Seismic imaging using GPGPU accelerated reverse time migration

Nader Moussa

ABSTRACT

In this report, I outline the implementation and preliminary benchmarking of a parallelized program to perform reverse time migration (RTM) seismic imaging using the Nvidia CUDA platform for scientific computing, accelerated by a general purpose graphics processing unit (GPGPU). This novel software architecture allows access to the massively parallel computational capabilities of a high performance GPU system, which is used instead of a conventional computer architecture because of its high throughput of numeric capabilities.
The key aspects of this research concern the hardware setup for an optimized GPGPU computer system, and investigations into coarse-grained, algorithm-level parallelism. I also perform some analysis at the level of the numerical solver for the Finite-Difference Time Domain (FDTD) wave propagation kernel. This paper demonstrates that the GPGPU platform is very effective at accelerating RTM, and this will lead to more advanced processing for better imaging results.

INTRODUCTION

Reverse time migration (RTM) is often used for seismic imaging, as it has preferable numerical and physical properties compared to competing algorithms, and thus generates better images (Zhang and Sun, 2009). These benefits come at a high computational cost, so research effort is required to make RTM a more economically competitive method for seismic imaging. This is the motivation for GPGPU parallelism of RTM.

The processing flow for imaging a seismic survey can be parallelized in many tiers. This multi-tiered parallelism has been noted in earlier computer architecture research for seismic imaging (Bording, 1996). This hierarchical parallelism is particularly prominent in RTM, and it provides opportunities for significant performance increase throughout the algorithm. A modern GPGPU platform, such as the Nvidia S1070, is uniquely capable of mirroring this tiered algorithm structure, because its architecture is similarly structured with both coarse-grain and fine-grain parallel capabilities.

At the highest level of abstraction, a data set can be divided into spatially separate regions of independent data (shot profiles). This is a Single Program, Multiple Data (SPMD) approach, and due to low data dependency, interprocess communi-

cation is generally not needed. This can directly map to a hardware multi-GPU implementation.

At each data subset, the migration can be further parallelized at a finer granularity. There are three potential stages for parallelism in the RTM algorithm, but there is a severe data dependency limitation. The imaging condition requires the computed results of both the forward wavefield, p_F, and the reverse wavefield, p_R, for each time step. Unfortunately, because the two wavefields are computed in opposite time directions, performing the imaging condition usually requires computing the complete wavefield p_F, writing it to disk, and reading its precomputed values for image condition correlation as soon as that time step is available from the reverse time wavefield. This data dependency is a major obstacle to parallelism at this stage, and constrains performance.

At the finest level of parallelism, the individual wavefield propagation steps can reduce the compational load by taking advantage of vectorization, floating-point math optimizations, and numerical reorganization. The imaging condition can also benefit from parallelization, because it is essentially a large 2D or 3D correlation. This is easily vectorizable, and is especially suitable for a GPU, which was originally designed as a large vector-computer.

Clearly, the GPGPU platform provides multi-tiered parallelism capability that matches the RTM structural design. The encouraging preliminary results seem to confirm that the GPGPU platform is well suited to RTM optimization, and suggest that further optimization can continue to yield dramatic execution time improvements. This will allow more advanced processing with correspondingly better subsurface image results.

CUDA PROGRAMMING METHODOLOGY

Nvidia's novel technology, "Compute Unified Device Architecture" (CUDA) is a software interface and compiler technology for general purpose GPU programming (Nvidia, 2008). The CUDA technology includes a software interface, a utility toolkit, and a compiler suite designed to allow hardware access to the massive parallel capabilities of the modern GPU, without requiring the programmer to construct logical operations as graphical instructions. The latest release of CUDA, version 2.1, exposes certain features only available in the Tesla T10 GPU series. Below, all specifications are given based on the capabilities of the T10 GPU using CUDA 2.1 software. For easy reference, Table 1 summarizes the terminology and acronyms that apply to the software and hardware tiers. An acronym-guide is also provided in Table 2 in the Appendix.

