# Exploring Novel Parallelization Technologies for 3-D Imaging Applications

Diego Rivera Dana Schaa Micha Moffie
David Kaeli
Department of Electrical and Computer Engineering
Northeastern University, Boston, MA*

## Abstract

*Multi-dimensional imaging techniques involve the processing of high resolution images commonly used in medical, civil and remote-sensing applications. A barrier commonly encountered in this class of applications is the time required to carry out repetitive operations on large matrices. Partitioning these large datasets can help improve performance, and lends the data to more efficient parallel execution.*

*In this paper we describe our experience exploring two novel parallelization technologies: 1) a graphical processor unit (GPU)-based approach which utilizes 128 cores on a single GPU accelerator card, and 2) a middleware approach for semi-automatic parallelization on a cluster of multiple multi-core processors. We investigate these two platforms and describe their strengths and limitations. In addition, we provide some guidance to the programmer on which platform to use when porting multi-dimensional imaging applications. Using a 3-D application taken from a clinical image reconstruction algorithm, we demonstrate the degree of speedup we can obtain from these two approaches.*

## 1. Introduction

There has been considerable effort devoted to developing techniques that allow us to better visualize the 2-D and 3-D structure of objects by integrating multiple views and processing high resolution images. We find these applications in medical, civil, and remote-sensing areas. One issue with utilizing large multi-dimensional datasets is the computational effort required to process the volume of data.

In this paper we focus on a subset of these applications: those that employ iterative algorithms and are solved by executing repetitive operations on large matrices. Since these applications typically execute the same tasks on very large data sets, they offer many opportunities to parallelize the execution. Some common examples of this class of computation include matrix inversions and 3D image reconstruction using multiple 2D projections. While most of these applications are CPU-bound, then are not *embarassingly parallel*. Most computations possess few data dependencies, and can be effectively processed in parallel. To accelerate this class of workload, we are exploring two recently developed technologies: a general purpose GPU-based approach using a unified architecture (CUDA) and an MPI-based semi-automated parallel Matlab framework (Star-P).

Unified architecture GPUs, such as NVIDIA's CUDA (Compute Unified Device Architecture), make use of the immense resources available on modern GPU's and allow developers to access the GPU through extensions to the C language and dedicated APIs. The CUDA toolkit enables developers to execute general purpose code on GPUs. This can also be accomplished using graphics APIs (e.g., OpenGL), but they are more difficult to use for general purpose programming since the abstractions presented (e.g., shading) are intended for graphics applications.

Star-P is a semi-automated parallel computing platform which allows a client machine to transparently connect to a high-performance parallel server and offload the computationally-intensive portions of the workload. Much of the parallelization is performed automatically using high-level language programming directives.

Our contributions in this paper are two-fold. First we explore the strengths and limitations of these technologies, and present a set of guidelines to determine whether code can be ported easily and parallelized effectively with these technologies. Second, we apply our guidelines to real code and measure the effectiveness of each approach. We select a medical image reconstruction application used in Tomosynthesis mammography. We also relate our experience applying these methodologies to a Cardiac-CT algorithm using CUDA and a hyperspectral imaging toolbox using Star-P. We are not trying to compare these two technologies di-

*This project is supported by the National Science Foundations Computing and Communication Foundations Division, grant number CCF-0342555 and the Institute of Complex Scientific Software. This work is also supported by an NSF STTR Phase I Award No. 0638034.

rectly since they each have their advantages. GPUs provide a high performance solution while Star-P provides performance for significantly less programming effort.

This paper is organized as follows. In Section 2 we review related work. In Section 3, we discuss the CUDA and Star-P technologies. Section 4 presents our guide and compares these approaches based on application characteristics. In Section 5 we present our implementation and results, and we summarize and propose future work in Section 6.

## 2. Related Work

In the past decade, GPUs have become commonplace for processing many medical, civil, and remote-sensing imaging applications. These imaging applications include many tomographic reconstruction algorithms such as filtered backprojection, and algebraic and expectation maximization methods [4, 14, 13]. Previously, these applications have been programmed using graphics programming interfaces such as OpenGL and DirectX. Implementing these general-purpose applications to run on GPUs was not trivial, and was heavily reliant on the development of high-level specialized programming languages and tools [2, 8, 12]. In contrast, the *unified-architecture* implementations do not require us to use a graphics programming interface. In this paper, we present the first unified-architecture implementations of two 3-D imaging applications.

