INVITED PAPER

GPU Computing

Graphics Processing Units—powerful, programmable, and highly parallel—are increasingly targeting general-purpose computing applications.

By John D. Owens, Mike Houston, David Luebke, Simon Green, John E. Stone, and James C. Phillips

ABSTRACT | The graphics processing unit (GPU) has become an integral part of today's mainstream computing systems. Over the past six years, there has been a marked increase in the performance and capabilities of GPUs. The modern GPU is not only a powerful graphics engine but also a highly parallel programmable processor featuring peak arithmetic and memory bandwidth that substantially outpaces its CPU counterpart. The GPU's rapid increase in both programmability and capability has spawned a research community that has successfully mapped a broad range of computationally demanding, complex problems to the GPU. This effort in *general-purpose computing on the GPU*, also known as *GPU computing*, has positioned the GPU as a compelling alternative to traditional microprocessors in high-performance computer systems of the future. We describe the background, hardware, and programming model for GPU computing, summarize the state of the art in tools and techniques, and present four GPU computing successes in game physics and computational biophysics that deliver order-of-magnitude performance gains over optimized CPU applications.

KEYWORDS | General-purpose computing on the graphics processing unit (GPGPU); GPU computing; parallel computing

Manuscript received May 11, 2007; revised October 21, 2007 and January 2008. The work of J. Owens was supported by the U.S. Department of Energy under Early Career Principal Investigator Award DE-FG02-04ER25609, the National Science Foundation under Award 0541448, the SciDAC Institute for Ultrascale Visualization, and Los Alamos National Laboratory. The work of M. Houston was supported by the Intel Foundation Ph.D. Fellowship Program, the U.S. Department of Energy, AMD, and ATI.

J. D. Owens is with the Department of Electrical and Computer Engineering, University of California, Davis, CA 95616 USA (e-mail: jowens@ece.ucdavis.edu).
M. Houston is with the Department of Computer Science, Stanford University, Stanford, CA 94305 USA (e-mail: Michael.Houston@amd.com).
D. Luebke and **S. Green** are with NVIDIA Corporation, Santa Clara, CA 95050 USA (e-mail: dave@luebke.us; sgreen@nvidia.com).
J. E. Stone and **J. C. Phillips** are with the Beckman Institute for Advanced Science and Technology, University of Illinois at Urbana-Champaign, Urbana, IL 61801 USA (e-mail: johns@ks.uiuc.edu; jim@ks.uiuc.edu).

Digital Object Identifier: 10.1109/JPROC.2008.917757

I. INTRODUCTION

Parallelism is the future of computing. Future microprocessor development efforts will continue to concentrate on adding cores rather than increasing single-thread performance. One example of this trend, the heterogeneous nine-core Cell broadband engine, is the main processor in the Sony Playstation 3 and has also attracted substantial interest from the scientific computing community. Similarly, the highly parallel graphics processing unit (GPU) is rapidly gaining maturity as a powerful engine for computationally demanding applications. The GPU's performance and potential offer a great deal of promise for future computing systems, yet the architecture and programming model of the GPU are markedly different than most other commodity single-chip processors.

The GPU is designed for a particular class of applications with the following characteristics. Over the past few years, a growing community has identified other applications with similar characteristics and successfully mapped these applications onto the GPU.

- *Computational requirements are large.* Real-time rendering requires billions of pixels per second, and each pixel requires hundreds or more operations. GPUs must deliver an enormous amount of compute performance to satisfy the demand of complex real-time applications.
- *Parallelism is substantial.* Fortunately, the graphics pipeline is well suited for parallelism. Operations on vertices and fragments are well matched to fine-grained closely coupled programmable parallel compute units, which in turn are applicable to many other computational domains.
- *Throughput is more important than latency.* GPU implementations of the graphics pipeline prioritize throughput over latency. The human visual system operates on millisecond time scales, while operations within a modern processor take nanoseconds. This six-order-of-magnitude gap means that the latency of any individual operation is unimportant. As a consequence, the graphics pipeline is quite

deep, perhaps hundreds to thousands of cycles, with thousands of primitives in flight at any given time. The pipeline is also feed-forward, removing the penalty of control hazards, further allowing optimal throughput of primitives through the pipeline. This emphasis on throughput is characteristic of applications in other areas as well.

Six years ago, the GPU was a fixed-function processor, built around the graphics pipeline, that excelled at three-dimensional (3-D) graphics but little else. Since that time, the GPU has evolved into a powerful programmable processor, with both application programming interface (APIs) and hardware increasingly focusing on the programmable aspects of the GPU. The result is a processor with enormous arithmetic capability [a single NVIDIA GeForce 8800 GTX can sustain over 330 giga-floating-point operations per second (Gflops)] and streaming memory bandwidth (80+ GB/s), both substantially greater than a high-end CPU. The GPU has a distinctive architecture centered around a large number of fine-grained parallel processors that we discuss in Section II.

Just as important in the development of the GPU as a general-purpose computing engine has been the advancement of the programming model and programming tools. The challenge to GPU vendors and researchers has been to strike the right balance between low-level access to the hardware to enable performance and high-level programming languages and tools that allow programmer flexibility and productivity, all in the face of rapidly advancing hardware. In Section III, we discuss how general-purpose programs are mapped onto the GPU. We then give a high-level summary of the techniques used in building applications on the GPU (Section V) and the rapidly advancing software environments that enable their development (Section IV).

Until recently, GPU computing could best be described as an academic exercise. Because of the primitive nature of the tools and techniques, the first generation of applications were notable for simply working at all. As the field matured, the techniques became more sophisticated and the comparisons with non-GPU work more rigorous. Our recent survey of the field (completed in November 2006) summarizes this age of GPU computing [1]. We are now entering the third stage of GPU computing: building real applications on which GPUs demonstrate an appreciable advantage.

For instance, as games have become increasingly limited by CPU performance, offloading complex CPU tasks to the GPU yields better overall performance. We summarize one notable GPGPU success in Section VI: "Havok FX" game physics, which runs on NVIDIA and AMD GPUs.

GPUs are also playing an increasing role in scientific computing applications. In Section VII, we detail three applications in computational biophysics: protein folding simulation, scalable molecular dynamics, and calculating electrostatic potential maps. These applications demonstrate the potential of the GPU for delivering real performance gains on computationally complex, large problems.

In Section VIII, we conclude by looking to the future: what features can we expect in future systems, and what are the most important problems that we must address as the field moves forward? One of the most important challenges for GPU computing is to connect with the mainstream fields of processor architecture and programming systems, as well as learn from the parallel computing experts of the past, and we hope that the audience of this paper will find common interests with the experts in our field.

II. GPU ARCHITECTURE

The GPU has always been a processor with ample computational resources. The most important recent trend, however, has been exposing that computation to the programmer. Over the past few years, the GPU has evolved from a fixed-function special-purpose processor into a full-fledged parallel programmable processor with additional fixed-function special-purpose functionality. More than ever, the programmable aspects of the processor have taken center stage.

We begin by chronicling this evolution, starting from the structure of the graphics pipeline and how the GPU has become a general-purpose architecture, then taking a closer look at the architecture of the modern GPU.

A. The Graphics Pipeline

The input to the GPU is a list of geometric primitives, typically triangles, in a 3-D world coordinate system. Through many steps, those primitives are shaded and mapped onto the screen, where they are assembled to create a final picture. It is instructive to first explain the specific steps in the canonical pipeline before showing how the pipeline has become programmable.

Vertex Operations: The input primitives are formed from individual vertices. Each vertex must be transformed into screen space and shaded, typically through computing their interaction with the lights in the scene. Because typical scenes have tens to hundreds of thousands of vertices, and each vertex can be computed independently, this stage is well suited for parallel hardware.

Primitive Assembly: The vertices are assembled into triangles, the fundamental hardware-supported primitive in today's GPUs.

Rasterization: Rasterization is the process of determining which screen-space pixel locations are covered by each triangle. Each triangle generates a primitive called a "fragment" at each screen-space pixel location that it covers. Because many triangles may overlap at any pixel location, each pixel's color value may be computed from several fragments.

Fragment Operations: Using color information from the vertices and possibly fetching additional data from global

memory in the form of textures (images that are mapped onto surfaces), each fragment is shaded to determine its final color. Just as in the vertex stage, each fragment can be computed in parallel. This stage is typically the most computationally demanding stage in the graphics pipeline.

Composition: Fragments are assembled into a final image with one color per pixel, usually by keeping the closest fragment to the camera for each pixel location.

Historically, the operations available at the vertex and fragment stages were configurable but not programmable. For instance, one of the key computations at the vertex stage is computing the color at each vertex as a function of the vertex properties and the lights in the scene. In the fixed-function pipeline, the programmer could control the position and color of the vertex and the lights, but not the lighting model that determined their interaction.

B. Evolution of GPU Architecture

The fixed-function pipeline lacked the generality to efficiently express more complicated shading and lighting operations that are essential for complex effects. The key step was replacing the fixed-function per-vertex and per-fragment operations with user-specified programs run on each vertex and fragment. Over the past six years, these *vertex programs* and *fragment programs* have become increasingly more capable, with larger limits on their size and resource consumption, with more fully featured instruction sets, and with more flexible control-flow operations.

After many years of separate instruction sets for vertex and fragment operations, current GPUs support the unified Shader Model 4.0 on both vertex and fragment shaders [2].

- The hardware must support shader programs of at least 65 k static instructions and unlimited dynamic instructions.
- The instruction set, for the first time, supports both 32-bit integers and 32-bit floating-point numbers.
- The hardware must allow an arbitrary number of both direct and indirect reads from global memory (texture).
- Finally, dynamic flow control in the form of loops and branches must be supported.

As the shader model has evolved and become more powerful, and GPU applications of all types have increased vertex and fragment program complexity, GPU architectures have increasingly focused on the programmable parts of the graphics pipeline. Indeed, while previous generations of GPUs could best be described as additions of programmability to a fixed-function pipeline, today's GPUs are better characterized as a programmable engine surrounded by supporting fixed-function units.

C. Architecture of a Modern GPU

In Section I, we noted that the GPU is built for different application demands than the CPU: large, parallel computation requirements with an emphasis on throughput rather than latency. Consequently, the architecture of the GPU has progressed in a different direction than that of the CPU.

Consider a pipeline of tasks, such as we see in most graphics APIs (and many other applications), that must process a large number of input elements. In such a pipeline, the output of each successive task is fed into the input of the next task. The pipeline exposes the *task parallelism* of the application, as data in multiple pipeline stages can be computed at the same time; within each stage, computing more than one element at the same time is *data parallelism*. To execute such a pipeline, a CPU would take a single element (or group of elements) and process the first stage in the pipeline, then the next stage, and so on. The CPU divides the pipeline in *time*, applying all resources in the processor to each stage in turn.

GPUs have historically taken a different approach. The GPU divides the resources of the processor among the different stages, such that the pipeline is divided in *space*, not time. The part of the processor working on one stage feeds its output directly into a different part that works on the next stage.

This machine organization was highly successful in fixed-function GPUs for two reasons. First, the hardware in any given stage could exploit data parallelism within that stage, processing multiple elements at the same time. Because many task-parallel stages were running at any time, the GPU could meet the large compute needs of the graphics pipeline. Secondly, each stage's hardware could be customized with special-purpose hardware for its given task, allowing substantially greater compute and area efficiency over a general-purpose solution. For instance, the rasterization stage, which computes pixel coverage information for each input triangle, is more efficient when implemented in special-purpose hardware. As programmable stages (such as the vertex and fragment programs) replaced fixed-function stages, the special-purpose fixed-function components were simply replaced by programmable components, but the task-parallel organization did not change.

The result was a lengthy, feed-forward GPU pipeline with many stages, each typically accelerated by special-purpose parallel hardware. In a CPU, any given operation may take on the order of 20 cycles between entering and leaving the CPU pipeline. On a GPU, a graphics operation may take thousands of cycles from start to finish. The latency of any given operation is long. However, the task and data parallelism across and between stages delivers high throughput.