CUDA programs have two parts: "host" code, which will run on the main computer's CPU(s); and "device" code, which is compiled and linked with the Nvidia driver to run on the GPU device. Most device code is a "kernel," the basic functional design block for parallelized device code. Kernels are prepared and dispatched by

Software Model		**Hardware Model**	
Element	Maximum	Physical Unit	#
Thread	512 threads per block Arranged in 3D block not exceeding $512 \times 512 \times 64$ in $< x, y, z >$ and 512 total	Scalar Processor (**SP**) or "Streaming **Core**" Each core executes one thread at a time	**8**
Warp	Each 32 threads are statically assigned to a warp	**SP Pipeline** A full warp (32 threads) executes in 4 clock cycles (pipelined 4-deep across 8 cores)	**16**
Block	Arranged in 2D grid not exceeding 65535×65535 in $< x, y >$	Streaming Multiprocessor (**SM**)	**30**
Kernel Grid	Problem or simulation representation	**GPU** Only one kernel is running on the GPU at a time (More are possible, but this is complicated).	**4**

Table 1: CUDA software and hardware mapping. This table briefly summarizes the CUDA software architecture and its implementation on a T10 GPU.

host code. When the kernel is dispatched, the host code specifies parallelism parameters, and the kernel is assigned to independent threads which are mapped to device hardware for parallel execution.

The coarsest kernel parallelism is the "block," which contains several copies of threads running the same code. Each block structure maps to a hardware multiprocessor. Blocks subdivide a large problem into manageable units which will execute independently. It should be noted that inter-block synchronization and communication is difficult without using expensive global memory space or coarse barriers. Inside each block there are up to 512 threads, organized into sub-groups or "warps": these groups of up to 32 threads. At this level of parallelism, shared memory and thread synchronization is very cheap, and specific hardware instructions exist for thread synchronization. As of CUDA 1.3, available on the Tesla T10, synchronization "voting" can be used to enable single-cycle inter-thread control. As an extra performance boost, threads which are running the same instructions are optimized with "Single Instruction, Multiple Thread" (SIMT) hardware, sharing the Instruction Fetch (IF) and Decode (DEC) logic and efficiently pipelining operations. If conditional program control flow requires different instructions, the threads must then serialize some of these pipeline stages. Peak performance is achieved when all conditional control-flow is identical for threads in a single warp. In the case of Finite Difference Time Domain (FDTD) wave propagation code, it is generally possible to have all threads operating in SIMT mode. The boundary conditions at the edges of thread blocks, and at the edges of the simulation space, are currently the only exceptions to this SIMT mode.

HARDWARE PLATFORM

During this research, testing was performed on an HP ProLiant with an attached Tesla S1070 GPGPU rack-mounted blade server. This unique platform implements the CUDA 2.1 software specification, with hardware Compute Capability 1.3 GPU acceleration (Nvidia, 2008). The S1070 provides four Tesla T10 GPUs, which provide vector-style parallelism for general purpose computing.

The basic architecture consists of a "Host System," using regular CPUs and running a standard Linux operating system. Attached is the "Device," a 1U rack-mounted GPGPU accelerator which provides the parallelism discussed in earlier sections. The CUDA technology uses the terms "host" and "device" to refer to the various hardware and software abstractions that apply to either CPU or GPU systems. Although the S1070 has four GPUs, it is considered one "device"; and similarly, there is one "host," although it has 8 CPU cores in this system. Below is a summary of the system specifications:

Host: HP ProLiant DL360 G5 (HP ProLiant series, 2009) (HP ProLiant DL360G5 Overview, 2009)

- 2× Quad-Core Intel Xeon E5430 @ 2.66 GHz

- 6144 KB L2 Cache (per core)
- L2 Cache Block size: 64 B
- 32 GB main memory
- 1333MHz Front Side Bus
- PCI-e #1: 8x pipes @ 250MB/s each
- PCI-e #2: 8x pipes @ 250MB/s each
- Gigabit Ethernet connection to SEP Intranet

Device: Nvidia Tesla S1070 Computing System (S1070 Product Information, 2009)

- 4× T10 GPU @ 1.44GHz
- 30 Streaming Multiprocessors (SM) per GPU
- 8 Scalar Processor (SP) cores per SM
- 32 threads per warp
- 16K 32-bit registers per block
- 16KB Shared Memory per block
- 4GB addressable main memory per GPU
- Memory controller interconnect for data transfer to host and other GPU address spaces
- Concurrent Copy & Execution feature: "Direct Memory Access" (DMA) style asynchronous transfer available on T10 GPUs
- Programmable in CUDA

The S1070 is a very recently released commercial platform specifically tailored for high-performance scientific computing. Nvidia recommends using it only with certain host hardware environments. The system installation procedure is explained in the appendix, along with solutions to difficulties that may be encountered.

