Investigating the Use of GPU-Accelerated Nodes for SAR Image Formation*

Timothy D. R. Hartley[1,2], Ahmed R. Fasih[2], Charles A. Berdanier[3],
Füsun Özgüner[2], Umit V. Catalyurek[1,2]

[1] Department of Biomedical Informatics,
[2] Department of Electrical and Computer Engineering,
The Ohio State University, Columbus, OH, USA.
[3] Air Force Research Laboratory,
Wright-Patterson Air Force Base, OH 45433
{hartleyt,umit}@bmi.osu.edu,
{fasiha,ozguner}@ece.osu.edu, charles.berdanier@wpafb.af.mil

Abstract

The computation of an electromagnetic reflectivity image from a set of radar returns is a computationally intensive process. Therefore, the use of high performance computing is required to form images from radar signals in a short time frame. This paper explores the use of distributed memory cluster computers and accelerator technologies such as GPUs for radar signal analysis applications, particularly backprojection image formation. We obtain good results with the use of GPUs and compare their performance in terms of execution time with distributed memory cluster computers. When using a configuration with 4 GPU-accelerated nodes, we achieve speedups up to 3.45x for different input and output data size combinations.

1 Introduction

Due to the rapid growth of the computational capacity of Graphics Processing Units (GPUs) over the past decade, researchers are increasingly using these emerging architectures to accelerate high performance applications. In fields such as data mining [7], image segmentation and clustering [8], numerical methods for finite element computations used in 3D interactive simulations [16], nuclear, gas dispersion and heat shimmering simulations [17], and biomedical imaging [9], GPUs have been used to speed up operations which are time-consuming on standard processors, dramatically affecting the overall execution times of the final applications.

Synthetic Aperture Radar (SAR) is a computationally intensive technique which can be used for, among other things, creating 2-D and 3-D images from radar signals gathered by a moving platform such as an aircraft. By combining signals gathered from multiple points in space (multiple angles of azimuth and elevation), higher resolution images can be constructed without needing a larger physical antenna or radar array. The computational burden increases with the image size and the amount of input, and so techniques for accelerating the processing of the input radar signals and generating the output images are necessary to be able to process the large amount of data in real time. Even if real time processing is not the goal, the sheer volume of data which SAR platforms can gather necessitates fast processing to enable fast decision making.

This paper investigates the use of a cluster of GPU-equipped processing nodes to perform SAR image formation by backprojection. We discuss the particulars of the backprojection algorithm and briefly present an overview of computed tomography in Section 2. We present the software technologies used to implement the algorithm on the target system in Section 3, and the algorithm design space and parallelization decisions in Section 4. We finish by presenting our experimental results in Section 5 and give some conclusions and future work afterwards.

2 Overview of Computed Tomography

In this section, we provide an introduction to tomographic imaging, the principle behind x-ray computer-

*This work was supported in part by the U.S. Department of Energy SciDAC Grant DE-FC02-06ER2775; by the U.S. National Science Foundation under Grants CNS-0643969, and CNS-0403342; by Ohio Supercomputing Center Grant PAS0052; by AFRL/DAGSI Ohio Student-Faculty Research Fellowship RY6-OSU-08-3.

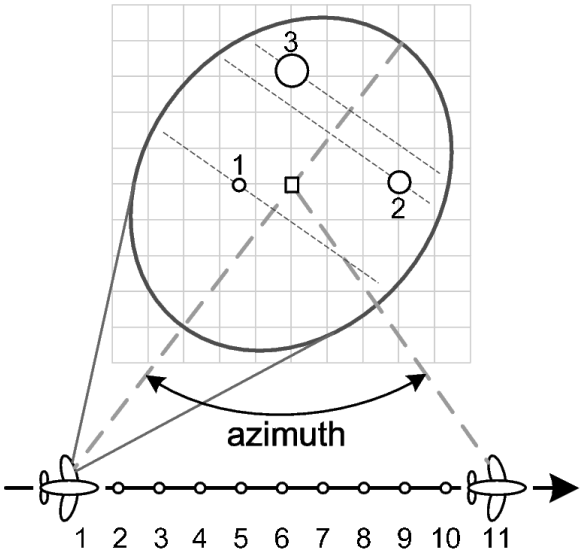

Figure 1. Schematic demonstrating operation of the tomographic principle. The scene consists of the three targets of different amplitudes (circles), and produces the range profiles shown in Figure 2. Note that the flight-path may be circular or straight.

