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

Jo Inge Buskenes, *Student Member, IEEE,* Jon Petter Åsen, *Student Member, IEEE,* Carl-Inge Colombo Nilsen, *Member, IEEE,* Andreas Austeng, *Member, IEEE*

Abstract

The Minimum Variance Distortionless Response (MVDR) beamformer has recently been proposed as an attractive alternative to conventional beamformers 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 continuously 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 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 a million pixels per second (1 MP/s). Compared to our initial CPU implementation the optimizations described herein led to a speedup of more than 2 orders of magnitude, or an expected 5-10 times improvement had the CPU received similar optimization effort. This throughput enables real-time processing of sonar data, and makes the MVDR a viable alternative to conventional methods in practical systems.

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

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.

(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 literature 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 for sonar systems up to 32 channels, since here 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.

Our implementation is adapted to a well sampled 100 kHz wideband active sonar system operating in the near field. The system has a range resolution comparable to the sample period so our MVDR implementation runs in near single snapshot mode. To deal with this and to avoid signal cancellation we compute and average the covariance estimates from overlapping subarrays [9]. This may resemble techniques such as "subarray MVDR" [15], but unlike that technique we are not using subarrays to reduce complexity but to make it feasible to use MVDR in an active system. In related work, we investigated using the same MVDR GPU implementation

in cardiac ultrasound imaging [16], [17], and in active sonar imaging we have compared the performance of this implementation to comparable adaptive methods [18].

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 efficiently our implementation makes use of the GPU's resources.

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 in range and direction is commonly referred to as beamforming, and involves assigning suitable delays and weights to the array's channels.

A. Beamforming

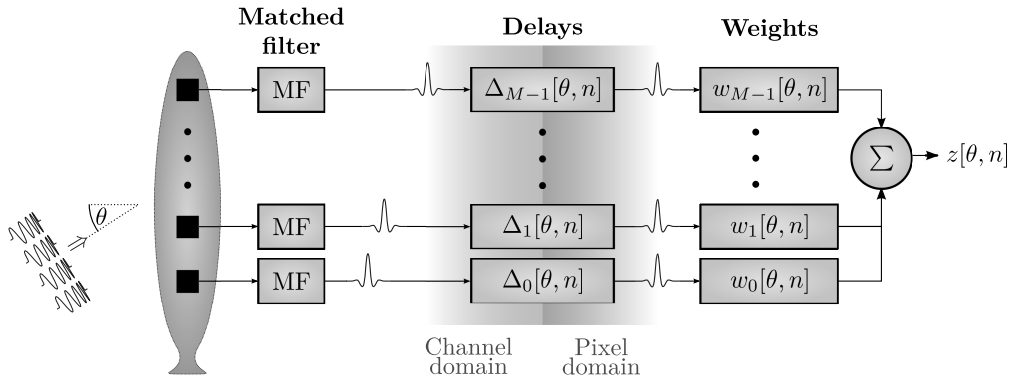

Fig. 1. Beamforming principle. Signal signature is first removed by matched filtering. Then - before summation - a suitable set of delays, Δ, and weights, w, are applied to focus on a pixel of interest at angle and range (θ, n).

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 focus at a pixel with azimuth angle θ and range sample n, such that the delayed data from the mth channel can be expressed as $x_m[\theta, n]$. To simplify notation, we make the dependence on θ 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:

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

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 [19].

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 [20]. This is a convex optimization problem that can be solved using Lagrange multipliers to yield the solution:

$$\boldsymbol{w}[n] = \frac{\boldsymbol{R}^{-1}[n]\boldsymbol{a}}{\boldsymbol{a}^T\boldsymbol{R}^{-1}[n]\boldsymbol{a}}, \tag{2}$$

where $\boldsymbol{a}$ is a steering vector 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 we pre-steer our data to every pixel in the image we simplify (2) by substituting $\boldsymbol{a}$ with a row vector $\mathbf{1}$ that represents broadside phase-steering. To estimate $\boldsymbol{R}$ we compute a sample covariance matrix $\hat{\boldsymbol{R}}$. In this computation we perform some degree of:

- *spatial averaging* to avoid signal cancellation by decorrelating coherent echoes [9],
- *temporal averaging* over an interval comparable to the pulse length (1-5 samples) to maintain true speckle statistics [21], and
- *diagonal loading* to improve robustness to parameter errors [22], [23].

Combined, these steps will also ensure a numerically well conditioned $\hat{\boldsymbol{R}}$.

We perform *temporal* and *spatial averaging* first and put the result in an intermediate sample covariance matrix $\breve{\boldsymbol{R}}$. To do this we need to segment our array into subarrays. If we let $x_l[n]$ represent the data vector from subarray l,

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

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

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

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]:

$$\hat{\boldsymbol{R}}[n] = \breve{\boldsymbol{R}}[n] + \boldsymbol{I} \frac{d}{L} \text{tr}\{\breve{\boldsymbol{R}}[n]\}, \tag{5}$$

where $\boldsymbol{I}$ is an identity matrix, $\text{tr}\{\cdot\}$ represents the matrix trace operation, and $\text{tr}\{\breve{\boldsymbol{R}}[n]\}$ 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, before computing the beamformer output as in (1). Or, equivalently, we may apply the weight set to the sum of all the subarrays:

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

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.

