Army Research Laboratory

GPUs: An Emerging Platform for General-Purpose Computation

by Sha'Kia Boggan and Daniel M. Pressel

ARL-SR-154 **August 2007**

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

Army Research Laboratory
Aberdeen Proving Ground, MD 21005-5067

ARL-SR-154 **August 2007**

GPUs: An Emerging Platform for General-Purpose Computation

Sha'Kia Boggan and Daniel M. Pressel
Computational and Information Sciences Directorate, ARL

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)* August 2007	**2. REPORT TYPE** Final	**3. DATES COVERED (From - To)** 1 October 2006–31 May 2007

4. TITLE AND SUBTITLE

GPUs: An Emerging Platform for General-Purpose Computation

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Sha'Kia Boggan and Daniel M. Pressel

5d. PROJECT NUMBER

7UH7CC

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

U.S. Army Research Laboratory
AMSRD-ARL-CI-HC
Aberdeen Proving Ground, MD 21005-5067

8. PERFORMING ORGANIZATION REPORT NUMBER

ARL-SR-154

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The affordability and computational power of GPUs have made them the focus of an emerging area of research designed to explore their performance for general-purpose computation. GPGPU, as this area of research is called, involves the exploration of the computational power of programmable GPUs and their suitability for non-graphics applications through algorithm and software development. Although there are some challenges with using these specialized devices for numerous applications, their attributes and significant speedup for some applications continue to make them an attractive platform for research.

15. SUBJECT TERMS

GPU, HPC, supercomputing, scientific computing

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Daniel Pressel
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**19b. TELEPHONE NUMBER** (*Include area code*)
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL	50	410-278-9151

Contents

List of Figures

List of Tables

INTENTIONALLY LEFT BLANK.

1. Introduction

Rendering graphics is a mathematically intensive task that, if tackled solely by the CPU, could cause performance to slow down. To offload this work from the CPU and onto hardware that has been optimized for the task, personal computers, workstations, and game consoles use a GPU.* The GPU is a combination of a coprocessor and tightly coupled high-speed memory (GRAM) that is responsible for rasterizing images through a graphics pipeline. The CPU is still required for other classes of work (e.g., solving the physics-based equations in a CFD simulation or a state-of-the-art video game).

GPUs can be grouped into the following two classes:

1. Low-end units that provide support for two-dimensional (2-D) applications (e.g., Microsoft Word, XTerm, and Telnet, surfing the web, and watching videos).
2. High-end units equipped with Z-buffers and full hardware support for sophisticated three-dimensional (3-D) applications. Up until a few years ago, these were very pricey and usually were only found in midrange to high-end products from companies like SGI.

As a result of advances in manufacturing technology and their widespread use in game consoles and PCs that have been optimized to play video games, the cost of high-end GPUs has declined dramatically. The rest of this report will assume that these are the units being discussed.

Historically, GPUs operated within a fixed-functionality pipeline, with limited capabilities for rendering. Modern GPUs consist of fully programmable floating-point pipelines with notable computational power and memory bandwidth. These architectural advances equip the GPU for more than just graphics. GPUs are best suited for applications that are highly parallel, computationally intensive, and have highly regular memory-access patterns.

Over the past few years, GPUs have surpassed CPUs in performance, in absolute terms and in relative speedup over time (i.e., at a rate greater than predicted for CPUs by Moore’s Law). Researchers and code developers are intrigued by the potential to use GPUs as an attached processor for the purpose of speeding up nongraphics algorithms. GPUs have been leveraged into a number of nongraphical applications, including signal and image processing, bioinformatics, CFD, chemical dispersion, database operations, and mathematical libraries (e.g., BLAS).

This report discusses the current uses of GPUs and their potential for use in general-purpose computing. Section 2 describes the evolution of the architecture of the GPU as it lends itself to general-purpose computation. Sections 3 and 4 discuss some limitations and issues that must be

*Note: definitions for many of the terms used in this report can be found in the glossary section.

considered when harnessing the GPU's power for nongraphics applications. Some software and programming tools that have been helpful in the programming of GPUs are explored in sections 5 and 6. Section 7 outlines some applications that have proven to be well-suited for computation on the GPU. Lastly, section 8 discusses performance results that various researchers have reported when using GPUs for nongraphics applications.

2. GPU

Driven by the economics of the game industry, GPUs have become an affordable and attractive platform for research techniques to increase the speed of general computation. The performance of GPUs has increased yearly and is projected to continue at rates that surpass the performance growth rate of CPUs (figure 1). Current GPUs have achieved significant gains in performance and memory bandwidth over CPUs. Stanford University, in their protein-folding simulation project, has observed performance gains of up to 40× that of an Intel Pentium 4 CPU while using high-end ATI GPUs (*1*). According to Fan et al. (*2*), the performance advantage of GPUs over CPUs can be attributed to the following:

1. "A current GPU has as many as 16 pixel processors and 6 vertex processors that execute four-dimensional (4-D) vector* floating-point instructions in parallel.
2. Pipeline constraint is enforced to ensure that data elements stream through the processors without stalls.
3. Unlike the CPU, which has long been recognized to have a memory bottleneck for massive computation, the GPU uses fast on-board texture memory, which has one order of magnitude higher bandwidth."

In his survey, Owens gives credit for the GPUs' better performance to the "highly data-parallel nature of graphics computations," which "enables GPUs to use additional transistors more directly for computation, achieving higher arithmetic intensity with the same transistor count" (*3*). Because of the performance disparity, the GPU has been established as an economical evolving research platform and its power is being used for general-purpose computation.

With the GPU as a computing device dedicated to rendering graphics tasks, it is important to understand its architecture. Early GPUs were the size of a large file cabinet. Over the years, advances in technology have shrunk them to the point that GPUs and their associated chips will fit onto a small printed circuit card, and, in some cases, are found directly on a computer's motherboard. The GPU renders a graphics task through a pipeline (figure 2) consisting of a

*This 4-D vector refers to the RGBA texture data.

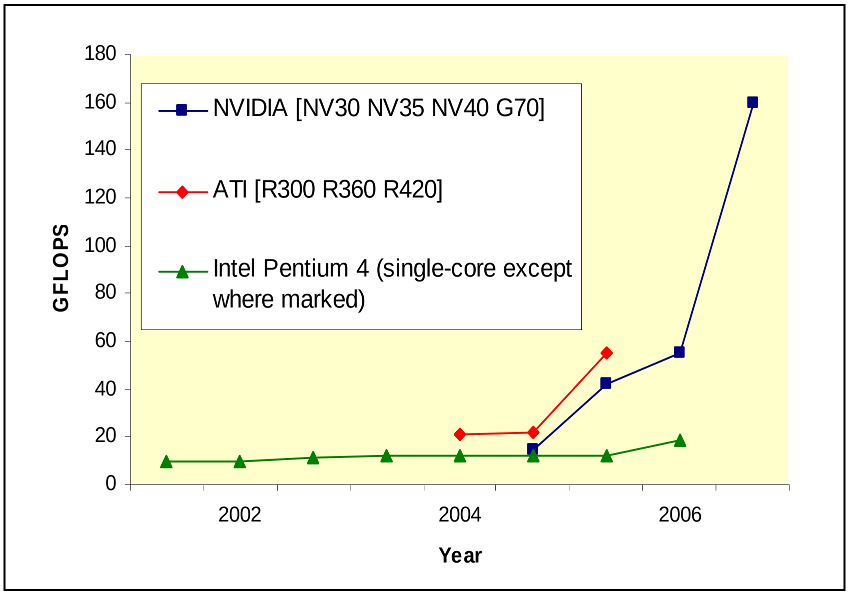

Figure 1. The programmable floating-point performance of GPUs (measured on the multiply-add instruction counting two floating-point operations per MADD) (*3*).