The major disadvantage of the GPU task-parallel pipeline is load balancing. Like any pipeline, the performance of the GPU pipeline is dependent on its slowest stage. If the vertex program is complex and the fragment program is simple, overall throughput is dependent on the performance of the vertex program. In the early days of

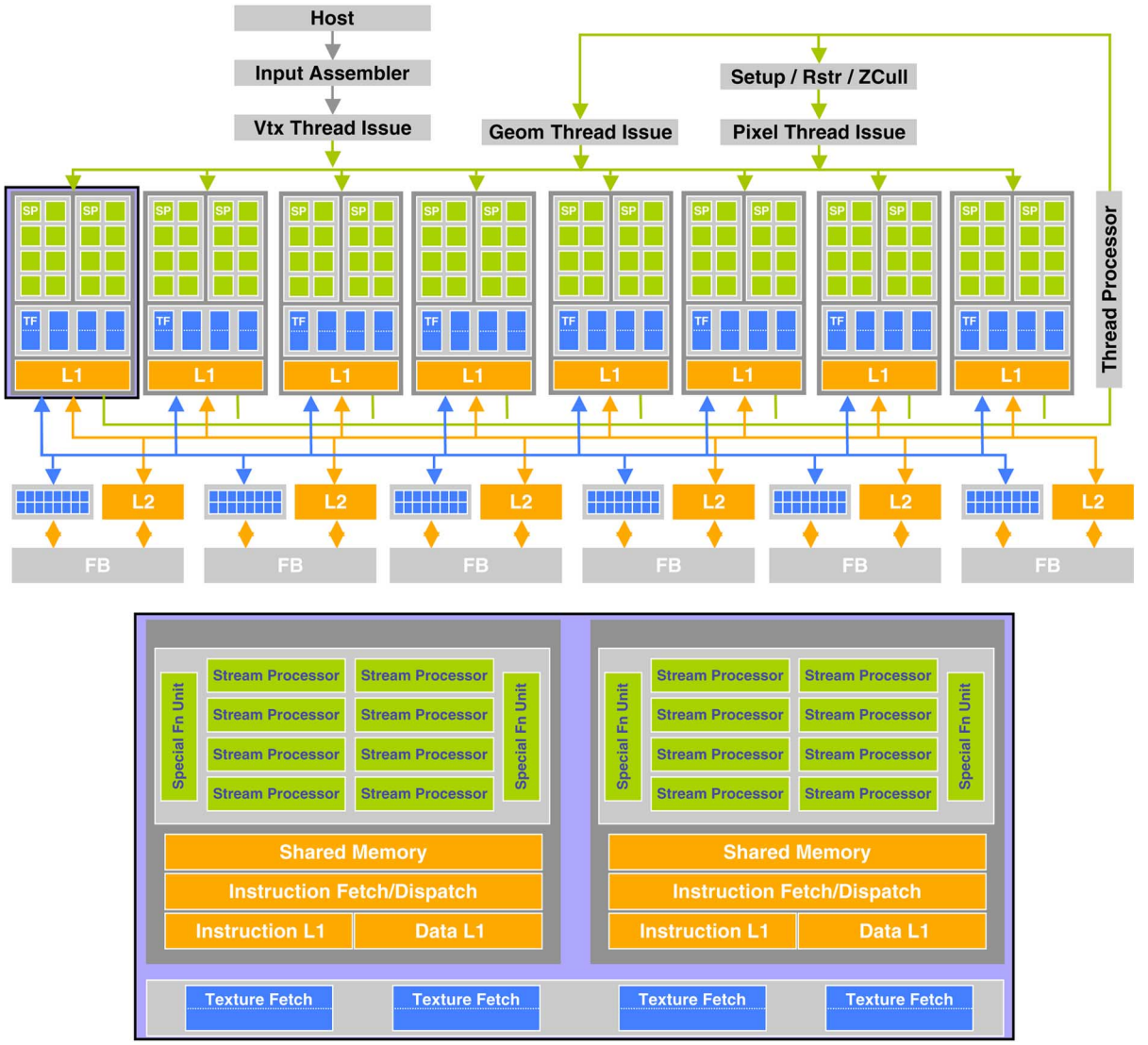

Fig. 1. ***Today, both AMD and NVIDIA build architectures with unified, massively parallel programmable units at their cores. (a) The NVIDIA GeForce 8800 GTX (top) features 16 streaming multiprocessors of 8 thread (stream) processors each. One pair of streaming multiprocessors is shown below; each contains shared instruction and data caches, control logic, a 16 kB shared memory, eight stream processors, and two special function units. (Diagram courtesy of NVIDIA.)***

programmable stages, the instruction set of the vertex and fragment programs were quite different, so these stages were separate. However, as both the vertex and fragment programs became more fully featured, and as the instruction sets converged, GPU architects reconsidered a strict task-parallel pipeline in favor of a *unified shader* architecture, in which all programmable units in the pipeline share a single programmable hardware unit. While much of the pipeline is still task-parallel, the programmable units now divide their time among vertex work, fragment work, and geometry work (with the new DirectX 10 geometry shaders). These units can exploit both task and data parallelism. As the programmable parts of the pipeline are responsible for more and more computation within the graphics pipeline, the architecture of the GPU is migrating from a strict pipelined task-parallel architecture to one that is increasingly built around a single unified data-parallel programmable unit.

AMD introduced the first unified shader architecture for modern GPUs in its Xenos GPU in the XBox 360 (2005). Today, both AMD's and NVIDIA's flagship GPUs feature unified shaders (Fig. 1). The benefit for GPU users is better load-balancing at the cost of more complex hardware. The benefit for GPGPU users is clear: with all the programmable power in a single hardware unit, GPGPU programmers can now target that programmable unit directly, rather than the previous approach of dividing work across multiple hardware units.

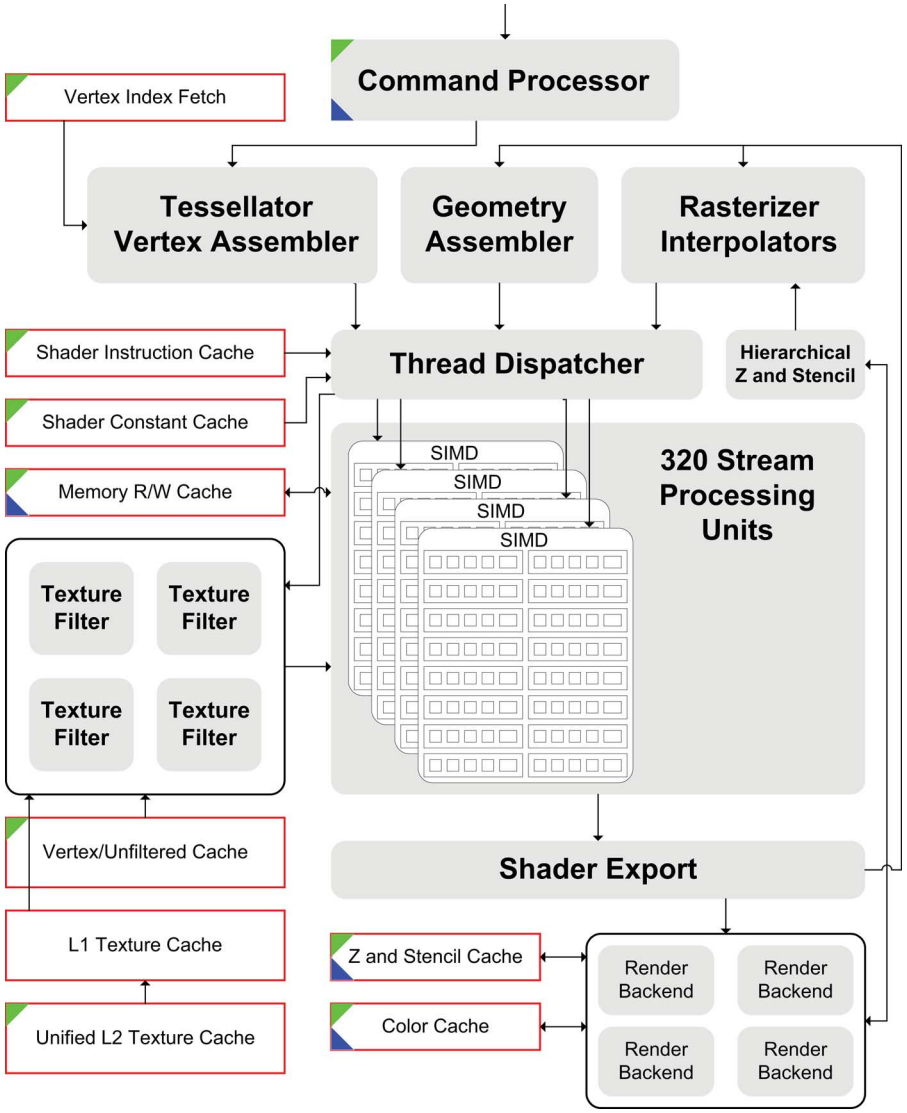

Fig. 1. *(continued) Today, both AMD and NVIDIA build architectures with unified, massively parallel programmable units at their cores. (b) AMD's Radeon HD 2900XT contains 320 stream processing units arranged into four SIMD arrays of 80 units each. These units are arranged into stream processing blocks containing five arithmetic logic units and a branch unit. In the diagram, gray ovals indicate logic units and red-bordered rectangles indicate memory units. Green triangles at the top left of functional units are units that read from memory, and blue triangles at the bottom left write to memory. (Diagram courtesy of M. Doggett, AMD.)*

III. GPU COMPUTING

Now that we have seen the hardware architecture of the GPU, we turn to its programming model.

A. The GPU Programming Model

The programmable units of the GPU follow a single-program multiple-data (SPMD) programming model. For efficiency, the GPU processes many elements (vertices or fragments) in parallel using the same program. Each element is independent from the other elements, and in the base programming model, elements cannot communicate with each other. All GPU programs must be structured in this way: many parallel elements, each processed in parallel by a single program.

Each element can operate on 32-bit integer or floating-point data with a reasonably complete general-purpose instruction set. Elements can read data from a shared global memory (a "gather" operation) and, with the newest GPUs, also write back to arbitrary locations in shared global memory ("scatter").

This programming model is well suited to straight-line programs, as many elements can be processed in lockstep running the exact same code. Code written in this manner is single instruction, multiple data (SIMD). As shader programs have become more complex, programmers

prefer to allow different elements to take different paths through the same program, leading to the more general SPMD model. How is this supported on the GPU?

One of the benefits of the GPU is its large fraction of resources devoted to computation. Allowing a different execution path for each element requires a substantial amount of control hardware. Instead, today's GPUs support arbitrary control flow per thread but impose a penalty for incoherent branching. GPU vendors have largely adopted this approach. Elements are grouped together into blocks, and blocks are processed in parallel. If elements branch in different directions within a block, the hardware computes both sides of the branch for all elements in the block. The size of the block is known as the "branch granularity" and has been decreasing with recent GPU generations—today, it is on the order of 16 elements.

In writing GPU programs, then, branches are permitted but not free. Programmers who structure their code such that blocks have coherent branches will make the best use of the hardware.

B. General-Purpose Computing on the GPU

Mapping general-purpose computation onto the GPU uses the graphics hardware in much the same way as any standard graphics application. Because of this similarity, it is both easier and more difficult to explain the process. On one hand, the actual operations are the same and are easy to follow; on the other hand, the terminology is different between graphics and general-purpose use. Harris provides an excellent description of this mapping process [3]. We begin by describing GPU programming using graphics terminology, then show how the same steps are used in a general-purpose way to author GPGPU applications, and finally use the same steps to show the more simple and direct way that today's GPU computing applications are written.

1) Programming a GPU for Graphics: We begin with the same GPU pipeline that we described in Section II, concentrating on the programmable aspects of this pipeline.

