Optimizing the Event Reconstruction of the Auger Engineering Radio Array for GPGPU

Vom Fachbereich C, Fachgruppe Physik der
Bergischen Universität Wuppertal zur Erlangung des akademischen Grades
Bachelor of Applied Science vorgelegte Arbeit

von

Marvin Gottowik

Erstprüfer:	Prof. Dr. Karl-Heinz Kampert
Zweitprüfer:	Prof. Dr. Francesco Knechtli

Contents

1 Introduction

Ultra high energy cosmic rays (UHECR) are the most energetic particles that have been observed. Their typical energy is much greater than the particle energy at the Large Hadron Collider or other particle accelerators. However, there is less than one particle per year and square kilometer at the highest energy, so huge detectors like the Pierre Auger Observatory are needed. The major task of the observatory is to find out, where this high energetic particles come from, and to understand the physical processes behind the acceleration to such high energies. To reconstruct the energy and direction of the particle from the detector data the "Offline Software Framework" is used. The aim of this thesis is to optimize the performance of the event reconstruction.

Especially in high-performance computing graphic cards are used to improve the performance of an application. This so-called "general purpose computing on graphics processing units (GPGPU)" can be a lot faster than computing on the CPU for certain problems and achieves a better performance per watt ratio. Therefore, the leading systems on the TOP500 list [1] of the most powerful computer systems in the world and on the Green500 list [2] of most energy-efficient supercomputers are using a combination of CPUs and GPUs.

In this thesis several performance improvements for the Auger Software Framework are described. These are achieved by enabling computation of selected intensive tasks on the GPU while keeping the interface of the modified files intact. This thesis is structured as follows. In chapter 1, a short introduction to cosmic rays, the Pierre Auger Observatory, the Auger Engineering Radio Array, and GPGPU is given. The Offline Software Framework is introduced in chapter 2. Chapter 3 describes several steps in replacing the FFTW with the cuFFT library and optimizations of the cuFFT usage. In chapter 4 the antenna pattern interpolation on the GPU with texture memory will be described. Finally, chapter 5 will summarize the results of this analysis and present its conclusions.

1.1 Cosmic Rays and the Pierre Auger Observatory

Cosmic rays are particles from outside our solar system with a wide range in energy and a flux between one particle per square meter and second for low energies to one particle per square kilometer and year at energies higher than 10 EeV. Fundamental questions like the sources of this ultra high energy cosmic rays and mechanisms for the acceleration of the particle to such energies are still unclear. Possible candidates are e.g. the acceleration in shocks in active galactic nuclei and gamma ray bursts [3].

When entering the earth's atmosphere the cosmic rays will hit an atmospheric nucleus and create secondary particles which will again interact with other nuclei in

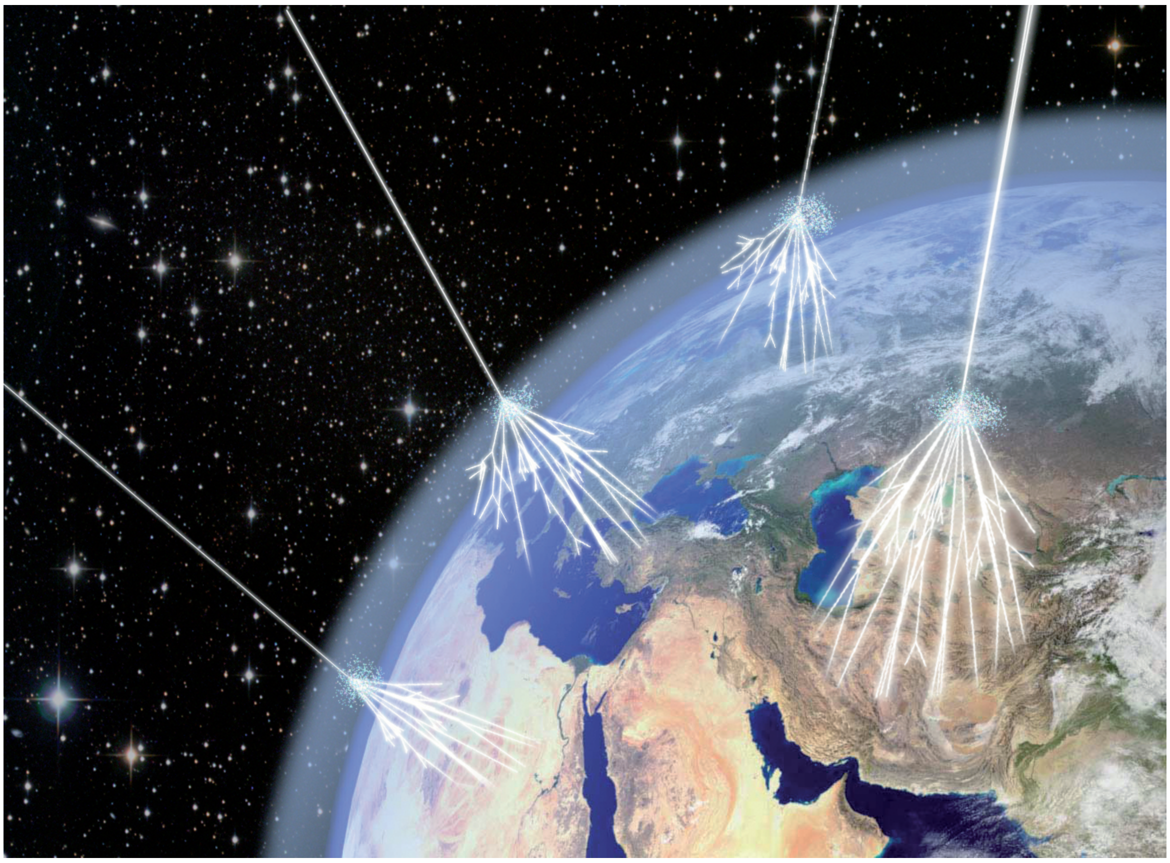


Figure 1.1: Visualization of air showers created by cosmic rays in the earth's atmosphere [4].

the atmosphere. This process is shown in figure 1.1. That way a cascade of particles, called "extensive air shower", is generated and can be detected on the surface. From the detected secondary particles direction and energy of the primary particle can be reconstructed.

Currently, the largest detector array is the Pierre Auger Observatory in Argentina with an area of 3000 km^2, which is larger than the size of the German federal state Saarland. Different methods are used to detect an extensive air shower.

The "Surface Detector (SD)" [5] consists of water filled tanks which will detect a passing particle via the emitted Cherenkov radiation. Combining the arrival time of the shower front from different SD stations the direction of the primary particle can be reconstructed.

Additionally to the SD, the "Fluorescence Detector (FD)" [6] is used in dark and clear nights. Extensive air showers create air fluorescence emitted by excited molecules while moving through the atmosphere. That allows to observe the air shower developing in the atmosphere and to measure the deposited energy. Using both SD and FD data for the reconstruction improves the results but the uptime of the fluorescence detectors is limited, as it operates only in clear and moonless nights.

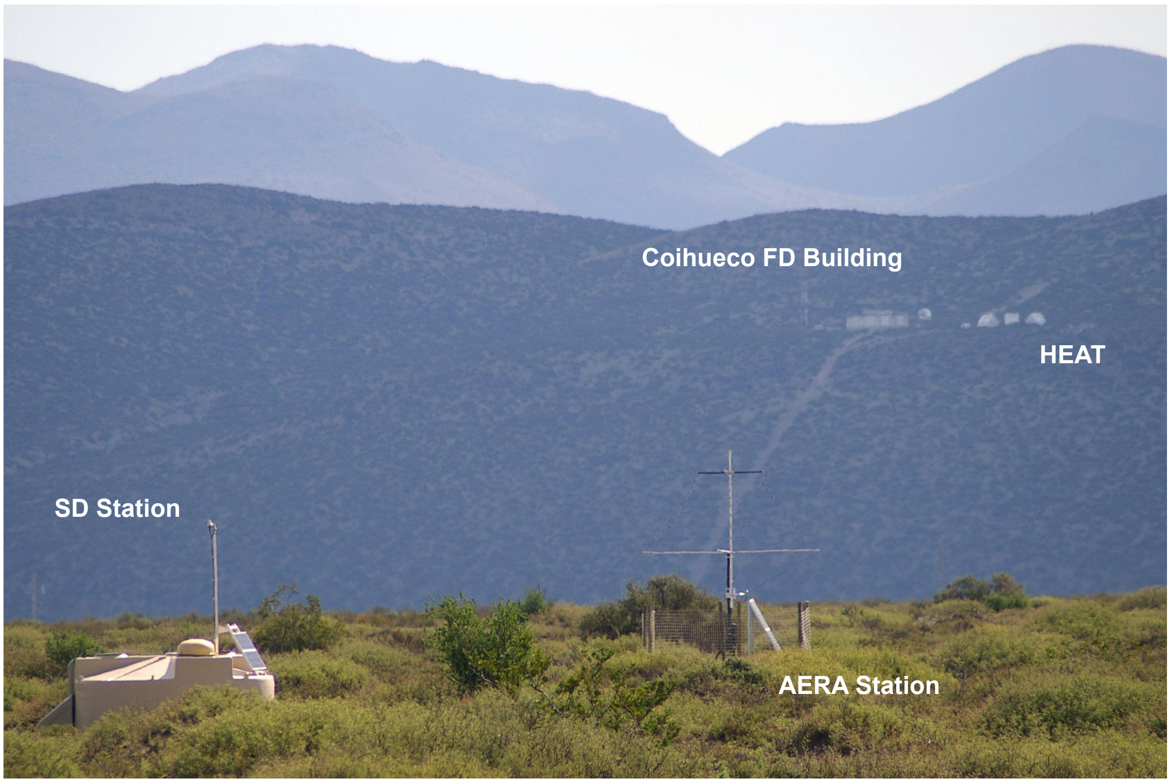


Figure 1.2: Photo of a SD station (foreground left), an AERA antenna (foreground right) and a FD building (background) [9].

1.2 The Auger Engineering Radio Array

In 2010, the Pierre Auger Observatory was extended by the Auger Engineering Radio Array (AERA) to detect radio signals from air showers. This radiation is emitted by two different sources. On the one hand electrons and positrons in the shower are deflected by the magnetic field of the earth. This effect is called geo-magnetic emission and is the dominant effect [7]. On the other hand there is a negative charge excess in the shower front due to knocked out electrons from the air molecules and annihilated positrons in the shower front. AERA measures the radiation in a frequency range of 30-80 MHz [8].

Observing the radio emission can provide information about the development of the air shower as the FD does, but it is not limited to dark and clear nights. An example of an AERA antenna and a SD Station can be seen in figure 1.2.

