BeBeC-2014-08

REDUCING BEAMFORMING CALCULATION TIME WITH GPU ACCELERATED ALGORITHMS

Steffen Schmidt
GFaI eV
Volmerstraße 3, 12489, Berlin, Germany

ABSTRACT

Beamforming algorithms make high demands on the computer hardware and the computation time is an important factor for the assessment of this method. This paper describes techniques for optimizing the implementation of beamforming algorithms in regard to calculation time. The main focus is on using the Graphic Processing Unit for accelerating beamforming. After a brief introduction to general purpose GPU computing, the realization of time-domain, frequency-domain and order-domain beamforming on the GPU is explained. Several benchmarks have been run and the results show that all three algorithms can gain shorter calculation time on the GPU. The tests also examine the differences in performance of the two main development toolkits, CUDA and OpenCL. Furthermore, the impact of the GPU hardware is analysed regarding computational power and scalability.

1 INTRODUCTION

Although modern computers provide high computational power, the calculation time is still a crucial factor for beamforming algorithms [1] and a criterion for the acceptance of beamforming in practice.

The complexity of a beamforming calculation and therefore the time needed for calculating the image depends on several parameters. The main factor is the resolution of the result image respectively the 3D model. The more single result points must be calculated the more time is required for the computation. Another factor is the complexity of the input data. The number of microphone channels as well as the number of samples in each channel has an impact on the computation time. In frequency-domain beamforming, the width of the analysed frequency band, i.e. the number of the integrated FFT result coefficients, also increases the computational costs.

Advanced beamforming algorithms, such as Acoustic Movie, Acoustic Eraser [2] and HDR [3], consist of several single beamforming calculations. So the calculation time rises with the complexity of these algorithms.

2 OPTIMIZATIONS ON THE CPU

For reducing computation time, several techniques have been evaluated and implemented in the NoiseImage software. A very pragmatic approach is to reuse already existing results. If an acoustic image from frequency-domain beamforming has been calculated for 5 to 7 kHz, an acoustic image for 5 to 10 kHz can be created by adding only the missing frequencies.

The most common practice for accelerating computations is multi-threading. Beamforming algorithms can be distributed over several concurrent threads, each calculating a subset of the result points. As figure 1 shows, the computation time decreases with the increasing number of parallel threads. This speed-up is not proportional, because a rising number of threads produces more overhead for synchronizing the threads. The optimal performance is reached if the number of threads coincides with the number of available physical processor cores.

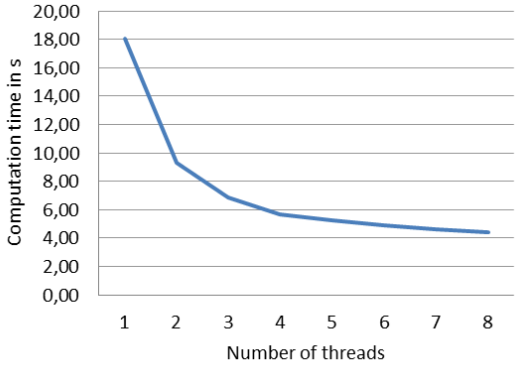

Fig.1. Calculation time for 3D acoustic photo as a function of number of parallel threads
Intel Core i7-2600 CPU 4 cores (32 channels, 50 ms, 192 kHz sampling rate, time-domain)

Another scope for optimizations is the source code itself. Setting the code optimization option of the Visual Studio C++ compiler to 'Maximize speed' leads to a reduced computing time by a factor of 5 for time-domain beamforming.

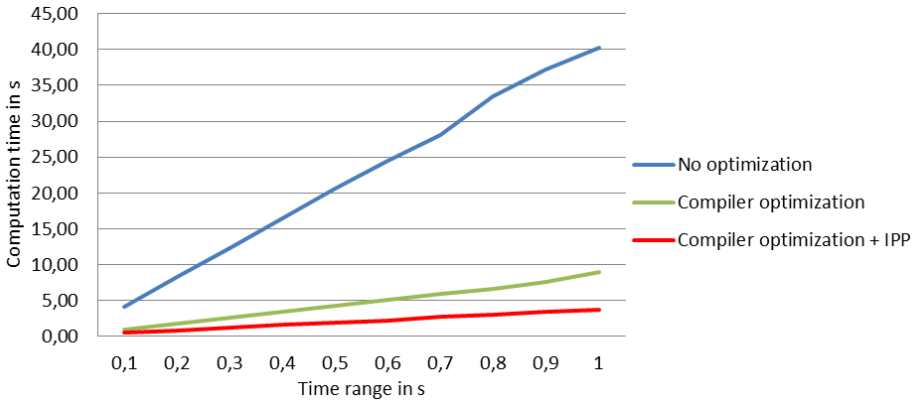

Fig.2. Effect of compiler optimization and IPP on calculation time for 2D acoustic photo
Intel Core i7-2600 CPU 4 cores (48 channels, 192 kHz sampling rate, time-domain, 8 threads)

