An Optimized GPU Implementation of the MVDR Beamformer for Active Sonar Imaging

Jo Inge Buskenes, Jon Petter Åsen, Carl-Inge Colombo Nilsen, Andreas Austeng

***Abstract*—The Minimum Variance Distortionless Response (MVDR) adaptive beamformer has recently been proposed as an attractive alternative to conventional beamforming in active sonar imaging. Unfortunately, it is very computationally complex because a spatial covariance matrix must be estimated and inverted for each image pixel. This may discourage its use unnecessarily in sonar systems which are being pushed to ever higher imaging ranges and resolutions.**

In this study we show that for active sonar systems up to 32 channels, the computation time can be significantly reduced by performing arithmetic optimizations, and by implementing the MVDR on a Graphics Processing Unit (GPU). We also seek to point out important hardware limitations for these devices, and assess the design in terms of how efficiently it is able to use the GPU's resources. On a quad-core Intel Xeon system with a high-end Nvidia GPU, our GPU implementation renders more than 1 Mpx/s. This represents a speedup of more than two orders of magnitude over our initial implementation in C. This enables real-time processing of sonar data, and for making the MVDR a viable alternative to conventional methods in practical systems.

***Index Terms*—Adaptive beamforming, MVDR, Capon, sonar, active sonar, GPU, graphics processing unit, complexity.**

I. Introduction

DATA driven methods have been introduced in various imaging fields over the years in attempts to improve image quality. One such method is the Minimum Variance Distortionless Response (MVDR) beamformer. When compared to conventional methods, MVDR is often able to improve image contrast and resolution, as shown in e.g. radar [1], ultrasound [2], and active sonar [3]–[6].

Despite its inherent potential, the MVDR beamformer has yet to see widespread adoption in the active sonar field. There may be several reasons for this. For one, the method is not inherently robust, and may suffer from a phenomenon called signal cancellation in active systems [7]. Another reason is that in its original form, the computational complexity is cubic with the number of channels, O(M^3), while conventional beamformers are at O(M). This is because a spatial covariance matrix is estimated and inverted for each image pixel.

To ensure MVDR robustness, the litterature suggests combining means such as temporal and spatial averaging, and regularization [8], [9]. The complexity issue, on the other hand, can be handled by introducing well-founded approximations. For instance, some studies assume data stationarity which allows the formation of a Toeplitz-structured covariance matrix that is simpler to invert [10], [11]. Other studies perform MVDR beamforming in beamspace (the spatial frequency domain) which can be considered sparse due to the limited angular extent of the received beam [8], [12]. Exploiting this can lead to considerable performance improvements, especially in high channel count systems.

In this work, we show that such approximations may not be necessary since the computation time can be reduced by 1-2 orders of magnitude by implementing MVDR on a Graphics Processing Unit (GPU). Similar work may be found in e.g. medical ultrasound imaging, where Chen *et al.* have demonstrated a GPU implementation of the MVDR that operates on real valued data [13], [14]. While this implementation is fast, the real valued data constraint does not allow the creation of asymmetric and shifted responses, and hence prevents the MVDR's potential to be fully reached. Our implementation is not restricted in this manner. In related work, we investigated using the same MVDR GPU implementation in cardiac ultrasound imaging [15], [16], and in active sonar imaging we have compared the performance of this implementation to comparable adaptive methods [17].

This article is outlined as follows: In section II we introduce the concept of adaptive beamforming and provide details on the MVDR method. Then in III we investigate the complexity issue, discuss means for reducing arithmetic complexity, and detail an implementation that makes efficient use of the GPU's parallel resources. The final design is assessed in IV-V, where we provide benchmarks, comparisons with similar CPU implementations, and measures of how well the full capacity of the GPU was reached.

II. Methods

Consider a sonar imaging scenario where an encoded signal is transmitted to highlight a surface of interest, and assume that the backscattered wavefield is properly sampled using a uniform linear array of receivers. An image can then be formed by coherently combining the receiver outputs to focus at one angle at the time. The principle of adjusting the array's focus is commonly referred to as beamforming, and involves assigning suitable delays and weights to the array's channels.

A. Beamforming

Let the receiver be an M element uniform linear array, and assume that the signature of the transmitted signal has been removed by a matched filter (Fig. 1). Further assume that the array channels are digitally delayed to create focus at a pixel with azimuth angle θ and range sample n, such that

J. I. Buskenes, C.-I. C. Nilsen and A. Austeng are with the Department of Informatics, University of Oslo, Norway.

J. P. Åsen is with MI-Lab, Norwegian University of Science and Technology, Trondheim, Norway.

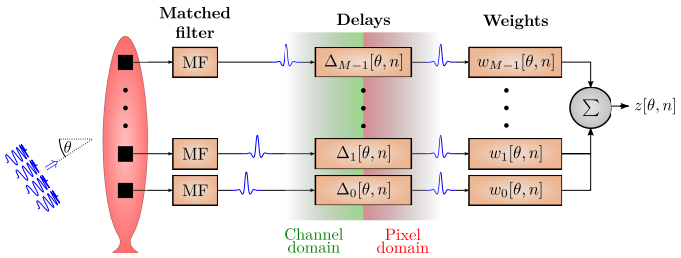

Fig. 1. Beamforming principle. Signal signature is first removed by matched filtering. Then - prior to summation - a suitable set of delays, Δ, and weights, w, are applied to focus on a pixel of interest at angle and range (θ, n).

the delayed data from the m'th channel can be expressed as $x_m[\theta, n]$. To simplify notation, the dependence on θ will be made implicit from now on.

By definition the beamformer output $z[n]$ can now be expressed as the weighted sum of all the delayed data samples:

$$z[n] = \boldsymbol{w}^H[n]\boldsymbol{x}[n] = \begin{bmatrix} w_0[n] \\ w_1[n] \\ \vdots \\ w_{M-1}[n] \end{bmatrix}^H \begin{bmatrix} x_0[n] \\ x_1[n] \\ \vdots \\ x_{M-1}[n] \end{bmatrix}, \quad (1)$$

where w_m is the weight factor assigned to channel m. With static weights this would be referred to as the conventional delay-and-sum (DAS) beamformer. A large variety of weighting functions exists here for trading lateral resolution for improved noise suppression (contrast), but one always ends up with a compromise between the two [18].

Various adaptive beamformers target this limitation by allowing the weights to change for each pixel to better fit the dynamic nature of the incoming wavefield. In other words, they attempt to use the *a priori* information present in the data to improve image quality. The MVDR beamformer is one such method. It finds the set of complex weights that minimizes the beamformer's expected output power, while ensuring unity gain in the look direction [19]. This is a convex optimization problem that can be solved using Lagrange multipliers to yield the solution:

$$\boldsymbol{w}[n] = \frac{\boldsymbol{R}^{-1}[n]\mathbf{1}}{\mathbf{1}^T\boldsymbol{R}^{-1}[n]\mathbf{1}}, \quad (2)$$

where $\mathbf{1}$ is a row vector of ones that represents broadside steering, and $\boldsymbol{R} = E\{\boldsymbol{x}[n]\boldsymbol{x}^H[n]\} \in \mathbb{C}^{M,M}$ is the spatial covariance matrix for the full array. Since $\boldsymbol{R}$ is unknown, we estimate it by computing a sample covariance matrix $\hat{\boldsymbol{R}}$. In this computation we will perform some degree of *spatial averaging* to avoid signal cancellation, *temporal averaging* to maintain true speckle statistics [20], and *diagonal loading* to improve robustness to parameter errors [21], [22]. Combined, these steps will also ensure a numerically well conditioned $\hat{\boldsymbol{R}}$(1).