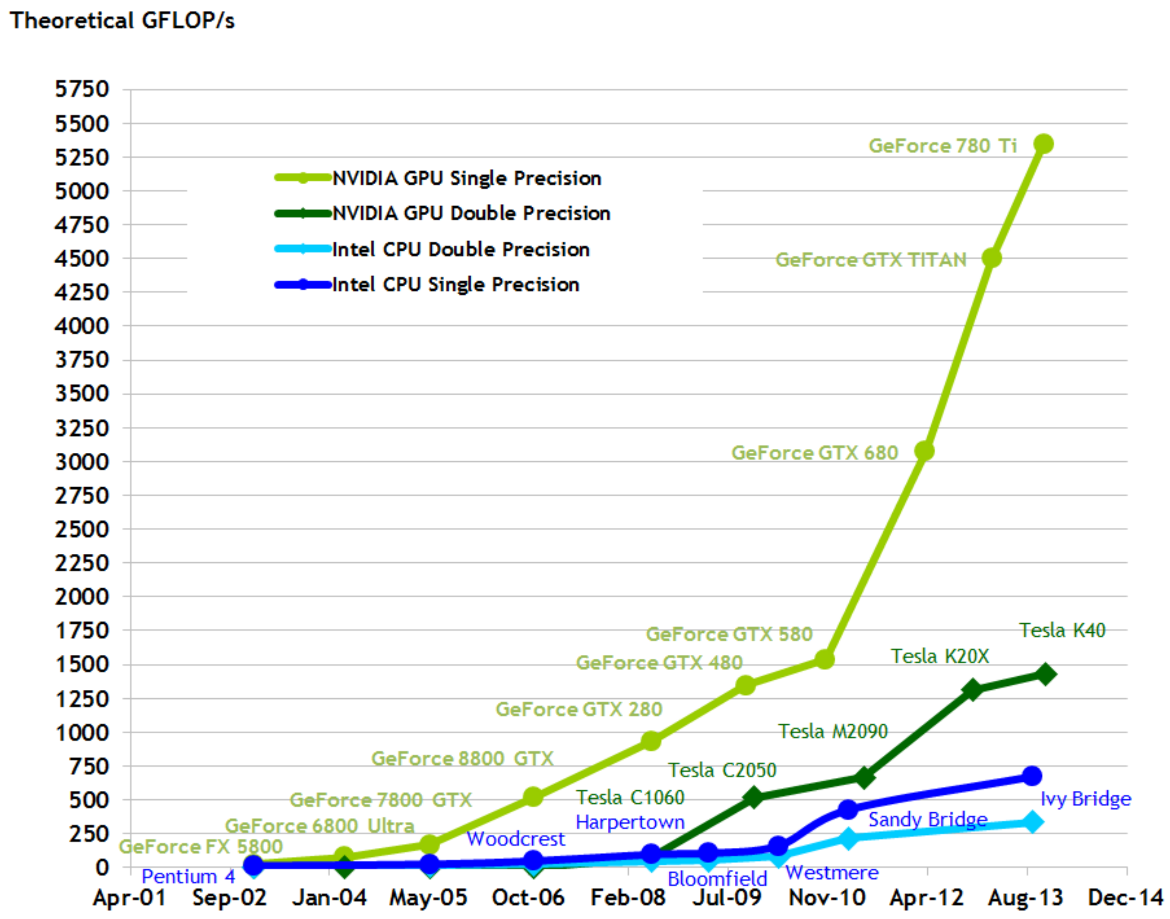


Figure 1.3: Development of the theoretical floating point operations per second for CPUs and GPUs [10].

1.3 General Purpose Computing on Graphics Processing Units

General purpose computing on graphics processing units means the utilization of graphic cards to perform computations beyond normal graphic computations. In figure 1.3 the theoretical maximum performance of selected CPUs and GPUs is shown in floating point operations per second (FLOP/s) as a function of time. Since 2002 GPUs outperform CPUs and are approximately ten times faster today.

The wide difference can be explained by the structure and functional differences of a CPU and a GPU. CPUs are constructed for single-threaded code where many different instructions are performed on the same data set with many branch commands. A reduction of the execution time can not be achieved by an increasing clock frequency so other techniques have to be used. Typical examples are reordering instructions for a better workload of the CPU and caches to reduce time for loading data. In figure 1.4 a schematic view of the components of a CPU and GPU is shown. One can see

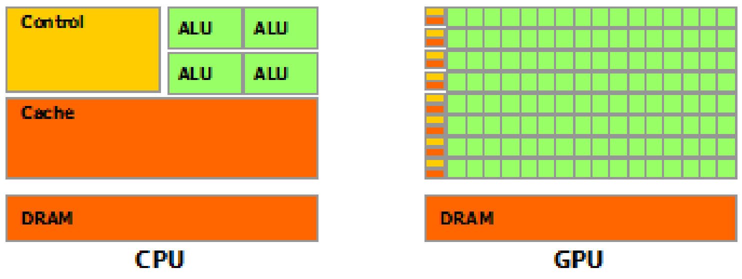


Figure 1.4: Schematic view of a CPU (left) and a GPU (right). Each box represents the space on the chip used for a specific function indicated by the color [10].

that a CPU owns only a few arithmetic logical units (ALU) which can work totally independent. Much space on a CPU is spent for cache and controlling units. In contrast to that, a GPU consists of way more ALUs and only small cache and controlling parts. The ALUs are grouped in arrays of Streaming Multiprocessors (SM). All cores of a SM must perform the same instructions but on different data. Therefore data parallel problems like e.g. a matrix vector multiplication will be much faster on a GPU than on a CPU.

For programming on the GPU different software packages can be used. The most important ones are OpenCL [11] and CUDA (Compute Unified Device Architecture) [10]. OpenCL is a framework that enables parallelism on CPU, GPU and other processors. CUDA is a parallel computing platform created by NVIDIA and is only supported by NVIDIA GPUs, but achieves better performance compared to an OpenCL program on a NVIDIA GPU.

In this thesis only CUDA is used. In CUDA, CPUs are called hosts and GPUs are called devices. A program can compute on the CPU as usual and use the GPU only for specific code parts.

CUDA C extends the C/C++ programming language by new keywords and syntax elements for executing code on a GPU. To illustrate this, a small "single precision $a \cdot \vec{x} + \vec{y}$ (SAXPY)" code example is shown in listing 1.1 [12]. This code takes two vectors $\vec{x}$, $\vec{y}$ and a scalar a and calculates a scalar multiplication and vector addition for every component.

Listing 1.1: SAXPY example code in CUDA

```
__global__
void saxpy(int n, float a, float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}

int main()
{
```

```
10     ...
11     int N = 1<<20;
12     cudaMalloc((void **)&d_x, sizeof(float)*N;
13     cudaMalloc((void **)&d_y, sizeof(float)*N;
14     cudaMemcpy(d_x, x, N, cudaMemcpyHostToDevice);
15     cudaMemcpy(d_y, y, N, cudaMemcpyHostToDevice);
16
17     // Perform SAXPY on 1M elements
18     saxpy<<<4096, 256>>>(N, 2.0, x, y);
19
20     cudaMemcpy(y, d_y, N, cudaMemcpyDeviceToHost);
21     ...
22   }
```

A function that is executed on a GPU is called a kernel and is defined by adding `__global__` (line 1) or `__device__` to the function declaration. A function executed on the CPU function can optionally be mark by `__host__`. If no specifier is used the function will be a host function (line 8).

For calling the global function `saxpy()` from the host function `main()` the syntax `saxpy<<<nGrid, nThread>>>()` (line 18) has to be used. This syntax will launch a kernel which is divided in `nGrid` blocks, each block consists of `nThread` threads. Threads and blocks have internal IDs. They can be used to calculate a unique ID for every thread in a kernel (line 4).

All threads in the same block perform identical instructions. Figure 1.5 shows a CUDA program with eight blocks executed on two different GPUs. Each SM can execute one block. Blocks will be executed parallel on the SM so a GPU with more SM will automatically execute the code in a shorter time period than a GPU with less SM.

Kernel function calls are asynchronous, so they may return before the device has completed its execution. To synchronize CPU and GPU execution the function `cudaDeviceSynchronize()` has to be called. All CUDA runtime functions including kernel launches set an internal error code, hence it should be checked after every call. This does not work for asynchronous calls so they might only be noticed later in the program.

Before calling the kernel, memory has to be allocated (line 12,13) and data have to be copied from CPU to GPU (line 14,15). After the kernel has finished, the data need to be copied back to the CPU (line 20). Since memory allocation and copy processes are slow, one should reuse allocated memory as often as possible and reduce the number of memory copies to a minimum.

In the next chapter the reconstruction software "Offline" is introduced and the main bottlenecks are identified. They will then be reimplemented for computation on the GPU using CUDA.

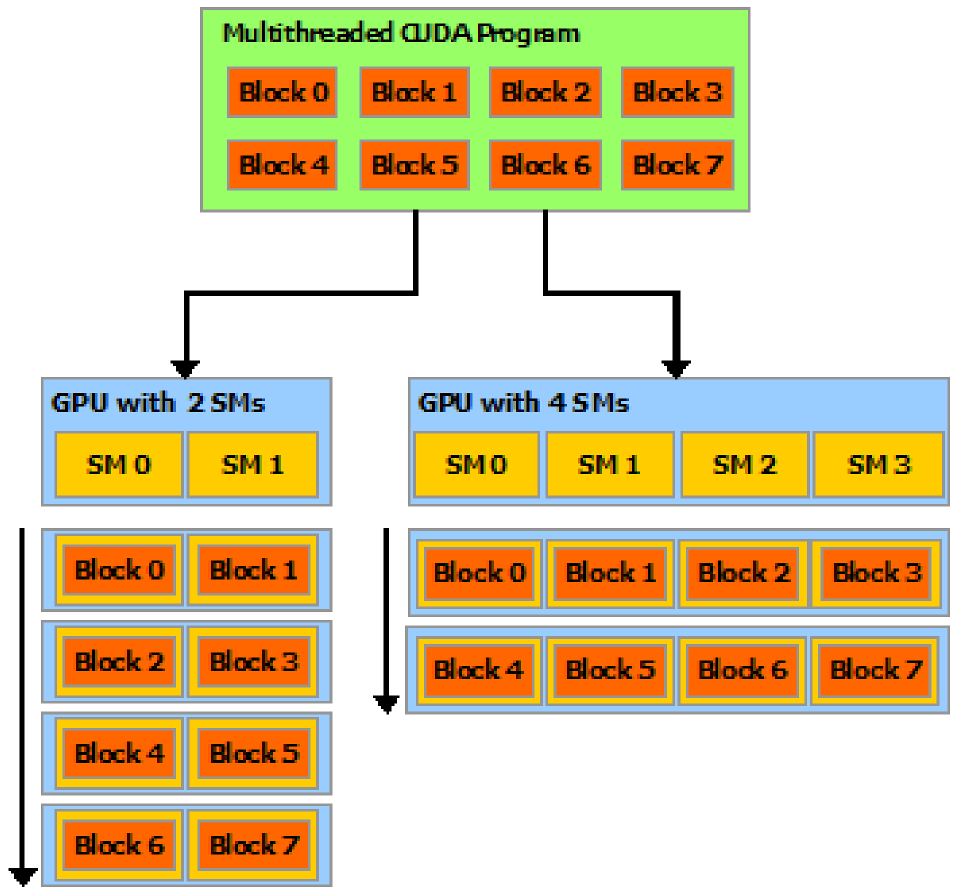


Figure 1.5: Execution of a multithreaded CUDA program on two GPUs with a different number of Streaming Multiprocessors [10].

2 The Offline Software Framework

Offline [13] is the C++ Framework developed for the reconstruction of the fluorescence and surface detector and was extended for radio-detection starting 2006 [14]. This section will briefly describe the data structure of Offline and the AERA event reconstruction. Furthermore, results of a performance profiling of the application are presented.

2.1 Structure of the Offline Framework

The Offline framework can be separated in three different parts as shown in figure 2.1. On the one side, there is the detector description handling information on the detector like the used hardware. On the other side, there is the detected event data from individual detectors. The third part are the algorithms from different modules which combine the event data with the detector description to reconstruct the physical process. Each module does one processing step of the reconstruction. The order of the modules can be changed in the `ModuleSequence.xml` file. The sequence used in this thesis is shown in appendix A.

Additionally, a variety of utility classes are used for e.g. unit conversions, error logging and Fourier transformations. Furthermore, there is a bundle of configuration files for the fine tuning of individual modules.

The event data are stored in the `Event` class. This class contains among other the event data from the surface, fluorescence and radio detectors. A closer look at the `REvent` class is presented in figure 2.2. The `REvent` handles the active `Stations` and `Channels` of the event each in a corresponding class. FFT data are stored on both levels in a `FFTDataContainer`. Data stored on the `Stations` represents the physical electric field

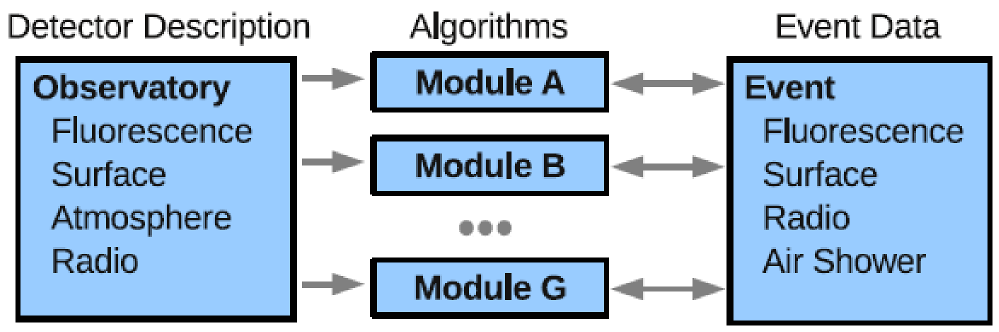


Figure 2.1: Separation of data and algorithm of the Offline Framework [15].