Additionally, the Intel Integrated Performance Primitives (IPP) [4] have been used. This library provides highly optimized software functions by making use of all available processor features. It contains functions for signal processing, image processing, vector and

matrix operations and other applications. For the beamforming algorithms, all operations on large data sets have been replaced by corresponding IPP functions. This way the processing time was reduced by factor of 2 again. (Fig. 2)

The delay-and-sum beamforming algorithm requires adding up samples from all input channels. So, for every single result point the same data has to be read from memory. Contemporary CPUs are equipped with cache that allows fast access on frequently used data. To take full advantage of the cache memory, the processing of the samples is split into data chunks that fit completely into the available L2 and L3 caches. Thus the channel data is read only once from RAM for the first result point and is available in cache for all other result points. This optimization is a massive performance gain mainly on big data sets. Figure 3 shows that the computing time, depending on the amount of data, exponentially increases, if no caching mechanisms are used. With cache optimization, it increases linearly.

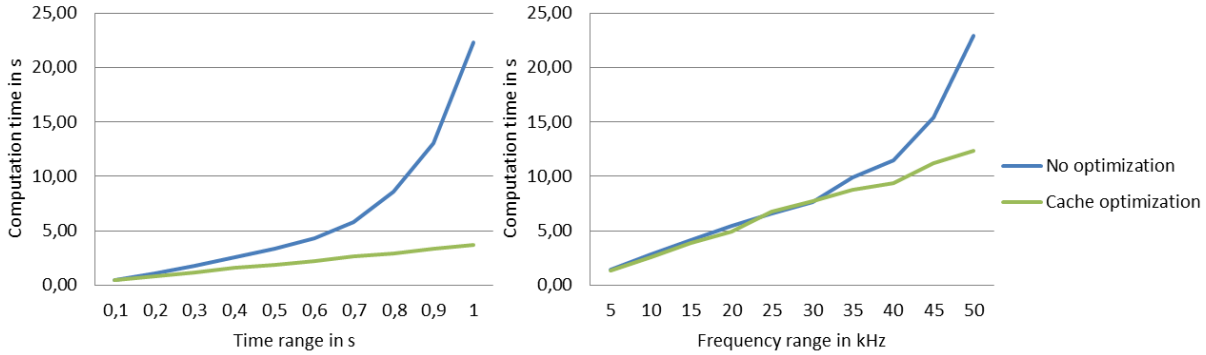

Fig.3. Effect of cache optimization on calculation time for 2D acoustic photo in time-domain (left) and frequency-domain (right)
Intel Core i7-2600 CPU 4 cores (48 channels, 192 kHz sampling rate, 8 threads)

3 GENERAL PURPOSE COMPUTATION ON GPU

Originally designed for 3D graphics, Graphics Processing Units (GPU) became more flexible with the introduction of shader technology. A shader is a programmable processing unit for implementing geometry and pixel transformation algorithms that run directly on the GPU. Since these pixel calculations can be performed in parallel very well, GPUs have a large number of shaders.

General Purpose Computation on Graphics Processing Unit (GPGPU) makes use of the processing power of GPUs for solving other problems than graphics. Thereby the GPU can be seen as coprocessor to the CPU for highly parallel computations. There are several technologies for implementing GPGPU. The most common frameworks are CUDA [5] and OpenCL [6]. CUDA is developed by NVIDIA and is limited to the hardware of this vendor, while OpenCL is a platform-independent approach.

Both frameworks provide specific programming languages for writing kernels, the program code that runs on the GPU. In CUDA, a kernel is executed by a grid of parallel threads, which is subdivided into blocks. Each block contains the same number of threads running the kernel code. The threads in a block reside on the same graphics processor core [7].

