COMPUTATIONAL OPTIMIZATION OF A TIME-DOMAIN BEAMFORMING ALGORITHM USING CPU AND GPU

Johannes Stier, Christopher Hahn, Gero Zechel and Michael Beitelschmidt
Technische Universität Dresden, Institute of Solid Mechanics,
Chair of Dynamics and Mechanism Design
Marschnerstraße 30, 01307 Dresden

ABSTRACT

In 2010, a special time-domain beamforming algorithm was presented at the Berlin Beamforming Conference [3]. This algorithm is primarily designed for the sound source localization on moving objects with known velocity (e.g. freight trains). By determining the object trajectory, the acoustic map's quality can be improved with respect to the Doppler effect.

The bottleneck of the algorithm is the time-consuming computational evaluation. Although computational effiency was considered in the algorithm's first implementation, it can take on hour or more to calculate an acoustic map for a passing train on a regular personal computer. There are several factors which affect the evaluation time, e.g. sampling rate, train speed or the train's length.

This paper mainly focusses on the computational implementation of the time-domain beamforming algorithm using CPU (Central Processing Unit) and GPU (Graphics Processing Unit). In general, the implementation on a CPU is rather straight foward if common parallelization libraries are used (e.g. OpenMP), offering only a few variation opportunities. The realizable speed-up is proportional to the number of physical cores in a CPU, and can attain a factor of 8 on recent workstations. Implementing the beamforming algorithm on a GPU with CUDA (Compute Unified Device Architecture) is more complicated and requires substantial knowledge of the GPU's processor architecture. Nevertheless, speed-ups of 30 times or even more compensate the high implementation effort.

Additionally, a modification of the algorithm according to [3] is presented. Specific calculation coefficients called "beamfactors" are introduced, which represent the shading factors in time-domain beamforming. Precomputing those factors before beamforming can reduce the evaluation time by a factor of 2, regardless of the computational implementation and the computer system used. Although the beamfactors-algorithm offers a sufficient reduction of computational costs, it has been parallelized on CPU and GPU as well.

1 INTRODUCTION

Beamforming associated with a microphone array is a powerful tool to locate sound sources. Generally, beamforming can be applied in frequency or time domain. When locating sound sources on moving objects, beamforming must be applied in the time domain due to frequency shifts caused by the Doppler effect. Knowing the object's trajectory eases the necessary De-Dopplerization.

In 2010, a special time-domain beamforming algorithm was presented, involving the object's velocity to consider the Doppler effect [3]. The algorithm presented is primarily applied to locate sound sources on (freight) trains, whose motion can be determined very easily. To locate the sound sources on the base of the data recorded by a microphone array, the object is rasterized by a grid of hypothetical monopole sources. For each pixel in the grid, the signal of the hypothetical monopole is reconstructed by tracking the pixel over a certain time period, determined by the microphone array used, and applying beamforming. Afterwards, the reconstructed signal is analyzed by Discrete Fourier Transform (DFT) to get the frequency spectrum, which is often filtered by octave bands and converted to a sound pressure level for each octave. The result is a set of acoustic maps, one per octave, being colorized images representing the sound pressure level of each pixel. By differences in the sound pressure level, and thus color differences, the object's sound sources are revealed.

Although, computational efficiency was taken into account and discussed in [3], creating the acoustic maps for a regular freight train is very time-consuming. For instance, evaluating the measurement of a freight train with $110\,\mathrm{m}$ of length, traveling at $76\,\mathrm{km\,h^{-1}}$, measured with a sampling frequency of $100\,\mathrm{kHz}$ and a desired acoustic map resolution of $0.1\,\mathrm{m}$ takes $6\,\mathrm{h}$ on a state-of-the-art personal computer. Besides accuracy considerations, the evaluation (computation) time is a high impact factor to allow the microphone array, in combination with beamforming, to be an effective acoustic analyzing tool. Therefor, a great demand on reducing the computation time arises. Because of the structure of the time-domain beamforming algorithm, there is much potential for computational optimization using advanced parallelization techniques.

Before describing the approaches developed and techniques applied to reduce the computational time of time-domain beamforming in Sec. 3.1, the most important fundamentals of the special beamforming algorithm are outlined in Sec. 2.