1) The programmer specifies geometry that covers a region on the screen. The rasterizer generates a fragment at each pixel location covered by that geometry.
2) Each fragment is shaded by the fragment program.
3) The fragment program computes the value of the fragment by a combination of math operations and global memory reads from a global "texture" memory.
4) The resulting image can then be used as texture on future passes through the graphics pipeline.

2) Programming a GPU for General-Purpose Programs (Old): Coopting this pipeline to perform general-purpose computation involves the exact same steps but different terminology. A motivating example is a fluid simulation computed over a grid: at each time step, we compute the next state of the fluid for each grid point from the current state at its grid point and at the grid points of its neighbors.

1) The programmer specifies a geometric primitive that covers a computation domain of interest. The rasterizer generates a fragment at each pixel location covered by that geometry. (In our example, our primitive must cover a grid of fragments equal to the domain size of our fluid simulation.)
2) Each fragment is shaded by an SPMD general-purpose fragment program. (Each grid point runs the same program to update the state of its fluid.)
3) The fragment program computes the value of the fragment by a combination of math operations and "gather" accesses from global memory. (Each grid point can access the state of its neighbors from the previous time step in computing its current value.)
4) The resulting buffer in global memory can then be used as an input on future passes. (The current state of the fluid will be used on the next time step.)

3) Programming a GPU for General-Purpose Programs (New): One of the historical difficulties in programming GPGPU applications has been that despite their general-purpose tasks' having nothing to do with graphics, the applications still had to be programmed using graphics APIs. In addition, the program had to be structured in terms of the graphics pipeline, with the programmable units only accessible as an intermediate step in that pipeline, when the programmer would almost certainly prefer to access the programmable units directly.

The programming environments we describe in detail in Section IV are solving this difficulty by providing a more natural, direct, nongraphics interface to the hardware and, specifically, the programmable units. Today, GPU computing applications are structured in the following way.

1) The programmer directly defines the computation domain of interest as a structured grid of threads.
2) An SPMD general-purpose program computes the value of each thread.
3) The value for each thread is computed by a combination of math operations and both "gather" (read) accesses from and "scatter" (write) accesses to global memory. Unlike in the previous two methods, the same buffer can be used for both reading and writing, allowing more flexible algorithms (for example, in-place algorithms that use less memory).
4) The resulting buffer in global memory can then be used as an input in future computation.

This programming model is a powerful one for several reasons. First, it allows the hardware to fully exploit the application's data parallelism by explicitly specifying that parallelism in the program. Next, it strikes a careful balance between generality (a fully programmable routine at each element) and restrictions to ensure good performance

(the SPMD model, the restrictions on branching for efficiency, restrictions on data communication between elements and between kernels/passes, and so on). Finally, its direct access to the programmable units eliminates much of the complexity faced by previous GPGPU programmers in coopting the graphics interface for general-purpose programming. As a result, programs are more often expressed in a familiar programming language (such as NVIDIA's C-like syntax in their CUDA programming environment) and are simpler and easier to build and debug (and are becoming more so as the programming tools mature). The result is a programming model that allows its users to take full advantage of the GPU's powerful hardware but also permits an increasingly high-level programming model that enables productive authoring of complex applications.

IV. SOFTWARE ENVIRONMENTS

In the past, the majority of GPGPU programming was done directly through graphics APIs. Although many researchers were successful in getting applications to work through these graphics APIs, there is a fundamental mismatch between the traditional programming models people were using and the goals of the graphics APIs. Originally, people used fixed function, graphics-specific units (e.g. texture filters, blending, and stencil buffer operations) to perform GPGPU operations. This quickly got better with fully programmable fragment processors which provided pseudo assembly languages, but this was still unapproachable by all but the most ardent researchers. With DirectX 9, higher level shader programming was made possible through the "high-level shading language" (HLSL), presenting a C-like interface for programming shaders. NVIDIA's Cg provided similar capabilities as HLSL but was able to compile to multiple targets and provided the first high-level language for OpenGL. The OpenGL Shading Language (GLSL) is now the standard shading language for OpenGL. However, the main issue with Cg/HLSL/GLSL for GPGPU is that they are inherently shading languages. Computation must still be expressed in graphics terms like vertices, textures, fragments, and blending. So, although you *could* do more general computation with graphics APIs and shading languages, they were still largely unapproachable by the common programmer.

What developers really wanted were higher level languages that were designed explicitly for computation and abstracted all of the graphics-isms of the GPU. BrookGPU [4] and Sh [5] were two early academic research projects with the goal of abstracting the GPU as a streaming processor. The stream programming model structures programs to express parallelism and allows for efficient communication and data transfer, matching the parallel processing resources and memory system available on GPUs. A stream program comprises a set of *streams*, ordered sets of data, and *kernels*, the functions applied to each element in a set of streams producing one or more streams as output.

Brook takes a pure streaming computation abstraction approach representing data as streams and computation as kernels. There is no notion of textures, vertices, fragments, or blending in Brook. Kernels are written in a restricted subset of C, notably the absence of pointers and scatter, and defined the input, output, and gather streams used in a kernel as part of the kernel definition. Brook contains stream access functionality such as repeat and stride, reductions over streams, and the ability to define domains, subsets, of streams to use as input and output. The kernels are run for each element in the domain of output streams. The user's kernels are mapped to fragment shader code and streams to textures. Data upload and download to the GPU is performed via explicit read/write calls translating into texture updates and framebuffer readbacks. Lastly, computation is performed by rendering a quad covering the pixels in the output domain.

Microsoft's Accelerator [6] project has a similar goal as Brook in being very compute-centric, but instead of using offline compilation, Accelerator relies on just-in-time compilation of data-parallel operators to fragment shaders. Unlike Brook and Sh, which are largely extensions to C, Accelerator is an array-based language based on C#, and all computation is done via operations on arrays. Unlike Brook, but similar to Sh, the delayed evaluation model allows for more aggressive online compilation, leading to potentially more specialized and optimized generated code for execution on the GPU.

In the last year, there have been large changes in the ecosystem that allow for much easier development of GPGPU applications as well as providing more robust, commercial quality development systems. RapidMind [7] commercialized Sh and now targets multiple platforms including GPUs, the STI Cell Broadband Engine, and multicore CPUs, and the new system is much more focused on computation as compared to Sh, which included many graphics-centric operations. Similar to Accelerator, RapidMind uses delayed evaluation and online compilation to capture and optimize the user's application code along with operator and type extensions to C++ to provide direct support for arrays. PeakStream [8] is a new system, inspired by Brook, designed around operations on arrays. Similar to RapidMind and Accelerator, PeakStream uses just-in-time compilation but is much more aggressive about vectorizing the user's code to maximize performance on SIMD architectures. PeakStream is also the first platform to provide profiling and debugging support, the latter continuing to be a serious problem in GPGPU development. Both of these efforts represent third-party vendors creating systems with support from the GPU vendors. As a demonstration of the excitement around GPGPU and the success of these approaches to parallel computation, Google purchased PeakStream in 2007.

Both AMD and NVIDIA now also have their own GPGPU programming systems. AMD announced and released their system to researchers in late 2006. CTM, or "close to the metal," provides a low-level hardware abstraction layer (HAL) for the R5XX and R6XX series of ATI GPUs. CTM-HAL provides raw assembly-level access to the fragment engines (stream processors) along with an assembler and command buffers to control execution on the hardware. No graphics-specific features are exported through this interface. Computation is performed by binding memory as inputs and outputs to the stream processors, loading an ELF binary, and defining a domain over the outputs on which to execute the binary. AMD also offers the compute abstraction layer (CAL), which adds higher level constructs, similar to those in the Brook runtime system, and compilation support to GPU ISA for GLSL, HLSL, and pseudoassembly like Pixel Shader 3.0. For higher level programming, AMD supports compilation of Brook programs directly to R6XX hardware, providing a higher level programming abstraction than provided by CAL or HAL.

NVIDIA's CUDA is a higher level interface than AMD's HAL and CAL. Similar to Brook, CUDA provides a C-like syntax for executing on the GPU and compiles offline. However, unlike Brook, which only exposed one dimension of parallelism, data parallelism via streaming, CUDA exposes two levels of parallelism, data parallel and multithreading. CUDA also exposes much more of the hardware resources than Brook, exposing multiple levels of memory hierarchy: per-thread registers, fast shared memory between threads in a block, board memory, and host memory. Kernels in CUDA are also more flexible that those in Brook by allowing the use of pointers (although data must be on board), general load/store to memory allowing the user to scatter data from within a kernel, and synchronization between threads in a thread block. However, all of this flexibility and potential performance gain comes with the cost of requiring the user to understand more of the low-level details of the hardware, notably register usage, thread and thread block scheduling, and behavior of access patterns through memory.

All of these systems allow developers to more easily build large applications. For example, the Folding@Home GPU client and large fluid simulation application are written in BrookGPU, NAMD and VMD support GPU execution through CUDA, RapidMind has demonstrated ray-tracing and crowd simulation, and PeakStream has shown oil and gas as well as computational finance applications. CUDA provides tuned and optimized basic linear algebra subprograms (BLAS) and fast Fourier transform (FFT) libraries to use as building blocks for large applications. Low-level access to hardware, such as that provided by CTM, or GPGPU-specific systems like CUDA, allow developers to effectively bypass the graphics drivers and maintain stable performance and correctness. The vendors' driver development and optimizations for graphics APIs tend to test only the latest released or most popular games. Optimizations performed to optimize for game performance can affect GPGPU application stability and performance.

V. TECHNIQUES AND APPLICATIONS

We now survey some important computational primitives, algorithms, and applications for GPU computing. We first highlight four data-parallel operations central to GPU computing: performing scatter/gather memory operations, mapping a function onto many elements in parallel, reducing a collection of elements to a single element or value, and computing prefix reductions of an array in parallel. We delve into these core computational primitives in some detail before turning to a higher level overview of algorithmic problems that researchers have studied on GPUs: scan, sort, search, data queries, differential equations, and linear algebra. These algorithms enable a wide range of applications ranging from databases and data mining to scientific simulations such as fluid dynamics and heat transfer to—as we shall see in Sections VI and VII—rigid-body physics for games and molecular dynamics.

A. Computational Primitives

The data-parallel architecture of GPUs requires programming idioms long familiar to parallel supercomputer users but often new to today's programmers reared on sequential machines or loosely coupled clusters. We briefly discuss four important idioms: scatter/gather, map, reduce, and scan. We describe these computational primitives in the context of both "old" (i.e., graphics-based) and "new" (direct-compute) GPU computing to emphasize the simplicity and flexibility of the direct-compute approach.

Scatter/gather: write to or read from a computed location in memory. Graphics-based GPU computing allows efficient gather using the texture subsystem, storing data as images (textures) and addressing data by computing corresponding image coordinates and performing a texture fetch. However, texture limitations make this unwieldy: texture size restrictions require wrapping arrays containing more than 4096 elements into multiple rows of a two-dimensional (2-D) texture, adding extra addressing math, and a single texture fetch can only retrieve four 32-bit floating point values, limiting per-element storage. Scatter in graphics-based GPU computing is difficult and requires rebinding data for processing as vertices, either using vertex texture fetch or render-to-vertex-buffer. By contrast, direct-compute layers allow unlimited reads and writes to arbitrary locations in memory. NVIDIA's CUDA allows the user to access memory using standard C constructs (arrays, pointers, variables). AMD's CTM is nearly as flexible but uses 2-D addressing.