Similarly, we explore a novel parallel implementation of Matlab. A large effort has been undertaken by academia and industry to produce parallel implementations of Matlab [3]. These efforts allow the programmer to exploit parallelism without having to consider low-level communication and synchronization details. MathWorks, the creators of Matlab, have produced a Distributed Computing Toolkit for parallelization, which is similar to the MatlabMPI created by Kepner at MIT Lincoln Labs [6]. Rice University is developing a parallel implementation of Matlab, MatlabD, with the goal of automatically compiling Matlab scripts into high performance Fortran [5]. One downside of many parallel implementations is that the data they use is limited to the memory size of a single CPU memory. In contrast, Star-P can load and store data directly from the distributed memories in a cluster, allowing for larger data set sizes. It also contains features such as semi-automated data distribution, and task parallelism support. Because Star-P is such a versatile tool, we selected it to serve as an interesting parallelization technology to contrast against GPU-based parallelization.

## 3 Description of Technologies

Next, we introduce the two novel parallelization technologies explored in the paper, describe their operation and interfaces, and compare them to existing technologies.

### 3.1 A GPU Unified Architecture Approach

The introduction of the DirectX 10 Unified Shader specification has laid a foundation for an important change in GPU architectures. Previously, GPU architectures have included two different types of processing units, pixel and vertex.

As mentioned in section 2, several middleware platforms have been developed to execute general propose applications on GPUs to exploit the pixel and vertex units on board. However, in the new specification, the two individual sets of instructions for pixel and vertex have been unified. CUDA, the new middleware from NVIDIA Corporation, allows programmers to use the C programming language to develop program that target this unified-architecture GPU.

In this work we use the recently introduced GeForce 8800 GTX GPU from NVIDIA [9]. The chipset contains 128 stream processors, each with three scalar units (integer, FP32 MAD, and FP32 MUL). The processors are organized in groups of 16, making a total of 8 16-way multiprocessors; each multiprocessor shares 16 KB memory and 8192 32-bit registers. The chipset includes 768 MB of global (device) memory shared among multiprocessors. The GPU off-chip memory bandwidth is equal to 86.4 GB/s.

Based on a Host(CPU)/Device(GPU) scheme, the CUDA programming model allows the execution of multiple threads on the 8800's 8 multiprocessors. Threads executing the same computational kernel are grouped into a grid of thread blocks. A block is processed by only one multiprocessor. At any time, there is one grid assigned to the GPU with one or several blocks per multiprocessor. The number of blocks on a multiprocessor depends on the thread's complexity which is determined during compilation. Threads being executing concurrently are split into groups of 32 which are executed in Single Instruction Multiple Data (SIMD) fashion. This allows the overlap of memory latency with useful computation, however it introduces a limitation related to the control flow. To avoid performance penalties due to idle threads, conditional statements should be limited to at most 7 instructions. If there are less than 7 instructions, conditional statement bodies are executed using predication and stalls are avoided.

There are several physical and functional limitations related to threads and computation kernels [10]. The maximum number of threads per multiprocessor is 768 and in a block is 512. In addition, the maximum number of blocks in a grid is $65535^3$. There is neither support for kernel recursion nor interthread communication. Consequently, to avoid correctness issues, threads must execute independently of one another.

### 3.2 A Semi-Automatic Parallel Matlab Approach

MPI is widely used as a parallel programming middleware because of its efficiency. However, it requires the programmer to delve into the parallelization details, including the initial environment setup, and the communication and synchronization of processes during execution. MPI is a practical option for experienced programmers, but is not desirable for scientists who prefer Matlab's flexible, easy-to-use development and test environment. There have been several approaches to parallelize Matlab [3]. Star-P, an MPI-middleware, preserves the familiarity of Matlab by hiding the details of parallelization with a semi-automatic approach running on top of Matlab-licensed clients.