1 should we mention variance reduction?

To form $\hat{\boldsymbol{R}}$ we will first compute an intermediate sample covariance matrix $\breve{\boldsymbol{R}}$, which is formed using temporal and spatial averaging. To do this we need to segment our array into subarrays. If we let $x_l[n]$ represent the data vector from

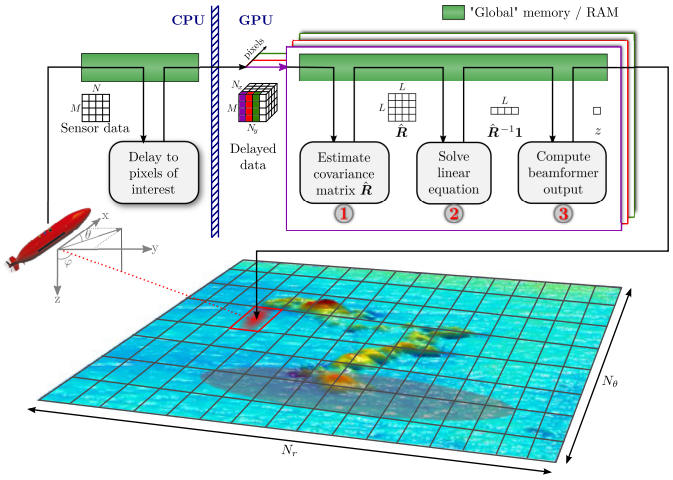

Fig. 2. MVDR beamforming. For each pixel in range and azimuth,
1. an $L \times L$ sample covariance matrix $\hat{\boldsymbol{R}}$ is computed,
2. the term $\hat{\boldsymbol{R}}^{-1}\mathbf{1}$ is found using a linear equation solver,
3. and the beamformer output is computed from z from (6), where $\boldsymbol{w}$ is found by substituting $\hat{\boldsymbol{R}}^{-1}\mathbf{1}$ into (2).

subarray l,

$$\boldsymbol{x}_l[n] = \begin{bmatrix} x_l[n] & x_{l+1}[n] & \dots & x_{l+L-1}[n] \end{bmatrix}^T, \quad (3)$$

then $\breve{\boldsymbol{R}}$ can be calculated as:

$$\breve{\boldsymbol{R}}[n] = \frac{1}{N_K N_L} \sum_{l=0}^{M-L} \sum_{n'=n-K}^{n+K} \boldsymbol{x}_l[n]\boldsymbol{x}_l^H[n] \in \mathbb{C}^{L,L}, \quad (4)$$

where $N_K = 2K + 1$ is the number of temporal samples to perform averaging over, and $N_L = M - L + 1$ is the number of subarrays.

The final estimate $\hat{\boldsymbol{R}}$ is found by adding a fraction d of the total power of $\breve{\boldsymbol{R}}[n]$ to its diagonal [2]:

$$\hat{\boldsymbol{R}}[n] = \breve{\boldsymbol{R}}[n] + \boldsymbol{I}\frac{d}{L}\mathrm{tr}\{\breve{\boldsymbol{R}}[n]\}, \quad (5)$$

where $\boldsymbol{I}$ is an identity matrix, $\mathrm{tr}\{\cdot\}$ represents the matrix trace operation, and $\mathrm{tr}\{\breve{\boldsymbol{R}}[n]\}$ is an estimate of the energy received from this pixel.

Note how subarray averaging led to a size reduction of $\hat{\boldsymbol{R}}$ from $\mathbb{C}^{M,M}$ to $\mathbb{C}^{L,L}$, and hence will produce an L-element weight set when substituted into (1). This weight set is applied to all the subarrays, prior to computing the beamformer output as in (1). Or, equivalently, we may apply the weight set to the sum of all the subarrays:

$$z[n] = \boldsymbol{w}^H[n] \sum_{l=0}^{M-L} \boldsymbol{x}_l[n]. \quad (6)$$

As summarized in Fig. 2, the MVDR method is applied to each pixel independently, by

1) computing the sample covariance matrix $\hat{\boldsymbol{R}}$ in (5),
2) computing $\hat{\boldsymbol{R}}^{-1}\mathbf{1}$ in (2), and
3) computing the beamformer output z in (6).

Next we will evaluate these steps in terms of arithmetic complexity, and then discuss their mappability to parallel hardware.

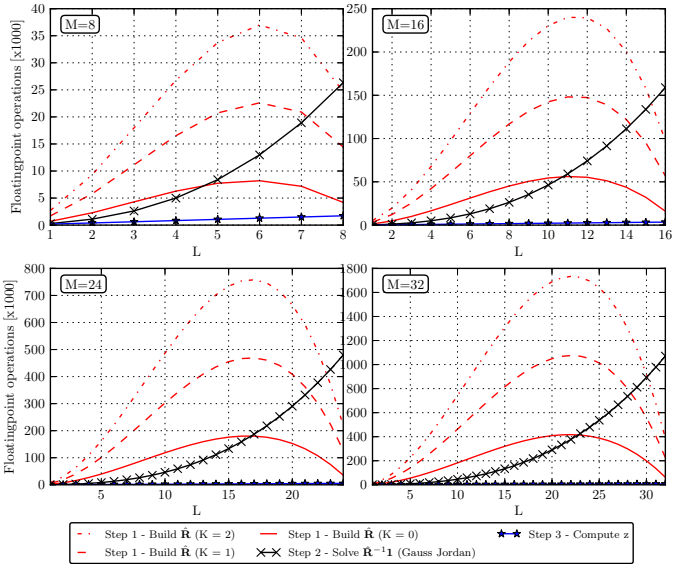

Fig. 3. Per-pixel computational complexity of the steps in MVDR beamforming (prior to any optimizations). To avoid signal cancellation in an active sonar system we usually set $L < \frac{M}{2}$, in which region the computation of $\hat{\boldsymbol{R}}$ dominates in terms of arithmetic complexity, especially when performing temporal averaging.

