Paper presented at the 2014 IEEE International Ultrasonics Symposium

Synthetic Aperture Sequential Beamforming implemented on multi-core platforms

Thomas Kjeldsen[1], Lee Lassen[1], Martin Christian Hemmsen[2], Carsten Kjær[3], Borislav G. Tomov[2], Jesper Mosegaard[1], and Jørgen Arendt Jensen[2]

[1]Computer Graphics Lab, Alexandra Institute, DK-8200 Aarhus N, Denmark

[2]Center for Fast Ultrasound Imaging Dept. of Elec. Eng. Technical University of Denmark DK-2800 Lyngby, Denmark

[3]BK Medical, Mileparken 34, DK-2730 Herlev, Denmark

To be published in the Proceedings of 2014 IEEE International Ultrasonics Symposium.

Synthetic Aperture Sequential Beamforming implemented on multi-core platforms

Thomas Kjeldsen[1], Lee Lassen[1], Martin Christian Hemmsen[2], Carsten Kjær[3], Borislav G. Tomov[2], Jesper Mosegaard[1], and Jørgen Arendt Jensen[2]

[1]Computer Graphics Lab, Alexandra Institute, DK-8200 Aarhus N, Denmark

[2]Center for Fast Ultrasound Imaging Dept. of Elec. Eng. Technical University of Denmark DK-2800 Lyngby, Denmark

[3]BK Medical, Mileparken 34, DK-2730 Herlev, Denmark

***Abstract*—This paper compares several computational approaches to Synthetic Aperture Sequential Beamforming (SASB) targeting consumer level parallel processors such as multi-core CPUs and GPUs. The proposed implementations demonstrate that ultrasound imaging using SASB can be executed in real-time with a significant headroom for post-processing. The CPU implementations are optimized using Single Instruction Multiple Data (SIMD) instruction extensions and multithreading, and the GPU computations are performed using the APIs, OpenCL and OpenGL. The implementations include refocusing (dynamic focusing) of a set of fixed focused scan lines received from a BK Medical UltraView 800 scanner and subsequent image processing for B-mode imaging and rendering to screen. The benchmarking is performed using a clinically evaluated imaging setup consisting of 269 scan lines x 1472 complex samples (1.58 MB per frame, 16 frames per second) on an Intel Core i7 2600 CPU with an AMD HD7850 and a NVIDIA GTX680 GPU. The fastest CPU and GPU implementations use 14% and 1.3% of the real-time budget of 62 ms/frame, respectively. The maximum achieved processing rate is 1265 frames/s.**

I. Introduction

Originally, medical ultrasound imaging systems were implemented using analog beamformers [1]. Later, digital beamformers were introduced and traditionally implemented using dedicated hardware [2]. With the recent development in consumer level parallel processors, such as multi-core CPUs and graphics processing units (GPUs), it is advantageous to move the beamformer to software for improved flexibility and cost reduction [3].

Recent advances in beamformer technology and processing performance have resulted in increased interest in synthetic aperture imaging implemented on GPU [4], [5]. One challenge is, however, that beamforming requires a high data bandwidth. A typical system could have 128 channels and use a 12-bit 40 MHz sampling system. This generates $128{\times}40{\times}10^6\mathrm{Hz}{\times}2\mathrm{B} = 9.54\,\mathrm{GB/s}$ which is problematic to transfer into a consumer oriented system, e.g., for a GPU, data needs to be transferred over a relatively slow PCI-Express bus.

Synthetic Aperture Sequential Beamforming (SASB) [6] is a technique that produces image quality comparable to dynamical receive focusing [7], but requires much lower data bandwidth ($25.3\,\mathrm{MB/s}$ for the setup used in the present work). The low data bandwidth is achieved using a dual stage procedure which reduces the data rate enough to substitute the analog communication link between probe and processing unit with wireless technology [8]. This paper presents several implementations of SASB for real-time imaging utilizing modern multi-core platforms. The hypothesis is, that current consumer level parallel processors, multi-core CPU and GPU, are fast enough to execute SASB in real-time.