One of the main advantages of Star-P is the automatic preparation of the communication framework, and the handling of data placement and transfers. Obviously, operations that do not require communication or data transmission will run fastest, but Star-P will also automatically handle operations that require data to be moved between processor memories at different times during execution. If the same were attempted with a lower-level framework such as MPI, data movement would have to be coded explicitly. Star-P inherently handles code that either contains independent data operations, operations that require complex communication, or a mixture of both.

In Star-P, data partitioning is implemented by distributing large datasets across the multiple processor memories of a clustered system. Appending $*p$ to an argument in a Matlab constructor (such as `ones`, `rand`, etc.) creates the data and partitions it along that dimension. In cases where one constructor argument creates multi-dimensional data, the data is partitioned along the highest dimension. These partitions are then stored on the distributed memories of the system. When operations are performed on the distributed data, each processor operates on its own subset. Operations on the distributed data then create data that is also distributed. Star-P also provides functions to save and load distributed data that do require the data to be sent back to the local machine [11].

Distributing data with Star-P also changes the data type. Dense matrices become $ddense$ objects, multi-dimensional matrices become $ddensend$ objects, and sparse matrices become $dsparse$ objects. In order for these distributed objects to work transparently with Matlab, Star-P overloads functions for each data type. These functions can be called exactly the same way with distributed objects as with regular objects, requiring no modification of the code.

In addition to supporting data parallelism, Star-P provides a construct called $ppeval$ for exploiting task-level parallelism. $ppeval$ operates by executing task parallel code for iterations of loops on separate processors. It also allows the user to split matrix dimensions across iterations to decrease the amount of data that must be transmitted to each processor. In Star-P terminology, $split$ refers to the partitioning of data on a dimension and transmitting one partition for each iteration. Similarly, $broadcast$ refers to transmitting an entire data structure to all processors. To use $ppeval$, the code or loop body must be coded as a function, with its input parameters specified as either $split$ or $broadcast$[11].

Since Star-P is built using MPI, it is difficult to achieve the same speedup obtained when using hand-tuned MPI code. However, Star-P takes advantage of the processors and memories of a clustered system, while requiring much less development effort. Just as with any other parallel implementation, the performance benefits from Star-P vary based on the characteristics of the serial code.

## 4. Application Acceleration Guides

In this section we provide guides to determine whether code can be ported and parallelized easily and effectively with both CUDA and Star-P. When providing our guidelines, we consider different aspects of the applications intended to be parallelized. These aspects include the degree of parallelization, how many parallel threads can be extracted, how much work is allocated for each thread, the amount of inter-process communication or thread synchronization, the memory usage requirements, and the complexity of each thread.

For these guides, we assume that a reference serial implementation of the code exists. The serial implementation may be in C, Fortran, or any other language in the case of using a GPU. For Star-P, we assume an existing Matlab program exists. It is not necessary that the serial implementation be optimized. However, it is important that the structure of the code does not hinder the extraction of parallelism.

### 4.1 GPGPU-CUDA Guide

In order to achieve scalable performance with CUDA, high utilization of all the stream processors is required. This can be achieved by creating a very large pool of threads; unfortunately all threads running on the same multiprocessor also share its resources. Requirements related to memory, registers, and communication among threads must be considered to determine if CUDA can be used effectively to accelerate an application.

#### 4.1.1 Registers Required Per Thread

The number of threads that can be executed in parallel on the multiprocessor is limited by the number of registers

available. This can become a problem when threads are complex - as is often the case for multi-dimensional imaging applications[1]. It is possible to reduce the number of registers to improve performance by making simple threads, reusing registers, and possibly using on-chip shared memory as an alternative to registers.

Using shared memory to increase the number of active threads per processor is possible by forcing CUDA to place variables in shared memory instead of registers. However, over-reliance on this method can have a negative impact on performance because of shared memory latency. At some point, which is application-dependent, a trade-off exists between running more threads simultaneously, and running fewer threads more efficiently. Thus, a GPU's high utilization in terms of the number of active threads does not necessarily mean less execution time.

Both reusing registers and using shared memory have the drawback that they require explicit hard coding that makes the code less portable and dependent on the problem size.

