GPU-accelerated synthetic aperture radar backprojection in CUDA

Ahmed Fasih
Department of Electrical and Computer Engineering
The Ohio State University
Columbus, Ohio 43210
fasih.1@osu.edu

Timothy Hartley
Departments of Electrical and Computer Engineering,
and Biomedical Informatics
The Ohio State University
Columbus, Ohio 43210
hartleyt@bmi.osu.edu

***Abstract*—Pleasingly parallel algorithms such as filtered backprojection have been documented to enjoy significant speedups when ported to run on a graphics processor instead of a standard CPU. Presented here is a two-dimensional SAR backprojection implementation for a single GPU using the NVIDIA CUDA framework. Given that input range projections may be too large to fit in graphics memory, our implementation allows the partitioning of the data along range as well as aperture. We show the efficacy of some CUDA optimizations, and apply it to two public datasets, one of which has varying start frequency.**

I. Introduction

Commodity graphics processing units (GPU) are an example of the sea change in computer architecture, underway for some years now. Following the plateauing of clockrates and instruction-level parallelism in 2003, we have witnessed an almost complete transition to multicore computer architectures [1]. GPUs are characterized by dozens of self-sufficient but simple processors, each with several execution units, and support high computational throughput for data-parallel algorithms. Following high-quality software interfaces for general purpose graphics processor programming, such as NVIDIA CUDA released in 2007, numerous studies have demonstrated the tremendous benefit of using GPUs as accelerators for tasks decomposable to the same operations performed on different chunks of data [2], [3], [4], [5]. Tomographic backprojection is an example of a pleasingly parallel algorithm and is well-suited to this platform.

In this paper, we describe an implementation of the direct filtered backprojection algorithm for reconstructing synthetic aperture radar (SAR) images in CUDA. We give a brief overview of tomographic imaging and reconstruction, focusing on its data-parallel essence. Then we examine how this kernel of parallelism maps to the GPU architecture, which allow us to obtain $40\times$ to $60\times$ speedups over a very similar single-threaded CPU implementation. We also explain how to partition the input data into two-dimensional chunks if graphics memory is limited.

II. Tomographic imaging

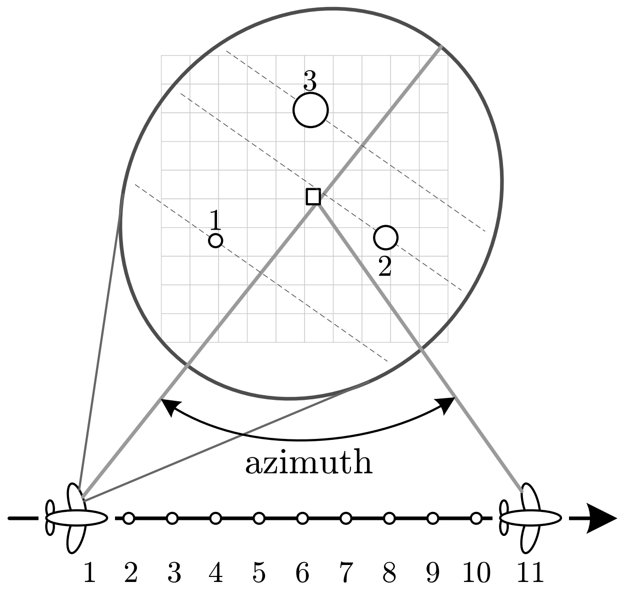
We briefly review the backprojection algorithm. A tomographic sensor collapses a 2-dimensional or 3-dimensional scene into 1-dimensional range profiles, each indexed by the sensor's 3-dimensional location. In the SAR case, the sensor is a radar transceiver placed on an aircraft circling a patch of interest on the ground. The radar repeatedly transmits an electromagnetic pulse at the ground scene and obtains 1-dimensional projections by timing echoes—see Figure 1 for an operational schematic and an example of the data collected, and [6] for a detailed study including the mathematics of the underlying Radon transform.