The final system environment runs `CENTOS 5.2 for x86_64` and using the Nvidia Tesla Driver (`Linux x86_64 - 177.70.11`). Two Host Interconnect Cards (HIC) are installed and configured in the ProLiant. Both cards are connected to the S1070 unit via two PCI-e cables. This setup produces a reliable and functional system for GPGPU computational acceleration.

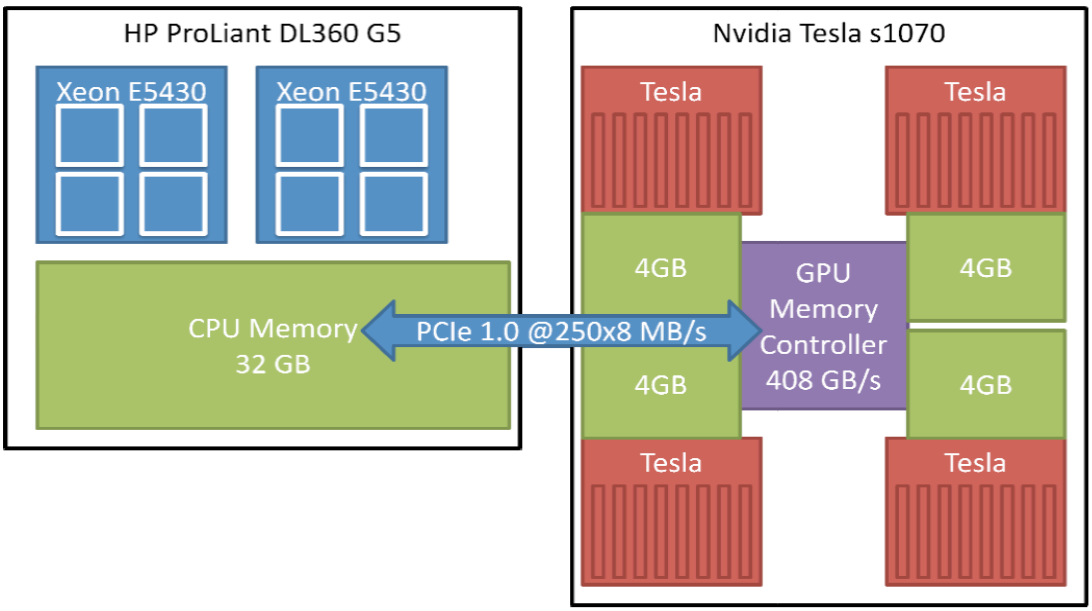

Figure 1: Schematic representation of the Host-Device (CPU-GPU) interconnection and memory structure. The compartmental memory structure on the Device side is problematic for multi-GPU programs, because it severely restricts shared memory methods. Much of my recent implementation efforts address this issue. [**NR**]

EVALUATION METRICS

There are many ways to compare and evaluate parallelization schemes for RTM. Because the GPGPU approach is so novel, it is difficult to perform direct comparison with other parallelization schemes for Reverse Time Migration. Other hardware platforms do not provide the same software abstractions. Many of the GPGPU metrics thus have no direct comparable equivalent on alternative systems. Of course, key performance metrics are directly comparable to serial or parallel CPU RTM implementations. These include:

- Total execution time
- Cost ($) per FLOPS
- FLOPS per Watt

Other internal performance metrics of my implementation can be compared to academic and industrial research progress in high-performance GPGPU wave propagation. Wave propagation has been previously implemented in Finite Difference Time Domain (FDTD) for nearly identical hardware (Micikeviciuis, 2008); the forward- and

Figure 2: Rack mount view of the SEP Tesla system. At the top is the S1070 1U 4xGPU GPGPU Computing Appliance. Below is the HP ProLiant Xeon 64 bit, 8 core (2xSMP, 4xCMP) system, `tesla0.stanford.edu` which runs the host operating system. [**NR**]

reverse-wavefield computation performance can be directly compared to such an implementation. FDTD performance measurements include:

