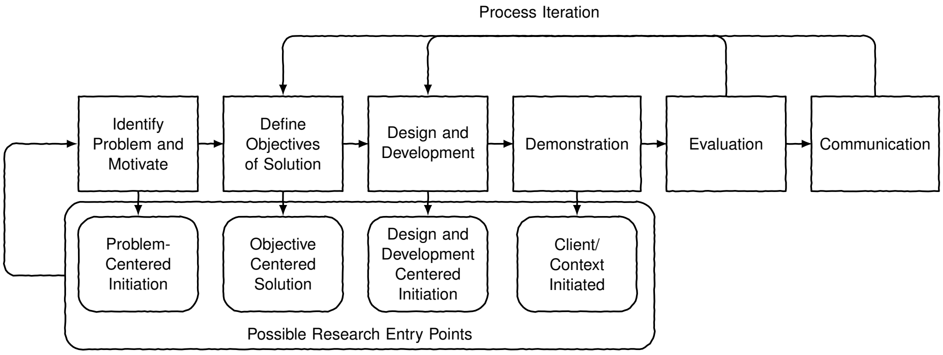
Process Iteration
Identify Problem and Motivate
Define Objectives of Solution
Design and Development
Demonstration
Evaluation
Communication
Problem-Centered Initiation
Objective Centered Solution
Design and Development Centered Initiation
Client/ Context Initiated
Possible Research Entry Points

Figure 8 Process of DSRM [cf. 24]

The second step, *Define Objectives of Solution*, is also realized in this chapter. It includes the definition of the research goal and the formulation of the research questions. Most importantly, the dimensions for comparison of the two approaches are prioritized and specified, which are the performance, maintenance, usability, and transferability. Moreover, the requirements for the programs to be applicable in practice are derived from consulting the MPIfR in this chapter.

In the *Design and Development* step, the implementation of a high- and a low-level program is discussed. This includes the explanation of the characteristics of the program. Moreover, the programs were iteratively redesigned to improve performance. In the DSRM this is called a process iteration; from the *Evaluation* step and the measured run times the artefacts are redesigned. The redesign of the solution is explained as part of the Prototype Design chapter.

The demonstration of the solution targets two platforms: The programs are run on the HPC Taurus of the University of Dresden, and they are executed on the computer of the MPIfR. The HPC is used to measure the performance of the programs. The usage of the program at the MPIfR proves the practical relevance and applicability.

The evaluation of the designed artefacts has an explicit focus on the performance of the programs. Only a program which enables fast processing of the data is meaningful for practice. Additionally, the effort to create the program, the transferability, and the maintenance are discussed. All results are included in Chapter 5. Last, the results of this research are documented and communicated with this thesis. In addition an open-source repository containing the low-level program[9] and the high-level program[10] is available.

3.2 Related Work

Designing solutions for PFBs have been previously topic for GPU programming. Adámek et al. have developed a high performant solution for calculating a PFB. Their work considered CPUs, GPUs, and a Xeon Phi as possible hardware architectures. The GPGPU solution considered the Fermi, Kepler, and Maxwell architecture. With their implementation, they performed better than previous implementations [34, 4]. CUDA is used as the parallel framework for the implementation of the GPU solution.

Multiple publication have discussed the implementation of filters with GPGPU high-level frameworks as examples for challenging task [cf. 7, 18, 3]. The problems discussed mostly refer to image processing. However, image processing and signal processing have many commonalities. In most cases, the current element is changed on the basis of surrounding elements. This entails the same challenges as in signal processing: to maintaining consistent data when elements are changed, deciding on a distribution of the data to different memory spaces, and the efficient calculation of parameters. Also, the implementation of the FFT algorithm has been discussed for high-level frameworks [14, 10].

To the best of the authors' knowledge, the research lacks a comparison of high- and low-level programs based on a real-world application example. The performance of high-level approaches in comparison to low-level approaches is measured based on commonly known benchmarks such as matrix multiplication or the Gaussian elimination [14]. Taking an application context from reality with specified requirements, enriches those findings, by possibly revealing missing functionalities and proving the applicability of high-level approaches. Moreover, it can precisely evaluate whether the possible loss in performance is negligible.

3.3 Research Objectives

The research conducted is structured into research questions which highlight different perspectives on the research topic. Generally, all questions serve to compare high- and low-level approaches. *RQ1* highlights the applicability of the approaches by evaluating

---

[9] https://github.com/viadee/cudapfb *Last accessed* 15.09.2019

[10] https://github.com/NinaHerrmann/musket_out *Last accessed* 15.09.2019

which implementation satisfied the requirements from the practical application context. *RQ2* targets to conduct a more in-depth analysis to define under which circumstances the high- or the low-level implementation is preferable. *RQ3* analyses the created artefact for possible improvements which are feasible by exploiting the functionalities the framework provides. Last, *RQ4* discusses how the high-level framework could be adapted. The low-level framework is excluded from this question since the code of the framework is not publicly available. Hence, no meaningful improvements can be suggested.

RQ1:	Which approach can satisfy the requirements from the practical application context?
RQ2:	What are the advantages and disadvantages of high- and low-level implementation?
RQ3:	What possible improvements can be made to the given implementations?
RQ4:	What improvements can be made to the high-level framework?

3.4 Requirements Analysis

One key condition to evaluate the applicability of the different approaches is to apply them to an existing and relevant application context. For this purpose, the requirements for a GPGPU program are collected from the MPIfR located in Bonn. The MPIfR has multiple telescopes, located in different countries, collecting data about radio frequencies from astronomical objects and bodies in the universe. Examples for such object are comets, moons, and planets. The raw data which is received by the telescopes contains signals from multiple sources and might be polluted by other sources. For this purpose, the data is processed with a PFB to restructure and clean the datasets. Since the amount of data is tremendous, those computations get time-intensive and should be executed as fast as possible. Moreover, it is desirable to process the data in real-time since new data is received every second. If the processing lasts longer than the time by which the data is recorded, sequences of data have to be chosen to be evaluated.

From this context, the following requirements were derived. First, the program needs to be capable of processing 2 billion samples per second to process data in a streaming manner. One sample is a floating-point number, and the incoming data is batched in sizes of 250 million samples, which means that batches of 1 GB need to be handled every eighth second. As input for the data and for the coefficients of the filter floating-point values can be expected. During the consultation with the MPIfR, it was decided that the random value generator of the C standard library is sufficient to produce test data. It is assumed that input data, as well as coefficient data, does not exceed the float data type.

Another important aspect is the number of channels and taps for the PFB. The implementation should flexibly adapt to different settings. For the MPIfR, the common settings are 16, 64, 2048, and 32768 channels with 32, 16, 8, and 4 taps. The implementation does not need to consider the Cartesian product of those settings, but the settings are combined in the sequence stated. This is a piece of important information since dependent on the number of channels and taps distinct implementations might be more efficient. Hence, during the development of the implementation, the differences in the calculations and their effect on the overall performance should be considered.

Last, the interfaces for the input and output need to be defined. The MPIfR runs software which transfers the data to the GPU memory and converts it from double precision to 32-bit floating-point values. Therefore, one requirement was that the program should process data which already is on the GPU. Consequently, it was decided to have a class which implements a method which takes a GPU pointer for the input and a GPU pointer for writing the output. The overall PFB process is depicted in Figure 9. The floating-point conversion is not part of the implemented program but conducted previously. Conclusively, the requirements for a program which is put into practice at the MPIfR are:

(1) process batches of 1 GB each eighth second,
(2) adapt taps and channel sizes,
(3) implement GPU-pointer for input and output.

Another important aspect which was deduced from the consultation of the MPIfR is that rounding errors can be omitted. Precisely, it was stated that changes in the last 2-3 decimal places can be omitted. To assure the correctness of the implementations, the results are compared to the results of a Python implementation of the PFB which was given as a reference implementation from the MPIfR. Although the Python script includes rounding errors, it is assumed that if the two implementations produce similar numbers neglecting the last 2 decimal places, the implementation is sufficiently precise for the application context.

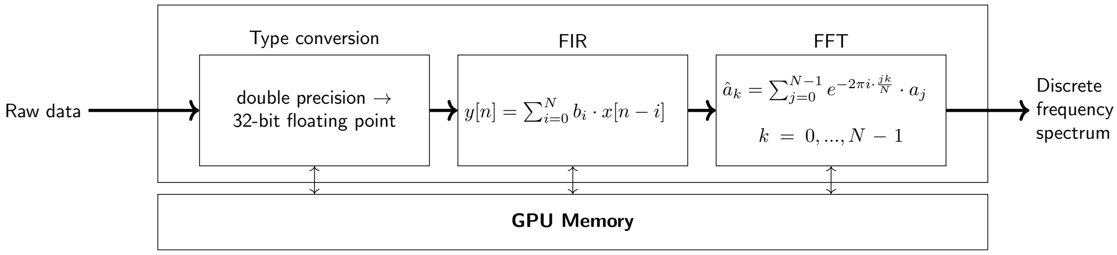
Type conversion
FIR
FFT
Raw data
double precision → 32-bit floating point
$y[n] = \sum_{i=0}^{N} b_i \cdot x[n-i]$
$\hat{a}_k = \sum_{j=0}^{N-1} e^{-2\pi i \cdot \frac{jk}{N}} \cdot a_j$
$k = 0, ..., N-1$
Discrete frequency spectrum
GPU Memory

Figure 9 Exemplary Process of PFB

4 Prototype Design

PFBs include multiple steps. This implementation considers one specific, exemplary sequence of steps, namely a FIR-filter and a DFT. The two steps discussed here have different run times. The FIR-filter is in the asymptotic complexity of $O(m \times n)$ where m is the number of taps and n the number of elements to process. The FFT algorithm for the DFT reduces the runtime from $O(n^2)$ to $O(n \times \log(n))$. For the PFB this results in a complexity of $O(n \times \log(j))$ with n begin the number of elements and j the number of channels. Usually, problems solved on the GPU are in higher runtime classes such as matrix multiplication which is in the complexity class $O(n^3)$. However, with a huge amount of data, it can be efficient to parallelize problems with smaller run time classes. This is the case for telescopes in radio astronomy. For the given settings of taps and channels of the PFB, both problems are comparably complex since they are in the same complexity class and $log(j)$ is roughly the same size as m. First, the low-level implementation is explained and discussed. Afterwards, the high-level implementation is elucidated.

4.1 Low-Level Implementation

The description of design decisions taken during the low-level implementation is split into three parts: the FIR-filter, the DFT, and memory management. Although all problems are important, it is shown that due to the support of libraries, the effort required varies.

4.1.1 Finite Impulse Response Filter

For the low-level implementation, multiple approaches exist to calculate the FIR-filter. Hence, three distinct variants are presented, and their advantages and disadvantages are highlighted.

Global Memory Approach

For calculating a single element of the output of the FIR-filter the current element and $m - 1$ previous elements are multiplied with the corresponding filter coefficients and the sum of all multiplications is taken as the new element (cf. Section 2.1.1). The most naive implementation of this approach is to let each thread calculate a single element. The corresponding kernel function is displayed in Listing 6. In all listings concerning the low-level implementation, CUDA keywords, data types, and functions are highlighted in yellow. Features of the C language are highlighted in blue and pink. In the depicted intuitive approach, all elements are accessed from global memory. They are passed as float arrays to the kernel. In line 2, the index of the executing thread is calculated from the block and thread indices. In line 4-7, all intermediate results are calculated and accumulated. The result is written to a distinct array since other threads might still need to access the old value of the element. This is inevitable since the order in which threads are started can not be scheduled, and therefore it is unknown at what point in time which thread requires which element. Thus, the calculations can not be executed in place.

With this approach, in total as many threads as elements are started with each thread processing one element. Since the maximum number of threads per block is limited, the number of elements is divided by the maximum number of threads per block and rounded for getting the number of blocks started. This intuitive implementation has characteristics which might induce a higher runtime. First, multiple threads might access the same memory space, due to the same indices used from the input data and the coefficients. Moreover, accessing global memory is slow [23, 9. Memory Optimization].

```
1 __global__ void FIR(float * d_data, float* d_output, float * d_coeff,
     int channels, int taps) {
2   int idx = blockIdx.x * blockDim.x + threadIdx.x;
3   float ftemp = 0.0f;
4   for (int j = 0; j < taps; j++) {
5     int channel_offset = j * channels;
6     ftemp += s_data[channel_offset+idx] * s_coeff[channel_offset+idx %
     (taps * channels)];
7   }
8   d_spectra[idx] = ftemp;
9 }
```

Listing 6 CUDA FIR-Filter Implementation: Global Memory

Shared Memory Approach

One of the major bottlenecks of the naive implementation are the repeated accesses to global memory. Hence, the usage of faster memory, i.e. shared memory, should be considered. Shared memory is available on-chip and roughly the latency drops by a factor of 100 [12]. Moreover, shared memory achieves a bandwidth of 64 bits every clock cycle on devices with a compute capability higher than 3.x [23, 9.2.2 Shared Memory]. Other solutions for problems which use shared memory, for example, matrix multiplication, store the input, which can be accessed by other threads. During the computation of the FIR-filter the coefficients and the input numbers are accessed multiple times by distinct threads. For this purpose, the computation might be accelerated by writing the input and the coefficients to shared memory.

The size of the shared memory depends on the architecture of the GPU. One block can allocate 48-96 KB of shared memory and one SM is restricted to 48-112 KB of shared memory, depending on the architecture [cf. 20, H.1 Features and Technical Specifications]. In Listing 7, an exemplary usage of shared memory is shown. Since the maximal available size depends on the GPU it is abstractly referred to as `DATA_SIZE` and `COEFF_SIZE`; this size is allocated in line 3-4. Afterwards, the input data and the coefficients are copied from global memory to shared memory (l. 5-6). For this purpose, each thread copies the element, which should be processed and one coefficient from global to shared memory. Multiple threads might try to copy the same coefficient since the index is not unique. However, this is just as time-intensive as to insert an if-Statement which prevents the repeated assignment. Afterwards, all threads have to be synchronized to assure that when the computation takes place, all necessary elements are available in the shared memory. The implementation of the multiplication steps is neglected since it does not vary from the previous implementation in Listing 6 besides that the data is accessed from the shared memory array.

```
1 __global__ void FIR_SM(float * d_data, float* d_output, float * d_coeff
    , int channels, int taps) {
2 int idx = blockIdx.x * blockDim.x + threadIdx.x;
3   __shared__ float s_data[DATA_SIZE];
4   __shared__ float s_coeff[COEFF_SIZE];
5   s_data[idx] = __ldg(&d_data[idx]);
6   s_coeff[idx%(taps * channels)] = __ldg(&d_coeff[idx%(taps * channels)
     ]);
7   __syncthreads();
8   // Calculate Filter
9 }
```

Listing 7 CUDA FIR-Filter Implementation: Shared Memory

However, this implementation only works for a small number of taps and channels. With an increasing number of taps and channels hence a growing number of coefficients and elements per spectra, not all data required can be loaded into the shared memory due to its restricted size. For example, if a device has 48 KB of shared memory, approximately 12,000 floats can be stored in the shared memory. In the case of 2048 channels and 8 taps merely the coefficient size requires 16,384 floats, which already exceeds the SM-size. Therefore, only the first 750 elements could be processed, since the coefficients and the remaining spectra occupy the rest of the memory. This processing would additionally require to adjust the indexing to the taps and channels. Since PFBs usually process up to 32,768 channels, this approach needs to be restructured.