2 TIME-DOMAIN BEAMFORMING FOR MOVING SOURCES

Being one of the main elements of the sound locating tool microphone array, the signal processing algorithm beamforming mainly influences the tool's effectiveness. According to [3], the origin for the following explanations is the well known equation for time-domain beamforming on moving sound sources with known trajectory:

$$p_n(t) = \frac{1}{M}\sum_{m=0}^{M-1} w_{m,n}(t)\cdot p_m(t+\Delta t_{m,n}(t)). \tag{1}$$

The moving hypothetical monopole source's signal $p_n(t)$ at pixel n is reconstructed by time-shifting the M microphone signals $p_m(t)$ by

$$\Delta t_{m,n}(t) = \frac{r_{m,n}(t)}{c}, \tag{2}$$

multiplying them with the weight factor

$$w_{m,n}(t) = \frac{r_{m,n}(t)}{r_0}, \tag{3}$$

and summing them up. The weighting factor $w_{m,n}(t)$ and time-shift $\Delta t_{m,n}(t)$ depend on the time-dependent euclidean distance $r_{m,n}(t)$ between microphone m and pixel n. c is the speed of sound and r_0 the reference distance to determine the sound sources' sound pressure level[1].

Because of the continuous time variable t, Eq. (1) is not suitable for a direct computational implementation. A discrete time, "digital", time-domain beamforming equation must be derived by sampling the continuous time variable with $t = k \cdot \Delta T$:

$$p_n[k] = \frac{1}{M} \sum_{m=0}^{M-1} w_{m,n}[k] \cdot p_m[k + \frac{\Delta t_{m,n}[k]}{\Delta T}]. \tag{4}$$

ΔT is the reciprocate of the sampling frequency f_s. Of course, the weighting factor $w_{m,n}[k]$ and time shift $\Delta t_{m,n}[k]$ have to be regarded in discrete time as well. Obviously, because of the discrete time variable k being an integer, a conflict arises of summing k with the time-shift normalized $\frac{\Delta t_{m,n}[k]}{\Delta T}$ being a real number. From the computational point of view, this would require to access the microphone data array (via a memory address) at a real numbered position, which is of course not possible. According to [1], interpolation is required to overcome inaccuracies when transforming exact delays to integer values.

In [3], linear interpolation between the time steps

$$\Delta k_{m,n}[k] = \lfloor \frac{\Delta t_{m,n}[k]}{\Delta T} \rfloor \quad \text{and} \quad \Delta k_{m,n}[k] + 1 \tag{5}$$

is used to approximate the exact time delay to reduce inaccuracies. In this case, $\lfloor \cdot \rfloor$ indicates rounding down to the nearest integer value. Furthermore, introducing the coefficients

$$a_{m,n}[k] = \frac{r_{m,n}[k]}{r_0 \cdot c^*} \cdot (r_{m,n}[k] - \lfloor r_{m,n}[k] \rfloor) \tag{6}$$

and

$$b_{m,n}[k] = \frac{r_{m,n}[k]}{r_0 \cdot c^*} \cdot (1 - r_{m,n}[k] + \lfloor r_{m,n}[k] \rfloor), \tag{7}$$

[1]For a detailed description see [3]

to allow interpolation, Eq. (4) can be rewritten as

$$p_n[k] = \frac{1}{M}\sum_{m=0}^{M-1}\left(a_{m,n}[k]\cdot p_m[k+\Delta k_{m,n}[k]] + b_{m,n}[k]\cdot p_m[k+\Delta k_{m,n}[k]+1]\right). \tag{8}$$

The speed of sound c in $\mathrm{m\,s^{-1}}$ is converted to the speed of sound in m/sample c^*. Equation (4) represents digital time-domain beamforming with linear interpolation for moving sources, being the base for all following explanations.

3 COMPUTATIONAL OPTIMIZATION

3.1 Digital Time-Domain Beamforming from the Computational Point of View

Setting up an algorithm on the base of Eq. (8), that processes all pixel N of the evaluation grid to calculate the acoustic maps in octaves A_{oct} by reconstructing the hypothetical source signals, results in Alg. 1. In addition to the beamforming algorithm (Li. 2 to Li. 8), further signal processing steps like windowing, frequency analysis, band filtering and sound pressure level calculation must be considered to complete the calculation of the acoustic maps (Li. 9 to Li. 11). Those steps are well known and are not evaluated in detail.