#### 4.1.2 Communication and Synchronization

CUDA provides limited support for synchronization between threads. This support only allows a thread to synchronize its execution with the rest of the threads in the same block, which ensures the completeness of different procedures in a computation kernel. For example, it is most efficient for all threads to load data, synchronize, and *then* operate on the data.

A GPU may not be the best choice if the application being parallelized requires intercommunication between threads. In order to avoid correctness problems, threads must execute entirely independently of each other and commit their results to dedicated memory locations. If it is necessary to communicate data between threads, the host CPU can be used for this purpose. However, this will require blocking the execution of all the threads, communicating data back to the CPU, and resuming the threads' execution. Clearly, even if this method can be implemented for a specific application, it will be an expensive solution due to the high cost associated with transferring data between the GPU and the CPU.

#### 4.1.3 Memory Limitations

The performance of an application can be limited if it requires more memory than is available on the device. If the application's dataset can be partitioned to work on different data subsets during execution then the subsets can be 'manually' swapped between the GPU memory and main memory using the CPU. To do this, the threads must finish execution on the current data subset, memory must be copied to or from the GPU, and another set of threads must be launched with the new subset. Another solution is to use the processing power of the CPU to compute the results that do not fit in GPU memory. This introduces a trade-off between communication speed and computation speed.

[1]Thread complexity increases if inner loops contain variable-length iterations or if sparse matrices are used.

### 4.2 Star-P Guide

Star-P provides several options to exploit different kinds of parallelism. The following sections help determine which approach should be taken based on data size, execution time, opportunities for vectorization, and code content.

#### 4.2.1 Distributed Memory

Star-P's approach to managing distributed data assumes that the memory available to the programmer includes the combined memories of all the processors in the cluster. This pooling of memory resources makes Star-P a viable choice for code that requires operations on very large data sets, on operations that may require a large amount of intermediate storage, or on data sets of indeterminate size. In addition, data can be loaded directly to distributed memory so that we do not have to worry about the implicit limitations of a single processor memory.

#### 4.2.2 Task Parallelism

The *ppeval* function distributes independent loop iterations to each processor so that they execute in parallel. *ppeval* is not appropriate in situations where a loop body's execution time cannot amortize the overhead of data transmission and process creation. Equation 1 explicitly expresses the factors that determine the execution time of *ppeval* on each processor.

$$T = t_{bcast} + I(t_{split} + t_{process_creation} + t_{body}) \quad (1)$$

In Equation 1, $T$ is the time required for *ppeval* to complete the overall execution of the tasks assigned to a processor, $t_{bcast}$ is the time to transmit the broadcast data, $I$ is the number of iterations executed on the processor, $t_{split}$ is the time to transmit the split data, $t_{body}$ is the execution time of the function body during each iteration, and $t_{process_creation}$ is the overhead time required to create a process on a processor. As the equation shows, the broadcast data only needs to be transmitted once to each processor, and then can be reused for all iterations.

It is important to note that $T$ may not be the same for all processors. For example, if there are 33 iterations being

split across 32 processors, one processor will have to execute a second iteration while the other 31 sit idle. Therefore, the total execution time of $ppeval$ is equal to the longest execution time of any one processor. It is only effective to use $ppeval$ when the longest execution time is less than the serial execution time. If this is not the case, then the only other option is to try to extract as much data parallelism as possible by distributing the data and operating on it in parallel.

In addition, since we focus on how Star-P parallelizes Matlab programs in this work, highly nested loops will be difficult to work with (as they are in Matlab). Whenever possible, inner loops should be vectorized. Since the nesting of $ppeval$ statements is not allowed, it is only possible to remove one degree of nesting using task parallelism in Star-P. However, with the expanded memory allowance permitted by distributing data, the programmer may be able to restructure code so that data structures can be duplicated, potentially allowing inner loops to be removed altogether.

#### 4.2.3 Compatibility with Matlab

Although Star-P is designed to run transparently on Matlab, there are some Matlab functions that are not currently supported for distributed data types (though all of the most commonly used functions are). If distributed data needs to be operated on by an unsupported function, it can easily be recombined on the local machine. However, frequently transmitting large data sets in this fashion will impact performance.