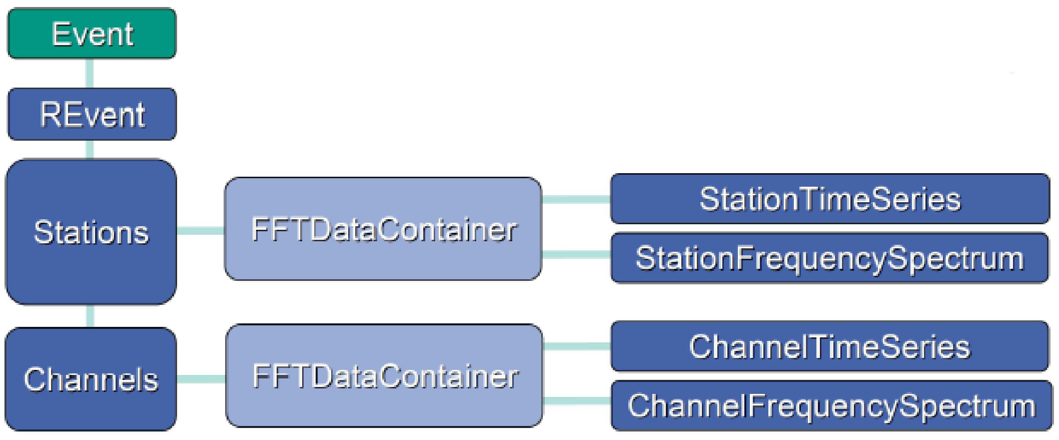


Figure 2.2: Structure of the REvent containing the stations and channels of the event [14].

devoid of any detector influence. The `Channel` data represents the measured data of one antenna of a station.

2.2 Radio Event Reconstruction

This section shortly describes the individual steps necessary for the reconstruction of a radio event. First, various modules are used to correct the impact of the used hardware and to enhance the signal of the two antenna channels. After that, the electric field vector $\vec{E}(t)$ needs to be extracted from the measured voltage trace. The challenge here is that the incident direction is calculated from the signal arrival times. But the arrival time depends on the antenna response of the station which itself depends on the direction.

Thus, direction and antenna response are reconstructed iteratively from a given initial direction. The antenna pattern is interpolated for this direction (see chapter 4) and the electric field vector is computed. Then, the arrival time is updated using the maxima of its Hilbert envelope (see chapter 3) leading to a new shower direction. This loop is repeated until a certain convergence criterion holds. If this loop does not converge after 10 steps the event will be skipped. This process is shown in figure 2.3.

The energy of the primary particle is correlated with the electric field strength at the core and the lateral distribution depends on the mass of the primary particle [15].

2.3 Performace Profiling of Offline

Profiling an Offline run provides detailed information on where most time is spent in the code. Various profilers are available, each with its own advantages and disadvantages. Here the Linux kernel profiler “perf” [17] is used for analyzing the CPU code and the “NVIDIA Visual Profiler (nvvp)” [18] for the GPU code.

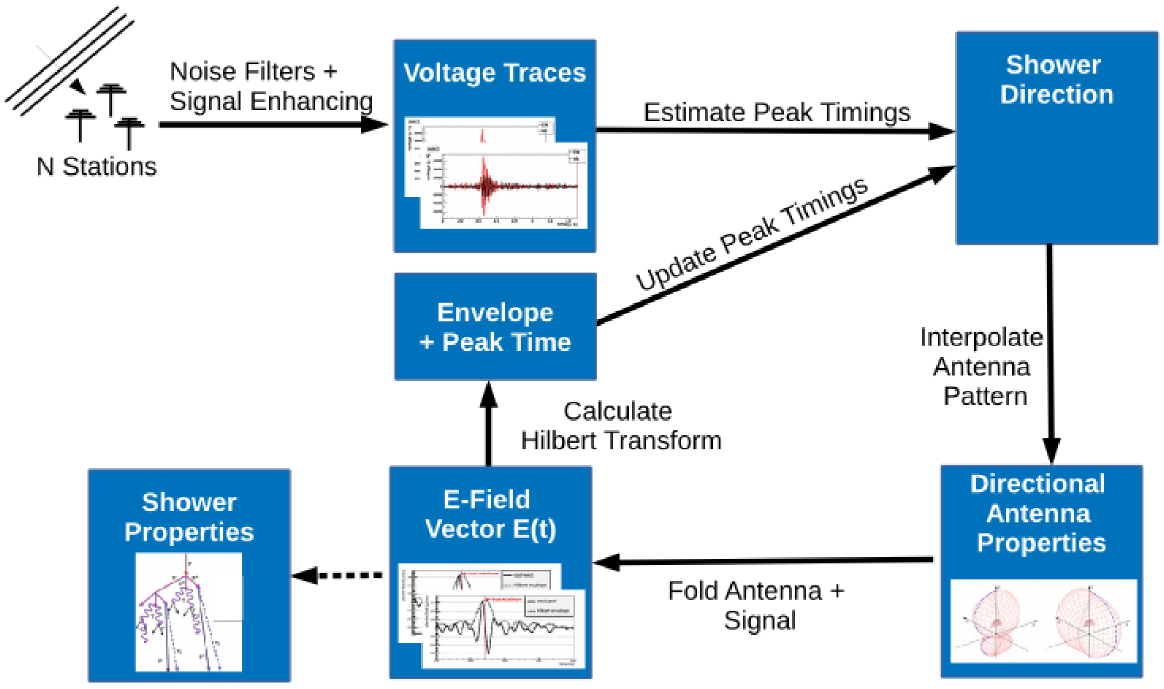


Figure 2.3: Sketch of the radio event reconstruction. Shown is the iterative reconstruction of incident direction and antenna response including calculation of the Hilbert envelope and interpolation of the antenna pattern [16].

For a correct profiling and analysis of an Offline run one has to know how much time is spent on the reconstruction and how much time is "overhead" for the initialization of Offline. To determine the overhead, test files are created in which the same event is written several times. As it is always the same event which is reconstructed the mean reconstruction time per event should be the same for all file sizes. Due to the overhead for the initialization of the Offline framework, which is a constant for a complete reconstruction, the time per event depends on the number of events per file.

In the end of an event reconstruction, Offline prints a small summary containing information about the time spent in each module and the total real time spent for the run. Real time means the time which passed between start and finish of the program. This time includes time spent by other processes and time where the system is waiting for e.g. I/O operations. Another possibility to measure execution time is using only the time where the CPU was working. This does not include the computing time on GPU, hence it is not sufficient in this context.

Here, the total time will be used to evaluate the impact of the individual changes, as it is the easiest accessible metric. The same file has to be reconstructed several times

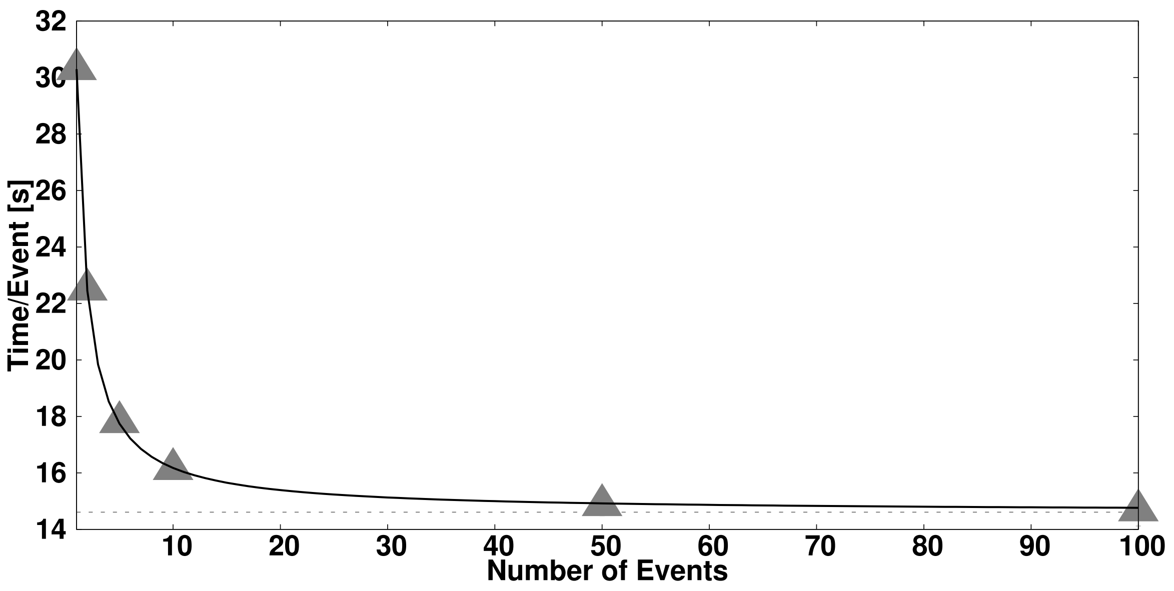


Figure 2.4: Average time t needed to reconstruct a specific event for different number x of events per file. The function $f(x) = A/x + B$ is fitted to the data. The dashed line shows the reconstruction time per event for infinite events per file B.

and the minimal time is used. Since parallel operation of the system can only increase the used time, the minimal time is the best estimation of the idealized execution time. Two different test systems are used for the analyzes, a desktop and a cluster system. Some important technical data of both system and their CPU and GPU are listed in appendix C.

In figure 2.4 the time per event is plotted as a function of events per file. The function $f(x) = A/x + B$ is fitted to the data where B is the time per event and A the overhead time. For small files the overhead has a great impact on the average time per event. Hence bigger files are needed to analyze the improvement on the reconstruction. For 50 events per file, the difference between the extrapolated time per event B and the measured time per event is less than 3 %. Therefore this file will be used for the analysis.

Using the profiler perf, the following two hotspots can be identified in a profiling run:

1. Around 15 % of the time is spent on calculating Fourier transformations. FFTs are used among others by the `RdStationSignalReconstructor` module as a part of the Hilbert envelope for updating the arrival time of the signal. But other radio modules also need them as they work on the time series and frequency spectrum of a signal.

2. Interpolations of the antenna patterns need nearly 25 % of the time. This is done in the `RdAntennaChannelToStationConverter` module.

If this two hotspots could be removed the time per event can at best decrease by 40 %. In the following two chapters GPGPU is used to eliminate these hotspots.

3 Fourier Transformations in Offline

In Offline, FFT data are stored in a class named `FFTDataContainer`. This class contains both the time series and the frequency spectrum of a signal and marks which one has been modified latest. To calculate only the necessary Fourier transformations a FFT is only computed when the `FFTDataContainer` is asked for the invalid representation. For calculating a FFT on CPU, the FFTW [19] library is used. This library is wrapped by an object orientated wrapper in which different classes for the different FFT types are defined. For e.g. a one dimensional complex to complex Fourier transformation an object of the class `fft1d` is created and its `fft()` method is called.

Fourier transformations can also be calculated efficiently on the GPU with the cuFFT [20] library. This chapter starts with a comparison between both libraries and describes different steps in replacing the FFTW with the cuFFT to achieve the best performance.

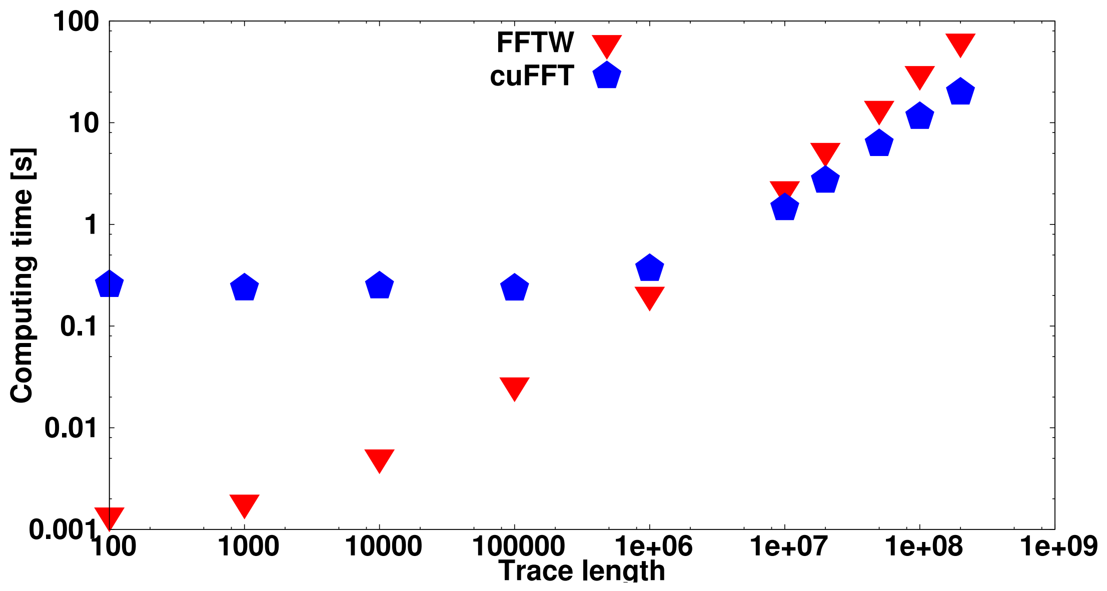
3.1 Comparison between cuFFT and FFTW

The NIVIDIA cuFFT library provides an interface alike the FFTW for computing FFTs on the GPU. For computing a FFT, a “plan” has to be created first which determines dimension, size and transformation type. Depending on the transform size different algorithms are chosen for the fastest execution of the specified transformation. After that, memory is allocated on the GPU and the input values are copied. Then the FFT can be executed and the results are copied back to the CPU.