To reconstruct an $N \times M$ pixel image from a collection of projections as in Figure 1b, each pixel must query each projection to estimate the amount of energy that this pixel, on the ground, initially contributed to that projection. If the radar-to-scene distance is large compared to the scene diameter, and a flat rather than spherical wavefront can be assumed, this process amounts to superimposing each range projection (1 through 11, in the figure's example), rotating it to match the angle it was obtained at relative to the image, and smearing the contents of the range projection perpendicularly across the image pixels. Since the pixels will almost never be exactly perpendicular to the discrete range bins constituting the projections, a costly interpolation step is needed to accurately distribute the energies to the respective pixels.

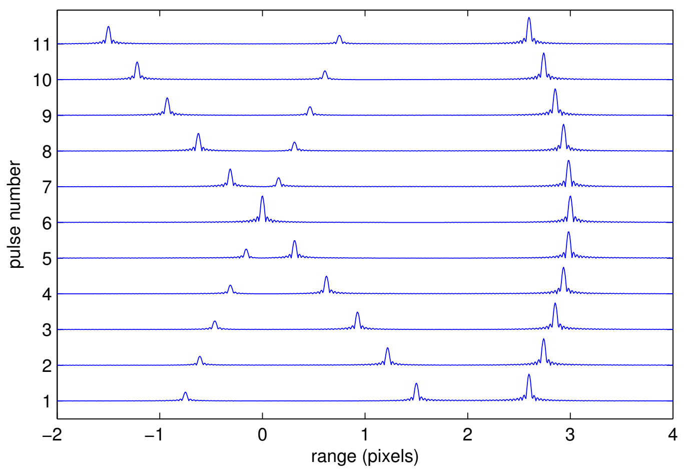
The above makes the algorithm have cubic complexity, growing as NMP for P projections. Although a fair amount of processing is saved by making the far-field assumption described above, it is not warranted for one of the datasets we consider, and so our implementation performs the complete near-field pixel-to-radar range calculation.

Another common tomographic sensor is x-ray CT. Although the method of reconstructing CAT scans and SAR images is similar in principle, there exist a number of important specifics in which they differ, elucidated in [7]. These differences arise from the fact that a CT sensor scans a scene of x-ray absorption values and transduces real-valued signals, whereas a radar obtains complex-valued projections due to the underlying scene being a complex-valued electromagnetic reflectivity function. Thus, there is significant cross-range information encoded in the phase of a SAR collection that is not available for x-ray CT. Practically this gives SAR data a holographic nature, in that a very small amount of azimuth allows one to image the entire scene—in this paper we look at 1° to 8° of projection

This work was supported in part by AFRL award FA8650-07-D-1220.



(a)



(b)

Fig. 1: (a) Schematic demonstration of the tomographic principle. The scene consists of the three targets of different amplitudes (circles), and would produce the range profile data shown in (b). Each line projection contains the linear contributions of all three scatterers.

data to form completely legible images.

In both of these contexts, the original projections must be pre-processed: they must be windowed, filtered for frequency deweighting, and in SAR's case, inverse Fourier transformed to obtain projections in the spatial domain. We currently load the raw phase history data into a programmable environment such as MATLAB or Python, apply this pre-processing, and then send the filtered projections to a compiled C application for projection-interpolation. It is this inner backprojection loop that we seek to accelerate using GPUs, since the preprocessing time is negligible compared to the backprojection time for the imagery we consider.

III. Parallelism in CUDA

A number of CUDA backprojection implementations have been described in the medical imaging literature [8], [9], and many parallelization concepts therein are directly applicable to SAR data. We very briefly outline the CUDA memory and execution models, and then discuss the present implementation.

A. The CUDA execution and memory model

In the CUDA model of the GPU, a kernel is the central data-parallel function that is applied to chunks of data. *Threads* that execute the kernel function can be spawned by the trillions into the device scheduler, partitioning them from the GPU's actual execution capabilities. These threads are arranged in *blocks* that may contain up to 512 threads in a 1d, 2d, or 3d layout (currently fixed for all blocks). Blocks themselves are arrayed in 1d, 2d, or 3d layouts within the one *grid* that occupies the GPU. Each thread is aware of its "location" in its parent block, and that block's location in the grid, and through these location indexes (and dimensions for blocks and the grid), a thread can calculate which chunk of data it is to operate on.

