Review of: Jo Inge Buskenes, Jon Petter Asen, Carl-Inge Colombo Nilsen, Andreas Austeng, "An Optimized GPU Implementation of the MVDR Beamformer for Active Sonar Imaging"

January 14, 2014

The authors describe their work on implementing an MVDR adaptive beamformer for a high-frequency imaging sonar on a particular graphical processing unit (GPU), the prevous-generation NVIDIA Quadro 6000, using NVIDIA's CUDA C environment. The authors report 1-2 orders of magnitude faster processing with their GPU implementation, compared to their Matlab and single-thread, unoptimized CPU implementations, without taking the input/output of data in and out of the GPU.

Because the paper is a valuable result to imaging sonar users, who will (to some extent) know that they can speed up their processing by 1-2 orders of magnitude if they use a GPU, I think the paper should be published.

However, I think the authors could greatly improve the appeal of the paper to a wider audience by adding helpful context and background on their implementation. I will suggest some questions that I think the authors could easily address toward this end.

This is a report from the "front lines" of migrating scientific computing to GPUs, a topic which seems to be of great interest to everybody in our field that does numerical computing. In my review, I will comment on the information it provides to a reader interested in starting to use GPUs (and the authors for some selective elaboration).

Suggestions for improving the exposition of the processing

I suspect many readers of the paper will not have the same assumptions as the authors, if they are not processing high-frequency narrowband signals with undersampled arrays. I see this disconnect between low-frequency passive processors and mid-frequency active processors, but the gap to imaging sonars is even greater. I venture that the authors can greatly broaden the appeal of their paper by addressing the questions I pose below.

1. Here are my calculations of how the 32 elements are spaced in the HISAS 1030 system:

$$f_c = 100000$$

$$c = 1500$$

$$\lambda = c/f_c = 15\,mm$$

$$d = \lambda/2$$

$$D = 1200\,mm$$

$$N = D/d = 160$$

2. What is impact of using only 32 elements? How are they spaced? It would seem that spatial aliasing would be an issue. Authors should explain what their assumptions are.

3. What is bandwidth of transmitted waveform? At 100 kHz, perhaps the authors are comfortably in a narrowband regime, and they do not need to worry about array response changing over their bandwidth (which would suggest that MVDR would need to be performed at different frequencies independently, unless some sort of pre-steering was used, which may be what the authors are doing anyway). Authors should explain what their assumptions are.

4. Authors show solution of MVDR optimization as $w = \frac{R^{-1}1}{1^T R^{-1}1}$ and state this is the broadside solution. Are the authors pre-steering the array for every angle? Or are they using $w = \frac{R^{-1}s}{s^H R^{-1}s}$ at off-broadside angles, with s the conventional steering vector (which would be a frequency-domain phase-shift vector, to be applied to a narrowband signal)? Pre-steering would enable disparate frequencies over a wide band to be coherently summed, to provide coherent gain in the look direction (over the band) by providing additional sample support for the sample covariance estimation.

5. Authors time average xx^H over $K \epsilon \{0, 1, 2\}$ time samples. Without doing some sort of reduced-rank processing, why is as little as 2 samples enough to estimate the covariance? Perhaps this is due to high SNR? At high SNR, though, why are the authors doing MVDR at all? What interference is being canceled, and what is its spatial distribution and stationarity?

6. I read the first paragraph of Section II, "..principle of adjusting the array's focus in range and bearing is commonly referred to as beamforming", and was confused - were the authors doing wavefront curvature beamforming, where every combination of range bin and beam angle required a distinct set of steering weights? After reading further, I would venture that the authors are steering only in angle, and that the index n is over time of arrival, at the resolution of the width in time of the matched filter output (after pulse compression... the authors do not say anything about the waveform, so I have no idea whether there WAS any pulse compression). What is the resolution in time or range?

Suggestions for additional GPU details so that paper can also serve as "field report" on GPU work in general

1. It is unfortunate that so little effort was made to speed up the CPU versions of the code. For example, there are public-domain versions of optimized LAPACK libraries that provide the same sort of linear algebra codes as the authors used in their GPU implementation (for example, GoToBLAS, or Intel's MKL libraries).

2. The extra input/output of data to the GPU is a burden only on the GPU implementation, so would penalize the GPU implementation only, unless the authors were to "hide" the i/o by setting it up to run in parallel with the GPU processing, which is easier and easier with the latest GPU architectures. This should perhaps be pointed out.

3. The authors do not provide much information about the GPU hardware they used, the NVIDIA Quadro 6000. How does this compare to the desktop video cards that reside in most desktop, workstation, server and even high-end laptop systems? What is the vintage of the architecture of this GPU? How has the architecture evolved? For example, the authors tell us the Quadro 6000 ($1900, 6GB VRAM, 448 cores) is a Fermi architecture - this is the previous architecture, having been superceded by the Kepler architecture, with a new Quadro K6000 board ($3600, 12 GB VRAM, 2880 cores). The Quadro boards are high-end boards with video output targeting the CADAM market - there are many cheaper alternatives targeted at gamers and an even more expensive compute-only board called TESLA targeted at high performance computing. It's difficult to say how much of this irrelevant.

4. The authors state that OpenCL is an equally attractive alternative to CUDA. This would be the case if the authors had been able to find something in OpenCL akin to the recently released batch-optimized linear solver code for CUDA. Unfortunately, this is a pervasive story - the support for scientific programming in CUDA is years ahead of OpenCL. The benefit of OpenCL is that it is supposed to be "cross-platform", which is true in theory, but it is hard to imagine being true in practice considering

the wide range of heterogeneous computing elements OpenCL is supposed to cover, from phones to DSP chips to FPGAs.