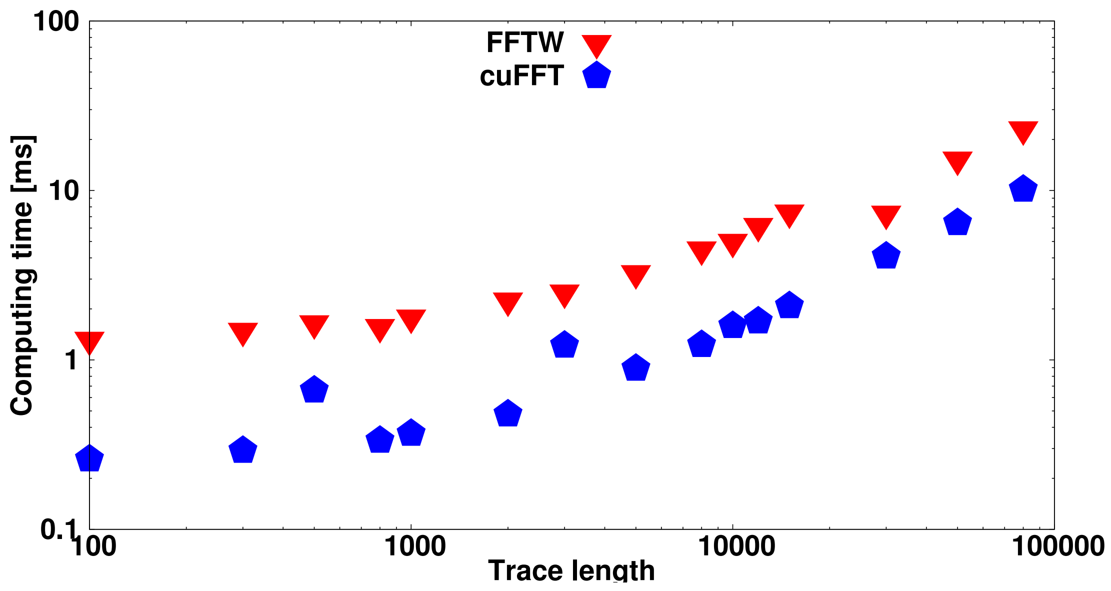
To compare the performance of both libraries, Fourier transformations for different sizes are computed on CPU and GPU and the execution time is compared. The result is shown in figure 3.1. In figure 3.1a the total execution times of both versions are plotted for trace lengths between 100 and 1×10^9 elements. If just a single FFT is computed, the trace length must be bigger than 1×10^6 to make the GPU execution faster than the execution on CPU due to initialization of the GPU and cuFFT library at first API call and the needed memory allocation on GPU.

However, in Offline many FFTs are computed. Hence the initialization time can be negated and memory will be allocated only once and then reused for all FFTs of the same size. Therefore, the comparison is repeated without measuring the initialization and allocation time on the GPU for trace lengths in a range of 100 to 100 000. Time for copying of memory is still included. This range includes the actually used sizes in Offline of 10 240 and 40 960 elements. As shown in figure 3.1b the GPU version is always faster in this setup.

Thus, a global replacement of the FFTW with the cuFFT is expected to yield a performance improvement, if repeated memory allocation can be avoided.



(a) Execution time including the automatic initialization of the GPU at first API call.



(b) Execution time without initialization of the GPU for typical trace lengths used in Offline. GPU is manually initialized, and the memory is allocated before starting the time measurement.

Figure 3.1: Comparison of the computing time for a 1D complex to complex Fourier transformation with cuFFT and FFTW library on the desktop test system.

3.2 Integration of cuFFT in Offline

To enable computation of the FFTs on GPU, the cuFFT is wrapped similar to the FFTW. Hence, only the included header file of the `FFTDataContainer` and the `FFTDataContainerAlgorithm` has to be changed. In the following, various steps of the

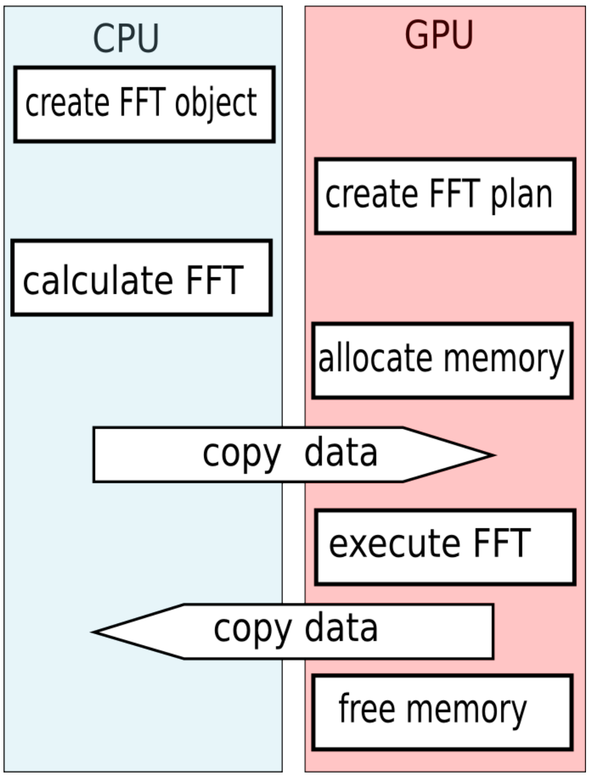


Figure 3.2: Sketch of the individual steps for a FFT on the GPU.

cuFFT usage and their performance are described. Each step includes the previous steps, the time per event is always compared to the baseline.

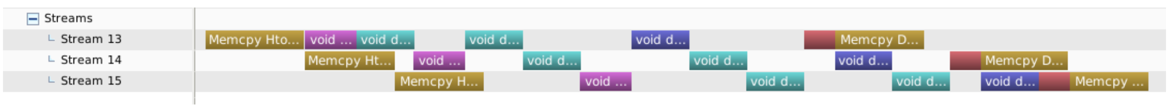
Naive Implementation

First, a naive implementation of the wrapper is written. This version will allocate and free the memory on the GPU for every transformation. The individual steps of its execution are shown in figure 3.2. The left box contains steps performed on the CPU and the right box shows steps performed on the GPU. Creating an FFT object creates the corresponding cuFFT plan. When the `fft()` method is called, memory is allocated and input data are copied to GPU before the execution. After the execution the result is copied back to the CPU and memory is freed.

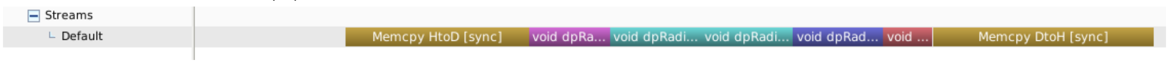
With this implementation the time per event is reduced by around 3.5 % from 14.7 s to 14.2 s on the desktop system. However, on the cluster the time per event has increased from 13.4 s to 17.2 s by 28 %. Notice that this does not correspond to the above shown FFT comparison. Also on the desktop system the cuFFT should be slower than the FFTW if memory is always allocated and freed. Thus, the FFTW wrapper seems to limit the performance of the FFTW in Offline.

Implementation with static memory on the GPU

In the next step, static memory is used for storing data on the GPU to reduce the number of expensive memory allocation. Before copying the data from the CPU to



(a) Parallel execution of the 3 independent FFTs.



(b) Serial execution of the 3 independent FFTs.

Figure 3.3: Screenshots from the nvvp for serial and parallel execution of the FFT, the shown block in the serial figure is repeated three times.

the GPU it checks if the current array on the GPU is big enough. If this is true the memory is directly copied, else the memory is reallocated with the needed size.

This implementation reduces the time per event on the desktop system from 14.7 s to 13.4 s by approximative 10 % compared to the baseline but still increases the time on the cluster from 13.4 s to 13.7 s by approximately 2 %.

It is noteworthy that the reconstruction time per event has still increased on the cluster. The cluster version of figure 3.1b looks similar to the shown desktop plot, thus the cuFFT should be faster than the FFTW. Possible explanations are internal compiler optimizations which are applied in Offline but not in the used test programs.

Static memory can be theoretically dangerous if more than one `fft` object exists, and their execution functions are called simultaneously, since they are sharing their data memory. However, this is not possible in the current configuration. The wrapper uses blocking memory copy hence a `fft()` method call returns after the complete execution. This makes it impossible that two `fft()` methods are executed in parallel.

Dynamic switch between cuFFT and FFTW

In this version a new wrapper file is invented allowing to use both libraries depending on the trace length. The cuFFT is used for the bigger trace length of 40 960 elements but the FFTW for 10 240 elements. This leads to the situation that the FFTW library is compiled by the NVIDIA CUDA Compiler Driver (nvcc) [21]. There is a bug in the used FFTW version 3.3.3 which will enable quad_precession although it is not supported by the nvcc. Thus the FFTW needs to be patched until this is fixed in a newer version. The patch file is included in appendix B.

Now the time per event on the cluster is again on baseline level with 13.4 s but on the desktop the improvement has reduced to 8 % and a time per event of 13.6 s compared to the 14.7 s of the baseline.

FFTs on station level

The Fourier transformations on station level consist of three independent sub-FFTs, one for each component of the electric field vector. Here different techniques for their execution are compared. First, there is the current serial execution with one fft object and three execution calls. The second version uses the cuFFT batch mode for computing the FFTs. Third, they are executed in parallel using different streams and asynchronous memory copy for each sub-FFT.

In figure 3.3 the difference between the parallel and serial execution is depicted by screenshots from the nvvp. In the serial version all internal kernel launches and memory copies are executed in order one after another. In the parallel execution, memory copies are overlapped by computing due to the asynchronous copy calls. The different kernels are executed out of order but do not overlap, because a single kernel always occupies the GPU entirely.

Small test programs are used to measure the total time of the three FFTs. They are executed several times and the minimal execution times are compared. For the parallel execution the minimal time is 224.1 ms, the serial execution needs minimal 226.7 ms and the batch mode 229.9 ms. The differences are small, nevertheless parallel version is tested in Offline as it is the fastest one. On the desktop system this change has no impact on the reconstruction time per event, whereas on the cluster the time per event has increased from 13.4 s to 13.5 s.

Upsampling Channel trace length to next power of 2

The cuFFT and also the FFTW use different algorithms depending on the trace length, the best performance of the cuFFT is achieved if the trace length is a power of 2 [20]. Thus the `RdChannelUpsampler` module is extended for upsampling the trace to the next power of two instead of a fixed factor. Again small test programs are used to measure the time for unpadded and padded execution with both libraries.

The minimal execution time of the FFTW has increased from 150 ms to 158 ms due to the padding, whereas it has reduced from 555 ms to 201 ms when using the cuFFT. Time for memory allocation is included in the measured cuFFT times, hence it seems to be slower than the FFTW.

Upsampling the Channel trace to a power of two has reduced the time per event on the desktop from 14.7 s to 12.5 s by 15 % and on the cluster from 13.4 s to 12.8 s by 5 %.

The time per event of all steps are shown in figure 3.4. Best performance is achieved when using static memory and the modified upsampling to a power of two. The switch between the FFTW and cuFFT library and different execution modes for station FFTs have no effect or partially increased the time per event. Thus, these steps are removed. The ratio of the baseline and improved implementation, called speedup, is used to evaluate the impact of the improvement. The resulting change in the time per event and the calculated speedup of the final implementation are shown in figure 3.5.

