

# Using Graphics Processors to Accelerate Synthetic Aperture Sonar Imaging via Backpropagation

Daniel P. Campbell, Daniel A. Cook

Georgia Tech Research Institute / Sensors and Electromagnetic Applications Laboratory  
{dan.campbell, dan.cook}@gtri.gatech.edu

## Abstract

This paper describes the use of graphics processors to accelerate the backpropagation method of forming images in Synthetic Aperture Sonar (SAS) systems. A GPU-based implementation of backpropagation was developed at GTRI and inserted into sonar and radar algorithm research testbed systems. The GPU accelerated implementation formed a 4000 x 4400 SAS image from 60 seconds of sonar data in 7 seconds, which was hundreds of thousands of times faster than the baseline testbed implementation developed in MATLAB, and 275x faster than a C-based implementation executing on an 8-core i7 platform.

## Sonar Backpropagation

The classical approach to both radar and sonar synthetic aperture image reconstruction is backprojection, or backpropagation. The scheme is straightforward. Each point in the scene being imaged contributes reflections to throughout the recorded data, and has a unique locus of echo returns in the observed data. To compute the value of a single output pixel in the reconstructed image, all that is required is to integrate the data along this locus while multiplying by the complex conjugate of the expected locus<sup>2</sup>. This operation has the form of an inner product, and the reconstructed image can be thought of as resulting from a spatially-varying correlation operation. The inner product measures how similar the measured data is to the expected locus, with a strong correspondence resulting in a bright pixel. To obtain the equation for the backpropagation, consider the ideal locus for a single point scatterer at  $\mathbf{x} = [x \ y \ z]^T$ , denoted by  $\ell(t, u, \mathbf{x})$ , which also depends on the time of recording  $t$  and the along-track position of the sensor  $u$ . The integration is performed over the locus given by the curve  $L$ :

$$f(\mathbf{x}) = \int_L \bar{E}_s(\ell) E_c(\ell) d\ell.$$

As a concrete example, consider the ideal case in which the sensor trajectory is a straight line. The resulting locus  $L$  is a hyperbola:

$$L = \sqrt{x^2 + (y - u)^2}.$$

This style of collection is known as stripmap imaging. It is a special case for which the much faster  $\omega$ - $k$ <sup>3</sup> Fourier-based reconstruction technique may be used. The primary limitation of  $\omega$ - $k$  is that a separate motion compensation step must be performed if the sensor does not fly a perfectly straight line. Motion compensation is not a problem for small perturbations of the flight path, but large distortions result in useless data. In addition, the motion compensation

works best for sonars with narrow beams. Wide beam sensors are also of interest, but motion compensation processing can limit their operating envelope in terms of tolerable deviations from straight-line flight<sup>1</sup>. Lastly, the typical platform speed (1.5 m/s) is an appreciable fraction of speed of sound underwater (1,500 m/s), so image reconstruction algorithms must account for sensor motion between transmission and reception.

Backpropagation for image reconstruction bypasses the limitations described above and enables the achievement of the best quality imagery that can be produced by the sensor. While computationally expensive, backpropagation actually simplifies the processing chain because it obviates the need for several pre- and post-processing steps. Most notably, backpropagation incorporates both motion compensation and georeferencing to any desired coordinate system.

## Accelerated Application

GTRI maintains a MATLAB-based synthetic aperture processing toolbox that supports study of new approaches for radar and sonar knowledge extraction. This toolbox includes an array of primitives to support experimental and prototype processing approaches. For any but the smallest output images, execution time for the backpropagation model are prohibitive. A test 4400 x 4000 image formation was estimated to take over 50 days with the toolbox executing on an Intel i7 based platform. Such runtimes require that algorithm prototyping be done with small images, inferior approaches, or long test and iterate cycles.

In order to improve the ability of algorithm researchers to explore new approaches, and to facilitate transition to deployable systems, we created an accelerated version of the backpropagation image formation module of the toolbox. The accelerated software exploits GPUs to form images much more quickly than the MATLAB version, allowing algorithm researchers to maintain a rapid prototype cycle while working with large images, and using the most flexible and accurate image formation approaches.

## Accelerated Implementation

Backpropagation is well suited to parallelization on many platforms. While the mathematical operations to form the image are embarrassingly parallel, each input datum is used by many output points, and the relationship between output point location and input data used is non-linear. It is therefore straightforward to create an implementation of backpropagation that benefits from parallelism, but challenging to fully optimize it for a specific platform.

Despite the non-linear mapping of input data to output points, the relationship is spatially coherent. For a given pulse of return data, nearby output pixels have similar total

propagation distance, and the therefore depend on sample points that are the same, or adjacent in memory. This enables backpropagation implementations to benefit from traditional caches, particularly those that are organized for two dimensional coherency, such as those used in GPUs.