Threads within a block may communicate through shared memory and synchronization barriers, so a block must run on a single "processor." The current generation of GeForce 200 processors (such as the 10-series Tesla) have 30 such units called "multiprocessors," each of which contains instruction logic, 64K registers, 16 KB of shared memory visible to all threads, eight parallel execution units ("processors"), etc. Threads in different blocks may communicate solely through the global video memory (currently 1–4 GB per GPU). However, a defining aspect of GPU programming is the *400–600 cycle delay for arbitrary global memory access.* In contrast, a single execution unit can do a single-precision floating-point multiply-add in four cycles. Access to on-multiprocessor shared memory and registers is as fast as access to the very large register file, provided that certain rules are obeyed about memory bank access distribution.

In order for GPUs' immense computational throughput to be effectively utilized, the enormous global memory access latency must be masked by the other defining aspect of GPU programming: *zero-penalty switching between the millions of blocks which can, but do not have to, execute in parallel.* When a group of threads is waiting for data, execution is switched to another group that is not waiting. Obtaining anywhere near peak performance hinges on designing kernels and block/grid layouts that maximize the number of active blocks per multiprocessor. Doing this while balancing the numerous competing and non-overlapping hardware constraints is the main challenge for the CUDA programmer.

This section is necessarily brief—[10], [11] are recommended for their more detailed expositions—but the concepts described above are needed in order to understand the next section, where we describe how the current backprojection implementation fits into this framework. Additional details about CUDA and GPU hardware that are relevant to backprojection are also introduced.

B. Backprojection in CUDA

First is the matter of block and grid layout. Given that each pixel has to query each projection for the latter's contribution to it, one may either have each CUDA thread be responsible for (i) projections, or (ii) pixels. The main reason to have pixel-based threads is to take advantage of 2d hardware texture fetching and filtering (detailed below). Having not yet studied the effect of having multiple threads for a pixel (effectively parallelizing over projections too), we currently assign one thread to a pixel.

Recall that there exists a 400–600 cycle delay for global memory accesses, which is completely uncached. Although main memory throughput can be maximized by reading contiguous memory locations (and avoiding the memory bank conflicts which serialize threads), a more robust solution is to make use of texture memory, where an array in global memory is designated to be a texture. Requests to data stored as a texture are handled by dedicated texture processors that implement a limited amount of caching, as well as fast - albeit simple - interpolation of data values. Given that there is great spatial correlation between the range bins contributing to neighboring pixels, the texture memory's cache will prevent

- the penalty of requesting the same projection indexes from global memory for close pixels, and
- the divergence of threads caused by intra-block coordination for global memory accesses. (If threads (pixels) in a block take different paths in a branch, e.g., due to an "if" statement, their execution is serialized, degrading performance.)

(We are not aware of a way to ensure that backprojection-like memory fetch patterns be both contiguous and without code branches. If such an algorithm does not currently exist, it certainly may be discovered in the future.)

As mentioned above, beyond the 6–8 KB texture cache available to each multiprocessor, the texturing unit provides *hardware-accelerated linear interpolation*. This allows threads to request non-integer indexes for an array in texture memory, and receive a (bi)linearly-weighted scalar. For the imaging scenarios we considered, the hardware's 8 bits of fractional precision (yielding 256 levels between adjacent array elements) was quite sufficient.

In summary, our implementation sets up the grid/block/thread layout to correspond to the image/sub-image/pixel levels, and leverages the texture cache and hardware interpolation of range profiles.

C. Additional implementation notes

We still have freedom to choose block dimensions, i.e., the dimensions of sub-images that will be processed simultaneously. Attempting to analytically balance the constraints imposed by the hardware memory and execution (such as registers by multiprocessor, but also maximum blocks per multiprocessor, threads per multiprocessor, etc.) can be done but it is very easy for unaccounted factors to render such calculations useless. Therefore, after implementing the kernel, we attempted various combinations of block/grid dimensions, and indeed observed unexpected interactions between these and runtime.