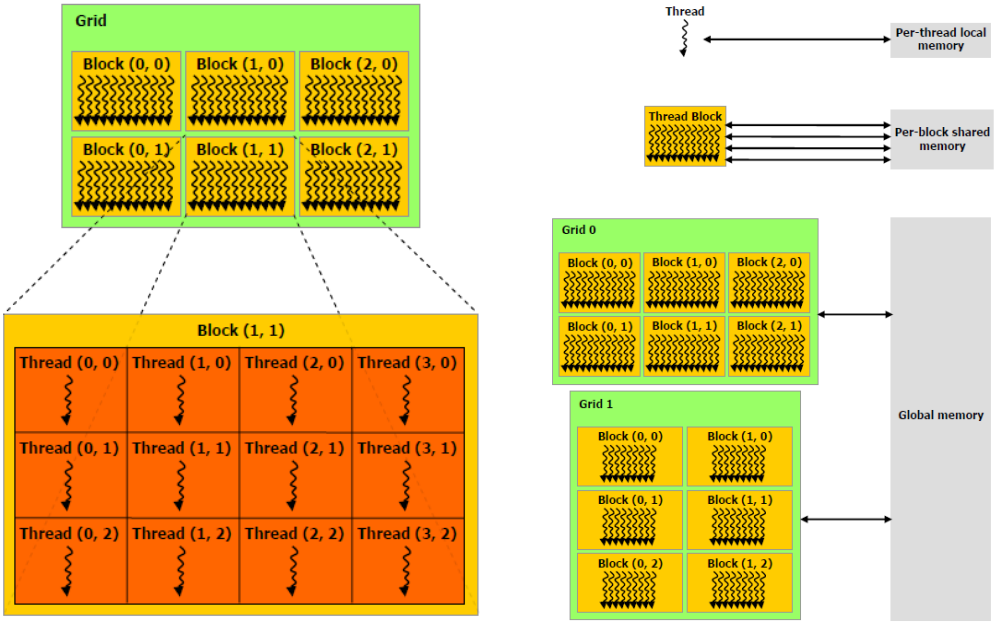

Fig.4.Thread hierarchy and memory classification for CUDA (taken from [7])

The kernel threads operate on the graphics memory, which is classified into different memory types. The global memory is accessible by all threads and is persistent across kernel calls. This memory space is used for the transfer of the input and result data between CPU memory and GPU memory. Every thread has its private local memory and each block has shared memory available to all threads in the block. Additionally, constant and texture memory can be used. OpenCL uses the same concepts for threads and memory as described above, with different naming [8].

For implementing an algorithm on the GPU, several aspects must be considered. As the main focus in GPU computing is on parallelization, the algorithm must be decomposable into parallel tasks, each calculating a part of the result. To obtain optimal performance the correct memory types must be chosen, which provide different access characteristics and access speed. There are also some drawbacks that must be considered. While a computation is running on the GPU, this unit is not available for rendering graphic updates. As a result the display freezes during this time. If the freeze lasts more than 2 seconds, it is detected by the "Timeout Detection and Recovery" mechanism of Windows which reinitializes the graphics driver and resets the GPU. To prevent this behavior, the display should be connected to an additional graphics card that is not used for computing.

4 IMPLEMENTING BEAMFORMING ON THE GPU

For an internal evaluation we have implemented time-domain beamforming with CUDA and OpenCL. The input data that has to be copied to the GPU are the microphone channel data and the delays for every channel and point. The parallelization is done on the result points, so the value of a point is calculated by a single block. The task of a thread is to add the samples of all channels and write the sum to the per-block shared memory. After all threads have finished, the partial results are combined.

For optimization, the point delays are copied to shared memory, which allows a higher access speed. On OpenCL, Built-In-Vectors have been used. An idea we have withdrawn, is

to use texture memory for the channel data on the CUDA implementation. The results showed that it is slower than working in global memory. Both implementations have been evaluated on a single standard desktop GPU and on a system with 4 high-end GPUs and an additional desktop GPU connected to the monitor. In the multi GPU configuration, every device computes a subset of the result points.

