DIGITAL BEAMFORMING USING A GPU

Carl-Inge Colombo Nilsen, Ines Hafizovic

Department of Informatics, University of Oslo, Norway
SquareHead Technology AS, Norway

ABSTRACT

In this paper we investigate the use of GPUs as digital beamformers. We specify a parallel implementation of a beamformer in time and frequency domain and measure its performance. We also give examples of the processing limits of NVIDIA Geforce 8800 GPU with respect to application parameters: number of sensors, sampling frequency, bandwidth, and number of simultaneous beams. The results are compared to those of algorithms similarly implemented on a Intel Xeon CPU. We find that the GPU is able to process a larger amount of information than the CPU, and that it can be used as a digital beamformer for arrays with a large number of elements sampled at high rates. Exact results are given for the abovementioned application parameters.

Index Terms— Array signal processing, parallel processing.

1. INTRODUCTION

In many signal processing applications it is necessary to perform *beamforming* on the data received from an *array* of sensors. The objective of the beamformer is to align in time/phase the signals arriving at the sensors from a certain direction, so that they can be added coherently. This means that signals coming from all other directions will be added incoherently, and as a consequence attenuated.

Beamforming is a data-intensive task, in which samples from large number of sensors are combined to one or more output channels (beams) through different arithmetic operations, depending on the algorithm in question. Most beamforming algorithms for sensor arrays of moderate size have traditionally not been well suited for implementation on general (application non-specific), programmable computers; and are therefore usually confined to DSP, FPGA, or ASIC implementations. Recently, a lot of attention has been paid to implementing data-intensive algorithms on Graphical Processing Units (GPUs), e.g. computer vision [1] and medical image reconstruction [2]. Implementing non-graphical algorithms has been facilitated by the release of the CUDA framework by NVIDIA [3]. In this article, we investigate the applicability of GPUs to digital beamforming.

2. DIGITAL BEAMFORMING

Our beamforming scenario will typically cover an array of M elements, generating an output signal of N samples from input signals. In its most basic form, beamforming can be performed as delay-and-sum:

$$y_l[n] = \sum_{m=1}^{M} w_{l,m}[n] x_m[n - \Delta_{l,m}[n]]. \qquad (1)$$

Here, $y_l[n]$ is the output signal for the l*th* beam, $x_m[n]$ is the input signal from element m at time n, $w_{l,m}[n]$ is the weight applied to the signal from element m at time n when contributing to the l*th* beam, and $\Delta_m[n]$ is the delay applied to the signal from element m at time n when contributing to the l*th* beam. The n-dependence of the weights and delays can be disregarded in many applications. If the signal is complex and narrow-band, the beamformer can be expressed as

$$\vec{y} = \vec{w}^H X, \qquad (2)$$

where X is the data matrix, and $\vec{w}$ can represent: a) phase shifts and spatial tapering; b) optimal (e.g. Capon) weights, see [4]. The input data matrix can be written as

$$X = \begin{bmatrix} x_1[0] & x_1[1] & \cdots & x_1[N-1] \\ x_2[0] & x_2[1] & \cdots & x_2[N-1] \\ x_3[0] & x_3[1] & \cdots & x_3[N-1] \\ \vdots & \vdots & \vdots & \vdots \\ x_M[0] & x_M[1] & \cdots & x_M[N-1] \end{bmatrix} \Rightarrow [X]_{m,n} = x_m[n-1] \qquad (3)$$

On a computer, it can be stored in either a row-major or a column-major format, making access sequential across time or space, respectively.

As seen above, operations in beamforming algorithms can be written as linear algebra, which is a suitable form of implementation on the GPU.

3. EFFICIENT IMPLEMENTATION ON A GPU

3.1. The GPU as a general parallel processing model

We will not go into the details about GPU structure, as this is outside the scope of this paper. The interested reader is

referred to [5] for more information. Suffice it to say, the GPU can be seen as a parallel processor with two parallelization levels. On the first level, the tasks are divided across multiple processors with no means of communicating with each other. On the second level, the tasks on one processor are divided among different threads which can share information and be synchronized. In the CUDA framework, these levels are referred to as *blocks* and *threads*. In general, all processors will have access to one large memory pool (termed ”global” memory), supporting a low transfer rate. Each processor will additionally have its own, smaller memory pool (termed ”local” memory) supporting a high transfer rate. The initial input and final output is assumed to be located in the global memory. The execution of a parallel algorithm (called a kernel invocation) is initiated from a centralized control unit, which receives a notification when all processors have finished executing. The following should be taken into consideration when developing parallel algorithms:

- Information used in two different blocks *should* not be read from the same location in global memory at the same time. Information coming from two different blocks *must* not be written to the same location in global memory. This restricts parallelization.
- Information that is used by more than one thread in a block should be moved from global to local memory, before being used. This makes thread synchronization within a block necessary [6].