II. Materials and Methods

The basic idea in SASB is to use two separate beamformers. The first beamformer creates fixed focused scan lines. The final image is created in the second beamformer by refocusing these scan lines. This study assumes that the first beamformer is implemented in an ultrasound scanner which transfers the outcoming fixed-focused scan lines to a PC for refocusing and subsequent image processing for B-mode imaging. The particular PC processing steps in this work include upsampling and frequency shift of baseband I/Q data, beamforming of N_l lines with each N_S complex samples, amplitude detection, log-compression, and scan-conversion to a final image. All these second stage processing algorithms are implemented on the PC with special focus on efficient utilization of multcore CPUs or GPUs. The following theory section focuses mainly on the beamformation since this is the most complex and time consuming component.

A. Second stage beamformation

The second stage beamformer creates an image, sampled at the position $\vec{r}_{ip}$ as the sum over N_l first stage lines $s_k(t)$ with fixed focus at $r_{\mathrm{focus},k}$ [9], [10]

$$I(\vec{r}_{ip}) = \sum_{k=0}^{N_l-1} W(\vec{r}_{\mathrm{focus},k}, \vec{r}_{ip}) s_k(t(\vec{r}_{ip}, \vec{r}_{\mathrm{focus},k})), \quad (1)$$

where $W(\vec{r}_{\mathrm{focus},k}, \vec{r}_{ip})$ is an apodization function. t is the delay $t(\vec{r}_{ip}, \vec{r}_{\mathrm{focus},k}) = (2d_{\mathrm{focus}} \pm 2|\vec{r}_{ip} - \vec{r}_{\mathrm{focus},k}|)/c$ where c is the speed of sound, d_{focus} is the focus depth, and the sign depends on whether the image point is above or below the focus point.

For a convex array, the image samples are conveniently expressed in a polar coordinate system $\vec{r} = (r, \phi)$ as shown in Fig. 1, and calculated as

$$I(r, \phi) = \sum_{k=0}^{N_l-1} W(r, |\phi - \phi_k|) s_k(t(r, \phi - \phi_k)), \quad (2)$$

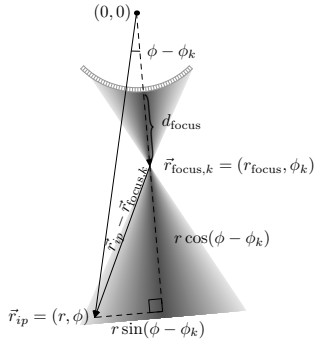

Fig. 1. Polar coordinates used in (2)-(5). The kth first stage line affects image points in the coloured area. The apodization weight is indicated by the color intensity.

where ϕ_k is the direction of the kth scanline. The delay is

$$t(r,\phi) = \frac{2}{c}\left(d_{\text{focus}} \pm \sqrt{r^2 + r_{\text{focus}}^2 - 2rr_{\text{focus}}\cos\phi}\right), \quad (3)$$

and the weight W depends only on the axial and lateral distances between the focus and the image points as indicated by the dashed lines in Fig. 1. It is now assumed that the image points are chosen along the scanlines and that the angular spacing between all consecutive scanlines is constant, $\Delta\phi$

$$I(r_i,\phi_j) = \sum_{k=0}^{N_l-1} W(r_i, |k-j|\Delta\phi)s_k(t(r_i,|k-j|\Delta\phi)). \quad (4)$$

Changing the summation index so that $k' = k - j$ and rearranging the terms in the sum

$$I(r_i,\phi_j) = W_{i0}s_j(t_{i0}) + \sum_{k'=1}^{N(r_i)} W_{ik'}\left[s_{j+k'}(t_{ik'}) + s_{j-k'}(t_{ik'})\right], \quad (5)$$

i.e., the apodization weights and the delays need only to be known at discrete values. Here $W_{ik} = W(r_i, k\Delta\phi)$ and $t_{ik} = t(r_i, k\Delta\phi)$, which can easily be precomputed for a given set of scanning parameters. In (5) the depth-dependent upper limit of the sum $N(r_i)$ indicates that image formation in $\vec{r}_{ip}$ is only affected by a limited set of neighboring lines [9]. Furthermore, $s_j(t)$ is defined to be zero for $j < 0$ and $j \geq N_l$.

The second stage beamforming formulation in (5) essentially only consists of table lookups, interpolations into $s_k(t)$, and a number of products and summations. While this type of algorithm is quite data intensive, it only involves basic arithmetic operations. Hence, efficient implementations depend both on fast memory access and parallel arithmetic operations.