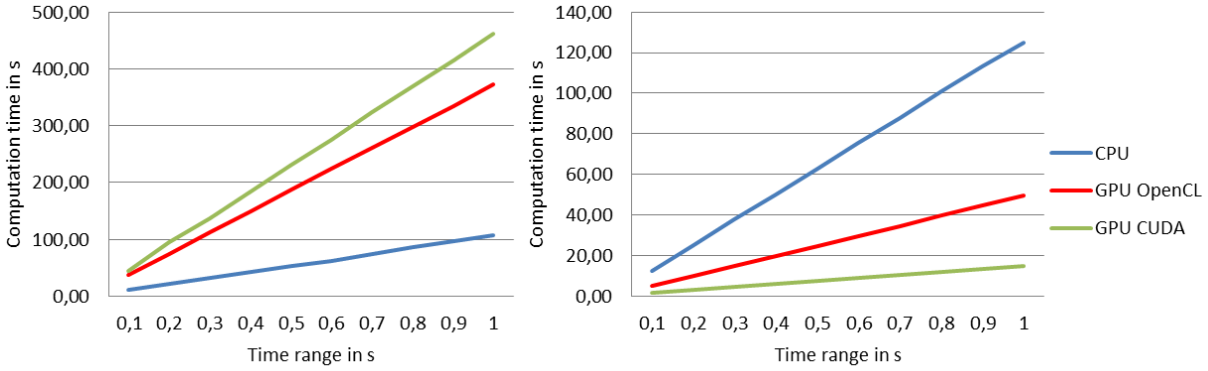

Fig.5.Calculation time for 3D acoustic photo in time-domain (48 channels, 192 kHz sampling rate) Left: NVIDIA GeForce GT 430 (96 CUDA cores) vs. Intel Core i7-2600 CPU 4 cores (8 threads) Right: 4x NVIDIA GeForce GTX 590 (4x512 CUDA cores) + display GPU vs. Intel Core i7-980 CPU 6 cores (12 threads)

As Figure 5 shows, the desktop GPU is much slower than the CPU implementation. In contrast, on the high-end system the OpenCL implementation is 2.5 times faster and CUDA is more than 8 times faster than the CPU version, which is already optimized by the methods mentioned above. The studies on the high end system have also shown that even a single GPU is faster than the CPU and the performance gain scales linearly with the number of GPUs used. (Fig 6)

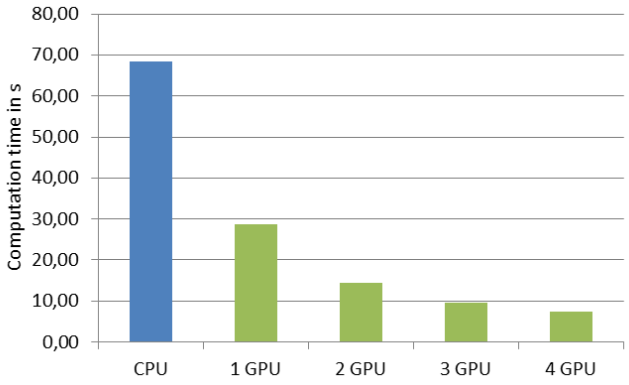

Fig.6. Comparison of calculation time for 3D acoustic photo on CPU (Intel Core i7-980 CPU 6 cores (12 threads)) and multiple CUDA GPUs (NVIDIA GeForce GTX 590) (48 channels, 192 kHz sampling rate, 500 ms, time-domain)

The other result of this comparison is that the CUDA implementation reaches shorter computation times than OpenCL on the GTX 590. The reason for this is probably that CUDA makes more efficient use of the NVIDA hardware, because it is not a generic approach as

OpenCL. We decided to concentrate on CUDA and the 4 GPU system for further implementations.

5 FREQUENCY-DOMAIN AND ORDER-DOMAIN BEAMFORMING

The algorithms for beamforming in frequency-domain and order-domain have also been transferred into CUDA implementations. In both cases, the result points are distributed over the kernel blocks again. For frequency-domain, the input data is not the raw channel data but the spectral result coefficients of the FFT, which is performed on the CPU. So the magnitudes and phases, the circular frequencies and again the delays must be copied to global GPU memory. A single kernel thread processes one coefficient of all channels and computes a partial result.

Order-domain beamforming is a technique combining spectral beamforming and order analysis of rotating parts. The acoustic data is resampled according to the rotation speed which must be recorded from a rotation sensor during the measurement. The input data for the CUDA kernel include the channel data, delays and the RPM data. The kernel reconstructs the time function of the processed point and does the resampling. Because the resampled time functions already reside in GPU memory, the spectral decomposition is performed on the GPU, too. For this the CUFFT library [9] is used, a CUDA accelerated FFT that is included in the SDK. Finally, spectral beamforming is done on the FFT result coefficients.