Map: apply an operation to every element in a collection. Typically expressed as a *for* loop in a sequential program (e.g., a thread on a single CPU core), a parallel implementation can reduce the time required by applying the operation to many elements in parallel. Graphics-based

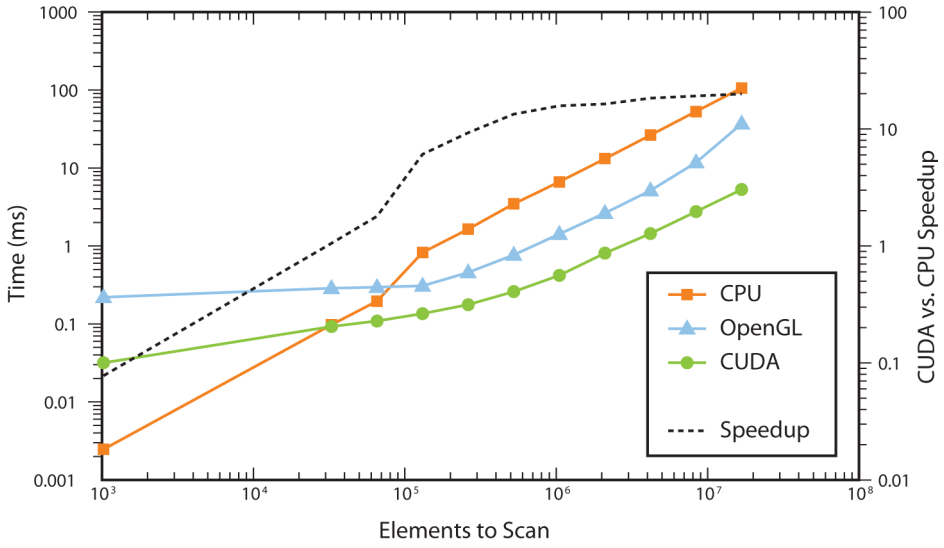

Fig. 2. ***Scan performance on CPU, graphics-based GPU (using OpenGL), and direct-compute GPU (using CUDA). Results obtained on a GeForce 8800 GTX GPU and Intel Core2-Duo Extreme 2.93 GHz CPU. (Figure adapted from Harris et al.*** **[10].)**

GPU computing performs *map* as a fragment program to be invoked on a collection of pixels (one pixel for each element). Each pixel's fragment program fetches the element data from a texture at a location corresponding to the pixel's location in the rendered image, performs the operation, then stores the result in the output pixel. Similarly, CTM and CUDA would typically launch a thread program to perform the operation in many threads, with each thread loading an element, performing the computation, and storing the result. Note that since loops are supported, each thread may also loop over several elements.

Reduce: repeatedly apply a binary associative operation to reducing a collection of elements to a single element or value. Examples include finding the sum (average, minimum, maximum, variance, etc.) of a collection of values. A sequential implementation on a traditional CPU would loop over an array, successively summing (for example) each element with a running sum of elements seen so far. By contrast, a parallel reduce-sum implementation would repeatedly perform sums in parallel on an ever-shrinking set of elements.[1] Graphics-based GPU computing implements reduce by rendering progressively smaller sets of pixels. In each rendering pass, a fragment program reads multiple values from a texture (performing perhaps four or eight texture reads), computes their sum, and writes that value to the output pixel in another texture (four or eight times smaller), which is then bound as input to the same fragment shader and the process repeated until the output consists of a single pixel that contains the result of the final reduction. CTM and CUDA express this same process more directly, for example, by launching a set of threads each of which reads two elements and writes their sum to a single element. Half the threads then repeat this process, then half of the remaining threads, and so on until a single surviving thread writes the final result to memory.

Scan: Sometimes known as parallel-prefix-sum, *scan* takes an array A of elements and returns an array B of the same length in which each element $B[i]$ represents a reduction of the subarray $A[1 \ldots i]$. Scan is an extremely useful building block for data-parallel algorithms; Blelloch describes a wide variety of potential applications of scan ranging from quicksort to sparse matrix operations [9]. Harris *et al.* [10] demonstrate an efficient scan implementation using CUDA (Fig. 2); their results illustrate the advantages of a direct-compute over graphics-based GPU computing. Their CUDA implementation outperforms the CPU by a factor of up to 20 and OpenGL by a factor of up to seven.

B. Algorithms and Applications

Building largely on the above primitives, researchers have demonstrated many higher level algorithms and applications that exploit the computational strengths of the GPU. We give only a brief survey of GPU computing algorithms and their application domains here; for a detailed overview, please see our recent survey [1].

Sort: GPUs have come to excel at sorting as the GPU computing community has rediscovered, adapted, and improved seminal sorting algorithms, notably *bitonic merge sort* [11]. This "sorting network" algorithm is intrinsically parallel and *oblivious*, meaning the same steps are executed

[1]Note that floating-point arithmetic is only pseudoassociative, so parallel and sequential reduce may produce different final values due to roundoff, etc.

regardless of input. Govindaraju *et al.* won the price-performance "PennySort" category of the 2005 "TeraSort" competition [12] using careful system design and a combination of many algorithmic improvements.

Search and database queries: Researchers have also implemented several forms of search on the GPU, such as binary search [13] and nearest neighbor search [14], as well as high-performance database operations that build on special-purpose graphics hardware (called the *depth* and *stencil buffers*) and the fast sorting algorithms mentioned above [15], [16].

Differential equations: The earliest attempts to use GPUs for nongraphics computation focused on solving large sets of differential equations. Particle tracing is a common GPU application for ordinary differential equations, used heavily in scientific visualization (e.g., the scientific flow exploration system by Krüger *et al.* [17]) and in visual effects for computer games. GPUs have been heavily used to solve problems in partial differential equations (PDEs) such as the Navier–Stokes equations for incompressible fluid flow. Particularly successful applications of GPU PDE solvers include fluid dynamics (e.g., Bolz *et al.* [18]) and level set equations for volume segmentation [19].

Linear algebra: Sparse and dense linear algebra routines are the core building blocks for a huge class of numeric algorithms, including many PDE solvers mentioned above. Applications include simulation of physical effects such as fluids, heat, and radiation, optical effects such as depth of field [20], and so on. Accordingly, the topic of linear algebra on GPUs has received a great deal of attention. One representative example is the work of Krüger and Westermann [21], which addressed a broad class of linear algebraic problems by focusing on the representation of matrices and vectors in graphics-based GPU computing (e.g., packing dense and sparse vectors into textures, vertex buffers, etc.). Other notable work includes an analysis of dense matrix–matrix multiplication by Fatahalian *et al.* [22] and a solver for dense linear systems by Gallapo *et al.* [23] that the authors show is able to outperform even highly optimized ATLAS implementations.

The use of direct-compute layers such as CUDA and CTM both simplifies and improves the performance of linear algebra on the GPU. For example, NVIDIA provides CuBLAS, a dense linear algebra package implemented in CUDA and following the popular BLAS conventions. Sparse linear algebraic algorithms, which are more varied and complicated than dense codes, are an open and active area of research; researchers expect sparse codes to realize benefits similar to or greater than those of the new GPU computing layers.

C. Recurring Themes

Several recurring themes emerge throughout the algorithms and applications explored in GPU computing to date. Examining these themes allows us to characterize what GPUs do well. Successful GPU computing applications do the following.

Emphasize parallelism: GPUs are fundamentally parallel machines, and their efficient utilization depends on a high degree of parallelism in the workload. For example, NVIDIA's CUDA prefers to run *thousands* of threads at one time to maximize opportunities to mask memory latency using multithreading. Emphasizing parallelism requires choosing algorithms that divide the computational domain into as many independent pieces as possible. To maximize the number of simultaneous running threads, GPU programmers should also seek to minimize thread usage of shared resources (such as local registers and CUDA shared memory) and should use synchronization between threads sparingly.

Minimize SIMD divergence: As Section III discusses, GPUs provide an SPMD programming model: multiple threads run the same program but access different data and thus may diverge in their execution. At some granularity, however, GPUs perform SIMD execution on batches of threads (such as CUDA "warps"). If threads within a batch diverge, the entire batch will execute both code paths until the threads reconverge. High-performance GPU computing thus requires structuring code to minimize divergence within batches.

Maximize arithmetic intensity: In today's computing landscape, actual computation is relatively cheap but bandwidth is precious. This is dramatically true for GPUs with their abundant floating-point horsepower. To obtain maximum utilization of that power requires structuring the algorithm to maximize the *arithmetic intensity* or number of numeric computations performed per memory transaction. Coherent data accesses by individual threads help, since these can be coalesced into fewer total memory transactions. Use of CUDA shared memory on NVIDIA GPUs also helps, reducing overfetch (since threads can communicate) and enabling strategies for "blocking" the computation in this fast on-chip memory.

Exploit streaming bandwidth: Despite the importance of arithmetic intensity, it is worth noting that GPUs do have very high peak bandwidth to their onboard memory, on the order of 10× the CPU-memory bandwidths on typical PC platforms. This is why GPUs can outperform CPUs at tasks such as sort, which have a low computation/bandwidth ratio. To achieve high performance on such applications requires *streaming* memory access patterns in which threads read from and write to large coherent blocks (maximizing bandwidth per transaction) located in separate regions of memory (avoiding data hazards).

Experience has shown that when algorithms and applications can follow these design principles for GPU computing—such as the PDE solvers, linear algebra packages, and database systems referenced above, and the game physics and molecular dynamics applications examined in detail next—they can achieve 10–100× speedups over even mature, optimized CPU codes.

VI. CASE STUDY: GAME PHYSICS

Physics simulation occupies an increasingly important role in modern video games. Game players and developers seek environments that move and react in a physically plausible fashion, requiring immense computational resources. Our first case study focuses on Havok FX (Fig. 3), a GPU-accelerated game physics package and one of the first successful consumer applications of GPU computing.

Game physics takes many forms and increasingly includes articulated characters ("rag doll physics"), vehicle simulation, cloth, deformable bodies, and fluid simulation. We concentrate here on rigid body dynamics, which simulate solid objects moving under gravity and obeying Newton's laws of motion and are probably the most important form of game physics today. Rigid body simulation typically incorporates three steps: integration, collision detection, and collision resolution.

Integration: The integration step updates the objects' velocities based on the applied forces (e.g., gravity, wind, player interactions) and updates the objects' position based on the velocities.

Collision detection: This step determines which objects are colliding after integration and their contact points. Collision detection must in principle compare each object with every other object—a very expensive ($O(n^2)$) proposition. In practice, most systems mitigate this cost by splitting collision detection into a broad phase and a narrow phase [24]. The broad phase compares a simplified representation of the objects (typically their bounding boxes) to quickly determine potentially colliding pairs of objects. The narrow phase then accurately determines the pairs of objects that are actually colliding, resulting in the contact points, contact normals, and penetration depths.

Collision resolution: Once collisions are detected, collision resolution applies impulses (instant transitory forces) to the colliding objects so that they move apart.

Due to the hard real-time constraints of game play, game physics systems usually employ iterative solvers rather than the matrix-based solvers more commonly described in the literature. This allows them to trade off accuracy for performance by varying the number of iterations.

In 2005, Havok, the leading game physics middleware supplier, began researching new algorithms targeted at simulating tens of thousands of rigid bodies on parallel

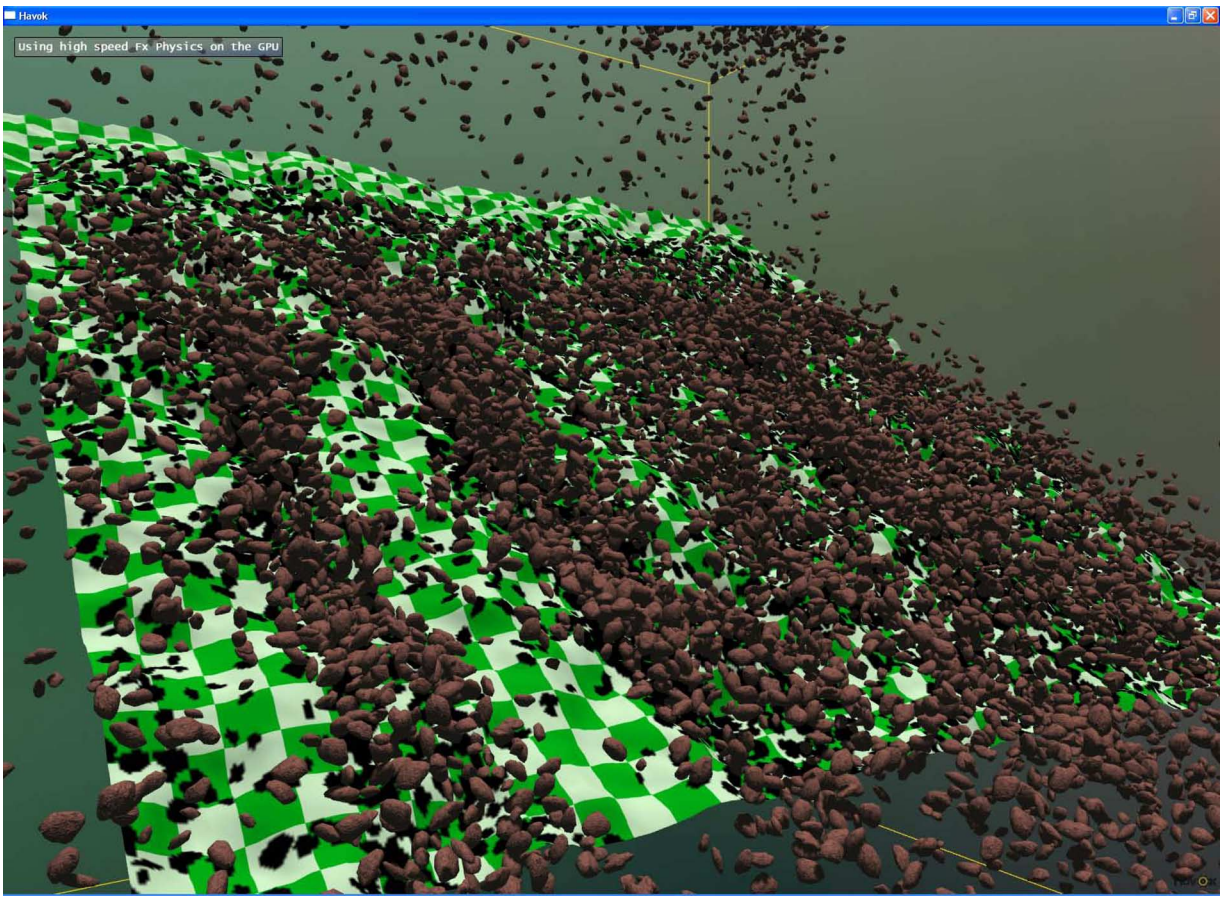

Fig. 3. *Havok FX can simulate 15 000 colliding boulders at more than 60 frames per second.*

processors. Havok FX was the result, and both NVIDIA and ATI have worked with Havok to implement and optimize the system on the GPU. Several reasons argue for moving some physics simulation to the GPU. For instance, many games today are CPU-limited, and physics can easily consume 10% or more of CPU time. Performing physics on the GPU also enables direct rendering of simulation results from GPU memory, avoiding the need to transfer the positions and orientations of thousands or millions of objects from CPU to GPU each frame.

Havok FX is a hybrid system, leveraging the strengths of the CPU and GPU. It stores the complete object state (position, orientation, linear and angular velocities) on the GPU, as well as a proprietary texture-based representation for the shapes of the objects. This representation is designed to handle convex–convex collisions very quickly, though potentially less accurately than a (much more expensive) full polyhedral intersection algorithm.

The CPU performs broad phase collision detection using a highly optimized sort and sweep algorithm after reading axis-aligned bounding boxes of each object back from the GPU each frame in a compressed format. The list of potential colliding pairs is then downloaded back to the GPU for the narrow phase. Both transfers consist of a relatively small amount of data (on the order of a few hundred kilobytes), which transfer quickly over the PCIe bus.

To improve simulation stability, the CPU splits colliding pairs into independent batches in which each object is involved in at most one collision, and each batch is processed in a separate pass. A major challenge facing the designers of Havok FX was to minimize the number of passes, increasing the amount of parallel work to do in each pass and thus reducing the total overhead of initiating computation on the GPU.

The GPU performs all narrow phase collision detection and integration. Havok FX uses a simple Euler integrator with a fixed time step. The quality of the collision solver is generally more important than the integrator for the stability of the simulation.

The system includes a simple friction model stable enough to handle basic stacking of objects, for example, to simulate brick walls composed of individually simulated bricks. Rendering takes place directly from data on the GPU, typically using the instancing feature of the graphics API for maximum performance.

The shader programs used in Havok FX were among the most complex ever developed at the time, comprising thousands of instructions and stretching the limits of available resources. Havok FX also allows user-defined shaders to be written for custom effects such as boundary conditions, vortices, attractors, special-case collision objects, etc.

The end result is an order of magnitude performance boost over Havok's reference single-core CPU implementation. Simulating a scene of 15 000 boulders rolling down a terrain, the CPU implementation (on a single core of an Intel 2.9 GHz Core 2 Duo) achieved 6.2 frames per second, whereas the initial GPU implementation on an NVIDIA GeForce 8800 GTX reached 64.5 frames per second. Performance continues to scale as new generations of GPUs are released. The system also supports multiple GPUs—one GPU can be used exclusively to perform the physics computations, while another is dedicated to rendering. The rendering data can either be transferred via the host CPU or transmitted directly using a peer-to-peer PCIe transfer.

Havok FX demonstrates the feasibility of building a hybrid system in which the CPU executes serial portions of the algorithm and the GPU executes data parallel portions. The overall performance of this hybrid system far exceeds a CPU-only system despite the frequent transfers between CPU and GPU, which are often seen as an obstacle to such hybrid system. Soon the increasing flexibility of GPUs should allow executing the complete pipeline, including broad phase collision, on the GPU for even greater performance.

VII. CASE STUDIES: COMPUTATIONAL BIOPHYSICS

Commercial GPUs deliver a high compute capacity at low cost, making them attractive for use in scientific computing. In this section, we focus on the application of GPU computing to three computationally demanding applications in the field of computational biophysics: Folding@Home, a protein-folding simulation project at Stanford University; NAMD, a scalable molecular dynamics package developed at the University of Illinois; and VMD, a molecular visualization and analysis tool also developed at the University of Illinois.

The classical N-body problem consists of obtaining the time evolution of a system of N mass particles interacting according to a given force law. The problem arises in several contexts, ranging from molecular-scale calculations in structural biology to stellar-scale research in astrophysics. Molecular dynamics (MD) has been successfully used to understand how certain proteins fold and function, which have been outstanding questions in biology for over three decades. The calculations required for MD are extremely compute intensive, and research is limited by the computational resources available. Even the calculations required for simulation preparation and analysis have become onerous as MD simulation protocols have become more sophisticated and the size of simulated structures has increased.

Every additional factor of ten in total simulation performance that is made available to the biomedical research community opens new doors in either simulation size or time scale. Unfortunately, increases in the performance of individual processor cores have recently stalled as faster clock speeds result in unacceptable heat

and power consumption. Thus, the developers of software such as NAMD, Folding@Home, and VMD are required to extend the limits of parallel computing and seek out new paths to increased performance on architectures such as GPUs. Compared against highly tuned CPU versions of these applications, the GPU implementations provide more than an order of magnitude gain in performance.

A. Folding@Home: Massively Distributed Simulation

Folding@Home is a distributed molecular dynamics application for studying the folding behavior of proteins and is designed to run on home and office computers donated by individuals and organizations around the world. The GPU implementation of Folding@Home, the first widely deployed scientific GPU computing application, delivers large improvements in performance over current-generation CPUs [25]. The Folding@Home project has achieved a massive increase in computing power from using these methods on GPUs, as well as on the Cell processor in the Sony Playstation 3.[2]

The simplest force models are pairwise additive: the force of interaction between two particles is independent of all the other particles, and the individual forces on a particle add linearly. The force calculation for such models is of complexity $O(N^2)$. Since typical studies involve a large number of particles (10^3 to 10^6) and the desired number of integration steps is usually very large (10^6 to 10^{15}), computational requirements often limit both problem size and simulation time, and consequently limit the useful information that may be obtained from such simulations. Numerous methods have been developed to deal with these issues. For molecular simulations, it is common to reduce the number of particles by treating the solvent molecules as a continuum. Folding@Home uses this technique, performing the $O(N^2)$ force calculations that constitute the major part of N-body protein-folding simulations with implicit water. In stellar simulations, one uses individual time stepping or tree algorithms to minimize the number of force calculations. Despite such algorithmic approximations and optimizations, the computational capabilities of traditional hardware remain the limiting factor.

Typically, N-body simulations utilize neighbor-lists, tree methods, or other algorithms to reduce the quadratic complexity of the force calculations. Building the neighbor-list is difficult without a scatter operation to memory along with synchronization primitives, and research on computing the neighbor-list on the GPU is still in progress. Interacting with the CPU for these updates is also impractical. However, we find we can do an $O(N^2)$ calculation significantly faster on the GPU than an $O(N)$ method using the CPU (or even with a combination of the GPU and CPU) for the protein sizes used in Folding@Home. This has direct applicability to biological simulations that use continuum solvent models. We note also that in many of the reduced order methods such as tree-based schemes, at some stage an $O(N^2)$ calculation is performed on a subsystem of the particles, so our method can be used to improve the performance of such methods as well.

1) GPU Algorithm: In its simplest form, the N-body force calculation can be described by the following pseudocode:

```
for i = 1 to N
    force[i] = 0
    ri = coordinates[i]
    for j = 1 to N
        rj = coordinates[j]
        force[i] = force[i] + force_function(ri, rj)
    end
end
```

Since all coordinates are fixed during the force calculation, the force computation can be parallelized for the different values of i. In terms of streams and kernels, this can be expressed as follows:

```
stream coordinates;
stream forces;
kernel kforce(ri)
    force = 0
    for j = 1 to N
        rj = coordinates[j]
        force = force + force_function(ri, rj)
    end
    return force
end kernel
forces = kforce(coordinates)
```

The kernel kforce is applied to each element of the stream coordinates to produce an element of the forces stream. Note that the kernel can perform an indexed fetch from the coordinates stream inside the j-loop. An out-of-order indexed fetch can be slow, since in general, there is no way to prefetch the data. However, in this case, the indexed accesses are sequential. Moreover, the j-loop is executed simultaneously for many i-elements; even with minimal caching, rj can be reused for many N i-elements without fetching from memory. Thus the performance of this algorithm would be expected to be high. The implementation of this algorithm on GPUs and GPU-specific performance optimizations are described in the following section.

There is, however, one caveat in using a streaming model. Newton's Third Law states that the force on particle i due to particle j is the negative of the force on particle j due to particle i. CPU implementations use this fact to halve the number of force calculations. However, the combination of GPU and programming system used in this implementation

[2]http://www.folding.stanford.edu/.

does not allow kernels to write an out-of-sequence element (scatter), so forces[j] cannot be updated while summing over the j-loop to calculate forces[i]. This effectively doubles the number of computations that must be done on the GPU compared to a CPU.

2) Kernel Optimization: The algorithm outlined in Section VII-A1 was implemented in BrookGPU and targeted for the ATI X1900XTX. Even this naive implementation performs very well, achieving over 40 GFLOPS, but its performance can be improved by carefully adjusting the implementation for better arithmetic intensity. We take a closer look at one particular kernel, the GA kernel, which corresponds to the gravitational attraction between two mass particles. GA is one of five force-calculation kernels, which input between 64 to 128 bytes per interaction, deliver 19–43 FLOPs on each interaction, and have inner loops from 104 to 138 instructions in length.

A naïve implementation of GA executes 48 G-instructions/s and has a memory bandwidth of 33 GB/s. Using information from GPUBench [26], we expect the X1900XTX to be able to execute approximately 30–50 G-instruction/s (it depends heavily on the pipelining of commands) and have a cache memory bandwidth of 41 GB/s. The nature of the algorithm is such that almost all the memory reads will be from the cache since all the pixels being rendered at a given time will be accessing the same j-particle. Thus this kernel is limited by the rate at which the GPU can issue instructions (compute bound).

To improve this kernel's performance, we utilize several techniques. We begin with loop unrolling, which achieves a modest speedup compared to our first implementation and results in a switch from compute-bound to bandwidth-bound (35 G-instructions/s and ~40 GB/s).

Further reducing bandwidth usage is somewhat more difficult. It involves using the multiple render targets capability of recent GPUs, which is abstracted as multiple output streams by BrookGPU. This reduces by a factor of four the bandwidth required by both input and output streams. This results in a kernel that is once more instruction-rate limited. Its bandwidth is half that of the maximum bandwidth available on the ATI X1900XT, but the overall performance has increased significantly, around a factor of two.

In all cases, performance is severely limited when the number of particles is less than about 4000. This is due to a combination of fixed overhead in executing kernels and the lack of sufficiently many parallel threads of execution to hide latency. In molecular dynamics, where forces tend to be short-range in nature, it is more common to use $O(N)$ methods by neglecting or approximating the interactions beyond a certain cutoff distance. However, when using continuum solvent models, the number of particles is small enough ($N \approx 1000$) that the $O(N^2)$ method is comparable in complexity while giving greater accuracy than $O(N)$ methods. To take maximal advantage of GPUs, it is therefore important to get good performance for small output stream sizes, and we do so by increasing the number of parallel threads by replicating the input stream and performing a reduction to get the final forces.

3) Performance: The GROMACS [27] molecular dynamics software is highly tuned and uses handwritten SSE assembly loops. As mentioned in Section VII-A1, the CPU can do out-of-order writes without a significant penalty. GROMACS uses this fact to halve the number of calculations needed in each force calculation step. In the comparison against the GPU in Table 1, this has been accounted for in the performance numbers. In MD it is common to use neighbor lists to reduce the order of the force computation to $O(N)$. The performance of GROMACS doing an $O(N^2)$ calculation as well as an $O(N)$ calculation for a 80-residue protein (lambda repressor, 1280 atoms) is shown in Table 1. Despite using a fairly modest cutoff length of 1.2 nm for the $O(N)$ calculation, the $O(N^2)$ GPU calculation represents an order-of-magnitude performance improvement over existing methods on CPUs.

4) Application to Folding@Home: Most biological phenomena of interest occur on time scales currently beyond the reach of MD simulations. For example, the simplest proteins fold on a time scale of 5 to 20 μs, while more complex proteins may take milliseconds to seconds. MD simulations on current generation CPUs are usually limited to simulating about 10 ns per day—it would take several years to obtain a 10 μs simulation. However, with the speed increases afforded by the algorithms and hardware discussed here, we are now be able to simulate protein dynamics with individual trajectories on the 10 μs time scale in under three months. This allows the direct simulation of the folding of fast-folding proteins. Moreover, by incorporating this methodology into a distributed computing framework, we are now situated to build Markovian state models to simulate even longer time scales, likely approaching seconds [28]. Thus with the

Table 1 Comparison of GROMACS (GMX) Running on a 3.2 GHz Pentium 4 Versus the GPU Showing the Simulation Time per Day for an 80-Residue Protein (Lambda Repressor)

Kernel	GMX $O(N^2)$ ns/day	GMX $O(N)$ ns/day	GPU ns/day
LJC (constant)	5.6	18.2	140
LJC (linear)*	2.06	9.08	140
LJC (sigmoidal)	2.5	11	127
	GMX Million Interactions/sec	GPU Million Interactions/sec	
LJC (constant)	66	1327	
LJC (linear)*	33	1327	
LJC (sigmoidal)	40	1203	

*Note GROMACS does not have an SSE inner loop for LJC (linear).

combined effort of GPUs and distributed computing, one would be able to reach time scales for folding of essentially all single-domain two-state folding proteins. Compared to the donations of CPUs from over 150 000 Windows computers currently producing 145 Tflops, we have 550 GPUs donated to the project producing over 34 Tflops. Thus each GPU is providing roughly 60 times the performance of the average donated x86 CPU.

The streaming model we have designed for the GPU is also the basis for the code executing on the Sony Playstation 3, showing the portability of the streaming model to other parallel processors. With the combined computing power of GPUs and the Playstation 3, we have more than tripled the computing power available to Folding@Home, approaching 1 Pflops. Although we are currently only running implicit solvent models, which represent a subset of the simulations performed by Folding@Home, work continues in efficient implementations of the data structures required to support $O(N)$ and $O(N \log N)$ models for explicit solvent routines. The main issue is not the efficient use of the data structures but the building and updating of these data structures as the simulation progresses. The limited synchronization and communication capabilities of current GPUs make this difficult, although there is promising research into this area.

B. NAMD: Large-Scale Molecular Dynamics

NAMD (nanoscale molecular dynamics)[3] is an award-winning package for the classical molecular dynamics simulation of biomolecular systems on large parallel computers [29]. In this section, we first describe NAMD usage and a bit of the underlying science, emphasizing the core NAMD algorithms including some (highly tuned) CPU implementation details. We then describe a GPU implementation that exploits new capabilities exposed by CUDA to achieve a 20× speedup [30].

NAMD models full atomic coordinates of proteins, nucleic acids, and/or lipids solvated in explicit water and ions based on known crystallographic or other structures An empirical energy function, which consists of approximations of covalent interactions (bonded terms) in addition to Lennard–Jones and electrostatic (nonbonded) terms, is applied to the system. The resulting Newtonian equations of motion are typically integrated by symplectic and reversible methods using femtosecond timesteps. Modifications are made to the equations of motion to control temperature and pressure during the simulation.

Continuing increases in high performance computing technology have rapidly expanded the domain of biomolecular simulation from isolated proteins in solvent to complex aggregates, often in a lipid environment. Such systems can easily comprise 100 000 atoms, and several published NAMD simulations have exceeded 1 000 000 atoms. At the same time, studying the function of even the simplest of biomolecular machines requires simulations of 100 ns or longer, even when employing techniques for accelerating processes of interest.

1) Optimization Strategies: The complexity of long-range electrostatic force evaluation is reduced from $O(N^2)$ to $O(N \log N)$ via the particle mesh Ewald (PME) algorithm, which combines an 8–12 Å cutoff direct calculation with an FFT-based mesh calculation. The short-range correction required for PME involves the expensive erfc() function, which is more efficiently evaluated through an interpolation table. NAMD on a CPU also uses interpolation tables for the r^{-12} and r^{-6} terms of the Lennard–Jones potential. For each pair of atoms, the distance-dependent interpolants are multiplied by parameters that depend on the types of the atoms and are combined for each pair of atoms through simple algebraic forms.

Bonded forces reflect the effect of covalent chemical bonds and involve only two to four nearby atoms in the molecular chain. Since bonded pairs of atoms share electrons, the normal nonbonded potential does not apply to them and we must avoid calculating nonbonded forces between such excluded pairs of atoms. The number of excluded pairs grows with the size of the simulation, but the repetitive nature of biomolecules allows all of the exclusion data to be compressed into a few hundred atom *signatures*.

NAMD uses a spatial decomposition strategy for parallelization in which atoms are assigned to boxes slightly larger than the cutoff distance in each dimension. These boxes are then distributed to processors. The computation of forces between atoms in neighboring boxes is then independently assigned to processors by a measurement-based load balancer. Even in serial runs, this box-based decomposition functions to determine atoms within the cutoff distance in $O(N)$ time. For greater performance on the CPU, a list of atom pairs is created periodically for every pair of neighboring boxes.

2) GPU Strategies: The GPU implementation of nonbonded forces for NAMD takes advantage of all of the G80 resources made available under the CUDA programming model. Forces between all pairs of neighboring boxes are evaluated in a single grid, with each thread block responsible for a pair of boxes. Thirty-two bytes of data are required per atom, allowing the 16 KB shared memory to store roughly 500 atoms of one box. A second box can be stored in the 32 KB of registers associated with individual threads, with one or two atoms per thread. Both sets of atoms can be efficiently loaded from device memory through coalesced reads. For each atom, the least and greatest excluded atom index as well as an index into the compressed exclusion table is loaded along with the coordinates, charge, and Lennard–Jones

[3]http://www.ks.uiuc.edu/Research/namd/.

parameters. The entire exclusion table, stored as individual bits, can then be stored in the 8 KB constant cache. The force interpolation table is implemented through the texture unit, which provides special hardware for linear interpolation. A total of 512 sample points for four different functions fit in the 8 KB texture cache.

All threads in a block simultaneously iterate through the atoms in shared memory, accumulating the forces on the atoms in their individual registers. Since all threads access the same shared memory location, data are broadcast to all threads by the hardware, and no bank conflict penalty is incurred. The force between any pair of atoms is only calculated if they are within the cutoff distance, which produces a small performance gain despite the added branch because, in a significant fraction of cases, all of the threads within a warp will have atoms outside of the cutoff distance. It is possible to use shared memory to accumulate reciprocal forces, reducing the number of force calculations by half, but the overhead of synchronizing threads between individual force calculations exceeds the cost of performing the calculations themselves. When the forces on the atoms in registers have been accumulated, the forces are written to a buffer in device memory, the atoms in registers and shared memory are swapped, and the force calculation is repeated for the new atoms. Once the force-calculation grid completes, a second grid with one block per box accumulates forces for each atom from all of its output buffers and writes the net force to a contiguous array that can be efficiently copied to the CPU. All force reads and writes are coalesced by the hardware.

When implemented in a test harness with box dimensions exactly equal to the cutoff distance, the GPU provided a factor of 20 performance increase over a single CPU core. In order to reduce the frequency of assigning atoms to boxes and to keep hydrogen atoms in the same processor as the heavy atoms to which they are bonded, NAMD uses larger boxes, 16 Å for a 12 Å cutoff. As a result, more than 93% of pairs of atoms in neighboring boxes are beyond the cutoff distance. On the CPU, this is dealt with through a pair list, updated every ten steps, that stores all atoms that are or may move within the cutoff distance of another atom before the next update. Experiments with using a pair list on the GPU, loading atoms randomly through the texture unit, showed double the speed of the box method above, but building the pairlist on the CPU and copying it to the GPU more than negated this advantage. Generating the pairlist efficiently on the GPU may require atomic memory operations not available in CUDA. The current GPU implementation has reduced the time for nonbonded force calculation in NAMD to the level that it can be overlapped with bonded forces and the PME long-range force calculation on the CPU. These other calculations must be ported to the GPU before further optimization of nonbonded forces will be useful.

C. VMD: Electrostatics of Biomolecules

VMD (visual molecular dynamics)[4] is a molecular visualization and analysis tool that includes scripting and plugin interfaces for user-extensibility and automation of complex tasks [31]. State-of-the-art GPUs offer new opportunities for accelerating computationally demanding analyses on desktop computers, which previously required batch mode runs on clusters or supercomputers [30].

While many nongraphical functions within VMD are well suited to GPU acceleration, one particularly computationally demanding feature of the program is the ability to calculate electrostatic potential maps. Electrostatic potential maps can be used for visualization, for placing ions during structure building, and can be time-averaged to better capture regions of structure where ions bind transiently, for example. Full-accuracy direct summation of Coulomb potentials proves to be a versatile method for many analyses, but it is far too computationally expensive for casual use on CPUs, particularly in the case of time-averaged analyses. The inherent data parallelism of the direct summation method makes it extremely well suited to execution on GPUs.

1) Direct Coulomb Summation Algorithm: The direct Coulomb summation algorithm computes the sum of the partial charges of all atoms scaled by their distance to the point in the electrostatic field being evaluated. Since electrostatic potentials can be computed entirely independently from one another, an effective parallel decomposition is achieved by assigning potential evaluations to individual GPU computation threads. Each GPU thread computes its assigned potential values by looping over all of the atoms, summing the results, and storing them out to GPU global memory. A simplified sequential form of the computation for a single point in the electrostatic field is shown in pseudocode below:

```
potential = 0.0;
for atomindex = 1 to numatoms
    r = distance (atomindex, voxelcoordinate);
    potential = potential + (charge[atomindex]/r)
```

2) GPU Kernel Optimization: The CUDA implementation of this algorithm makes use of register-speed constant memory to store atom coordinates and charges, since they are concurrently accessed in the inner loop of all of the GPU threads. Due to the limited size of the constant memory, potentials are evaluated in multiple passes, reloading the constant memory with new atom data in each successive pass. Significant performance improvements were achieved through loop unrolling optimizations and precomputation of partial components of atom distance vectors that are constant over individual rows

[4]http://www.ks.uiuc.edu/Research/vmd/.