3.2. Parallel execution

If the GPU is to perform efficiently, algorithms must be suitably parallelized. We will talk about two levels of parallelization, level 1 (PL1) being the division between multiprocessors/blocks and level 2 (PL2) being the division between threads executing on each multiprocessor/block. Different parallelization schemes that have been considered are:

1. PL1-parallelization in space (1): Each block fully forms one or more beams using all samples from all elements. This is problematic because two different beams are usually formed from the same input samples. The advantage is that the output data from each block will always be stored in different places in global memory.
2. PL1-parallelization in space (2): Each block computes the contribution to all beams for all time indices for one single element. This is an impracticable approach, as neither reading from or writing to overlapping memory segments can be synchronized across processors and will lead to access conflicts.
3. PL1-parallelization in frequency: Each block operates on one single frequency component across all elements, and performs beamforming for all beams. Reading from and writing to global memory is completely without overlap across blocks. This approach is only suited for frequency domain beamforming algorithms. Suitably parallelized batched versions of FFT and IFFT algorithms are implemented independently of the beamforming process.
4. PL1-parallelization in time: Each block creates the output signal for one block in time, for all beams from all elements. This is the initially most satisfying solution with respect to parallelization, because all blocks read from and write to separate locations of global memory. It does however pose a problem for real-time systems, as a larger delay must be introduced in the system to fully exploit the multiple processors for parallelization.

It is possible to interchange schemes 1 and 4 by duplicating sample blocks in device memory, and acting on different instances of the same samples as if they were temporally disjoint blocks. The same applies to scheme 3, when exchanging temporal samples for DFT-coefficients.

3.3. Memory considerations

Data transfer between the CPU and the GPU is a time consuming operation, and should be avoided when possible. Additionally, data should be transferred as a few large blocks instead of a larger number of small ones. Accessing data from the global memory on the GPU is optimal when we allow operations across threads to be *coalesced*. This means that a sequence of N different memory accesses for single samples is interpreted as one large memory access covering all N samples. This is only possible when the addresses are:

1. Sequential.
2. Aligned (meaning that the starting address must be a multiple of the alignment size, which in this case is 68 bytes or 16 single precision floating point values).

PL2-parallelization should therefore be performed in a dimension in which the data-matrix is sequential. We see from Eq. 2 that reasonable sequences would be across elements or temporal samples for each block. The question of alignment will be discussed later.

4. IMPLEMENTING A BEAMFORMER IN THE TIME DOMAIN

A time domain beamformer (TDBF) can be implemented directly from Eq. (1). To generate each output sample, the corresponding input samples from all channels are fetched from memory, multiplied by different weights, and added together.

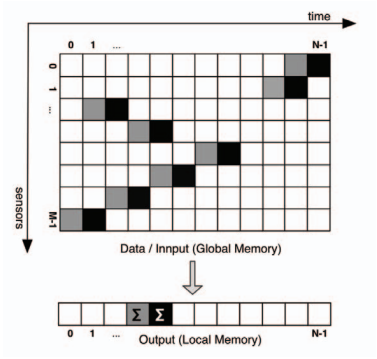

Fig. 1: *Naive time domain implementation*. Memory access when creating one output sample is generally scattered across a block of samples, regardless of the order in which the samples are stored.

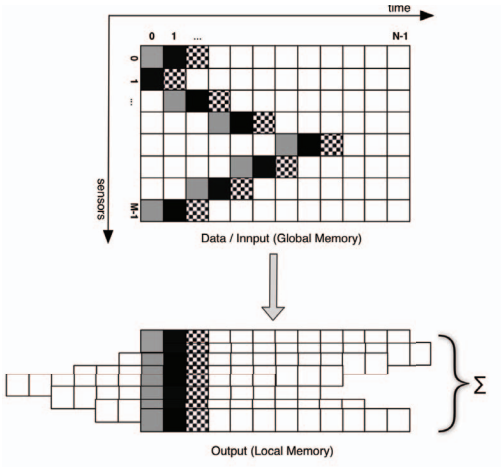

Fig. 2: *Optimal time domain implementation.* Samples are read from memory in their stored order, shifted, and placed in local memory. Summations with intermediate results are performed upon placement.