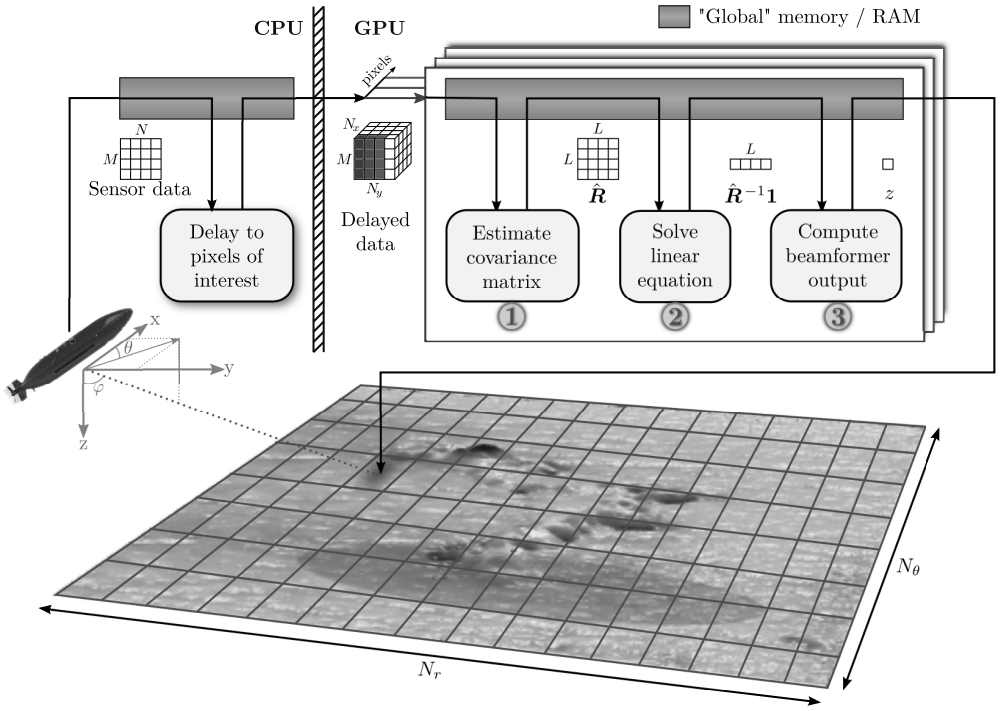

Fig. 2. MVDR beamforming. For each pixel in range and azimuth,

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 z is computed from (6), where $\boldsymbol{w}$ is found by substituting $\hat{\boldsymbol{R}}^{-1}\mathbf{1}$ into (2).

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 account for the total number of arithmetic operations for each step in the MVDR process (see appendix A). The only excluded operation was partial pivoting used in the inversion step, but this should 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.

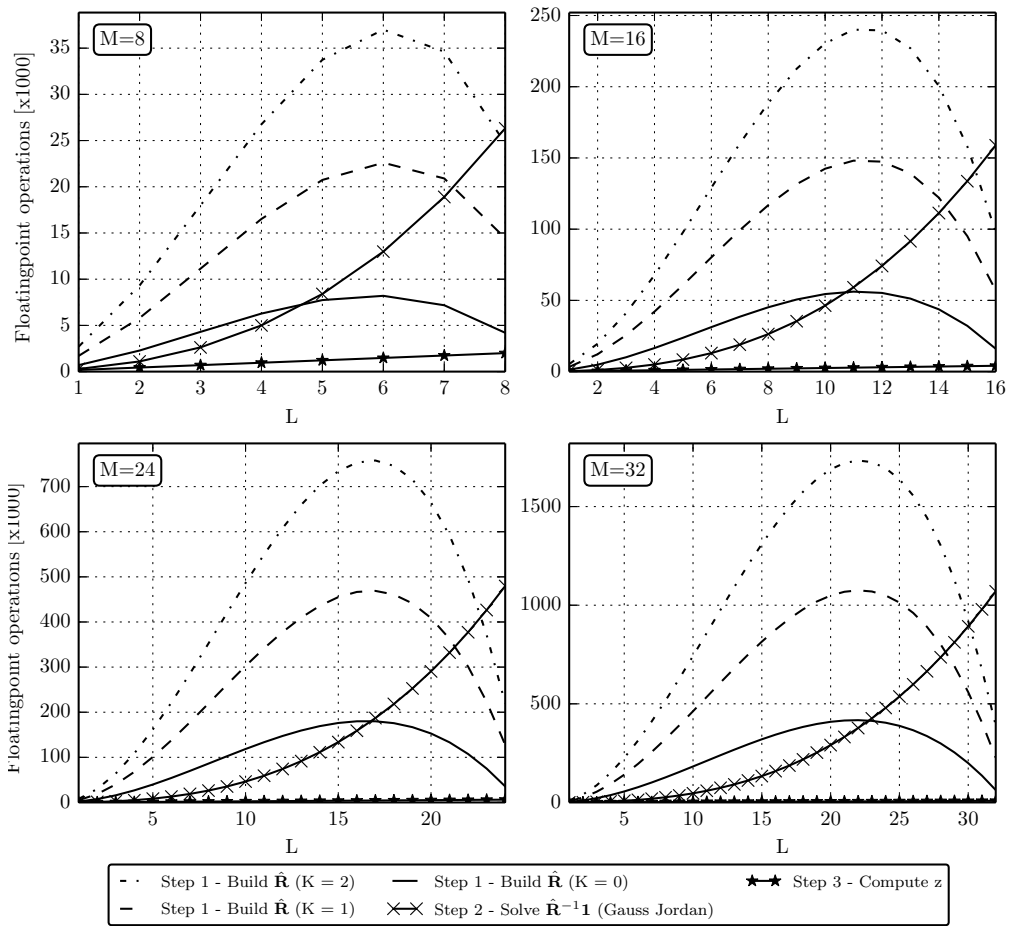

Fig. 3. Per-pixel computational complexity of the steps in MVDR beamforming (before any optimizations). To avoid signal cancellation in an active sonar system we usually set $L < \frac{M}{2}$, in which region the computation of $\hat{\boldsymbol{R}}$ dominates in terms of arithmetic complexity, especially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

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, which performs similarly to the consumer grade Geforce GTX 480. This is a high-end Compute Unified Device Architecture (CUDA) enabled GPU based on Nvidia's Fermi architecture, which as now been superseded by the Kepler architecture. The code was written in Nvidia's "C for CUDA" framework, which adds a few extra features to the C language to specify how to distribute work across a large number of threads. It would have been possible to use GPUs from AMD and the cross-platform OpenCL framework from the Khronos group instead, but CUDA has a batch linear equation solver [24] that no OpenCL library provides. As will be discussed later, this is a key component in our design.