### 4.3 Comparing the architectures based on application characteristics

We consider the GPU to have a large porting effort mainly because of the many considerations that need to be taken into account to achieve good performance. Although CUDA allows us to easily port C code and provides high level constructs to do so, in order to achieve high performance we need to understand all the limiting factors described above (i.e., number of threads, register allocation, etc.). The GPU's high core density of processing should translate to high performance, but there are a number of limiting factors that make the actual performance highly application-dependent.

Star-P, on the other hand, provides a very simple parallelization interface. But application performance is limited by the degree of data parallelism inherent in the operations in the code, the amount of communication required by an operation, and the computing resources available in the cluster. Code that runs well on Matlab and lends itself well to distributed memory or task parallel operations should run fast with Star-P.

Table 1 shows a summary of some important application characteristics to consider when choosing either of these parallel frameworks.

| | GPU | Star-P |
|---|---|---|
| Number of Threads | High number of threads (thousands) | Not important if execution time amortizes data transfers |
| Thread complexity | Simple: no or few control statements | Complex or simple |
| Synchronization | Only between threads on same multiprocessor | Automated |
| Memory | Limited device memory (768MB on GeForce 8800) | Sum of all processor memories |

**Table 1. Desired algorithm characteristics by technology.**

## 5. Implementations and Results

Two parallel clustered systems were used to execute MPI and Star-P code:

Cluster A: 33 Servers, each one contains two 2.0 GHz Intel Xeon dual core processors and 8/16GB of RAM. Servers are interconnected by Gigabit Ethernet. Compiler GCC version 3.4.5(Red Hat 3.4.5-2), optimization level O3.

Cluster B: 65 Servers, each one contains two 3.2 GHz Intel Xeon processors and 4 GB of RAM. Servers are interconnected by Gigabit Ethernet. Compiler GCC version 3.2.3(Red Hat Linux 3.2.3-42), optimization level O3.

Serial versions of the applications were run on a 1.86GHz Intel Core2 processor with 3GB RAM.

### 5.1 Tomosynthesis Mammography

To demonstrate the potential speedup of these approaches, we used a challenging 3-D reconstruction application. Digital Tomosynthesis Mammography is a technique that helps to improve the early detection of breast cancer by integrating multiple views and processing high resolution projection images [15]. Even though this technique has been shown to be highly effective, it is a time-consuming process due to the amount of processing necessary to reconstruct a single 3-D image.

A total of 15 x-ray digital mammograms, or 2D views, are acquired by moving an x-ray source, which hits a high-definition detector (1900 x 2304 pixels). This data is processed using an iterative image reconstruction algorithm to recreate a 3D structure of the breast, which is used in the detection and diagnosis of tumorous tissue. The image reconstruction algorithm consists of two phases per iteration:

a forward projection and a backward projection. Given an initial estimated 3D volume of an object, the forward phase simulates the x-ray traveling through the object, estimating projection images. The backward phase takes each value in the 3D volume and corrects it based on the difference between the estimated projection and the actual x-ray projections. The complexity of the algorithm, as well as the volume of the input and output data, increase linearly with the size of the detector and the number of projection (usually between 50 and 100). Parallelization for this application can be exploited by partitioning the projected images and the 3D volumes into multiple segments. These data subsets can be processed in parallel on a clustered system.

Another alternative to expose parallelism is by processing views in parallel during the forward phase, and projections in parallel during the backward phase. Although we cannot create a full vectorization of the problem due to data dependencies in the algorithm, it is possible to expose parallelism in each segment, projection, or view by creating multiple threads to execute some of the inner loops.

#### 5.1.1 Tomosynthesis on CUDA

The guidance provided in Section 4.1 was applied to evaluate the use of shared memory, registers, and communication among threads. Our implementation of Tomosynthesis using the GPU exploits the parallelism exposed in views and projections. The dimensions of the detector and the number of projections make it is possible to have a significant number of threads running (2,755,584 threads, based on number of pixels per view) during the forward and backward phases. The large number of active threads is able to hide the memory latencies associated with the use of device memory. Consequently, the use of shared memory was not required.