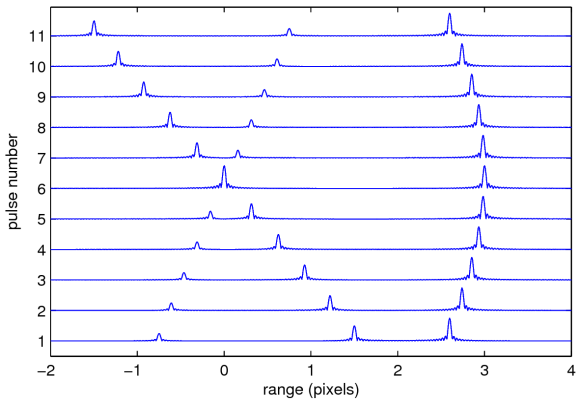

Figure 2. Line projections obtained by a sensor flying the large aperture of Figure 1. Each range profile contains the linear contributions of all three scatterers. Note the merging and crossing of the two left-most scatterers at pulse 6 due to them possessing identical range displacement.

aided tomography (CAT), magnetic resonance imaging (MRI), and synthetic aperture radar (SAR) imaging. A tomographic system involves a sensor capable of taking one-dimensional line projections through a two- or three-dimensional scene, and then reconstructing this underlying scene from a collection of line projections taken from different aspect angles.

The mathematics of line projections are given by the Radon transform. The specific way that this transform enters each of these modalities is slightly different, due to differences in their respective sensors, but the common element is this: the two-dimensional scene is collapsed into a one-dimensional projection by means of a dense set of line integrals penetrating the scene (in three dimensions, the projection is obtained by a set of slice integrals). This line integral is sampled and stored as a one-dimensional data vector, and tagged with the location of the sensor when the projection was taken.

Figure 1 presents a diagram of a SAR antenna, which is mounted on an aircraft and pointed at a scene of interest on the ground. The antenna broadcasts a very short radio pulse (lasting microseconds) at the scene and records any reflections. It is assumed that there is negligible aircraft motion during this process; the aircraft moves and the process is repeated at a pulse repetition frequency of several thousands per second. With a single pulse, the scene cannot be reconstructed: although the one-dimensional range profile gives good range resolution, cross-range resolution is non-existent because two reflectors equidistant from the sensor would be indistinguishable. However, by combining many one-dimensional pulse returns collected over a large azimuth extent, multi-dimensional reconstruction of the scene becomes feasible. Azimuth functions as the second dimension, variation along which suffices for a two-dimensional reconstruction. Height or elevation angle diversity, is required for three-dimensional reconstruction. Complete details of SAR reconstruction are provided in [11].

Mathematically, it can be shown that a multi-dimensional Fourier transform of a continuous function is equivalent to the one-dimensional Fourier transform of that function's Radon transform (along each projection). The discrete version of this relationship is used by a large class of tomographic reconstruction algorithms which use a fast Fourier transform (FFT). Because most FFT implementations require a rectangular grid, whereas projections are usually collected along a radial grid, polar-to-rectangular interpolation is an important pre-processing step. Such polar-formatting algorithms are attractive because the central step of multi-dimensional FFT is $\mathcal{O}(N^n \log N)$, where n is the dimensionality of the scene.

Another class of of popular algorithms is filtered backprojection (also known as convolved backprojection) which, put simply, reverses the action of the Radon transform. An image is initialized to zero; then for each projection (shown in Figure 2), every pixel that may have contributed to an element of the sampled projection vector is incremented by that element. For any given sample of a pro-

jection vector, the pixels that could have contributed to its value when the line integral was taken correspond to those pixels that are equidistant from the radar at the given azimuth and elevation angles. Because each pixel must query a single point along each projection, backprojection has $\mathcal{O}(N^{n+1})$ complexity, where n again is the dimensionality of the scene. It is called "filtered" backprojection because each projection is given a frequency weighting to adjust for a larger number of pixels clustering around the center of the image, due to a radial acquisition mode. This clustering is also visible in Figure 1: whether the aircraft flies past a scene or circles it, the radially-sampled range profiles obtained sample the center of the scene more finely than the edges.

In theory, both algorithms produce equivalent outputs, and frequently, commercial tomographic reconstruction systems are locked into one or the other. Experts on both sides have compiled lengthy lists of pros and cons, for various modalities, imaging scenarios, and scene sizes over successive generations of computer capabilities, because the image outputs of the two algorithms do differ. We sidestep the controversy by noting that while both algorithms have been demonstrated to scale well to distributed systems, backprojection very easily allows an image to be formed on a previously-obtained digital elevation map (DEM). For this major reason, we have chosen to implement backprojection for a GPU cluster processing environment.