One of the peculiarities of coherent imaging (on complex-valued, phase-aware data like with SAR) is that a phase offset must be applied to each projection's pixel-wise contribution, based on the distance between the aircraft's position corresponding that projection and the pixel's location in world coordinates. The complex exponential $\exp(j\zeta)$, for the phase offset may be evaluated with arithmetic and a tan() operation. CUDA provides access to low-precision hardware-accelerated evaluation of certain common mathematical functions (trigonometric, exponential, etc.), which we are able to leverage, ensuring that we compute $\tan(\zeta) = \tan(\zeta \bmod 2\pi)$ with the latter expression, since ζ may be large and beyond the accuracy of the hardware. This allows us to use an 84-cycle operation (approximately: 32 cycles per evaluation of $\sin$ and $\cos$ each, 16 cycles for reciprocal division, and 4 cycles for multiplication) instead of a much more expensive software implementation of $\tan(\zeta)$.

We also used a 1-dimensional textured array of quad-floats (`float4` type) to store the location of the sensor (which uses three of the four available slots) as well as the distance between it and the scene center for each projection. A thread interrogating a range profile will thus first copy that projection's sensor location data from texture memory (leveraging only cache, not interpolation).

Finally, some data sets have frequency support that varies across the aperture; that is, the minimum frequency of the returned data changes from pulse-by-pulse in a manner dependent on the aircraft's buffeting. We would like to accommodate this within the reconstruction, although it introduces yet another pulse-wise time series that the kernel must receive and process. Given that there are no 5-float data structures, we are forced to store the varying minimum frequency as its own array, either as a texture or simply in uncached global memory. We will show that runtime is much better when this array is stored in global memory.

The phase offset, in addition to dealing with complex-valued data results to a kernel requiring 21 registers (each 32 bits wide). According to NVIDIA's Occupancy Calculator spreadsheet, standard block sizes of 8×8 and 16×16 allow us to reach 50% "occupancy" of each available multiprocessors: half as many warps (groups of 32 threads) as can be run on a multiprocessor will be run. The maximum blocks per multiprocessor for such a configuration (21 registers and negligible shared memory per thread, 8×8 or 16×16 threads per block) is 8 blocks per multiprocessor, and is limited by the maximum number of warps per multiprocessor constraint.

D. Input data tiling

The GPU's texture processors place an upper limit on the size 2d array of range profiles stored as a texture, $2^{15} \times 2^{16}$, and large images may very well have range profiles too large to fit. Thus, we are very interested in partitioning the data in order to fit on the GPU.[1]

Because SAR is a coherent and additive system, the complex-valued images reconstructed from disjoint subsets of range profile data can be summed to obtain a single image. The simplest data partitioning scheme is across the aperture, where for example projections over non-overlapping 1° increments may be imaged separately. Such partitioning requires no change in the kernel or setup; only fewer projections are sent to the accelerated backprojection engine, and this can be handled at the user's level in, e.g., high-level data manipulation environments like MATLAB or Python.

However, sometimes we have range profile data with more range bins than will fit in texture memory. Then, the input data must be partitioned along range bins, and the kernel is given an range bin offset, a scalar float, that allows it to normalize the range bins it requests with the quantity of data actually available in texture memory.

The properties of texturing make this fairly simple. CUDA textures can be clamped, so that requests of indexes less than zero or greater than an array height/width are mapped to the smallest or largest index available. When range profiles are partitioned along range bins, however, this presents a problem: pixels requesting range bins beyond those available will receive incorrect clamped projection values. We resolve this problem by zeroing the incoming range profile data. When all values which can be clamped are zero, range-partitioned sub-images are correctly formed, at the cost of omitting a very small number of range bins of data. This approach may not be suitable for all cases.

IV. RESULTS

Having discussed the parallelization of the backprojection algorithm, we present some empirical results. Despite having stated previously that backprojection is considered to have cubic complexity, we do not present flop counts because a non-negligible portion of each kernel handles ancillary parts of the algorithm. It is also difficult to count texture fetches given the indeterminacy introduced by caching, as well as NVIDIA's secrecy about the texturing hardware.

Furthermore, citing the number of pixel-projections per second is useful only given a specific radar and imaging configuration. Some imaging scenarios, especially in radar signal processing research, involve lightly-sampled range profiled data and images that cover the entire scene scanned. Industrial radar data is often highly oversampled in range, and only a small part of the scene may need reconstruction.

Therefore, we content ourselves by simply comparing execution times between CPU and CUDA implementations of backprojection (as we did previously when reviewing backprojection on heterogeneous clusters with GPU-equipped nodes [5]), as well as image accuracy.

[1]Interestingly, laying the range profiles data so that the aperture runs horizontally results in a faster runtime than the other way.

We demonstrate the importance of the CUDA-specific choices made by comparing different versions of the application to an existing single-threaded CPU implementation, written in C. Both CUDA and CPU versions are called from Matlab's MEX interface. (The GPU kernel is callable from Python via PyCUDA for easy testing and tuning [12]). Both implementations are lightly optimized.

Because we use both CPU and CUDA implementations as serial and parallel accelerators, the runtime comparisons we give will include all memory transfer and conversion overhead (conversion is needed as CPU-side complex-valued data is stored as an structure of arrays, whereas CUDA provides good built-in support for array of structures). This way, we preclude the need to carefully differentiate memory overhead from kernel runtime.

Two SAR datasets made public by the Air Force Research Laboratory have been used in the runtime and image accuracy comparison.

- In the Volumetric Dataset, a 100×100 m patch has been digitally spotlit. It has 429 range bins per projection and a 129 projections per degree of azimuth [13].
- The GMTI Dataset also contains a digitally spotlit data collection, but with a moving scene center. It has 384 range bins per projection and approximately 5000 projections per degree of azimuth [14].

In Figure 2, we illustrate the speedups obtained for a varying aperture extents, as well as image dimensions and CUDA block sizes, for the Volumetric Dataset. Here, we see that 8×8 block dimensions are consistently better than 16×16. These results compare a quad-core 2.66 GHz Intel Xeon with an NVIDIA Quadro FX 5600.

In Table I, we tabulate the effects of variable frequency support for imaging a 90×90 m patch from the GMTI Dataset. We use 4300 projections and 384 range bins, upsampled to 8192 range bins, to produce Figure 3. These results compare a quad-core 2.66 GHz Intel Xeon to an NVIDIA Tesla C1060.

Regarding Table I, for reference note that if frequency variation had be negligible, the CUDA kernel would have taken 1.19 s, and the total accelerator time would have been just 1.8 s. Comparing these to the times achieved for implementations correctly handling variable start frequency, we can see that this feature imposes a non-negligible cost in runtime, as well as code complexity. Assuming that this is necessary, storing the pulse-wise minimum frequencies in global memory is clearly better than using a texture. We also demonstrate the sensitivity of the algorithm to block sizes: 16×16 is, for this dataset, clearly better than 8×8. And finally, the last row shows the performance hit when the range profiles have been partitioned into two equal chunks along range bins. The total accelerator time is significantly greater than the kernel times (the sums of the two individual kernel times), indicating that memory overhead is significant.

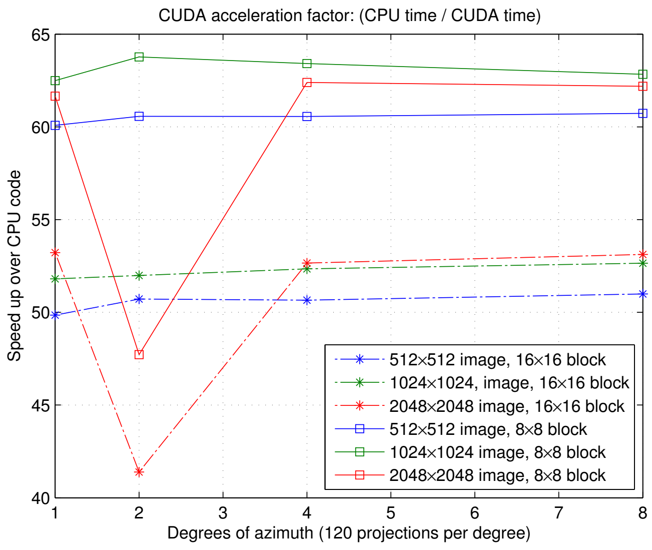


Fig. 2: Volumetric Dataset: CUDA acceleration factors over single-threaded CPU implementation.

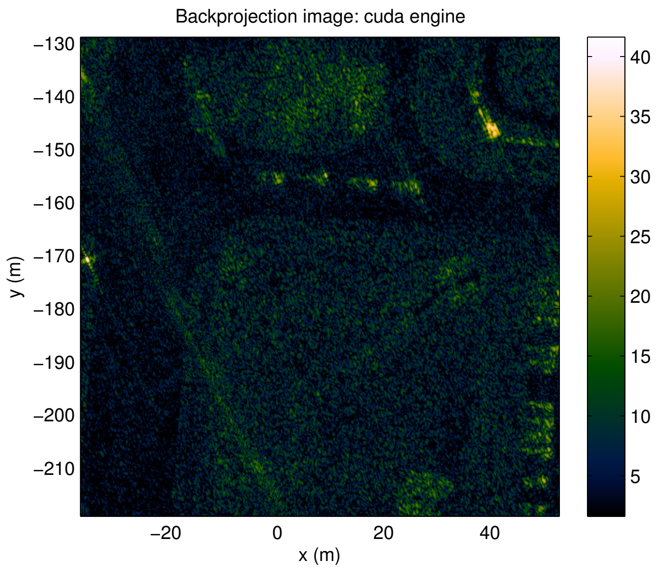


Fig. 3: CUDA image of the GMTI Dataset from Table I.

We may compare the two reconstructions pixel-wise with the following quasi-SNR:

$$\frac{|I_1[n,m]|}{|I_1[n,m]| - |I_2[n,m]|}.$$

Averaging this measure over all pixels in a CUDA- and a CPU-generated image shows an average SNR of > 30 dB.

V. Conclusion

We have outlined the CUDA implementation and design choices made when porting a lightly-optimized C implementation of 2d near-field backprojection to CUDA. We realized significant improvement in runtime, with imperceptible degradations of quality, and also presented the simple modifications

TABLE I: GPU-acceleration timings for GMTI Dataset. Preprocessing, kernel, and accelerator application time in seconds, and acceleration factor above CPU.

Name	Prep.	Kernel	App.	Factor
CPU, 1-threaded	3.3	860	860	1.0
CUDA, freq. in texture	3	2.41	3.1	277
CUDA, freq. in gmem	3.2	1.28	1.9	453
CUDA, freq. in gmem, 8x8	3.1	1.83	2.5	344
CUDA, freq. in gmem, tiled 2	3.1	1.75	3.8	266

needed to tile single phase history collections too large to fit in a single GPU's memory.

Any number of extensions can be imagined. The simplest would be to study more sophisticated CPU backprojection implementations [15], with the expectation that numerous flop-saving techniques also apply to GPU hardware.

We have not yet studied the usefulness of having multiple threads for a given pixel (thereby parallelizing over individual projections). This would be equivalent to creating a number of incomplete images using non-overlapping portions of the data, and then summing them in parallel. Taken to the extreme, this would involve creating a single incomplete $N \times M$ image for each projection, which is required for novel autoregressive formulations of the SAR imaging problem [16].

Copying the final pixel value from each thread to global memory can be skipped if we use zero-copy memory, where a portion of the host CPU's system memory is allocated for GPU use. Each thread would copy its pixel's complex value to the host memory across the PCI-Express bus directly. Although NVIDIA finds it difficult to predict which applications benefit from zero-copy memory, for small imaging scenarios, it may appreciably reduce runtime.

Finally, we note that for most of the discussion above, the context was research-based image formation. If the backprojection step of the SAR toolchain is relatively fast compared to other steps, then by Amdahl's law, GPUs may *not* find a place in end-to-end radar systems. The amenability of I-Q balancing and autofocus of raw phase history, as well as imaging to non-flat surfaces (with GIS height information, e.g.), may be considered for GPU acceleration.

References

[1] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," *Dr. Dobb's Journal*, vol. 30, March 2005.

[2] M. Hadwiger, C. Langer, H. Scharsach, and K. Buhler, "State of the art report on GPU-based segmentation," Tech. Rep. TR-VRVIS-2004-17, VRVis Research Center, Vienna, Austria, 2004.

[3] W. Wu and P. Heng, "A hybrid condensed finite element model with GPU acceleration for interactive 3d soft tissue cutting: Research articles," *Computer Animation and Virtual Worlds*, vol. 15, no. 3-4, pp. 219–227, 2004.

[4] S. Guha, S. Krisnan, and S. Venkatasubramanian, "Data visualization and mining using the gpu," in *Data Visualization and Mining Using the GPU, Tutorial at 11th ACM International Conference on Knowledge Discovery and Data Mining (KDD 2005)*, 2005.

[5] T. Hartley, A. Fasih, C. Berdanier, F. Ozguner, and U. Catalyurek, "Investigating the use of GPU-accelerated nodes for SAR image formation," in *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, pp. 1 –8, 31 2009-Sept. 4 2009.

[6] C. Jakowatz, D. Wahl, P. Eichel, and D. Ghiglia, *Spotlight-Mode Synthetic Aperture Radar: A Signal Processing Approach*. New York: Springer, 1996.

[7] M. Desai and W. Jenkins, "Convolution backprojection image reconstruction for spotlight mode synthetic aperture radar," *Image Processing, IEEE Transactions on*, vol. 1, pp. 505–517, Oct 1992.

[8] H. Scherl, B. Keck, M. Kowarschik, and J. Hornegger, "Fast GPU-based CT reconstruction using the Common Unified Device Architecture (CUDA)," in *Nuclear Science Symposium Conference Record, 2007. NSS '07. IEEE*, vol. 6, pp. 4464–4466, Oct.–Nov. 2007.

[9] P. B. Noël, A. M. Walczak, J. Xu, J. J. Corso, K. R. Hoffmann, and S. Schafer, "GPU-based cone beam computed tomography," *Computer Methods and Programs in Biomedicine*, vol. In Press, Corrected Proof, 2009.

[10] D. Kanter, "NVIDIA's GT200: Inside a parallel processor," *Real World Technologies*, 2008. http://www.realworldtech.com/page.cfm?ArticleID=RWT090808195242.

[11] NVIDIA Corporation, *CUDA Programming Guide, version 2.3.1*, August 2009.

[12] A. Klöckner, N. Pinto, Y. Lee, B. C. Catanzaro, P. Ivanov, and A. Fasih, "PyCUDA: GPU run-time code generation for high-performance computing," *Computing Research Repository*, vol. abs/0911.3456, 2009.

[13] J. Curtis H. Casteel, L. A. Gorham, M. J. Minardi, S. M. Scarborough, K. D. Naidu, and U. K. Majumder, "A challenge problem for 2D/3D imaging of targets from a volumetric data set in an urban environment," in *Algorithms for Synthetic Aperture Radar Imagery XIV* (E. G. Zelnio and F. D. Garber, eds.), vol. 6568, p. 65680D, SPIE, 2007.

[14] S. M. Scarborough, J. Curtis H. Casteel, L. Gorham, M. J. Minardi, U. K. Majumder, M. G. Judge, E. Zelnio, M. Bryant, H. Nichols, and D. Page, "A challenge problem for SAR-based GMTI in urban environments," in *Algorithms for Synthetic Aperture Radar Imagery XVI* (E. G. Zelnio and F. D. Garber, eds.), vol. 7337, p. 73370G, SPIE, 2009.

[15] T. Pipatsrisawat, A. Gacic, F. Franchetti, M. Puschel, and J. Moura, "Performance analysis of the filtered backprojection image reconstruction algorithms," in *Acoustics, Speech, and Signal Processing, 2005. Proceedings. (ICASSP '05). IEEE International Conference on*, vol. 5, pp. v/153 – v/156 Vol. 5, March 2005.

[16] R. L. Moses and J. N. Ash, "An autoregressive formulation for SAR backprojection imaging," *Aerospace and Electronic Systems, IEEE Transactions on*, submitted.