Volume 26 (2007), number 1 pp. 80–113

COMPUTER GRAPHICS *forum*

A Survey of General-Purpose Computation on Graphics Hardware

John D. Owens[1], David Luebke[2], Naga Govindaraju[3], Mark Harris[2], Jens Krüger[4], Aaron E. Lefohn[5] and Timothy J. Purcell[2]

[1]University of California, Davis, USA
jowens@ece.ucdavis.edu
[2]NVIDIA
{dluebke,mharris,tpurcell}@nvidia.com
[3]Many-core Technology Incubation Group, Microsoft Corporation
nagag@microsoft.com
[4]Technische Universität München
kruegeje@in.tum.de
[5]Neoptica
lefohn@neoptica.com

Abstract

The rapid increase in the performance of graphics hardware, coupled with recent improvements in its programmability, have made graphics hardware a compelling platform for computationally demanding tasks in a wide variety of application domains. In this report, we describe, summarize, and analyze the latest research in mapping general-purpose computation to graphics hardware.

We begin with the technical motivations that underlie general-purpose computation on graphics processors (GPGPU) and describe the hardware and software developments that have led to the recent interest in this field. We then aim the main body of this report at two separate audiences. First, we describe the techniques used in mapping general-purpose computation to graphics hardware. We believe these techniques will be generally useful for researchers who plan to develop the next generation of GPGPU algorithms and techniques. Second, we survey and categorize the latest developments in general-purpose application development on graphics hardware.

Keywords: GPGPU, general-purpose computing on graphics hardware, parallel computing, GPU, graphics hardware, SIMD, stream processing, stream computing, data-parallel computing, high-performance computing, HPC

ACM CCS: I.3.1 Computer Graphics: *Hardware architecture*, I.3.6 Computer Graphics: *Methodology and techniques*, D.2.2 Software Engineering: *Design tools and techniques*.

1. Introduction: Why GPGPU?

Commodity computer graphics chips, known generically as Graphics Processing Units or GPUs, are probably today's most powerful computational hardware for the dollar. Researchers and developers have become interested in harnessing this power for general-purpose computing, an effort known collectively as GPGPU (for "General-Purpose computing on the GPU"). In this article we summarize the principal developments to date in the hardware and software behind GPGPU, give an overview of the techniques and computational building blocks used to map general-purpose computation to graphics hardware, and survey the various general-purpose computing tasks to which GPUs have been applied. We begin by reviewing the motivation for and challenges of general-purpose GPU computing. Why GPGPU?

1.1. Powerful and inexpensive

Recent graphics architectures provide tremendous memory bandwidth and computational horsepower. For example, the flagship NVIDIA GeForce 7900 GTX ($378 as of October 2006) boasts 51.2 GB/sec memory bandwidth; the similarly priced ATI Radeon X1900 XTX can sustain a measured 240 GFLOPS, both measured with GPUBench [BFH04a]. Compare to 8.5 GB/sec and 25.6 GFLOPS theoretical peak

Submitted May 2006
Revised August 2006
Accepted October 2006

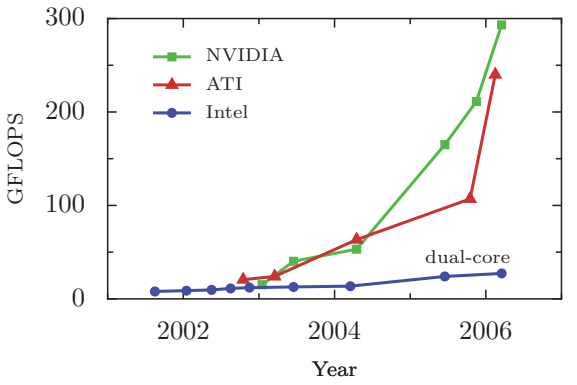

Figure 1: *The programmable floating-point performance of GPUs (measured on the multiply-add instruction, counting 2 floating-point operations per MAD) has increased dramatically over the last four years when compared to CPUs.*

for the SSE units of a dual-core 3.7 GHz Intel Pentium Extreme Edition 965 [Int06]). GPUs also use advanced processor technology; for example, the ATI X1900 contains 384 million transistors and is built on a 90-nanometer fabrication process.