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 B). 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.

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 *minimizing arithmetic operations*, then aim to get as close to the peak *arithmetic throughput* on the GPU as possible.

1) Minimizing Arithmetic Operations: 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

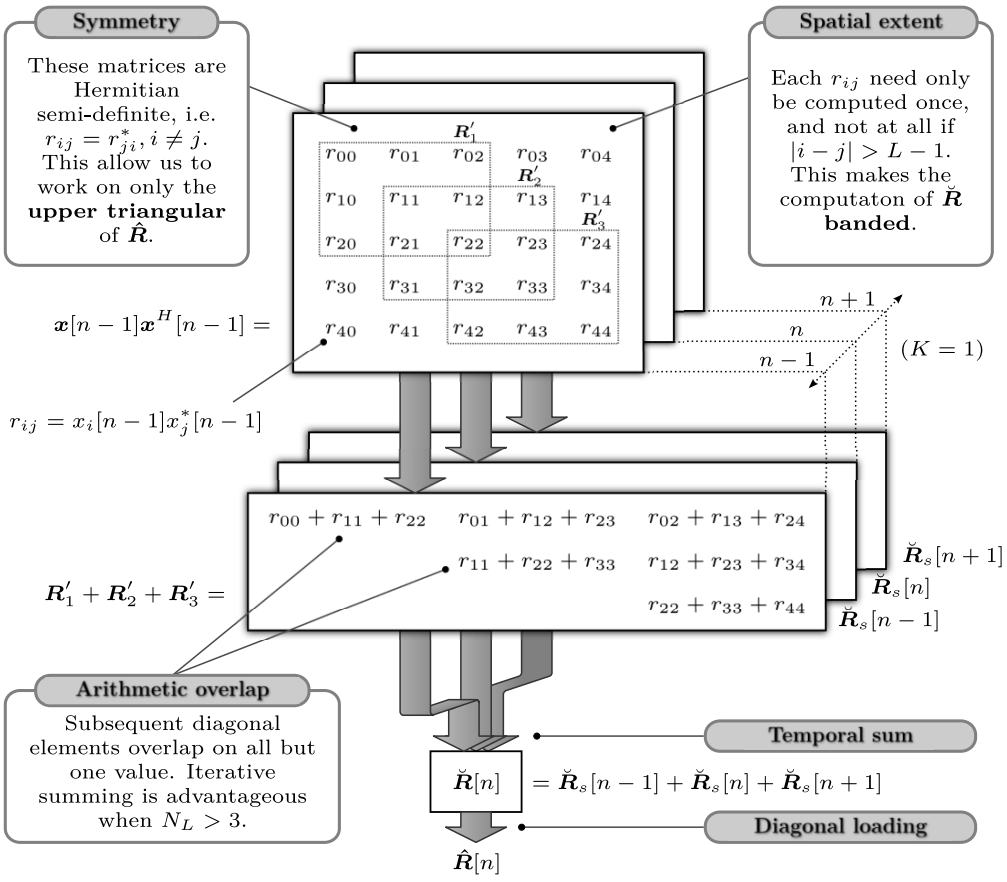

Fig. 4. Step 1: Building $\hat{\boldsymbol{R}}$. This is a visualization of how $\hat{\boldsymbol{R}}$ could be built in a case with $M = 5$ sensors, with subarray size $L = 3$ and temporal averaging set to $K = 1$. Here $\boldsymbol{R}_l'$ is the sample covariance matrix for the lth subarray, and $\breve{\boldsymbol{R}}$ 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}.

3) avoid redundant summations by first computing the upper row of $\hat{\boldsymbol{R}}$ and then add/subtract to these iteratively to find 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. When all optimizations are applied we call it a ”minimum instructions” solution, and when step 2 is left out we call it a ”minimum memory” solution. Fig. 5 compares the two solutions, and formulas may be found in appendix A. Here the complexity curves are relative to the reference implementation in Fig. 3. Both solutions reduce the complexity considerably,