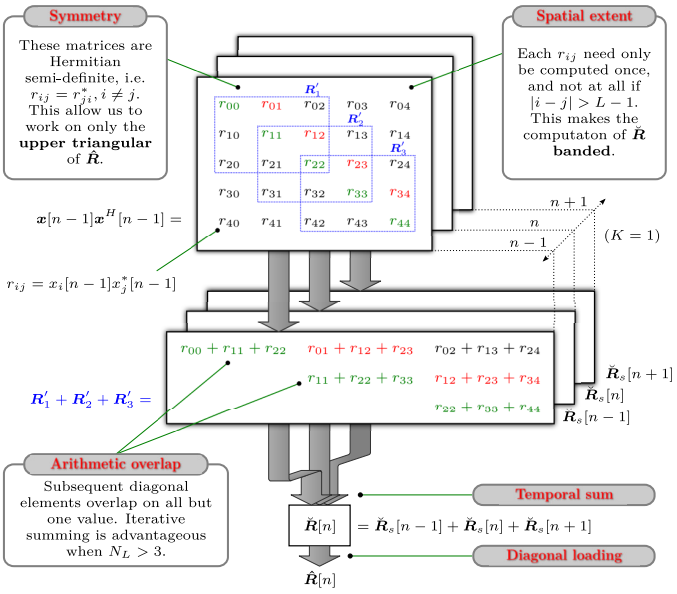

Fig. 4. Step 1: Building $\hat{\boldsymbol{R}}$. This is a visualization of how $\hat{\boldsymbol{R}}$ could be built in a case with $M = 5$ sensors, with subarray size $L = 3$ and temporal averaging set to $K = 1$. Here $\boldsymbol{R}_l'$ is the sample covariance matrix for the l'th subarray, and $\breve{\boldsymbol{R}}$ is the average of N_K of these. Note that instead of performing the temporal sum last as here, one could take more temporal samples into consideration in the computation of each r_{ij}.

B. Computational Complexity

If we neglect spatial and temporal averaging, then the computation of $\hat{\boldsymbol{R}}$ is reduced to a single outer product with complexity of O(M^2), and the inversion is then of O(M^3). This might lead us to believe that the inversion step is by far the most complex. But if we implement spatial and temporal averaging as in (4), then computing $\hat{\boldsymbol{R}}$ is of O($N_K N_L L^2$) and the inversion of O(L^3). Computing $\hat{\boldsymbol{R}}$ is now the most complex operation whenever $N_L N_K > L$.

To visualize these relations, and include the effects of using complex numbers, we set up complexity formulas that accounts for the total number of arithmetic operations for each step in the MVDR process. We only excluded diagonal loading performed in the computation of $\hat{\boldsymbol{R}}$, and partial pivoting used in the inversion step, since these operations contribute marginally to the end result. The entire range of possible subarray sizes from $L \in [1, M]$ was finally evaluated, with temporal averaging set to $K \in \{0, 1, 2\}$, and the number of channels set to $M \in \{8, 16, 32\}$. The results are shown in Fig. 3. We regarded the formulas themselves as were omitted for the sake of clarity, and We omitted the formulas long and not are omitted for the sake of clarity, since they

We note how the computation of $\hat{\boldsymbol{R}}$ completely dominates at smaller subarray sizes, that solving $\hat{\boldsymbol{R}}^{-1}\mathbf{1}$ only plays a notable role for larger array and subarray sizes, and that the computation of z has a negligible impact on computation time. Also notice how temporal averaging comes with a high computational penalty. This is because building $\hat{\boldsymbol{R}}$ is heavy on complex multiplications, which require 3 times as many arithmetic instructions as a complex addition, and a lot of these are repeated unnecessarily.

III. Mapping the MVDR to a GPU

An important feature of MVDR beamforming, as with beamforming in general, is that each pixel can be computed independently and in an identical fashion. Furthermore, a typical sonar image may contain millions of pixels. This represents a level of data parallelism that appears very well suited for a massively parallel architecture such as the GPU.

We decided to investigate the feasibility of running MVDR on a GPU by mapping it to a Nvidia GeForce Quadro 6000. This is a high-end Compute Unified Device Architecture (CUDA) enabled GPU based on Nvidia's Fermi architecture. The code was written in Nvidia's "C for CUDA" framework. We believe that GPUs from AMD, or the cross-platform OpenCL framework from the Khronos group, are equally attractive alternatives, but comparing them is beyond the scope of this study.

To use Nvidia's own terminology, the Quadro 6000 is comprised of 14 streaming multiprocessors (SMs), each having 32 CUDA cores that execute a common program called a kernel. Combined these cores deliver a peak performance of more than 1 Tflop/s (appendix A). In practice this performance is hard to obtain, but one can get fairly close by balancing the load evenly on all cores, and by trying to avoid that some cores are forced to idle due to a pending data transfer or thread synchronization. As the steps in the MVDR method require different strategies for achieving this, we have designed a different kernel and configuration for each of them. We will pay particular attention to building $\hat{\boldsymbol{R}}$, since the potential for gaining overall speedups are greatest here.

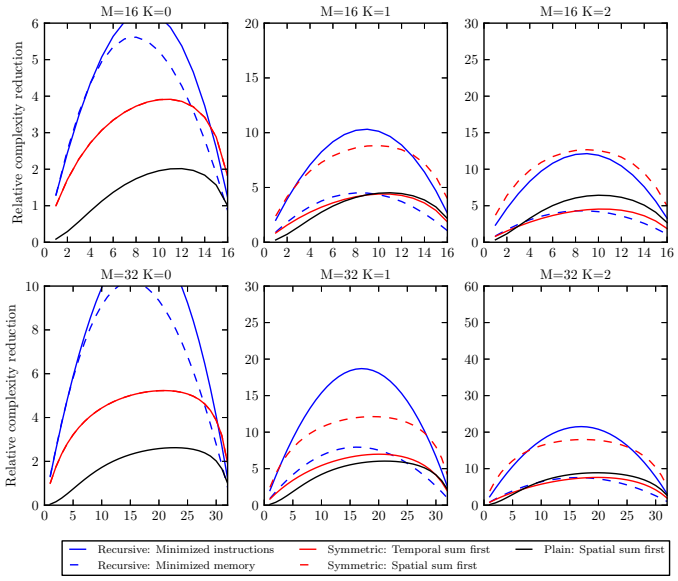

Fig. 5. Arithmetic optimization of computing $\hat{\boldsymbol{R}}$: Relative reduction in arithmetic complexity (higher is better). All plots are normalized to the complexity curve of the reference implementation in Fig. 3.