For the following explanations, the loop in Li. 1 is supposed to be processed serially. For all pixel N, the hypothetical monopole source in pixel n is tracked over the time interval $[l_{e,n}, l_{x,n}]$ and the source's signal $p_n[k]$ is reconstructed according to Eq. (8). $l_{e,n}$ and $l_{x,n}$ are determined by the microphone array's geometry. After the signal has been reconstructed, the frequency spectrum P_n is calculated, octave band filtered and sound pressure level $L_{p,oct}$ for each band are

Algorithm 1: Digital time-domain beamforming according to Eq. (8), serial and parallel implementation on CPU

```
Data: microphone signals $p_m[k]$
Result: acoustic map $A_{oct}$
1  for $n = 0$ to $N-1$ do (in parallel)
2      initialize $p_n[k] = 0$
3      for $l = l_{e,n}$ to $l_{x,n}$ do
4          for $m = 0$ to $M-1$ do
5              calculate $a_{m,n}[l]$, $b_{m,n}[l]$ and $\Delta k_{m,n}[l]$
6              calculate $p_{m,n}[l]$
7              add $p_{m,n}[l]$ to $p_n[l]$
8          divide $p_n[l]$ by $M$
9      window $p_n[k]$
10     calculate $P_n$ for $p_n[k]$
11     calculate $L_{p,oct,n}$ from $P_n$
12     assign $L_{p,oct,n}$ to $A_{n,oct}$
```

calculated. The result is an acoustic map for each octave, being a colorized representation of the octave sound pressure level for all pixel.

Implementing Alg. 1 computationally is straightforward, it only takes little effort to transfer the algorithm to program code. Analyzing the algorithm's structure reveals three ”for” loops, each being an iteration over a defined number of elements. As the experienced programmer knows, loops, more precisely nested loops, can be very time-consuming operations when being executed, depending on the number of loop counts. In this case, computation time is mainly influenced by the number of

- pixel N in the acoustic map, determined by the *object length* and *height* and the desired *acoustic map resolution*,
- time steps $L = l_{x,n} - l_{e,n}$ to reconstruct the signal for, determined by the *object velocity* and *sampling rate*,
- microphones M used in the microphone array.

Deriving the algorithm's order results in $\mathscr{O}(N \cdot L \cdot M)$.

Figure 1 and Fig. 2 confirm the influence of number of pixels N and time steps to process L on the computation time[2]. In the evaluation results shown in Fig. 1, all impact factors besides the number of pixel N were constant. The result is an increasing computation time by a constant factor of 10 when N increases by the same factor. Figure 2 shows the computation time for an evaluation with all impact factors being constant besides the velocity. In this case, the computation time decreases by the reciprocate of increasing velocity. A more detailed investigation of all other impact factors on computation time is not necessary because of their linear influence.

The results presented in Fig. 1 and Fig. 2 also evidence the great demand on reducing the computation time. Locating sound sources on slow and long freight trains with high sampling rates and a sufficient resolution according to the microphone array's capabilities is very inefficient. There are two evident approaches to reduce the computation time:

1. modifying the algorithm,
2. using advanced techniques to parallelize the algorithm.

While approach one requires detailed knowledge of the physical base of the algorithm, and may not lead to success without further restrictions to the algorithm, state-of-the-art computer offer partially easy-to-apply tools to parallelize algorithms. Fortunately, because of the plain structure of the beamforming algorithm described with the three independent ”for” loops, approach two can be applied easily. The resulting algorithms for parallelization on CPU and GPU and the results in reducing the computational effort are described in the next two sections. Nevertheless, Sec. 3.4 introduces an algorithmic modification of Alg. 1 leading to a reduced computational effort, but, as said before, in combination with further constraints.

[2]In this case, constant sampling frequency and varying object velocity

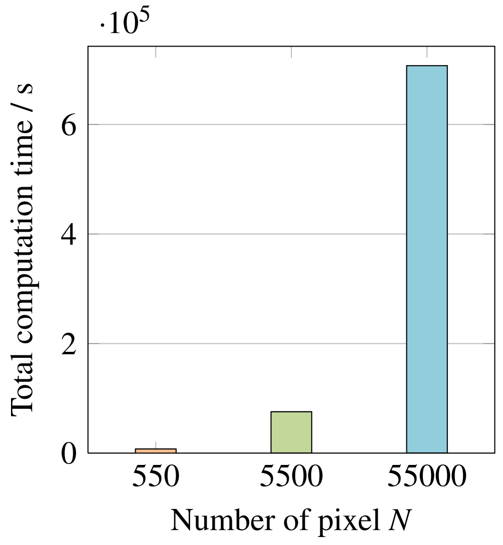

Figure 1: Computation time for digital time-domain beamforming depending on number of pixels N