Restructure Indexing

Instead of processing data in time-logical order, the reuse of data can be optimised by loading data of the same channel in shared memory and processing other channels in other blocks; i.e. a subset of the channels is processed per block, and therefore more spectra can be loaded into shared memory. This idea has been proposed by Adámek et al. for the optimization of an FIR-filter [1]. This new structure for splitting the input data is shown in Figure 10. Instead of imagining the data as a time logical sequence of numbers, it is ordered in a 2D structure with one row representing one spectrum of data. Thus, the data is still ordered time logically but row by row. Therefore, each column in the figure contains data for one channel. Each colour in the depiction represents one block started within the program. In this example, each block processes 32 channels. This number is critical since choosing a too big or a too small value will slow down performance. On the one hand, loading multiple channels within one block will slow down the performance since a lower number of elements can be loaded into shared memory and therefore, fewer elements can be reused. This diminishes the advantage of loading the elements into shared memory. Consequently, not too many channels should be loaded by one block. On the other hand, it is not efficient to process few channels per block since distinct threads would request the same memory spaces, which would result in bank conflicts. Bank conflicts occur when threads from distinct warps request different addresses in the same memory bank, and their requests are serialized since one bank can only serve one request. It was decided to process 32 channels per block. This number performs especially well since one warp consists of 32 threads. However, it should be kept in mind that when processing 32 channels per block in case of 16 channels halve of all threads idle. In CUDA, a maximum of 1024 threads can be started per block [cf. 20, H.1 Features and Technical Specifications]. Assuming that one thread loads one element and 32 channels and 32 spectra are loaded in one block.

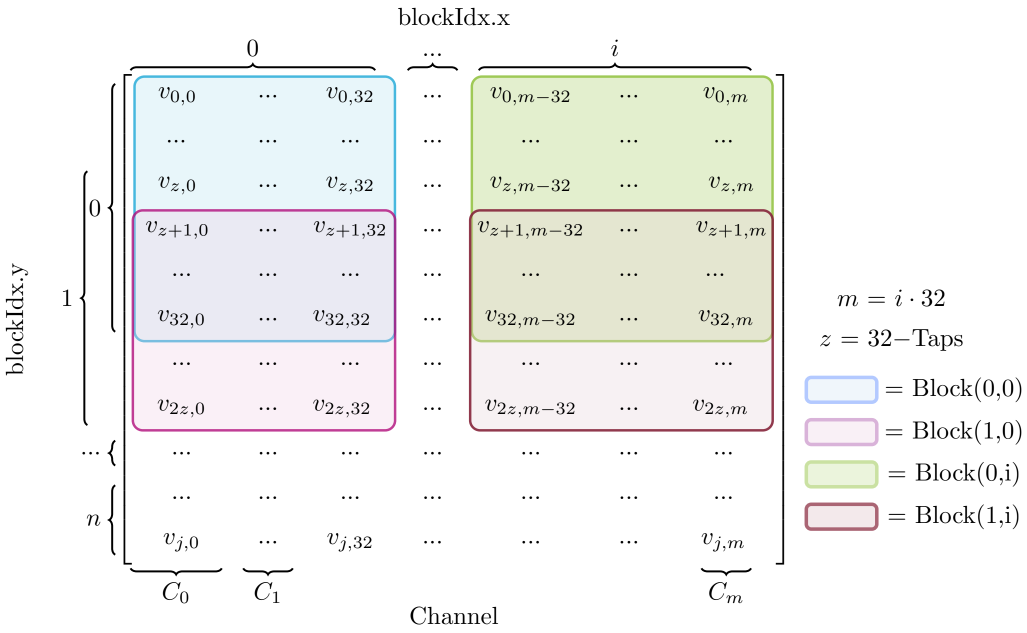

Figure 10 FIR-Filter Data Division to with Fixed Number of Channels per Block

The number of rows, which can be processed per block depends on the number of taps. In this dimension, the data from the blocks is overlapping since the calculation of one element requires taps-1 elements from the past. For this reason, the number of rows which are not only loaded but processed is $z = 32 - taps - 1$. The following rows are then processed by the next block. For the ease of computation, different block dimensions are used to locate the data which needs to be processed. As visible in the figure in the y-dimension exist an overlap. This processing has the advantage that with a growing number of channels the program can still reuse the shared memory since the additional coefficients are part of another block and therefore do not need to be loaded into the limited shared memory. The resulting program is shown in Listing 8.

Due to the restructuring, the index calculation becomes a key aspect of the implementation. First, the required size of the shared memory needs to be allocated in line 2-3. Due to the maximum number of threads for the data, 1024 floats are allocated; each thread loads one element. For the coefficients, $32 \times taps$ elements are allocated since one block always processes 32 columns. Afterwards, the indices to access the global and shared memory are calculated. Each thread has a distinct index within its block (`tx`), a local index within its warp (`localId`), and a warp index (`warpId`) (l. 5-7). Since one warp contains 32 elements, the warp index is the thread index divided by 32 and rounded to the next smaller integer. The parameter `rows` specifies how many rows can be processed per block. This number depends on the number of taps since this number defines the number of successive elements which are required. Since always 32 columns of data are processed per block, and 1024 threads are started in total, 32 rows of data are loaded into shared memory. From those rows the number of rows which cannot be calculated, since their successive values are not available within the block, have to be subtracted, i.e. the number of taps is subtracted. The global index is calculated from those numbers. The column offset is the x-Dimension of the block multiplied with 32. The row offset consist of two parts: the number of channels multiplied with the warp id to calculate the offset within the block and second the y-Dimension of the block multiplied with the number of rows which can be processed per block and the number of channels to calculate the offset of different blocks.

With those indices, the data is loaded from the global array `d_data` to the shared memory. The design of the program assumes that the number of taps does not exceed 32. In this case, not enough shared memory is allocated to load all necessary elements into the shared memory, and no new element could be calculated. This is not a drawback for the application context since none of the listed settings has more than 32 taps. Noticeably, this design decision also results in programs which perform worse on bigger tap sizes since with the fixed number of rows loaded into shared memory, fewer items can be calculated.

For one block the number of columns multiplied with the number of taps elements are required for the array containing the coefficients. Since the coefficients do not change over time, the x-Dimension of the block is not relevant, and only the column offset is used as Listing 7 and the row offset within the block is used. After all data has been loaded, the elements are accessed within a for-loop with their local index, and one new element is calculated for the result (l. 16-19).

```
1  __global__ void FIR(float * d_data, float* d_output, float * d_coeff,
     int channels, int taps) {
2    __shared__ float s_data[1024];
3    __shared__ float s_coeff[taps*32];
4    int tx = threadIdx.x;
5    int warpId = ((int)threadIdx.x / 32);
6    int localId = threadIdx.x - warpId * 32;
7    int rows = 32 - TAPS;
8    int g_id = localId + nChannels * warpId + rows * blockIdx.y *
      nChannels + 32 * blockIdx.x;
9    s_data[tx] = d_data[g_id];
10   if (tx < TAPS * 32) {
11     s_coeff[tx] = ldg(&d_coeff[warpId * nChannels + localId +
      blockIdx.x * 32]);
12   }
13   __syncthreads();
14   if ((tx) < (rows * 32) && g_id < (nChannels * NSPECTRA)) {
15     ftemp = 0.0f;
16     for (int j = 0; j < TAPS; j++) {
17       ftemp += s_coeff[localId + j * 32] * (s_data[tx + j * 32]);
18     }
19     d_spectra[g_id] = ftemp;
20   }
21 }
```

Listing 8 CUDA FIR-Filter Implementation: Data Reuse

Another aspect that is important here is to avoid branch divergence. Generally, branch divergence arises when threads within a warp execute different code flows. For example, one thread has to execute the then part of an if statement and another thread the else part. In this scenario, the threads can not be started concurrently but are started sequentially. For this reason, generally, if statements should be used cautiously in kernels [23, 12. Control Flow]. However, this does not slow down the given program since the number of channels processed equals the number of threads started. Hence one warp always processes one spectrum. For this reason, all threads or no thread enters the if statement and no branch divergence arises.

Generally, this design uses shared memory more efficiently due to the reuse of elements. Moreover, it scales with a growing channel size. Disadvantages of this design are that threads idle during computation since are just used to load elements into shared memory. Moreover, although elements are reused the number of accesses should be increased to compensate for the time-consuming memory transfer from global to shared memory.

Optimise Data Reuse

Using shared memory is efficient when the time saved due to shorter memory request compensates for the time required to load the data to the shared memory. Hence, it might be beneficial to process elements sequentially and therefore reuse the shared memory instead of parallelizing the computations and reload the data multiple times into the shared memory. For this purpose, the previous implementation can be adjusted to let one thread calculate multiple elements reusing data and coefficients which have already been loaded. Intuitively, this dissents the idea of parallelism, since those operations are computed sequentially. However, this can be beneficial in case it decreases bank conflicts and optimises shared memory reuse. Hence, the choice of the number of elements calculated per thread has to balance the advantage of parallelism and the advantage of reusing data. This idea is depicted in Figure 11. The thread highlighted in blue calculates s elements. Noticeably, for calculating the new element also element succeeding the elements highlighted in blue are required. The figure shows which elements are newly calculated, not which elements are read by a thread. Calculating multiple elements is efficient since the same coefficients are reused. Moreover, the overlap between different threads reading elements from the input is smaller.

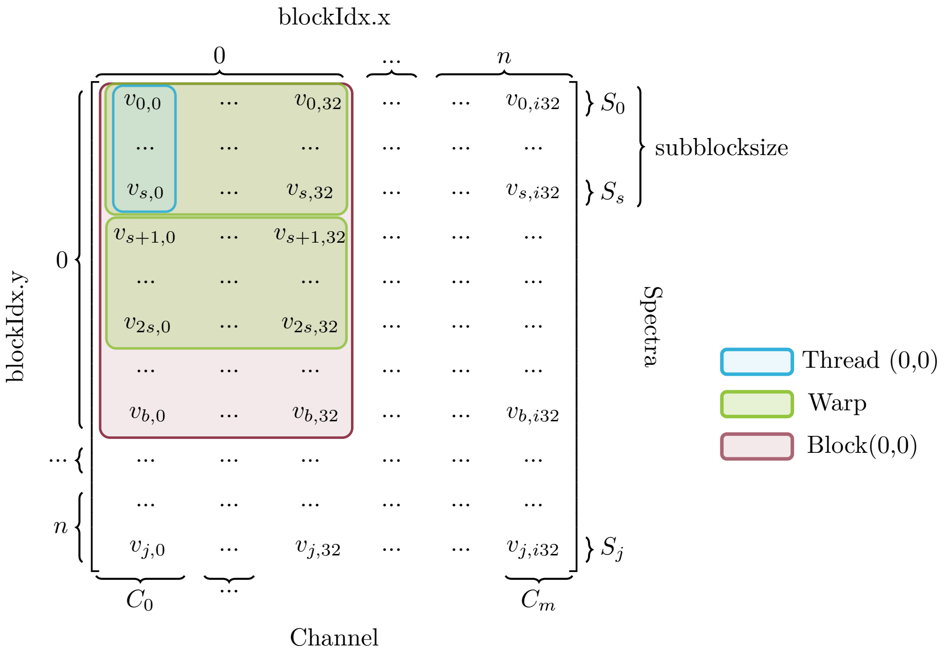

Figure 11 FIR-Filter Data Division Calculating Multiple Elements per Thread

The same approach was used by ADÁMEK ET AL., who found that 3 is an optimal number of elements per thread [1]. In the figure, this number is not fixed but abstractly represented as s. With this approach, threads from different warps still have overlapping memory accesses to the shared memory since for example, the calculation of the new element of $v_{s,0}$ requires the element $v_{s+1,0}$ as well as the calculation of the new element of $v_{s+1,0}$ requires this value. But with calculating multiple elements, the concurrent memory access is demagnified. This design is combined with a new calculation for threads and blocks. The x- and y-dimensions of the blocks are used to represent the channel and spectra processed. Complementary, when 3 elements should be processed, only 672 threads are started per block since for more threads the shared memory would not be big enough in most cases. The exact number of threads which can be started and the size of the shared memory depends on the GPU used. The y-dimension of blocks is the number of channels divided by 32. The x-dimension of blocks is calculated from the overall rows divided by the rows which can be processed by a block. When processing multiple elements per thread, the number of rows that can be processed per block rises. Instead of $32 - taps - 1$ rows per block, now $(s \times 32) - taps - 1$ rows per block can be processed.

```
__global__ void FIR(float * d_data, float* d_output, float * d_coeff,
    int channels, int taps) {
  __shared__ float s_data[DATA_SIZE*SUBBLOCK_SIZE];
  __shared__ float s_coeff[32*taps];
  // tx, warpId and localId calculated as previously
  int warp_offset = warpId * SUBBLOCK_SIZE;
  int rows = (THREADS_PER_BLOCK * SUBBLOCK_SIZE - taps);
  int offset = blockIdx.x * 32 + blockIdx.y * rows * nChannels +
    localId;
  for (int i = 0; i < SUBBLOCK_SIZE; i++) {
    int start_column = warp_offset + i;
    if (start_column < DATA_SIZE / 32) {
      s_mempos = start_column * 32 + localId;
      g_mempos = start_column * nChannels + offset;
      s_data[s_mempos] = (&d_data[g_mempos]);
    }
  }
  itemp = (int)(taps / (THREADS_PER_BLOCK / 32)) + 1;
  for (int f = 0; f < itemp; f++) {
    int start_column = warpId + f * (THREADS_PER_BLOCK / 32);
    if (start_column < taps) {
      s_coeff[start_column*32 + localId] = ldg(&d_coeff[start_column*
    nChannels + blockIdx.x*32 + localId]);
    }
  }
  __syncthreads();
  float ftemp = 0.0f;
  for (int i = 0; i < SUBBLOCK_SIZE; i++) {
    start_column = warp_offset + i;
    if (start_column < rows) {
      s_mempos = start_column * 32 + localId; ftemp = 0.0f;
      for (int j = 0; j < taps; j++) {
        ftemp += s_coeff[j*32 + localId] * (s_data[s_mempos + j * 32]);
      }
      if (start_column * nChannels + offset < CHANNELS * NSPECTRA) {
        d_spectra[start_column * nChannels + offset] = ftemp;
}}}}
```

Listing 9 CUDA FIR-Filter Implementation: Multiple Elements per Thread

The implementation for calculating multiple elements per thread is close to the previous implementation. Changes made to the previous implementation include a different index calculation since more rows can be processed per block. The number of elements to calculate *s* is abstracted with the variable `SUBBLOCK_SIZE`. Moreover, data is loaded and calculated multiple times. Precisely, this can be seen in Listing 9. The amount of shared memory required increases since more input data is needed. The size of the coefficients remains constant since the same number of channels is processed. For the index calculation, a new offset is introduced called `warp_offset` (l. 5) that calculates how many rows have already been processed by previous warps. Precisely, this is the number of rows processed per warp multiplied with the warp index. The parameter `rows` is complemented with a multiplication to include the number of rows processed by one warp. With those indices in line 8-22, the input data and the coefficients are loaded. In each iteration of the for loop, the indices have to be newly calculated. At last, for multiple elements, the new element is calculated. To prevent out of memory accesses l. 27 and 32 check if the address is out of bounds.

4.1.2 Discrete Fourier Transform

As part of the Nvidia Development Toolkit Nvidia provides multiple libraries. One of them is cuFFT, a library to speed up computations for executing DFT with the FFT algorithm. For using the library, it is distinguished between two operations: first, to create a plan for conducting a DFT and second to execute the plan.

The library supports one, two, or three-dimensional input, and the customization of the output format. The creation of the plan uses as arguments the size of the batch to be processed, the overall input size, and the type of transformation which can be real to complex, complex to complex, or complex to real. Unfortunately, the code of the library is not publicly available. Therefore, only assumptions about the operations performed can be made. Most likely, when creating the plan, memory spaces are allocated from those parameters which are linked to speed up computations. Noticeable, at that point in time, the data on which the computation is performed is not needed. For our use case, a one-dimensional plan is created. An exemplary use can be seen in Listing 10. It takes the number of channels and the data size as arguments. Notice, that for this application context not an FFT over the complete data size is required, but for each spectra the FFT is calculated. This means for a data size of 2^{27} elements and $32,768$ channels, 4096 independent FFTs are executed. In the function which executes the plan, a pointer to the input data, a pointer to the output data, and the previously created plan are passed as parameters. The parameter `dft_input` in line 4 is the output pointer from the previous FIR-filter calculation, a float pointer which is typecasted to `cufftReal`.