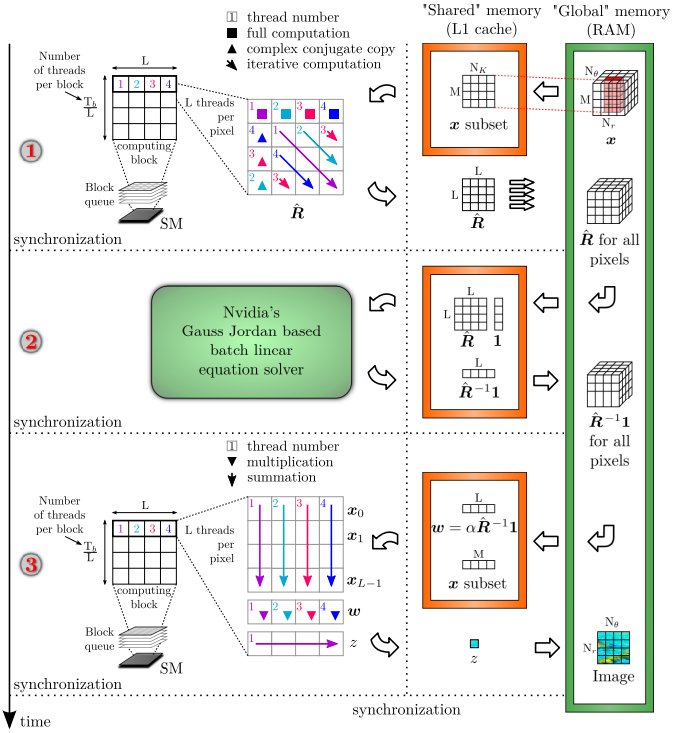

Fig. 6. MVDR implemented on a GPU. We do this in 3 steps, where each step process the full image before moving on to the next step. *Step 1*: The sample covariance matrix $\hat{\boldsymbol{R}}$ is formed by threads running along its diagonals. This allows spatial averaging to be implemented in a computationally efficient manner and minimizes inter-thread communication. *Step 2*: $\hat{\boldsymbol{R}}^{-1}$ is computed using the heavily optimized batched linear equation solver from Nvidia. *Step 3*: The beamformer output z is each image pixel are computed .

A. Computing the Spatial Covariance Matrix, $\hat{\boldsymbol{R}}$

As observed in Fig. 3, a direct computation of $\hat{\boldsymbol{R}}$ by implementing (4) and (5) is the greatest computational burden. We target this issue by first performing *arithmetic reductions*, then aim to get as close to the peak *arithmetic throughput* on the GPU as possible.

1) Arithmetic Reductions: In Fig. 4 we illustrate how to reduce the arithmetic complexity of building $\hat{\boldsymbol{R}}$ in a system with $M = 5$ sensors, a subarray size of $L = 3$ and temporal averaging is set to $K = 1$. Here the spatial sum is carried out before the temporal sum. To reduce arithmetic operations we may

1) exploit the fact that $\hat{\boldsymbol{R}}$ is Hermitian positive semi-definite to compute only one half of it,
2) avoid redundant multiplications, and
3) implement the spatial sum iteratively by sliding along the diagonals of $\hat{\boldsymbol{R}}$.

To study the effect of these optimizations we altered the complexity formulas to take them into account. Exploiting symmetry and performing iterative summing (step 1 and 3) is always desirable, but avoiding redundant multiplications (step 2) comes at the cost of increased memory consumption. The effect of adding all the optimizations, or all but saving multiplications, is shown in Fig. 5. Here the complexity curves are relative to the reference implementation in Fig. 3. Both solutions reduce the complexity considerably, and recomputing the multiplications does not make that much of a difference. We selected the memory minimized version since our algorithm is severely memory bound, as we will see next.

2) Arithmetic Throughput: When we compute $\hat{\boldsymbol{R}}$, we very rarely perform more than 1-3 floating point operations for every float read or written to memory. Unfortunately, the GPU prefers kernels to be more computationally intensive than this. This can be inferred from Table I, where the peak bandwidth of the three types of GPU memory are compared to the peak arithmetic throughput (see appendix A for derivations). The global memory(2), in which all data must reside at some point, is at best only able to move one float for every 30 floats processed(3) by the CUDA cores. This is why global memory usage should be avoided. Shared memory, however, is peak shared memory performance is only 4 times slower than the arithmetical throughput so we really need to use it. To achieve this we must

1) keep all the data relevant to the pixels being computed in shared memory, while
2) trying to use it as efficiently as possible.

These challenges are very closely linked. Since the Quadro 6000 architecture is of compute capability 2.0, it has 48 kB of shared memory (L1 cache) and 128 kB of registers per SM [24]. This memory is shared by all active threads on that

2 if there's space somewhere, introduce memory layout

3 well, not entirely true. rephrase

	B_{arith}	B_{mem}	B_{mem}/B_{arith}
Arithmetic	1.03 Tflop/s		
Global memory		36 Gfloats/s	1:30
Shared memory		257 Gfloats/s	1:4
Registers		>1.5 Tfloats/s	>3:2 [23]

TABLE I
NVIDIA QUADRO 6000: MEMORY THROUGHPUT, B_{MEM}, COMPARED TO ARITHMETIC THROUGHPUT, B_{ARITH}.

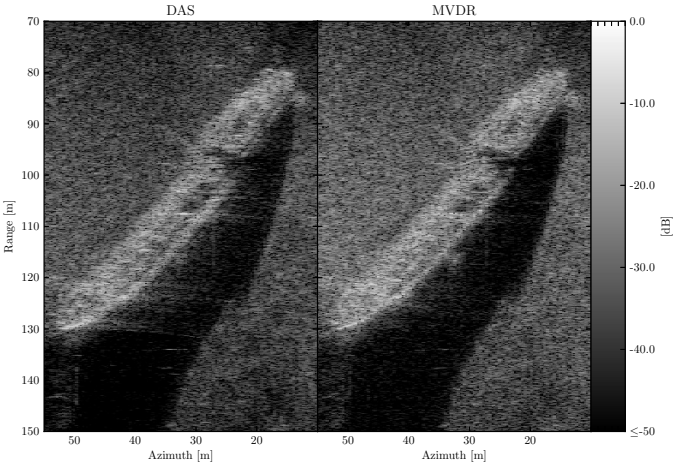

Fig. 7. HISAS sidescan sonar (SSS) image of the shipwreck Holmengraa. Imaging parameters was $M = 32$, $L = 16$, $K = 1$ and $d = 1\%$. Prior to display the image was linearly upinterpolated by a factor 2 in azimuth making its total size 1.46 Mpx. Note how MVDR improves edge definition and reduces noise in shadow regions.

SM, a number that should be no less than 768 [25]. This is to expose a sufficient level of data parallelism to ensure that memory latency is completely hidden (Little's law). By dividing the shared memory evenly on all 768 threads we find that each thread should store no more than 8 single precision complex numbers in shared memory(4), and 24 stored in registers. This should make it apparent why computing a single pixel per thread is a bad idea, and why we need to keep each thread as light on memory consumption as possible.

In Fig. 6 we illustrate how we got around these challenges. First we make each SM compute entire range lines of pixels, then load the subset of $\boldsymbol{x}$ that all these pixels depend upon into shared memory. This lets us perform temporal averaging without the need for reading additional channel data.

Second we assign L threads per pixel to traverse the diagonals of $\hat{\boldsymbol{R}}$. On the top row a full computation is carried out, then that entire row is saved back to global memory following a collective access pattern that maximizes global memory bandwidth. For subsequent rows the threads move along the diagonals while performing iterative summations; the result from the previous element on the diagonal is updated by adding and removing the correlation coefficients that enters and exits the sum, respectively. To minimize memory consumption, we compute the coefficients again every time we need them. When a thread has finished up a diagonal, we have them wrap around to compute one of the diagonals in the lower triangular of $\hat{\boldsymbol{R}}$. Since $\hat{\boldsymbol{R}}$ is conjugate symmetric, the values in the leftmost column is obtained by a complex conjugate copy of the relevant value in the first row. Combined these steps balance the load evenly on all threads, is almost completely free of arithmetic redundancy, and consumes less memory.