Table 2 Direct Coulomb Summation Kernel Performance Results

Kernel	Normalized vs. Intel C	Atom evals per second	GFLOPS
CPU-GCC-SSE	0.052	0.046 billion	0.28
CPU-ICC-SSE	1.0	0.89 billion	5.3
CUDA-Simple	16.6	14.8 billion	178
CUDA-Unroll8x	38.9	34.6 billion	268
CUDA-Unroll8y	40.9	36.4 billion	191
CUDA-Unroll8clx	44.4	39.5 billion	291

and planes within the potential grid. GPU shared memory can be used for storage of intermediate potential sums, counteracting register pressure induced by the loop unrolling optimizations. These optimizations decrease the number of memory references per atom potential evaluation, making the kernel compute-bound. The CUDA-Unroll8y kernel precomputes constant portions of atom distance vectors using GPU shared memory, at the cost of limiting CUDA thread block geometries that negatively impact performance on smaller potential maps.

3) Single GPU Performance Results: The performance results in Table 2 compare the performance levels achieved by highly tuned CPU kernels using SSE instructions versus CUDA GPU kernels, all implemented in C. Benchmarks were run on a system consisting of a 2.6 GHz Intel QX6700 quad-core CPU with GeForce 8800 GTX GPUs. The CPU-based SSE kernels take advantage of all algorithmic optimizations and measure peak performance obtainable without resorting to assembly language. The CUDA-Simple GPU kernel illustrates performance gains achieved with a direct mapping of the computation to GPU threads, without the arithmetic or loop unrolling optimizations used in the other GPU kernels. The CUDA-Unroll8y kernel uses shared memory and additional precomputation techniques to achieve high performance for the number of floating-point arithmetic operations, providing an example of the utility of these techniques, though it falls short of the peak-performing CUDA-Unroll8clx kernel. The CUDA-Unroll8clx kernel reorganizes global memory references so that they occur only at the end of the kernel, eliminating the need for shared memory storage as a means of reducing register pressure. Like the CUDA-Unroll8y kernel, this kernel benefits from global memory coalescing. The CUDA-Unroll8clx kernel outperforms all others due to the comparative simplicity of its inner loop, and even though it does significantly more arithmetic than the CUDA-Unroll8y kernel. All floating-point arithmetic operations are counted as 1 flop with the exception of multiply–add and reciprocal–sqrt, which are counted as 2 flops.

4) Parallel Multi-GPU Runs: When calculating potentials for multimillion atom structures and particularly when time-averaging the potential calculations, ample parallelism is available to usefully occupy multiple GPUs. Since the amount of computation is identical for each potential calculation, a coarse, statically load-balanced, round robin parallel decomposition of individual 2-D potential map slices onto the pool of GPUs works very well. For each GPU, a CPU thread is created on the host machine and is assigned the task of managing one GPU. Each of these GPU management threads loops over the potential grid, calculating the 2-D slices it is responsible for by running GPU kernels and managing I/O to and from the GPU. The main limitation to parallel scaling of multiple GPUs within a single host system is the bandwidth of the PCI Express bus connecting the GPUs to the host. When possible, multi-GPU applications should ensure that host CPU threads, GPU direct memory access buffers, and GPUs are matched according to CPU affinity, NUMA memory topology, and bus topology yielding the highest bandwidth between the host thread and GPU. Fortunately, in the present instance, for all but the smallest test cases, the computation time dominates and the available bus bandwidth is inconsequential. Benchmarks were run on a system consisting of two dual-core 2.4 GHz AMD Opteron 2216 CPUs with two QuadroPlexes, each containing two Quadro FX 5600 GPUs. Performance scaled linearly with up to four GPUs (maximum GPU capacity of the test system), yielding an aggregate performance of 157 billion atom evaluations per second and 1161 Gflops.

5) Incorporation of CUDA GPU Kernels Into VMD: The software engineering required to integrate the CUDA-based GPU accelerated kernels into VMD was straightforward. In practice, incorporating GPU accelerated kernels into a large application like VMD is quite similar to doing so for highly tuned CPU kernels. While CUDA allows much greater flexibility than previous generation GPU programming environments in terms of the sophistication of data structures and data types that can be used, the performance considerations of state-of-the-art hardware still favor compact, dense data structures that are often rearranged, aligned, or padded for most efficient access by the GPU. CPU-based algorithms using SSE acceleration face similar requirements, so this scenario is not unusual in high-performance scientific software. VMD detects the availability of CUDA GPU acceleration at startup, storing the number and the characteristics of CUDA-capable GPUs. When calls are made to GPU accelerated functions within VMD, a computation strategy routine refers to this information and creates host threads to manage each of the GPUs. If the GPU strategy routine encounters a problem size that cannot be handled by the GPUs or an unrecoverable error occurs, VMD falls back to using multithreaded SSE CPU routines instead. One open problem with this "CPU fallback" approach is that if it occurs during a noninteractive batch mode run, the CPU performance may be tens to hundreds of times slower than a GPU-accelerated run. Such a large performance drop would be a very unwelcome surprise to a researcher waiting for a large

analysis job to complete. In some cases, it may be preferable for such situations to result in termination of the calculation rather than running an unusably slow CPU-based kernel.

VIII. THE FUTURE OF GPU COMPUTING

With the rising importance of GPU computing, GPU hardware and software are changing at a remarkable pace. In the upcoming years, we expect to see several changes to allow more flexibility and performance from future GPU computing systems:

- At Supercomputing 2006, both AMD and NVIDIA announced future support for double-precision floating-point hardware by the end of 2007. The addition of double-precision support removes one of the major obstacles for the adoption of the GPU in many scientific computing applications.
- Another upcoming trend is a higher bandwidth path between CPU and GPU. The PCI Express bus between CPU and GPU is a bottleneck in many applications, so future support for PCI Express 2, HyperTransport, or other high-bandwidth connections is a welcome trend. Sony's PlayStation 3 and Microsoft's XBox 360 both feature CPU–GPU connections with substantially greater bandwidth than PCI Express, and this additional bandwidth has been welcomed by developers. We expect the highest CPU–GPU bandwidth will be delivered by . . .
- . . . future systems, such as AMD's Fusion, that place both the CPU and GPU on the same die. Fusion is initially targeted at portable, not high-performance, systems, but the lessons learned from developing this hardware and its heterogeneous APIs will surely be applicable to future single-chip systems built for performance. One open question is the fate of the GPU's dedicated high-bandwidth memory system in a computer with a more tightly coupled CPU and GPU.
- Pharr notes that while individual stages of the graphics pipeline are programmable, the structure of the pipeline as a whole is not [32], and proposes future architectures that support not just programmable shading but also a programmable pipeline. Such flexibility would lead to not only a greater variety of viable rendering approaches but also more flexible general-purpose processing.
- Systems such as NVIDIA's 4-GPU Quadroplex are well suited for placing multiple coarse-grained GPUs in a graphics system. On the GPU computing side, however, fine-grained cooperation between GPUs is still an unsolved problem. Future API support such as Microsoft's Windows Display Driver Model 2.1 will help multiple GPUs to collaborate on complex tasks, just as clusters of CPUs do today.

A. Top Ten Problems in GPGPU

At Eurographics 2005, the authors presented their list of top ten problems in GPGPU. At the time we hoped that these problems would help focus the efforts of the GPGPU community on broad problems of general interest. In the intervening two years, substantial progress has been made on many of these problems, yet the problems themselves continue to be worthy of further study.

The killer app: Perhaps the most important question facing the community is finding an application that will drive the purchase of millions of GPUs. The number of GPUs sold today for computation is minuscule compared to the overall GPU market of half a billion units per year; a mass-market application that spurred millions of GPU sales, enabling a task that was not previously possible, would mark a major milestone in GPU computing.

Programming models and tools: With the new programming systems in Section IV, the state of the art over the past year has substantially improved. Much of the difficulty of early GPGPU programming has dissipated with the new capabilities of these programming systems, though support for debugging and profiling on the hardware is still primitive. One concern going forward, however, is the proprietary nature of the tools. Standard languages, tools, and APIs that work across GPUs from multiple vendors would advance the field, but it is as yet unclear whether those solutions will come from academia, the GPU vendors, or third-party software companies, large or small.

GPU in tomorrow's computer?: The fate of coprocessors in commodity computers (such as floating-point coprocessors) has been to move into the chipset or onto the microprocessor. The GPU has resisted that trend with continued improvements in performance and functionality and by becoming an increasingly important part of today's computing environments—unlike with CPUs, the demand for continued GPU performance increases has been consistently large. However, economics and potential performance are motivating the migration of powerful GPU functionality onto the chipset or onto the processor die itself. While it is fairly clear that graphics capability is a vital part of future computing systems, it is wholly unclear which part of a future computer will provide that capability, or even if an increasingly important GPU with parallel computing capabilities could absorb a CPU.

Design tradeoffs and their impact on the programming model: GPU vendors are constantly weighing decisions regarding flexibility and features for their programmers: how do those decisions impact the programming model and their ability to build hardware capable of top performance? An illustrative example is data conditionals. Today, GPUs support conditionals on a thread granularity, but conditionals are not free; any GPU today pays a performance penalty for incoherent branches. Programmers want a small branch granularity so each thread can be independent; architects want a large branch granularity to build more efficient hardware. Another important design

decision is thread granularity: less powerful but many lightweight threads versus fewer, more powerful heavyweight threads. As the programmable aspects of the hardware increasingly take center stage, and the GPU continues to mature as a general-purpose platform, these tradeoffs are only increasing in importance.

Relationship to other parallel hardware and software: GPUs are not the only innovative parallel architecture in the field. The Cell Broadband Engine, multicore CPUs, stream processors, and others are all exploiting parallelism in different ways. The future health of GPU computing would benefit if programs written for GPUs run efficiently on other hardware and programs written for other architectures can be run on GPUs. The landscape of parallel computing will continue to feature many kinds of hardware, and it is important that GPUs be able to benefit from advances in parallel computing that are targeted toward a broad range of hardware.

Managing rapid change: Practitioners of GPU computing know that the interface to the GPU changes markedly from generation to generation. This is a very different model than CPUs, which typically maintain API consistency over many years. As a consequence, code written for one generation of GPUs is often no longer optimal or even useful in future generations. However, the lack of backward compatibility is an important key in the ability of GPU vendors to innovate in new GPU generations without bearing the burden of previous decisions. The introduction of the new general-purpose programming environments from the vendors that we described in Section IV may finally mark the beginning of the end of this churn. Historically, CPU programmers have generally been able to write code that would continue to run faster on new hardware (though the current focus on multiple cores may arrest this trend; like GPUs, CPU codes will likely need to be written as parallel programs to continue performance increases). For GPU programmers, however, the lack of backward compatibility and the lack of roadmaps going forward make writing maintainable code for the long term a difficult task.

Performance evaluation and cliffs: The science of program optimization for CPUs is reasonably well understood—profilers and optimizing compilers are effective in allowing programmers to make the most of their hardware. Tools on GPUs are much more primitive—making code run fast on the GPU remains something of a black art. One of the most difficult ordeals for the GPU programmer is the performance cliff, where small changes to the code, or the use of one feature rather than another, make large and surprising differences in performance. The challenge going forward is for vendors and users to build tools that provide better visibility into the hardware and better feedback to the programmer about performance characteristics.