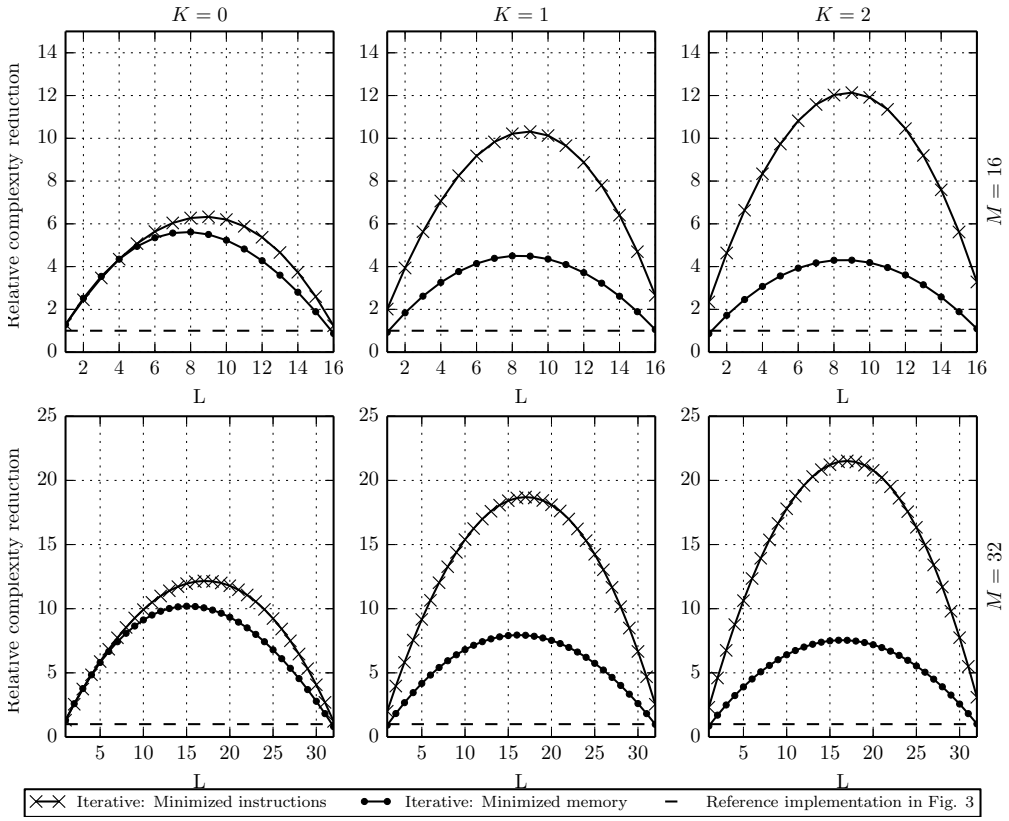

Fig. 5. Arithmetic optimization of computing $\hat{\boldsymbol{R}}$: Relative reduction in arithmetic complexity compared to the initial implementation shown in Fig. 3 (higher is better) . Note how the arithmetic count is reduced by a factor 4-10 in the memory optimized case, and by a factor 6-22 in the instruction optimized case.

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 arithmetically 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 B for derivations). Global memory (RAM), in which all data must reside at some point, is at best only able to move one float for every 30 floats processed by the CUDA cores, and potentially much worse if care is not taken to use ”coalesced” memory access patterns (that avoid memory bank conflicts). This is why the usage of global memory should be minimized. Shared memory, on the other hand, is a very fast level 1 cache that is

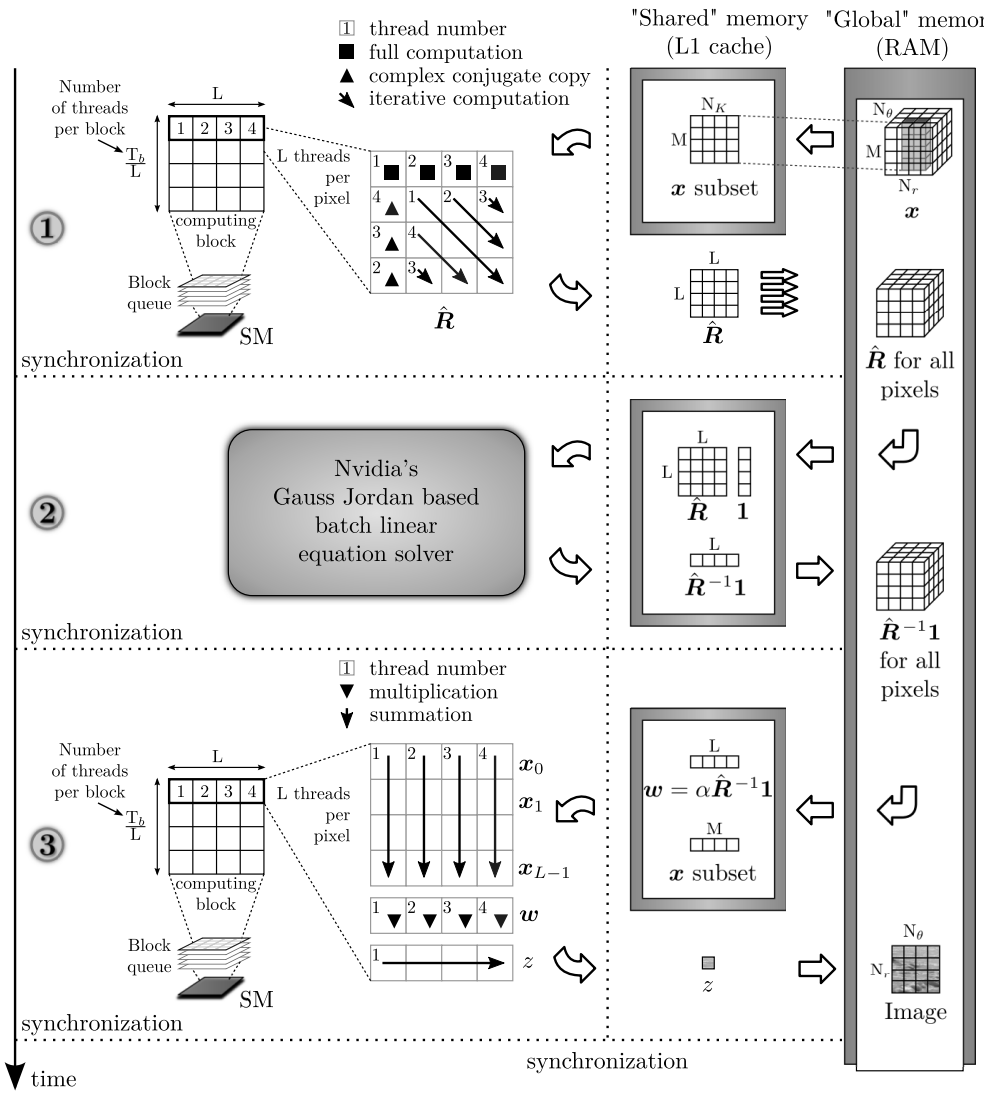

Fig. 6. MVDR implementated on a GPU. We do this in 3 steps, where each step process the full image before moving on to the next step. *Step 1*: The sample covariance matrix $\hat{\boldsymbol{R}}$ is formed by threads running along its diagonals. This allows spatial averaging to be implemented in a computationally efficient manner and minimizes inter-thread communication. *Step 2*: $\hat{\boldsymbol{R}}^{-1}\mathbf{1}$ is computed using the heavily optimized batched linear equation solver from Nvidia [24]. *Step 3*: The beamformer output z is computed in a straight forward fashion using L threads that first sum the subarrays and then apply the MVDR weighting function. A single thread finally sum all the channels up.