[26](5)

B. Solving $\hat{\boldsymbol{R}}^{-1}\mathbf{1}$

As intuition might suggest that the matrix product $\hat{\boldsymbol{R}}^{-1}\mathbf{1}$ can be carried out by first inverting the matrix $\hat{\boldsymbol{R}}$, then performing the inner product with $\mathbf{1}$. However, one can also solve the linear equation $\hat{\boldsymbol{R}}\boldsymbol{b} = \mathbf{1}$ for $\boldsymbol{b}$, where $\boldsymbol{b} = \hat{\boldsymbol{R}}^{-1}\mathbf{1}$. In terms of arithmetic count, and when comparing direct implementations, the latter appears to be the preferred option.

An important thing to note, however, is that unlike the problems most libraries tries to solve we do not attempt so solve large linear equations or invert large matrices; our matrices are small, but we have a very large number of them. Fortunately Nvidia recently released a highly optimized batched linear equation solver aimed for this particular purpose. It is a Gauss Jordan based and supports partial pivoting and complex numbers. The only downside to using it in our application is that it does not exploit the Hermitian property of our sample covariance matrix. A better choice would be a Cholesky solver, which is designed for Hermitian positive semi-definite matrices and is expected to offer roughly a factor 2 speedup over the Gauss Jordan one.(6)

C. Computing z

The beamformer output z is computed on a per-pixel basis by first computing the MVDR weights $\boldsymbol{w}$, which are a mere scaled version of $\hat{\boldsymbol{R}}^{-1}\mathbf{1}$ (2). Then the sum in (6) is found by assigning a group of L threads to respective elements of x_l, which then proceed to accumulate these for all N_L subarrays. The resulting data vector is finally multiplied with the weight vector using the same threads, and then a single thread is used to sum these products to obtain z(7).

IV. Images and Benchmarks

To demonstrate the imaging capability of the MVDR beamformer, we have processed experimental datasets from the M=32 element Kongsberg Maritime HISAS1030 sonar mounted on a HUGIN AUV [27]. HISAS1030 is a high resolution synthetic aperture sonar with an array length of 1.2 m and operating frequency of 100 kHz. To produce the image shown in Fig. 7 it was operated in sidescan mode. The studied object is the 1500 dwt oil tanker wreck Holmengraa. It is 68 m long and 9 m wide, and lies at a slanted seabed at 77 m depth outside of Horten, Norway. The MVDR image were processed with parameters $L = 16$, $K = 3$, and $d = 1\%$.

The computational performance of our implementation was first assessed on a test system with a quad-core Intel Xeon E5620, 64 GB of RAM and an Nvidia Quadro 6000 card. The results were obtained by processing a 1 Mpx image with a 32 channel array, for all subarray sizes L, and for $K \in \{0, 1, 2\}$. Normal run time measurements are shown in Fig. 8, the run times of memory-only and arithmetic-only GPU kernels are depicted in Fig. 9, and an estimate of computation efficiency is presented in Fig. **??**. We also took the code for a test drive at the Boston HPC centre on a machine with an Intel Xeon E5-2670, 32 GB RAM and an Nvidia K20mm. The results from this run is shown in Fig. 11.

As seen in the run time comparisons are presented in Fig. 8, the GPU method outperforms the CPU implementations by

4 introduce memory structure -shared for data shared between threads

5 Find a good spot for these

6 somewhere a short remark regarding inversion/solver alternatives should be made

7 Rakende lik-egyldig hva vi gjr her.. Kan kuttes.

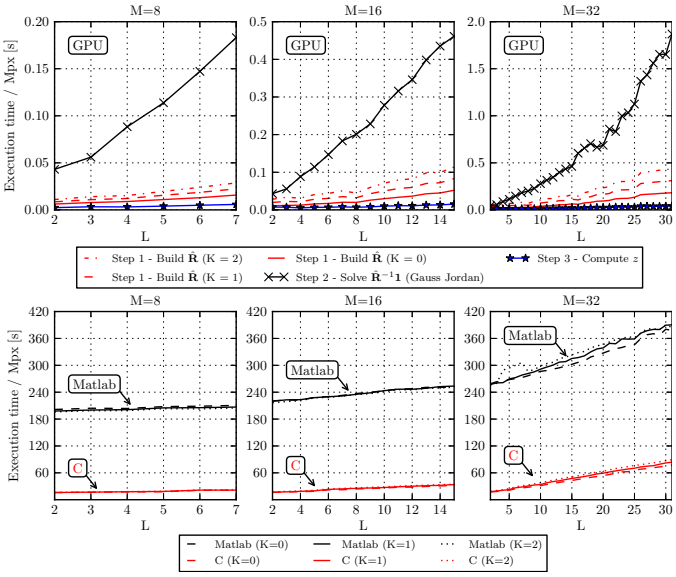

Fig. 8. MVDR benchmarks. A 1 Mpx image from a $M = 32$ channel array was processed for all L, and for $K \in \{0, 1, 2\}$.
Top: The time the GPU spent on building $\hat{\boldsymbol{R}}$, solving $\hat{\boldsymbol{R}}^{-1}\boldsymbol{a}$, and computing z. Note the major speedup of building $\hat{\boldsymbol{R}}$ when compared to the complexity plot in Fig. 3.
Bottom: Compared to a reference C and Matlab implementation running on a CPU the GPU offered was 2-3 orders of magnitude slower than the GPU implementation.

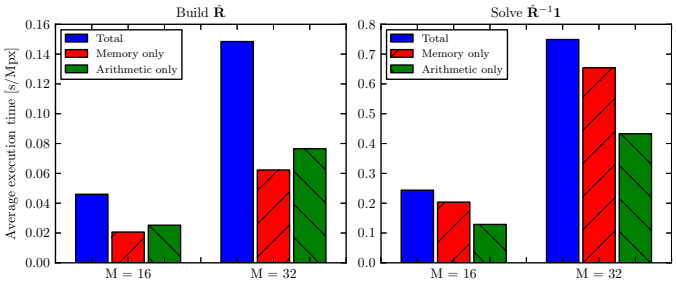

Fig. 9. Execution time of an arithmetic-only and a memory-only version of the MVDR code. A 1 Mpx dataset from an $M = 32$ array was processed for all L using $K = 1$, and only the mean execution time was used here. The kernel building $\hat{\boldsymbol{R}}$ can here seen to be memory bound, as the time the kernels spends performing memory transactions are higher than the corresponding time it spends carrying out arithmetical operations. Furthermore, the gap between the total run time and the restricted kernels can usually be attributed to latency.

