IMPLEMENTING ADAPTIVE BEAMFORMING ON THE GPU FOR REAL TIME ULTRASOUND IMAGING

J. P. Åsen[1], *J. I. Buskenes*[2], *C.-I. C. Nilsen*[2], *A. Austeng*[2] *and S. Holm*[1,2]

[1]Mi Lab, Norwegian University of Science and Technology, Trondheim, Norway
[2]Department of Informatics, University of Oslo, Oslo, Norway

ABSTRACT

1. INTRODUCTION

Introduction to Capon adaptive beamforming, ultrasound imaging and GPU/CUDA.

Delay and sum beamformer (DAS):

$$z[n] = \sum_{m=0}^{M-1} \mathbf{w}_m^* \mathbf{x}_m[n - \Delta_m[n]] = \mathbf{w}^H \mathbf{x}[n], \qquad (1)$$

where M is the number of elements.

Capon weights are found by solving the following minimization problem:

$$\min_{\mathbf{w}} E\{|z[n]|^2\} = \mathbf{w}^H \mathbf{R} \mathbf{w} \qquad (2)$$

$$\text{subjected to } \mathbf{w}^H \mathbf{1} = 1, \qquad (3)$$

where $\mathbf{R} = E\{\mathbf{x}[n]\mathbf{x}[n]^H\}$.

The solution to this minimization problem is

$$\mathbf{w}[n] = \frac{\mathbf{R}[n]^{-1}\mathbf{a}}{\mathbf{a}^H \mathbf{R}[n]^{-1}\mathbf{a}}, \qquad (4)$$

where $\mathbf{a} = \mathbf{1}$ when $\mathbf{x}$ are pre-delayed and $\mathbf{w} \in \mathbb{C}^M$.

Estimation of $\mathbf{R}[n]$: $\hat{\mathbf{R}}[n] = \mathbf{x}[n]\mathbf{x}[n]^H$.

In order to get a well conditioned $\hat{\mathbf{R}}$, avoid signal cancellation and to get DAS-like speckle, $\hat{\mathbf{R}}$ has to be averaged over $L \leq M/2$ subarrays and N_{avg} time samples.

$$\hat{\mathbf{R}} = \frac{1}{(2N_{avg}+1)K} \sum_{n'=n-N_{avg}}^{n+N_{avg}} \sum_{l=0}^{K} \mathbf{x}_l[n']\mathbf{x}_l[n']^H, \qquad (5)$$

where $K = M - L + 1$, $Y_{avg} \sim \tau/T_s$ (τ is the imaging pulse length and T_s is the sampling period.) and $\mathbf{x}_l$ is the l^{th} subarray $[x_l[n], \ldots x_{l+L}[n]]$.

Finally, $\hat{\mathbf{R}}$ is loaded with diagonal factor ϵ, $\hat{\mathbf{R}} = \hat{\mathbf{R}} + \epsilon\mathbf{I}$. A weighting proportional to the output power is beneficial, and the trace of $\hat{\mathbf{R}}$, $\epsilon = d * tr\{\hat{\mathbf{R}}\}$, has previously shown good results in the literature (cite).

Building and calculating the inverse of $\hat{R}$ is computational demanding, limiting the methods accessibility. Following the innovation in GPU computing, this has now started to change.

2. METHOD

2.1. Building the Sample Covariance Matrix

Building $\hat{\mathbf{R}}$ using a sliding window approach across K and N_{avg}. We have a limited amount of fast, near-core memory on the GPU. On NVIDIA architecture this memory is known as shared memory, and is restricted to 48KB per compute block. The maximum block size of 32x32, further restrict how large arrays we can handle inside a single compute block. It is therefore natural to restrict L to a maxima of 32 elements. Since $L \leq M/2$, $M \leq 64$. We can not afford to hold the full $\hat{\mathbf{R}}$ ($M = L$) when $\hat{\mathbf{R}} > 32$, because of limited amount of memory. ... Explain in detail how `buildR` is implemented in a kernel. ... Give more details on how much shared memory we have, and how it can be divided between compute blocks occupying one stream multiprocessor (SM).

Discuss the bottleneck adaptive diagonal loading causes.

Time averaging takes a lot of resources. It has been shown that time averaging can maintain both resolution and delay-and-sum-like speckle. The same speckle statistics can be obtain with small subarrays, but then we loose resolution. Small sub-arrays benefits from being less computational demanding.

2.2. Solving Multiple Small Linear Systems

Most focus on large matrices in the literature. We have used an unreleased GPU implementation of Gauss Jordan (GJ) elimination (by NVIDIA) to solve $\mathbf{R}\mathbf{b} = \mathbf{1}$, where $\mathbf{b} = \mathbf{R}^{-1}\mathbf{1}$. Can include details on GPU implementation of $\mathbf{U}^H\mathbf{D}\mathbf{U}$, but this has not proved to be faster than NVIDIA's GJ implementation. However, the complexity for solving with $\mathbf{U}^H\mathbf{D}\mathbf{U}$ decomposition is supposed to be $1/3$ of GJ.

In the final journal article we could discuss Conjugated gradient and Woodbury and the benefits they bring.

3. RESULTS

Present images before and after Capon weights has been applied. Comment on the effect of selecting different parameters. Present graph showing running times for Cython-Capon v.s. CUDA-Capon for different choices of parameters.

(A prerequisite for this is to get the GPU-processing up and running on simulated data before Easter!! Maybe also a Matlab plug-in.)

4. DISCUSSION

Discuss results presented in Section 3.

5. CONCLUSION

...