Philosophy of faults and lack of precision: The hardware graphics pipeline features many architectural decisions that favored performance over correctness. For output to a display, these tradeoffs were quite sensible; the difference between perfectly "correct" output and the actual output is likely indistinguishable. The most notable tradeoff is the precision of 32-bit floating-point values in the graphics pipeline. Though the precision has improved, it is still not IEEE compliant, and features such as denorms are not supported. As this hardware is used for general-purpose computation, noncompliance with standards becomes much more important, and dealing with faults—such as exceptions from division by zero, which are not currently supported in GPUs—also becomes an issue.

Broader toolbox for computation and data structures: On CPUs, any given application is likely to have only a small fraction of its code written by its author. Most of the code comes from libraries, and the application developer concentrates on high-level coding, relying on established APIs such as STL or Boost or BLAS to provide lower level functionality. We term this a "horizontal" model of software development, as the program developer generally only writes one layer of a complex program. In contrast, program development for general-purpose computing on today's GPUs is largely "vertical"—the GPU programmer writes nearly all the code that goes into his program, from the lowest level to the highest. Libraries of fundamental data structures and algorithms that would be applicable to a wide range of GPU computing applications (such as NVIDIA's FFT and dense matrix algebra libraries) are only just today being developed but are vital for the growth of GPU computing in the future.

Wedding graphics and GPU computing: One of the most powerful motivations for GPU computing in the near term is the use of general-purpose elements within traditional graphics applications. The GPU physics example of Section VI is an excellent example of such a trend. As new programming environments for GPU computing offer easier and more flexible ways to share data between computing and graphics, and the performance of that sharing improves from the relatively inefficient methods of today, we expect to see an increasing amount of GPU-based general-purpose computation within traditional graphics applications such as games and visualization applications. ■

Acknowledgment

The authors wish to thank E. Elsen, V. Vishal, E. Darve, and V. Pande, as well as P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten, for their contributions to this paper. J. Owens thanks M. Doggett for his helpful comments and figure contribution. J. Phillips and J. Stone thank Prof. W. Hwu and the members of the IMPACT group at the University of Illinois at Urbana-Champaign and D. Kirk and the members of the NVIDIA CUDA development team for their helpful insight and support.

REFERENCES

[1] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. Purcell, "A survey of general-purpose computation on graphics hardware," *Comput. Graph. Forum*, vol. 26, no. 1, pp. 80–113, 2007.

[2] D. Blythe, "The Direct3D 10 system," *ACM Trans. Graph.*, vol. 25, no. 3, pp. 724–734, Aug. 2006.

[3] M. Harris, "Mapping computational concepts to GPUs," in *GPU Gems 2*, M. Pharr, Ed. Reading, MA: Addison-Wesley, Mar. 2005, pp. 493–508.

[4] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream computing on graphics hardware," *ACM Trans. Graph.*, vol. 23, no. 3, pp. 777–786, Aug. 2004.

[5] M. McCool, S. Du Toit, T. Popa, B. Chan, and K. Moule, "Shader algebra," *ACM Trans. Graph.*, vol. 23, no. 3, pp. 787–795, Aug. 2004.

[6] D. Tarditi, S. Puri, and J. Oglesby, "Accelerator: Using data-parallelism to program GPUs for general-purpose uses," in *Proc. 12th Int. Conf. Architect. Support Program. Lang. Oper. Syst.*, Oct. 2006, pp. 325–335.

[7] M. McCool, "Data-parallel programming on the cell BE and the GPU using the RapidMind development platform," in *Proc. GSPx Multicore Applicat. Conf.*, Oct.–Nov. 2006.

[8] PeakStream, *The PeakStream platform: High productivity software development for multi-core processors*. [Online]. Available: http://www.peakstreaminc.com/reference/peakstream_platform_technote.pdf

[9] G. Blelloch, *Vector Models for Data-Parallel Computing*. Cambridge, MA: MIT Press, 1990.

[10] M. Harris, S. Sengupta, and J. D. Owens, "Parallel prefix sum (scan) with CUDA," in *GPU Gems 3*, H. Nguyen, Ed. Reading, MA: Addison-Wesley, Aug. 2007, pp. 851–876.

[11] K. E. Batcher, "Sorting networks and their applications," in *Proc. AFIPS Spring Joint Comput. Conf.*, Apr. 1968, vol. 32, pp. 307–314.

[12] N. K. Govindaraju, M. Henson, M. C. Lin, and D. Manocha, "Interactive visibility ordering of geometric primitives in complex environments," in *Proc. 2005 Symp. Interact. 3D Graph. Games*, Apr. 2005, pp. 49–56.

[13] D. Horn, "Stream reduction operations for GPGPU applications," in *GPU Gems 2*, M. Pharr, Ed. Reading, MA: Addison-Wesley, Mar. 2005, pp. 573–589.

[14] B. Bustos, O. Deussen, S. Hiller, and D. Keim, "A graphics hardware accelerated algorithm for nearest neighbor search," in *Proc. 6th Int. Conf. Comput. Sci.*, May 2006, vol. 3994, pp. 196–199, Lecture Notes in Computer Science.

[15] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha, "Fast computation of database operations using graphics processors," in *Proc. 2004 ACM SIGMOD Int. Conf. Manage. Data*, Jun. 2004, pp. 215–226.

[16] N. K. Govindaraju and D. Manocha, "Efficient relational database management using graphics processors," in *Proc. ACM SIGMOD Workshop Data Manage. New Hardware*, Jun. 2005, pp. 29–34.

[17] J. Krüger, P. Kipfer, P. Kondratieva, and R. Westermann, "A particle system for interactive visualization of 3D flows," *IEEE Trans. Vis. Comput. Graphics*, vol. 11, pp. 744–756, Nov.–Dec. 2005.

[18] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder, "Sparse matrix solvers on the GPU: Conjugate gradients and multigrid," *ACM Trans. Graph.*, vol. 22, no. 3, pp. 917–924, Jul. 2003.

[19] A. E. Lefohn, "A streaming narrow-band algorithm: Interactive computation and visualization of level-set surfaces," Master's thesis, Univ. of Utah, Dec. 2003.

[20] M. Kass, A. Lefohn, and J. Owens, "Interactive depth of field using simulated diffusion on a GPU," Pixar Animation Studios, Tech. Rep. 06-01. [Online]. Available: http://www.graphics.pixar.com/DepthOfField/

[21] J. Krüger and R. Westermann, "Linear algebra operators for GPU implementation of numerical algorithms," *ACM Trans. Graph.*, vol. 22, no. 3, pp. 908–916, Jul. 2003.

[22] K. Fatahalian, J. Sugerman, and P. Hanrahan, "Understanding the efficiency of GPU algorithms for matrix-matrix multiplication," in *Proc. Graph. Hardware 2004*, Aug. 2004, pp. 133–138.

[23] N. Galoppo, N. K. Govindaraju, M. Henson, and D. Manocha, "LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware," in *Proc. ACM/IEEE Conf. Supercomput.*, Nov. 2005, p. 3.

[24] P. M. Hubbard, "Collision detection for interactive graphics applications," *IEEE Trans. Vis. Comput. Graphics*, vol. 1, no. 3, pp. 218–230, 1995.

[25] E. Elsen, V. Vishal, M. Houston, V. Pande, P. Hanrahan, and E. Darve, "N-body simulations on GPUs," Stanford Univ., Stanford, CA, Tech. Rep. [Online]. Available: http://www.arxiv.org/abs/0706.3060

[26] I. Buck, K. Fatahalian, and P. Hanrahan, "GPUBench: Evaluating GPU performance for numerical and scientific applications," in *Proc. 2004 ACM Workshop General-Purpose Comput. Graph. Process.*, Aug. 2004, p. C-20.

[27] E. Lindahl, B. Hess, and D. van der Spoel, "GROMACS 3.0: A package for molecular simulation and trajectory analysis," *J. Mol. Mod.*, vol. 7, pp. 306–317, 2001.

[28] G. Jaychandran, V. Vishal, and V. S. Pande, "Using massively parallel simulations and Markovian models to study protein folding: Examining the Villin head-piece," *J. Chem. Phys.*, vol. 124, no. 6, pp. 164 903–164 914, 2006.

[29] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kale, and K. Schulten, "Scalable molecular dynamics with NAMD," *J. Comp. Chem.*, vol. 26, pp. 1781–1802, 2005.

[30] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten, "Accelerating molecular modeling applications with graphics processors," *J. Comp. Chem.*, vol. 28, pp. 2618–2640, 2007.

[31] W. Humphrey, A. Dalke, and K. Schulten, "VMD-visual molecular dynamics," *J. Mol. Graph.*, vol. 14, pp. 33–38, 1996.

[32] M. Pharr, "Interactive rendering in the post-GPU era," in *Proc. Graph. Hardware 2006*. keynote. [Online]. Available: http://www.graphicshardware.org/previous/www_2006/presentations/pharr-keynote-gh06.pdf

ABOUT THE AUTHORS

John D. Owens received the B.S. degree in electrical engineering and computer sciences from the University of California, Berkeley, in 1995 and the Ph.D. degree in electrical engineering from Stanford University, Stanford, CA, in 2003.

He is an Assistant Professor of Electrical and Computer Engineering at the University of California, Davis. His research interests are in GPU computing (GPGPU) and more broadly commodity parallel hardware and programming models.

Mike Houston received the B.S. degree from the University of California, San Diego, and the M.S degree from Stanford University, Stanford, CA, in 2001 and 2003, respectively, both in computer science. He is currently pursuing the Ph.D. degree in computer science from Stanford University.

His doctoral work is with the Stanford University Graphics Lab researching parallel programming languages, architectures, and algorithms. His research interests are in parallel architectures, programming models, and algorithms. His current research is Sequoia, a programming language centered around a parallel memory hierarchies abstraction, allowing for portability and efficient execution on many parallel architectures.

David Luebke received the B.A. degree in chemistry from Colorado College, Colorado Springs, and the Ph.D. degree in computer science from the University of North Carolina, Chapel Hill.

He is a Research Scientist with NVIDIA Corporation. His principal research interests are general-purpose GPU computing and realistic real-time computer graphics. His specific recent projects include fast multilayer subsurface scattering for realistic skin rendering, temperature-aware graphics architecture, scientific computation on graphics hardware, advanced reflectance and illumination models for real-time rendering, and image-based acquisition of real-world environments. His other projects include *Level of Detail for 3D Graphics,* for which he is lead author, and the Virtual Monticello exhibit in the major exhibition "Jefferson's America and Napoleon's France" at the New Orleans Museum of Art.

Simon Green is a Senior Software Engineer in the Developer Technology Group, NVIDIA, Santa Clara, CA. His work focuses on developing new rendering and simulation techniques and helping application developers take maximum advantage of GPU hardware. He is a regular presenter at the Game Developer and Siggraph conferences and has contributed to the *GPU Gems* series of books. His research interests include physical simulation, image processing, and GPU ray tracing.

John E. Stone received the B.S. and M.S. degrees in computer science from the University of Missouri at Rolla in 1998 and 1994, respectively.

He is a Senior Research Programmer in the Theoretical and Computational Biophysics Group, Beckman Institute for Advanced Science and Technology, University of Illinois at Urbana-Champaign. His research interests include scientific visualization, GPU computing (GPGPU), parallel rendering, virtual reality and haptic interfaces for interactive simulation, and high-performance computing. He is Lead Developer of the VMD molecular visualization and analysis program.

James C. Phillips received the B.S. in physics and mathematics from Marquette University, Milwaukee, Wisconsin, and the Ph.D. degree in physics from the University of Illinois at Urbana-Champaign.

He is a Senior Research Programmer in the Theoretical and Computational Biophysics Group, Beckman Institute for Advanced Science and Technology, University of Illinois at Urbana-Champaign. Since 1999, he has been Lead Developer of the highly scalable parallel molecular dynamics program NAMD. His research interests include improving the performance and accuracy of biomolecular simulations through parallelization, optimization, hardware acceleration, better algorithms, and new methods.

Dr. Phillips received the Gordon Bell Award in 2002.