- Maximum computational grid size
- Block subdivision size
- Wavefield grid points per second
- Numerical order of spatial derivatives

One goal of SEPs investigations into various parallelization technologies is to subjectively evaluate the feasibility for future performance, ease of development, and maintainability of code. Technologies like the CUDA/GPGPU approach are compared subjectively to other systems, such as the SiCortex SC072 "Desktop Cluster" as well as conventional multicore and multi-node CPU parallelization. The following metrics can be roughly estimated for each technology, noting that there is some ambiguity in direct comparisons across widely varying exotic architectures:

- Cost ($) per FLOPS

- FLOPS per Watt
- FLOP operations needed for complete migration
- Execution time for complete migration
- Accuracy of the wave propagation operator

As I am not trained as an interpretational geologist, subjective assessment of the image quality is difficult for me. Nonetheless, it has been widely established in industrial contexts that the correct implementation of RTM yields better images for decision-making and analysis. Certain computational architectures can enhance this effect by enabling higher-accuracy RTM, (e.g. using higher-order wavefield operators). By providing very cheap floating-point math, the GPGPU approach enables more operations per data point, allowing more accurate wave modeling with minimal execution time overhead. The overall speedup that a GPGPU implementation can provide can allow additional iterations as part of larger inversion problems, increasing the accuracy of these processes. The result is a subjectively better migrated image.

Finally, it is worth noting the benefits of GPGPU parallelization from a software engineering and code-maintenance standpoint. CUDA is designed to be simple, consisting of a set of extensions to standard C programming. The programming environment is easy to learn for most programmers. The code is systematically separated into host setup code and device parallelization code; and CUDA can interoperate with C or C++, allowing functional- or object-oriented system design, as the situation requires.

IMPLEMENTATION

I developed a wave propagation kernel, implemented in CUDA, for use in forward- and reverse-time wave propagation. I also implemented a simple correlation imaging condition. Due to time constraints, I was not able to implement a more advanced imaging condition with true-amplitude correction, noise-removal, and angle-gather decomposition.

For the purposes of this report, I will focus on single-GPU kernels. I made significant progress towards multi-GPU asynchronous parallelization, but this code is not yet ready to provide benchmark results. The eventual goal is to perform the forward-wave, reverse-wave, and imaging condition subroutines on independent GPUs. However, preliminary benchmark results cast doubt on whether that approach will decrease total execution time, because the bottleneck appears to be host-device transfer time rather than computational limitations.

Implementation of an eighth-order spatial derivative added negligible computational overhead to the problem, as compared to the naive second-order wave operator. This suggests that other more sophisticated time-stepping methods, such as Arbitrary

Difference Precise Integration (ADPI) wave solvers (Lei Jia and Guo, 2008), may also have negligible computational overhead. Such methods will enable coarser time-steps without the numerical stability limitations that are inherent in FDTD approaches. Thus the use of those methods may reduce overall execution time.

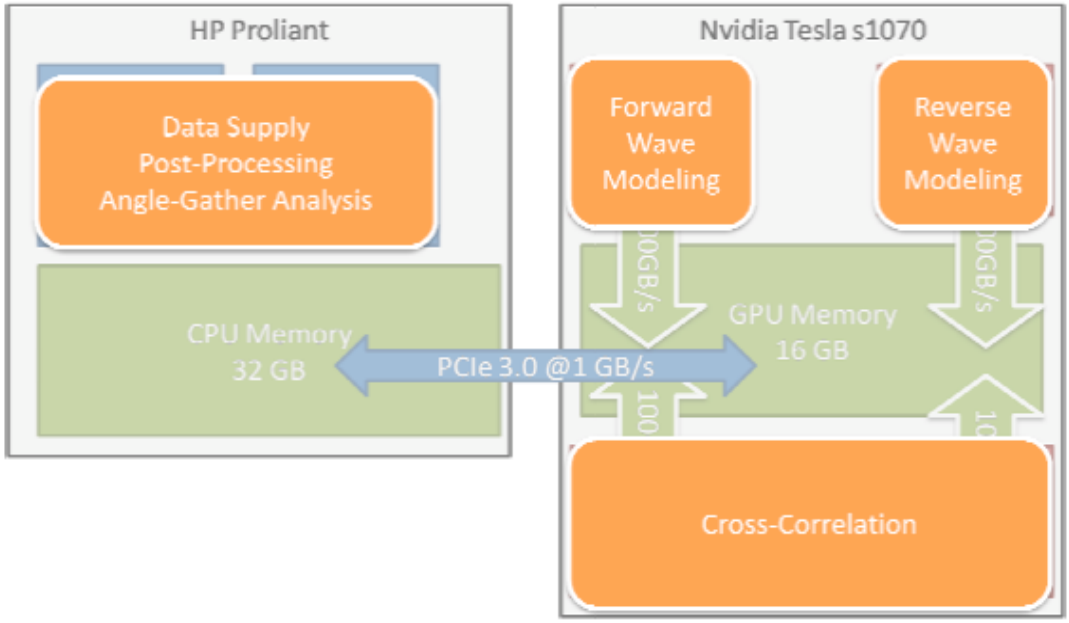

Figure 3: Schematic view of the multi-GPU algorithm for coarse-grained parallelism. The intent is to perform the overall RTM process with separate stages executing in parallel on independently controlled GPUs. This coarse parallelism can help pipeline the process and hide the memory transfer time. My current implementation and benchmark-code does not yet implement this strategy. [**NR**]

The expansion of the solver to a full 3D model space will require significant extra programming. The code base for the 2D model is intended to be extensible, and the CUDA framework allows block indexing to subdivide a computational space into 3 dimensions, assigning an (X,Y,Z) coordinate to each block and each thread. Because of time constraints, I did not complete the full 3D modeling for benchmark comparison.

In the current iteration, the host code does not perform significant parallelism. Earlier efforts used `pthread` parallelism on the host CPUs for data preprocessing while the input loaded from disk, but the time saved by this workload parallelism was negligible compared to the overall execution time.

The result of my implementation is a propagation program, `waveprop0`, and an imaging program, `imgcorr0`, written in CUDA. These are piped together with a set of Unix shell scripts to manage the overall RTM sequence for forward and reverse-time propagation with an imaging condition.

Future implementations will seek to integrate these programs into one tool with

several CUDA kernels, but the overlying data-dependence issue must be solved theoretically before the processes can be entirely converted to a streaming methodology. For trivial-sized problems, the entire computational result of forward and backward wave propagation can remain in graphics device memory for use, but this approach has inherent problem-size limitations. Other methods of eliminating the costly host-device transfers have been proposed (Clapp, 2009). Such methods eliminate the bottleneck by preserving the wavefield state in GPU memory at the final timestep, and backward-propagating to recompute the wavefield at arbitrary time. This takes advantage of the cheap and fast wave propagation kernel. Another approach is the effective pipelining of the RTM process to allow arbitrary-sized input data sets. Finally, a major area of continuing work is the complete linking of CUDA research code with the standard `SEPlib` programming environment and toolkit. This will be extremely beneficial from the standpoint of code portability and interoperability with other research areas.

PERFORMANCE AND BENCHMARK SUMMARY

For the sake of simplicity and consistency, I tested my RTM code on synthetic data. I used a simple subsurface velocity model with a few reflecting layers. This same velocity model has been used by other SEP students and researchers, and although it does not represent the complex subsurface behavior of a real earth model, it provides sufficient complexity to evaluate the correct functionality of the RTM implementation. My current work to integrate `SEPlib` with the GPGPU environment will enable benchmarking and testing on more standard data, eventually including field recorded data sets. This will be an important step to verify and compare GPGPU performance to more traditional paralellism schemes.

Unless otherwise noted, the benchmark results I report were computed on a two dimensional wavefield space, with grid size 1,000 x 1,000.

GPU execution time is shown in Figure 5 for a 1,000,000 point grid, (1,000 x 1,000 2D computational space). It is compared to a serial implementation of RTM on the CPU. Due to time constraints, I was not able to compare the GPGPU parallelization to other parallel RTM versions.

Evidently, GPU parallelization has a dramatic effect on the total execution time, reducing it by a factor of more than 10x. With 240x as many cores, however, this is sublinear parallelization. Closer profiling of the CUDA algorithm execution time revealed the computational breakdown shown in Figure 6. This profiling was accomplished using timer variables compiled in the device code, as standard code profilers have difficulty working with the GPGPU environment. Most of the bottleneck is clearly the memory transfers between host and device, which are required for the imaging condition. The primary focus of further research is to work around this limitation: first, by optimizing the memory transfers as much as possible; and more importantly, by developing numerical schemes that can perform the imaging step

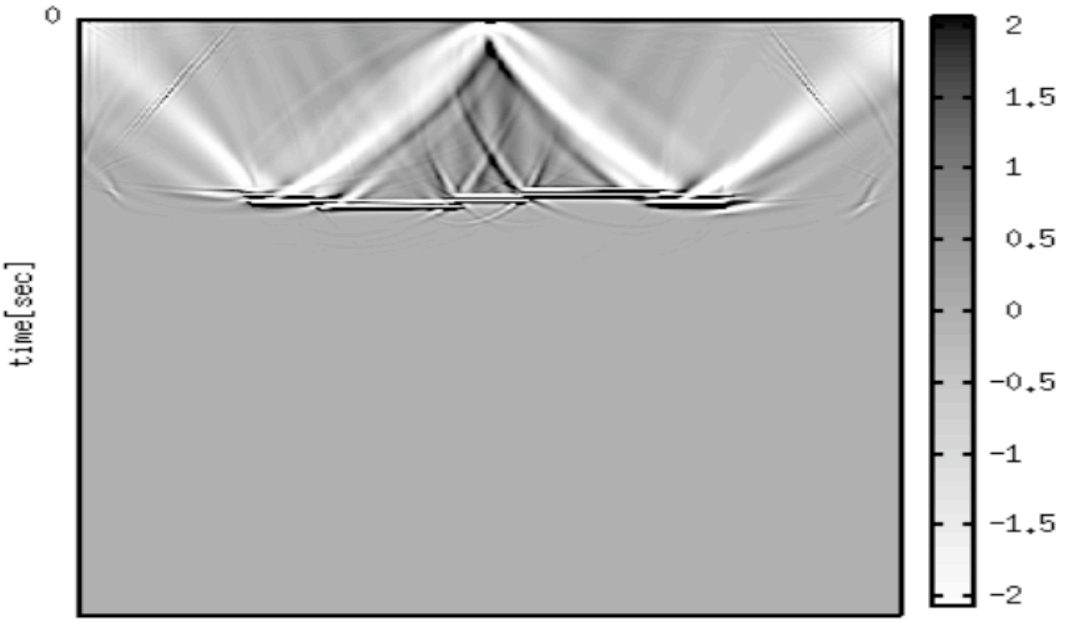

Figure 4: Preliminary RTM image results on a synthetic data set with a few simple horizontal reflectors. This test verified functionality of my preliminary RTM implementation on the GPGPU system. Wave diffraction is visible at the corners, probably due to the unrealistic, abrupt end of the layers in this synthetic model. [**NR**]

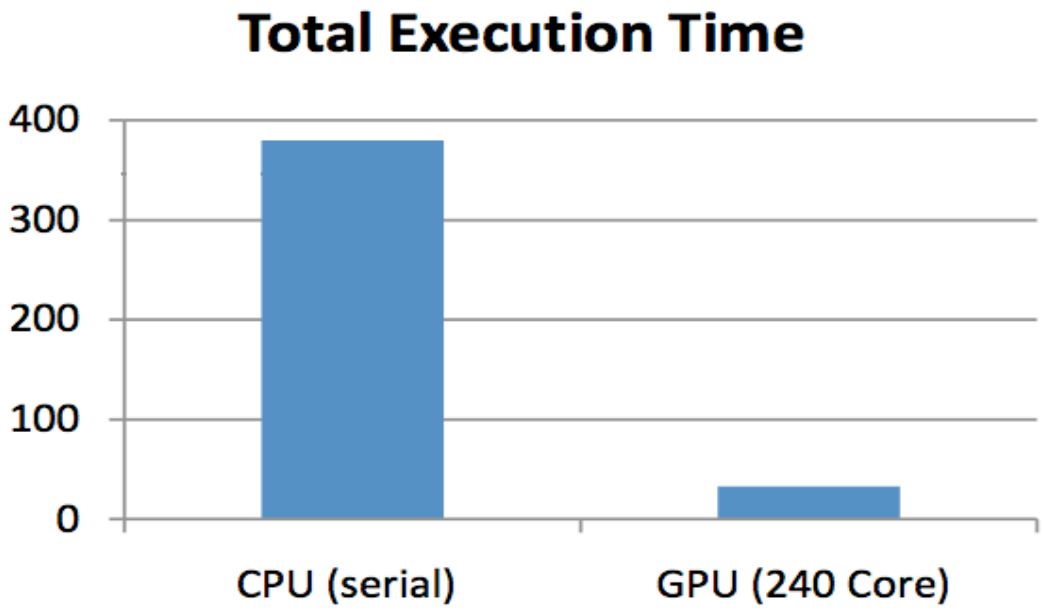

Figure 5: Total execution time for RTM imaging, comparing serial implementation on CPU (executed on the ProLiant Xeon host), compared to a single GPU CUDA parallelization. [**NR**]

without as much expensive transfer overhead.

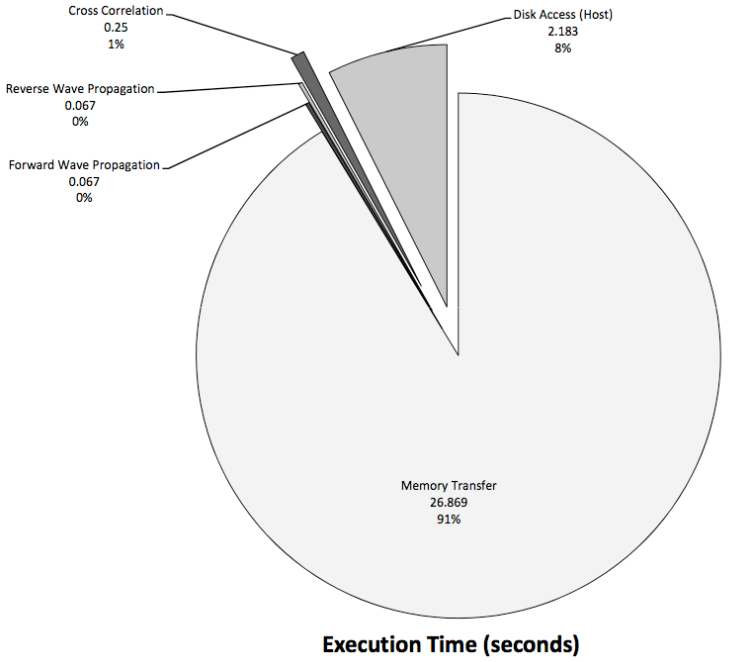

Figure 6: Breakdown of program execution time for the CUDA implementation. Very little time is spent executing numerical processing code (wave propagation or imaging condition). The vast majority of time is spent in host-device memory transfer over the PCI-e bus (between the ProLiant CPU system and the Nvidia Tesla S1070). [**NR**]

CONCLUSION

The dramatic speedup of the computational kernel provides strong motivation for continued work in GPGPU parallelism. Benchmark results suggest that the most important area to tackle is Host-Device (PCI-e) bus bandwidth, which accounts for 90% of the total system utilization time.

At present, my implementation does not have any tasks for the high-performing Xeon processors on the host. These CPUs are suitable for performing a lot of useful work, such as data post-processing or visualization. An alternative architecture could tightly couple CPU and GPU processes to maximize system utilization.

Another suggested research area is the implementation of compression during transfer. Velocity models, which contain large quantities of redundant data, could easily be compressed; seismic records will probably not compress well with a lossless algorithm such as GZIP (LZ77) because they do not contain the same amount of redundancy as velocity models. In future work, I will quantify these compression ratios for real data sets, which will help validate the utility of compressed Host-Device communication.

The system-level transition towards exotic computing platforms is always an engineering tradeoff. The performance benefits of such an environment must be sufficiently high to offset the development and maintenance cost with the new system. The GPGPU platform and its CUDA programming environment is sufficiently familiar to a geophysical programmer and exposes massive parallel capability in a straightforward way. The immediate performance boost is evident from the preliminary benchmarks presented in this report. Significant further optimizations can be realized in future work via more analysis and refinement of this GPGPU approach.

APPENDIX

Tesla S1070 system setup

This section details the procedure to install and configure the hardware for the Tesla S1070 GPGPU system.

During my early work, I configured the ProLiant system to run Ubuntu 8.10 and Nvidia 180.22 drivers. This required recompiling Nvidia Debian kernel modules (.ko files). The recompiled modules successfully connected to the S1070 system, but incorrectly identified it as an Nvidia C1060. Downgrading the operating system to Ubuntu 8.04 enabled the modules to correctly connect and recognize the Nvidia S1070, but also produced an unstable system, which occasionally crashed. Following advice from Nvidia, I switched the ProLiant operating system to CENTOS and had significantly more success. However, the Nvidia 180.22 drivers have not been fully tested on the 1U rack-mount S1070 systems with four GPUs. There were several system hang-ups and unexpected, non-repeatable crashes. It should be noted that the GPGPU driver for the 1U Tesla system interferes with some automatic configuration of the Linux operating system (specifically graphic configuration for X11). This happens because the S1070 appears to X11 to be a video accelerator and display driver even though it cannot be connected to a physical display monitor.

Another potential configuration problem arises from the presence of two Host Interconnects on the S1070 1U unit. The Nvidia documentation mentions that these interconnects allow the S1070 to optionally connect to two separate host CPU systems. However, even though only one host is used in our system, we found that both interconnects should be used because Connecting and configuring only one card results in access to only 2 out of the 4 available Tesla T10 GPUs on the S1070 1U server. Using both interconnects allows access to all four GPUs, and also doubles the PCI-e bandwidth available to the S1070 memory controller.

The final system environment runs `CENTOS 5.2 for x86_64` and using the Nvidia Tesla Driver `(Linux x86_64 - 177.70.11)`. Two Host Interconnect Cards (HIC) are installed and configured in the ProLiant. Both cards are connected to the S1070 unit via two PCI-e cables. This setup produces a reliable and functional system for GPGPU computational acceleration. Much difficulty can be avoided by using exactly

these system and driver versions.

Acronyms Quick Reference

CUDA	Compute Unified Device Architecture
FDTD	Finite Difference, Time Domain (wave simulation)
GPU	Graphics processing unit
GPGPU	General purpose graphics processing unit
RTM	Reverse Time Migration
SEP	Stanford Exploration Project
SM	Streaming Multiprocessor
SP	Streaming Processor
SPMD	Single program, multiple data

Table 2: Quick reference for CUDA acronyms.

REFERENCES

Bording, R. P., 1996, Seismic modeling with the wave equation difference engine: SEG Expanded Abstracts, **15**, 666–669.

Clapp, R. G., 2009, Reverse time migration with random boundaries: SEP-Report, **138**.

HP ProLiant DL360G5 Overview 2009, HP ProLiant DL360 Generation 5 - overview. Hewlett Packard. (`http://h18004.www1.hp.com/products/quickspecs/12476_na/12476_na.html`).

HP ProLiant series 2009, HP ProLiant DL360 G5 Server series specifications. Hewlett Packard. (`http://h10010.www1.hp.com/wwpc/us/en/en/WF06a/15351-15351-3328412-241644-241475-1121486.html`).

Lei Jia, W. S. and J. Guo, 2008, Arbitrary-difference precise integration method for the computation of electromagnetic transients in single-phase nonuniform transmission line: IEEE Transactions on Power Delivery, **23**.

Micikeviciuis, P., 2008, 3d Finite Difference Computation on GPUs using CUDA.

Nvidia, ed., 2008, Nvidia CUDA Programming Guide, version 2.1: Nvidia Corporation.

S1070 Product Information 2009, Nvidia S1070 Computing System. Nvidia. (`http://www.nvidia.com/object/product_tesla_s1070_us.html`).

Zhang, Y. and J. Sun, 2009, Practical issues in reverse time migration: First Break, **27**, 53–59.