2-3 orders of magnitude in terms of processing speed. The bottleneck in the final design is now the inversion step, which is typically 5 times slower than the build step. In most cases the processing speed of the Quadro 6000 is above 1 Mpx/s, and this was only improved by 150-200% when run on the new K20m GPU. The memory-only and arithmetic-only kernels show that the kernels are roughly bound by both both, so optimising only one of these further will have marginal effect. All kernels were compiled with gcc at optimization level `O2`. Excluded from these benchmarks is the data transfer time from CPU to GPU, which account for 2-20% of the total processing time. This is to limit the scope of the article, but will need

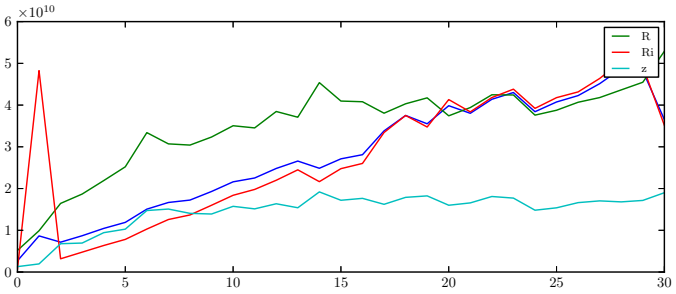

Fig. 10. Code efficiency. An estimate of the number of floating point operations per second (Flop/s), found by dividing the theoretical complexity curves by actual run times. This is a cruel measure as it does not include any other instructions than the actual arithmetic operations in the MVDR computation.

to be taken into consideration in an implementation of a full sonar system.

V. Discussion

In accordance to previous studies, Fig. 7 demonstrates the MVDR beamformer's ability to produce images with suppressed interference and improved detail resolution. Compared the DAS beamformer, the ship's edges appear sharper and the shadows less noisy. In this scenario the MVDR's performance was not particularly sensitive to parameter adjustments. Similar performance was obtained with arbitrary combinations of $L \in \{12, 16, 24\}$, $K \in \{1, 2, 3\}$ and $d \in [0.01, 0.05]$, and only minor or no adjustments had to be made when processing another part of the scene.

As observed in Fig. 8 the combined effect of performing arithmetic reductions and implementing the MVDR on a GPUs significantly improved the processing speed. Compared to a fairly non-optimized C and Matlab implementation a speedup of 2-3 orders of magnitude was achieved, but we do not consider this comparison fair. In fact, the theoretical peak throughput of this GPU is roughly 20 times higher than that the CPU in question, meaning the potential of the CPUs was far from reached in our initial implementations. This is mainly because the entire GPU design is focused on efficient use of fast memory cache, while no similar optimisation work was carried out on the CPU implementations.

Note from Fig. 8 is how the building of $\hat{\boldsymbol{R}}$ now only takes up a fraction of the total processing time, when it was the other way around when we presented the theoretical complexity curves in Fig. 3. Also, the benchmark curves for building $\hat{\boldsymbol{R}}$ now seem linear. The main reason for this is that the optimization step negated much of the extra complexity introduced by averaging, i.e. reduced the complexity from $\mathrm{O}(N_K N_L L^2)$ to less than $\mathrm{O}(L^2)$. Another reason for the run time to appear more like $\mathrm{O}(L)$ is likely the GPU, since our design makes better use of the GPU's resources when there is more processing involved per pixel. The inversion step, on the other hand, gains less from being implemented on a GPU. This is because its nature is less data parallel, and the back substitution involved in its computation is more of a sequential nature. This difference can observed in Fig. 3. Alternative

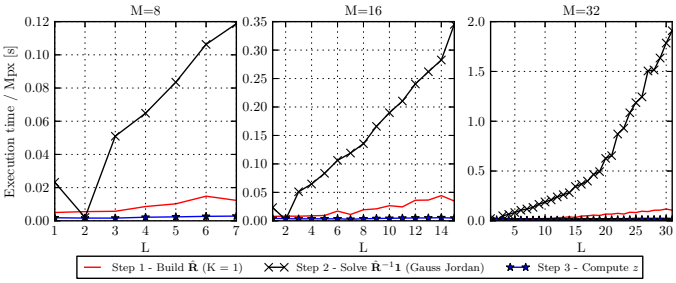

Fig. 11. MVDR benchmarks from Boston HPC centre with the new high-end Nvidia K20m Kepler GPU. The exact same scenario and code as in Fig. 8 was used here. With no code alterations the performance was only improved marginally compared to running the code on the Quadro 6000.

solvers, such as one based on Cholesky decomposition that exploits the Hermitian property of $\hat{\boldsymbol{R}}$, can in theory reduce complexity by a factor 2, but we question whether this result can be obtained in practice using a GPU. The CPU implementations perform Cholesky based inversion, but this does not explain why the C and Matlab implementation have a total run time that is linearly dependent on L. We find this observation odd, but are confident that it is accurate.

It is often often hard to pinpoint whether a GPU design is limited by memory bandwidth or arithmetical throughput. In Fig. 9 we made an estimate of this by creating two versions of the kernel, one that only performs the arithmetical operations, and one that only performs memory operations. Although these are fairly balanced, we have chosen throughout to minimize memory consumption at the expense of some extra computations. Latency only plays a minor role, which can be inferred from the relatively narrow gap between the total run time bar and the bar that the algorithm is bound by.

Even with all our efforts we were only able to obtain processing rates of 20-30 MFlop/s on average in the desirable range of subarray size (Fig. 3). This is mainly due to the inversion step, since we typically get 60-80 MFlop/s when building $\hat{\boldsymbol{R}}$. While these numbers are likely slightly underestimated, they are still far away from the peak of 1 Flop/s. Again, we believe this is due to memory bandwidth. This belief is further supported by the test results from running the code on the new K20m Kepler card. The K20m has a theoretical peak of 3.5-4 Tflop/s, but we experienced a modest 150-200% performance boost from our run.

VI. Conclusion

The MVDR beamformer is an algorithm capable of producing images with improved detail resolution and contrast compared to conventional DAS beamforming. The downside is its inherent need for robustification and the high computational complexity associated with estimating and inverting the spatial covariance matrix.

We have shown that for systems containing up to 32 channels the problem can be largely mitigated by building the covariance matrix in a clever way, and by making use of the massive computational power inherent in modern GPUs. We were able to improve upon the run time of a moderately optimized C-implementation by roughly 2 orders of magnitude. This performance is sufficient for computing properly sampled full-coverage sectorsscan images from the HISAS1030 sonar in real-time.

The MVDR maps well to the GPU since the computations involved are independent on a pixel level, and partially also within each pixel. However, we found it challenging to adapt the MVDR implementation to make use of the limited amount of cache available to each GPU computing core, which is a prerequisite for obtaining peak performance.

Appendix

MVDR Complexity Formulas

To estimate the number of floating point operations needed to MVDR beamform a single pixel, we formed expressions that accumulate the number of complex arithmetic operations found in the MVDR process. From observing the generated assembly code we inferred that each complex addition and multiplication would require $O_\mathrm{a} = 2$ and $O_\mathrm{m} = 6$ floating point operations, respectively. The formulas are listed for each step below for reference.