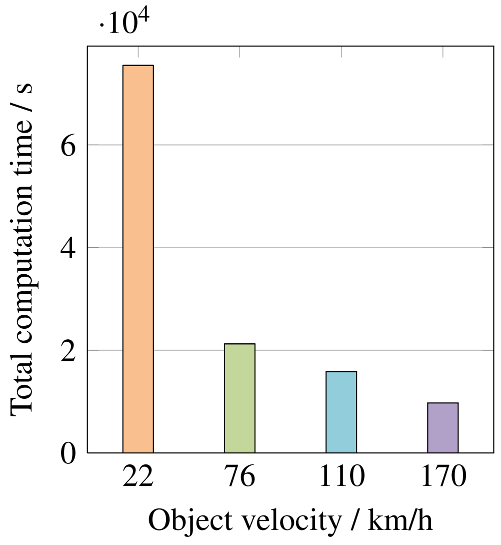

Figure 2: Computation time for digital time-domain beamforming depending on object velocity

3.2 Parallelization on CPU

Today, one of the most convenient means of reducing the computational effort for algorithms is parallelizing on a computer's Central Processing Unit (CPU). There is a great variety of tools and programming libraries supporting the programmer to parallelize. For instance, the commercial numerical computation program Matlab offers an easy-to-use toolbox for parallelization. This toolbox uses the special command ”parfor” to simply parallelize ”for” loops. By this mean, existing algorithm implementations can easily be modified to reduce computation time.

Basically, parallelization on a CPU is achieved by splitting a process to be executed in a certain number of threads, that will be executed in parallel. The threads' underlying tasks to perform have to be independent, otherwise parallelization is more complicated or may even not be possible in some cases. Opposite to former CPU architectures only having one processing unit (core), advanced CPUs offer more than one core, in state-of-the-art processors even up to six cores. While parallelization in the first case would not lead to any reduction in computation time, because of the serial execution of threads, using multiple cores can. In this case, the threads are indeed executed in parallel. Thus, the computation time with N_{core} cores can theoretically be reduced to the $1/N_{core}$th part.

Observing the time-domain beamforming algorithm in Alg. 1 from the parallelizational point of view, it offers two main approaches to parallelize on a CPU:

1. Parallelizing the outer loop over pixel N, thus a certain number of pixel will be processed simultaneously.

2. Parallelizing all commands in the outer loop, treating the calculation of $p_n[k]$ as one command (Li. 1 to Li. 8).

Using the loop over the pixel N for parallelization revealed to be the most efficient approach (Li. 1 in Alg. 1). The resulting computation time of the parallelized algorithm according to Alg. 1 with a varying number of threads N_{thread} shows Fig. 3. The results were created by an implementation with the programming language C by using the compiler extension OpenMP for parallelization. To get results being independent from the CPU type used for evaluation (clock speed, architecture, ...), the results are normalized to the serial algorithm's computation time. Using one thread for parallelization, which means only one pixel n is processed at the same time, leads, of course, to the same computation time like the serial implementation. With increasing number of threads, the computation time decreases. When two pixel n are calculated in parallel, the computation time is halved, processing four pixel n simultaneously quarters the computation time. Certainly, using more threads will further decrease the computation time. But this is only efficient when using CPUs with the same number of cores as threads used. But, depending on the CPU's architecture, the gap between the theoretical reduction by N_{thread} and the measured reduction will grow due to effects like memory bandwidth or memory access collisions, thus limiting the efficiency of parallelization. Modern PC architectures try to overcome this problem by using more then one multi-core CPUs or using new technologies like Intel's Hyperthreading.

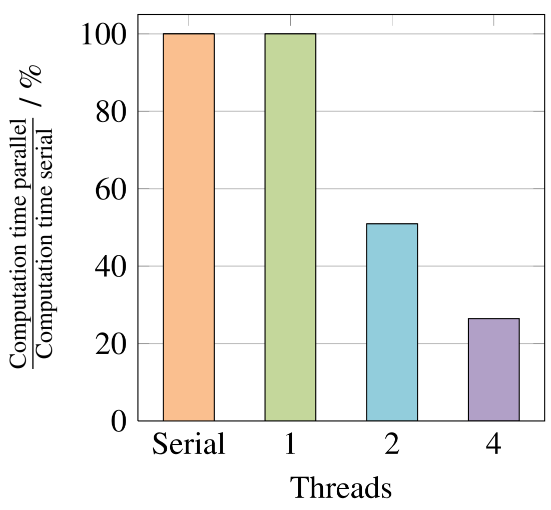

Figure 3: Computation time for digital time-domain beamforming depending on the number of threads for parallel implementation on CPU according to Alg. 1