3.3 Hilbert Envelope

Several FFTs are used in context of the calculation of the Hilbert envelope. This allows a significant speedup using a suited implementation on the GPU, which will be described in this section.

The radio signal $x(t)$ is an oscillating signal as shown by figure 3.6. For the event reconstruction the time with the maximal electric field strength is needed. This does not necessarily correspond with the maximal value of the reconstructed field because

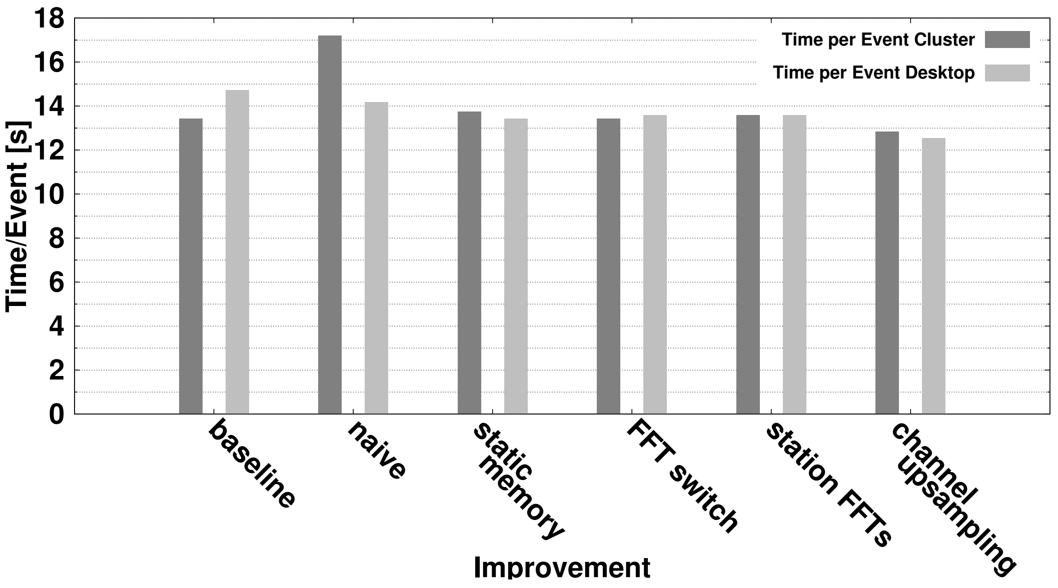


Figure 3.4: Time per event for all FFT improvement steps, each step is based on the previous.

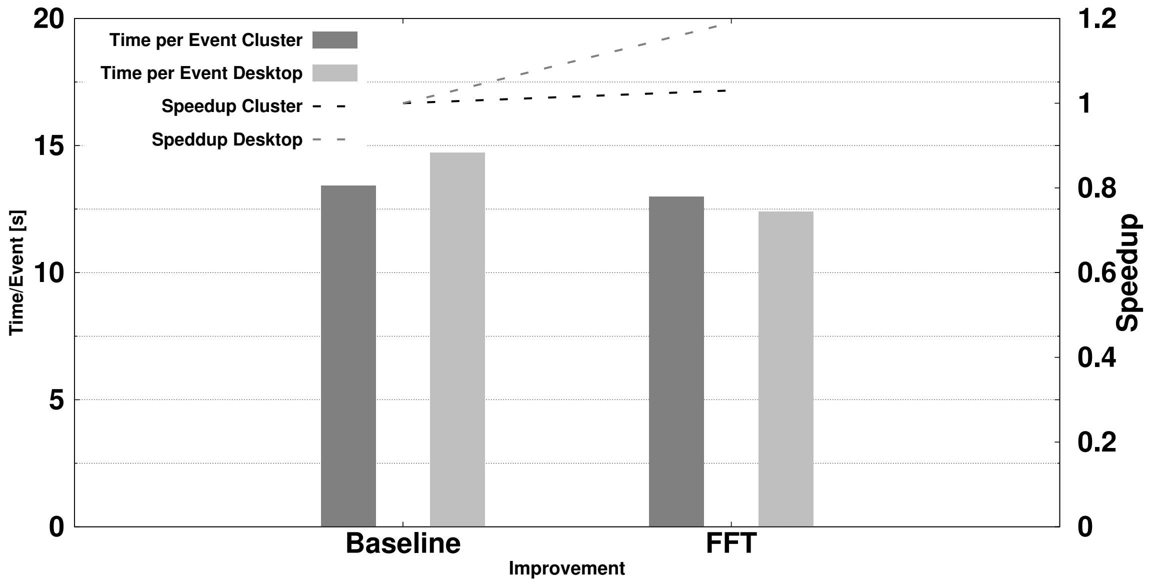


Figure 3.5: Comparison of the time per event and the Speedup after final implementation of cuFFT.

of the possible zero crossing of the electric field of the incoming electric-magnetic wave at the position of the true maximum. An estimation is given by the Hilbert envelope $E(t)$ [15].

The Hilbert envelope $E(t)$ of a signal $x(t)$ is defined as

$$E(t) = \sqrt{x^2(t) + (H(x(t)))^2} \tag{3.1}$$

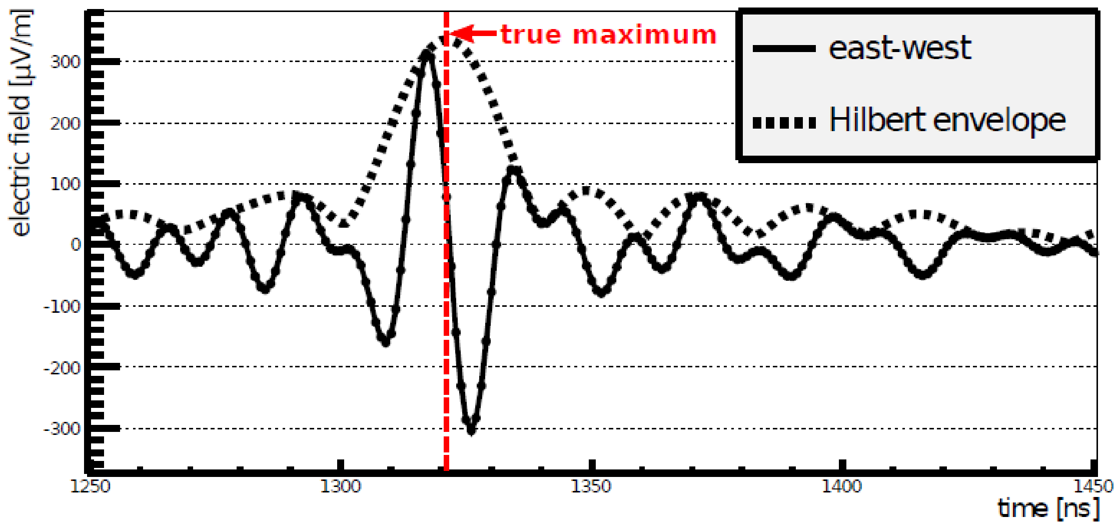


Figure 3.6: Example of a east-west component of a reconstructed electric field vector and the Hilbert envelope [15].

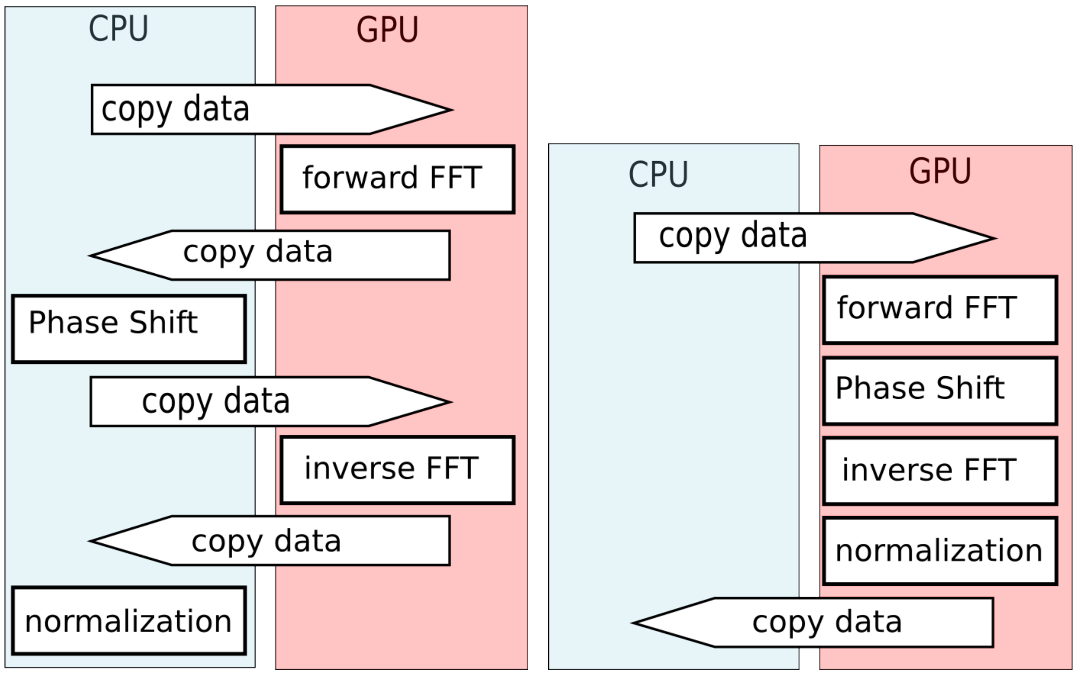
where $H(x(t))$ is the Hilbert transformed of $x(t)$. The Hilbert transformation of a signal $x(t)$ shifts the negative frequencies by $90°$ and the positive frequencies by $-90°$. In this context negative frequencies are frequencies left from the median of the frequency spectrum, positive frequencies are right from it. To compute the envelope a time signal is Fourier transformed to the frequency representation. Then the frequencies are shifted and converted back to the time domain with an inverse Fourier transformation.

In the current implementation the data are copied to the GPU and back for the forward FFT and again for the inverse FFT. Phase shift and square root are calculated on CPU. However, it is more efficient to compute the whole envelope on the GPU for two reasons. First, the phase shift and the normalization are similar instructions operating on multiple data and can thus be calculated efficiently in parallel. Second, only half of the memory copies are necessary if the total calculation is done on the GPU. The difference of both implementations is shown in figure 3.7.

With the modified Hilbert Envelope, 11.6 s are needed per event on the desktop system which is an reduction of 7 % compared to the best cuFFT from above. On the cluster one event needs 12.4 s which is 5 % faster than before.

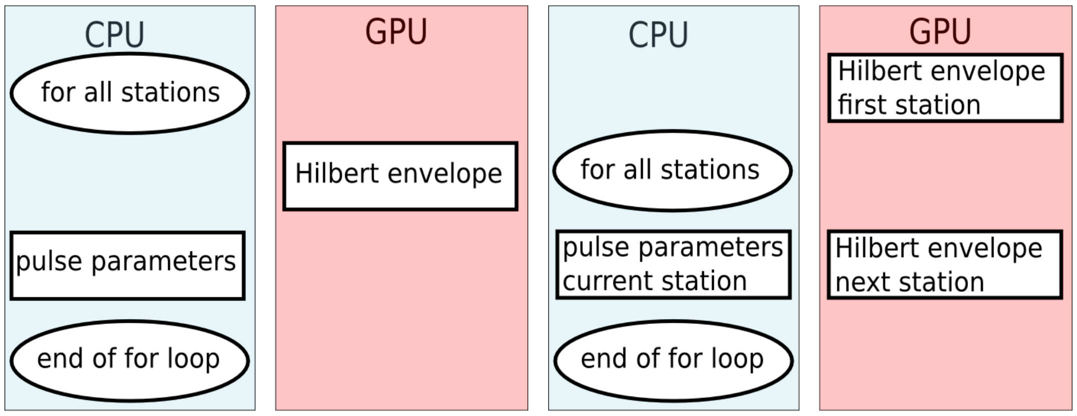
In a further step CPU-GPU parallelism is added. So far, the CPU is working in the `RdStationSignalReconstructor` until the Hilbert envelope is computed. During that period the GPU is idle. Then it starts computing the envelope while the CPU is idle waiting for the result.