Not clear how R is being calculated in GPU

Page 5, last paragraph in 1st column is puzzling.

Authors say that "when a thread has finished up a diagonal, we have them wrap around to compute one of the diagonals in the lower triangular of $\hat{R}$." The diagonals are all of different lengths, and, presumably the code to do a copy and conjugation from the upper triangular part to the lower triangular part of $\hat{R}$ requires different instructions than the moving average being calculated along a diagonal in the upper triangular part. The cores MUST execute the exact same code, or "diverge" (with one set executing one set of code while the complementary set sitting idle, then vice versa, as you would get with two if-then-else code sections), since this is a SIMD architecture. What is going on here?

This requires a better explanation.

Opportunities for additional GPU tutorial content?

The authors describe the methodology they used to get things to run fast on the GPU. The analysis of the workload was informative and I think it is very informative for readers to see how the analysis progresses from less hardware specific (operations counts), to algorithm restructuring to reduce operations, to more hardware specific concerns (relative throughput of operations and memory transfers).

Let me suggest some concepts that could be discussed a little bit more to give readers more of a sense of what programming GPUs is like.

1. What does somebody need to program in CUDA? Authors can mention that CUDA C is just C with a few extra language features to specify how to distribute the work across the cores. Authors can mention that there is a nvcc compiler, a profiler, a debugger, math libraries (CUFFT, CUBLAS), and a large number of examples of varying complexity, all from NVIDIA. Authors can mention that CUDA can be used on Windows, MacOSX, or Linux, with an NVIDIA graphics card. Authors can mention that there are also third party CUDA interfaces in Fortran and Python. When discussing the batch-optimized linear solver code from NVIDIA, the authors should cite the NVIDIA site, and also links to CULA and MAGMA, other CUDA versions of LAPACK or subsets thereof.

2. Problems need to have high *"arithmetic intensity"* (the authors use *"computational intensity"*) so that the cores stay busy. If there is not enough arithmetic, the cores sit idle waiting for data to be moved around. The authors do a nice job describing this situation on page 4 and the start of page 5.

3. The authors use the term *"arithmetic reduction"*. Unfortunately, reduction is an important word/topic in GPU work. Perhaps "*minimizing arithmetic operations*" would be better. Reduction is any N-to-1 mapping, which poses a conundrum for N cores having to combine their individual results at a single location all at the same time... you don't want the cores to all line up and wait their turn.

4. On page 4, the authors talk about avoiding global memory, without getting into *"coalesced"* memory access patterns. This might merit a mention - for example, "is at best only able to move one float for every 30 floats processed, and potentially much worse if care is not taken to use "coalesced" memory access patterns (that avoid memory bank conflicts)". On page 5, the authors talk about "a collective access pattern that maximizes global memory bandwidth". Here too, the authors should perhaps use the "coalesced" term, since this is the term used in all of the GPGPU documentation.

Localized comments

1. Equation (2) using a "broadside steering vector" - this is confusing... are the authors only steering at broadside? clearly not... do the authors pre-steer all of the elements prior to forming R so that can "focus" in a wideband sense?

2. Paragraph after equation (2) mentions "spatial averaging to avoid signal cancellation" - this problem might be better identified for readers by citing the Kailath reference and using "signal cancellation due to coherent multipath".

3. Equation 4 has second summation using index n', but this index does not appear in any of the terms inside the summation.

4. Page 2 - R is averaged over K time samples... at what rate is beamformer output produced? is R averaged using a moving window process, so that $R[n]$ is just a rank-1 update to $R[n-1]$ and there is one beamformer output for every input time sample?

5. Page 4, in list of three optimizations - item 3 is not self-evident, but if I understand what the authors are doing, it is described by the 1985 reference by Kailath.

6. Page 4, last paragraph of 1) Arithmetic reduction - authors should identify that the "Minimized memory" version skips step 2) from list of three optimization, and that the "Minimized instructions" version uses all three steps... I had a hard time understanding how the three "arithmetic reduction steps" were related to the plots in Figure 5, and was puzzled about where the "minimized memory" and "minimized instruction" variants of the processing had come from. Perhaps the authors can introduce these variants more clearly and more prominently.

7. On page 6, authors say "The bottleneck in the final design is now the inversion step, which is typically 5 times slower than the build step". Surely, this merits an explanation. What is the final design being compared with? What was the expectation? What changed? Figure 3 shows FLOPs before optimization - this shows build and inversion steps of roughly the same order of magnitude in the right half of the figure. Does Figure 3 set the expectation? Figure 5 shows "relative complexity reduction" of entire process, so there is no information on relative speed of build and inversion steps. Figure 10 shows estimated FLOPs for the two steps, but since both steps have different theoretical complexities (operations counts), and the two steps do not optimize the same way, this does not provide an expectation of the relative times.

8. On page 6, authors say their implementation was improved by factor of 1.5 - 2 when run on a K20. Authors could say a few words about what a K20 is, how it differs from the Quadro 6000, that the Kepler architecture actually has features that are dramatically better for scientific computing if the code is adapted to use them (kernels can call kernels, more registers, more shared memory, ability to eliminate some host-to-GPU i/o).

9. Section I says 1-2 orders of magnitude. Section IV, page 6, says 2-3 orders of magnitude.

10. Reference 9 has two authors, Shan and Kailath.

11. Reference 17 seems to be the reviewed manuscript?