SAR differs from CAT or MRI reconstruction in that the underlying scene being Radon-transformed is a complex-valued electromagnetic reflectivity function, encoding attenuation and absorption properties of the materials, rather than a real-valued x-ray absorption function or hydrogen energy release map. What this practically means is that with coherent SAR processing, which backprojection accomplishes, a modern system can produce fully legible image with less than 5° of azimuth. (MRI and CAT systems typically need 180° to reconstruct a two-dimensional slice.)

For mapping or surveillance applications, SAR is valuable because it has range-independent resolution, operates day and night, and is to a large degree impervious to weather conditions. Many deployments successfully deploy it with camera or LIDAR sensors to accomplish many varied tasks, but perhaps the most common is forming an image.

Having described the computational complexity of the backprojection algorithm, we next describe our choice of software engineering frameworks to write parallel implementations.

3 Software Support

This section describes the software tools and libraries used during the development of our radar signal analysis application. To parallelize the computation across multiple nodes, we have chosen DataCutter, a component-based programming framework and runtime engine. With DataCutter, we are able to easily make use of multicore processors and accelerators, decompose the input and output domains, and efficiently execute the overall application in and end-to-end fashion. To program the GPU, we have chosen to use Nvidia's CUDA programming environment and hardware solution. With CUDA, we are able to efficiently make use of the GPU's vast computational throughput as well as integrate seamlessly with DataCutter for parallelization across a full GPU cluster. Below we present a brief discussion of each of these software solutions.

3.1 DataCutter

DataCutter [3] is a component-based middleware framework [1, 4, 10, 13, 14] designed to support coarse-grain dataflow execution on heterogeneous computational resources. In DataCutter, the application processing structure is implemented as a set of components, referred to as *filters*, that exchange data through *logical streams*. A *stream* denotes a uni-directional data flow from one filter (i.e., the producer) to another (i.e., the consumer). Data flows along these *streams* in *buffers* so as to minimize various system overheads. A *layout* is a filter ontology which describes the set of application tasks, streams, and the connections required for the computation. A *placement* is one instance of a *layout* with actual filter copy to physical processor mappings.

As a dataflow system, naturally, DataCutter supports *pipeline-parallelism* (also called as *task-parallelism*) through concurrent execution of dependent components (filters) on different data items. The DataCutter runtime system also supports data-parallelism at multiple levels. At the application level, multiple copies of the application layout can be instantiated and executed. At the filter level, multiple copies of filters can be either transparently or explicitly created and executed, while providing the illusion of a single filter to all upstream and downstream filters. Processing, network, and data copying overheads can be minimized by intelligent placement scheduling of filters. The runtime engine performs all steps necessary to instantiate filters on the desired machines and cores, to connect all logical endpoints, and to call the filter's interface functions for processing work.

3.2 CUDA

The Compute Unified Device Architecture (CUDA) [5] is a programming interface and set of supported hardware to enable general purpose computation on Nvidia GPUs. The programming interface is ANSI C extended by several keywords and constructs which translate into a set of

C language library functions; a special compiler generates the executable code for the GPU.

In CUDA compatible hardware, the GPU cores are organized into multiple multi-processors (16 in G80), each having a large set of registers (8,192 in G80), a small shared memory (16 KB) very close to registers in speed (both 32 bits wide), and constants and texture caches of a few kilobytes. The GPU also has a large DRAM which is divided into three types: *constant memory*, *texture memory* and *global memory*.

The CUDA programming model simply consists of a collection of threads running in parallel. Programs are decomposed into *blocks* of threads, arranged logically in a regular grid pattern. Each block is mapped to a single multiprocessor, and multi-processors can run (potentially) more than one block of threads concurrently. The local resources (registers and shared memory) are divided among blocks, hence threads. A *warp* is the largest set of threads the GPU as a whole can execute concurrently; at 32 threads on the G80, a warp size is less than the number of 128 cores due to memory access limitations. In any given cycle, each core in a multi-processor executes the same instruction of a warp on different data for each thread (based on a unique `threadId`). Communication between multi-processors is performed through global memory on the GPU. No inter-block communication or specific schedule-ordering mechanism for blocks or threads is provided, which allows each thread block to run on any multi-processor at any time.