To keep both working at he same time the structure of the module is changed. Currently, the `RdStationSignalReconstructor` module contains a loop over all stations of an event. For each station the Hilbert envelope is computed and the pulse parameters on the Station level are determined. With the new structure the envelope of the first station is computed outside the loop. In the loop the pulse parameters of this station



(a) Implementation with just FFTs on GPU. **(b)** Implementation with total execution on GPU.

Figure 3.7: Structure of the Hilbert Envelope execution in different versions.



(a) Old structure, no CPU-GPU parallelism. **(b)** New structure with CPU-GPU parallelism.

Figure 3.8: Structure of the Signal Reconstruction module.

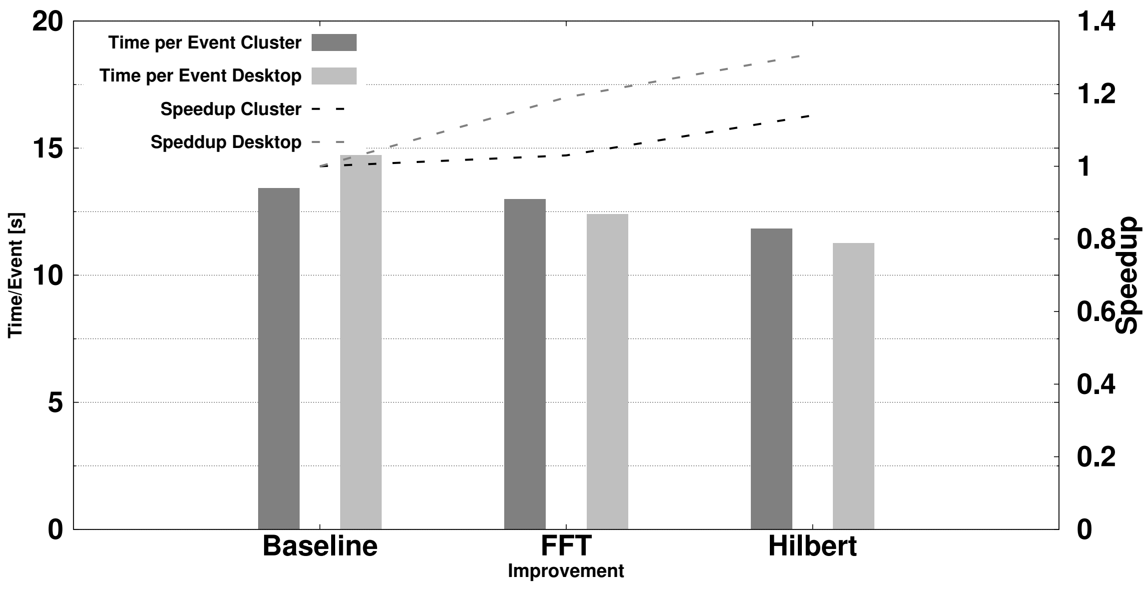


Figure 3.9: Comparison of the time per event and the Speedup after final implementation of cuFFT and improved Hilbert envelope.

are determined on CPU while the envelope of the next Station, if it exists, is calculated on the GPU. Both structures are depicted in figure 3.8. Additionally, a Hilbert envelope on station level consists of three independent sub-envelopes like the station FFT. Each one can be calculated parallel on the GPU.

Combining both effects the time per event has reduced to 11.8 s (10 %) on the cluster and to 11.3 s (-9 %) on the desktop system. The new times per event and the achieved speedup with the improvements for FFTs and Hilbert envelopes are shown in figure 3.9.

4 Interpolation of the antenna pattern

The response $V(t)$ of an antenna to an incoming electric field $\vec{E}(t)$ depends on the incident direction of the field. This can be parameterized with the vector effective length (VEL) $\vec{H}$ of the antenna in the frequency domain by

$$\mathcal{V}(\omega) = \vec{\mathcal{H}}(\omega) \cdot \vec{\mathcal{E}}(\omega) \tag{4.1}$$

where $\mathcal{V}$, $\vec{\mathcal{H}}$ and $\vec{\mathcal{E}}$ are the Fourier transformed of V, $\vec{H}$ and $\vec{E}$. In a spherical coordinate system with zenith angle θ and azimuth angle ϕ this can be written as

$$\begin{aligned} \mathcal{V}_1(\omega) &= \mathcal{H}_{1,\theta}(\omega)\mathcal{E}_\theta(\omega) + \mathcal{H}_{1,\phi}(\omega)\mathcal{E}_\phi(\omega) \\ \mathcal{V}_2(\omega) &= \mathcal{H}_{2,\theta}(\omega)\mathcal{E}_\theta(\omega) + \mathcal{H}_{2,\phi}(\omega)\mathcal{E}_\phi(\omega) \end{aligned} \tag{4.2}$$

if the $\vec{e}_r$ points in the incident direction of the electric wave. Here, $V_{1,2}$ are the voltages measured in the two antennas of a AERA station. One antenna is east-west and the other north-south polarized. Since the shower signal is a transverse wave the $\mathcal{E}_\mathrm{r}(\omega)$ component is always zero. Solving this equation system for the electric field leads to [8]

$$\begin{aligned} \mathcal{E}_\theta(\omega) &= \frac{\mathcal{V}_1(\omega)\mathcal{H}_{2,\phi}(\omega) - \mathcal{V}_2(\omega)\mathcal{H}_{1,\phi}(\omega)}{\mathcal{H}_{1,\theta}(\omega)\mathcal{H}_{2,\phi}(\omega) - \mathcal{H}_{1,\phi}(\omega)\mathcal{H}_{2,\theta}(\omega)} \\ \mathcal{E}_\phi(\omega) &= \frac{\mathcal{V}_2(\omega) - \mathcal{H}_{2,\theta}(\omega)\mathcal{E}_\theta(\omega)}{\mathcal{H}_{2,\phi}(\omega)}. \end{aligned} \tag{4.3}$$

Hence, the VEL has to be known to reconstruct the electric field vector from the measured voltage trace. This is achieved by using simulated or measured antenna response for discrete values of ϕ, θ and ω and interpolate this pattern for the needed directions and wavelengths. This is done in the `AntennaType` class called from the `RdAntennaChannelToStationConverter` module. Examples of an antenna response for a fixed frequency are shown in figure 4.1.

4.1 Interpolation with texture memory

Interpolations are one of the fundamental tasks of a GPU as they are needed e.g to display textures in games. A special memory area is used allowing interpolation on dedicated hardware to make it particularly fast. In CUDA this memory is called texture memory.

In the baseline implementation the antenna pattern is a struct containing vectors of the simulated ϕ, θ and ω values. A map called antenna response is used to get the VEL for a specific response key. This response key is a tuple of integers, each one the

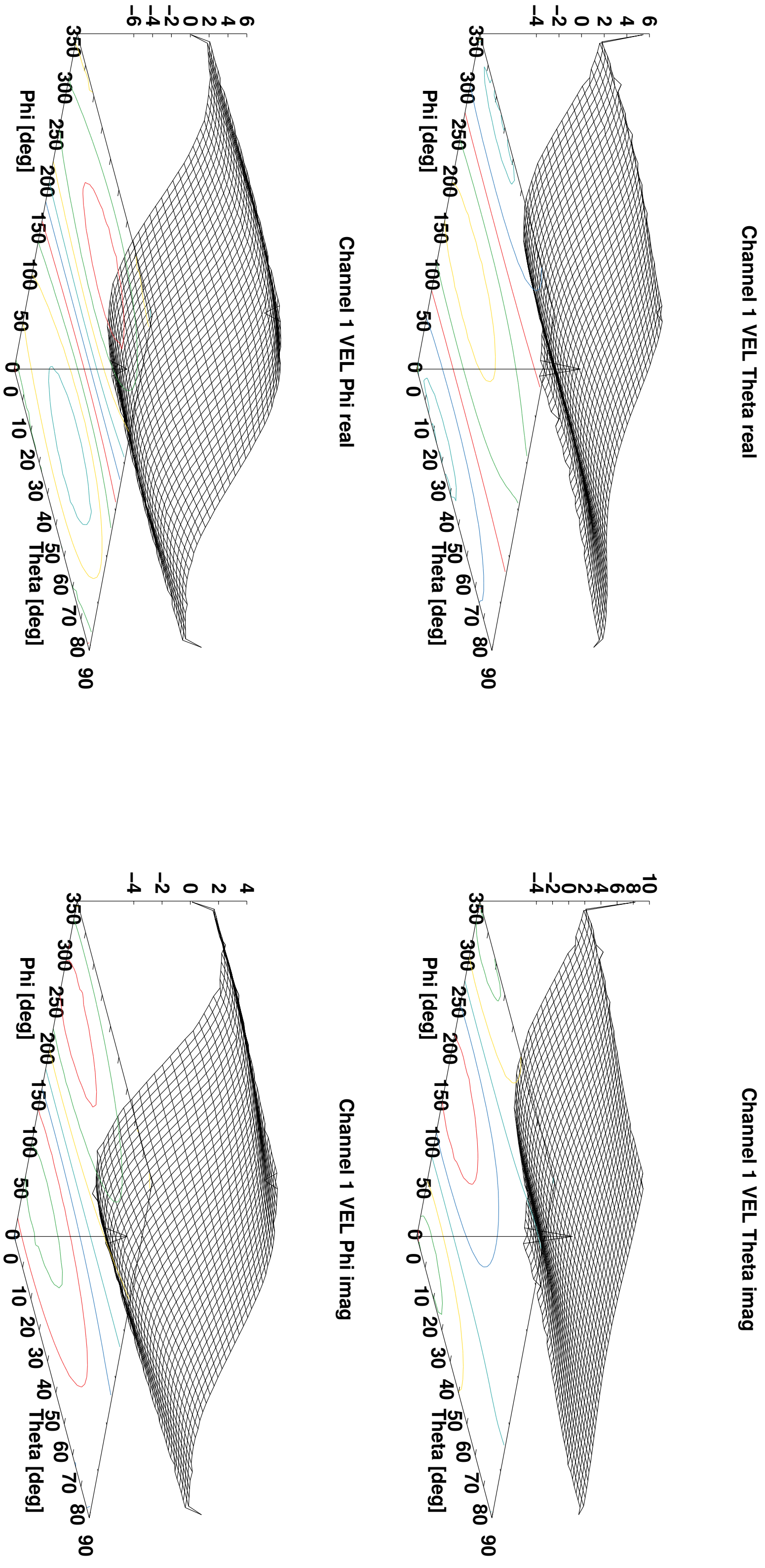


Figure 4.1: Example of an antenna response pattern at $\nu = 35\,\mathrm{MHz}$.

index in the vectors for given ϕ, θ and ω. Additionally, the lowest and highest value in each vector and the size of each vector are stored. Each pattern is only buffered and stored once at first usage. After that it is stored in a static map and all antennas of the same type are using the same pattern.

To copy the data to the GPU this map is converted to two three dimensional arrays, one for the θ and one for the ϕ component of the VEL. With the header information of each pattern, the response map is asked for all stored VEL, which are then written in the arrays. The old data structure is not replaced by the new arrays because this would require to modify the complete implementation of the `AntennaType` class and to verify its correctness.

The `PatternTexture` class is added to maintain the textures on the GPU. The VEL arrays are copied to the GPU and stored in two objects of type `cudaArray`. A texture is then binded to each `cudaArray`. Another static map stores one `PatternTexture` object for all antennas of the same type. The conversion to the GPU is done only once and can be neglected, as it needs in total less than 1 s. However, it would simplify the code if the old data structure could be completely replaced by the new one.

The biggest pattern consists of 98 frequency values, 31 theta values and 49 phi values, so 148 862 complex values in single precision have to be stored for each component of the VEL. Assuming a size of float of 4 Byte the total size of the biggest pattern is 2.4 MB. This is small compared to the total available memory size of a few GB on modern GPUs and much below the maximal size of a 3D texture of 2048×2048×2048 or 4096×4096×4096 elements depending on the device [10].