vertex processor, a rasterizer, a fragment processor, and a frame buffer. In order to enhance the performance of the GPU, it is now common practice for the GPU to contain multiple vertex and fragment processors working in parallel. The newest GPU architectures are merging the vertex and fragment processors into a single unified pool of processors, which leads to more optimal performance because of the increased capacity of the GPU to respond to the varying shading demands of an application. The vertex processor assigns or modifies color and texture coordinates for each vertex and performs any necessary updates of a vertex's positional data. The GPU stores pixel properties, such as texture, in the high-speed graphics memory, which uses a combination of multibanking, data streaming, specialized cache designs, and other techniques to provide superior levels of bandwidth and latency (*4*). A nonprogrammable unit with fixed-functionality within the pipeline then transforms the streams of vertices of a 3-D geometric scene description to a 2-D screen position (*5*). The vertices are then grouped into geometric primitives (normally, triangles or quadrilaterals) and sent to the rasterizer, which generates a stream of fragments for each pixel covered by the primitive. In the fragment processing stage, a battery of tests is conducted on each fragment to determine if it will affect the final image. If all tests pass, the fragment is written to the frame buffer. GPUs employ the paradigm of stream programming, which allows high-efficiency parallel programming without the complicated software engineering and design issues (*6*). GPUs also have been labeled as vector processors operating within the SIMD programming model. The basic architecture of GPUs has remained stagnant for the last 20 years; however, they have gained flexibility with increased programmability, with modern GPUs containing programmable vertex and fragment processors.

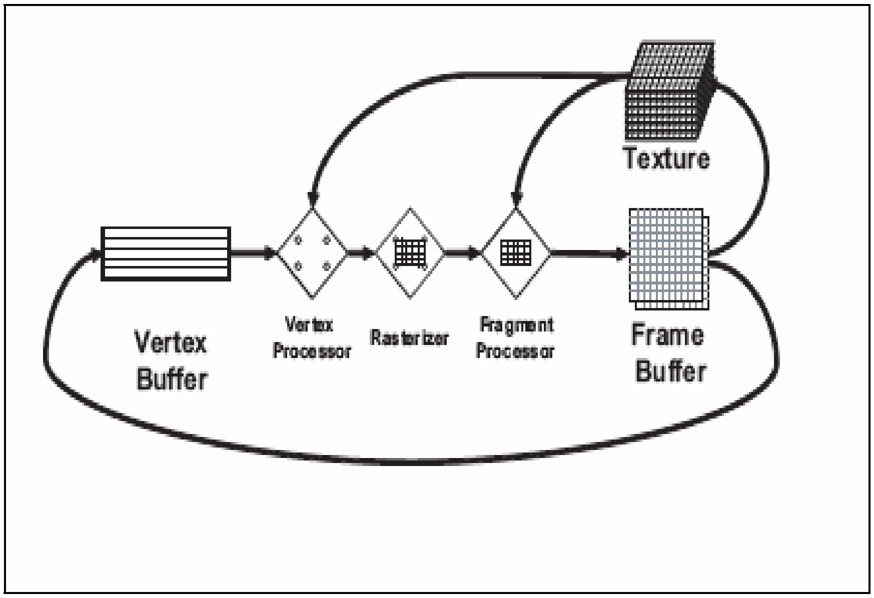

Figure 2. The graphics hardware pipeline (*3*).

GPGPU

The increased flexibility of GPUs has created a new area of research that explores their performance for general-purpose computation. This area of research entitled general-purpose computation on GPUs, or GPGPU, is a highly evolving one because of GPU's cost effectiveness and computational power (*7*). Historically, the graphics pipeline described earlier was a "fixed function pipeline, where the limited number of operations available at each stage of the graphics pipeline was hard-wired for specific tasks" (*3*). Evolution has allowed a more flexible programmable pipeline, to emerge with efforts concentrating primarily on the vertex and fragment stages. The light and transformation operations on vertices that were present in the vertex stage of the earlier design of the graphics pipeline have been replaced by a user-defined vertex program; moreover, the fixed fragment operations that took place in the fragment stage have been replaced with a user-defined fragment program. "A programmer can now implement custom transformation, lighting, or texturing algorithms by writing programs called shaders" (*8*). Owens' survey highlights the "vital step for enabling general-purpose computation" on GPUs as being "the introduction of fully programmable hardware and an assembly language for specifying programs to run on each vertex or fragment" (*3*). GPUs now include fully programmable processing units with support for 32-bit floating-point precision. Each generation of GPUs is expanding on its already favorable qualities as the next generations from NVIDIA and ATI (the major players in this part of the market) are expected to support 64-bit floating-point precision (*9, 10*). The speed, increasing precision, and rapidly expanding programmability of GPUs continue to promote them as a platform for general-purpose computation.

3. Limitations

Although the expanding capabilities of modern GPUs make them a compelling platform for general-purpose computation, limitations and difficulties exist and must be considered. GPUs are highly specialized devices and were designed to make images of 3-D scenes; the evolution and tuning of these devices have mainly been to optimize the highly parallel tasks of computer graphics. Although various applications have been enabled by this performance tuning, many applications are outside the narrow focus and are not well-suited for computation on a GPU. The lack of integers, logical operations, and fixed-point arithmetic renders the GPU ineligible for many computationally intensive tasks. Furthermore, "the lack of double precision hampers or prevents GPUs from being applicable to many very large-scale computational science problems" (*3*). In addition to their limited applicability, GPUs are not trivial to program; they require the programmer to consider an application that is normally nongraphics in graphics terms, which requires the programmer to be knowledgeable of the design, evolution, and limitations of the underlying GPU architecture. However, by hiding many of their architectural features, the vendors of GPUs have made it almost impossible to predict on an a priori basis what the potential performance of a specific algorithm on a specific GPU should be. This architectural black box makes it difficult to identify the correct chip for the job and determine when it is time to stop tuning the code and move to a new project. A multidisciplinary effort by developers in computer graphics and the field of interest is usually required to program efficiently. Since the GPU is designed to process images for the screen, "it may handle as many pixels as the maximum resolution of the image it can process", which translates to "the largest size for a dimension of a data stream is 2,048 floating-point elements" (*11*). This limitation, as well as the number of input parameters that can be passed to a GPU function being limited to eight, affects the GPUs' relevance to large-scale scientific problems. Transferring results back to main memory has proven to take a considerable amount of time on GPUs, although this limitation has been significantly alleviated by the higher bandwidth offered with PCI-Express ports.

4. The Curse of Memory

Most HPC applications require large amounts of memory (several gigabytes to many terabytes) and multiple fast processors (frequently with a combined peak speed exceeding gigaflops, with the largest systems having peak speeds that exceed 10 teraflops). It is instructive to look at how such systems are programmed. There are three main approaches to parallelizing applications on the distributed memory parallel architectures used in most of today's HPC systems:

1. Pseudo shared memory, also called globally addressable distributed memory, such as UPC, CAF, Global Arrays (used in NWCHEM), and the SCALAPACK library.[*] This approach assumes that the application requires a large amount of memory and, therefore, spreads the data across multiple nodes. However, it treats the data as though they were in multiple gigantic arrays that span nodes. In most cases, this will result in fine-grained communication. Although some systems have a system interconnection that can support such a communication pattern with an acceptable level of performance, these interconnections tend to be more expensive and, therefore, are not commonly found. As a result, this programming paradigm is used less frequently than the third approach.

2. Replicating a significant portion of the data on each of the nodes is required for some applications in chemistry, ray tracing, applications using databases, etc. Although this allows one to achieve high levels of speedup through parallelization, it limits the maximum problem size to what will fit in the memory on a single node. As such, this approach is only used when it is not possible to use the third approach.

3. Domain decomposition allows one to break large grids into a large number of smaller grids (normally called subdomains). Each of these subdomains is then worked on by a separate processor. This allows one to efficiently work on very large problem sizes, since the communication occurs primarily at the end of each time step (coarse-grained communication) and the amount of data being transferred is proportional to the surface area of each subdomain (*12*). For the remainder of this section, it will be assumed that this is the programming paradigm being used.