shared by all threads on an SM. At peak performance it is almost able to keep up with the computing units, but to obtain this performance we must make sure that:

1) the data needed to compute a pixel can fit in shared memory, while
2) the access patterns we use promote maximum bandwidth.

These challenges are very closely linked. Since the Quadro 6000 architecture is of compute capability 2.0, it has 48 kB shared memory (L1 cache) and 128 kB registers per streaming multiprocessor (SM) [26]. This memory is shared by all active threads on that SM, a number that should be no less than 768 [27]. This is to expose a sufficient level of data parallelism to ensure that memory latency is completely hidden (in accordance with Little's law). If we divide 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, 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 get 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 having to read 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 row is saved back to global memory following a coalesced 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. This is slightly non-optimal as it causes divergent branching, but

	$\mathbf{B}_{\text{arith}}$	$\mathbf{B}_{\text{mem}}$	$\mathbf{B}_{\text{mem}}/\mathbf{B}_{\text{arith}}$
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 [25]

TABLE I

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

is needed because the solver is Gauss Jordan based and thus require a full $\hat{\boldsymbol{R}}$. Combined these steps balance the load evenly on all threads, is almost completely free of arithmetic redundancy, and consumes less memory.

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

As intuition might suggest 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 $\boldsymbol{b} = \hat{\boldsymbol{R}}^{-1}\mathbf{1}$. This is the preferred approach since the solution is obtained directly and without the added computational cost of the final inner product.

An important thing to note, however, is that unlike the problems most GPU 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 tailored for this particular task. It is 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 solver based on Cholesky decomposition. These are designed for Hermitian positive semi-definite matrices such as our covariance matrix, and require only half the arithmetic operations of a Gauss Jordan solver.

C. Computing z

The beamformer output z is computed on a per-pixel basis by first computing the MVDR weights $\boldsymbol{w}$, which are merely scaled versions 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.

D. Implementation summary

The GPU design is summarized and compared to our CPU designs in table II.

	GPU	CPU
Pixel processing	In batches	One by one
Threads per pixel	Multiple	One
1. Building $\hat{\boldsymbol{R}}$:		
Values computed	All	Only upper triangular
Optimization	Minimized memory consumption	Minimized number of instructions
2. Computing $\hat{\boldsymbol{R}}^{-1}\mathbf{1}$:		
Method	Nvidia's batch	Custom inline
	Gauss Jordan solver	Cholesky solver
Cache utilization	Efficient	Not efficient
Data transfers	Minimum	Moderate

TABLE II

IMPLEMENTATION SUMMARY.

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 [28]. HISAS1030 is a high resolution synthetic aperture sonar with an array length of 1.2 m, operating frequency of 100 kHz, and bandwidth of 30 kHz. The element size and spacing is 2.5λ and the opening angle is 25°. To produce the image shown in Fig. 7 the sonar 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 1 megapixel (MP) MVDR image were here processed with parameters $L = 16$, $K = 1$, 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 MP image from the data from a 32 channel array, for all subarray sizes L, and for $K \in \{0, 1, 2\}$. 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. 10. We were also granted a test drive at the Boston HPC centre on a machine with an Intel Xeon E5-2670, 32 GB RAM and an Nvidia K20. The results from this run are shown in Fig. 11.