Even though the present discussion focuses on a convex array in polar coordinates, an expression similar to (5) is easily derived in Cartesian coordinates for a linear array. The discussions below will, therefore, also apply to linear arrays.

B. Optimized SIMD implementation

Modern x86 CPUs support Single Instruction Multiple Data (SIMD) instruction extensions, such as Streaming SIMD Extensions (SSE) or Advanced Vector Extensions (AVX). These

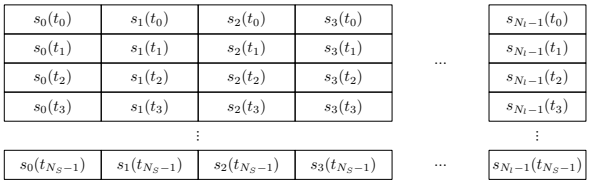

Fig. 2. Memory layout for N_l first stage lines with N_S temporal samples. The rows must be contiguous in memory for efficient SIMD utilization.

```
for i ← 0 : N_s − 1 do                                   ▷ Loop over depth samples
    for j ← 0 : N_l/4 − 1 do                             ▷ Loop over every 4th scanline
        I ← 0                                            ▷ I is an SSE register
        for k ← 0 : N_l − 1 do
            W ← weight[i, k]
            if W = 0 then
                break
            end if
            t ← delay[i, k]
            s ← _mm_load_ps(samples[t, 4 ∗ j + k])       ▷ Load aligned floats to SSE register
            I ← I + W ∗ s
            if k > 0 then
                s ← _mm_load_ps(samples[t, 4 ∗ j − k])
                I ← I + W ∗ s
            end if
        end for
    end for
end for
```

Fig. 3. Beamformation with SIMD optimizations. SIMD registers are indicated by bold faces.

extensions enable certain operations, e.g. load, store, and arithmetics, on four (SSE) or eight (AVX) single precision floating point numbers in a single instruction. However, these extensions can only be fully utilized if data is properly aligned in memory. For simplicity, the formulas below assume four way parallelism, corresponding to SSE. Generalization to eight way parallelism for AVX is straightforward.

One way to use SIMD is to beamform four image points in parallel. If these image points are chosen as consecutive points in the angular coordinate $\{\phi_{4j}, \phi_{4j+1}, \phi_{4j+2}, \phi_{4j+3}\}$, (5) can be written in vectorized form

$$\begin{bmatrix} I(r_i,\phi_{4j+0}) \\ I(r_i,\phi_{4j+1}) \\ I(r_i,\phi_{4j+2}) \\ I(r_i,\phi_{4j+3}) \end{bmatrix} = W_{i0}\begin{bmatrix} s_{4j+0}(t_{i0}) \\ s_{4j+1}(t_{i0}) \\ s_{4j+2}(t_{i0}) \\ s_{4j+3}(t_{i0}) \end{bmatrix} + \sum_{k=1}^{N(r_i)} W_{ik}\left(\begin{bmatrix} s_{k+4j+0}(t_{ik}) \\ s_{k+4j+1}(t_{ik}) \\ s_{k+4j+2}(t_{ik}) \\ s_{k+4j+3}(t_{ik}) \end{bmatrix} + \begin{bmatrix} s_{-k+4j+0}(t_{ik}) \\ s_{-k+4j+1}(t_{ik}) \\ s_{-k+4j+2}(t_{ik}) \\ s_{-k+4j+3}(t_{ik}) \end{bmatrix}\right). \quad (6)$$

Now it is assumed that the N_l first stage lines are sampled in N_S temporal samples, and these samples are stored in memory (cf. Fig. 2) in row-major format. Note that emissions are most naturally received sequentially, i.e., column by column in Fig. 2 and, hence, the assumed layout requires an initial transposition in memory. The row-major format ensures that all vectors on the right hand side of (6) lie contiguously in memory, at least if a nearest neighbor interpolation scheme is chosen for $s_k(t)$. Other interpolation schemes that evaluate $s_k(t)$ as a weighed sum of samples $s_k(t_{\text{sampled}})$ in the neighborhood of t also only involve vectors that are contiguous in memory. The workflow is summarized in pseudocode in Fig. 3, where the loop over j is reduced by a factor of four

compared to a non-SIMD version. This lowers the instruction count for the total loop significantly, which often leads to improved performance.