Our implementation of the SAS image formation module is accelerated with nVidia GPUs, and was developed using CUDA. The implementation uses one thread per output pixel, with two-dimensional blocks to improve intra-block range coherency. Each thread loops over the input pulses, and accumulates correlated output as it traverses the input set. Linear interpolation of return data is implemented with texture sampling operations to exploit dedicated logic. Our implementation accelerated the formation of a test image from an estimated 50 days using the original software to 48 seconds on an nVidia 280GTX-enhanced workstation, and under 7 seconds using a dual Tesla S1070-enhanced server.

Our initial, naïve implementation yielded a significant speedup from the MATLAB version, but we added several additional optimizations to further improve execution times. Some of the most significant are described below.

The biggest improvement came from elimination of register spills. The current CUDA toolchain uses global memory, which has access latency of several hundred cycles, as register spill space, instead of attempting to use shared memory. Because the number of registers used by a kernel can limit the number of concurrent threads on a multiprocessor, it is often beneficial to instruct the CUDA compiler to limit register usage. We found that this practice led to register spills, causing inner loop variables to be temporarily stored in global memory. Instead, we selectively changed local variables to shared memory arrays. Additionally, we found that some calls to trigonometric functions resulted in register spills. These were corrected by using the reduced precision equivalents.

We used texture sampling to improve the performance of the linear interpolation. Once the propagation distance for a pixel-pulse has been calculated, that range must be checked against the bounds of the pulse return data, and converted to a real-numbered sample index. The data are linearly interpolated for an estimated return for the pixel. This step requires 6 flops and 2 memory reads for each of real and imaginary return data, but is nearly identical to graphical texture mapping which is supported by dedicated bounds checking, interpolation, and caching logic on GPUs.

GPU memory accesses can be very expensive relative to math operations, especially in terms of foregone math operations. As a result, the balancing point between best use of precalculation and lookup versus recalculation on the fly can differ from general purpose processors. For example, we found that precalculating the transmitter position, which is shared for all pixels on a given pulse, did not significantly improve execution time.

## Performance Testing & Results

We tested the software on a platform with two Intel Nehalem-based quad core processors operating at 2.4GHz, and two Tesla S1070. We formed a 4400 x 4000 pixel output image from data sets containing 4000 pulses each.

The input data for each image corresponded to sonar data collected over a period of approximately 60 seconds. The data used were collected by the SAS12 system, which was constructed under the sponsorship of the Office of Naval Research<sup>4, 5</sup>. We tested using from one to all of the eight available GPUs, as well as using all eight available CPU cores for a lightly optimized C-based implementation. We also tested the formation of a single, representative column of pixels near the center of the image using the MATLAB implementation to estimate its total runtime. The initial launch of the GPU module incurred a startup delay that is not repeated on subsequent images. We report results for freshly started and pre-warmed state.

The measured runtimes are summarized in Table 1, below. Flops/s was estimated by treating square roots, divisions, and trigonometric functions as single floating point operations. Under these assumptions, the minimum flops per output pixel was 102 per input pulse for SAS. Synthetic aperture radar is similar to SAS, but the much higher ratio of propagation speed to platform speed allows simplifications that reduce that number to 34.

	Runtime (s)		
GPUs	Pre-warm	Fresh	flops/s
1	40.41	44.40	177.7E+9
2	21.08	24.97	340.6E+9
4	11.35	15.41	632.7E+9
8	7.02	11.04	1.0E+12
0 (8xCPU)	1931	1931	3.7E+9
0 (Matlab)	4320000	4320000	1.7E+6

Table 1 - Benchmark Results

These results showed a speedup of approximately 275x for our implementation relative to a lightly optimized, 8-core C based implementation, and a speedup of several hundred thousand times relative to the original MATLAB software. We found that our implementation scaled fairly well up to 8 GPUs, achieving 72% linear speedup.

## References

- [1] H. J. Callow, *Signal Processing for Synthetic Aperture Sonar Image Enhancement*. PhD thesis, Department of Electrical and Electronic Engineering, University of Canterbury, Christchurch, New Zealand, 2003.
- [2] D. A. Cook, "Synthetic Aperture Sonar Motion Estimation and Compensation," Master's Thesis, School of Electrical and Computer Engineering, Georgia Institute of Technology, 2007.
- [3] D. W. Hawkins, *Synthetic Aperture Imaging Algorithms: with application to wide bandwidth sonar*, PhD thesis, Department of Electrical and Electronic Engineering, University of Canterbury, Christchurch, New Zealand, 1996.
- [4] A. D. Matthews, T. C. Montgomery, D. A. Cook, J. W. Oeschger, and J. S. Stroud, "12.75-inch synthetic aperture sonar (SAS), high resolution and automatic target recognition," in MTS/IEEE Oceans 2006 Boston, 2006.
- [5] <http://oceanexplorer.noaa.gov/explorations/08auvfest/logs/mam19/may19.html>