A disadvantage of TDBF with respect to GPU implementation is that memory access is not sequential across channels, as shown in Fig. 1, which makes parallelization in the spatial dimension less efficient due to time consuming memory access. Memory access across time for one single channel is sequential for time-invariant delays, but generally not aligned unless the difference between delays are restricted to being integer multiples of the alignment block size. This can be fixed by moving an entire block of sequential and aligned samples from global to shared memory, performing the entire beamforming in local memory, and moving the beamformed output back to aligned global memory. This can be problematic, as the available pool of local memory is often quite small (16 kb per block on the Nvidia GeForce 8800). Another method is the following:

n=aA *where* a *is an integer;* A *is the alignment size;* ix_t *is the thread number from* 0 *to* T-1*;* ix_b *is the block number from* 0 *to* B-1.

1. Δ_{max} is $max_m\{del[ix_b,m]\}$
2. for m=1:M
   $local\_out[\Delta_{max} + ix_t - delay[ix_b,m]] \mathrel{+}= w[m]global\_in[n + ix_t]$
3. $global\_out[ix_t] = local\_out[2\Delta_{max} + ix_t]$

This implementation assures sequential and aligned memory access across time both when reading and writing samples. The delay has been moved from input to output, as illustrated in Fig. 2, which does not affect the result as long as there is a one-to-one index mapping between the output samples for one beam and the input samples from one channel. This is always the case when the delays are time invariant. The algorithm is visualized in Fig. 3, where the gray cells represent input samples that contribute to a block of output samples. By processing the aligned region from $n \in [\Delta_{max}, 3\Delta_{max} + N]$, and adding to results in the region $n \in [0, 4\Delta_{max} + N]$, we get N samples of usable output data in the region $n \in [2\Delta_{max}, 2\Delta_{max}+N]$. The region $n \in [2\Delta_{max}+N, 4\Delta_{max}+N]$ can be shifted to $n \in [0, 2\Delta_{max}]$ before the next iteration of the algorithm. We observe that global memory access is performed on lines 2 (reading) and 3 (writing), while the intermediate results are stored in local memory.

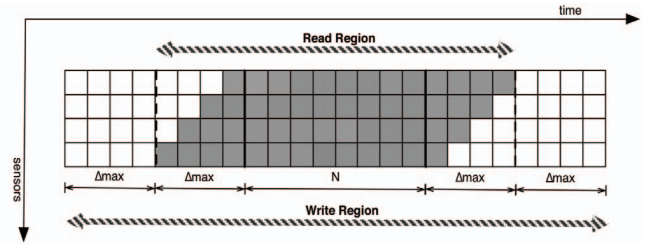

Fig. 3: *Memory access for TD beamformer*. Elements from the read region are placed within the write region. Only elements in the middle block of size N will be fully usable after each iteration.

5. IMPLEMENTING A BEAMFORMER IN THE FREQUENCY DOMAIN

We implement a frequency domain beamformer (FDBF) by taking the DFT of a block of samples, and then applying steering to each frequency component in the form of phase shifts. The frequency domain beamforming algorithm can be written as:

1. Perform MxN-point DFTs: $X_m[k] = DFT\{x_m[n]\}$, m=0,…,M-1, k=0,…,N-1
2. For each frequency bin k=0,..., N-1 $Y_l[k] = S_l X_m[k]$
3. Perform LxN-point IDFTs: $y_l[n] = IDFT\{Y_l[k]\}$

The disadvantage of FDBF is that it involves additional computational complexity due a DFT for each sensor and an IDFT for each output beam. These can be efficiently implemented as FFTs on both CPU and GPU. The general advantage of FDBF is that most advanced beamforming algorithms assume narrow-band signals, and in the discrete frequency domain each frequency of a broadband signal can

be processed independently as a narrow-band signal. It also allows most operations to be represented as matrix or vector multiplication, which can be quickly implemented using available functions. The advantage of FDBF with respect to GPU implementation is that data-access is always sequential in time and can be made aligned across sensors, which offers increased performance due to faster memory access. This is assuming that the data matrix of Eq. 2 is stored in a row-major format. Alignment is assured by padding the data with zeros so that the number of samples stored for one time index is a multiple of the alignment size.