Since the creation of the plan is independent of the execution as part of the creation of the low-level program, it was decided to implement the plan as a parameter of the class. Within the constructor, the plan is created, making it reusable for subsequent calls. This entails the disadvantage that for a changing number of channels, a new instance of the class has to be created. Since an adoption would also require to change the filter coefficients, this constraint is reasonable. Overall, it can be seen that due to the usage of the library, the implementation effort is considerably lower than the effort required for the FIR-filter.

```
cufftHandle plan;
cufftPlan1d(&plan, nchans, CUFFT_R2C, NSPECTRA);
cufftComplex* dft_output = thrust::raw_pointer_cast(&dft_output[0]);
cufftExecR2C(plan, (cufftReal*)dft_input, (cufftComplex*)dft_output);
```

Listing 10 CUDA DFT Implementation

4.1.3 Memory Management

The specification of the MPIfR did not define a collection type to use for the GPU pointer to memory. Therefore, different options for memory management are evaluated. First, the data type has to be determined. For the input and the filter coefficients float values are passed, while for the output, a complex data type is required. For the complex data type `float2` or `cuFFTComplex` can be used. Since the cuFFT library is used, it was decided to use the cuFFT data type. However, regarding performance, no difference could be notified. Second, it needed to be decided whether a library for collection types should be used. Libraries have the advantage that they provide simple interfaces to common functionalities, for example, determine the size of the collection. This is useful in our case to determine how many spectra should be processed. For this purpose, the thrust library is used to create host and device vectors. This also simplifies data transfer between host and GPU memory and does not show performance differences in this application context. In a multi-GPU environment, this should be re-evaluated to, for example, use asynchronous memory transfers. Moreover, for the specific use case at the MPIfR, thrust vectors are used after the floating-point conversion step. Although from this finding, it can not be generalized that for every PFB thrust vectors are used, this is not a drawback of the implementation since the kernels are written independently from this design decision. As arguments for the implemented class pointers are required, so the program can be easily adapted to changing data types.

4.2 High-Level Implementation

High-Level approaches abstract from low-level operations to simplify the creation of parallel programs. From the different approaches, Musket was chosen. It has to be kept in mind that Musket is an ongoing project. Hence, some of its planned and proposed features are not implemented. The tool is chosen since the outcome of this comparison can be used to incorporate requirements into Muskets design. The concrete implementation of the FIR-filter and the FFT will be discussed, highlighting possible shortcomings and possible enhancements.

4.2.1 Finite Impulse Response Filter

When examining the calculations of the FIR-filter intuitively, the Map skeleton provides the functionalities to solve the problem. Each element is changed and replaced due to its time-logical successive elements. As already argued in the low-level implementation, the execution of the computation can be accelerated by exploiting the knowledge about the elements accessed within the computation. For this purpose, the compiler needs to notice which elements are used. This problem has also been recognized and discussed within the development of multiple other skeleton frameworks such as the LIFT project, SkePU, Muesli, and PASTHA ([11], [7], [14], [16]). Proposed solutions include specification of the map skeleton as *MapStencil* and *MapOverlap*. The implementation for the *MapStencil* skeleton differ but most commonly include a function which specifies which elements from the data structure should be used. For the *MapOverlap* skeleton, data regions are defined which are available on multiple devices.

Currently, the only Map variants available in Musket are Map, MapInPlace, MapIndex, MapLocalIndex, MapIndexInPlace, and MapLocalIndexInPlace. Therefore, the use of the MapStencil or MapOverlap skeleton is neglected. The configuration block is not displayed as part of the implementation in Listing 11 since it is negligible for understanding the implementation of the FIR-filter. Likewise, the assignment of values to the arrays is not displayed. In case the program should be used in practice, it has to be discussed how the data can be read; especially since it is likely that data should be processed in a streaming manner. There would have been multiple ways to implement the FIR-filter with skeletons. In this implementation the Map skeleton is called on a different data structure than the data is read from. As denoted during initialisation of the data structures (l.6-7) they have different sizes. This is necessary since the last $(taps - 1) * channels$ elements have no successive values; hence, no new values can be calculated.

Another option could be to map a data structure in place and skip the last $(taps - 1) * channels$ elements. With this approach, fewer data structures would be required, which improves the overall runtime since no memory space has to be allocated for them. Contrarily, it might slow down the program since an additional if statement would be checked each time the user function is executed, and it would impede the implementation of the DFT. Moreover, the advantage gained from having fewer data structures is not as big as expected since the filter requires to access elements of the same data structure. This means that it has to be assured that the values before the execution of the filter are used and not newly calculated values. For this purpose, it is quite likely that either way a second data structure is built, diminishing the advantage of using the inplace skeleton. The current inplace implementation of Musket is not capable of checking array calls. Therefore, it was not used as an alternative implementation.

The implemented skeleton call is a MapIndexInPlace skeleton (Listing 11). The index is required to access the right elements from the input data structure, and the current value is useful to calculate the first addend. All other addends are calculated in a for-loop in line 18-21. Noticeable, other data structures can be referenced in Musket without being passed as function arguments. This loop iterates for each tap. The chosen output array is of type float2 since it is already known that the output should be processed in the DFT. For computing a DFT a complex data type is required.

```
 6 struct float2{
 7   float x;
 8   float y;
 9 };
10 array<float,1216,dist> input;
11 array<float2,1024,dist> input_double;
12 array<float,256,dist> coeff;
13 
14 float2 FIR(int taps, int channels, int spectra, int Index, float a) {
15   float2 newa;
16   newa.y = 0;
17   newa.x = a * coeff[Index%(taps*channels)];
18   for (int j = 1; j < taps; j++) {
19     newa += input[Index+(j*channels)] * coeff[(Index%(taps*channels))+(
        j*channels)];
20   }
21   return newa;
22 }
23 main {
24   // Initialisation of data structures
25   int ntaps = 4;
26   int nchans = 64;
27   int nspectra = 16;
28   input_double.mapIndexInPlace(FIR(ntaps, nchans, nspectra));
29 }
```

Listing 11 Musket Implementation: FIR-Filter

4.2.2 Discrete Fourier Transform

The implemented solution for the DFT is derived from Quinn [28]. It splits the computation of the DFT into two steps which are implemented with two map skeletons. This includes a fetch step which reorders the elements and a combine step which calculates new entries. The steps are repeated $\log m$ times were m is the number of channels processed. The fetch step implements the butterfly pattern from the FFT algorithm to speed up the computation. Until now Musket does not support some bitwise operations for example, >> and ˆ. To improve the readability of the program it was abstracted from this shortcoming in Listing 12. In the fetch function the elements from the original data structure are shifted. Depending on the iteration of the for loop referenced as `counter` the elements are shifted to implement the butterfly pattern. The combine method calculates the term $e^{\frac{2\pi i}{N}k} = cos(\frac{2\pi}{N}k) + i \cdot sin(\frac{2\pi}{N}k)$ in line 6-13. The if statement checks how many iteration have already been performed and dependent on that first adds or multiplies the values. The complete program can be found in Appendix A.1.

```
 6 const double PI = 3.141592653589793;
 7 struct float2{
 8   float x;
 9   float y;
10 };
11 array<float2,16,dist> c_input;
12 array<float2,16,dist> output;
13 float2 fetch(int counter, int log2size, int i, float2 Ti){
14   return c_output[i ^ (1 << (log2size - 1 - counter))];
15 }
16 float2 combine(int counter, int log2size, double pi, int Problemsize,
       int Index, float2 Ai) {
17   int b = i >> (log2size - counter - 1);
18   int b2 = 0;
19   for(int l = 0;l <= counter;l++) {
20     b2 = (b & 1) ? 2 * b2 + 1 : 2 * b2;
21     b >>= 1;
22   }
23   double v = 2.0 * pi / Problemsize * (b2 << (log2size - counter - 1));
24   float2 value (cos(temp, sin(temp)))
25   if (i & (1 << (log2size - 1 - counter))) {
26     newa = c_input[Index] + value * Ai;
27   } else {
28     newa = Ai + value * c_input[Index];
29   }
30   return newa;
31 }
32 main{
33   // Initialize Data stuctures
34   for (int j=0; j<log2size; j++) {
35     c_input.mapIndexInPlace(fetch(j, log2size));
36     c_output.mapIndexInPlace(combine(j, log2size, PI, 16));
37   }
38 }
```

Listing 12 Musket Implementation: FFT

This approach only works for channel sizes which are a power of two. Moreover, the method could be optimised by pipelining operations. Since the batch size is smaller than the total size of elements to process, the computation for those batches can already continue while other threads might compute the combine part for the last elements.

4.2.3 Implications

From the generated programs, it can be derived that for using Musket in practice some additional features are required to simplify the usage. First, commonly used operators and functions available in C and C libraries might help to simplify programming. In this use case, this included the calculation of the sinus and cosine function and additional binary operators such as bit shifts. Although those operations can be implemented with the supported operators, especially for common mathematical functions which are available in libraries this produces additional implementation effort. This is especially relevant since most programs which are suitable for parallel computing include mathematical operations.

Second, interfaces for the import of data have to be defined. As shown by the application context, the import of the data can not be regarded as a simple task. This included the processing of streaming data and to specify the origin of the data. In our use case, the data resided already on the GPU. For using a program in practice, the origin of the data requires to be defined in some way, which should be considered before using Musket in practice. Otherwise, the import of the data has to be defined manually by subsequently adjusting the generated program.

5 Evaluation

The evaluation of the created programs serves three purposes. First, it should be researched whether or not a GPU program is a preferable approach for solving the given problem compared to a CPU solution. Second, bottlenecks of the GPU programs are identified. This enables the discussion of weaknesses of the programs and possible future enhancements to answer *RQ 3* and *RQ 4*. Last, it should be reasoned which of the two GPGPU approaches is preferable in the given application context, answering *RQ 2*.

The most important aspect of this discussion, the runtime of the programs, was measured on the HPC Taurus of the Technical University of Dresden. First, the settings under which the runtime of the programs are measured are explained. After that, the next section of the chapter compares the runtime of CPU and GPU programs, to prove that the GPU program is preferable. Afterwards, the runtimes of the two GPU programs are discussed independently to identify strengths and weaknesses. Last, the two approaches are compared. The comparison is split into two parts. First, and most importantly, the runtimes of the two approaches are opposed. Second, supplementary aspects are discussed. This includes the expertise required to create the programs, the maintainability of the programs, and the transferability to different GPU-architectures. Those aspects are discussed abstractly since a study would exceed the scope of this thesis. The precise analysis would require questionnaires to test the usability and access to a variety of different GPUs; the sophisticated comparison might be a topic to future work. The chapter is concluded by an interpretation of the results and highlighting limitations.

5.1 Runtime Measurements settings

For the comparison, three groups of programs are started, a CPU program, a GPU program developed with CUDA, and a GPU program designed with Musket. Each program group included programs for different settings; an overview of the settings is listed in Table 1. Those settings differ in two dimensions. First, the number of elements processed varies, which is indicated by the first number of the setting. Second, the number of taps and channels varies specified with the second number of the setting.

The starting point for choosing the size of the input data is given due to the usually processed data-sizes; for a real-time application, the program needs to process 2 billion samples per second. The data is transferred in batches of 250 million floats every eighth second. Therefore, it was decided to have the next biggest power of two (268,435,456) as one data size to test the applicability. In case the runtime requirement of 125 ms is fulfilled for 2^{28} elements it can be assumed that it will be fulfilled for 250 million floats. Complementary, two further data sizes are tested: The next lower power of two (134,217,728) and the middle between the two sizes (201,326,592). All data sizes are multiples of 32 and of all applied channel sizes since this is a requirement for the programs. In the settings, a rising first number means an increase in the number of elements used as input.

For each data size programs with 16, 64, 2048, and 32768 channels are build, which entails different taps settings of 32, 16, 8, and 4 taps. An increasing second number in the settings means that more channels and fewer taps are processed. The program has to adapt to the different tap and channel sizes to be applicable in practice. This distinction is important to judge whether specific settings are more complex than other settings. The number of channels influences the number of iterations in the FFT algorithm, and the number of taps determines the amount of multiplications per element in the FIR-filter calculation. Hence, it is to be expected that settings with a high number of taps and channels are especially computational intensive. It can be seen that with an increasing number of channels, the number of taps is decreasing. Since the two factors do not scale equally, the complexity of the different settings should be discussed.

The number of taps decreases linearly with the different configurations named by the MPIfR: the number halves for each setting. Since the asymptotic complexity of the FIR-filter is in $O(m \times n)$ this means that the number of computations necessary halves in case the number of taps (m) halves. The channel sizes determine the complexity of the FFT algorithm; the column *Number of FFT steps* calculates the complexity given the classification of $O(\log_2(channels) \times n)$. When subsuming the complexity classes this would mean that x.1 settings are in $36 \times n$, x.2 in $22 \times n$, x.3 in $19 \times n$, and x.4 in $19 \times n$ with n being the data size. Hence, x.1 and x.2 are more complex than the other two settings. However, it should also be kept in mind that the calculation of one step in the FFT algorithm is more complex than one step in the FIR-filter. While the filter only computes one multiplication, one step of the FFT includes the calculation of the sinus, cosine, and a multiplication of a complex number.

The runtime of all programs is measured for the previously defined settings on a K80X GPUs on the HPC Taurus of the Technical University of Dresden. Although the K80X has two GPUs, only one is used, since the program is run on one GPU at the MPIfR. The measured runtimes are the average of at least 100 cycles. From each dataset the first runs were removed since they serve to warm-up the GPU and do not represent exact runtimes.

Setting Number	Data size	Channels	Taps	Number of Multiplications for the FIR-filter	Iterations FFT	Number of FFT steps
1.1	134,217,728	16	32	4,294,967,296	4	536,870,912
1.2		64	16	2,147,491,584	6	805,309,344
1.3		2048	8	1,073,741,824	11	1,476,400,464
1.4		32768	4	536,872,896	15	2,013,273,360
2.1	201,326,592	16	32	6,442,450,944	4	805,306,368
2.2		64	16	3,221,225,472	6	1,207,959,552
2.3		2048	8	1,610,612,736	11	221,459,249
2.4		32768	4	805,306,368	15	3,019,898,880
3.1	268,435,456	16	32	8,589,934,592	4	1,073,741,824
3.2		64	16	4,294,967,296	6	1,610,612,736
3.3		2048	8	2,147,483,648	11	2,952,790,016
3.4		32768	4	1,073,741,824	15	4,026,531,840

Table 1 Settings for Runtime Measurements

5.2 Comparison of CPU and GPU Programs

Using a GPU program is only meaningful when it outperforms a comparable program on the CPU. For this context, it was decided to compare the parallel GPU programs to a CPU program written in Python. This decision was taken since it was given from the MPIfR as a reference implementation. The logical consequence is that without a GPU program, the Python script would have been used to solve the given problem. The runtimes for the CPU program were also measured on the Taurus HPC with an Intel(R) Xeon(R) CPU E5-2680 v3 (12 cores) with 2.50GHz and multi-threading was disabled. The comparison of the runtime is shown in Figure 12.

In every setting, both GPU programs are considerably faster than the CPU programs. The precise numbers are shown in Table 2. The low-level program is 17-96 times faster, depending on the setting. The high-level program achieves a speedup with a factor of 5 to 37. Settings with a high number of multiplications in the FIR-filter (x.1) achieve the best speedup.

An explanation for the speedup gains is that the CPU programs require increasingly time for the FIR-filter. From settings with 32 multiplications per element (x.1) to settings with 4 multiplications per element (x.4) the time required by the CPU program decreases by a factor of 13-11 depending on the size for the data. Moreover, in settings where the DFT is more complex (x.3, x.4), the FIR-filter requires approximately half of the overall time. In all settings with only 4 multiplications per element and 15 iterations of the FFT, the filter requires 45 % of the time. This hints that the filter is implemented inefficient since problems in $O(4 \times n)$ should be faster than problems in $O(15 \times n)$.