Reading data from a texture is called a "fetch". The texture can be configured to support different fetch modes which are described in the following. By setting the filter mode to `cudaFilterModeLinear` the texture T interpolates a fetched point linear by the eight surrounding grid points. This is only possible for floating point textures in single precision. The influence of using single precision instead of double precision is discussed in the following section. The returned value of the fetch tex(x,y,z) is given by

$$\begin{aligned}\mathrm{tex}(x,y,z) = {} & (1-\alpha)(1-\beta)(1-\gamma)T[i,j,k] + \alpha(1-\beta)(1-\gamma)T[i+1,j,k] \\ & + (1-\alpha)\beta(1-\gamma)T[i,j+1,k] + \alpha\beta(1-\gamma)T[i+1,j+1,k] \\ & + (1-\alpha)(1-\beta)\gamma T[i,j,k+1] + \alpha(1-\beta)\gamma T[i+1,j,k+1] \\ & + (1-\alpha)\beta\gamma T[i,j+1,k+1] + \alpha\beta\gamma T[i+1,j+1,k+1].\end{aligned} \tag{4.4}$$

Here, the weights α, β, γ and the indexes i, j, k of the surrounding grid points are defined as

$$\begin{aligned} x_B &= x - 0.5 & \alpha &= \mathrm{frac}(x_B) & i &= \mathrm{floor}(x_B) \\ y_B &= y - 0.5 & \beta &= \mathrm{frac}(y_B) & j &= \mathrm{floor}(x_B) \\ z_B &= z - 0.5 & \gamma &= \mathrm{frac}(z_B) & k &= \mathrm{floor}(x_B). \end{aligned}$$

α, β and γ are stored in 9 bit fixed point format with 8 fractional bits [10]. The index order of the texture is reverse to the array on CPU. Thus, if one wants to fetch a texture

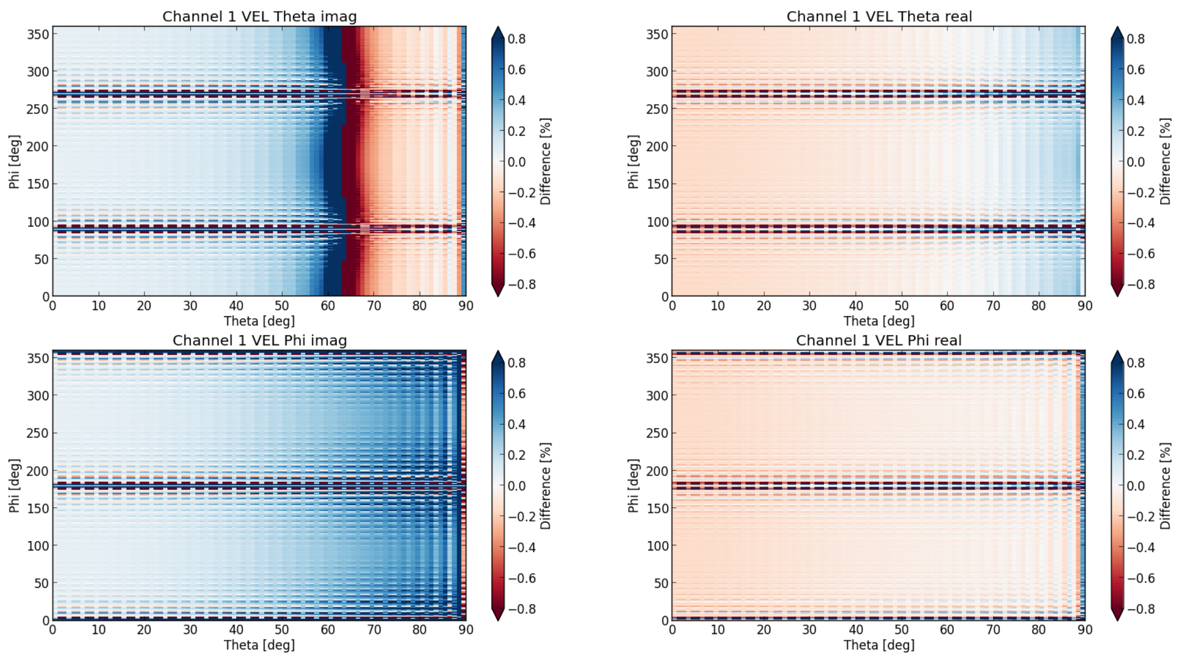


Figure 4.2: Differences between the interpolated pattern on the GPU and CPU. Shown is the relative deviation $(H^{\text{CPU}}_{\phi,\theta} - H^{\text{GPU}}_{\phi,\theta})/H^{\text{CPU}}_{\phi,\theta}$

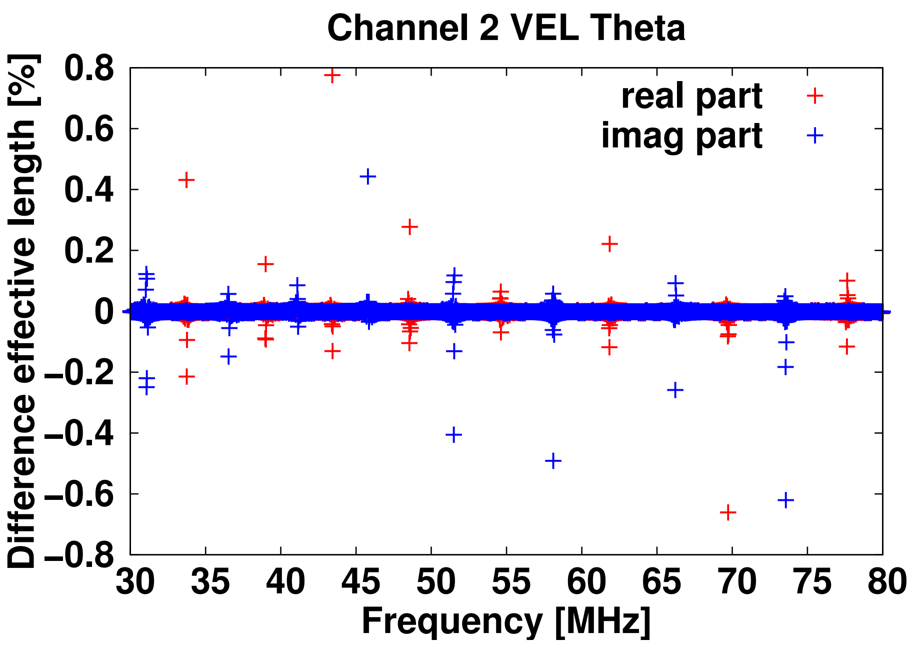
with `tex(x,y,z)`, the CPU array has to be allocated as `array[z][y][x]`. Additionally, 0.5 has to be added to all coordinates to move the sample point to the middle of the pixel.

Alternatively the filter mode could be set to `cudaFilterModePoint`. In this mode a fetch would return the value of the nearest point in the texture. This would correspond to the optionally available lookup implementation on the CPU.

In figure 4.2 the relative differences $(H^{\text{CPU}}_{\phi,\theta} - H^{\text{GPU}}_{\phi,\theta})/H^{\text{CPU}}_{\phi,\theta}$ between both interpolations are shown for all components of the pattern. Typically, the differences are below $\pm 0.8\,\%$, but there are also small regions with a greater relative difference. They correspond to the zero crossing of the pattern and are evenly distributed around zero. An example of the relative differences of an antenna response for a fixed direction calculated using the CPU and the CPU implementation are shown in figure 4.3. Again, the differences are typically below $\pm 0.8\,\%$.

The `RdAntennaChannelToStationConverter` module has to be modified to profit from the parallel interpolation on the GPU. In the baseline implementation the antenna response is computed in a loop over the individual frequencies for both channels. Using the texture interpolation this would require launching an individual kernel for every frequency. However, it is more efficient to compute the antenna responses for all frequencies in parallel on the GPU.

With the texture interpolation on the GPU, the reconstruction time per event has been reduced from 13.4 s to 9 s on the cluster. This is an reduction by 25 % compared to the Hilbert version and 33 % compared to the baseline. On the desktop system the time per event has reduced from 14.7 s to 7.2 s. This is an reduction by nearly 35 %



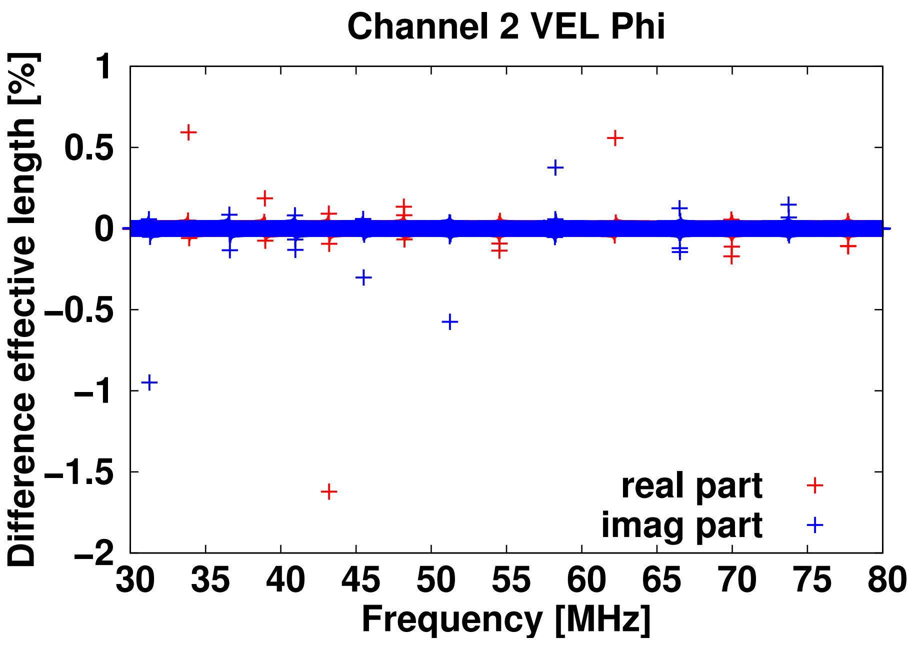


Figure 4.3: Real and imaginary part of the antenna response for $\theta \approx 52°$ and $\phi \approx 273°$ calculated on CPU and GPU.

compared to the Hilbert version and 51 % compared to the baseline. This is visualized again in figure 4.4.

This Offline version is analyzed by the nvvp to examine the GPU usage. The computing time of a kernel is low compared to the time needed for memory copy with a ratio of approximately 1.32. Just 7 % of the memory copy is overlapped by computation. Additionally, the memory copy throughput is low. There are never two

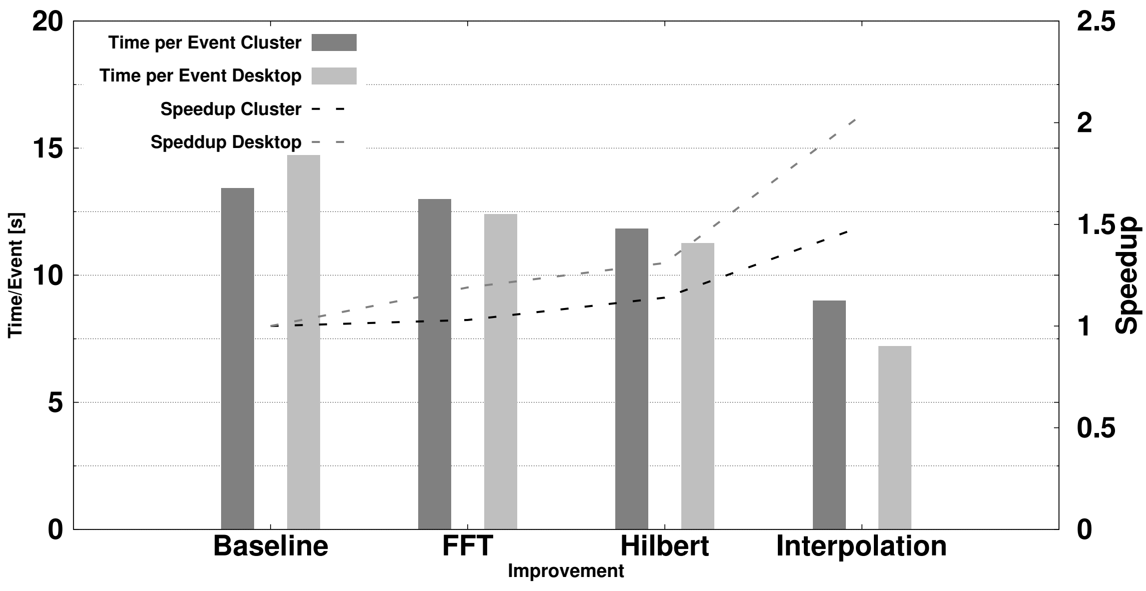


Figure 4.4: Comparison of the time per event and the Speedup after final implementation of cuFFT, improved Hilbert envelope and interpolation.

kernels executed in parallel and the multiprocessor of the GPU is mostly idle. Hence the GPU usage can still be increased.