In addition to SIMD, current CPUs also contain multiple cores that can operate concurrently in separate threads. It is very easy to distribute the workload when beamforming a full image by, e.g., distributing the radial coordinates in separate threads. The present work uses OpenMP to distribute the outer loop in Fig. 3 among multiple threads.

C. Optimized GPU implementation

Current high-end Graphics Processing Units (GPUs) are characterized by large internal memory bandwidths and more than one thousand parallel processing units. This paper presents GPU results using the two high level APIs, OpenCL and OpenGL. OpenCL is specifically designed for computations using the GPU, while OpenGL was originally intended for pure graphics applications. However, OpenGL can often also be used for more general purpose computations, e.g., by calculating individual pixel values such as (5) in fragment shaders, and storing results in offscreen framebuffers. The key benefit of OpenGL over OpenCL is that OpenGL is supported on a much broader range of devices, including mobile devices.

Obtaining a reasonably good performance on current GPUs turns out to be somewhat simpler than the CPU implementation described in Sec. II-B. The reason is that the data in Fig. 2 can be stored in texture memory, which is cache optimized for memory reads that are close both vertically and horizontally. This caching mechanism turns out to be very efficient for the present case where $s_k(t)$ is sampled repeatedly for adjacent lines k. An actual profiling shows that the cache hit ratio is more than 96% for the present case. Another advantage of texture memory is that simple interpolation schemes, such as nearest neighbor and bilinear, are implemented in hardware.

Utilization of texture memory provides good performance without much programming effort. Even higher performance can be achieved by hand-optimizing memory access and compute core configurations to fit the underlying hardware, e.g., by using local memory cache within an OpenCL workgroup.

One common concern about GPU computing for real-time processing is that data must be uploaded continuously from main memory to GPU memory through a relatively slow PCI-Express bus. The present work tries to hide this transfer time by implementing a double buffered asynchronous upload where upload of one frame is overlapped with the processing of the preceeding frame in another buffer. Notice, however, that the SASB algorithm itself efficiently reduces this problem, because a significant data reduction occurs early in the pipeline.

III. Results

For the real-time implementation, image sequences are acquired using an STI 3ML 3.5CLA192 convex transducer, and consist of 269 emissions for each frame. The scanner beamforms the received echo signals using a fixed receive profile with subsequent transformation to baseband I/Q data. This data is sampled 1472 times per emission and is then transferred to a PC for second stage processing. The scanner acquisition frame rate is 16 Hz, which requires that all processing steps on the PC must be completed in 62 ms for real-time performance. For benchmarking the maximum computation rate of the various implementations, a 24-frame prerecorded data set [11] is stored locally on the second stage processing PC which makes it possible to simulate acquisition rates of more than 1200 Hz. The recorded data set is acquired using the same scan setup as the real-time implementation described above.

All optimized implementations are validated with a straightforward calculation of (1) using Matlab. Figure 4 shows B-mode images of the same data set generated by (a) Matlab, (b) multithreaded SIMD code, (c) the GPU using OpenGL, and (d) the GPU using OpenCL. All images agree very well. Compared to (a), the root mean square errors are (b) 0.0044, (c) 0.0042, and (d) 0.0040 when the 8-bit grayscale is mapped to [0,1]. The corresponding peak signal to noise ratios are 47.23dB, 47.59dB, and 48.01dB for SIMD, OpenGL, and OpenCL compared to Matlab, respectively. The small differences can be explained by the fact that the optimized codes use only up to 32-bit numerical representation, while Matlab works entirely in 64 bits double precision.

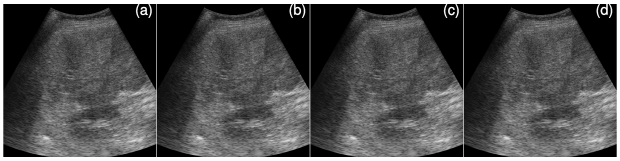

Fig. 4. In-vivo B-mode images generated by (a) Matlab, (b) multithreaded SIMD code written in C, (c) GPU using OpenGL, and (d) GPU using OpenCL. The dynamic range is 60 dB.