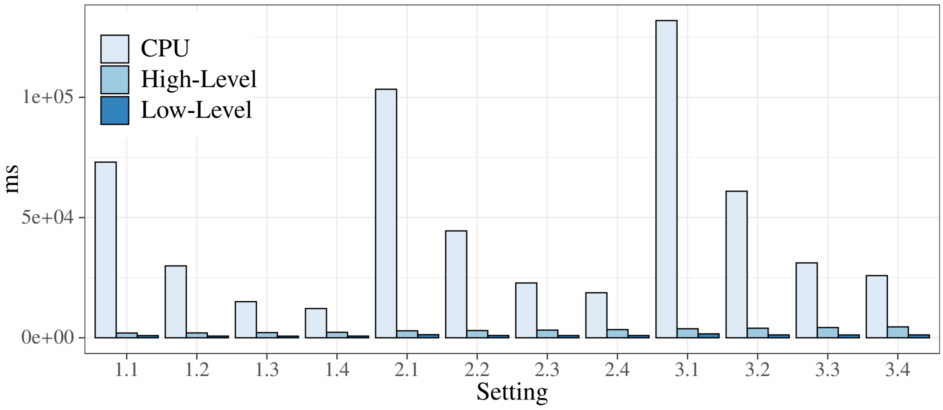

Figure 12 CPU and GPU Programs: Complete Runtime

The time required for the FFT varies less across the distinct taps and channels settings within the same data size. With a rising number of channels, the number of iteration increases and therefore the computations take longer. From settings with 4 iterations (x.1) to settings with 15 iterations (x.4) the runtime increases by a factor of 2-2.4, although the number of calculations necessary increases by a factor of ~3.75. This is caused by the usage of the NumPy library to execute the FFT. This library is optimised and also includes parallel code.

Setting	CPU					GPU (ms)		Speed-up	
	FIR (ms)	FIR (%)	FFT (ms)	FFT (%)	sum (ms)	Low-level	High-level	LL	HL
1.1	69742.93	95.6	3326.456	4.4	73069.3859	779.3884	1981.8408	93.75	36.87
1.2	23993.7058	80.2	5906.6973	19.8	29900.40	709.68	2014.84	42.13	14.84
1.3	7988.3107	53.2	7032.6508	46,8	15020.96	681.00	2147.32	22.05	6.99
1.4	5497.1093	45.2	6663.3926	54.8	12160.50	708.68	2273.31	17.15	5.35
2.1	99205.66	96	4165.30	4	103370.91	1093.12	2903.29	94.57	35.6
2.2	35593.1005	80.1	8857.1285	19.9	44450.23	951.95	3009.19	46.69	14.77
2.3	12082.1071	53	10714.0694	47	22796.18	922.71	3194.96	24.70	7.14
2.4	8469.9701	45.2	10256.1142	54.8	18726.08	940.66	3403.59	19.90	5.5
3.1	125998.02	95.5	5943.96	4.5	131941.984	1373.51	3758.07	96.06	35.11
3.2	48565.7706	79.7	12393.5265	20.3	60959.3	1191.65	3977.41	51.15	15.33
3.3	16406.8383	52.7	14736.5951	47.3	31143.43	1170.74	4251.42	26.60	7.33
3.4	11736.9867	45.4	14129.8411	54.6	25866.83	1196.57	4525.25	21.61	5.72

Table 2 CPU and GPU Programs: Complete Runtime

On the one hand, it could be argued that the comparison to a partly parallelized program is not meaningful since the speedup to a completely sequential program is not measured. On the other hand, the comparison with the presented program reasons which program should be used in practice. In practice, the DFT would not have been implemented, but the library would have been used. Hence, the optimised library is kept for the CPU program. For examining the exact techniques used the code is publicly available [11].

5.3 Intra Program Comparison

The individual analysis of the programs serves to identify which task are especially time-intensive within one program and to evaluate how flexible the programs performs under different settings. The settings influence the taps and channel size as well as the data size. From those findings, it can be discussed for every single program which aspects are in need to be optimised.

5.3.1 Low-Level Program Analysis

The runtime of the low-level program is split into five parts: The time required to allocate the GPU memory spaces, the creation of the FFT plan, the execution of the FFT, the execution of the FIR-filter, and the time required for data transfers between GPU and CPU. In Figure 13 all five categories are depicted.

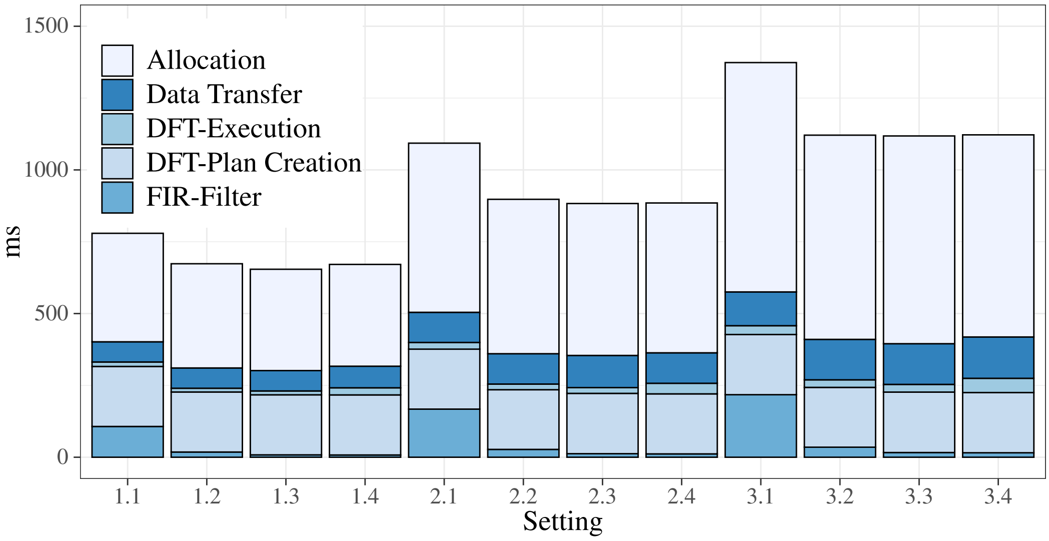

Figure 13 Low-Level Program: Grouped Runtime

Strikingly, the allocation of memory space requires the most time. Precisely, the times and the percentual shares are listed in Table 3. The allocation requires approximately 40-53% of the overall runtime. With a growing data size, the share rises. This proves that for growing data size, the allocation of memory is more time consuming than the calculations performed. The second most time-consuming operation is, in most cases, the

---

[11] https://github.com/numpy/numpy/tree/master/numpy/fft *Accessed: 02.09.2019*

creation of the cuFFT plan with 18-31%. The cuFFT documentation states that the plan “uses internal building blocks to optimise the transform for the given configuration and the particular GPU hardware selected“ [21, 2. Using the cuFFT API]. The code is not publicly available to retrace the operations performed. However, with growing data size, the time required for creating the cuFFT plan does not rise. Therefore, the share of the overall execution time decreases. This means that the action performed when creating the plan are mostly independent from the data size. Possibly, event listeners are initialized which enable pipelining operations.

Setting	FIR		DFT		Transfer		Allocation		cuFFT Plan		Total Time
	ms	%	ms	%	ms	%	ms	%	ms	%	ms
1.1	107.09	13.74	15.45	1.98	130.13	16.7	317.85	40.78	208.87	26.8	779.39
1.1	18.16	2.7	13.02	1.93	123.38	18.32	309.94	46.03	208.88	31.02	673.39
1.1	8.25	1.26	13.43	2.05	122.44	18.72	301.07	46.02	209.02	31.95	654.21
1.1	7.83	1.17	24.66	3.67	128.68	19.18	300.79	44.82	209.13	31.16	671.09
1.1	167.45	15.32	23.32	2.13	195.44	17.88	498.2	45.58	208.7	19.09	1093.12
1.1	27.28	3.04	19.64	2.19	190.81	21.26	452.26	50.38	207.7	23.14	897.7
1.1	12.31	1.39	20.08	2.27	196.27	22.23	444.37	50.32	209.98	23.78	883.01
1.1	11.62	1.31	36.84	4.16	182.67	20.64	444.86	50.27	208.91	23.61	884.91
1.1	217.62	15.84	30.62	2.23	256.51	18.68	659.23	48	209.53	15.25	1373.51
1.1	34.96	3.12	26.24	2.34	247.73	22.1	603.81	53.87	208.18	18.57	1120.91
1.1	16.37	1.46	26.75	2.39	261.42	23.38	603.17	53.95	210.38	18.82	1118.09
1.1	15.57	1.39	49.5	4.41	251.34	22.4	595.98	53.12	209.57	18.68	1121.97

Table 3 Low-Level Program: Runtime and Percentual Share of Grouped Operations

After that, the memory transfers require a significant amount of time with a share of 18-22 %. Again with growing data size, the percentage rises, which shows that with a growing data size memory operations are more time-consuming than the computations performed in parallel. The computation of the FIR-filter and the DFT merely requires a minor part of the overall execution time in most cases. An exception are the x.1 settings which have 32 multiplication in the FIR-filter. In those settings, the filter requires approximately 13-15% of the overall runtime. In all other settings, the share of the FIR-filter is smaller. Hence, the FIR-filter should be discussed more precisely.

FIR-Filter Analysis

Independent from the total program runtime it can be evaluated how efficient the parallelization of the computations is. Theoretically, all operations of the FIR-filter could be executed in parallel since they are independent of each other. Hence, the execution time would remain nearly constant independently from the setting. In practice, this is not possible since the number of threads which can be started concurrently in CUDA is limited. The precise number depends on the GPU. For example, for the K80x used on the HPC Taurus 26 SM are available which can start 2048 threads each. Hence, a maximum of

53.248 threads can be started at the same time. Therefore, the execution time increases with growing data size and an increasing number of operations.

In the design phase of the prototype, it was decided to reuse the shared memory to speed up computations. Hence, it should be shown that usage is efficient. For this purpose, three distinct programs were tested: the naive implementation which uses the global memory, a program which loads one floating-point number per thread into the shared memory, and a program which loads three floating-point numbers into the shared memory.

The runtimes can be seen in Table 4 and are graphically shown in Figure 14. Noticeably, the runtimes for settings with 32 multiplications are not included in the graph since they impede the clarity of the graph. The shared memory implementation loading one element requires approximately 582.3-1165.7 ms for a tap size of 32. Including those runtime would impede to see the differences between the other programs in the graph, hence, it is neglected.

Data size	Taps	Global Memory Kernel	One Element Shared Memory Kernel	Three Elements Shared Memory Kernel
134218224	32	67.4595	582.3749	107.0851
134218224	16	37.6329	25.1683	18.1641
134218224	8	22.784	15.1943	8.2546
134218224	4	17.7618	11.9191	7.8312
201,326,592	32	99.9541	881.9506	167.4489
201,326,592	16	56.5526	38.2451	27.2807
201,326,592	8	34.4382	22.0643	12.3132
201,326,592	4	26.7876	17.2396	11.6231
268,435,456	32	135.0443	1165.6761	217.619
268,435,456	16	75.6756	38.2006	34.9607
268,435,456	8	45.7706	28.3577	16.3748
268,435,456	4	35.9665	22.5229	15.5684

Table 4 Low-Level Program: FIR-Filter Runtime (ms)

The runtime for the global memory program doubles in case the data size doubles for all tap settings. It does not halve when the number of taps halves, although the number of operations necessary halves since more operations can be performed in parallel. The runtime of the program starting a kernel which loads one element into the shared memory is faster than the global memory variant for settings with 16, 8, and 4 multiplications. For 32 multiplications it is a lot slower. This is caused by the way the shared memory is used. With an increasing number of taps, fewer rows can be processed within one block since a bigger data overlap has to be loaded to calculate the new elements. For 4 taps 29 rows can be processed, for 8 taps 25 rows, and for 16 taps 16 rows. For 32 taps, only 1 row per block is processed. Therefore, more blocks need to be started, and fewer calculations can be executed in parallel. Moreover, each block requires to load the data again to the shared memory. This makes the kernel for 32 multiplications extremely inefficient.

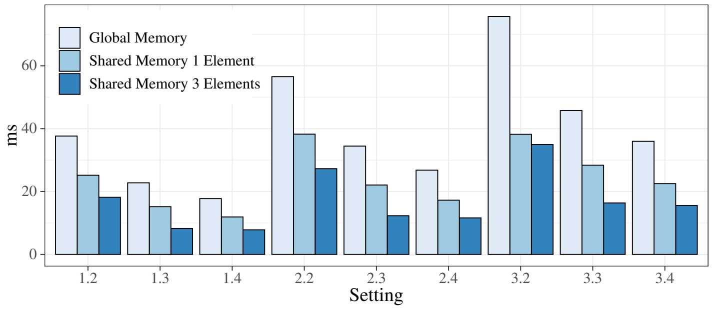

Figure 14 Low-Level Program: Runtime of FIR-Filter

When loading three elements per thread into the shared memory, it can be seen that the difference between 16 and 32 multiplications gets smaller. While for one element, the execution time increased by a factor of approximately 23 for 3 elements, the processing time increases by a factor of nearly 6. In total, the shared memory variant is faster than the global memory variant for setting x.2, x.3, and x.4. For 32 multiplications the kernel using the global memory is more efficient. The number of three elements was proposed by Adámek et al. [1]. However, in their application context they had different tap sizes and did process complex numbers instead of floats. Hence, the number can not be taken as the optimal number. To find the most efficient approach further programs testing different sizes are required. With the discussed example programs it was shown that for distinct tap settings different programs are suitable.

FFT Analysis

For using the cuFFT library, two methods are executed: First, a method to create a plan and second a method to execute the plan. For the tested sizes, the creation of the plan requires far more time than the execution of the DFT, which is shown in Figure 15 and the precise runtimes are listed in Table 5. The time required for generating the plan stays nearly constant (~208-210 ms). This was already visible in the cumulative figure and indicate that the operations executed are not dependent on the data size, but initialize additional components to speed up computations.

It was expected that more channels result in a higher runtime since the number of iterations to execute the FFT increases. They cannot be completely parallelized since the computations of one iteration depend on the results of the previous iteration. In most cases, the calculations of the FFT require more time for more channels. The time is increasing from 64 to 2048 and 32768 channels. However, for 16 channels, the execution of the FFT even takes slightly longer than for 64 and 2048 channels. The precise reason is not known since the library is not publicly available. A possible cause might be that for smaller sizes, it is not efficient to move elements and intermediate results into faster memory spaces. Therefore, the data transfers take longer than the computational time saved.

Data size	Operation	Channels			
		16	64	2048	32768
134218224	Exec	15.45336544	13.0233	13.4263	24.658
	Plan	208.86782	208.8817	209.0196	209.1281
201326592	Exec	23.31726589	19.6409113	20.08209531	36.84016936
	Plan	208.7030079	207.704869	209.9760394	208.9057291
268435456	Exec	30.61984011	26.2361342	26.74617678	49.50419861
	Plan	209.5276387	208.1782369	210.3822991	209.5720171

Table 5 Low-Level Program: FFT Runtime (ms)

Moreover, the performance differences between the different channel sizes are unexpected. The discrepancy between 64 and 2048 channels is minimal while the difference from 2048 to 32768 channels is considerably higher. This is surprising since the difference in the number of iterations ($log(channel)$) is smaller for 2048 and 32768 channels than for 64 and 2048 channels. Since the code of the cuFFT library is not publicly available, only assumptions about the reasons can be made. This could be caused by pipelining parallelization techniques. Since with 32768 channels bigger chunks of data are processed, they can not be pipelined; hence, the execution would require more time. However, it cannot be guaranteed that this is the cause for the increased execution time.