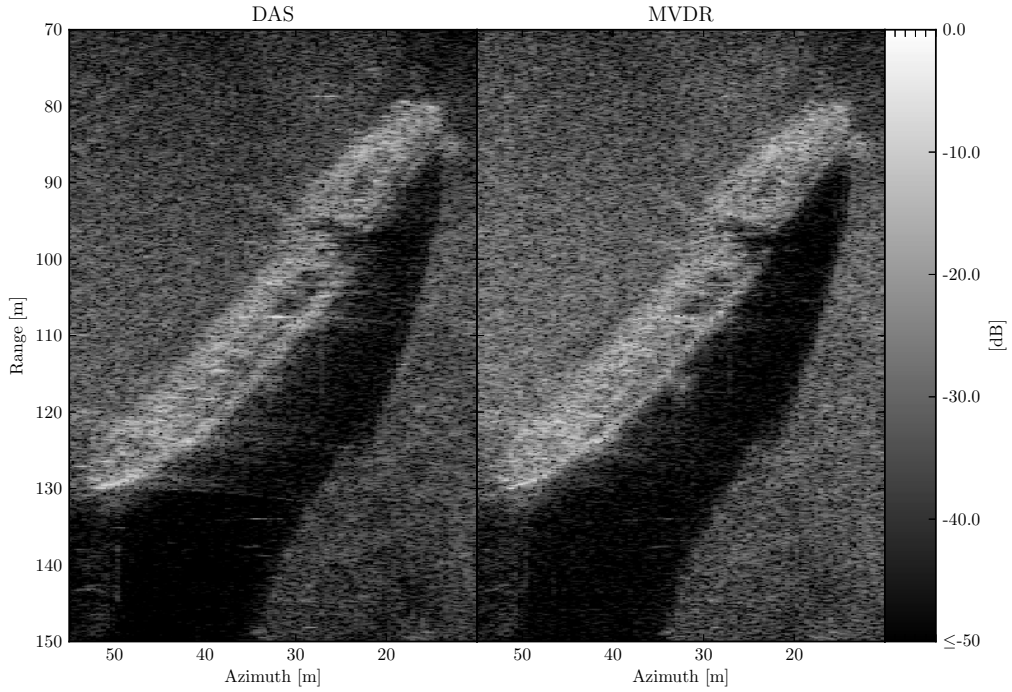

Fig. 7. HISAS sidescan sonar (SSS) image of the shipwreck Holmengraa that lies on a slanted seabed at 77 m depth outside of Horten, Norway. The image was processed with $M = 32$, $L = 16$, $K = 1$ and $d = 1\%$. The image was linearly upinterpolated by a factor 2 in azimuth making its total size 1.46 megapixels (MP). Note how MVDR improves edge definition and reduces noise in shadow regions.

As seen in the run-time comparisons presented in Fig. 8, the GPU method became 2-3 orders of magnitude faster than the C implemtation we started out with. The remaining bottleneck in the final design is 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 MP/s, and this was only improved by a factor 1.5-2 when run on the new K20 GPU. The benchmarks of our memory-only and arithmetic-only kernels show that the kernels spend roughly the same time on both these tasks, so optimising only one of these further will have marginal impact. All GPU kernels were compiled with `nvcc` 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. We believe it makes little sense to include them since it keeps getting easier to perform these data transfers in parallel by offloading the task to the Direct Memory Access (DMA) controllers present on modern GPUs. Furthermore, the data rates in active sonar are relatively low compared to the bandwidth available for these transfers.

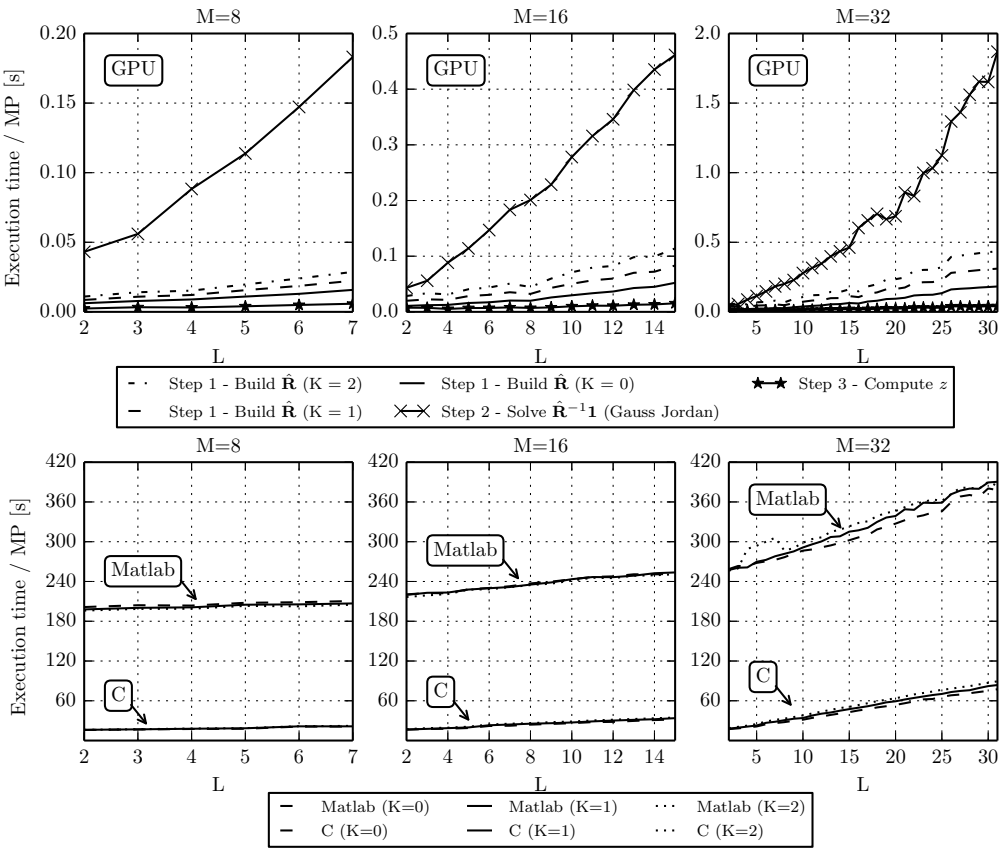

Fig. 8. MVDR benchmarks. A 1 MP image from a $M = 32$ channel array was processed for all L, and for $K \in \{0, 1, 2\}$.
Top: The time the GPU spent on building $\hat{\boldsymbol{R}}$, solving $\hat{\boldsymbol{R}}^{-1}\mathbf{1}$, and computing z. Note the major speedup of building $\hat{\boldsymbol{R}}$ when compared to the complexity plot in Fig. 3.
Bottom: Compared to a reference MATLAB and single thread C implementation running on a CPU the GPU offered a speedup of 2-3 orders of magnitude, but these numbers are somewhat misleading. If the C implementation was properly optimized we expect the GPU to be no more than a factor 5-10 faster, even if its theoretical peak performance is 20 times higher than that of the CPU.