Even with domain decomposition, it is desirable to minimize the amount of communication. Ideally, the amount of communication should be kept to a size and structure that allow it to be fully overlapped with computation. This implies that, as the delivered performance (on a per-processor basis) increases, the amount of work assigned to each processor should also increase. In particular, if using a GPU as an attached processor will improve the overall per-processor performance of an application (ignoring communications costs) by a factor of 10 or more, then it is reasonable to assume that the amount of work assigned to each processor will consume most of the available memory. In fact, one might even want to increase the amount of memory each node is equipped with. Currently, most clusters are equipped with at least 512 MB of main memory per processor (1 GB on a dual-processor node).

As far back as 1995, the term "memory wall" was coined to describe the observation that the speed of the CPU was increasing faster than the speed of main memory (*13*). As a result, the delivered performance of today's microprocessors is highly dependent upon an application's

[*] Additional information on these can be found at http://upc.gwu.edu, http://www.co-array.org, http://www.emsl.gov/docs/nwchem /nwchem.html, and http://www.netlib.org/scalapack, respectively.

ability to “live” out of cache. In other words, unless there is a significant amount of data reuse in one or more levels of cache, there simply is an insufficient amount of memory bandwidth to keep the processor busy. Even with a reasonable level of data reuse, the memory latency associated with a cache miss missing all the way back to main memory would normally be considered excessive (hundreds of possibly even a few thousand, CPU cycles). This problem is normally addressed through some combination of stream buffers, nonblocking cache misses (allowing multiple misses to overlap their latencies), and/or pre-fetching (*14*).

If the memory wall is a problem for the CPU, then one can imagine the potential for it to be an even bigger problem when one is talking about any attached processor, with which one hopes to significantly accelerate the performance of an application. The good news is that most GPUs, and all high-end GPUs, are equipped with a significant amount of dedicated graphics memory (also sometimes referred to as video memory or GRAM). Additionally, this graphics memory is organized into multiple banks (*4*). As such, it supports multiple memory accesses per cycle, while the tight coupling between the GPU and the graphics memory keeps the latency low. The effective latency is reduced even further when the GPU streams data into and out of the graphics memory. All of this is reminiscent of vector processors, except there are no vector registers.

Now for the bad news: high-end GPUs are routinely equipped with 128 or 256 MB of graphics memory (older, less expensive, and/or GPUs meant for use in portable devices will usually have significantly less graphics memory, e.g., 16–64 MB). The problem is that since this is significantly smaller (by at least a factor of 2) than the size of per-processor main memory (when one is using GPUs for nongraphics applications), one must anticipate the need to stream data between main memory and the graphics memory. This data will move through the PCI-Express bus or one of the older buses that are totally inadequate for this task (e.g., AGP, PCI, or PCI-X). Although the PCI-Express bus provides significantly better bidirectional bandwidth and latency than its predecessors, it is still in no way capable of supplying the voracious appetite of a state-of-the-art GPU for data.

This demonstrates the importance of implementing the algorithm in a manner that supports a high level of data reuse in terms of the graphics memory. If the algorithm uses functions such as matrix multiply and fast Fourier transform (FFT), this should not be difficult. However, when one is using level 1 BLAS functions such as dot product, there is little or no potential for data reuse within a single call. In general, such a function would be considered poorly suited for use on a GPU. However, it is sometimes possible to group successive or concurrent calls to such functions in a manner that produces the necessary level of data reuse. Therefore, although this does not preclude the use of GPUs and other attached processors, it can represent a significant obstacle to the successful use of these devices for nongraphics applications. Clearly, one of the simplest solutions is to buy the GPUs with the largest amount of graphics memory possible. Some GPUs can be purchased with 512 MB of graphics memory, and at least one device supports more than 1 GB of graphics memory (*9*, *10*).

Anticipating the Speedup

What kind of speedup might one reasonably expect to see when using GPUs in a cluster computer? In this discussion, we make the following assumptions:

- A high end GPU is being used.
- The algorithm lends itself to the GPU and parallelization using MPI on the cluster.
- Implementing the GPU will maximize the amount of work allocated to each processor, possibly requiring the cluster to be equipped with additional memory per processor.
- The system interconnect used by MPI (e.g., Myrinet, Infiniband, or Quadrix) cannot easily be enhanced for clusters equipped with attached processors, such as a GPU.
- The CPU-only implementation and the GPU-augmented implementation represent best efforts.

For a specified problem, this means that in order to keep the time spent on communication in balance with the computation and therefore maintain the potential to fully overlap the two, one will be restricted to using fewer GPU-equipped nodes than were used with the conventional cluster solution. In other words, the predicted speedup would be $S_G/S_C \times N_G/N_C$ where:

- S_G is the speed of the application on a per-processor basis when run on the GPU,
- S_C is the speed of the application on a per-processor basis when run on the CPU,
- N_G is the number of nodes used when using the GPU (for a fixed total problem size), and
- N_C is the number of nodes used when using just the CPUs (for a fixed total problem size).

The subscripts G and C represent the GPU- and the CPU-only-based solutions, S is the speed per processor in some appropriate unit (e.g., GFLOPS or time steps per hour), and N are the number of GPUs or CPUs being used. As we have been explaining, $N_G < N_C$, and one might hope that S_G would be a factor of 10 or more times greater than S_C. Therefore, it would be reasonable to assume that the overall speedup will be at least a factor of 2 smaller than the per-node speedup. In reality, for programs that are highly successful at taking advantage of an attached processor, such as a GPU, one should investigate the options for improving the performance of the system interconnect (e.g., using a newer interconnect from the same vendor, a faster interconnect from a different vendor, or multiple rails of the same interconnect).

5. Software

According to Owens, “successful programming for any development platform requires at least three basic components: a high-level language for code development, a debugging environment, and profiling tools" (*3*). Because high-level GPU programming languages are designed with graphics as a central theme, the languages are often referred to as shading languages. As this nomenclature suggests, these languages compile a shader program into a vertex shader and a fragment shader to produce the image described by the shader program. Cg, HLSL, OpenGL are all such shading languages that allow the programmer to write GPU programs in a more familiar C-like programming environment. Sh is a shading language that offers a C++-like programming environment. All of these languages remain close to the specialized nature of GPUs and contain graphics-specific constructs, i.e., vertices, fragments, and textures. ASHLI operates one level higher than the aforementioned shading languages and reads as input shaders written in HLSL or OpenGL and "automatically compiles and partitions the input shaders to run on a programmable GPU" (*3*).

Currently, the two major standards for the GPU interface are OpenGL and DirectX. These standards were designed to program graphics operations; thus, it is not trivial to use them for nongraphical applications. Because GPU programming requires a programmer to view their GPGPU application in terms of geometric primitives, languages and libraries have been developed to provide GPGPU functionality while relieving the programmer of the GPU-specific details. BrookGPU is a programming language extension to the ANSI C standard that can use the GPU as a compilation target. BrookGPU affords the programmer advantages such as code reuse, as code can execute on the CPU or the GPU using the OpenGL or DirectX interfaces, and an indirection layer relieves the user of having to know whether the code is executing on an NVIDIA or an ATI chip (*15*). Scout is a GPU programming language designed for scientific visualization that "allows runtime mapping of mathematical operations over data sets for visualization" (*16*). Accelerator and Computer Graphics in Scientific programming (CGiS) have similar aims at simplifying GPU programming through high-level data-parallel implementations. The Glift template library simplifies GPU data structure design and separates GPU algorithms from data structures so that interfacing with CPU-based parallel data structures is possible. Several attributes of each of these programming tools are contained table 1. Significant efforts have been made to ameliorate the demanding task of GPU programming so that GPGPU developers can utilize the computational power of the GPU at a higher level of programming. Appendices A–G highlight examples of nongraphical paradigms implemented in several of the aforementioned programming languages.

A related issue is the limited amount of memory a GPU has for holding the shader program. This limitation is similar to a program running on the microprocessor. If the inner loop fits into the level 1 instruction cache, then it has a chance of running efficiently. However, with a GPU, if the shader program does not fit into the GPU's program memory, the shader program will not run at all. It is important to remember that the size of this memory is driven by the needs of graphics applications.