Since CUDA is specifically designed for generic computing, it does not suffer from excessive constraints when accessing memory, although the memory access times do vary for different memory types. All of the threads can access all of the GPU memory, but, as expected, there is a performance boost when each thread reads data resident in a special shared memory area (up to two orders of magnitude).

When developing applications for GPUs with CUDA, the management of resources (registers, shared memory) becomes important as a limiting factor for the amount of parallelism we can exploit. Each multi-processor of the G80 hardware (which we have used for our experiments) contains 8,192 registers which will be split evenly among all the threads of the blocks assigned to that multi-processor. Hence, the number of registers needed in the computation will directly affect the number of threads able to be executed simultaneously.

4 Implementation Details

Having given a high-level overview of tomographic reconstruction in Section 2 and having presented the software solutions we will leverage to accelerate image reconstruction, we delve into the backprojection algorithm in further detail.

We first present this caveat: before being used for imaging, each one-dimensional projection vector must be pre-processed. This involves windowing (to adjust the mainlobe-sidelobe tradeoff) and filtered for frequency deweighting. The former step involves element-wise multiplication of each projection vector by the window of choice. As SAR data is typically sampled and stored in the Fourier domain, filtering involves multiplying each projection by a filter frequency response and inverse Fourier bringing each projection to the spatial domain (via an inverse FFT with $\mathcal{O}(N \log N)$ complexity). As mentioned above, the filter in question adjusts for the fact that projections have a coordinate origin, and are thus more densely sampled in some areas than others, due to the radial nature of acquisition. The most common filter, the Ram-Lak filter, is simply a ramp filter with a frequency response equal to $|f|$, for frequency f.

In medical imaging, the computational burden of preprocessing has to be accounted for, and it can frequently be a critical factor (e.g., in the Cell Broadband Engine implementation of [15]). Traditionally, too much data is sampled by a SAR system to be either processed on board the aircraft or wirelessly broadcast to a ground station, making imaging non-real-time. For this reason, we chose to not parallelize the one-time pre-processing: we simply perform windowing and filtering on a central CPU, and off-load the actual backprojection to the acceleration system. We will discuss the advantages and trade offs of each acceleration method both here and in Section 5 where we present our experimental results.

4.1 Backprojection with DataCutter

In backprojection, each one-dimensional projection has a contribution for each given pixel. A number of algebraic operations have to be performed to obtain the index for each projection; then, that element of the projection vector must be fetched and incremented to the pixel. Therefore, one may assign subsets of projections to cluster nodes and partition the input, or one may assign sub-image tiles of the output image to cluster nodes, and partition the output.

Figure 3 shows the basic processing pipeline. The major tasks are *Read Input Data*, *Form Partial Image*, and *Aggregate Partial Images*. For processing pipelines with no task parallelism, the second stage will consist of only one processing element, and as such the partial image formed during this stage will in fact be the final image. However, in more complex processing pipelines (specific instances of tasks mapped to CPUs, GPUs or Cell processors) with task parallelism, the work to be performed is partitioned amongst all of the *Form Partial Image* pipeline tasks.

With SAR image formation, the work to be performed is

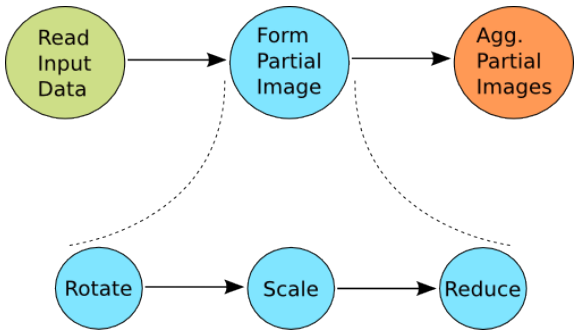

Figure 3. SAR Imaging Pipeline

the calculation of each input vector's contribution to each pixel in the output image. As such, both the input and the output can be partitioned amongst the imaging pipeline tasks. Figures 4 and 5 show small examples of partitioning the input and output of copied imaging pipeline tasks, respectively. When partitioning the input, the amount of computation which each imaging task performs is reduced to C/N where N is the number of imaging tasks and C is the total amount of computation required to form the total final image. When the input is partitioned, the output data size stays constant among all of the imaging tasks.

When partitioning the output in SAR image formation, the input data set size stays constant (it is broadcasted by the Read task to all of the downstream imaging tasks). Then, each imaging task only computes the input data set's contribution to a subset of the pixels of the output image.