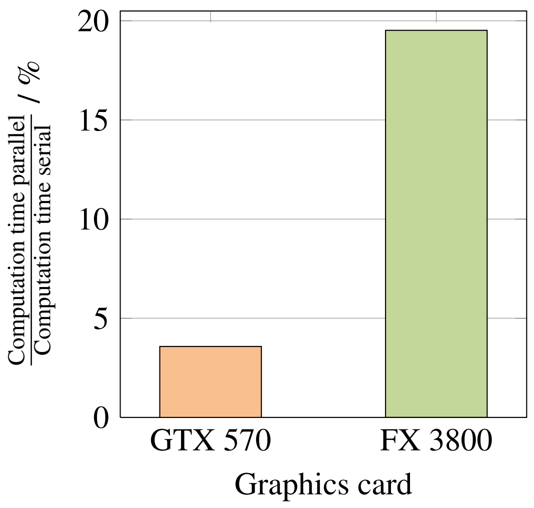

Figure 4: Computation time for digital time-domain beamforming for parallel implementation on GPU according to Alg. 2

3.3 Parallelization on GPU

In 2001, the first NVIDIA graphics card from the Geforce 3 Series was capable of being almost freely programmed, caused by the graphics card's very specific architecture becoming more flexible. Since that day, the graphics card metamorphosed from a device, only applied to 2D- or

3D-graphics applications, to a device for General Purpose Computation on Graphics Processing Units (GPGPU). With the introduction of NVIDIA's CUDA (Compute Unified Device Architecture) in 2006, which allows programmers to use NVIDIA graphics cards for GPGPU, this technology became very famous and has been applied in a great variety of applications. Because of the graphics processing units (GPU) applying the principle of array processors, which means executing the same command on many identical processors on various data, they offer much potential for massive parallel calculations. By using millions of parallel threads, an optimal workload can be achieved. For instance, the advanced graphics card NIVIDIA Geforce GTX 570 has 480 processors, is actually used in gaming applications, but can also be applied to GPGPU using CUDA. In comparison to GPUs, CPUs are designed for fast serial processing of commands and can be programmed more flexible than GPUs, thus offering less capabilities for parallelization.

But the high potential on parallel programming involves, because of the GPU's architecture, some restrictions on programming. In general, transferring algorithms directly to the graphics card for parallelization is thus not possible - algorithms need to be modified, demanding more effort than parallelizing on CPU. NVIDIA's CUDA technology is widely used, and has also been applied in the work presented. CUDA uses the concept of partitioning the problem in small problems, called kernels. Therefor, the algorithm to be parallelized has to be divided in small kernels which are executed on much data at the same time.

As described in Sec. 3.1, the digital time-domain beamforming's algorithmic structure is not very sophisticated. Thus, it can easily be modified to fulfill the demands for parallelization on graphics card, but offers only few variations. In [2], a frequency-domain and a time-domain beamformer have successfully been brought to the graphics card, substantiating the potential of parallelizing beamforming on the graphics card.

Applying the kernels concept to Alg. 1 results in Alg. 2 for parallelization on the graphics card. The basic structure of the algorithm did not change. The following four kernels were introduced:

1. Li. 5: Kernel to reconstruct the source signal $p_n[k]$ for N_{calc} pixel for time interval $[l_{e,n}, l_{x,n}]$.

2. Li. 11: Kernel to window the reconstructed source signal before analyzing with DFT.

3. Li. 13: Kernel to calculate the frequency spectrum P_n by applying DFT[3].

4. Li. 15: Kernel for octave band filtering and sound pressure level calculation $L_{p,oct,n}$.

N_{calc} is the number of pixels calculated in parallel to allow an optimal work load for the graphics card, depending on the graphics card's properties. It is determined before the algorithm starts (Li. 1) and can be calculated with a tool supplied by NVIDIA. Because the number of pixel N of the acoustic map is greater than N_{calc}, the kernels are executed serially N/N_{calc} times. As mentioned before, an optimal work load of the graphics card is achieved when a high number of calculation threads is started. For instance, the kernel in Li. 5 is executed in $N_{calc} \cdot L$ threads, which means that the time steps l for N_{calc} pixel are calculated in parallel. The same approach is also applied in the cases of the other kernels.

[3]using Fast Fourier Transform

Algorithm 2 was implemented using CUDA in the programming language C. It has been tested on the state-of-the-art graphics card NVIDIA Geforce GTX 570 for personal computer and on the five years old NVIDIA Quadro FX 3800 mainly used in work stations. The two cards mainly differ in their architecture and the number of CUDA cores available (GTX 570: 480 cores, FX 3800: 192 cores).