Debugging on GPUs was fairly limited until the recent surge of tools, yet the need for effective tools was much greater. The most commonly used debugging tool for programs running on a CPU, the print statement/printf function, has no counterpart on the GPU. It was established that because GPUs are now being used for general-purpose computing, a GPU debugger should be similar in capabilities to a CPU debugger. Variable watches, program break points, and single-step execution were deemed important features. Along with these standard features for CPU debuggers, an effective GPU debugger should include some other features, because of the interactive aspect of GPU programming. "The ideal GPGPU debugger would automate printf-style (the values of interest are printed to the screen) debugging, including programmable scale and bias for values outside the display, while also retaining the true data value at each point if it is needed" (*3*). Several tools have been developed for GPU debugging; however, nearly all are missing one or more of the aforementioned important features. gDEBugger and GLIntercept are tools for debugging OpenGL programs, the Microsoft shader debugger provides the functionality of runtime variable watches and breakpoints for shaders, and the Shadesmith fragment program debugger was the first debugger to implement the printf-style paradigm. Although there is still work to be done in the area of debugger development, the tools that currently exist have proven sufficient to validate vertex and fragment programs. As the GPGPU field emerges, the debugging tools will be challenged to become more robust.

Table 1. GPU programming tools.

Tool	Cost	Support	Developer	Platform(s)	Advantages	Applications
Accelerator (*17*)	Free	Open source	Microsoft Research	Platform-independent	High-level data parallel programming model in a library accessible to most programming languages	GPGPU programming; translates data-parallel operations on the fly to GPU pixel shaders
ASHLI (*18*)	Free	Open source	ATI Technologies	HLSL, OpenGL shading language code, or a subset of Renderman as input and compiles the shader to run on GPU	Provides a framework for mapping arbitrary complex shaders onto graphics shading hardware	Digital content creation; bridging the gap between low level shading constructs and shading description of programmer
Brahma (*19)*	Free	Open source	Brahma	NET 2.0	Eliminates the need for learning a shading language but gives same speed and performance; internally handles most of GPU programming	High-level graphical and general purpose GPU programming
Brook GPU (*15*)	Free	Open source	Stanford University's graphics group	DirectX (requires newer cards—ATI 9700 & above, NVIDIA 5200 and above); OpenGL (Windows & Linux)—better supports NVIDIA cards	Useful tool for GPGPU programmers; utilizes stream programming model for easy parallelization	General-purpose GPU programming; stream programming
Cg (*20*)	Free	Open source	NVIDIA	Windows, Linux, Mac, OpenGL and DirectX as APIs	Cg compiler can optimize code and do lower level tasks; familiar C-like programming language	Interactive effects into 3-D applications; shader programs
CGiS (*21*)	Free	Open source (upon completion of compiler frame work)	Saarland University Compiler Design Lab	Platform-independent	Raises GPU abstraction level	Scientific programming
CUDA (*22*)	—	Beta release	NVIDIA	OpenGL and Microsoft DirectX drivers from NVIDIA	Uses C to create programs called threads; CUDA technology processes thousands of threads simultaneously enabling a higher capacity of information flow	Data-intensive applications; physics computation (giving gamers great performance and visual effects)

Table 1. GPU programming tools (continued).

Tool	Cost	Support	Developer	Platform(s)	Advantages	Applications

Glift template (*23*) library	Free	Open source	Scientists from UC-Davis, Stanford, and University of Utah	Integrates with C++, Cg, and OpenGL development environment	Simplifies algorithmic development;
					code reuse;
					interchangeable data structures
						Defining complex, random-access GPU data structures

HLSL (*24*)	Free	Open source	Microsoft	Windows	Very similar to Cg; can use for vertex and fragment shading	High-level GPU vertex and fragment shader programming
Open GL shading language (*25*)	Free	Open source	3Dlabs	Platform-independent	Enables direct compilation of C-like programs to graphics hardware machine code	High-level shader programming; real-time cinematic quality graphics
Peak stream (*26*)	License required (limited time no-cost evaluation program)	Commercially available	Peakstream Inc.	OS: Linux 4.0 GPU: ATI R580-based graphics card	Cuts development time by up to 90% due to easy to learn API; integrates with existing developer tools (i.e. GCC, GDB, and Intel compilers)	Standard arithmetic and geometry; 1-D and 2-D stream arrays; matrix solver; single and double precision
Rapid mind (*27*)	Free	Beta release	RapidMind Inc.	RapidMind development platform	Allows developer to use standard C++ programming to easily create high-performance and massively parallel applications that run on the GPU	BLAS dense linear algebra operations; Fast Fourier Transform
Scout (*16*)	Free	Open source	Los Alamos National Laboratory	Platform-independent	Allows user to process multivariate data, express derived data, and define mappings to final image in a familiar environment	Scientific visualization; hardware acceleration
Sh (*28*)	Free	Open source	Michael McCool-University of Waterloo Computer Graphics Lab	For hardware acceleration, requires ATI Radeon 9600s and up	Portable; object-oriented programming; familiar syntax (built on top of C++)	High-level GPU programming

6. GPU Programming Model

While programming on the CPU requires a sequential programming model, the GPU achieves its superior performance through data parallelism, which employs the stream programming model. The stream programming model structures programs in a manner that affords high efficiency in computation and communication (*29*). It exploits the parallelism of the application by structuring the data into streams and performing computation on the streams with kernels. BrookGPU offers a stream programming system for GPUs (*3*). It implements stream programming concepts with streams as variables and kernels and reductions as functions that operate on streams; it automatically maps kernels and streams into fragment programs and texture memory.

Because typical scenes have more fragments than vertices, the highest volume of arithmetic computation is done in the fragment processing stage. A GPGPU program is structured as follows (*3*):

1. Initially, the programmer defines the data-parallel portions of the application and segments the application into independent parallel sections. Each of these sections can be considered a kernel and is implemented as a fragment program. The input and output of each kernel program is one or more data arrays, which are stored in textures in GPU memory. The data in these textures are considered as streams, and a kernel is invoked in parallel on each stream element.

2. By passing vertices to the GPU of geometric primitives (typically quadrilaterals) orientated parallel to the image plane and sized to match the desired size of the output array, the range of computation is defined, thus invoking a kernel.

3. The rasterizer generates a fragment for every pixel location in the primitive.

4. Each of the generated fragments is then processed by the active kernel fragment program. The fragment program can read from arbitrary global memory locations but can only write to memory locations corresponding to the fragment's location in the frame buffer.

5. The output of the fragment program is a value (or vector of values) per fragment. This output may be stored as a texture and used for the next pass through the pipeline or it may be the final result of the application.

7. GPGPU Applications

The performance advantages of GPUs have been explored within various applications, including physical based simulations, signal and image processing, geometric computing, and databases and data mining. The increased demand and deployment of GPUs in the last several years has resulted in increasing experimental research with graphics hardware. Several groups have used the GPU to successfully implement physically based simulations. The Center for High Performance Computing at Stony Brook University developed a parallel flow simulation using the Lattice-Boltzmann model (LBM) on a GPU cluster and simulated the dispersion of air-borne contaminants in the Times Square area of New York City. The numerical method employed by the LBM is highly parallelizable, which lends it well to computation on the GPU; moreover, the specification of boundary shapes is not governed by strict conditions. In order to implement this model, the LBM operations (e.g., streaming, collision, and boundary conditions) are formulated into fragment programs to be executed by the fragment processor during the rendering process. The fragment program fetches current state information from the appropriate textures (arrays), computes the LBM equations to evaluate the new states, and then passes the results to a pixel buffer. After the fragment pass is completed, the results are copied back to textures for use in the next step. This group was able to produce results using 30 GPU nodes that were 4.6× faster than the same implementation of a 480 × 400 × 80 LBM on a CPU cluster that contained 64 Pentium Xeon 2.4-GHz processors and 2.5-GB memory (*2*). The demanding applications of signal and image processing benefit from the high computational rates of the GPU. Image segmentation has been a prominent research area within these applications and GPGPU segmentation approaches have provided speedups of more than 10× by coupling the fast computation of the GPU to an interactive volume renderer. Database and data mining algorithms are known to be highly computation and memory intensive, which makes them attractive candidates for GPU computation. The high memory bandwidth has accelerated the performance of many essential database queries—Govindaraju et al. (*30*) compared the performance of SQL queries on an NVIDIA GeForce 6800 against a 2.8-GHz Intel Xeon processor, and these comparisons indicate as much as an order of magnitude improvement for the GPU over a SIMD-optimized CPU implementation. As GPUs evolve and become better equipped for general-purpose computation, the success of their use within applications continues to rely on the developer's understanding of the circumstances under which the GPU is likely to outperform the CPU. Goodnight notes that "efficient GPGPU applications almost always take advantage of the vector processing and memory access capabilities of the GPU" (*8*).

8. Results

Table 2 summarizes the results discussed in section 7 of this report and several other examples of code that were ported to GPUs. Unfortunately, there do not appear to be many examples of applications being run on GPU-equipped clusters. In table 2, we have attempted to provide opportunities to make the following types of comparisons:

- The same problem run on a CPU or cluster of CPUs vs. on a GPU or GPU-equipped clusters
- Performance results obtained when using different languages
- Performance as a function of problem size

It is interesting to note that, frequently, the delivered performance on a GPU can be heavily influenced by the problem size. In general, the larger the problem, the better the performance. In many cases, for smaller problem sizes, using the GPU can actually result in a slower run. Additionally, it should be pointed out that the optimal algorithm for the GPU is frequently not the algorithm of choice for the CPU. One final note to point out is that we have made no attempt to compare the performance of competing GPUs.

9. Conclusion

As general-purpose computation on graphics hardware continues to grow as a research area, the research community is continually being challenged to think about non-graphics problems from a graphics perspective and attempt to effectively design algorithms that are well suited for graphics architecture. Simultaneously, GPU vendors are expected to increase programmability and generality of future generations of GPUs without sacrificing the specialized architecture and heightened performance that have compelled its surge into the non-graphics research arena. As the field of GPGPU computing matures, researchers are hopeful that the GPU continues to be a sturdy platform for enhanced computation and performance.

Table 2. Application performance CPU-based vs. GPU-accelerated implementations.

	CPU Performance				**GPU Performance**						
Benchmark /Application	**Type**	**Clock Rate (GHz)**	**No. of Processors**	**Measured Performance**	**Type**	**Clock Rate (MHz)**	**No. of Processors**	**Measured Performance**	**Speedup**	**Reference**	**Language**
SQL queries (multi-attribute query 360-K records)	Intel Xeon	2.8	1	8 ms	NVIDIA GeForce 6800	450	1	4 ms	2	(*30*)	OpenGL
SQL queries (multi-attribute query 1-M records)	Intel Xeon	2.8	1	18 ms	NVIDIA GeForce 6800	450	1	12 ms	1.5	(*30*)	OpenGL
SQL queries (semi-linear queries 360-K records)	Intel Xeon	2.8	1	15 ms	NVIDIA GeForce 6800	450	1	2 ms	7.5	(*30*)	OpenGL
SQL queries (semi-linear queries 1-M records)	Intel Xeon	2.8	1	40 ms	NVIDIA GeForce 6800	450	1	5 ms	8	(*30*)	OpenGL
LBM	Pentium Xeon	2.4	64	1.44 second step	NVIDIA GeForce FX 5800 Ultra	500	30	0.312 second /step	4.6	(*2*)	Cg
SGEMM (single precision matrix multiply)	—	—	—	—	—	—	—	94 GFlops	—	(*31*)	Peak-stream
SGEMM (single precision matrix multiply)	—	—	—	—	—	—	—	32 GFlops	—	(*31*)	Rapid-mind
SGEMM (single precision matrix multiply)	—	—	—	—	—	—	—	15 GFlops	—	(*31*)	Brook
SGEMM (single precision matrix multiply)	—	—	—	—	—	—	—	7 GFlops	—	(*31*)	Accelerator
16-bit data std: sort (256 × 256 field size)	Pentium 4	3	1	82.5 full sorts/s	—	—	—	—		(*29*)	—
16-bit data std: sort (512 × 512 field size)	Pentium 4	3	1	20.6 full sorts/s	—	—	—	—	—	(*29*)	—
16-bit data std: sort (1024 × 1024 field size)	Pentium 4	3	1	4.7 full sorts/s	—	—	—	—	—	(*29*)	—

Table 2. Application performance CPU-based vs. GPU-accelerated implementations (continued).

Benchmark /Application	**CPU Performance**				**GPU Performance**				**Speedup**	**Reference**	**Language**
	Type	**Clock Rate (GHz)**	**No. of Processors**	**Measured Performance**	**Type**	**Clock Rate (MHz)**	**No. of Processors**	**Measured Performance**			
Bitonic merge sort (16-bit float data) (256 × 256 field size)	—	—	—	—	NVIDIA GeForce 6800 Ultra	425	1	90.07 full sorts/s	1.09	(*29*)	GLSL
Bitonic merge sort (16-bit float data) (512 × 512 field size)	—	—	—	—	NVIDIA GeForce 6800 Ultra	425	1	18.3 full sorts/s	0.89	(29)	GLSL
Bitonic merge Sort (16-bit float data) (1024 × 1024 field size)	—	—	—	—	NVIDIA GeForce 6800 Ultra	425	1	3.6 full sorts/s	0.77	(29)	GLSL
2-D complex FFT (256 × 256)	AMD Opteron	2.6	1	2.8 GFLOPS	NVIDIA 7900 GTX	—	1	1.5 GFLOPS	0.54	(*32*)	Rapid mind
2-D complex FFT (1024 × 1024)	AMD Opteron	2.6	1	2.5 GFLOPS	NVIDIA 7900 GTX	—	1	7 GFLOPS	2.8	(*32*)	Rapid mind
Option pricing (Black-Scholes 64-K options)	AMD Opteron	2.6	1	9-m options/s	NVIDIA 7900 GTX	—	1	20-m options/s	2.2	(*32*)	Rapid mind
Options pricing (Black-Scholes 1-M options)	AMD Opteron	2.6	1	9-m options/s	NVIDIA 7900 GTX	—	1	200-m options/s	22.2	(*32*)	Rapid mind
SAXPY	Pentium 4	3	1	—	NVIDIA 7800 GTX	—	1	—	8.4	(*33*)	Brook
Segment	Pentium 4	3	1	—	NVIDIA 7800 GTX	—	1	—	4.0	(*33*)	Brook
SGEMV	Pentium 4	3	1	—	NVIDIA 7800 GTX	—	1	—	3.3	(*33*)	Brook
FFT	Pentium 4	3	1	—	NVIDIA 7800 GTX	—	1	—	2.0	(*33*)	Brook
Ray	Pentium 4	3	1	—	NVIDIA 7800 GTX	—	1	—	2.6	(*33*)	Brook

10. References

1. Stanford Folding at Home Website. http://folding.stanford.edu (accessed 30 March 2007).

2. Fan, Z.; Qiu, F.; Kaufman, A.; Yoakum-Stover, S. GPU Cluster for High Performance Computing. *ACM/IEEE Supercomputing Conference* **2004**, 47.

3. Owens, J.; Luebke, D.; Govindaraju, N.; Harris, M.; Kruger, J.; Lefohn, A.; Purcell, T. A. Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum* **2007**, *27*, 80–113.

4. Luebke, D.; Humphreys, G. How GPUs Work. *IEEE Computer* **2007**, *40*, 96–100.

5. Bolstad, M. U.S. Army Research Laboratory, Aberdeen Proving Ground, MD. Private communications, **2007**.

6. Macedonia, M. The GPU Enters Computing's Mainstream. *Computer* **2003**, *36*, 106–108.

7. General-Purpose Computation Using Graphics Hardware Home Page. http://www.gpgpu.org (accessed 30 March 2007).

8. Goodnight, N.; Wang, R.; Humphreys, G. Computation on Programmable Graphics Hardware. *IEEE Computer Graphics and Applications* **2005**, *25* (5), 12–15.

9. NVIDIA Developer Home Page. http://developer.nvidia.com/page/home.html (accessed 30 March 2007).

10. ATI Technologies Inc. Home Page. http://www.ati.com (accessed 30 March 2007).

11. Trancoso, P.; Charalambous, M. Exploring Graphics Processor Performance for General Purpose Applications. *Proceedings of the 2005 8th Euromicro Conference on Digital System Design*, Porto, Portugal, 30 August–3 September 2005.

12. Culler, D. E.; Jaswinder P. S.; Anoop G. *Parallel Computer Architecture: A Hardware /Software Approach*. Morgan Kaufmann Publishers, Inc.: San Francisco, CA, 1999; 131–135.

13. Wulf, W. A.; McKee, S. A. Hitting the Memory Wall: Implications of the Obvious. *Computer Architecture News, Association for Computing Machinery* **1995**, *23* (1), 20–24.

14. Pressel, D. M. Fundamental Limitations on the Use of Prefetching and Stream Buffers for Scientific Applications. *Proceedings of the 16th ACM Symposium on Applied Computing, Association for Computing Machinery*, Las Vegas, NV, 2001.

15. BrookGPU Home Page. http://graphics.stanford.edu/projects/brookgpu/ (accessed 30 March 2007.

16. McCormick, P.; Inman, J.; Ahren, J.; Hansen, C.; Roth, G. Scout: A Hardware-Accelerated System for Quantitatively Driven Visualization and Analysis. *IEEE Visualization* **2004**, 171–178.

17. Tarditi, D.; Puri, S.; Oglesby, J. Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses. *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems,* San Jose, CA, 21–25 October 2006.

18. ASHLI. http://ati.amd.com/developer/ashli.html (accessed 30 March 2007).

19. Brahma. http://brahma.ananthonline.net (accessed 30 March 2007).

20. Cg. http://developer.nvidia.com/page/cg_main.html (accessed 30 March 2007).

21. Lucas, P.; Fritz, N.; Wilhelm, R. The Development of the Data-Parallel GPU Programming Language CGIS. *Proceedings of ICCS* **2006**, *3994*, 200–203.

22. CUDA. http://developer.nvidia.com/object/cuda.html (accessed 30 March 2007).

23. Glift Template Library. http://www.idav.ucdavis.edu/projects/glift (accessed 30 March 2007).

24. HLSL. http://ati.amd.com/developer/ShaderX2_IntroductionToHLSL.pdf (accessed 30 March 2007).

25. OpenGL Shading Language. http://www.opengl.org/documentation/glsl/ (accessed 30 March 2007).

26. Peakstream. http://www.peakstreaminc.com/reference/PeakStream_datasheet.pdf (accessed 30 March 2007).

27. RapidMind. http://www.rapidmind.net/sc06_hp_rapidmind_cpugpu_summary.php (accessed 30 March 2007).

28. Sh. http://libsh.org/ (accessed 30 March 2007).

29. Pharr, M., Ed.. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (NVIDIA)*; Addison-Wesley: Upper Saddle River, NJ, 2005; 457–470.

30. Govindaraju, N. K.; Lloyd, B.; Wang, W.; Lin, M.; Manocha, D. Fast Computation of Database Operations Using Graphics Processors. *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, Los Angeles, CA, 2004; 215–226.

31. Papakipos, M. Stream Programming on the PeakStream Platform. http://www.gpgpu.org/sc2006/workshop/presentations /PeakStream_SC06.pdf (accessed 30 March 2007).

32. McCool, M.; Wadleigh, K.; Henderson, B.; Lin, H.-Y. Performance Evaluation of GPUs Using the RapidMind Development Platform. http://www.rapidmind.net/pdfs/RapidMind GPU.pdf (accessed 30 March 2007).

33. Buck, I. High Level Languages for GPUs. Presented as part of the tutorial on GPGPUs at VIS05. http://www.gpgpu.org/vis2005/PDFs/gpgpu/viscourse05 .pdf (slide 164) (accessed 30 March 2007).

Appendix A. Code Examples for BrookGPU

```
kernel  void mul (float a<>, float b<>,
                    out float result<>) {
   result = a*b;
}

reduce void sum (float a<<>,
                   reduce float result<>) {
   result += a;
}

float matrix <20,10>;
float vector <1, 10>;
float tempmv <20,20>;
float result <20, 1>;

mul (matrix,vector,tempmv);
sum (tempmv,result);
```

Figure A-1. Matrix vector multiply implemented in BrookGPU.[1]

```
kernel void k(float s<>, float3 f, float a[10][10], out float o<>);

  float a<100>;
  float b<100>;
  float c<10,10>;

  streamRead(a, data1);
  streamRead(b, data2);
  streamRead(c, data3);

  // Call kernel "k"
  k(a, 3.2f, c, b);

  streamWrite(b, result);
```

Figure A-2. BrookGPU kernel definition.[2]

[1] Buck, I. High Level Languages for GPUs. http://www.gpgpu.org/vis2005/PDFs/gpgpu/viscourse05.pdf (slides 158-–159) (accessed 30 March 2007).

[2] BrookGPU. http://graphics.stanford.edu/projects/brookgpu/ (accessed 30 March 2007).

Appendix B. Code Programming Example for Cg

for (int j = 1 ; j < height - 1 ; ++j)
{
   for (int i = 1; i < width - 1; ++i)
   {	

// get velocity at this cell
Vec2f v = grid(x, y);
 	
// trace backwards along velocity field
float x = (i - (v.x * timestep / dx));
float y = (j - (v.y * timestep / dy));
 	
grid (x, y) = grid.bilerp (x, y);
	void advect (float2 uv : WPOS,
	        out float4 xNew : COLOR,
	
	    uniform float dt, // timestep
	    uniform float dx, // grid scale
	    uniform samplerRECT u, // velocity
	    uniform samplerRECT x, // state
	{
	  // trace backwards along velocity field
	  Float2 pos = ub - dt * f2texRECT (u, uv) / dx;
	
	  xNew = f4texRECTbilerp (x, pos);
	}

} }	
C++	**Cg**

Figure B-1. Transforming a section of code for performing an Advect from C++ to Cg.[1]

Note: For those who are not familiar with the Black-Scholes model, it is the most frequently used method for estimating what the price of an option should be. To the extent that the actual price of an option varies from the estimated price, one might want to either buy or sell the option.

[1] Luebke, D. Introduction to GPGPU Programming. http://www.gpgpu.org/sc2006/slides/01.luebke.Introduction.pdf (slide 35) (accessed 30 March 2007).

```
float BlackScholesCall (float S, float X, float T, float r, float v) {
  float d1 = (log(S/X) + (r + V * V * .5f) * T) / (v * sqrt(T));
  float d2 = d1 - v * sqrt(T);
  return S * CND(d1) - X * exp(-r * T) * CND(d2);
}
```

Figure B-2. Implementing the Black-Scholes model in Cg.[2]

```
float CND(float X)
{
   float L= abs(X);
   // Set up float4 so that K.x = K, K.y = K^2, K.z = K^3, K.w = K^4
   float4 K;
   K.x = 1.0 / (.0 + 0.2316419 * L);
   K.y = K.x * K.x;
   K.zw = K.xy * K.yy;

   // compute K, K^2, K^3, and K^4 terms, reordered for efficient
   // vectorization.  Above, we precomputed the K powers, here we'll
   // multiply each one by its corresponding scale and sum up the
   // terms efficiently with the dot() routine.
   //
   // dot (float4(a, b, c, d), float4(e, f, g, h)) efficiently computes
   // the inner product a*e + b*f + c*g +d*h, making much better
   // use of the 4-way vector floating-point hardware than a
   // straightforward implementation would.
   float w = dot(float4 (0.31938153f, -0.356563782f,
                         1.781477937f, -1.821255978f), K);
   // and add in the K^5 term on its own
   w += 1.330274429f * K.w * K.x;
   w *= rsqrt(2.f * PI) * exp(-L * L * .5f);  // rsqrt() == 1/sqrt()

   if (X > 0)
      w = 1.0 - w;
   return w;
}
```

Figure B-3. Implementing the cumulative normal distribution function.

[2] Kolb, C.; Pharr, M. Options Pricing on the GPU. In *GPU Gems 2*; Pharr, M., Ed.; Addison-Wesley: Boston, MA, 2005; 722–724.

INTENTIONALLY LEFT BLANK.

Appendix C. Code Programming Example for GLSL

```
uniform sampler2D  PackedData;

// contents of the texcoord data
#define OwnPos gl_TexCoord[0].xy
#define SearchDir gl_TexCoord[0].z
#define CompOp gl_TexCoord[0].w
#define Distance gl_TexCoord[1].x
#define Stride gl_TexCoord[1].y
#define Height gl_TexCoord[1].z
#define HalfStrideMHalf gl_TexCoord[1].w

void main(void)
{
  // get self
  vec4 self = texture2D (PackedData, OwnPos);

  // restore sign of search direction and assemble vector to partner
  vec2 adr = vec2( (SearchDir < 0.0) ? -Distance : Distance , 0.0);

  // get the partner
  vec4 partner = texture2D(PackedData, OwnPos + adr);

  // switch ascending/descending sort for every other row
  // by modifying comparison flag
  float compare = CompOp * -(mod(floor(gl_TexCoord[0].y * Height),
                               Stride) - HalfStrideMHalf);

  // x and y are the keys of the two items
  // → multiply with comparison flag
  vec4 keys = compare * vec4( self.x, self.y, partner.x, partner.y);

  // compare the keys and store accordingly
  // z and w are the indices
  // → just copy them accordingly
  vec4 result;
  result.xz = (keys.x < keys.z) ? self.xz : partner.xz;
  result.yw = (keys.y < keys.w) ? self.yw : partner.yw;

  // do pass 0
  compare *= adr.x;
  gl_FragColor =
      (result.x * compare < result.y * compare) ? result : result.yxwz;
}
```

Figure C-1. GLSL Fragment program implementing the combined passes 1 and 0 for row-wise sorting of the bitonic merge sort.[1]

[1] Kipfer, P.; Westermann, R. Improved GPU Sorting. *In GPU Gems 2*; Pharr, M., Ed.; Addison-Wesley: Boston, MA, 2005; 744.

Appendix D. Code Programming Example for PeakStream

```
#include <peakstream.h>

#define NSET 1000000                   //  number of monte carlo trials

Arrayf32 Pi = compute_pi();            //  get the answer as a 1x1 array
float_pi = Pi.read_scalar();           //  convert answer to a simple float
printf(“Value of Pi = %f\n”, pi);

Arrayf32
Compute_pi (void)
{
    RNGf32 G(SP_RNG_DEFAULT, 271828);  // create an RNG
    Arrayf32 X = rng_uniform_make(G, NSET, 1, 0.0, 1.0);
    Arrayf32 Y = rng_uniform_make(G, NSET, 1, 0.0, 1.0);
    Arrayf32 distance_from_zero = sqrt (x * X + Y * Y);
    Arrayf32 inside_circle = (distance_from_zero <= 1.0f) ;
    Return 4.0f * sum(inside_circle) / NSET ;
}
```

Figure D-1. Computing PI with PeakStream.[1]

[1] Papakipos, M. Stream Programming on the PeakStream Platform. http://www.gpgpu.org/sc2006/workshop /Presentations/PeakStream_SC06.pdf (slide 15) (accessed 30 March 2007).

Appendix E. Code Programming Example for Scout

```
float:shapeof(temp) new_temp: //time step result goes here …
// Data parallel computation of the diffusion…
compute with(shapeof(temp)) {
 // Don’t compute over boundary conditions …
 where (mask > 0) {
  float temp_x;
  temp_x =(alpha/(dx*dx))*(temp[i+1][j]-2*temp[i][j]+
                temp[i-1][j]);
  temp_y =(alpha/(dy*dy))*(temp[i][j+1]-2*temp[i][j]+
                temp[i][j-1];
  new_temp = dt * (temp_x + temp_y) + temp[i][j];
 } else {
  new_temp = temp; // boundary conditions stay constant…
 }
}
// New temperatures need to become our initial conditions for
// the next pass.
temp = new_temp;
```

Figure E-1. Heat diffusion implemented in Scout.[1]

[1] McCormick, P. Scout: Case Studies. http://www.gpgpu.org/vis2005/PDFs/gpgpu/viscourse05.pdf (slide 490) (accessed 30 March 2007).

Appendix F. Code Programming Example for CGiS

```
PROGRAM viswave;
INTERFACE
extern inout float LAST<_,_> : texture (1) A; // _ is a size wildcard.
extern in float CURRENT<_,_> : texture (2) A; // Flipped on each step.
extern in float RINDEX, DAMP, WID, HEI; // Pass as program parameters.
intern float X<_,_> : texture (4) R; // These two streams shall reside
intern float Y<_,_> : texture (4) G; //
in the same texture (id=4).
extern in float3 TEXTURE<_,_>: texture (3) RGB; // Use RGB components
extern out float3 IMAGE<_,_> : texture (5) RGB; // for visualization.
CODE
... // Declare kernels called from this section and from CONTROL.
CONTROL
// Single step wave propagation:
forall (float last in LAST; float current in CURRENT){
propagate (last, current, indexX(last), indexY(last), DAMP, WID, HEI);
}
// Compute refractions in X- and Y-dimension:
forall (float x in X; float y in Y; float height in LAST){
refractionX (RINDEX, x, height, indexX(height), WID);
refractionY (RINDEX, y, height, indexY(height), HEI);
}
// Compute refracted image:
forall (float3 pixel in IMAGE; float height in LAST;
float x in X; float y in Y){
render (TEXTURE, pixel, height, x, y);
}
// Display image on screen:
show(IMAGE);
```

Figure F-1. Part of a CGiS program for calculating refractions.[1]

[1] Lucas, P.; Fritz, N.; Wilhelm, R. The Development of the Data-Parallel GPU Programming Language CGiS. *Proceedings of ICCS (2006)*, Reading, UK, 28–31 May 2006; 3994, 200–203.

Appendix G. Code Programming Example for Accelerator

```
using Microsoft.Research.DataParallelArrays;

static float[,] Blur(float[,] array, float[] kernel)
{
     float[,] result;
     DFPA parallelArray = new DFPA(array);

     FPA resultX  = new FPA(0f, parallelArray.Shape);
     for (int i=0; i<kernel.Length; i++) {
        int[] shiftDir = new int[] {0,i};
        resultX += PA.Shift(parallelArray, shiftDir) * kernel[i];
     }

     FPA resultY = new FPA(0f, parallelArray.Shape);
     for (int i=0; i<kernel.Length; i+=) {
        int[] shiftDir = new int[] {i,0};
        resultY += PA.Shift(resultX, shiftDir) * kernel[i];
     }

     PA.ToArray(resultY, out result);
     parallelArray.Dispose();
     return result;
}
```

Figure G-1. A 2-D convolution implementation using C# version of Accelerator.[1]

[1] Tarditi, D.; Puri, S.; Oglesby, J. Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses. *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, 21–25 October 2006.

Glossary

AGP A bus designed by Intel specifically for connecting graphics processors to the CPU and main memory. Unfortunately, its characteristics make it poorly suited for the needs of general-purpose programming using GPUs.

ASHLI Advanced shading language – one of several languages designed to facilitate the programming of GPUs.

ANSI C The C standard as adopted by the American National Standards Institute. Supersedes the informal standard version of C known as Kernighan and Ritchie C.

ATI The name of one of the two main companies that manufacture GPUs (recently acquired by Advanced Micro Devices, AMD).

BLAS A commonly used library of linear algebra subroutines that run on a single processor, although parallelized implementations have been created under such names as PBLAS and SCALAPACK.

BrookGPU One of several languages designed to facilitate programming GPUs.

CAF Co-array Fortran – an example of a language that conforms to the PGAS programming model.

CFD Computational fluid dynamics

CG C for graphics – one of several languages designed to facilitate programming GPUs.

CGiS Computer Graphics in Scientific programming

CPU Central processing unit

FFT Fast Fourier transform

GPGPU General-purpose computation on GPUs

GPU Graphics processing unit

GRAM Graphics RAM – a high-speed memory optimized for the requirements of processing graphical data.

HLSL High-level shading language – one of several languages designed to facilitate programming GPUs.

HPC High-performance computing

MADD Multiply-add instruction – allows a single instruction to specify two floating-point operations. This is one of two commonly used techniques to effectively double the theoretical peak speed of a processor without increasing the maximum number of instructions that can be started in a single cycle.

Moore’s Law The observation that the number of transistors on a chip was roughly doubling every 12–24 months. This was taken to infer that the performance of the chips was increasing at a similar rate. Unfortunately, it became clear that it was desirable to use this transistor budget in other ways. Originally, this meant moving units such as the floating-point unit, memory management unit (especially the TLB), and the L2 cache on chip. More recently, system designers have opted to put multiple processors on a single chip (in some cases at a lower clock rate).

MPI Message passing interface – the most commonly used message passing library.

NWCHEM A computational chemistry program originally developed at the Department of Energy’s Pacific Northwest Laboratories.

NVIDIA The name of one of the two main companies that manufacture GPUs.

PCI/PCI-X/PCI-Express An evolving set of standards for connecting peripheral components to the CPU and main memory.

Printf function A C function that implements formatted output in a manner similar to that of the Print statement in Fortran.

PGAS Partitioned global address space – an emerging programming model for writing parallel programs.

RGBA Red green blue alpha – texture data for the GPU.

SCALAPACK The name of a popular parallelized mathematics library for manipulating dense matrices.

Sh One of several languages designed to facilitate programming GPUs.

SIMD Single instruction, multiple data – a programming model in which multiple functional units execute the identical instruction/group of instructions on a set of data elements.

SQL Standard query language

UPC Unified parallel C – an example of a language which conforms to the PGAS programming model.

Z-Buffers A buffer used to accumulate the color information for each pixel on the screen. What makes this buffer special is that in addition to color information, it also includes the depth of the frontmost item visible in each pixel. Then, as the image is assembled and data is written into the buffer, the depth of each item is compared to what is already in the Z-Buffer to determine which item is frontmost and, therefore, visible. Only the visible items will be kept.

NO. OF
COPIES ORGANIZATION

1 (PDF ONLY)
DEFENSE TECHNICAL
INFORMATION CTR
DTIC OCA
8725 JOHN J KINGMAN RD
STE 0944
FORT BELVOIR VA 22060-6218

1
US ARMY RSRCH DEV &
ENGRG CMD
SYSTEMS OF SYSTEMS
INTEGRATION
AMSRD SS T
6000 6TH ST STE 100
FORT BELVOIR VA 22060-5608

1
DIRECTOR
US ARMY RESEARCH LAB
IMNE ALC IMS
2800 POWDER MILL RD
ADELPHI MD 20783-1197

3
DIRECTOR
US ARMY RESEARCH LAB
AMSRD ARL CI OK TL
2800 POWDER MILL RD
ADELPHI MD 20783-1197

ABERDEEN PROVING GROUND

1
DIR USARL
AMSRD ARL CI OK TP (BLDG 4600)

NO. OF
COPIES ORGANIZATION

1 C HENRY
PRGM DIR
1010 N GLEBE RD STE 510
ARLINGTON VA 22201

1 L DAVIS
DPUTY PRGM DIR
1010 N GLEBE RD STE 510
ARLINGTON VA 22201

1 B COMMES
HPC CTRS PRJCT MGR
1010 N GLEBE RD STE 510
ARLINGTON VA 22201

1 D POST
CHF SCNTST
1010 N GLEBE RD STE 510
ARLINGTON VA 22201

1 DIRECTOR USARL
AMSRD ARL CI
ADELPHI LAB CTR
J W GOWENS II
BLDG 205 RM 3A012C
ADELPHI MD 20783-1197

1 J OSBURN
CODE 5594
BLDG A49 RM 15
4555 OVERLOOK RD
WASHINGTON DC 20375-5340

1 AIR FORCE RSRCH LAB
MTRLS AND MFG DIRCTRT
R PACHTER
AFRL MLPJ 3005 HOBSON WAY
BLDG 651 RM 189
WRIGHT PATTERSON AFB OH
45433-7702

1 AIR FORCE RSRCH LAB
K HILL
AFRL SNS
BLDG 254 2591 K ST
WRIGHT PATTERSON AFB OH
45433-7602

1 AFRL IF
R W LINDERMAN
525 BROOKS RD
ROME NY 13441-4505

NO. OF
COPIES ORGANIZATION

1 US ARMY RSRCH DEV &
ENGRG CMD
AEROFLIGHT DYNAMICS DIRCTRT
AMES RSRCH CNTR MS T27B 1
R MEAKIN
MOFFETT FIELD CA 94035-1000

1 ARMY RSRCH OFC
AMSRD ARL RO EN
A M RAJENDRAN
PO BOX 12211
RESEARCH TRIANGLE PARK NC
27709-2211

1 NAVAL OCEANOGRAPHIC OFC
OFC OF THE TECHNICAL
DIRECTOR
J M HARDING
CODE OTT
STENNIS SPACE CENTER MS 39529

1 INFORMATION TECHNOLOGY
LAB
US ARMY ENGINEER RSRCH
AND DEVELOPMENT CTR
D R RICHARDS
VICKSBURG MS 39810

1 SPAWAR SYSTEMS CTR
C B PETERS
BLDG 606 RM 318
53360 HULL ST
SAN DIEGO CA 92152

1 ARNOLD ENGRG DEV CTR
C R VINING
1099 SCHRIEVER AVE STE E205
ARNOLD AIR FORCE BASE TN
37389

1 AIR FORCE RSRCH LAB
SENSORS DIRCTRT
T A WILSON
2241 AVIONICS CIR
WRIGHT PATTERSON AFB OH
45433

NO. OF
COPIES ORGANIZATION

1 US ARMY RSRCH AND
DEV CTR
NVL CMND CNTRL AND OCEAN
SURVEILLANCE CTR
HPC COORDINATOR & DIRECTOR
DOD DISTRIBUTED CENTER
NCCOSC RDTE DIV D3603
L PARNELL
49590 LASSING ROAD
SAN DIEGO CA 92152-6148

1 ASSOCIATE DIR
INNOVATIVE COMPUTING LAB
COMPUTER SCIENCE DEPT
UNIV OF TENNESSEE
S MOORE
1122 VOLUNTEER BLVD STE 203
KNOXVILLE TN 37996-3450

2 EXEC DIR LS SCAMP
SOUTH CAROLINA STATE UNIV
S ALLEY
J GUYDON
300 COLLEGE ST NE
PO BOX 7212
ORANGEBURG SC 29117

1 SHA’KIA D BOGGAN
1528 STAFFORD ST EXT
MONROE NC 28110

ABERDEEN PROVING GROUND

18 DIR USARL
AMSRD ARL CI H
C NIETUBICZ
AMSRD ARL CI CB
M LEE
D PRESSEL
R NAMBURU
R VALISETTY
D SHIRES
P CHUNG
J CLARKE
C ZOLTANI
S J PARK
AMSRD ARL CI HS
BROWN
K SMITH
T KENDALL

NO. OF
COPIES ORGANIZATION

AMSRD ARL CI HM
R PRABHAKARAN
P MATTHEWS
AMSRD ARL WM BC
J SAHU
K HEAVEY
P WEINACHT