After determining the dataset size, including the input, output, and temporary structures, we noted that the largest data structure common to both phases could be maintained in device memory throughout the entire program execution. The other structures had to be swapped during execution. Keeping this large structure in device memory throughout the execution provided the best performance, because it eliminates much of the data transfer time and helps to avoid saturating the GPU off-chip memory bandwidth.

Figure 1 shows performance results for Tomosynthesis on CUDA along with the MPI code run on the two clusters. We achieved an $11X$ speedup for Tomosynthesis using a GPU-based approach - a speedup comparable to our 32-node clusters (3X slower than Cluster A, 1.5X faster than Cluster B). Figure 2 shows a cost-performance relationship based on the execution time of the application and the market value[2] of the evaluation environments for various numbers of iterations. Larger values in the graph represent a higher cost of running the algorithm.

[2]Market value is based on current price in dollars for each server in Clusters A and B. This value does not include cost of installation, interconnection, or maintenance. Market value for the GPU includes the price of the GPU and workstation

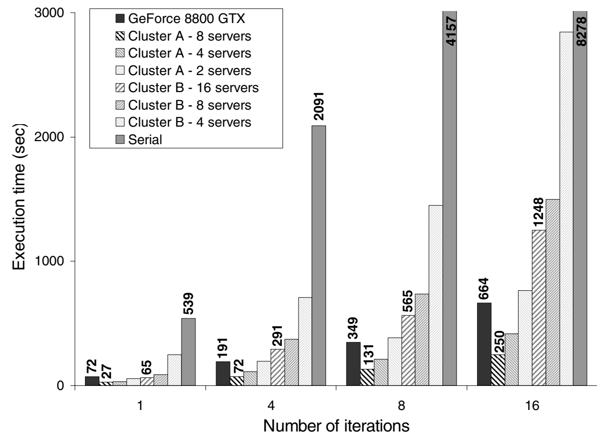


**Figure 1. Tomosynthesis reconstruction, performance**

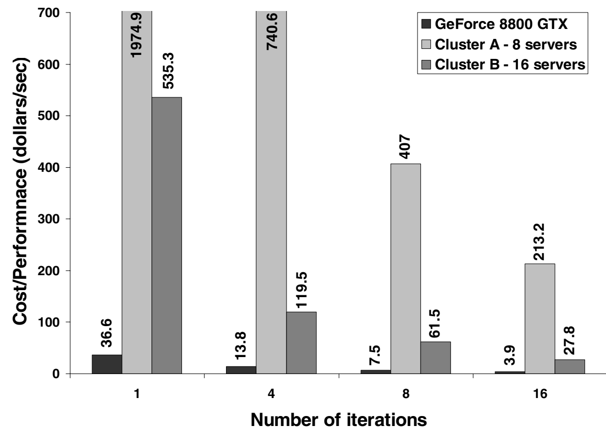


**Figure 2. Cost/performance for Tomosynthesis reconstruction on our 3 platforms.**

#### 5.1.2 Tomosynthesis on Star-P

Running Tomosynthesis on Star-P does not require as many memory considerations. Instead, the interesting consider-

ations are related to extracting parallelism efficiently from the algorithm. In Tomosynthesis, both forward and backward projection in the serial algorithm have four nested $for$ loops, which are devastating to performance in Matlab. Computing the forward projection, only the inner-most loop can be vectorized. The number of iterations in the innermost loop varies per pixel, which inhibits vectorization across the three outermost loops. Our options for parallelizing the algorithm are to distribute the data and operate on it in parallel, or apply $ppeval$ to one of the three outer loops.

Our guidelines can be applied to this problem in a number of ways. First, we determine if we can rule out any of these methods based on the algorithm characteristics. Checking the types of operations in the algorithm showed that there are a number of conditional statements that modify data structures on a per-pixel basis, impeding data parallelism. Although this situation is not ideal, enough data parallel operations occur that operating on the image using this method is still a viable choice. Applying Equation 1 we determined that the time to broadcast the data is too large to justify using $ppeval$ on the inner loop, which executes a large number of fast operations. Alternatively, using $ppeval$ on the outer loop is a good option. Using $ppeval$ on the outer loop is equivalent to dividing the pixels evenly between all processors (similar to the approach followed in [15]). Though one inner loop iteration is short, in this situation all iterations combine to make up the $t_{body}$ value used in Equation 1, thus amortizing the cost of data transmission.