Figure 4 shows the computation times normalized to the serial algorithm executed on the CPU. Compared to the serial implementation, the computation time could be reduced to almost four percent. Thus, parallelizing on the graphics card offers much more potential than on the CPU. Although, as the results in Fig. 4 clearly show, the computation time highly depends on the graphics card used. When using the older graphics card FX 3800, the reduction achieved is only twenty percent compared of serial implementation on CPU.

Algorithm 2: Digital time-domain beamforming, parallel implementation on GPU

```
Data: microphone signals $p_m[k]$
Result: acoustic map $A_{n,oct}$
1  determine $N_{calc}$
2  for $j = 0$ to $N/N_{calc}$ do
3      for $n = 0$ to $N_{calc}$ do
4          initialize $p_n = 0$
5      foreach tuple $(n,l)$ with $n \in \{N_{calc}\}$, $l \in [l_{e,n}, l_{x,n}]$ do in parallel
6          for $m = 0$ to $M - 1$ do
7              calculate $a_{m,n}[l]$, $b_{m,n}[l]$ and $\Delta k_{m,n}[l]$
8              calculate $p_{m,n}[l]$
9              add $p_{m,n}[l]$ to $p_n[l]$
10         divide $p_n[l]$ by $M$
11     foreach $n \in \{N_{calc}\}$ do in parallel
12         window $p_n[k]$
13     foreach $n \in \{N_{calc}\}$ do in parallel
14         calculate $P_n$ for $p_n[k]$
15     foreach $n \in \{N_{calc}\}$ do in parallel
16         calculate $L_{p,oct,n}$ from $P_n$
17         assign $L_{p,oct,n}$ to $A_{n,oct}$
```

3.4 Beamfactors

As mentioned before, the beamforming algorithm explained in Sec. 2 is mainly applied to the sound source localization on trains. In the majority of train measurements, the train moves horizontally along the microphone array with negligible motion in other directions. Furthermore, assuming the train is passing with constant velocity, algorithmic simplifications to Alg. 1 can

Algorithm 3: Digital time-domain beamforming using beamfactors, serial and parallel implementation on CPU

```
Data: microphone signals $p_m[k]$
Result: acoustic map $A_{n,oct}$
// Calculate beamfactors
for $i = 0$ to $N_y - 1$ do
    for $l = 0$ to $l_{x,i} - l_{e,i} + 1$ do (in parallel)
        calculate $a_{m,i}[l]$, $b_{m,i}[l]$ and $\Delta k_{m,i}[l]$
// Do beamforming
for $i = 0$ to $N_y - 1$ do (in parallel)
    for $j = 0$ to $N_x - 1$ do
        calculate $n = j \cdot N_x + i$
        initialize $p_n = 0$
        for $l = l_{e,n}$ to $l_{x,n}$ do
            for $m = 0$ to $M - 1$ do
                set $l^* = l - l_{e,n}$
                calculate $p_n[l]$ with $a_{m,i}[l^*]$, $b_{m,i}[l^*]$ and $\Delta k_{m,i}[l^*]$
            divide $p_n[l]$ by $M$
        window $p_n[k]$
        calculate $P_n$ for $p_n[k]$
        calculate $L_{p,oct,n}$ from $P_n$
        assign $L_{p,oct,n}$ to $A_{n,oct}$
```

be made. The distance $r_{m,n}[k]$ between pixel n and microphone m, used to determine the interpolation coefficients and time delays (see Eq. (6), Eq. (7) and Eq. (8)), has the same shape for every pixel in a row of the evaluation grid. More precisely, $r_{m,n}[l]$ evaluated over the time period $l \in [l_{e,n}, l_{x,n}]$ can be affiliated to a basic function common to all pixels in a row. Thus, the interpolation coefficients, $a_{m,n}[k]$ and $b_{m,n}[k]$, and time delays, $\Delta k_{m,n}[k]$, only have to be calculated once for all microphones and time steps l, and can be applied to reconstruct the source signal $p_n[k]$ to all pixel of a row in the evaluation grid. These precomputed factors are called "beamfactors" and were actually introduced in [3].

Involving the beamfactors to the digital time-domain beamforming algorithm leads to Alg. 3. As described above, the interpolation coefficients and time delays for all pixel in a row are precomputed (Li. 1 to Li. 3) before reconstructing the source signals. Thus, the necessary calculation steps in the inner loop in Alg. 1 Li. 4 are reduced. Because of the inner loop being executed $N \cdot L$ times, precomputing the beamfactors leads to a great reduction in computation time, although the algorithm's order $\mathcal{O}(N \cdot L \cdot M)$ did not change.