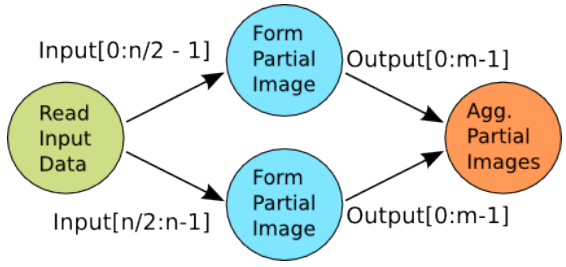

Figure 4. SAR Imaging Input Partitioning

Although more complex hybrid solutions to the decomposition of the input and output data can be developed [12], their performance is typically close to that of the output-partitioned scheme when the output size is large. Also since the input-partitioned and output-partitioned parallelization schemes represent the extremes of the SAR image formation pipeline design space and constitutes a good base cases for comparisons, in this work we only develop and present these two parallelization schemes.

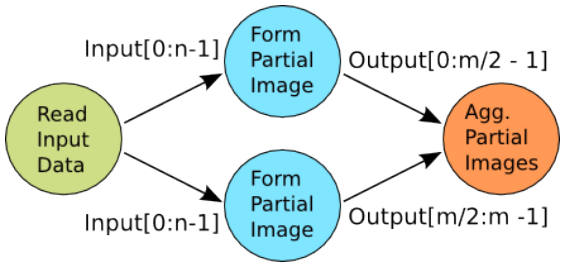

Figure 5. SAR Imaging Output Partitioning

4.2 Backprojection with GPU

Our CUDA backprojection implementation partitions the output image subset assigned to the kernel; this could be the whole range of output pixels or just a subset, depending on the output partitioning conducted across the image formation tasks. During the remainder of this section, we will assume the simple case where the pipeline includes only one imaging task. CUDA blocks correspond to rectangular sub-images, and CUDA threads correspond to individual pixels. This partitioning is appealing because we can take advantage of texture caching if we store the projections as a texture. Another advantage of using texture memory is the hardware for linear interpolation of textures [15]. This is beneficial because interpolation is required to choose an intermediate value between two samples of a given projection during the image formation, and the GPU hardware provides this for free.

As the cached-texture memory is read-only, it cannot be used for storing portions of the output image. Therefore, individual CUDA blocks allocate small image tiles in shared memory, and each member thread increments its assigned tile pixel by independently computing the index of each projection. After all the projections have been queried and the sub-image completed, it is copied back to global memory in a fully-coalesced write operation.

4.3 Combining DataCutter and CUDA

Since DataCutter is a component-based programming framework, it is ideally suited to leveraging other programming frameworks to ease implementation within components. Further, DataCutter allows encapsulation all of the low-level details of making use of accelerator architectures such as GPUs. Provided the interfaces exposed to the other components stay consistent, the implementations of each of the components are quite flexible.

Our combined DataCutter/CUDA backprojection algorithm used DataCutter's ability to return pointers to specific portions of data buffers which are the quantum of data the runtime engine handles. These pointers are then simply passed to a function which transfers data to the GPU

and executes the CUDA kernel. By preallocating outgoing data buffers, we are able to transfer the results of the GPU computation directly to the outgoing data buffer, rather than requiring an extra copy operation.

5 Application Experiments

This section details the experiments we conducted to investigate the performance of using GPU clusters for SAR image formation. Following the description of the computer hardware, we discuss the dataset we used for our experiments. Then, a presentation of our results and a discussion of the interesting points follows.

All of the CPU and GPU experiments were conducted on the Ohio Supercomputer Center's (OSC) BALE Visualization GPU cluster [2]. The BALE GPU nodes consist of two dual-core 2.6 GHz AMD Opteron processors, 8 GB of main memory, Nvidia Quadro 5600 graphics cards with 1.5 GB of memory, and Infiniband network cards.

For our experiments, we used four nodes, which provides 16 cores and four GPUs. The CPUs have a peak performance of 17.6 GFLOPS in single-precision arithmetic, while the GPUs have a peak performance of 330 GFLOPS. We restrict our discussion of hardware specifications to single-precision floating-point arithmetic, because our application's implementations only make use of single-precision data types.

The Air Force Research Lab's Sensor Data Management System (SDMS) web site has released certain data sets to the public, one of which is the GOTCHA[1] data set. The GOTCHA data set consists of SAR phase history data collected with a 640 MHz bandwidth. We use a single elevation angle and up to 11° of azimuth coverage. The imaged area is that of a parking lot, and is populated with various cars and a few construction vehicles.