V. Discussion

In accordance with previous studies, Fig. 7 demonstrates the MVDR beamformer's ability to produce images with suppressed interference and improved detail resolution. Compared to 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 no adjustments had to be made when processing other parts of the

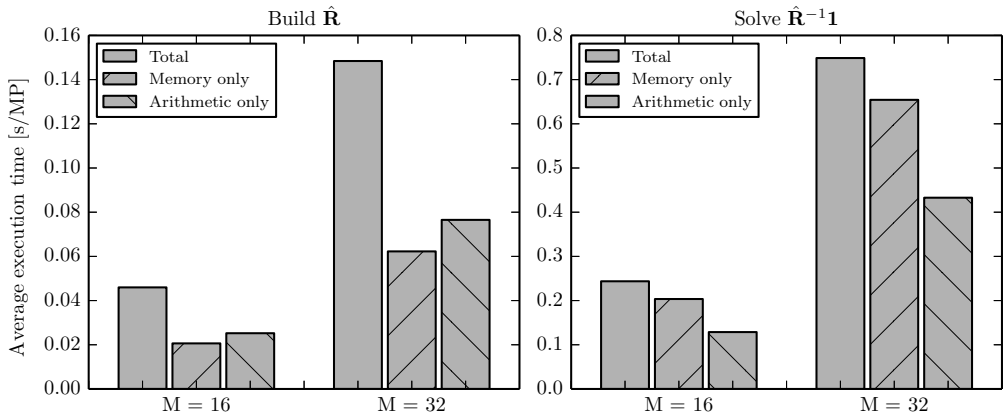

Fig. 9. Execution time of an arithmetic-only and a memory-only version of the MVDR code. A dataset from an $M = 32$ array was processed for all L using $K = 1$, and the mean execution time for a 1 megapixel (MP) image was used here. From this plot we can infer that the kernel building $\hat{\boldsymbol{R}}$ is memory bound, as the time the kernel spends performing memory transactions is higher than the corresponding time it spends carrying out arithmetical operations. Furthermore, when the total runtime is larger than the restricted kernels this can largely be attributed to latency, which we can see that building $\hat{\boldsymbol{R}}$ suffers from with the chosen parameters.

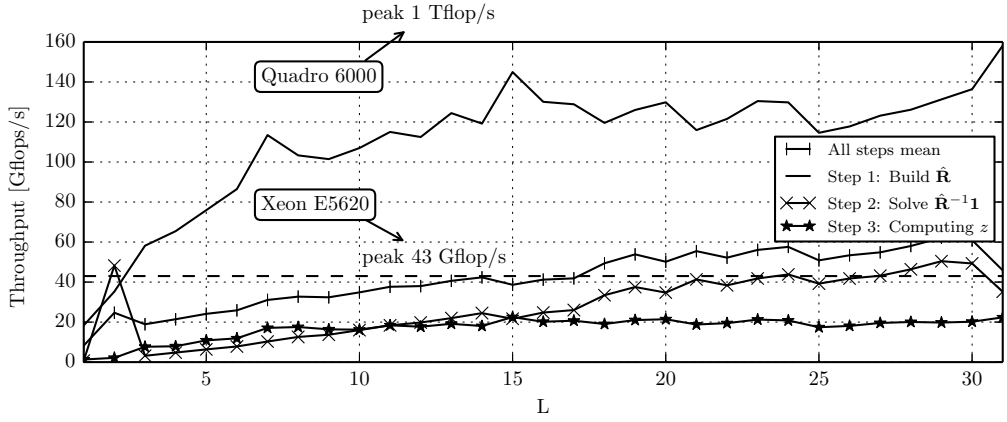

Fig. 10. Code efficiency. An estimate of the number of floating point operations per second (Flop/s), found by dividing the theoretical complexity curves by actual run-times. This is a crude measure as it does not include any other instructions than the actual arithmetic operations in the MVDR computation.

scene.

As observed in Fig. 8 the combination of minimizing arithmetic operations and implementing the MVDR on a GPU lead to a significant improvement in processing speed. Compared to a MATLAB and single thread C implementation an order 2 and 3 speedup was achieved,

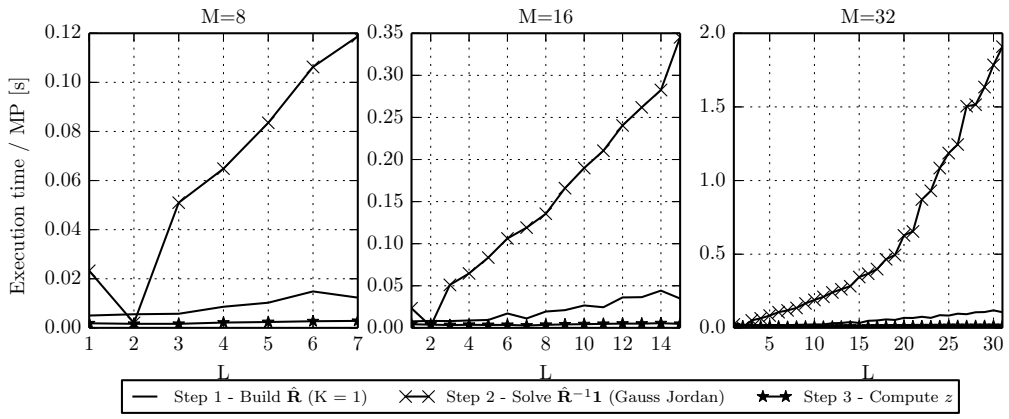

Fig. 11. MVDR benchmarks from Boston HPC centre with the new high-end Nvidia K20 Kepler GPU. The exact same scenario and code as in Fig. 8 was used here. With no code alterations the performance was only improved marginally compared to running on the Quadro 6000.

respectively. Note that 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 CPU was far from reached in our initial implementations. The key reason for this is that the GPU design process pixels in batches, which allows us to minimize data transfers and maximize the use of fast memory cache. In other aspects the designs are similar, both CPU implementations compute $\hat{\boldsymbol{R}}$ efficiently, and they make use of either an optimized custom Cholesky based solver or one from the Intel Math Kernel Library (MKL). Unfortunately we had little time to write a custom batch based solver and covariance builder for the CPU, but we expect the speed difference to be 5-10 times if we had compared two such designs.