Graphics hardware is fast and getting faster quickly. For example, the arithmetic throughput (again measured by GPUBench) of NVIDIA's current-generation launch product, the GeForce 7800 GTX (165 GFLOPS), more than triples that of its predecessor, the GeForce 6800 Ultra (53 GFLOPS). In general, the computational capabilities of GPUs, measured by the traditional metrics of graphics performance, have compounded at an average yearly rate of 1.7 (pixels/second) to 2.3 (vertices/second). This rate of growth significantly outpaces the often-quoted Moore's Law as applied to traditional microprocessors; compare to a yearly rate of roughly 1.4 for CPU performance [EWN05] (Figure 1).

Why is graphics hardware performance increasing more rapidly than that of CPUs? Semiconductor capability, driven by advances in fabrication technology, increases at the same rate for both platforms. The disparity can be attributed to fundamental architectural differences: CPUs are optimized for high performance on sequential code, with many transistors dedicated to extracting instruction-level parallelism with techniques such as branch prediction and out-of-order execution. On the other hand, the highly data-parallel nature of graphics computations enables GPUs to use additional transistors more directly for computation, achieving higher arithmetic intensity with the same transistor count. We discuss the architectural issues of GPU design further in Section 2.

1.2. Flexible and programmable

Modern graphics architectures have become flexible as well as powerful. Early GPUs were fixed-function pipelines whose output was limited to 8-bit-per-channel color values, whereas modern GPUs now include fully programmable processing units that support vectorized floating-point operations on values stored at full IEEE single precision (but note that the arithmetic operations themselves are not yet perfectly IEEE-compliant). High level languages have emerged to support the new programmability of the vertex and pixel pipelines [BFH*04b,MGAK03,MDP*04]. Additional levels of programmability are emerging with every major generation of GPU (roughly every 18 months). For example, current generation GPUs introduced vertex texture access, full branching support in the vertex pipeline, and limited branching capability in the fragment pipeline. The next generation will expand on these changes and add "geometry shaders", or programmable primitive assembly, bringing flexibility to an entirely new stage in the pipeline [Bly06]. The raw speed, increasing precision, and rapidly expanding programmability of GPUs make them an attractive platform for general-purpose computation.

1.3. Limitations and difficulties

The GPU is hardly a computational panacea. Its arithmetic power results from a highly specialized architecture, evolved and tuned over years to extract maximum performance on the highly parallel tasks of traditional computer graphics. The increasing flexibility of GPUs, coupled with some ingenious uses of that flexibility by GPGPU developers, has enabled many applications outside the original narrow tasks for which GPUs were originally designed, but many applications still exist for which GPUs are not (and likely never will be) well suited. Word processing, for example, is a classic example of a "pointer chasing" application, dominated by memory communication and difficult to parallelize.

Today's GPUs also lack some fundamental computing constructs, such as efficient "scatter" memory operations (i.e., indexed-write array operations) and integer data operands. The lack of integers and associated operations such as bit-shifts and bitwise logical operations (AND, OR, XOR, NOT) makes GPUs ill-suited for many computationally intense tasks such as cryptography (though upcoming Direct3D 10-class hardware will add integer support and more generalized instructions [Bly06]). Finally, while the recent increase in precision to 32-bit floating point has enabled a host of GPGPU applications, 64-bit double precision arithmetic remains a promise on the horizon. The lack of double precision hampers or prevents GPUs from being applicable to many very large-scale computational science problems.

Furthermore, graphics hardware remains difficult to apply to non-graphics tasks. The GPU uses an unusual programming model (Section 2.3), so effective GPGPU programming is not simply a matter of learning a new language. Instead, the computation must be recast into graphics terms by a programmer familiar with the design, limitations, and evolution of the

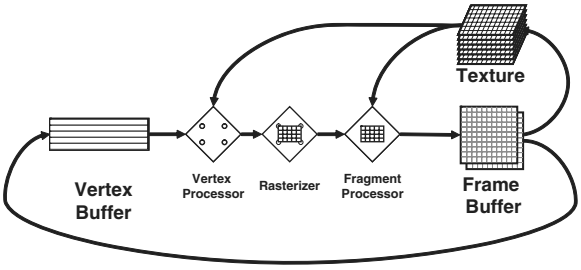

Figure 2: *The modern graphics hardware pipeline. The vertex and fragment processor stages are both programmable by the user.*

underlying hardware. Today, harnessing the power of a GPU for scientific or general-purpose computation often requires a concerted effort by experts in both computer graphics and in the particular computational domain. But despite the programming challenges, the potential benefits—a leap forward in computing capability, and a growth curve much faster than traditional CPUs—are too large to ignore.