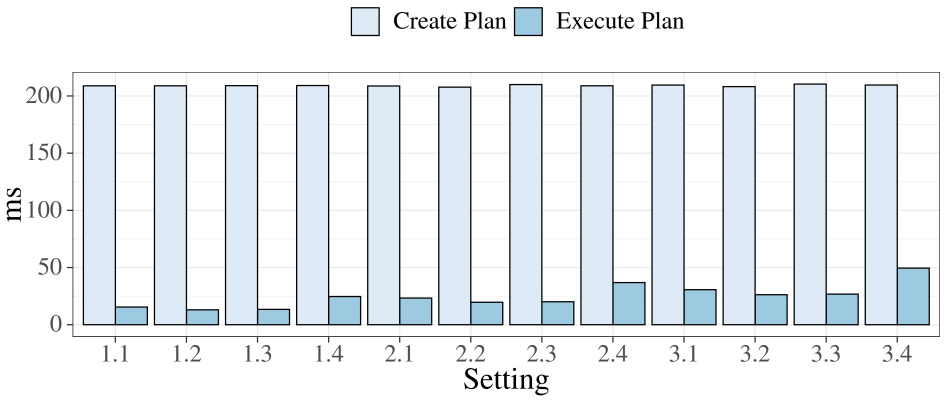

Figure 15 Low-Level Program: Runtime of FFT

Step Comparison

Figure 16 abstracts from the data transfer and the allocation of memory space and focuses on the computational operations of the program. Since the FIR-filter and the DFT are in similar asymptotic complexity classes, it is interesting to compare which one is faster. However, this is troublesome for the FFT, since the library consists out of two steps. The creation of the plan is necessary to calculate the DFT with the cuFFT library. However, the plan itself does not require any input data. Hence no computations are executed at that point in time. Therefore, the figure contains three different bars: One for the FIR-filter, one for the execution of the DFT, and one which subsumes the execution and the plan creation.

It can be seen that the FIR-filter requires a lot more time for settings with 32 multiplications (x.1). In settings with 64 channels and 16 taps (x.2), the FIR-filter requires still more time than the actual execution of the DFT while in all other settings the execution of the DFT requires more time than the filter. Noticeably, the runtimes do not precisely scale equally good; hence, the difference between the two executions varies. In all cases, the sum of the plan creation and the DFT execution require more time. Hence, based upon the observed overhead of the plan creation especially for smaller sizes a handwritten implementation is preferable to the library usage.

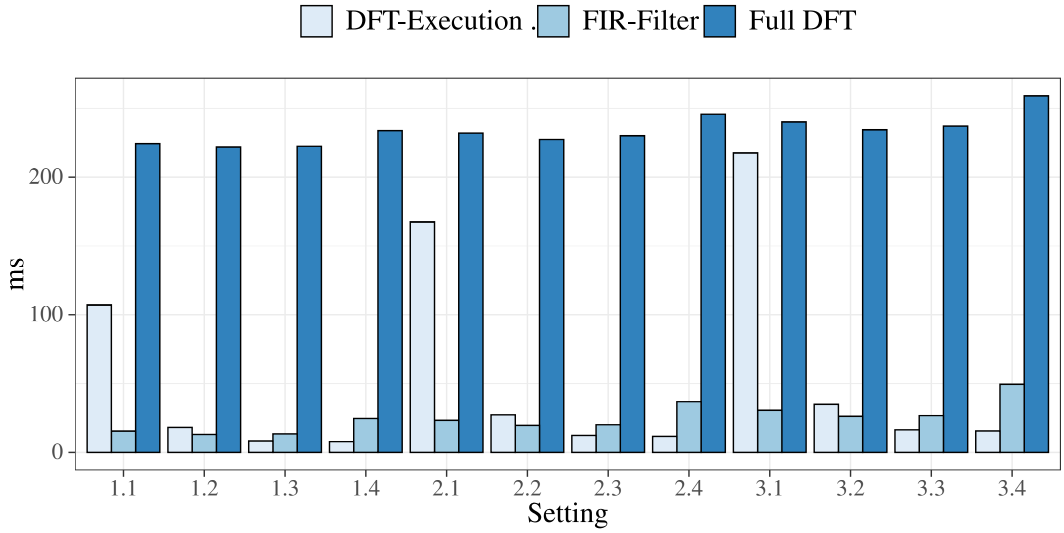

Figure 16 Low-Level Program: Runtime of PFB Steps

5.3.2 High-Level Program Analysis

The categories for measuring the runtime cannot be transferred to the high-level program without adaptions. The reason for this is that the code generator designs the program differently. Therefore, the runtime is grouped into the time required for the DFT, the FIR-filter, the data transfer, and for the array initialisation. For the data management Musket creates `DArrays` which allocate memory of the required size on the CPU and GPU. For this case, the allocation is simple; all GPU sizes are equal to the CPU sizes. In case of multi CPU and GPU environments, the local sizes must be calculated. Those operations are subsumed in the category array initialisation.

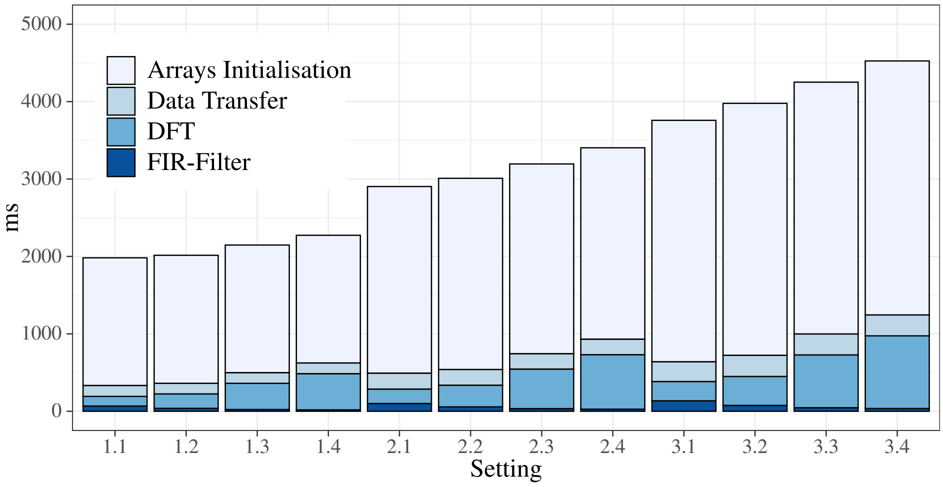

Figure 17 High-Level Program: Grouped Runtime

As depicted in Figure 17 and Table 6 the array initialisation takes the most time with approximately 72 – 83 % of the overall runtime. Although the percentage varies the time required for the allocation is nearly constant for the same input size. The minor variation in the time required within one data size is caused by the varying number of coefficients which are required, and the additional spectra which are required to calculate the last row. Precisely $(Taps - 1) * channels + taps * channels$ floats are required per setting. For the x.1 settings, 31×16 additional floats are required for the input, and $16 * 32 = 512$ floats for the coefficients are required. This number increases with the setting: For x.2 settings $15 \times 64 + 1024 = 1984$ floats are required, for the x.3 setting $7 \times 2048 + 16384 = 30.720$ floats, and for x.4 $3 \times 32768 + 131072 = 229.376$ floats. Due to this variation, the time required to allocate the memory slightly rises with the changing number of taps and channels.

The variation in the percentage also arises due to the increased time required for the DFT with a rising number of channels. This can also be seen in Figure 18. When comparing the time required for the computation, the FFT calculations require more time than the FIR-filter. With an increasing number of channels and with an increasing data size, the time required for the DFT calculations rises. The time required for the computation of the FIR-filter decreases but the difference is negligible compared to the DFT. Since this is considerably longer than the time of the low-level program, possible causes are discussed in the next section.

Setting	FIR		DFT		Allocation		Data Transfer		TotalTime
	ms	%	ms	%	ms	%	ms	%	ms
1.1	67.46	3.4	124.38	6.28	1648.61	83.19	141.4	7.13	1981.84
1.2	37.63	1.87	186.32	9.24	1653.36	81.98	137.52	6.82	2016.7
1.3	22.78	1.06	338.35	15.75	1648.46	76.73	137.73	6.41	2148.38
1.4	17.76	0.78	468.13	20.59	1649.17	72.52	138.24	6.08	2274.09
2.1	99.95	3.44	186.96	6.44	2410.55	83.03	205.83	7.09	2903.29
2.2	56.55	1.88	280.19	9.31	2469.38	82.01	203.08	6.74	3011.07
2.3	34.44	1.08	510.99	15.99	2450.8	76.68	198.74	6.22	3196.04
2.4	26.79	0.79	704	20.68	2471.56	72.6	201.24	5.91	3404.38
3.1	135.04	3.59	249.05	6.63	3118.08	82.97	255.89	6.81	3758.07
3.2	75.68	1.9	373.89	9.4	3254.38	81.78	273.46	6.87	3979.31
3.3	45.77	1.08	681.94	16.04	3252.3	76.48	271.4	6.38	4252.49
3.4	35.97	0.79	938.1	20.73	3280.74	72.49	270.43	5.97	4526.04

Table 6 High-Level Program: Runtime and Percentual Share of Grouped Operations

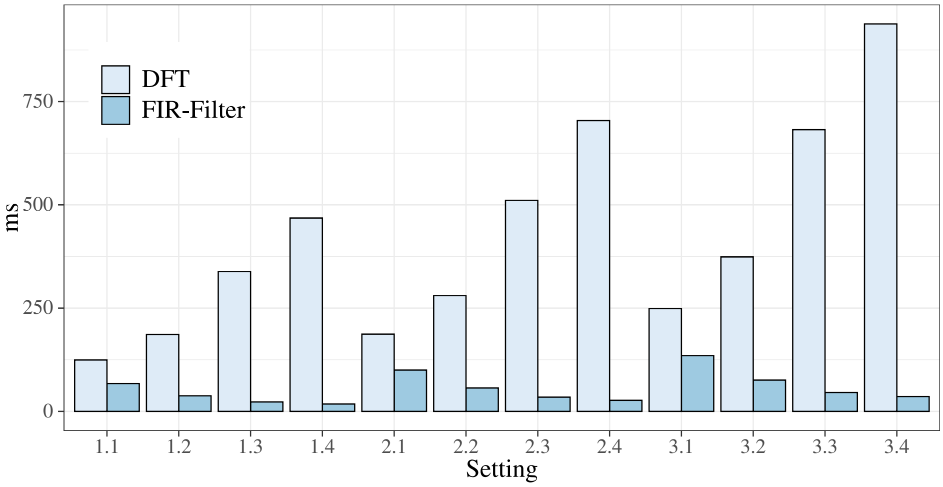

Figure 18 High-Level Program: Runtime of PFB Steps

5.4 GPGPU Program Comparison

Since both GPGPU programs outperform the CPU implementation, in the following, it is evaluated which GPGPU program is more efficient for which reason. This comparison is split into the different parts of the PFB: the FIR-filter, the DFT, and the memory allocation.

5.4.1 Finite Impulse Response Filter

The low-level implementation of the FIR-filter requires, in most cases, merely half of the time or less compared to the high-level implementation as shown in Figure 21. As seen previously, the shared memory kernel does not perform well for many multiplications. In those settings, the high-level implementation is faster than the low-level implementation. In settings with 16 multiplications (x.2) the high-level program is slower than the low-level program by a factor of two. For fewer multiplications, this number gets worse with an approximately slowdown by the factor of 2.8 for 8 multiplications and 2.3 for 4 multiplications. The precise numbers can be seen in Table 7.

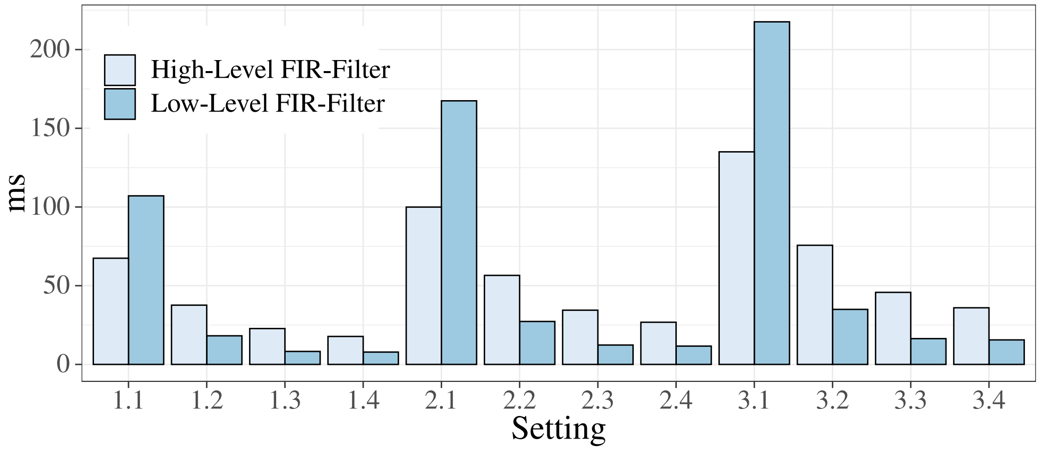

Figure 19 GPGPU Programs: Comparison of Runtime of FIR-Filter

The reason for the time differences can be derived from the different CUDA programs. The most crucial difference between the two implementations is that the low-level implementation uses the shared memory, while the high-level implementation access all element from global memory. In the evaluation of the low-level program, the runtime of a FIR-filter using global memory where discussed. Those runtimes are equal to the high-level program. Hence, it can be derived that the performance difference is caused by the use of the shared memory.

Setting	Low-Level (ms)	High-level (ms)	Speed-up
1.1	107.0851	67.4595	0.63
1.2	18.1641	37.6329	2.0718
1.3	8.2546	22.784	2.7602
1.4	7.8312	17.7618	2.2681
2.1	167.4489	99.9541	0.5969
2.2	27.2807	56.5526	2.073
2.3	12.3132	34.4382	2.7969
2.4	11.6231	26.7876	2.3047
3.1	217.619	135.0443	0.6206
3.2	34.9607	75.6756	2.1646
3.3	16.3748	45.7706	2.7952
3.4	15.5684	35.9665	2.3102

Table 7 GPGPU Programs: Runtime and Speed-up for FIR-Filter Calculations

Due to multiple reasons, the exploitation of the shared memory in high-level frameworks is not trivial. First, data structures might require to be restructured to be efficiently processed. In particular, this is the case when the required elements do not reside next to each other but are distributed across the data structure. Intuitively, the first thread processes the first or the first subset of elements. However, when using faster memory spaces, this might not be longer beneficial. In case one thread processes one element, as commonly done on GPUs, threads which have access to the same memory spaces should be started together since it increases the reuse of elements in the limited shared memory space. In case threads work on multiple elements, as commonly done on CPUs, it is important to group the elements which require the same elements together for one thread. Second, the optimal number of threads and blocks to be started might change depending on the problem. Commonly, the maximum number of threads is started to achieve as much parallelization as possible. However, since faster memory spaces are limited in their size, it might be more efficient to execute fewer threads but have all necessary elements in the shared memory. Moreover, the efficiency of loading data into the shared memory has to be checked. In case the elements are only used once it might not be beneficial to load them into faster memory spaces while repeated usage might profit from the additional memory transfers to faster memory spaces.

As mentioned previously, other skeleton tools faced the same problem and provide distinct skeletons for exploiting similar patterns ([11], [7], [14], [16]). The skeletons *MapStencil* as well as *MapOverlap* commonly require a way of defining a data region which has to be available. In Musket this could be a user function or a completely new type. For the context of the FIR-filter, the *MapStencil* skeleton is the most promising approach to speedup the function. When extending the Musket the design should be as general as possible to enable the reuse in other skeletons. Moreover, the concept of using shared memory should be evaluated for the whole tool and not only for single skeletons.

5.4.2 Discrete Fourier Transform

For the comparison of the DFT, it was decided to include the creation of the cuFFT plan for the low-level implementation, since, without the plan, the execution of the DFT would not be possible. Figure 20 shows how much time the DFT requires. For smaller settings with 32 channels (x.1) the high-level program is closer to the low-level implementation. In the setting with the smallest data size, the high-level implementation even outperforms the low-level implementation. This is the case for setting 1.1, 1.2, and 2.1. With an increasing number of channels, and therefore more iterations of the for-loop in the FFT algorithm, the difference between the implementation gets bigger. Moreover, with bigger input data, the cuFFT library performs better.