In our first set of experiments, we tested the scalability of our basic DataCutter-based parallelization scheme, and compared it with an existing, simple C/MPI parallel implementation of the backprojection algorithm [6]. Figure 6 shows the CPU-only results, where we show the scalability of these two implementations while varying the number of processes. As the figures show, our DataCutter implementation is as efficient at using the parallel machines as the straight MPI version, and as such, will introduce no unwanted overheads when transitioning to the parallel GPU implementation. Indeed, due to the streaming nature of DataCutter, and the staggered start times of the processing nodes receiving messages from the initial Read filter, the aggregation filter does not act as a bottleneck in the application, leading to better scalability as the number of nodes increases.

[1] https://www.sdms.afrl.af.mil/datasets/gotcha/index.php

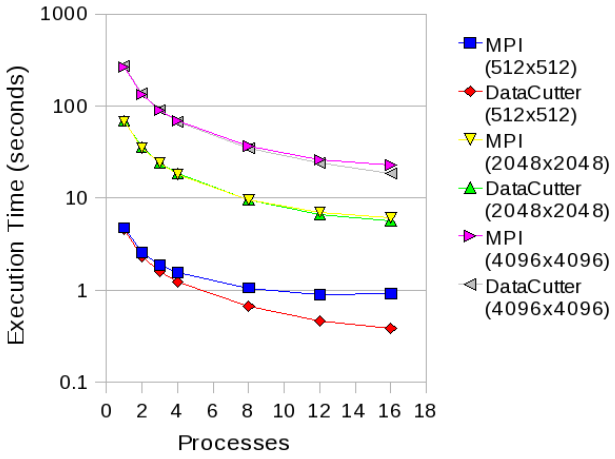

Figure 6. Execution times of C/MPI and DataCutter backprojection implementations with 1° of input data and varying image sizes.

Our next set of experiments present the performance gains that can be achieved by GPU parallelization of backprojection code. Figure 7 shows the execution times of a single GPU running the backprojection algorithm on 1° of azimuth data with two different implementations. The figure shows that DataCutter introduces a slight overhead to single GPU executions, which is to be expected for a pipelined parallel code written under the assumption either input or output data will be partitioned. Necessary additional steps (the aggregation of output sub-images for the output partitioning case, for instance) need to be executed even when only one GPU will perform the image formation. Also note the exponential growth in execution times when the output image size is increased from 2048x2048 to 4096x4096. This calls for a multi-node/multi-GPU parallelism, especially for larger image sizes.

The comparison between parallel CPU and GPU implementations is striking in Figure 8; as expected, the GPU implementations are significantly faster than CPU implementations. For example, running the same 1° of azimuth data using C/MPI code takes about 4.7 seconds for 512x512 image size, whereas it only takes 0.15 seconds using the GPU, hence resulting in just over 31x speedup. The performance gap increases with increasing image size; for 2048x2024 image size, the GPU's speedup over CPU is about 55x, while for 4096x4096 images, the speedup climbs to about 58x.

Figure 9 shows the largest-scale multi-GPU results. Here, we use 11° of data in order to highlight the issues we can solve within the multi-GPU domain. The results show that the use of additional GPUs can help to further reduce the execution time. Using 4 GPUs, we achieved up

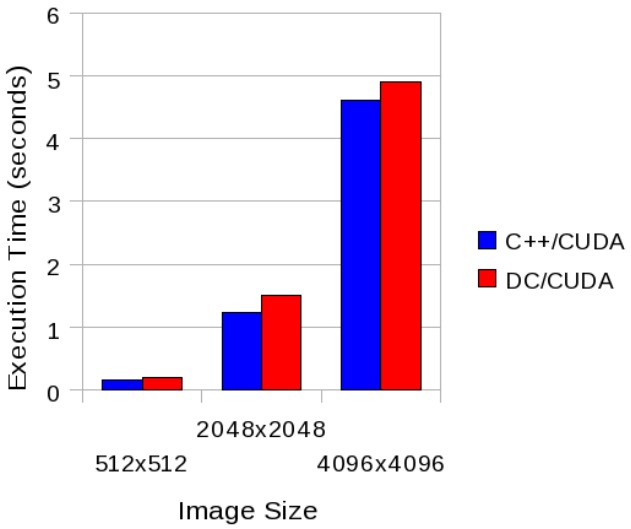

Figure 7. Execution times of single GPU implementations with 1° of input data.