Furthermore, the beamforming algorithm using beamfactors was parallelized on CPU and GPU as well, using the same approaches as described in Sec. 3.2 and Sec. 3.3. The resulting algorithm for the parallelization on CPU is shown in Alg. 3, with the loops in Li. 1 and Li. 4

being executed in parallel. The algorithm for GPU parallelization is omitted, but can easily be derived from Alg. 2.

To assess the algorithmic simplifications and their influence on computation time, the algorithm using beamfactors has initially been executed serially on CPU. Without parallelization, using beamfactors leads to a reduction in computation time of one third of the original algorithm executed serially. Because the interpolation coefficients and time delays are only calculated once for every row N_y, the necessary operations in the inner loop are thus reduced by $N_y \cdot (N_x - 1)$.

Using parallelization techniques reduces the computation time further. As Fig. 5 shows, the more threads used on CPU, the less the computation time. For instance, executing the beamforming algorithm with beamfactors on CPU with four threads, decreases the computation time to about 9 percent compared to the serial implementation, which is on fourth of the non-parallelized algorithm. The resulting computation times for the parallelized algorithm on GPU are summarized in Fig. 6. Analogous to the results in Alg. 3, the computation time can be decreased to about 5 percent of the serial's algorithm without beamfactors on the advanced graphics card GTX 570.

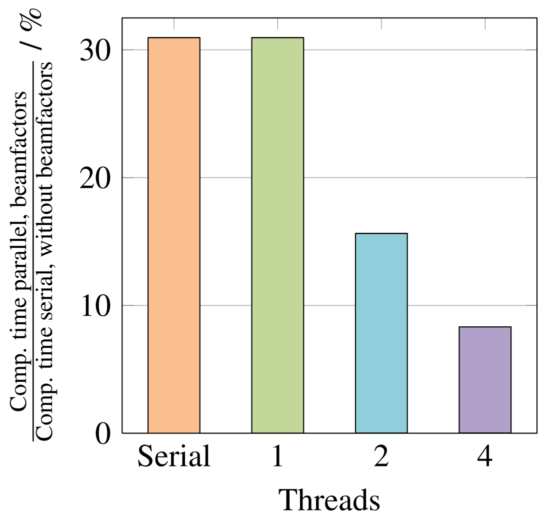

Figure 5: Computation time for digital time-domain beamforming using beamfactors, serial and parallel implementation on CPU

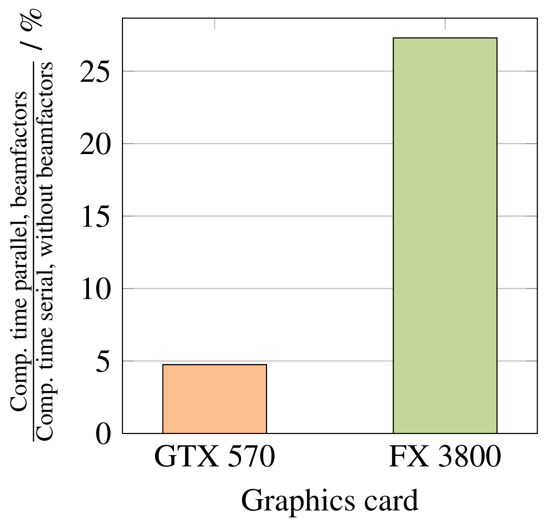

Figure 6: Computation time for digital time-domain beamforming using beamfactors, parallel implementation on GPU

4 SUMMARY

This article mainly focused on comparing different approaches to computationally optimize the time-domain beamforming algorithm presented in [3]. Applying the original implementation to evaluate a regular microphone array measurement took many hours, reducing the microphone array's efficiency and thus its potential as sound analyzing tool.