More precisely this can be seen in Table 8. For setting 1.1, the high-level approach requires nearly half of the time compared to the low-level approach. For settings 1.2 and

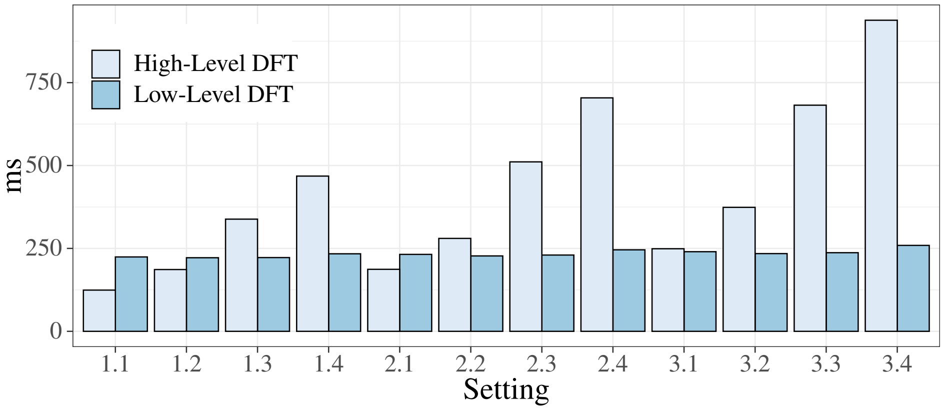

Figure 20 GPGPU Programs: Comparison of Runtime of FFT

2.1, the high-level approach requires approximately 80 % of the time of the low-level approach. For all other settings, the low-level approach is faster. For a small batch size, the difference is comparably small (~1.23 and ~ 1.59). In contrast, with growing input data, and multiple iterations the library performs way better than the high-level implementation (~3.6).

Setting	Low-Level (ms)	High-Level (ms)	Speed-up
1.1	224.3212	124.3783	0.5545
1.2	221.905	186.325	0.8397
1.3	222.4458	338.3462	1.521
1.4	233.786	468.1305	2.0024
2.1	232.0203	186.9578	0.8058
2.2	227.3458	280.1853	1.2324
2.3	230.0581	510.9893	2.2211
2.4	245.7459	704.0037	2.8648
3.1	240.1475	249.0541	1.0371
3.2	234.4144	373.8871	1.595
3.3	237.1285	681.943	2.8758
3.4	259.0762	938.1018	3.6209

Table 8 GPGPU Programs: Runtime and Speed-up of FFT

5.4.3 Memory Management

The runtimes measured for the memory allocation include all array initialisation and the memory transfers between CPU memory and GPU global memory. For the high-level program, memory transfers caused by skeleton calls are not included. Precisely, for the high-level program the initialisation of the data structures, and the copy steps of the input data and the coefficients to the GPU memory and the transfer of the output array to the CPU are included. Transfers and updates which are made implicitly during skeleton calls are included in the runtime of the skeletons.

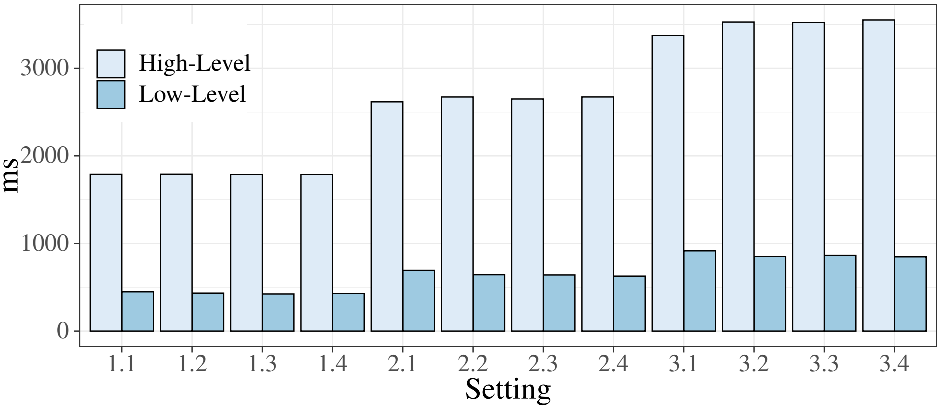

Figure 21 GPGPU Programs: Comparison of Runtime for Memory Allocations

Setting	Low-Level	High-Level	Speed-up
1.1	447.9801	1790.0029	3.9957
1.2	433.3215	1794.7305	4.1418
1.3	423.5102	1784.6591	4.214
1.4	429.4712	1785.0992	4.1565
2.1	693.6451	2616.3794	3.7719
2.2	643.069	2671.5693	4.1544
2.3	640.6371	2653.2254	4.1415
2.4	627.5369	2669.9919	4.2547
3.1	915.7377	3373.9728	3.6844
3.2	851.5365	3535.785	4.1522
3.3	864.5885	3514.7552	4.0652
3.4	847.3214	3552.1804	4.1922

Table 9 GPGPU Programs: Runtime For Memory Management Operations (ms)

The difference between the two approaches is immediately visible in Figure 21. The high-level program requires more time, approximately 3.9 – 4.2 the time compared to the low-level program. As observable in Table 9, the factor stays nearly constant for changing data sizes. However, a factor of approximately 4 is unexpectedly big. Moreover, memory management operations have a relatively big impact on the overall runtime since the data transfer requires the biggest share in the overall runtime. Therefore, it is crucial to examine what causes the differences.

The memory allocation operations are split into transfer operations and allocation operations to identify the weaknesses of the low-level program. Those times are listed in Table 10. In columns 2-4, the time required for copying the input and the coefficients to the GPU global memory is listed. The fourth column divides the LL-time by the HL time. The time varies a little bit, but the low-level program is mostly slower by a factor of 1.5-1.6. For the copy step of the result to the CPU the low-level approach requires 50-65% of the time compared to the high-level approach. However, the most crucial difference is the time required for the allocation of the data structures. The low-level program requires 20% of the time compared to the high-level program.

Setting	HL transfer in	LL transfer in	LL/HL	HL transfer out	LL transfer out	LL/HL	HL allocation	LL allocation	LL/HL
1.1	49.3381	70.298	1.4248	92.0585	59.8278	0.6499	1648.6063	317.8543	0.1928
1.2	49.1993	70.6118	1.4352	92.1748	52.7669	0.5725	1653.3564	309.9428	0.1875
1.3	45.3476	70.8729	1.5629	90.8494	51.5638	0.5676	1648.4621	301.0735	0.1826
1.4	46.8815	75.1677	1.6034	89.0444	53.5169	0.601	1649.1732	300.7866	0.1824
2.1	71.7867	104.857	1.4607	134.0432	90.5855	0.6758	2410.5494	498.2026	0.2067
2.2	70.7226	105.6754	1.4942	131.466	85.1371	0.6476	2469.3807	452.2566	0.1831
2.3	71.6094	111.9172	1.5629	130.8196	84.3542	0.6448	2450.7964	444.3657	0.1813
2.4	67.9167	106.0525	1.5615	130.5194	76.6209	0.587	2471.5558	444.8636	0.18
3.1	75.9544	117.5624	1.5478	179.9401	138.9499	0.7722	3118.0783	659.2253	0.2114
3.2	96.6203	140.7937	1.4572	184.7824	106.9325	0.5787	3254.3823	603.8103	0.1855
3.3	88.6801	141.588	1.5966	173.776	119.8327	0.6896	3252.2991	603.1678	0.1855
3.4	97.6273	143.8647	1.4736	173.8117	107.4757	0.6183	3280.7413	595.981	0.1817

Table 10 GPGPU Programs: Runtime for Grouped Memory Management Operations (ms)

One reason for the performance differences is the initialisation of the arrays. The constructor of a `DArray` in Musket allocates the memory and copies the data to the GPU. This copy step is not necessary for the PFB program. Within the high-level PFB-program first the arrays are initialised then they are filled with data on the CPU, and then the data is copied to the GPU. This makes the first copy step in the constructor unnecessary. For example, for settings 1.1, removing the copy steps improves the allocation time from 1648 ms to 1153 ms on average.

One essential difference between the programs is that the low-level program requires more memory space. An overview of the created data structures and the required memory space can be seen in Table 11. In the high-level program, three arrays are allocated on the CPU and on the GPU. In contrast, the low-level program merely allocates two arrays in the CPU memory and three in the GPU memory. Moreover, the arrays require less memory space due to the requirements posed by the algorithm used. It can be seen that, especially when looking at the data structures which are needed to store intermediate results, the low-level program is more efficient. The output of the FIR-filter is merely saved on the GPU and can be reused as input for the DFT. In the high-level implementation, the output of the FIR-filter is saved to an array of type complex, which is double the data size. Moreover, this array exists on the CPU and on the GPU memory. Last, the output of the FFT requires in the low-level approach merely half the size, compared to the high-level approach. In case of setting 1.2 (134,217,728 elements, 64 channels, 16 taps) the low level implementation would require 2.2817 KB subsuming GPU and CPU while the high-level implementation requires approximately 5.3688 KB. This means more than double the size is allocated (~ 2.36).

Purpose	Low-Level			High-Level		
	GPU	CPU	Total	GPU	CPU	Total
input	✓	✓	$channels \cdot (spectra + taps - 1) \cdot 4$	✓	✓	$channels \cdot (spectra + taps - 1) \cdot 4$
input_double	✓	✓	$channels \cdot spectra \cdot 8$	✓	✗	$channels \cdot spectra \cdot 4$
DFT_output	✓	✓	$channels \cdot spectra \cdot 8$	✓	✓	$\frac{channels \cdot spectra}{2} \cdot 8$

Table 11 GPGPU Programs: Allocated Memory Space

A possibility to optimise the data structures would be to automatically identify which structures are never required on the CPU and save the allocation of CPU memory for those data structures. However, analysing code for those properties might be a rather troublesome task. Another option could be to let the user set flags which results should be available on the CPU and which data structures should be available only on the GPU. This would simplify the detection of potential data structures. In this specific context, additional memory space was also lost since the FFT-algorithm requires additional space in comparison to the approach of the cuFFT library.

5.4.4 Overall Runtime

From the algorithmic perspective, the runtimes of the computations are the most interesting part. For deploying the program, the overall runtime is essential. This includes, most importantly, memory transfers, which have shown to be the most time-consuming operations as soon as the calculations are parallelised. The complete runtime for all settings is displayed in Figure 22 with the concrete numbers in Table 12. In total, the high-level program is slower by a factor of ~2.54 to ~4. Two characteristics of the program have an influence on the variation in the factor.

First, the high-level programs scale worse for bigger data sizes. The factor increases for one setting of channels and taps. For example, for setting x.1 the factor for small data sizes is 2.54 (1.1), for the medium data size 2.65 (2.1), and for the biggest tested data size 2.73 (3.1). This slow down is caused by the share the memory allocation operations have of the overall runtime. From all operations, those belonging to memory management have the worst scaling factor. The share of those operations increases for bigger data sizes. Hence, the overall time difference increases. Noticeably, the difference is not precisely linear. For example, for settings x.1 the difference between the data sizes was approximately 0.1, while for x.2, the difference is 0.35 and 0.20. Those differences can be ascribed to the next factor.

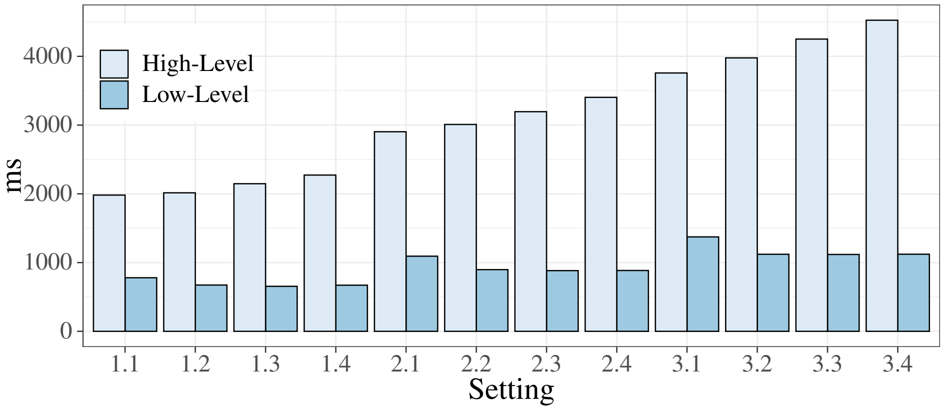

Figure 22 GPGPU Programs: Comparison of Complete Runtime

Setting	Low-Level	High-Level	Speed-up
1.1	779.3884	1981.8408	2.5428
1.2	673.3925	2018.6905	2.9978
1.3	654.2126	2145.7914	3.28
1.4	671.0903	2270.9935	3.384
2.1	1093.1162	2903.2934	2.656
2.2	897.6975	3008.3092	3.3511
2.3	883.0104	3198.655	3.6224
2.4	884.9078	3400.7853	3.8431
3.1	1373.5061	3758.0733	2.7361
3.2	1120.9136	3985.3498	3.5554
3.3	1118.0939	4242.4708	3.7944
3.4	1121.9681	4526.2507	4.0342

Table 12 GPGPU Programs: Runtime and Speed-up of Complete Programs (ms)

Second, for a high number of iterations for the FFT algorithm (x.4), the high-level program performs slower than the low-level program. For example, for setting 1.x, the runtime of the high-level program increases from 1981.84 ms to 2270.99 ms, while for the low-level program, the overall runtime decreases from 779.39 ms to 671.09 ms. For setting x.1, x.2, x.3, and x.4 the time required for memory allocations are nearly constant. Therefore, the difference is caused by the difference in the efficiency of the FIR-filter and the FFT algorithm. In the previous section, it was shown that the high-level programs perform in comparison to the low-level program better for settings with an increasing number of multiplications in the FIR-filter, and additionally in settings which have little iterations in the FFT algorithm. Contrarily, the low-level program performed better for a low number of multiplications in the FIR-filter and the performance stays nearly constant with a rising number of iterations in the FFT. For this purpose, the high-level program comes closer to the low-level implementation for x.1 settings, and with an increasing number of channels and a decreasing number of taps loses performance.

5.4.5 Applicability of Programs

The most essential requirement for the use of the programs is the runtime of the program. For the application context, the program should be capable of processing batches of 250 million samples of data in 0,125 seconds. In this context, one sample is one floating-point number. An uncommon peculiarity was that it could be assumed that the data already resides on the GPU. Hence, it can be abstracted from data transfers and allocation operations. For the applicability, it should be evaluated how much time is required for processing approximately 250,000,000 elements. In this context, 268,435,456 elements are used since GPU programs scale especially well for data sizes which are a power of two.

An overview of the runtimes excluding memory transfers is shown in Table 13. It is distinguished between three types of programs. For the low-level program, two variants exist: One including the creation of the cuFFT plan one without the plan. This differentiation is made since the creation of the plan is independent of the input data. For the application context, this means that the plan can be created before the data is processed. Therefore, for applying the program in practice, it is valid to exclude the cuFFT plan creation. The last program listed is the high-level program.

The benchmark of 125 ms is fulfilled by the low-level program for settings 3.2, 3.3, and 3.4 in case the creation of the plan is not included in the runtime. The program requires approximately 34%-65% of the time available. For setting 3.1, none of the programs fulfils the requirement. When the creation of the plan is included, the low-level program requires approximately double the time. The high-level program performs slower and requires approximately 3,5-7,8 of the time. The runtime of the FFT causes an increase in the runtime. Hence, the performance of the high-level program is not sufficient for the application context.