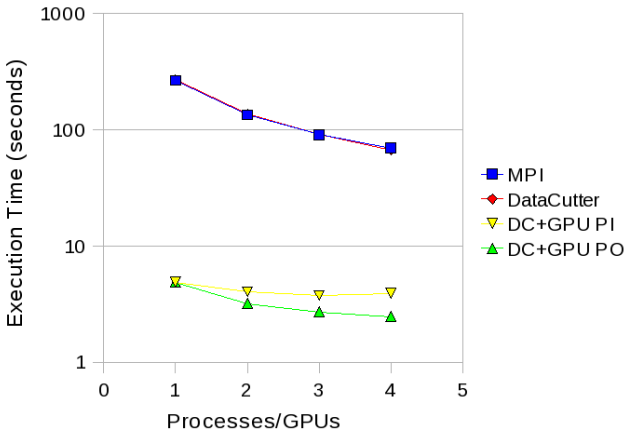

Figure 8. Execution times of CPU and GPU implementations with 1° of input data. The data series suffices 'PI' and 'PO' denote those implementations where the input or output is partitioned amongst the GPUs, respectively.

3.45 speedup with this particular combination of input and output sizes. Also note that especially on the larger image size, the output-partitioned parallelization scheme has slightly better performance. For example, on 4096x4096 image size we achieve 3.05 speedup using input partitioning and 3.45 speedup using output partitioning. This is undoubtedly due to the fact that when the output image is partitioned, the GPU need only calculate a subset of the output image, and must only copy that subset from the GPU back to the host's memory. Whereas, when the input is partitioned, the whole image needs to be transfered between host and GPU. This is known to be a slow operation, due to the relatively anemic bandwidth of the PCI Express bus, to which GPUs are connected.

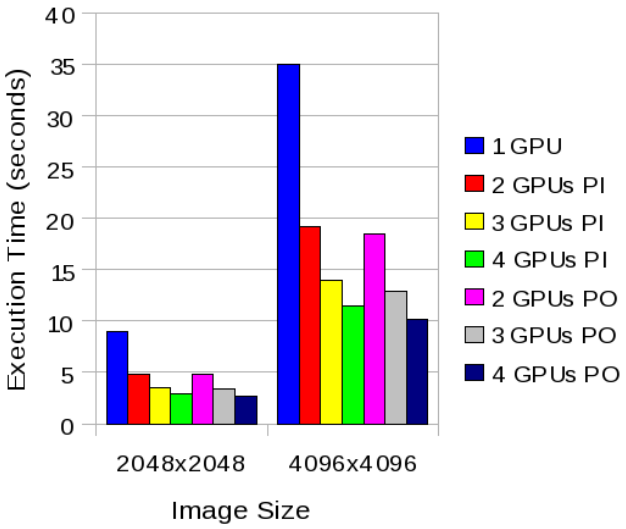

Figure 9. Execution times of DataCutter/GPU implementation running on up to 4 GPUs with 11° of input data. The data series suffices 'PI' and 'PO' denote those implementations where the input or output is partitioned amongst the GPUs, respectively.

Our last set of experiments, depicted in Figure 10, shows the effect of varying the number of degrees of input azimuth data while using 4 GPUs. Since the number of projections in each azimuth degree is roughly the same, there is a linear increase in processing time for each degree of input data added to the image formation. As expected, the slopes of the lines are different for each output image size, because for each input projection, the amount of computation is highly dependent on the number of pixels in the output image.

6 Conclusions

In this paper we have presented a method for performing 2-D image formation from SAR on a cluster of GPUs. By using DataCutter for the internode parallelization and CUDA for the GPU programming, we have shown that our solution is efficient at making use of the computational resources. Further, by making use of 4 GPU-equipped processing nodes to perform the 2-D backprojec-

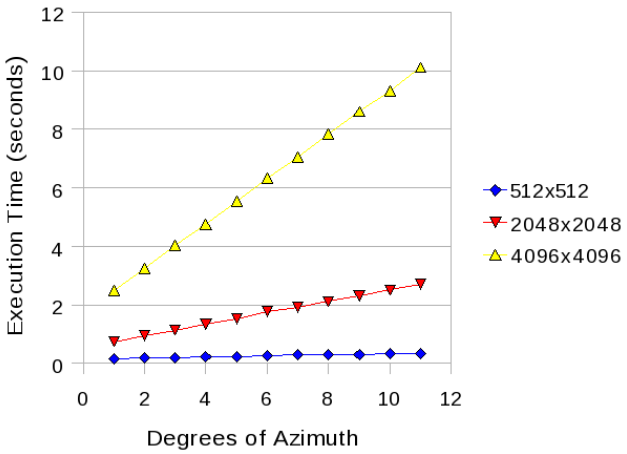

Figure 10. Execution times of DataCutter/GPU implementation running on 4 GPUs while varying the input data set size.