Building $\hat{\boldsymbol{R}}$. The initial complexity of this step can be inferred directly from (4) and (5):

$$O_{\underset{\text{initial}}{\text{Build }\hat{\boldsymbol{R}}}} = \underbrace{O_\mathrm{m} N_\mathrm{k} N_\mathrm{l} L^2}_{\text{Multiplications}} + O_\mathrm{a}(N_\mathrm{k} + N_\mathrm{l} - 2)L^2 + O_\mathrm{dload}, \quad (7)$$

where $O_\mathrm{dload} = (2L-1)O_\mathrm{a} + O_\mathrm{m}$ is the minor cost of performing diagonal loading. If we apply the optimization strategies discussed in section III-A we can arrive at the following instead:

$$\begin{aligned} O_{\underset{\text{min arith}}{\text{Build }\hat{\boldsymbol{R}}}} = {} & \underbrace{O_\mathrm{m}\frac{M+N_\mathrm{l}}{2}L}_{\text{Multiplications}} + \underbrace{O_\mathrm{a}(N_\mathrm{k}-1)(N_\mathrm{l}-1)L}_{\text{First row additions}} \\ & + \underbrace{\frac{(L-1)(L-2)}{2}\left[2O_\mathrm{a} + 2(N_\mathrm{k}-1)O_\mathrm{a}\right]}_{\text{Iteration additions}} \\ & + O_\mathrm{dload} \end{aligned} \quad (8)$$

Of the solutions discussed, this is the least expensive in terms of arithmetic instructions. However, if memory bandwidth is a limiting factor a better solution is to recompute multiplications where they are needed:

$$\begin{aligned} O_{\underset{\text{min mem}}{\text{Build }\hat{\boldsymbol{R}}}} = {} & \underbrace{O_\mathrm{m} N_\mathrm{k} N_\mathrm{l} L}_{\text{First row multiplications}} + \underbrace{O_a(N_\mathrm{k}-1)(N_\mathrm{l}-1)L}_{\text{First row additions}} \\ & + \underbrace{\frac{(L-1)(L-2)}{2}\left[2O_\mathrm{a} + 2(N_\mathrm{k}-1)O_\mathrm{a} + 2N_\mathrm{k}O_\mathrm{m}\right]}_{\text{Iteration multiplications and additions}} \\ & + O_\mathrm{dload} \end{aligned} \quad (9)$$

Solving $\hat{\boldsymbol{R}}^{-1}\boldsymbol{a}$ is achieved by using a batched Gauss Jordan solver with support for complex numbers and partial pivoting. It complexity - with partial pivoting excluded - can

be expressed as:

$$O_{\text{Solve } \hat{\boldsymbol{R}}^{-1}\boldsymbol{a}} = \sum_{r=0}^{L} \Bigg[\underbrace{\big(L-r\big)\big((L-r+2)O_{\text{a}} + (L-r+3)O_{\text{m}}\big)}_{\text{Reduction}} + \underbrace{(r-1)O_{\text{a}} + rO_{\text{m}}}_{\text{Backsubstitution}} \Bigg], \tag{10}$$

where r is a running variable r that indexes rows in [TODO result matrix] $\big[\hat{\boldsymbol{R}}|\boldsymbol{a}\big]$.

Computing z is very simple once the covariance matrix $\hat{\boldsymbol{R}}$ is built and inverted, and has an near negligible impact on performance:

$$O_{\text{Compute } z} = (2L-2)2 + O_{\text{m}}3L \tag{11}$$

GPU THROUGHPUT

In the context of determining whether an implementation is computationally bound or memory bound, one should first compare the target platform's sustained computational throughput to sustained memory throughput. Let us start with the former.

The Quadro 6000 has 32 CUDA cores per SM, each operating at a rate of 1148 MHz and being able to perform 2 floating point operations (flop) per clock cycle when multiply-add instructions are used. Theoretical peak arithmetic throughput is then given as:

$$\begin{aligned} \text{B}_{\text{arith}} &= 2 \cdot 1148 \text{ flop/s/core} \cdot 32 \text{ cores/SM} \cdot 14 \text{ SMs} \\ &= 1.03 \text{ Tflop/s}. \end{aligned} \tag{12}$$

Now let us compare this to the memory throughput. The "global" GDDR5 memory bus is 384 bit wide, and operates at 3 Ghz where 2 bits are sent every cycle. Its peak bandwidth is then:

$$\begin{aligned} \text{B}_{\text{gmem}} &= \frac{2 \cdot 3 \text{ Gbit/s} \cdot 384 \text{ bit}}{8 \text{ bit/byte}} \\ &= 144 \text{ GB/s } (36 \text{ Gfloats/s}). \end{aligned} \tag{13}$$

The shared memory, on the other hand, is organized into 32 banks per SM, each being 32 bit wide and operating at 1148 MHz where 1 bit is sent per cycle. Its peak aggregated bandwidth is then:

$$\begin{aligned} \text{B}_{\text{smem}} &= \frac{\frac{1148}{2} \text{ Mbit/s} \cdot 32 \text{ bit/bank} \cdot 32 \text{ banks/SM} \cdot 14 \text{ SMs}}{8 \text{ bit/byte}} \\ &\approx 1.03 \text{ TB/s } (257 \text{ Gfloats/s}). \end{aligned} \tag{14}$$

The bandwidths are compared in Tab. I. Note that even when using shared memory at least 4 floating point operations must be carried out per float transferred to the CUDA cores, otherwise the algorithm will be memory bound and the peak arithmetic throughput can not be reached.

ACKNOWLEDGMENT

The authors would like to thank Kongsberg Maritime and the Norwegian Defence Research Establishment (FFI) for providing experimental data, and thank Nvidia for providing support for running the batched linear equation solver as well as for granting us a testdrive at the Boston HPC center.

REFERENCES

[1] G. R. Benitz, "High-Definition Vector Imaging," *Lincoln Laboratory Journal*, vol. 10, no. 2, pp. 147–170, 1997. [Online]. Available: http://www.ll.mit.edu/publications/journal/journalarchives10-2.html

[2] J.-F. Synnevag, A. Austeng, and S. Holm, "Adaptive beamforming applied to medical ultrasound imaging." *IEEE transactions on ultrasonics, ferroelectrics, and frequency control*, vol. 54, no. 8, pp. 1606–13, Aug. 2007. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/19811995

[3] A. E. A. Blomberg, A. Austeng, and R. E. Hansen, "Adaptive Beamforming Applied to a Cylindrical Sonar Array Using an Interpolated Array Transformation," *IEEE Journal of Oceanic Engineering*, vol. 37, no. 1, pp. 25–34, Jan. 2012. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6112183

[4] A. E. A. Blomberg, R. E. Hansen, S. A. V. Synnes, and A. Austeng, "Improved interferometric sonar performance in shallow water using adaptive beamforming," in *Proceedings of the International Conference & Exhibition on Underwater Acoustic Measurements (UAM), Kos, Greece, June*, 2011.

[5] S. Dursun, A. Austeng, R. E. Hansen, and S. Holm, "Minimum variance beamforming in active sonar imaging," in *Proceedings of the 3rd International Conference & Exhibition on Underwater Acoustic Measurements: Technologies and Results*, B. r. e. John S. Papadakis & Leif, Ed., 2009, pp. 1373–1378.