Setting	Low-Level				High-Level	
	ms	%	ms	%	ms	%
3.1	248.2389	1.9859	457.7665	3.6621	384.0984	3.0728
3.2	61.1968	0.4896	269.3751	2.155	449.5627	3.5965
3.3	43.121	0.345	253.5033	2.028	727.7136	5.8217
3.4	65.0726	0.5206	274.6447	2.1972	974.0683	7.7925

Table 13 GPGPU Programs: Runtime of PFB Steps (ms)

For the application context, the FIR-filter needs to be optimised for settings with 32 multiplications. The solution generated by the high-level approach using the global memory performs sufficiently good to fulfil the requirement combined with the cuFFT library. Precisely, using global memory for the FIR-filter requires 67 ms. The cuFFT library implementation of the FFT requires 15 ms. In total, this results in a runtime of 82 ms which satisfies the requirement. Therefore, a possible solution for practice is to make a case distinction and use the global memory kernel settings with 32 multiplications in the FIR-filter. Alternatively, the shared memory kernel could be optimised. With the current implementation, two problems exist: First, half of the threads idle since they have no channel to process. Therefore, the program could be adjusted to process 16 channels and two spectra per warp. A disadvantage of this design might be that threads request the same coefficients from shared memory. However, the precise impact has to be measured to evaluate the effect on the runtime of the concurrent memory requests. Second, due to the design of the program, each thread calculates three elements. Hence, per thread 3×32 multiplications are performed. This number might be too high to be efficient. This number could be reduced.

5.5 Non-Functional Requirements

Hitherto, the evaluation merely included the comparison of the runtime. Although this is an important factor, other aspects might influence the use of the high- or low-level approach. While the runtime can be seen as a minimum requirement, other aspects can be seen as additional benefits which favour a solution.

Cole used the term “sustained performance“ [6, p. 393] to highlight the importance of transferability and maintainability. Concretely, the term is used to describe that with a tool which is portable to different hardware and is updated to newer software versions of the parallel framework used, retains the performance of a program. This means that when the program is newly generated, the performance can be maintained or improved, although the surrounding circumstances change. In case a user buys a new GPU or even changes from a single-GPU architecture to a multi-GPU architecture, a Musket program is designed to adjust to those changes. If the number of GPUs changes, the program must be newly generated. In Musket and most other high-level frameworks, the number of GPUs and nodes used can be adjusted effortlessly. In the case of a changing GPU-architecture, the program is already designed to adjust to different GPUs, by reading the device properties and adapt the program accordingly. Moreover, when new features are introduced in the parallel framework used by the high-level approach, the inclusion of the feature must only be included once in the high-level framework. All programs generated with the high-level framework have to be merely generated again to include the changes.

In contrast, changes in the software and in the architecture are arduous to include in a low-level program. When changing the GPU, a low-level program might include hard-coded configurations specific to the hardware which can not be transferred to other GPUs.

In case the number of GPU changes, a low-level program would require to include multiple communication steps. The adjustment of a low-level program to multiple GPUs and nodes might require to learn an additional tool for parallel programming such as message passing interface (MPI). Moreover, when new features of the toolkit used are introduced, the suitability of the feature for the problem to solve has to be checked. This inclusion requires expertise.

Another important aspect is the usability and the effort required to create the program. While low-level programs require sophisticated instructions, for example, for the memory transfers and the number of threads started, high-level approaches abstract from those instructions. Thus, they can be created considerably faster, and the overall program is usually shorter than the corresponding low-level programs. However, the exact difference highly depends on the programmers' skills and experience. The scope of this thesis can not include a study to precisely evaluate the usability. In some context, the software metric logical lines of code (LLOC) is used to derive an indicator for the complexity. This metric counts the lines of a program which execute one operation. For example, `y = x+1;` is one logical line of code while `y = x+1; z = y+1;` are counted as two LLOC although they might be written in the same line. Moreover, operations which do not execute logical statements are excluded, for example, closing brackets are not counter as LLOC. The exact definition depends on the used language. This metric is not easily applicable to the implemented programs since the high-level implementation missed some functionalities such as a function calculating the sinus. In the given number for the LLOC, it is assumed that those functions are implemented.

The high-level program has 62 LLOC, while the low-level program has approximately 160 LLOC. This number depends on the functionalities included such as possible error checks. It should be merely used as an indicator that the low-level program is more complex. It should also be considered that one of the major criticism of this metric is that not every statement requires the same effort. For example, thinking of an ideal number of threads requires more effort than assigning the number of channels to a variable. Consequently, for a precise analysis, a study would need to be conducted, which measures the time required to create the program, for programmers with different levels of expertise.

In summary, high-level approaches have an advantage of transferability, maintainability, and usability. The magnitude of this advantage depends on the requirements of the application context and therefore needs to be evaluated separately. Especially for contexts where limited expertise is available, high-level approaches might be more suitable.

5.6 Interpretation of Results

The presented evaluation is used to answer *RQ1* and *RQ2* stated in Chapter 3.

RQ1: Which approach can satisfy the requirements from the practical application context?

From all evaluation results, it is shown that with a huge amount of data using a parallel GPU program is faster than a CPU program for the given problem. Moreover, the speedup is growing with increasing data size. Moreover, for settings with am increasing number of multiplications, the program is faster since the calculation of the FIR-filter is especially time-consuming in the used CPU program. This reasons that for the practical application using a GPU program is preferable to a CPU program.

For the application context, in three of four cases, the low-level program was sufficiently fast to satisfy the runtime requirement. For the setting with 32 taps and 16 channels, either the low-level filter has to be used, or the existing kernel using the shared memory has to be optimised to satisfy the requirement. Non-functional requirements are not of major importance for the given application context but should be kept in mind when further evaluating high-level approaches.

RQ2: What are the advantages and disadvantages of high- and low-level implementation?

The major disadvantage of the high-level program is that for the given problem, the high-level program is slower by a factor of ~2.5-4 for the given settings. It was found that the major cause for the performance difference is the time required for memory allocations which always takes approximately the quadruple compared to the low-level program. For the FIR-filter, the high-level program is faster for the settings with 32 multiplications requiring 60% of the time. For the other settings the program is slower by a factor of ~2-2.7. This factor stays constant with growing data sizes. For calculating the DFT the high-level program faster than the low-level program for few iterations and small data size. With an increasing data size and more iterations, the low-level program is faster, and it is to be expected that this factor increases with growing data sizes.

The findings of the FIR-filter and the DFT show that depending on the configuration, and the data size, different approaches can be more effective. Therefore, the advantage of a high-level program is to reveal possible unexpected solutions. Another advantage of the high-level solution is that creating the program requires less expertise and can be done faster. Moreover, the solution can be easily configured for multi-GPU environments, and the maintenance of the solution is easier.

In contrast, low-level solutions are highly customizable to specific features of a problem. In this application context, an example was the outsourcing of the cuFFT plan creation to achieve the required runtime and the usage of GPU pointers since the data already resides on the GPU.

5.7 Implications of Results

The results can be interpreted as implications on three levels of abstraction. First, it can be concluded what the results mean for our application context. Second, the impacts on Musket can be discussed. Last, deductions for generating GPU code are made. Those aspects answer *RQ3* and *RQ4*.

RQ3: What possible improvements can be made to the given implementations?

Most importantly, it was shown for the context of processing telescope data, that fine-tuned accelerations are essential to achieve the desired performance. However, it was also found that the high-level approach performed better for the FIR-filter with 32 multiplications, which means that the code generation served as a supplement and performance comparison to conclude which approach is most suitable for solving the problem. To improve the current implementation, the shared memory use could be optimised by adjusting parameters, or a program could be build which executes dependent on the taps size a different kernel.

RQ4: What improvements can be made to the high-level framework?

For Musket it is derived that for improving the high-level approach, memory operations have to be improved. This could include enabling users to specify whether data structures are only required on the GPU or in which cases data structures should be updated. Alternatively, also the code compilation could conduct a more precise code analysis which data structures are required on which device. Moreover, the approach could be improved by using shared memory. Specifically, this would improve the runtime of the FIR-filter. However, as could be seen in the application context, this is not efficient in all cases. Hence, it should be evaluated how it can be assured that Musket only uses the shared memory in case it is efficient. In comparison to memory management, this should be queued since the performance difference in the FIR-filter is comparatively small. The cause of the difference for conducting the FFT can not be specified since the code of the cuFFT library is not available. It is deduced that it is troublesome to achieve the performance of a sophisticated library developed by Nvidia engineers who have precise knowledge of the hardware.

Last, findings for the approach of generating GPU-code, more precisely parallel GPU-code, are that it is confirmed that parallel programs can be created much faster with less expertise in parallelism, and the underlying framework. However, the benchmarks for the performance show that for a context where the performance should be completely exploited, high-level frameworks might lack the features to achieve the performance. Moreover, it was shown that processing streaming data and including software patterns such as classes might improve the integration of high-level code into an application context.

6 Outlook

This thesis evaluated one exemplarily high-level approach, Musket, and one exemplarily low-level approach, CUDA, in one application context. To generalize those findings, they need to be complemented with other approaches and a variety of real-world applications.

The application context could be extended by adding supplementary steps in PFBs, such as oversampling data. Next, other applications such as the calculations of heat equations can be used to evaluate how the approaches perform on other problems. Problems in physics are often especially suitable since the amount of data is enormous. However, also, alternative application contexts such as forecasting could be considered. From those applications context, further features and requirements for high-level approaches can be deduced. This might include new skeletons, or data structure enhancements to allow to solve problems more efficiently. However, it is essential not to create an overwhelming amount of skeletons since in that case, the user is overwhelmed, and the approach gets increasingly complex. In case the high-level approach already performs sufficiently good, the applicability is proven.

Another dimension is to evaluate the strength and weaknesses of other approaches to find suitable improvements for the tested frameworks. This is achieved by creating several high and low-level programs for solving the same problem. Supplementary to the skeleton approach, the BSP approach should be evaluated. Moreover, other low-level approaches such as OpenAcc might be topic to further studies. Those comparisons possibly identify helpful features to improve efficiency. Furthermore, perhaps differentiation can be made which problem is suitable for which approach. This might include identifying characteristics of problems which make them suitable for specific approaches.

A further research question is the precise analysis of non-functional requirements. As shortly discussed transferability, maintainability, and usability influence the suitability of an approach and should be considered and measured. This opens an entirely new topic since new methods to measure those dimensions are required. Especially for measuring the usability, the design of a sophisticated study has to be developed, including the survey of users. This topic is no longer only suspect to the applicability of high-level approaches but also considers aspects of software acceptance. Moreover, the integration of programs generated with high-level approaches can be discussed. The Fastflow framework integrates streaming interfaces into their tool [2], which simplify the generation of programs for the streaming context. However, it should be discussed generally how the source of data is included when writing programs with high-level frameworks.

Additional research might also include the discussion of the addresses of high-level frameworks. For using Musket, knowledge about skeletons is required for creating programs. Complementary, it improves the program if the basic concepts of parallel programming are understood. For example, to evaluate whether a problem is more suitable to be solved on the GPU or CPU. High-level approaches are still evolving, and further consideration of this development process might be how much knowledge the user is expected to have. A simple example would be to let users flag which data structures are required on the CPU and which structures might be only available on the CPU. A more complex example was discussed in case of the MapStencil skeleton. There exist multiple possibilities to decide whether the stencil should be loaded into shared memory. First, the tool evaluates whether the stencil data should be loaded into faster memory. For this purpose, it would be required to check if the data is reused sufficiently often. Second, it could be assumed that the user of the tool has a sufficient amount of knowledge, and the data could always be loaded into faster memory. The last of the named possibilities here could be to generate different programs and evaluate their performance before deciding which approach is taken. This approach was pursued by STEUWER[30].

Last, another interesting topic not discussed in this thesis is the comparison of the performance in multi-GPU environments. Creating programs for multi-GPU environments is a strenuous task since it requires knowledge about communication between different GPUs and nodes. This feature is currently work in progress in Musket [29]. The given problem could be used to test how efficient a Musket implementation with multiple GPUs and possibly multiple nodes is. Other tools have already included this feature, such as Muesli [8], SkelCL [31], and SkePU [7]. Especially the efficient computation of the FFT might be an interesting topic since the intermediate results have to be communicated between the different GPUs. Generally, communication between GPUs is rather inefficient. For this purpose, techniques like memory transfers from GPU to GPU should be discussed for GPUs residing on the same node, and MPI for communication between distinct nodes. This might also include adjustments to algorithms such as copying more data than necessary to a GPU to allow the calculation of more elements.

Conclusively, there exist a variety of different research direction which can be pursued. However, the most important goal of parallel programs, to provide a sufficient speedup should be prioritized until the further design of high-level approaches and their usability is evaluated and improved.

7 Conclusion

During the development of the programs it has been shown that a low-level implementation requires sophisticated considerations regarding the memory used, the number of threads started, and the distribution of data to different threads. Most importantly, shared memory is discussed as a possibility to speed up the computation of the FIR-Filter. For the implementation of the DFT the cuFFT library was used. For the high-level implementation, *Map* skeletons are used to realize the FIR-Filter and the FFT. The creation of the high-level program showed that application-specific requirements might not be realizable in the used high-level framework. An example is the assumption that the data to process already resides on the GPU. In total, the time required to develop the programs confirms that the creation of low-level programs requires more effort than the creation of a program with a skeleton tool.

««< HEAD Within the evaluation of the programs, it is shown that for all cases, the GPU programs outperform a comparable CPU program. In three of four cases, the low-level program could satisfy the runtime requirement. The program developed with a high-level approach could not satisfy the runtime requirement. Moreover, it was shown, that for the overall runtime, the low-level program is remarkably faster than the high-level program with a speedup of 2.5-4. ======= Within the evaluation of the programs, it is shown that for all cases, the GPU programs outperform a comparable CPU program. In three of four cases, the low-level program could satisfy the runtime requirement. The program developed with a high-level approach could not satisfy the runtime requirement. Moreover, it was shown, that for the overall runtime, the low-level program is remarkably faster than the high-level program with a speed-up of 2.5-4. »»»> 876be9d02c59980523f47d104eb35fe72c4220a8 To identify the causes for the performance differences the program was split into the distinct parts: (1) memory management operations, (2) the calculation of the FIR-filter, and (3) the calculations of the FFT. It was shown that the difference is mainly caused by the inefficient memory management of the high-level program. Unnecessary memory copies and the use of additional memory space require a significant amount of time. This has a major impact on the overall program since the memory management operations are the most time-consuming part of the program.

Regarding the comparison of the calculation steps, the findings for the two GPU programs vary. For the FIR-filter the high-level approach performs better for 32 multiplications per element while the low-level implementation is faster for 16, 8, and 4 multiplications per element. Two characteristics caused the performance difference. First, the way the shared memory is used by the low-level program results in having idle threads when the channel size is smaller than 32. Hence, for a setting with 32 multiplications and 16 channels, the low-level program is remarkably slower than the high-level program. Second, loading data into shared memory becomes less efficient for the setting since the space available is limited. Therefore, fewer items can be calculated by each block started. Both findings show that using shared memory is not trivial. Although its use can accelerate a program the inclusion into a high-level framework should be systematically thought-out to assure that it is only used in case it improves the runtime.

For the realization of the DFT it was shown that the library used by the low-level program performs especially well for big data sizes and multiple iterations. For smaller data sizes, the high-level implementation was faster. Since the code of the library is not publicly available, merely assumptions about the causes for the differences could be made. Since the library is provided by the manufacturer of the GPU used, it can be assumed that the library exploits all building blocks available. Possibly, faster memory spaces are allocated, or internal building blocks to pipeline calculations are set up. However, the improvement of the used skeletons for executing the FFT might be a topic to future work.

Conclusively, in the given application context, the high-level approach is not fast enough to satisfy the requirements, and the low-level approach satisfies the requirement in three of four cases. The main cause for the performance differences between the GPU programs is the time required for memory management operations which should be optimized. Regarding the calculations, varying results are found. Depending on the parameters of the calculation performed, different approaches are more suitable. A possible future enhancement for the high-level framework might be the structured use of the shared memory. Generally, the generation of GPU code is a promising approach to simplify parallel programming and develop efficient programs but has to be adapted to suit a real application context and can be optimized to be more efficient.