1.4. GPGPU today

A vibrant community of developers has emerged around GPGPU (http://GPGPU.org/), and much promising early work has appeared in the literature. We survey GPGPU applications, which range from numeric computing operations, to non-traditional computer graphics processes, to physical simulations and "game physics", to data mining. We cover these and more applications in Section 5.

2. Overview of Programmable Graphics Hardware

In this section we will outline the evolution of the GPU and describe its current hardware and software.

2.1. Overview of the graphics pipeline

The application domain of interactive 3D graphics has several characteristics that differentiate it from more general computation domains. In particular, interactive 3D graphics applications require high computation rates and exhibit substantial parallelism. Building custom hardware that takes advantage of the native parallelism in the application, then, allows higher performance on graphics applications than can be obtained on more traditional microprocessors.

All of today's commodity GPUs structure their graphics computation in a similar organization called the *graphics pipeline*. This pipeline is designed to allow hardware implementations to maintain high computation rates through parallel execution. The pipeline is divided into several stages. All geometric primitives pass through each stage: vertex operations, primitive assembly, rasterization, fragment operations, and composition into a final image. In hardware, each stage is implemented as a separate piece of hardware on the GPU in what is termed a *task-parallel* machine organization. Figure 2 shows the pipeline stages in current GPUs. For more detail on GPU hardware and the graphics pipeline, NVIDIA's GeForce 6 series of GPUs is described by Kilgariff and Fernando [KF05] and Montrym and Moreton [MM05]. From a software perspective, the OpenGL Programming Guide is an excellent reference [OSW*03].

2.2. Programmable hardware

As graphics hardware has become more powerful, one of the primary goals of each new generation of GPU has been to increase the visual realism of rendered images. The graphics pipeline described above was historically a fixed-function pipeline, where the limited number of operations available at each stage of the graphics pipeline were hardwired for specific tasks. However, the success of offline rendering systems such as Pixar's RenderMan [Ups90] demonstrated the benefit of more flexible operations, particularly in the areas of lighting and shading. Instead of limiting lighting and shading operations to a few fixed functions, RenderMan evaluated a user-defined shader program on each primitive, with impressive visual results.

Over the past seven years, graphics vendors have transformed the fixed-function pipeline into a more flexible programmable pipeline. This effort has been primarily concentrated on two stages of the graphics pipeline: the vertex stage and the fragment stage. In the fixed-function pipeline, the vertex stage included operations on vertices such as transformations and lighting calculations. In the programmable pipeline, these fixed-function operations are replaced with a user-defined *vertex program*. Similarly, the fixed-function operations on fragments that determine the fragment's color are replaced with a user-defined *fragment program*.

Each new generation of GPUs has increased the functionality and generality of these two programmable stages. 1999 marked the introduction of the first programmable stage, NVIDIA's register combiner operations that allowed a limited combination of texture and interpolated color values to compute a fragment color. In 2002, ATI's Radeon 9700 led the transition to floating-point computation in the fragment pipeline.

The vital step for enabling general-purpose computation on GPUs was the introduction of fully programmable hardware and an assembly language for specifying programs to run on each vertex [LKM01] or fragment. This programmable shader hardware is explicitly designed to process multiple data-parallel primitives at the same time. As of 2006, the vertex shader and pixel shader standards are both in their third revision, and the OpenGL Architecture Review Board maintains extensions for both [Ope04, Ope03]. The instruction

sets of each stage are limited compared to CPU instruction sets; they are primarily math operations, many of which are graphics-specific. The newest addition to the instruction sets of these stages has been limited control flow operations.

In general, these programmable stages input a limited number of 32-bit floating-point 4-vectors. The vertex stage outputs a limited number of 32-bit floating-point 4-vectors that will be interpolated by the rasterizer; the fragment stage outputs up to 4 floating-point 4-vectors, typically colors. Each programmable stage can access constant registers across all primitives and also read-write registers per primitive. The programmable stages have limits on their numbers of inputs, outputs, constants, registers, and instructions; with each new revision of the vertex shader and pixel [fragment] shader standard, these limits have increased.

GPUs typically have multiple vertex and fragment processors (for example, the ATI Radeon X1900 XTX features 8 vertex processors and 48 fragment processors). Fragment processors have the ability to fetch data from textures, so they are capable of memory *gather*. However, the output address of a fragment is always determined before the fragment is processed—the processor cannot change the output location of a pixel—so fragment processors are incapable of memory *scatter*. Vertex processors recently acquired texture capabilities, and they are capable of changing the position of input vertices, which ultimately affects where in the image pixels will be drawn. Thus, vertex processors are capable of both gather and scatter. Unfortunately, vertex scatter can lead to memory and rasterization coherence issues further down the pipeline. Combined with the lower performance of vertex processors, this limits the utility of vertex scatter in current GPUs.