[6] K. Lo, "Adaptive Array Processing for Wide-Band Active Sonars," *IEEE Journal of Oceanic Engineering*, vol. 29, no. 3, pp. 837–846, Jul. 2004. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1353435

[7] B. Widrow, K. Duvall, R. Gooch, and W. Newman, "Signal cancellation phenomena in adaptive antennas: Causes and cures," *IEEE Transactions on Antennas and Propagation*, vol. 30, no. 3, pp. 469–478, May 1982. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1142804

[8] H. L. Van Trees, *Optimum Array Processing*. New York, USA: John Wiley & Sons, Inc., Mar. 2002. [Online]. Available: http://doi.wiley.com/10.1002/0471221104

[9] T. Kailath, "Adaptive beamforming for coherent signals and interference," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 33, no. 3, pp. 527–536, Jun. 1985. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1164583

[10] B. M. Asl and A. Mahloojifar, "A low-complexity adaptive beamformer for ultrasound imaging using structured covariance matrix." *IEEE transactions on ultrasonics, ferroelectrics, and frequency control*, vol. 59, no. 4, pp. 660–7, Apr. 2012. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/22547277

[11] A. Jakobsson, S. Marple, and P. Stoica, "Computationally efficient two-dimensional Capon spectrum analysis," *IEEE Transactions on Signal Processing*, vol. 48, no. 9, pp. 2651–2661, 2000. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=863072

[12] C.-I. C. Nilsen and I. Hafizovic, "Digital beamforming using a GPU," in *2009 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, Apr. 2009, pp. 609–612. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4959657

[13] J. Chen, Y. Yiu, and H. So, "Real-time GPU-based adaptive beamformer for high quality ultrasound imaging," *IEEE Ultrasonics Symposium*, vol. 1, no. 1, pp. 1–4, 2011. [Online]. Available: http://hub.hku.hk/handle/10722/140228

[14] J. Chen, B. Y. Yiu, B. K. Hamilton, A. C. Yu, and H. K.-H. So, "Design space exploration of adaptive beamforming acceleration for bedside and portable medical ultrasound imaging," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 4, p. 20, Dec. 2011. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2082156.2082162

[15] J. P. Å sen, J. I. Buskenes, C.-I. C. Nilsen, A. Austeng, and S. Holm, "Implementing Capon Beamforming on the GPU for Real Time Cardiac Ultrasound Imaging," in *Proceedings IEEE Ultrasonics Symposium*, 2012.

[16] ——, "Implementing Capon Beamforming on a GPU for Real-Time Cardiac Ultrasound Imaging," *Submitted to Transactions on Ultrasonics, Ferroelectrics and Frequency Control*, 2013.

[17] J. I. Buskenes, J. P. Å sen, C.-I. C. Nilsen, and A. Austeng, "An optimised GPU implementation of the MVDR beamformer for active sonar imaging," 2013.

[18] F. Harris, "On the use of windows for harmonic analysis with the discrete Fourier transform," *Proceedings of the IEEE*, vol. 66, no. 1, pp. 51–83, 1978. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1455106
[19] J. Capon, "High-resolution frequency-wavenumber spectrum analysis," *Proceedings of the IEEE*, vol. 57, no. 8, pp. 1408–1418, 1969. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1449208
[20] J.-F. Synnevag, A. Austeng, and S. Holm, "Benefits of minimum-variance beamforming in medical ultrasound imaging," *IEEE Transactions on Ultrasonics, Ferroelectrics and Frequency Control*, vol. 56, no. 9, pp. 1868–1879, Sep. 2009. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5278437
[21] H. Cox, R. Zeskind, and M. Owen, "Robust adaptive beamforming," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 35, no. 10, pp. 1365–1376, Oct. 1987. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1165054
[22] J. N. Maksym, "A robust formulation of an optimum cross-spectral beamformer for line arrays," *The Journal of the Acoustical Society of America*, vol. 65, no. 4, p. 971, 1979. [Online]. Available: http://link.aip.org/link/JASMAN/v65/i4/p971/s1&Agg=doi
[23] V. V. U. Berkeley), "GTC: Better Performance at Lower Occupancy," 2010. [Online]. Available: http://nvidia.fullviewmedia.com/gtc2010/0922-a5-2238.html
[24] Nvidia, *CUDA C Programming Guide v4.2*, 2012. [Online]. Available: http://developer.nvidia.com/cuda/nvidia-gpu-computing-documentation
[25] ——, *CUDA C Best Practices Guide v4.1*, 2012. [Online]. Available: http://developer.nvidia.com/cuda/nvidia-gpu-computing-documentation
[26] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A Survey of General-Purpose Computation on Graphics Hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, Mar. 2007. [Online]. Available: http://doi.wiley.com/10.1111/j.1467-8659.2007.01012.x
[27] R. E. Hansen, H. J. Callow, T. O. Sæ bø, S. A. Synnes, P. E. Hagen, G. Fossum, and B. Langli, "Synthetic aperture sonar in challenging environments: Results from the HISAS 1030." in *Proceedings of the 3rd International Conference & Exhibition on Underwater Acoustic Measurements: Technologies and Results*, 2009.

Jo Inge Buskenes received the B.Sc. degree in electrical engineering from Gjøvik College University, Norway, in 2007, and the M.Sc. degree in instrumentation for particle physics from the University of Oslo, Norway, in 2010. He is currently pursuing the Ph.D. degree in image reconstruction and technology at the University of Oslo.

His industry experience includes the European Organization for Nuclear Research (CERN), Geneva, Switzerland (2007-2008), and the Norwegian Defence Research Establishment, Kjeller, Norway (2009). He has lectured in digital signal processing at the Gjøvik College University (2009), and at the University of Oslo (2010-2012). His research interests include adaptive beamforming, digital image reconstruction, high performance computing, intelligent detector design and open source software.

Jon Petter Åsen (S'12) was born in Porsgrunn, Norway in 1986. He received the B.Sc. and M.Sc. degree in computer science from the University of Oslo, Norway, in 2010. He is currently pursuing his Ph.D. degree in medical ultrasound technology at the Norwegian University of Science and Technology (NTNU) Medical Imaging Lab (MI-Lab), Trondheim, Norway. His research interests include adaptive ultrasound processing techniques and acceleration of ultrasound algorithms using Graphics Processing Units (GPUs).

Carl-Inge Colombo Nilsen (S'06-M'10) received the M.Sc. and Ph.d. degrees in computer science from the University of Oslo, Norway, in 2005 and 2010. He is currently working at the University of Oslo as a postdoctoral research fellow. His research interests include signal and array processing for ultrasound imaging and other acoustic applications.

Andreas Austeng was born in Oslo, Norway, in 1970. He received the M.Sc. degree in physics in 1996 and the Ph.D. degree in computer science in 2001, both from the University of Oslo. Since 2001, he has been working at the Department of Informatics, University of Oslo, first as a postdoctoral research fellow and currently as an associate professor. His research interests include signal and array processing for acoustical imaging.