tion computation, we can get (versus a single CPU core executing a relatively simple backprojection implementation) 29.9x speedup on a 512x512 image, 92.1x speedup on a 2048x2048 image, and 109.9x speedup on a 4096x4096 image.

Our future work will include extending our use of accelerator architectures to the Cell Broadband Engine. Through the use of DataCutter for coarse-grain parallelization, the addition of other accelerators for the computationally demanding portions of the application is very straightforward. Additionally, we intend to investigate the use of DataCutter to perform the preprocessing steps for SAR image formation, leading to a full end-to-end runtime system for backprojection. Further improvements to the GPU backprojection implementation include the use of some of the newer CUDA features such as asynchronous memory copying and zero-copy memory access. This will improve the overlap of computation and communication, allowing further scalability.

References

[1] K. Amiri, D. Petrou, G. R. Ganger, and G. A. Gibson. Dynamic function placement for data-intensive cluster computing. In *the USENIX Annual Technical Conference*, San Diego, CA, June 2000.

[2] The BALE cluster at the ohio supercomputer center. http://www.osc.edu/supercomputing/hardware.

[3] M. D. Beynon, T. Kurc, U. Catalyurek, C. Chang, A. Sussman, and J. Saltz. Distributed processing of very large datasets with DataCutter. *Parallel Computing*, 27(11):1457–1478, Oct. 2001.

[4] Common Component Architecture Forum. *http://www.cca-forum.org*.

[5] CUDA. Home page maintained by Nvidia. http://developer.nvidia.com/object/cuda.html.

[6] L. A. Gorham, U. K. Majumder, P. Buxa, M. J. Backues, and A. C. Lindgren. Implementation and analysis of a fast backprojection algorithm. In *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, volume 6237 of *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, June 2006.

[7] 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.

[8] M. Hadwiger, C. Langer, H. Scharsach, and K. Buhler. State of the art report on gpu-based segmentation. Technical Report TR-VRVIS-2004-17, VRVis Research Center, Vienna, Austria, 2004.

[9] T. D. R. Hartley, Ü. V. Çatalyürek, A. Ruiz, F. Igual, R. Mayo, and M. Ujaldon. Biomedical image analysis on a cooperative cluster of GPUs and multicores. In *Proceedings of the 22nd Annual International Conference on Supercomputing, ICS 2008*, pages 15–25, 2008.

[10] C. Isert and K. Schwan. ACDS: Adapting computational data streams for high performance. In *14th International Parallel & Distributed Processing Symposium (IPDPS 2000)*, pages 641–646, Cancun, Mexico, May 2000.

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

[12] T. Kurc, F. Lee, G. Agrawal, U. Catalyurek, R. Ferreira, and J. Saltz. Optimizing reduction computations in a distributed environment. In *ACM/IEEE SC2003*, Phoenix, AZ, November 2003.

[13] R. Oldfield and D. Kotz. Armada: A parallel file system for computational grids. In *Proceedings of CCGrid2001: IEEE International Symposium on Cluster Computing and the Grid*, Brisbane, Australia, May 2001. IEEE Computer Society Press.

[14] B. Plale and K. Schwan. dQUOB: Managing large data flows using dynamic embedded queries. In *IEEE International High Performance Distributed Computing (HPDC)*, August 2000.

[15] H. Scherl, B. Keck, M. Kowarschik, and J. Hornegger. Fast GPU-based CT reconstruction using the Common Unified Device Architecture (CUDA). *Nuclear Science Symposium Conference Record, 2007. NSS '07. IEEE*, 6:4464–4466, 26 2007-Nov. 3 2007.

[16] 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*, 15(3-4):219–227, 2004.

[17] Zhao, Y., Y. Han, Z. Fan, F. Qiu, Y. Kuo, Kaufman, and K. A., Mueller. Visual simulation of heat shimmering and mirage. *IEEE Trans. on Visualization and Computer Graphics*, 13(1):179–189, 2007.