2.3. Introduction to the GPU programming model

As we discussed in Section 1, GPUs are a compelling solution for applications that require high arithmetic rates and data bandwidths. GPUs achieve this high performance through data parallelism, which requires a programming model distinct from the traditional CPU sequential programming model. In this section, we briefly introduce the GPU programming model using both graphics API terminology and the terminology of the more abstract stream programming model, because both are common in the literature.

The stream programming model exposes the parallelism and communication patterns inherent in the application by structuring data into streams and expressing computation as arithmetic kernels that operate on streams. Purcell *et al.* [PBMH02] characterize their ray tracer in the stream programming model; Owens [Owe05] and Lefohn *et al.* [LKO05] discuss the stream programming model in the context of graphics hardware, and the Brook programming system [BFH*04b] offers a stream programming system for GPUs.

Because typical scenes have more fragments than vertices, in modern GPUs the programmable stage with the highest arithmetic rates is the fragment stage. A typical GPGPU program uses the fragment processor as the computation engine in the GPU. Such a program is structured as follows [Har05a]:

1. First, the programmer determines the data-parallel portions of his application. The application must be segmented into independent parallel sections. Each of these sections can be considered a *kernel* and is implemented as a fragment program. The input and output of each kernel program is one or more data arrays, which are stored (sometimes only transiently) in textures in GPU memory. In stream processing terms, the data in the textures comprise *streams*, and a kernel is invoked in parallel on each stream element.
2. To invoke a kernel, the range of the computation (or the size of the output stream) must be specified. The programmer does this by passing vertices to the GPU. A typical GPGPU invocation is a quadrilateral (quad) oriented parallel to the image plane, sized to cover a rectangular region of pixels matching the desired size of the output array. Note that GPUs excel at processing data in two-dimensional arrays, but are limited when processing one-dimensional arrays.
3. The rasterizer generates a fragment for every pixel location in the quad, producing thousands to millions of fragments.
4. Each of the generated fragments is then processed by the active kernel fragment program. Note that every fragment is processed by the same fragment program. The fragment program can read from arbitrary global memory locations (with texture reads) but can only write to memory locations corresponding to the location of the fragment in the frame buffer (as determined by the rasterizer). The domain of the computation is specified for each input texture (stream) by specifying texture coordinates at each of the input vertices, which are then interpolated at each generated fragment. Texture coordinates can be specified independently for each input texture, and can also be computed on the fly in the fragment program, allowing arbitrary memory addressing.
5. The output of the fragment program is a value (or vector of values) per fragment. This output may be the final result of the application, or it may be stored as a texture and then used in additional computations. Complex applications may require several or even dozens of passes ("multipass") through the pipeline.

While the complexity of a single pass through the pipeline may be limited (for example, by the number of instructions, by the number of outputs allowed per pass, or by the limited control complexity allowed in a single pass), using multiple passes allows the implementation of programs of arbitrary complexity. For example, using an OpenGL simulation with the addition of floating-point compositing operations,

Peercy *et al.* [POAU00] demonstrated that even the fixed-function pipeline, given enough passes, can implement arbitrary RenderMan shaders.

2.4. GPU program flow control

Flow control is a fundamental concept in computation. Branching and looping are such basic concepts that it can be daunting to write software for a platform that supports them to only a limited extent. The latest GPUs support vertex and fragment program branching in multiple forms, but their highly parallel nature requires care in how they are used. This section surveys some of the limitations of branching on current GPUs and describes a variety of techniques for iteration and decision-making in GPGPU programs. Harris and Buck [HB05] provide more detail on GPU flow control.

2.4.1. *Hardware mechanisms for flow control*

There are three basic implementations of data-parallel branching in use on current GPUs: predication, MIMD branching, and SIMD branching.

Architectures that support only predication do not have true data-dependent branch instructions. Instead, the GPU evaluates both sides of the branch and then discards one of the results based on the value of the Boolean branch condition. The disadvantage of predication is that evaluating both sides of the branch can be costly, but not all current GPUs have true data-dependent branching support. The compiler for high-level shading languages like Cg or the OpenGL Shading Language automatically generates predicated assembly language instructions if the target GPU supports only predication for flow control.