Backward projection could be parallelized identically to forward. However, with backward we also have the option of vectorizing the outer three loops because the number of iterations for the innermost loop does not vary per pixel. The entire backward algorithm can be condensed into a single $for$ loop that iterates as many times as the innermost loop. When this vectorization was attempted using serial Matlab, we found that we did not have enough memory to hold the input, intermediate, and output data structures (even with 16GB of memory), because the size of the data structures had to be significantly increased. Using distributed memory with Star-P, and splitting these structures across memories, we were easily able to adhere to the memory restrictions.

## 5.2 Other Applications

### 5.2.1 Cardiac-CT Algorithm

A second application that we studied is a Cardiac-CT algorithm [7]. This code employs a much more complex geometric model than the model used for Tomosynthesis. This increased complexity in geometry translates to more significant code customization due to the associated computational requirements. A high number of projections are produced for a spiral movement of the detector around the target object. In addition, more data dependencies are produced by assuming this geometry.

Applying the GPGPU-CUDA guide suggests that the best strategy to produce high utilization was to reuse some registers. This is because data dependencies and thread complexity prevent the creation of a large number of threads. Reusing registers allowed us to increase the number of active threads per processor. Even with the less favorable conditions of this algorithm, the GPU performance ($7X$ over serial execution) is still comparable to the performance of Cluster B ($6X$ over serial execution), though not nearly as fast as Cluster A ($19X$ over serial execution). However, the cost-performance still greatly favors execution on the GPU as shown in Figure 3.

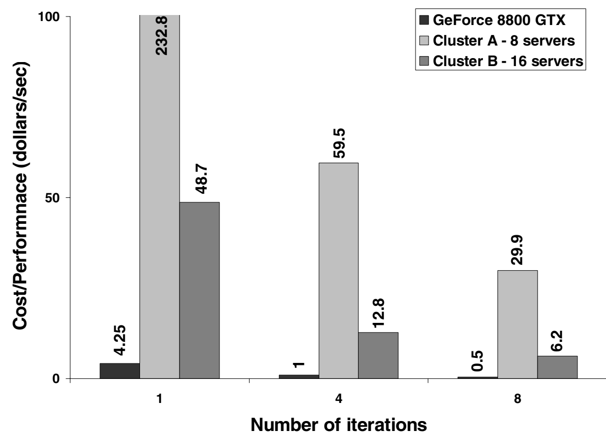


**Figure 3. Cost/performance for Cardiac-CT on our 3 platforms.**

### 5.2.2 Hyperspectral Toolbox

The code for the Matlab-based Hyperspectral imaging toolbox is a collection of functions designed to analyze multi- and hyper-spectral images [1]. The majority of the operations performed by the toolbox functions are completely vectorized (they are performed on the entire image matrix). When loops do exist in the code, they are mainly based on the number of clusters specified for the classification algorithms.

The Star-P guide presented in Section 4.2 was applied in several ways. First, since hyperspectral images may contain hundreds of 2D images, the image is initially loaded directly into distributed memory. When loops were encountered in the code, the decision to use data parallelism versus $ppeval$

required some thought because so many factors were involved. Two variable factors, the number of processors and the number of loop iterations, were especially critical (image size was assumed to be large as there is less penalty for being non-optimal with small images). If the number of classification clusters for a hyperspectral image is small and $ppeval$ is used, many processors may sit idle while a few do a large amount of work. Therefore, Equation 1 was applied to the code, and the method of parallelization is determined dynamically based on the system resources and algorithm characteristics. Finally, in one loop body an unsupported function was encountered for distributed data. Instead of serializing the data and redistributing it with each iteration, the data was serialized before entering the loop and redistributed upon loop completion.

For the images used in this study, most algorithms saw approximately a $2X$ speedup when using 4-processors. Some of the longer algorithms had speedups closer to $3X$ because of their ability to amortize the overhead costs of broadcasting the large image to each node.