6. RESULTS AND CONCLUSION

We have implemented TDBF and FDBF on both the CPU and GPU. The CPU implementation was done using Intel Math Kernel Library 10.0 (BLAS for FDBF and VML for TDBF) and the measurements were performed on a Xeon Quad-Core CPU. The GPU implementation was done using Nvidia CUDA (CUBLAS for FDBF) and the measurements were performed on a Nvidia GeForce 8800 GPU. Memory access was similarly sequential and aligned for both processors, to assure a fair comparison.

In Tab. I we have recorded the average running times for TDBF and FDBF algorithms when using arrays with sizes varying from 76 to 1216 elements. The arrays simulated are uniformly spaced linear arrays, which does not restrict the validity of the measurements as no special optimization due to their regular structure is applied. It is observed that the GPU performs TDBF faster than the CPU almost by an order of magnitude. The FDBF is performed using about 16% of the time used by the CPU.

The number of samples generated by the beamformer, as reported in Tab. II, can contribute to either different beams or different time blocks. For instance when considering a 152-element array, the GPU can generate output of some 16 Msamples/sec. These samples can be divided into L beams for signals with sampling rates of 16/L MHz. If L=1, then signals with sampling rates up to 16 MHz are supported by this implementation.

These measurements should be regarded as indicative of actual performance, since these experiments have been performed on a single type of CPU/GPU and for a general beamformer. Results will vary when application-specific considerations are taken into account. The comparisons are made for what is thought to be fairly equal implementations of identical algorithms. Some points to consider are:

- The transfer of input and output samples to and from the GPU is time consuming for both GPU and CPU.
- The CPU time used to transfer data to and from the GPU is less than what is used by the CPU for performing the actual algorithm, thus freeing the CPU to do other work, like communicating with a user.

Based on the results presented in this paper, it is our conclusion that using the GPU is a viable solution for performing digital beamforming on off-the-shelf hardware, even for large arrays sampled at high rates.

TABLE I. RESULTS 1

Measured Running Times						
Array Size	***GPU TD(s)***	***GPU FD(s)***	***CPU TD(s)***	***CPU FD(s)***	***G/C TD(%)***	***G/C FD(%)***
76	***0.028***	***0.027***	***0.240***	***0.180***	***11.6***	***11.3***
152	***0.059***	***0.053***	***0.490***	***0.354***	***12.2***	***15.0***
304	***0.114***	***0.109***	***0.980***	***0.691***	***11.6***	***15.8***
608	***0.224***	***0.213***	***1.950***	***1.370***	***11.5***	***15.5***
1216	***0.445***	***0.423***	***3.890***	***2.720***	***11.5***	***15.5***

The numbers in columns two through five are given in seconds. The numbers in the last two columns describe the time ratio GPU to CPU. The measurements are based on 1 Msamples of data per sensor.

TABLE II. RESULTS 2

Processing Capabilities (Msamples/sec.)						
Array Size	***GPU TD***	***GPU FD***	***CPU TD***	***CPU FD***	***G/C TD(%)***	***G/C FD(%)***
76	***35.7***	***37.0***	***4.2***	***5.6***	***850***	***660***
152	***16.9***	***18.9***	***2.0***	***2.8***	***850***	***680***
304	***8.8***	***9.2***	***1.0***	***1.4***	***880***	***660***
608	***4.5***	***4.7***	***0.5***	***0.7***	***900***	***670***
1216	***2.2***	***2.4***	***0.3***	***0.4***	***730***	***600***

The numbers given in columns two through five are given in Msamples/second. The numbers in the last two columns describe the ratio GPU to CPU.

7. ACKNOWLEDGEMENT

The authors wish to thank SquareHead Technology AS for providing the necessary hardware and data for implementation and testing.

8. REFERENCES

[1] J. Fung, S. Mann, "Computer vision signal processing on graphics processing units," *in Proc. IEEE International Conference on Acoustics*, *Speech and Signal Processing*, vol. 5, pp. 79-88, May 2004.

[2] T. Sumanaweera, D. Liu, "Medical image reconstruction with the FFT," in *GPU Gems 2*, M. Pharr, Ed. Addison Wesley, 2005, ch. 48, pp. 765-784.

[3] CUDA Zone: http://www.nvidia.com/cuda/

[4] Van Trees, H.L., *Optimum Array Processing*. Wiley-Interscience, 2002.

[5] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, S. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," in Computer Graphics Forum, vol. 26, 2007, pp. 80-113.

[6] CUDA 2.0 Programming Guide: http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf