Techniques for Mapping Synthetic Aperture Radar Processing Algorithms to Multi-GPU Clusters

Eric Hayden, Mark Schmalz,
William Chapman, Sanjay Ranka, Sartaj Sahni
Department of CISE, University of Florida
Gainesville, FL 32611-6120, USA
{hayden, mssz, wchapman, ranka, sahni}@cise.ufl.edu

Gunasekaran Seetharaman
Information Directorate
Air Force Research Laboratory
Rome, NY 13441-4514, USA
Gunasekaran.Seetharaman@rl.af.mil

***Abstract* - This paper presents a design for parallel processing of synthetic aperture radar (SAR) data using multiple Graphics Processing Units (GPUs). Our approach supports real-time reconstruction of a two-dimensional image from a matrix of echo pulses and their response values. Key to runtime efficiency is a partitioning scheme that divides the output image into tiles and the input matrix into a collection of pulses associated with each tile. Each image tile and its associated pulse set are distributed to thread blocks across multiple GPUs, which support parallel computation with near-optimal I/O cost. The partial results are subsequently combined by a host CPU. Further efficiency is realized by the GPU's low-latency thread scheduling, which masks memory access latencies. Performance analysis quantifies runtime as a function of input/output parameters and number of GPUs. Experimental results were generated with 10 nVidia Tesla C2050 GPUs having maximum throughput of 972 Gflop/s. Our approach scales well for output (reconstructed) image sizes from 2,048 x 2,048 pixels to 8,192 x 8,192 pixels.**

***Index Terms*—High performance computing, Synthetic aperture radar, Image reconstruction, Graphics processing unit**

I. Introduction

The use of radio-frequency electromagnetic radiation for object detection in a synthetic aperture radar (SAR) configuration comprises a key technique for surveillance imaging [1,2]. In practice, an electromagnetic pulse from an emitter location $\mathbf{p}_P$ is reflected by a target, and the reflected pulse components are sensed at a receiver location $\mathbf{p}_R$ and quantized temporally into N_B bins. Given N_P pulses, where each pulse is taken at different sensor configurations of form $(\mathbf{p}_P, \mathbf{p}_R)$, an N_P x N_B element pulse data matrix P is obtained.

Computational transformation of P into an NxN-pixel image **b** that depicts targets within the sensor's field-of-view (FOV) requires superposition of pulse effects, implying an additive process where each pulse and its range bins in P potentially influences each pixel in **b**. The globality of this approach, which uses multiple pulses from different emitter locations, helps resolve ambiguities resulting from the fact that a pulse at given time delay references multiple points in the target plane that are equidistant from the emitter [3]. Advantageously, since the SAR reconstruction concept is similar in structure (but not identical mathematically) to a tensor product, one can adapt various distributed processing strategies developed for the tensor product [4] to implement SAR image reconstruction on a cluster of GPUs.

In this paper, we present techniques for parallelizing SAR image reconstruction algorithms to run efficiently on a CPU-controlled cluster of multiple graphics processing units (GPUs). Our technique is illustrated in terms of a single-stage backprojection algorithm whose computer code and data are available publicly [5,6]. Performance of our approach varies quasilinearly with N_{PR}, and does not appreciably compromise the numerical accuracy of the reconstructed image **b** with respect to a reference image **a**. Our technique employs algorithm-to-architecture mapping methods that feature a mix of distributed- and shared-memory models that reflect key GPU organizational constraints.

This paper is organized as follows. Section II provides theoretical and practical background. Section III presents the parallelized computer codes that comprise our approach. Timing and error measurements and analysis are given in Section IV, with conclusions and suggestions for future work given in Section V.

II. Theoretical and Practical Background

We begin with a discussion of the single-stage SAR backprojection algorithm (Section II.A) and then discuss previous work in parallelizing backprojection algorithms (Section II.B). An overview of the CPU and GPU architectures employed (Section II.C) provides practical background for the subsequent discussion of algorithm-to-architecture mapping in Section III.

A. Single-Stage SAR Backprojection Algorithm

Several published algorithms for SAR reconstruction from sensed radar pulse data have been optimized for sequential systems with relatively low throughput [2]. Here, algorithm design seems to be influenced by computational performance, rather than quality of the reconstructed image. In contrast, a naïve implementation of the single-stage backprojection algorithm (BP1) [5], is computationally costly but yields high numerical quality in the reconstructed image.

In practice, BP1 examines the pairing of every received (postprocessed) pulse with every reconstructed pixel to estimate object reflectivity at each point in the spatial representation. BP1 inputs pulse response matrix P, an N_P x N_B element matrix of N_P pulses and N_B response bins obtained by

temporally quantizing the received reflections of each radar pulse (Section I). For a given pulse, responses sensed later in time are physically further from emitter location $\mathbf{p}_P$, thusly are placed into higher-numbered range bins. For each pixel in the output image, every pulse and range bin in that pulse is considered separately. The value in a given range bin is added to the value of the associated pixel of the reconstructed SAR image. Brighter areas in the reconstructed image are associated with greater target reflectivity.

Implementationally, we observe that for matrices A and B, the tensor product $C = A \otimes B$ pairs each element a_{ij} of A with a sub-matrix $\boldsymbol{b}$ in B to form the partial product $c_{ij} = a_{ij} * \boldsymbol{b}$, where $(*)$ denotes pointwise multiplication. In practice, a host processor marshals products c_{ij} to form the matrix C.

We exploit this tensor product structure for SAR image reconstruction (albeit with different mathematical details) by associating each pulse and bin in P with each pixel of $\mathbf{b}$. In particular, let $P[i]$.bin[j] denote the j^{th} range bin of pulse i after phase correction, and let ($P[i].x$, $P[i].y$, $P[i].z$) denote the source of pulse i with respect to a prespecified origin. If each pulse has range [R_{start}, R_{end}], then the intensity of output image $\mathbf{b}$ corresponding to physical location (x,y) can be written as:

$$\mathbf{b}(x,y) = \sum_{i=1}^{P} P[i].bin\left[\frac{\sqrt{(x-P[i].x)^2+(y-P[i].y)^2+(P[i].z)^2}}{(R_{end}-R_{start})/\,|\,P[i].bin\,|}\right] \quad (1)$$

Note that this model captures the data movement patterns of BP1, although our implementation includes several enhancements that improve output quality. For instance, in Equation (1), it was assumed that $\mathbf{b}(x,y)$ maps to a range bin indexed by an integer. In practice, reconstructed pixels are associated with temporal intervals that can overlap range bins, and can be thought of as mapping to bins that have a fractional index. In such cases, an interpolated (smoother) image can be generated by examining range bins adjacent to a fractionally-indexed bin, then computing their weighted average based on their distance from this bin. However, such enhancements are not the focus of this paper; we view any implementation of the backprojection algorithm with a data access pattern as shown in Equation 1 as being logically equivalent. For additional details, the reader is referred to [2].

B. Related Work

The approach presented herein implements fast parallel processing of SAR data as well as reconstruction of a two-dimensional SAR image. Additionally, our techniques could be applied to a domain where sensor response data is projected back to form an image of a surface or object. For example, computer-aided tomography, featuring the construction of a 3-D model from a set of 2-D cross-sectional views, as in medical imaging for obtaining a three dimensional view of tissue *in vivo*. Here, each volumetric unit or *voxel* has properties similar to a reconstructed SAR pixel, and the tensor-product-like structure holds. Namely, each voxel in each cross-sectional view is associated with a response value that is spatially near the response of its neighboring voxel. Similarly, a given volume of output data will require a predictable quantity of sensor response data in each cross section. The preservation of these properties ensures that our partitioning scheme provides efficient data access mechanisms for generating tomograms [7,8].

Other potential applications differ from the preceding applications involving image sampling. For example, network tomography infers network characteristics from observations taken at known locations. In this case, each node in the network core is analogous to a pixel in the SAR output image, and each network location is conceptually similar to a radar pulse. Observe that the latency or reliability of a channel can be inferred from the repeated transfer of packets between locations. The projection of this data back onto the nodes through which packets have travelled can produce a visual representation of the network at any time, without requiring explicit assistance from core nodes. In this paradigm, spatial locality follows from the route optimization properties of routing protocols [3,9].

A similar problem involves estimation of ocean temperatures from acoustic wave propagation latencies, as the speed of sound in water varies inversely with water temperature. By measuring acoustic wave propagation times between multiple emitters and receivers, a set of cross sectional images can be constructed analogous to cross sections in computer-aided tomography. Small three dimensional units of water, not unlike the voxels of computer aided tomography, become the unit onto which acoustic sensor response data is projected. Spatial locality of input data follows from the output partitioning scheme [10].

C. Graphics Processing Unit (GPU)

GPU devices primarily support fast, photorealistic image rendering for video gaming applications. GPUs have recently been applied to supercomputing, due to fast SIMD processing and the ability of some GPUs to compute double-precision floating point arithmetic [11]. Additionally, GPUs have a huge installed base that amortizes significant research and development expenditures, thereby achieving relatively low unit cost (<100 USD) with frequent upgrades.

Unfortunately, GPUs feature a hierarchical memory structure that is understandably designed for fast graphics operations, but does not necessarily support all scientific computations. This particularly holds for image and signal processing with large operands, and for grid- and mesh-based simulation using huge model domains. Typical GPU multi-level memory structure combines a shared memory model (DRAM or global memory and L2 cache or shared memory in Fig. 1) in the context of groupings of *cores* (processing elements) called *multiprocessors* or *streaming multiprocessors* having local memory (L1 cache in Fig. 1). As a result of the varying latencies between DRAM, L2 and L1, as well as small local memories, data movement across a GPU is fraught with challenges that feature memory access latencies which can be quite large for apparently simple operations.

As a result, the porting of legacy code to GPUs has emphasized manual optimization, although recent reports indicate some automation is possible [12]. In practice, application-to-architecture mapping can be supported by an

underlying code-oriented architecture. For example, the Nvidia Tesla C2050 GPU used in this study has 14 streaming multiprocessors (SMs), each containing 32 processing elements or *cores* – totaling 448 cores per device. To coordinate many cores, GPUs arrange *threads* into *thread blocks*. In particular, each thread block embodies the execution of a single code fragment (or *kernel*) across multiple parallel threads that execute on a single SM. So, at least 14 thread blocks are needed to engage all 14 SMs in a C2050 GPU and each thread block must comprise at least 32 threads to engage the 32 cores in an SM. In practice, to effectively hide memory latency, may more thread blocks and many more threads per block are often needed.

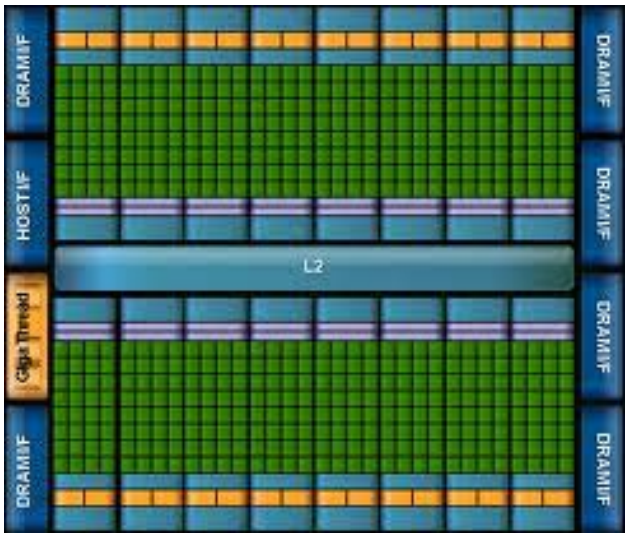

Fig. 1. High-level view of Nvidia Tesla C2050 GPU architecture.

During program design, a programmer specifies the parts of an application that are suited for threaded (parallel) execution, by labeling them as kernels using syntax specified by the GPU programming language (e.g., CUDA for an Nvidia Tesla processor). At runtime, the threads of a thread block can access a common shared memory, thereby supporting a form of inter-thread communication. A much slower global memory, accessible from all thread blocks, helps transfer data to and from the host as well as to and from the shared memory of the SMs [11]. This global memory also facilitates inter-block and inter-SM communication. As in other hierarchically-configured memory systems, high performance implies emphasizing local (intra-SM) memory operations, with minimal data transfer to and from slower memory devices.

GPUs can operate efficiently with thousands of threads running simultaneously, as thread scheduling is implemented directly in hardware with negligible overhead compared to program execution time. Support for many threads allows masking of memory access latencies: other threads can operate while paused threads wait for I/O operations to complete.

The BP1 algorithm is an ideal candidate for GPU implementation, since (a) each output pixel can be viewed as the sum of the contribution of all input pulses, and (b) the set of operations used to calculate this contribution is not dependent on the value of the input. However, some clever data partitioning is required to realize efficiency. Efficient single GPU implementations of BP1 were described by us in [3,13].

III. Multi GPU Algorithm Design and Implementation

The SAR backprojection algorithm (BP1) was implemented via partitioning of the pulse matrix P and reconstructed image **b**, as discussed in Section III.A. Code structure is discussed in Section III.B.

A. Algorithm-to-Architecture Mapping Technique

In order to restrict the pulse data in P to fit in GPU local memory, but not to be so small to cause excessive latency due to repetitive memory accesses, we employed a projection function f that relates the sensing geometry for a given response $P(i,j)$, specific to the i^{th} pulse and j^{th} range bin, to the corresponding pixel coordinate(s) in **b** [3]. Let P have N_P x N_B domain **Y**, and let **b** have N_x x N_y domain **X**. The *forward projection function* $f: \mathbf{Y} \rightarrow 2^{\mathbf{X}}$ forms a basis for derivation of a *reverse projection function* $g: 2^{\mathbf{X}} \rightarrow 2^{\mathbf{Y}}$. The function g is useful for our *BP1mg* multi GPU implementation, because it accepts the coordinates of a reconstruction neighborhood (also called a *tile*) denoted by $T \subset \mathbf{X}$, and produces the neighborhood $\mathcal{N}(\mathbf{Y})$ which denotes the coordinates of pulses corresponding to the restriction $\mathbf{b}|_T$. Due to sensing geometry at typical altitudes, we have $|\mathcal{N}(\mathbf{Y})| << |p_1(\mathbf{Y})|$, where $|S|$ denotes cardinality of set S, and p_k denotes projection to the k^{th} coordinate.

In particular, in this study we specified a KxK-pixel tile T to which we applied g, thereby supporting the following restriction of the pulse matrix:

$$P' = P|_{g(T)} \tag{2}$$

which was then assigned to a thread block for processing by a streaming multiprocessor to yield $\mathbf{b}|_T$. This partial result was sent to the CPU for accumulation into reconstructed image **b**.

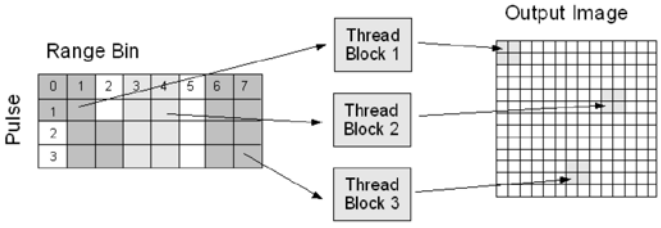

Fig. 2. Tile- and thread-block-based mapping technique for multi-GPU computation of algorithm BP1mg *shown in Figure 6* (from [3]).

Given this approach (see Fig. 2), efficiency is realized by (a) minimizing $|P'|$ without compromising reconstruction accuracy, (b) balancing K, N_x, and N_y to minimize I/O and memory latencies associated with the tiling of **b**, and (c) designing an efficient reverse projection g which can be done with (i) table lookup or (ii) explicit coordinate computations. The Nvidia Tesla C2050 GPU that was employed in this study, being optimized for graphics, is well suited to either i) or ii). In this paper, due to space limitations, we focus on

techniques a) and b). The reader is referred to [3,13] for a discussion of the technique in method c).

B. Code Examples

Parallelization of BP1, which was specified in Matlab™ , as overviewed in Fig. 3), was first expressed in C as outlined in Fig. 4, then in the CUDA language, as outlined at a high level in Fig. 5 (single-GPU version) and Fig. 6 (multi-GPU cluster version).

```
L.1  // Read all pulse data into memory
L.2  for each pulse in the Npulses dataset
L.3     // Perform Inverse FFT (ifft)
L.4     // Perform FFT Shift
L.5     for each pixel in the Nx x Ny output image
L.6        // Perform image reconstruction
L.7     end
L.8  end
L.9  // Write image to file
```

Fig. 3. Outline of Matlab code for BP1 backprojection algorithm.

```
L.1  For each set of pulses P' from the dataset P
L.2     // Read in a the next pulse set P' serially,
           using one thread
L.3     // Perform Inverse FFT and Shift operations on
           this set of pulses in parallel
L.4     #pragma parallel for each pixel in the Nx x Ny
           output image b
L.5        // Perform image reconstruction
L.6     end
L.7  end
```

Fig. 4. Outline of C code for BP1 parallel backprojection algorithm. Note the use of the MPI `#pragma parallel` *construct for the distributed memory model illustrated in Fig. 1.*

Given the code for *BP1mg* outlined in Fig. 6, which uses the mapping approach portrayed notionally in Fig. 2, we measured kernel runtime and numerical error associated with the serial and parallel BP1 codes, for a multicore CPU controlling up to 10 Nvidia Tesla C2050 GPUs, as follows.

```
L.1     // Read all pulse data into host CPU memory
L.2     // Write all pulse data onto GPU device memory
L.3     // Perform Inverse FFT for all Pulse data on the GPU
L.4     // Perform FFT Shift for all Pulse data on the GPU
L.5     Call kernel to perform reconstruction
L.6        for each pulse in the Npulses dataset
L.7           // Perform reconstruction of four pixels per
                 CUDA thread
L.8        end
L.9     end of kernel
L.10    // Read reconstructed image from GPU device
L.11    // Write image to file
```

Fig. 5. Outline of single GPU code for BP1sg backprojection algorithm.

IV. Experimental Results and Discussion

The multi-GPU version of backprojection algorithm BP1, hereafter called *BP1mg*, was compiled with the C/MPI compiler *gcc* version 4.1.2 Revision 20080704 (Red Hat 4.1.2-52) and the CUDA compiler Release 4.0, V0.2.1221, on five remote nodes of the Condor architecture [14], where each remote node has two Nvidia C2050 GPUs (see Fig. 2). One thread was used on each of six CPUs: one on the *localhost* (which had no GPU) and five on the remote nodes. The *localhost* and each remote node had an Intel Xeon 6-core CPU X5650 running at 2.67GHz with hyperthreading. The GPU code, *BP1mg*, is structured as shown in Fig. 6.

```
L.1  // Master thread sends arguments to slave threads (STs)
L.2  // Master thread reads data file and broadcasts it to STs
L.3  // Each slave thread is assigned a subset P' of the total
        number of pulses for each device.
L.4     // Each device performs Inverse FFT and Shift on P'
L.5     // Each device performs image reconstruction given
           the pulse subset P'
L.6        for each set P' of pulses from the dataset P
L.7           for each pixel assigned to thread in the Nx x Ny
                 construct image tile b|T
L.8                 // Perform reconstruction on the tile
L.9           end
L.10       end
L.11  // Each slave thread merges the reconstructed image
        given the reconstructed tiles from each device
L.12  // Master thread merges the reconstructed tiles from
        each slave thread to yield reconstructed image b
L.13  // Master Thread writes the reconstructed image to file
```

Fig. 6. Outline of multi-GPU code BP1mg for backprojection algorithm.

A hand-optimized version of *BP1mg* was tested using the Tesla C2050's double precision floating point arithmetic. The input data (pulse data, antenna locations, etc.) as well as the reconstructed image are single precision floating point; all operations in the reconstruction kernel are performed in double precision. Pulses before (respectively, after) the Inverse Fast Fourier Transform (IFFT) are 5,592 pulses (resp. 16,384 pulses).

A. Runtime Measurement and Analysis

Timing data for *BP1sg* and *BP1mg* are presented in Tables I (total runtime) and II (kernel runtime), with the same parameters as for the reference single-GPU algorithm *BP1sg* similar to that described in [3]. Total execution time (sec, including I/O, message passing, memory allocation and

TABLE I. Total Runtime in Seconds for Single- and Multi-GPU Versions of the BP1 Backprojection Algorithm.

Runtime	Algorithm Version		
N_x x N_y	BP1sg	BP1mg/*max*	Speedup
2048x2048	15.357	2.494	6.158 x
4096x4096	48.080	6.468	7.434 x
8192x8192	178.806	22.039	8.113 x

transfer, and IFFT/Shift) for double-precision GPU cluster version *BP1mg* (labeled "Multi") of the single-stage backprojection algorithm (N_{pulses} = 5,004). Speedup is computed from the maximum runtime for all cases of *BP1mg*, with respect to the single-GPU version *BP1sg*. Timing data for total run time are summarized graphically in Fig. 7.

As shown in Table II, the kernel speedup for image sizes ranging from 4Kx4K to 8Kx8K effectively equals the expected theoretical value of 10x. Slight differences in speedup (e.g., 10.019x instead of 10x) result from systematic errors due to limited temporal resolution of the *tic…toc* timer mechanism.

TABLE II. KERNEL RUNTIME IN SECONDS FOR SINGLE- AND MULTI-GPU VERSIONS OF THE BP1 BACKPROJECTION ALGORITHM.

Runtime	Version / Number of GPUs					
N_x x N_y	BP1sg	BP1mg/1	BP1mg/2	BP1mg/3	BP1mg/4	BP1mg/5
2048x2048	11.39	1.133	1.132	1.132	1.231	1.232
4096x4096	44.05	4.333	4.384	4.387	4.383	4.415
8192x8192	173.79	17.051	17.247	17.254	17.255	17.246

Runtime	Version / Number of GPUs (cont'd)					
N_x x N_y	BP1mg/6	BP1mg/7	BP1mg/8	BP1mg/9	BP1mg/10	Speedup
2048x2048	1.198	1.200	1.201	1.199	1.152	9.243x
4096x4096	4.384	4.386	4.387	4.399	4.415	9.978x
8192x8192	17.346	17.334	17.334	17.335	17.346	10.019x

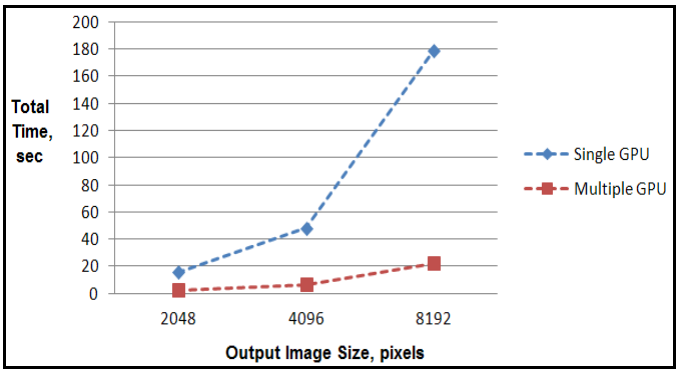

Fig. 7. Total Execution Time, BP1mg algorithm running in double precision floating point arithmetic on one to five Condor nodes with MPI.

B. Reconstruction Error Measurement and Analysis

In the following discussion, we refer to *average error* and *maximum error*, which are computed as follows. Assume that a reference image **a** is an N_x x N_y pixel array, which is used as a basis for comparing the reconstructed image **b**, also an N_x x N_y pixel array, by computing a difference image **d** as:

$$\mathbf{d}(i,j) = \mathbf{c}(i,j) - \mathbf{a}(i,j),\ 1 \le i \le N_x \text{ and } 1 \le j \le N_y\ . \qquad (3)$$

The maximum error and average error are defined from **d**, as follows:

$$e_{max} = \bigvee_{i,j} \mathbf{d}(i,j) \,/\, range \qquad (4)$$

and

$$e_{ave} = \Sigma_{i,j}\, \mathbf{d}(i,j) \,/\, (range \cdot N_x \cdot N_y)\ , \qquad (5)$$

where *range* denotes the range of pixel values in **b**.

In tests using Equations 3-5, we found that the average and maximum output errors were identical to the output error for the double-precision single-GPU algorithm *BP1sg* outlined in Fig. 5. This is reasonable, since *BP1mg* is mathematically the same as *BP1mg*, but runs on multiple GPUs instead of a single GPU device.

We have also found that measured error figures depend on the version of the Tesla C2050 architecture (e.g., SM10 or SM20), as well as the trigonometric functions employed. In particular, *sincosf* and *_sincosf* refer to a combined trigonometric function that computes in single precision, while *sincos* refers to a combined trigonometric function that computes in double precision.

For GPU version SM10 operating in single precision mode, the maximum error (relative to double precision version SM20 using *sincos*) for the 4Kx4K output image was 4.42 / 255 (or 2.8 percent), and 5.45 / 255 (or 3.45 percent) for the 8Kx8K image. However, for both images the average error was the same (0.0511 / 255 or 0.03 percent). We have found that the maximum error and average error were unaffected by the precision of the trigonometric functions.

In the case of GPU version SM20 operating in single precision mode, the maximum and average errors were similarly unaffected by the precision of trigonometric functions. For the 4Kx4K image, the max error was 2.32 / 255 or 1.65 percent, while for the 8Kx8K image, the maximum error was 2.60 / 255 or 1.65 percent. The average error was 0.046 / 255 or 0.03 percent for both image sizes. While the maximum error was approximately half that obtained from SM10 single precision computation, the average error was reduced by only 10 percent.

For GPU version SM10 operating in double precision mode, *double* datatypes are demoted to *float* datatypes. Although trigonometric precision does not affect the maximum and average errors, these are not the same as when SM10 single precision is used – but they are quite close. When GPU version SM20 is run in double precision mode, *sincos* had zero error, because all errors were measured with respect to this run. *Sincosf* and *__sincosf* produced maximum error of 0.00302 (effectively zero percent) for the 4Kx4K reconstructed image, and 0.00339 (effectively zero percent) for the 8Kx8K image. The average error was 0.0000585 and 0.0000584 (as before, effectively zero percent).

In the case of GPU version SM10 operating in hybrid precision mode, the maximum and average errors were the same as for SM10 operating in double precision mode. For GPU Version SM20 operating in hybrid precision mode, the maximum and average errors were the same for *sincosf*, as well as for the case when these trigonometric functions run in SM20 double precision mode. However, when *sincos* was used in SM20 hybrid mode, the maximum and average errors were approximately 10^{-4} and 10^{-7}, respectively, compared to the "gold standard" SM20 double precision mode with *sincos*. As noted previously, SM10 in double precision mode produced the same errors as SM20 double precision mode.

V. Conclusions

Synthetic aperture radar image reconstruction via backprojection is an instance of a class of problems that are based structurally on the tensor product. With some clever manipulation, backprojection algorithms can be adapted to a parallelization strategy based on tiling of the output data structure (in this case, a reconstructed SAR image). However, one must also have a projection function that associates a computationally useful subset of the input data structure (in this case a SAR pulse array) with each output tile.

In this paper, we present a tiling algorithm for SAR image reconstruction from thousands of pulses or views. This algorithms, called *BP1*, is adapted for implementation on a single GPU (algorithm *BP1sg*) and a multi-GPU cluster (*BP1mg*) controlled by a multi-core CPU. As theory predicts, execution time scales inversely with the number of GPU slave nodes, reaching a speedup of 10x for 10 Nvidia Tesla C2050 GPUs on the Condor architecture. Average image reconstruction error is within $\pm$0.05 percent of grayscale range, and is negligible in a large portion of the reconstruction scenarios.

We also present ideas for future work, in which the image reconstruction algorithm presented herein could be applied to problems in computed tomography, fluid dynamics, and other imaging or simulation scenarios based on multiple views.

Acknowledgment

The authors thank the Air Force Research Laboratory for support for this research under Contract #FA8750-11-C-0182.

References

[1] Mita D. Desai and W. Kenneth Jenkins, "Convolution Backprojection Image Reconstruction for Spotlight Mode Synthetic Aperture Radar" IEEE Transactions on Image Processing, Vol. 1 No. 4, pp. 505-517, 1992.

[2] Lars M. H. Ulander, Hans Hellsten, Gunnar Stenstrom, "Synthetic-aperture radar processing using fast factorized back-projection", *IEEE Transactions on Aerospace and Electronic Systems*, Vol. 39 No. 3, pp. 760-776, 2003.

[3] Chapman, W., S. Ranka, S. Sahni, M. Schmalz, U. Majumder, L. Moore, and B. Elton. "Parallel processing techniques for the processing of synthetic aperture radar data on GPUs", *Proceedings of the IEEE International Symposium on Signal Processing and Information Technology* (2011).

[4] Yu, J.F., H.-C. Hsiao, Y.-J. Kao, "GPU accelerated tensor contractions in the plaquette renormalization scheme", *Computers & Fluids*, Vol. 45, pp. 55–58 (2011).

[5] Gorham, L.A. and L.J. Moore, "SAR image formation toolbox for MATLAB," in *Proceedings of the SPIE Conference on Algorithms for Synthetic Aperture Radar Imagery XVII* **7669** (2010).

[6] Casteel, C.H. Jr, L.A. Gorham, M.J. Minardi, S.M. Scarborough, K.D. Naidu, and U. Majumder. "A challenge problem for 2D/3D imaging of targets from a volumetric data set in an urban environment", *Proceedings of SPIE* Vol. **6568**, pp. 65680D-1 (2007).

[7] Gac, N., S. Mancini, M. Desvignes and D. Houzet. "High speed 3D tomography on CPU, GPU and FPGA", *EURASIP Journal on Embedded Systems* - Special issue on design and architectures for signal and image processing, Vol. 2008, Article 5 (2008).

[8] Ledley, R.S. "Introduction to computerized tomography", *Comput. Biol. Med.*, Vol. 6, pp. 239–246 (1976).

[9] Vardi Y, "Network tomography: Estimating source-destination traffic intensities from link data". *Journal of the American Statistical Association* 91: 365-377 (1996).

[10] Brekhovskikh, L. *Fundamentals of Ocean Acoustics*. Third Edition, Springer Verlag (2003).

[11] "NVIDIA Programming Guide - Version 2.2." April 2009.

[12] Bueno, J., L. Martinell, A. Duran, M. Farreras, X. Martorell, R.M. Badia, E. Ayguadé, and J. Labarta: "Productive cluster programming with OmpSs". *Proceedings Euro-Par* **12**:555-566 (2011).

[13] W. Chapman, S. Ranka, S. Sahni, M. Schmalz, and U. Majumder, "Parallel processing techniques for the processing of synthetic aperture radar data on FPGAs", *Proc. IEEE International Symposium on Signal Processing and Information Technology* (2010).

[14] Luley, R., C. Usmail, and M. Barnell. "Energy efficiency evaluation and benchmarking of AFRL's Condor high performance computer", in *Proceedings of the Department of Defense High Performance Computing Modernization Program's 2011 User Group Conference*, Portland, OR (2011).