4.2 Comparison of Single and Double Precision

The trilinear interpolation on CPU is identical to 7 successive linear interpolations

$$y = y_0 + (y_1 - y_0) \cdot \frac{x - x_0}{x_1 - x_0}. \tag{4.5}$$

Evaluating this equation will produce uncertainties due to the limited machine accuracy ϵ. A rough estimation of the resulting uncertainty Δy is given by

$$\frac{||\Delta y||}{||y||} \leq n \cdot \epsilon \tag{4.6}$$

where $n = 6$ is the amount of elementary operations in each interpolation.

The IEEE 754 standard defines the machine epsilon for single precision to $\epsilon = 2^{-24} \approx 6 \times 10^{-8}$ and for double precision to $\epsilon = 2^{-53} \approx 1.1 \times 10^{-16}$. Hence the biggest possible uncertainty of the trilinear interpolation is $4.7 \times 10^{-13}\,\%$ for double precision and $2.5 \times 10^{-4}\,\%$ for single precision. Using the texture interpolation of equation (4.4) 51 operations are needed. Thus the uncertainty is $\leq 3 \times 10^{-4}\,\%$. As this is far below of the other uncertainties in the reconstruction switching from double to single precision has no negative effect on the reconstruction.

5 Conclusion

In this bachelor thesis performance improvements of the Auger Offline Software Framework for the event reconstruction of the Auger Engineering Radio Array have been developed. The presented solutions enable computation of CPU intensive tasks on the GPU.

The calculation of Fourier transformations and the interpolation of antenna response patterns have been identified as the two biggest hotspots in the AERA event reconstruction by a performance profiler. Both tasks have been reimplemented for computation on the GPU while keeping the interface of the modified classes and the structure of Offline intact.

Of several approaches that have been tested, the best performance has been achieved using static memory for storing the FFT data on the GPU and an upsampling of the trace length to a power of 2. Additionally the computation of the Hilbert envelope has been implemented to be executed solely on the GPU. Together, this yields a speedup of approximately 1.14 to 1.31 depending on the test system.

In a further step, the interpolation of the antenna pattern has been implemented on the GPU. Here, the dedicated memory and interpolation circuits available on GPUs for interpolation of textures are used. Using the texture interpolation the speedup has been increased to approximately 1.49 to 2.04.

Further speedup could be achieved by e.g. changing the `FFTDataContainer` to store the traces of all stations in the event permanently on the GPU and use them for the full reconstruction loop. However, this requires greater changes in Offline which have been avoided in this minimal invasive approach.

The presented improvements are published in a separate Offline branch. GPGPU optimizations can optionally be enabled with a compiler flag, if the CUDA Toolkit is found.

Appendices

A Modulesequence

```
<moduleControl>

    <loop numTimes="unbounded">

      <module> EventFileReaderOG                  </module>
      <module> RdEventPreSelector                 </module>

      <module> RdEventMerger                      </module>

      <module> EventCheckerOG                     </module>
      <module> SdQualityCutTaggerOG               </module>
      <module> SdPMTQualityCheckerKG              </module>
      <module> TriggerTimeCorrection              </module>
      <module> SdCalibratorOG                     </module>
      <module> SdBadStationRejectorKG             </module>
      <module> SdSignalRecoveryKLT                </module>
      <module> SdEventSelectorOG                  </module>
      <module> SdPlaneFitOG                       </module>
      <module> LDFFinderKG                        </module>
      <try>
        <module> SdHorizontalReconstruction       </module>
      </try>

      <module> RdEventInitializer                 </module>
      <module> RdStationRejector                  </module>
      <module> RdChannelADCToVoltageConverter     </module>
      <module> RdChannelSelector                  </module>
      <module> RdChannelPedestalRemover           </module>
      <module> RdChannelResponseIncorporator      </module>
      <module> RdChannelBeaconSuppressor          </module>
      <module> RdChannelTimeSeriesTaperer         </module>
      <module> RdChannelBandstopFilter            </module>
      <module> RdChannelUpsampler                 </module>

      <loop numTimes="unbounded">
        <module> RdDirectionConvergenceChecker      </module>
        <module> RdAntennaChannelToStationConverter </module>
        <module> RdStationSignalReconstructor       </module>
        <module> RdClusterFinder                    </module>
        <module> RdPlaneFit                         </module>
      </loop>
      <module> RdLDFMultiFitter                  </module>
      <module> RdChannelRiseTimeCalculator       </module>
      <module> RdStationEFieldVectorCalculator   </module>
      <module> RdEventPostSelector               </module>

      <try>
        <module> FdCalibratorOG                  </module>
        <module> FdEyeMergerKG                   </module>
        <module> FdPulseFinderOG                 </module>
        <module> FdSDPFinderOG                   </module>
        <module> FdAxisFinderOG                  </module>
        <module> HybridGeometryFinderOG          </module>
        <module> HybridGeometryFinderWG          </module>
        <module> FdApertureLightKG               </module>
        <module> FdEnergyDepositFinderKG         </module>
        <module> FdProfileReconstructorKG        </module>
      </try>

      <module> RdStationTimeSeriesWindowCutter   </module>
      <module> RdStationTimeSeriesTaperer        </module>
      <module> EventFileExporterOG               </module>
      <module> RecDataWriterNG                   </module>

    </loop>

  </moduleControl>
```

B FFTW Patch

```
--- ../../ape/External/fftw/3.3.3/include/fftw3.h~      2014-07-10
   14:45:39.359386088 +0200
+++ ../../ape/External/fftw/3.3.3/include/fftw3.h       2014-07-10
   14:46:22.315386004 +0200
@@ -357,7 +357,7 @@
 /* __float128 (quad precision) is a gcc extension on i386, x86_64,
    and ia64
    for gcc >= 4.6 (compiled in FFTW with --enable-quad-precision) */
 #if (__GNUC__ > 4 || (__GNUC__ == 4 && __GNUC_MINOR__ >= 6)) \
- && !(defined(__ICC) || defined(__INTEL_COMPILER)) \
+ && !(defined(__ICC) || defined(__INTEL_COMPILER) || defined(
   __CUDACC__)) \
  && (defined(__i386__) || defined(__x86_64__) || defined(__ia64__))
 #  if !defined(FFTW_NO_Complex) && defined(_Complex_I) && defined(
    complex) && defined(I)
 /* note: __float128 is a typedef, which is not supported with the
    _Complex
```

C Test Systems

Table 1: Comparison of the desktop and cluster test systems [22, 23, 24, 25]

	Desktop	Cluster
Operating system	Debian GNU/Linux	Debian GNU/Linux
CUDA Toolkit version	6.0	4.2
RAM	8 GB	48 GB
CPU	1x AMD A8-6600K	24x Intel Xeon X5650
Cores	4	6
Threads	4	12
Multiprocessing	Uniprocessor	Up to 2 processors
L1 cache size (instruction)	2 x 64 KB instruction	6 x 32 KB instruction
L1 cache size (data)	4 x 16 KB data	6 x 32 KB data
L2 cache size	2 x 2 MB	6 x 256 KB
L3 cache size	None	12 MB
GPU	1x GeForce 750 Ti	4x Tesla M2090
Cores	640	512
Processor clock	1020 MHz base clock	
	1085 MHz boost clock	1.3 GHz
Memory clock	5.4 Gbps	1.85 GHz
Memory Interface	GDDR5	GDDR5
Memory Interface width	128 bit	384 bit
Memory Bandwidth	86.4 GB/s	177 GB/s
Memory size	2048 MB	6 GB

References

[1] *TOP500 list*. June 14. URL: http://www.top500.org/lists/2014/06/ (visited on 09/19/2014).

[2] *The GREEN 500 List*. June 14. URL: http://www.green500.org/lists/green201406 (visited on 09/19/2014).

[3] K. Kotera and A. V. Olinto. *The Astrophysics of Ultrahigh-Energy Cosmic Rays*. Annual Review of Astronomy and Astrophysics 49 (2011), pp. 119–153. DOI: 10.1146/annurev-astro-081710-102620. arXiv: 1101.4256 [astro-ph.HE].

[4] A. Schaeffer. *LHCf sheds new light on cosmic rays*. URL: http://cds.cern.ch/journal/CERNBulletin/2011/18/News%20Articles/1345733 (visited on 09/19/2014).

[5] I. Allekotte et al. *The Surface Detector System of the Pierre Auger Observatory*. Nuclear Instruments and Methods in Physics Research A586 (2008), pp. 409–420. DOI: 10.1016/j.nima.2007.12.016. arXiv: 0712.2832 [astro-ph].

[6] The Pierre Auger Collaboration. *The Fluorescence Detector of the Pierre Auger Observatory*. Annual Review of Astronomy and Astrophysics 49 (2009), pp. 119–153. DOI: 10.1146/annurev-astro-081710-102620. eprint: 0907.4282 (astro-ph.IM).

[7] A. Aab et al. *Probing the radio emission from air showers with polarization measurements*. Physical Review D89 (2014), p. 052002. DOI: 10.1103/PhysRevD.89.052002. arXiv: 1402.3677 [astro-ph.HE].

[8] P. Abreu et al. *Antennas for the detection of radio emission pulses from cosmic-ray induced air showers at the Pierre Auger Observatory*. Journal of Instrumentation 7.10 (2012), P10011.

[9] T. Winchen. *The Principal Axes of the Directional Energy Distribution of Cosmic Rays Measured with the Pierre Auger Observatory*. PhD thesis. RWTH Aachen University, 2013.

[10] *CUDA C Programming Guide*. NVIDIA. 2014. URL: http://docs.nvidia.com/cuda/cuda-c-programming-guide/ (visited on 09/19/2014).

[11] *OpenCL - The open standard for parallel programming of heterogenous systems*. URL: https://www.khronos.org/opencl/ (visited on 09/19/2014).

[12] M. Harris. *Six Ways to SAXPY*. URL: http://devblogs.nvidia.com/parallelforall/six-ways-saxpy/ (visited on 09/24/2014).

[13] S. Argiro et al. *The Offline Software Framework of the Pierre Auger Observatory.* Nuclear Instruments and Methods in Physics Research A580 (2007), pp. 1485–1496. DOI: 10.1016/j.nima.2007.07.010. arXiv: 0707.1652 [astro-ph].

[14] P. Abreu et al. *Advanced functionality for radio analysis in the Offline software framework of the Pierre Auger Observatory.* Nuclear Instruments and Methods in Physics Research A 635 (2011), pp. 92–102. DOI: 10.1016/j.nima.2011.01.049. arXiv: 1101.4473 [astro-ph.IM].

[15] C. Glaser. *Energy Measurement and Strategy for a Trigger of Ultra High Energy Cosmic Rays Measured with Radio Technique at the Pierre Auger Observatory.* MA thesis. RWTH Aachen University, 2012.

[16] M. Gottowik, J. Rautenberg, and T. Winchen. *Prospects of GPGPU in the Offline Software Framework.* In *GPU Computing in High Energy Physics Workshop.* 2014.

[17] *Perf Wiki.* URL: https://perf.wiki.kernel.org/index.php/Main_Page (visited on 09/19/2014).

[18] *NVIDIA Visual Profiler.* URL: https://developer.nvidia.com/nvidia-visual-profiler (visited on 09/19/2014).

[19] *FFTW Home Page.* URL: http://www.fftw.org/ (visited on 09/22/2014).

[20] *cuFFT.* URL: https://developer.nvidia.com/cuFFT (visited on 09/22/2014).

[21] *NVIDIA CUDA Compiler Driver NVCC.* URL: http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/#axzz3DkaNiDPu (visited on 09/19/2014).

[22] URL: http://www.cpu-world.com/CPUs/Bulldozer/AMD-A8-Series%20A8-6600K%20-%20AD660KWOA44HL.html (visited on 09/22/2014).

[23] URL: http://www.cpu-world.com/CPUs/Xeon/Intel-Xeon%20X5650%20-%20AT80614004320AD%20(BX80614X5650).html (visited on 09/22/2014).

[24] URL: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-750-ti/specifications (visited on 09/22/2014).

[25] URL: http://www.nvidia.com/docs/IO/43395/Tesla-M2090-Board-Specification.pdf (visited on 09/22/2014).