## 6. Summary and Future Work

In this work we have studied two approaches to parallelizing multi-dimensional imaging applications with state-of-the-art technologies. We examined the strengths and weaknesses of the technologies, and presented a guide that will help scientists and programmers determine the effectiveness of using these approaches based on code characteristics. Next, we applied our guidelines to several imaging applications and found that we were able to obtain significant speedups by coordinating our parallelization methods with the inherent strengths of the technologies. For the Tomosynthesis algorithm a cost-performance analysis was presented that highlights some of the tradeoffs associated with executing this code on a single GPU compared with two more-expensive clusters.

Our future work will include enhancing our investigation of both technologies. Our first goal will be to make use of two or more GPUs to accelerate the Cardiac-CT application to extract more parallelism with fewer memory restrictions. We also intend to investigate ways to utilize a GPU from within the Star-P framework; this effort will start by using a local GPU for accelerating a subset of computations. We then plan to create a cluster where each node contains a GPU. This will allow us to explore new ways to divide and conquer computationally intensive applications by utilizing both the CPU and GPU on each node. Finally, we intend to explore ways to improve parallelization automation both for Star-P and CUDA.

## References


[1] E. Arzuaga, L. O. Jimenez, M. Velez, D. Kaeli, E. Rodriguez, H. T. Velazquez, A. Castrodad, L. E. Santos, and C. Santiago. A MATLAB toolbox for hyperspectral image analysis. In *Proceedings of Geoscience and Remote Sensing Symposium, IGARSS*, volume 7, pages 4839–4842, September 2004.

[2] I. Buck. *Stream computing on graphics hardware*. PhD thesis, Stanford University, Stanford, CA, USA, 2005. Adviser-Pat Hanrahan.

[3] R. Choy and A. Edelman. Parallel MATLAB: Doing it right. *Proceedings of the IEEE*, 93(2), 2005. special issue on "Program Generation, Optimization, and Adaptation".

[4] J. De Beenhouwer, R. Van Holen, S. Vandenberghe, S. Staelens, Y. D'Asseler, and I. Lemahieu. Graphics hardware accelerated reconstruction of spect with a slat collimated strip detector. In H. Arabnia, editor, *Proceedings of the 2006 International Conference on Image Processing, Computer Vision and Pattern Recognition*, volume I, page 7 pp, Las Vegas, 6 2006.

[5] M. Fletcher, C. McCosh, K. Kennedy, and G. Jin. Strategy for compiling parallel matlab for general distributions. Technical Report TR06-877, Rice University, Houston, TX, 2006.

[6] J. Kepner. MatlabMPI. *J. Parallel Distrib. Comput.*, 64(8):997–1005, 2004.

[7] Z. Liang, S. Do, W. C. Karl, U. Hoffmann, T. Brady, and H. Pien. Calcium De-blooming in Coronary CT Images. Technical report, Depart. of Biomedical engineering,BU and Department of Radiology, MGH, 2007.

[8] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: a system for programming graphics hardware in a c-like language. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 896–907, New York, NY, USA, 2003. ACM Press.

[9] Nvidia web page, http://www.nvidia.com/.

[10] NVIDIA CUDA Compute Unified Device Architecture programming guide, version 1.0. NVIDIA Corporation, 2007.

[11] Star-P user guide, release 2.4.1. Interactive Supercomputing, 2007.

[12] The peakstream platform: High productivity software development for multi-core processors. PeakStream Inc, 2006.

[13] Z. Wang, G. Han, T. Li, , and Z. Liang. Speedup osem image reconstruction by pc graphics card technologies for quantitative spect with varying focal-length fan-beam collimation. *IEEE Transactions on Nuclear Science*, 52(5):1274–1280, 2005.

[14] F. Xu and K. Mueller. Accelerating popular tomographic reconstruction algorithms on commodity pc graphics hardware. In *IEEE Transactions on Nuclear Science*, volume 52, pages 654–663, June 2005.

[15] J. Zhang, W. Meleis, D. Kaeli, and T. Wu. Acceleration of maximum likelihood estimation for tomosynthesis mammography. In *ICPADS '06: Proceedings of the 12th International Conference on Parallel and Distributed Systems*, pages 291–299, 2006.