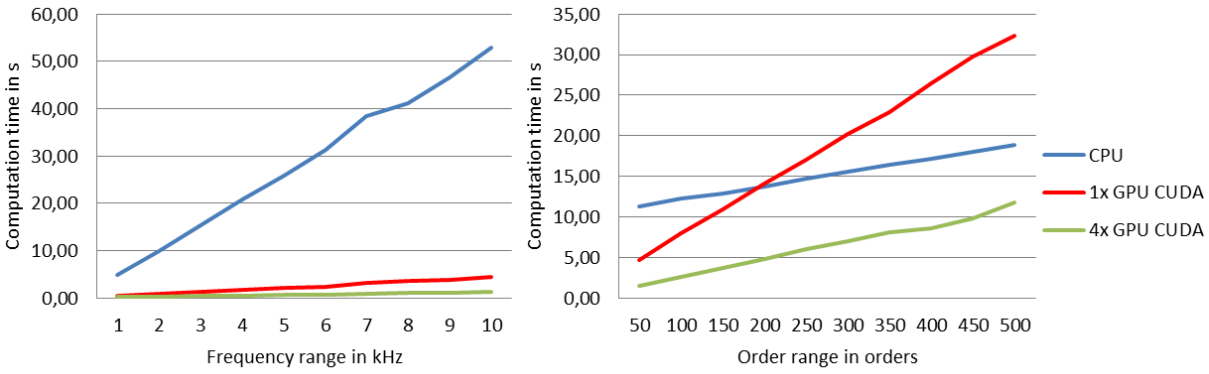

Fig.7.Calculation time for 3D acoustic photo in frequency-domain (left) and order-domain (right) 4x NVIDIA GeForce GTX 590 (4x512 CUDA cores) vs. Intel Core i7-980 CPU 6 cores (12 threads) (48 channels, 192 kHz sampling rate, 500 ms time range)

Particularly in frequency-domain a remarkable performance gain was achieved. Figure 7 depicts a decrease of calculation time up to factor 40. In order-domain the speed-up factor is 7 for the range of 50 orders. But as figure 7 shows, the factor decreases while the order range rises. So, for a range of 350 orders the GPU implementation is 2 times faster than the CPU. The reason for that is the limitation of available memory on the GPU. The order-domain algorithm requires the complete channel data and the resampled time functions to reside in GPU memory. The remaining memory is available for the per-point result data. Because the size of the result data grows with the size of the calculated order range, fewer points can be computed in parallel.

6 CONCLUSION

Different beamforming algorithms have been implemented for running on the GPU by using CUDA. The result points of an acoustic map can be calculated independently from each other, so a high degree of parallelization can be achieved. Test results have proved that standard desktop graphic cards do not have enough computational power to perform beamforming faster than the CPU. Therefore GPUs with a large number of cores are required. On our test system with 4 GPUs all three algorithms run faster on the GPU than on the CPU. Currently, the trend in hardware development is to put more cores on single GPUs and to provide GPUs that are optimized for general purpose computing, without any graphic functionality. So, more increased performance can be expected.

Complex beamforming calculations can also make use of the CUDA accelerated algorithms. So, in NoiseImage on our test system, a 3D acoustic movie[1] that is calculated in 510s on the CPU can be done in 56,7s on the GPU. A 3D HDR photo[2] needs 27,2s on the GPU compared to 197,3s on the CPU.

REFERENCES

[1] D.H. Johnson and D.E. Dudgeon. “Array Signal Processing. Concepts and Techniques”, PTR Prentice Hall, 1993

[2] D. Döbler and Dr.R. Schröder. “Contrast improvement of acoustic maps by successive deletion of the main sources”, BeBeC-2010-19, Proc of the 3rd Berlin Beamforming Conference, 2010

[3] D. Döbler and Dr.R. Schröder. “Contrast improvement and sound reconstruction of quiet sound sources using a high dynamic range algorithm”, BeBeC-2012-12, Proc of the 4th Berlin Beamforming Conference, 2012

[4] http://software.intel.com/en-us/intel-ipp

[5] http://www.nvidia.com/object/cuda_home_new.html

[6] https://www.khronos.org/opencl/

[7] NVIDIA Corporation. “NVIDIA CUDA C Programming Guide Version 4.0“, 9-11, 5/6/2011

[8] A. Munshi, B.R. Gaster, T.G. Mattson, J.Fung, D.Ginsburg. „OpenCL Programming Guide“, Addison Wesley, 2012

[9] J.Sanders and E. Kandrot. „CUDA by Example – An Introduction to General-Purpose GPU Programming“, Addison Wesley, 2011

[1] 48 channels, 192 kHz sampling rate, time-domain, 92 frames, 25 fps

[2] 48 channels, 192 kHz sampling rate, time-domain, 100 ms, max 8 HDR iterations