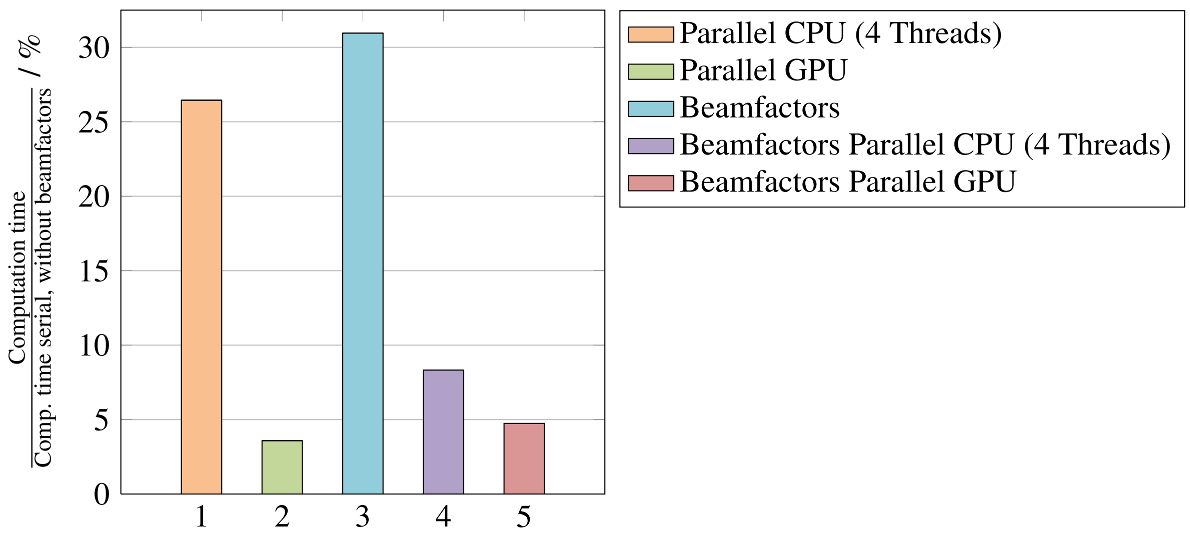

Figure 7: Comparison of computation time for the approaches described in Sec. 3.2 to Sec. 3.4

As the starting point, the fundamental equations of the beamforming algorithm were described in Sec. 2, followed by an examination of the algorithms structure in Sec. 3.1. The two main platforms which exist to parallelize algorithms, were applied to the beamforming algorithm.

The parallelization on CPU, described in Sec. 3.2, revealed to be the most effortless approach and leads to very satisfying results. Depending on the number of threads used for the calculations, the computation time can be reduced to one fourth of the serial's computation time on regular PCs (see Fig. 7). Even more can be achieved on workstations with more advanced CPUs. Despite the possible reduction in computation time, referring back to the example listed in Sec. 1, using four threads decreases the evaluation time to 1.5 h in this case. The results are little surprising and well known since the introduction of CPU parallelization techniques.

Modifying the beamforming algorithm for parallel calculations on GPU, results in a great reduction in computation time. The results presented in Sec. 3.3 substantiated the GPU's potential for parallel calculations. Although there are a couple of constraints associated with parallel programming on the GPU, which causes much more effort for implementation, the very high reduction in computation time is worth it. Seizing the example in Sec. 1 again, the computation time is now decreased to about 5 min. Using a more powerful graphics card than the Geforce GTX 570 used for the investigation will lead to further reductions.

The algorithmic modification of the time-domain beamforming algorithm ”beamfactors” described in Sec. 3.4, revealed to be a powerful algorithmic improvement. Although, associated with certain constraints, which are satisfied by regular microphone array measurements on moving trains with known velocity, a high reduction in computation time without parallelization is achieved. Parallelizing the algorithm on CPU and GPU leads to further reductions.

The results presented are the base for further work. It was a first attempt to optimize the evaluation/computation time of the time-domain beamforming algorithm used. The capabilities of parallelization on CPU and GPU were examined, revealing that the graphics card is the most powerful tool. Compared to the computation time of all other approaches presented, summarized in Fig. 7, parallelizing the time-domain beamforming algorithm on the GPU leads to the highest reduction in computation time. Although beamfactors on GPU were less efficient than the pure beamforming algorithm, the algorithmic improvements of beamfactors showed to have a lot potential on reducing the computation time when used on CPU. Probably, further modifications in the GPU implementation of beamfactors will lead to a higher reduction of computation time than presented.

REFERENCES

[1] D. H. Johnson and D. E. Dudgeon. *Array Signal Processing: Concepts and Techniques*. P T R Prentice-Hall, Inc., 1993.

[2] C. Nilsen and I. Hafizovic. "Digital beamforming using a gpu." In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing, (ICASSP)*, pages 609–612. 2009. ISSN 1520-6149. doi:10.1109/ICASSP.2009.4959657.

[3] G. Zechel, A. Zeibig, and M. Beitelschmidt. "Time-domain beamforming on moving objects with known trajectories." In *Proceedings of the 3rd Berlin Beamforming Conference*. Berlin, Germany, 2010. URL `http://bebec.eu/Downloads/BeBeC2010/Papers/BeBeC-2010-12.pdf`.