A. Evaluation of the processing performance

Benchmark timings are obtained with an Intel Core i7 2600 CPU (8 GB DDR3 memory) with an AMD HD7850 and an Nvidia GTX 680 GPU (both with 2 GB GDDR5 memory) connected on a PCI-Express 2.0x16 port.

The refocusing of the fixed focused scan lines and subsequent image processing for B-mode imaging are implemented as shown in Fig. 5. The CPU code is written in C with SIMD intrinsics and OpenMP multithreading. Timings are measured with the C++ std::chrono::steady_clock class for each subprocess in Fig. 5 (a) as an average over 1000 repetitions. The CPU has four hyperthreaded cores which provide a total of eight logical processors to the operating system and, correspondingly, CPU benchmarks are performed with up to eight threads. Each of the three last GPU passes in Fig. 5 (b) corresponds to an OpenCL kernel or an OpenGL shader program where intermediate results are rendered to an OpenCL image or an OpenGL framebuffer object. The reason why the GPU workflow contains fewer steps than the CPU is that it is relatively expensive to change framebuffer state and, thus, it is desirable to minimize the number of passes. High precision timings on the GPUs are measured with AMD CodeXL, AMD GPU PerfStudio, and NVIDIA Nsight.

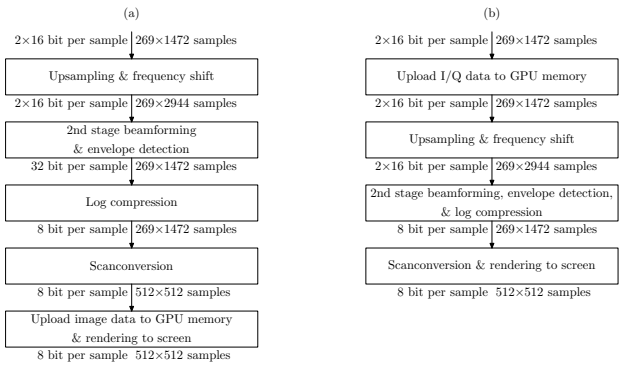

Fig. 5. Workflow diagram for the CPU (a) and GPU (b) implementations.

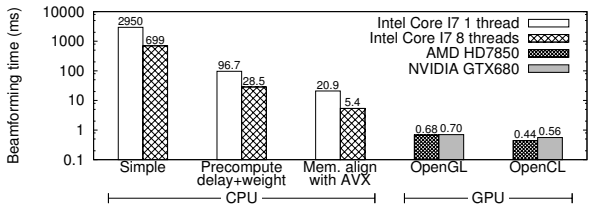

Fig. 6. Beamformation time with different levels of optimization. All CPU codes are written in C and compiled with GCC 4.8 using "-O3" compile flags. The three CPU series "Simple", "Precompute", and "Mem. align" correspond to (4), (5), and (6), respectively. Only the final, optimized OpenGL and OpenCL GPU results are shown. Note the logarithmic scale.

Figure 6 shows how various optimization steps improve the beamformation timing. For a well written initial C implementation of (4), the beamformation takes 2950 ms (699 ms with multithreading), i.e., far from the real-time limit of 62 ms for all second stage operations. Precomputation of delays and weights, and reordering of the summation in (5) give a speedup factor of $\sim 25-30$ which is fast enough for real-time processing when using multiple threads. Next, changing memory layout in order to take advantage of SIMD instructions, cf. (6), yields another factor of $\sim 3-5$ to enable even singlethreaded real-time performance. Finally, Fig. 6 shows the optimized GPU timings which are around an order of magnitude faster than the fastest CPU timing. The results reported for OpenCL are optimized with respect to workgroup configuration and caching strategies using local memory within a workgroup. These optimizations make OpenCL perform slightly better than OpenGL on the AMD GPU.

Aggregated timings for *all* second stage processing steps are summarized in Fig. 7 for the best optimized multithreaded CPU and GPU implementations. The figure clearly shows that all timing results are far below the real-time limit. For the CPU, SIMD provides good overall speedup factors (~ 2.5 for SSE and ~ 4.2 for AVX), and, in accordance with the beamformation in isolation (Fig. 6), the total GPU results are again around an order of magnitude faster than the fastest CPU result.