Note from Fig. 8 how the building of $\hat{\boldsymbol{R}}$ now only takes up a fraction of the total processing time, and recall that 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 O($N_K N_L L^2$) to less than O(L^2). Another reason for the run-time to appear more like O(L) is likely that 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. In particular, the back substitution step involved in its computation is mostly sequential. This difference can be

observed in Fig. 3. Alternative 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 two, 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 believe this happens because these implementations are dominated by the calculation of the covariance matrix and data movement, and not by the inversion step.

When optimizing a GPU design it is important to know whether it is limited by memory bandwidth or arithmetical throughput. To measure this we made two versions of the MVDR kernel, one that only performs arithmetic operations, and one that only performs memory operations. Then we benchmarked these kernels and compared their run-times to the total run-time (Fig. 9). A first thing to note is how these kernels seem equally occupied performing memory and arithmetic operations. This is a good sign, but since we consistently minimized memory consumption at the expense of some extra computations when building $\hat{\boldsymbol{R}}$, we know it to be bound by memory bandwidth. It also has a problem with latency, which can be inferred from the total run-time being significantly higher than that of the two single-function kernels. Since the GPU hardware can carry out memory and arithmetic operations concurrently and independently, this gap indicates that the GPU sometimes does "nothing". In our case this is due to synchronization hold-ups when performing temporal averaging, which is carried out in a sequential manner.

Even with all our efforts we were only able to obtain processing rates of 40 GFlop/s on average in the desirable range of subarray sizes (Fig. 3). This is mainly due to the inversion step, since we for the most part obtained more than 100 GFlop/s when building $\hat{\boldsymbol{R}}$. While these numbers are slightly underestimated, they are regardless far away from the Quadro 6000's peak of 1 TFlop/s. Again, we believe this is due to the design being bound by memory bandwidth. This belief is further supported by the test results from running the code on the new K20 Kepler card, where we saw a modest factor 1.5-2 speedup. Although the Kepler card peaks at 3.5-4 Tflop/s, its shared memory bandwidth is only approximately twice that of the Quadro 6000, hence matches the observed speed-up. However, it is likely that the Kepler card would perform better had we optimized our design for it. For scientific computing the Kepler boards are considered more attractive, with the introduction of features such as kernels calling kernels, more registers, more shared memory, and better mechanisms for hiding or removing data transfers.

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 consisting of 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 available in modern GPUs. We were able to improve upon the run-time of a single thread C-implementation by roughly two orders of magnitude. For most choices of parameters the GPU was able to create images at $\sim$1 MP/s at an average data processing rate of $\sim$40 GFlop/s. This is less than 5% of the peak performance of the GPU, but we believe it to be near optimal given the constraints of memory bandwidth and the sequential nature of some parts of the MVDR implementation.

All in all, the MVDR maps well to the GPU since the computations involved are independent on the pixel level, and partially also within each pixel. The GPU allows MVDR to be used in real-time processing of sonar data, and makes the MVDR a viable alternative to conventional methods in practical systems.

Appendix A
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):

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

where $O_{\text{dload}} = (2L-1)O_{\mathrm{a}} + O_{\mathrm{m}}$ 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:

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

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:

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

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

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

where r is a running variable r that indexes rows in the augmented matrix $\left[\hat{\boldsymbol{R}}|\mathbf{1}\right]$.

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

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

APPENDIX B
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. The theoretical peak arithmetic throughput is then given as:

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

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:

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

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:

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

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 on running the batched linear equation solver and 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.

[2] J.-F. Synnevåg, 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.

[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.

[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. 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.

[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.

[8] H. L. Van Trees, *Optimum Array Processing*. New York, USA: John Wiley & Sons, Inc., Mar. 2002.

[9] T. Kailath and T.-J. Shan, "Adaptive beamforming for coherent signals and interference," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 33, no. 3, pp. 527–536, Jun. 1985.

[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.

[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.

[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.

[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.

[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.

[15] D. Chapman, "Partial adaptivity for the large array," *IEEE Transactions on Antennas and Propagation*, vol. 24, no. 5, pp. 685–696, Sep. 1976.

[16] J. P. Åsen, 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.

[17] J. P. Åsen, J. I. Buskenes, C.-I. Colombo Nilsen, A. Austeng, and S. Holm, "Implementing capon beamforming on a GPU for real-time cardiac ultrasound imaging." *IEEE transactions on ultrasonics, ferroelectrics, and frequency control*, vol. 61, no. 1, pp. 76–85, Jan. 2014.

[18] J. I. Buskenes, J. P. Åsen, C.-I. C. Nilsen, and A. Austeng, "Adapting the minimum variance beamformer to a graphics

processing unit for active sonar imaging systems," *The Journal of the Acoustical Society of America*, vol. 133, no. 5, p. 3613, 2013.

[19] 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.

[20] J. Capon, "High-resolution frequency-wavenumber spectrum analysis," *Proceedings of the IEEE*, vol. 57, no. 8, pp. 1408–1418, 1969.

[21] J.-F. Synnevåg, 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.

[22] 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.

[23] 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.

[24] Nvidia, "Nvidia registered developers program."

[25] V. Volkov, "GTC: Better Performance at Lower Occupancy," 2010.

[26] Nvidia, *CUDA C Programming Guide v4.2*, 2012.

[27] ——, *CUDA C Best Practices Guide v4.1*, 2012.

[28] R. E. Hansen, H. J. Callow, T. O. Sæbø, 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 acoustical 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.