Appendix

A Musket Programs

A.1 Complete PFB

The following program depicts the complete musket implementation of a PFB. However, since musket is not fully developed, it has to be noticed that some features are not fully functional.

(1) The functions `sin`, `cos`, and `std::pow()` are not translated into C code.

(2) Bitwise operants such as ˆ are not supported. Bit-shifts were replaced with the corresponding multiplication with the power of two. Bitwise ands were replaced with if statements.

(3) Structs are not supported in the CUDA generator. As a workaround, the struct was named as the corresponding C data-type which allows to work with the type as expected.

(4) The build-in time measurement with the `mkt::roi` call cannot be used multiple times since the variables are created repeatedly and the C compiler throws an error.

(5) Array indexing in user functions does not work. In this case, indexing is not complex since all data resist on one GPU. However, when the data is distributed on different GPUs or even on different nodes requesting the right data gets complex.

For this use case, the missing functionalities were manually implemented. Their inclusion in the code generator should be topic to future projects.

```
#config PLATFORM GPU CUDA
#config PROCESSES 1
#config GPUS 1
#config MODE release

const double PI = 3.141592653589793;

struct float2{
  float x;
  float y;
};

array<float,134225920,dist> input;
array<float2,134217728,dist> c_input_double;
array<float2,134217728,dist> c_output;
array<float,512,dist> coeff;

float init(int x){
  return (float) mkt::rand();
}

float FIR(int taps, int channels, int spectra, int Index, float a) {
  float newa = 0;
  for (int j = 0; j < taps; j++) {
    newa += input[Index+(j*channels)] * coeff[Index%(taps*channels)+(j*
   channels)];
  }
  return newa;
}

float2 combine(int counter, int log2size, double PI, int Problemsize,
   int Index, float2 Ai) {
  float2 newa;
  newa.x = 0.0f;
  newa.y = 0.0f;
  int b = Index / (std::pow(2, log2size-1-counter));
  int b2 = 0;
  for (int l=0; l<=counter; l++){
    if (b == 1) {b2 = 2*b2+1;}
    else { b2 = 2*b2;}
    b = b / (2);
  }
  float temp = 2.0 * pi/ Problemsize * (b2 * (std::pow(2, log2size-1-
   counter)));
  float2 intermediateresult;
  intermediateresult.x = cos(temp);
  intermediateresult.y = sin(temp);
  if (Index == (std::pow(2, log2size-1-counter))) {
    float2 mult_res;
    mult_res.x = intermediateresult.x * Ai.x - (intermediateresult.y *
   Ai.y);
    mult_res.y = intermediateresult.x * Ai.y + intermediateresult.y *
   Ai.x;
    float2 add_res;
    add_res.x = c_input_double[Index].x + mult_res.x;
    add_res.y = c_input_double[Index].y + mult_res.y;
    newa = add_res;
  }
  else {
    float2 mult_res2;
    mult_res2.x = intermediateresult.x * c_input_double[Index].x - (
   intermediateresult.y * c_input_double[Index].y);
    mult_res2.y = intermediateresult.x * c_input_double[Index].y +
   intermediateresult.y * c_input_double[Index].x;
    float2 add_res2;
    add_res2.x = Ai.x + mult_res2.x;
    add_res2.y = Ai.y + mult_res2.y;
    newa = add_res2;
  }
  return newa;
}

float2 fetch(int counter, int log2size, int i, float2 Ti){
  return c_output[i ^ (int) std::pow(2, log2size-1-counter)];
}

main{
  int ntaps = 32;
  int nchans = 16;
  int nspectra = 8388608;
  int log2size = 4;
  input.mapInPlace(init());
  coeff.mapInPlace(init());

  mkt::roi_start();
  c_output.mapIndexInPlace(FIR(ntaps, nchans, nspectra));

  for (int j=0; j<log2size; j++) {
    c_input_double.mapIndexInPlace(fetch(j, log2size));
    c_output.mapIndexInPlace(combine(j, log2size, PI, 16));
  }
  mkt::roi_end();
}
```

Listing 13 Musket Implementation: Complete PFB

B Run Times

Setting	Rows of Spectra Processed per Kernel Call	FIR-Filter Time	FFT Time	CuFFT initialization	Transfer CPU to GPU	Transfer GPU to CPU	Allocation	cuFFT Plan Creation	Total time
1.1	25	107.0851492	15.45336544	0.001976889	70.29797011	59.82780033	317.8542956	208.86782	779.3883776
1.2	41	18.16409723	13.023272	0.001908343	70.61183729	52.76685947	309.9427773	208.8817226	673.3924742
1.3	49	8.254646857	13.42624731	0.0019216	70.87289807	51.56378031	301.0735256	209.0196011	654.2126208
1.4	53	7.831157943	24.65794836	0.001953829	75.167659	53.51694275	300.7865711	209.128079	671.090312
2.1	25	167.4488678	23.31726589	0.001969778	104.8570498	90.58550433	498.2025519	208.7030079	1093.116217
2.2	41	27.28069749	19.6409113	0.001974629	105.6753964	85.13705241	452.2565744	207.704869	897.6974757
2.3	49	12.31319634	20.08209531	0.001934857	111.9172384	84.3542116	444.3656573	209.9760394	883.0103732
2.4	53	11.62307954	36.84016936	0.001921829	106.0524673	76.62085838	444.8635953	208.9057291	884.9078208
3.1	25	217.6190357	30.61984011	0.001944889	117.562422	138.9499407	659.2253214	209.5276387	1373.506143
3.2	41	34.96069806	26.2361342	0.001992457	140.7937372	106.9325037	603.8102679	208.1782369	1120.91357
3.3	49	16.37483839	26.74617678	0.002048229	141.5880376	119.8326983	603.167777	210.3822991	1118.093875
3.4	53	15.56844139	49.50419861	0.0019808	143.8647328	107.4756832	595.9809964	209.5720171	1121.96805

Table 14 Low-Level Program: Runtime (ms)

Setting	FIR-Filter Time	FFT Time	Allocation	Transfer CPU to GPU	Transfer GPU to CPU	Total Time
1.1	67.45950075	124.3783414	1648.606327	49.33806271	92.05851853	1981.84075
1.2	37.63294566	186.3249989	1653.35642	49.19925728	92.17480851	2018.68843
1.3	22.7840103	338.3462197	1648.462109	45.3475921	90.84936601	2145.789297
1.4	17.7617508	468.1305304	1649.173248	46.88152186	89.04441854	2270.99147
2.1	99.95411	186.957816	2410.549444	71.78672092	134.0432304	2903.291322
2.2	56.55258014	280.1853454	2469.380691	70.7225672	131.466	3008.307184
2.3	34.4382474	510.9893463	2450.796414	71.60939544	130.8195946	3198.652997
2.4	26.7876048	704.0036931	2471.555798	67.916693	130.5194	3400.783189
3.1	135.0442986	249.054148	3118.078311	75.95439588	179.940111	3758.071265
3.2	75.67561944	373.8870602	3254.382349	96.62026372	184.7823896	3985.347682
3.3	45.7706053	681.9429931	3252.299119	88.68009581	173.776	4242.468813
3.4	35.966464	938.1018329	3280.741291	97.62733943	173.811737	4526.248665

Table 15 High-Level Program: Runtime (ms)

Setting	CPU_FIR	CPU_FFT	CPU_total
1.1	69742.92992	3326.455929	73069.38585
1.1	23993.70582	5906.697269	29900.40309
1.1	7988.3107	7032.650814	15020.96151
1.1	5497.109301	6663.392636	12160.50194
1.1	99205.61573	4165.30137	103370.9171
1.1	35593.10049	8857.128492	44450.22898
1.1	12082.10712	10714.06943	22796.17655
1.1	8469.970112	10256.11416	18726.08427
1.1	125998.0235	5943.960566	131941.984
1.1	48565.77062	12393.52654	60959.29716
1.1	16406.83837	14736.59507	31143.43344
1.1	11736.98672	14129.8411	25866.82782

Table 16 CPU Program: Runtime (ms)

References

[1] K. Adámek, J. Novotný, and W. Armour. A polyphase filter for many-core architectures. *Astronomy and Computing*, 16:1 – 16, 2016.

[2] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati. Fastflow: high-level and efficient streaming on multi-core. *Programming multi-core and many-core computing systems, parallel and distributed computing*, 2017.

[3] M. Aldinucci, G. P. Pezzi, M. Drocco, C. Spampinato, and M. Torquati. Parallel visual data restoration on multi-GPGPUs using stencil-reduce pattern. *The International Journal of High Performance Computing Applications*, 29:461–472, 2015.

[4] J. Chennamangalam, S. Scott, G. Jones, H. Chen, J. Ford, A. Kepley, D. R. Lorimer, J. Nie, R. Prestage, D. A. Roshi, and et al. A GPU-based wide-band radio spectrometer. *Publications of the Astronomical Society of Australia*, 31, 2014.

[5] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, Cambridge, MA, USA, 1991.

[6] M. Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30:389 – 406, 2004.

[7] J. Enmyren and C. W. Kessler. SkePU: A multi-backend skeleton programming library for multi-GPU systems. In *Proceedings of the Fourth International Workshop on High-level Parallel Programming and Applications*, HLPP '10, pages 5–14, New York, NY, USA, 2010. ACM.

[8] S. Ernsting and H. Kuchen. Algorithmic skeletons for multi-core, multi-GPU systems and clusters. *Int. J. High Perform. Comput. Netw.*, 7:129–138, 2012.

[9] A. Gerbessiotis and L. Valiant. Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing*, 22:251 – 267, 1994.

[10] S. Gorlatch. Programming with divide-and-conquer skeletons: A case study of FFT. *The Journal of Supercomputing*, 12:85–97, 1998.

[11] B. Hagedorn, L. Stoltzfus, M. Steuwer, S. Gorlatch, and C. Dubach. High performance stencil code generation with Lift. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, pages 100–112, New York, NY, USA, 2018. ACM.

[12] M. Harris. Using shared memory in CUDA C/C++. https://devblogs.nvidia.com/using-shared-memory-cuda-cc/, 2013. Accessed: 2019-08-29.

[13] A. R. Hevner, S. T. March, J. Park, and S. Ram. Design science in information systems research. *MIS Quarterly*, 28:75–105, 2004.

[14] H. Kuchen. A skeleton library. In *Euro-Par 2002 Parallel Processing*, pages 620–629, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[15] H. Kuchen and M. Cole. The integration of task and data parallel skeletons. *Parallel Processing Letters*, 12:141–155, 2002.

[16] M. Lesniak. PASTHA: Parallelizing stencil calculations in haskell. In *Proceedings of the 5th ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming*, DAMP '10, pages 5–14, New York, NY, USA, 2010. ACM.

[17] T. Lobato Gimenes, F. Pisani, and E. Borin. Evaluating the performance and cost of accelerating seismic processing with CUDA, OpenCL, OpenACC, and OpenMP. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 399–408, 2018.

[18] R. Marques, H. Paulino, F. Alexandre, and P. D. Medeiros. Algorithmic skeleton framework for the orchestration of GPU computations. In *Euro-Par 2013 Parallel Processing*, pages 874–885, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[19] Nvidia. Openacc programming and best practices guide. https://www.openacc.org/sites/default/files/inline-files/OpenACC_Programming_Guide_0.pdf, 2015. Accessed: 2019-09-12.

[20] Nvidia. Cuda toolkit v10.1.243. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#features-and-technical-specifications, 2019. Accessed: 2019-08-29.

[21] Nvidia. Cuda toolkit v10.1.243 cufft. https://docs.nvidia.com/cuda/cufft/index.html, 2019. Accessed: 2019-08-30.

[22] Nvidia Corporation. CUDA C programming guide, 2019. available at: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[23] Nvidia Corporation. *CUDA toolkit documentation*, 2019.

[24] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee. A design science research methodology for information systems research. *Journal of Management Information Systems*, 24:45–77, 2007.

[25] M. Poldner and H. Kuchen. Algorithmic skeletons for branch and bound. In *Software and Data Technologies*, pages 204–219, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[26] M. Poldner and H. Kuchen. On implementing the farm skeleton. *Parallel Processing Letters*, 18:117–131, 2008.

[27] D. C. Price. Spectrometers and Polyphase Filterbanks in Radio Astronomy. *arXiv e-prints*, page arXiv:1607.03579, 2016.

[28] M. Quinn. *Parallel Computing: Theory and Practice*. Computer science series. McGraw-Hill, 1994.

[29] C. Rieger, F. Wrede, and H. Kuchen. Musket: A domain-specific language for high-level parallel programming with algorithmic skeletons. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, SAC '19, pages 1534–1543, New York, NY, USA, 2019. ACM.

[30] M. Steuwer. *Improving programmability and performance portability on many-core processor*. PhD thesis, University of Münster, 2015.

[31] M. Steuwer, M. Friese, S. Albers, and S. Gorlatch. Introducing and implementing the allpairs skeleton for programming multi-GPU systems. *International Journal of Parallel Programming*, 42:601–618, 2014.

[32] G. Strang and T. Nguyen. *Wavelets and Filter Banks*. Wellesley-Cambridge Press, 1996.

[33] C. Su, P. Chen, C. Lan, L. Huang, and K. Wu. Overview and comparison of OpenCL and CUDA technology for GPGPU. In *2012 IEEE Asia Pacific Conference on Circuits and Systems*, pages 448–451, 2012.

[34] K. van der Veldt, R. van Nieuwpoort, A. L. Varbanescu, and C. Jesshope. A polyphase filter for GPUs and multi-core processors. In *Proceedings of the 2012 Workshop on High-Performance Computing for Astronomy Date*, Astro-HPC '12, pages 33–40, New York, NY, USA, 2012. ACM.

[35] F. Wrede and C. Rieger. https://github.com/wwu-pi/musket-material, 2019. Accessed: 2019-09-12.

[36] F. Wrede, C. Rieger, and H. Kuchen. Generation of high-performance code based on a domain-specific language for algorithmic skeletons. *The Journal of Supercomputing*, 2019.

Declaration of Authorship

I hereby declare that, to the best of my knowledge and belief, this Master Thesis titled "Exploring High- and Low-Level Approaches for GPGPU Processing of Telescope Data." is my own work. I confirm that each significant contribution to and quotation in this thesis that originates from the work or works of others is indicated by proper use of citation and references.

Münster, 16th September 2019

Nina Herrmann

Consent Form

for the use of plagiarism detection software to check my thesis

Last name: Herrmann **First name**: Nina
Student number: 417708 **Course of study**: Information Systems
Address: Dieckstraße 6, 48145 Münster
Title of the thesis: “Exploring High- and Low-Level Approaches for GPGPU Processing of Telescope Data.”

What is plagiarism? Plagiarism is defined as submitting someone else’s work or ideas as your own without a complete indication of the source. It is hereby irrelevant whether the work of others is copied word by word without acknowledgment of the source, text structures (e.g. line of argumentation or outline) are borrowed or texts are translated from a foreign language.

Use of plagiarism detection software The examination office uses plagiarism software to check each submitted bachelor and master thesis for plagiarism. For that purpose the thesis is electronically forwarded to a software service provider where the software checks for potential matches between the submitted work and work from other sources. For future comparisons with other theses, your thesis will be permanently stored in a database. Only the School of Business and Economics of the University of Münster is allowed to access your stored thesis. The student agrees that his or her thesis may be stored and reproduced only for the purpose of plagiarism assessment. The first examiner of the thesis will be advised on the outcome of the plagiarism assessment.

Sanctions Each case of plagiarism constitutes an attempt to deceive in terms of the examination regulations and will lead to the thesis being graded as “failed”. This will be communicated to the examination office where your case will be documented. In the event of a serious case of deception the examinee can be generally excluded from any further examination. This can lead to the exmatriculation of the student. Even after completion of the examination procedure and graduation from university, plagiarism can result in a withdrawal of the awarded academic degree.

I confirm that I have read and understood the information in this document. I agree to the outlined procedure for plagiarism assessment and potential sanctioning.

Münster, 16th September 2019

Nina Herrmann