IV. Conclusion

Several implementations of Synthetic Aperture Sequential Beamforming (SASB) targeting consumer level parallel processors such as multi-core CPUs and GPUs are presented. The fastest CPU and GPU implementations use 14% and 1.3% of the real-time budget of 62 ms/frame, respectively. These results demonstrate that SASB can be executed in-time for real-time ultrasound imaging, even with much lower hardware specifications than benchmarked, and with significant headroom for image post-processing.

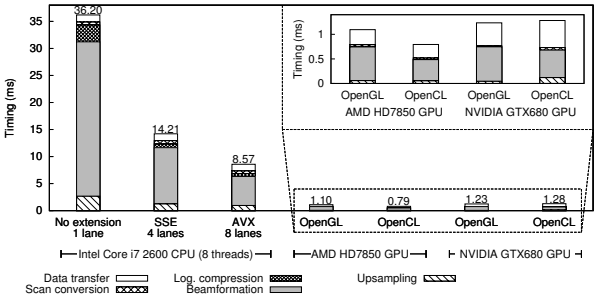

Fig. 7. Total benchmark timings on CPU and GPU for all second stage processing steps. The resolution of the final image was 512×512 pixels.

Acknowledgment

This work was supported by grant 82-2012-4 from the Danish National Advanced Technology Foundation and by BK Medical.

References

[1] K. E. Thomenius, "Evolution of ultrasound beamformers," in *Proc. IEEE Ultrason. Symp.*, vol. 2, 1996, pp. 1615–1621.

[2] B. D. Steinberg, "Digital beamforming in ultrasound," *IEEE Trans. Ultrason., Ferroelec., Freq. Contr.*, vol. 39, pp. 716–721, 1992.

[3] C. J. Thompson, S. Hahn, and M. Oskin, "Using modern graphics architectures for general-purpose computing: a framework and analysis," in *Proc of IEEE/ACM International Symposium on Microarchitecture*, 2002, pp. 306–317.

[4] B. Y. S. Yiu, I. K. H. Tsang, and A. C. H. Yu, "GPU-based beamformer: Fast realization of plane wave compounding and synthetic aperture imaging," *IEEE Trans. Ultrason., Ferroelec., Freq. Contr.*, vol. 58, no. 7, pp. 1698–1705, 2011.

[5] J. M. Hansen, D. Schaa, and J. A. Jensen, "Synthetic aperture beamformation using the gpu," in *Proc. IEEE Ultrason. Symp.*, 2011, pp. 373–376.

[6] M. C. Hemmsen, J. M. Hansen, and J. A. Jensen, "Synthetic Aperture Sequential Beamformation applied to medical imaging using a multi element convex array transducer," in *EUSAR*, Apr. 2012, pp. 34–37.

[7] M. Hemmsen, P. M. Hansen, T. Lange, J. M. Hansen, K. L. Hansen, M. B. Nielsen, and J. A. Jensen, "In vivo evaluation of synthetic aperture sequential beamforming," *Ultrasound Med. Biol.*, vol. 38, no. 4, pp. 708–716, 2012.

[8] M. C. Hemmsen, T. Kjeldsen, L. Lassen, C. Kjær, B. G. Tomov, J. Mosegaard, and J. A. Jensen, "Implementation of synthetic aperture imaging on a hand-held device," in *Proc. IEEE Ultrason. Symp.*, 2014, p. in press.

[9] J. Kortbek, J. A. Jensen, and K. L. Gammelmark, "Synthetic aperture sequential beamforming," in *Proc. IEEE Ultrason. Symp.*, 2008, pp. 966–969.

[10] ——, "Sequential beamforming for synthetic aperture imaging," *Ultrasonics*, vol. 53, no. 1, pp. 1–16, 2013.

[11] M. C. Hemmsen, S. I. Nikolov, M. M. Pedersen, M. J. Pihl, M. S. Enevoldsen, J. M. Hansen, and J. A. Jensen, "Implementation of a versatile research data acquisition system using a commercially available medical ultrasound scanner," *IEEE Trans. Ultrason., Ferroelec., Freq. Contr.*, vol. 59, no. 7, pp. 1487–1499, 2012.