In Multiple Instruction Multiple Data (MIMD) architectures that support branching, different processors can follow different paths through the program. In Single Instruction Multiple Data (SIMD) architectures, all active processors must execute the same instructions at the same time. The only MIMD processors in a current GPU are the vertex processors of the NVIDIA GeForce 6 and 7 series and NV40- and G70-based Quadro GPUs. Classifying GPU fragment processors is more difficult. The programming model is effectively Single Program Multiple Data (SPMD), meaning that threads (pixels) can take different branches. However, in terms of architecture and performance, fragment processors on current GPUs process pixels in SIMD groups. Within a SIMD group, when evaluation of the branch condition is identical for all pixels in the group, only the taken side of the branch must be evaluated. However, if one or more of the processors evaluates the branch condition differently, then both sides must be evaluated and the results predicated. As a result, divergence in the branching of simultaneously processed fragments can lead to reduced performance.

2.4.2. *Moving branching up the pipeline*

Because explicit branching can hamper performance on GPUs, it is useful to have multiple techniques to reduce the cost of branching. A useful strategy is to move flow-control decisions up the pipeline to an earlier stage where they can be more efficiently evaluated.

Static Branch Resolution On the GPU, as on the CPU, avoiding branching inside inner loops is beneficial. For example, when evaluating a partial differential equation (PDE) on a discrete spatial grid, an efficient implementation divides the processing into multiple loops: one over the interior of the grid, excluding boundary cells, and one or more over the boundary edges. This *static branch resolution* results in loops that contain efficient code without branches. (In stream processing terminology, this technique is typically referred to as the division of a stream into *substreams*.) On the GPU, the computation is divided into two fragment programs: one for interior cells and one for boundary cells. The interior program is applied to the fragments of a quad drawn over all but the outer one-pixel edge of the output buffer. The boundary program is applied to fragments of lines drawn over the edge pixels. Static branch resolution is further discussed by Goodnight *et al.* [GWL*03], Harris and James [HJ03], and Lefohn *et al.* [LKHW03].

Pre-computation In the example above, the result of a branch was constant over a large domain of input (or range of output) values. Similarly, sometimes the result of a branch is constant for a period of time or a number of iterations of a computation. In this case we can evaluate the branches only when the results are known to change, and store the results for use over many subsequent iterations. This can result in a large performance boost. This technique is used to pre-compute an obstacle offset array in the Navier-Stokes fluid simulation example in the NVIDIA SDK [Har05b].

Z-Cull Precomputed branch results can be taken a step further by using another GPU feature to entirely skip unnecessary work. Modern GPUs have a number of features designed to avoid shading pixels that will not be seen. One of these is Z-cull. Z-cull is a hierarchical technique for comparing the depth (Z) of an incoming block of fragments with the depth of the corresponding block of fragments in the Z-buffer. If the incoming fragments will all fail the depth test, then they are discarded before their pixel colors are calculated in the fragment processor. Thus, only fragments that pass the depth test are processed, work is saved, and the application runs faster. In fluid simulation, "land-locked" obstacle cells can be "masked" with a z-value of zero so that all fluid simulation computations will be skipped for those cells. If the obstacles are fairly large, then a lot of work is saved by not processing these cells. Sander *et al.* described this technique [STM04] together with another Z-cull acceleration technique for fluid simulation, and Harris and Buck provide pseudocode [HB05]. Z-cull was also used by Purcell *et al.* to accelerate GPU ray tracing [PBMH02].

Data-Dependent Looping With Occlusion Queries Another GPU feature designed to avoid drawing what is not visible is the hardware occlusion query (OQ). This feature provides the ability to query the number of pixels updated by a rendering call. These queries are pipelined, which means that they provide a way to get a limited amount of data (an integer count) back from the GPU without stalling the pipeline (which would occur when actual pixels are read back). Because GPGPU applications almost always draw quads with known pixel coverage, OQ can be used with fragment kill functionality to get a count of fragments updated and killed. This allows the implementation of global decisions controlled by the CPU based on GPU processing. Purcell *et al.* demonstrated this in their GPU ray tracer [PBMH02], and Harris and Buck provide pseudocode for the technique [HB05]. Occlusion queries can also be used for subdivision algorithms, such as the adaptive radiosity solution of Coombe *et al.* [CHL04].

2.5. Impact of DX10 hardware

The next major generation of GPUs are widely expected to support Microsoft's Direct3D 10 API, part of Microsoft's DirectX multimedia APIs, and appear sometime from late 2006 to early 2007. Blythe's recent Siggraph paper [Bly06] summarizes the major hardware and software changes that will characterize these upcoming GPUs.

The impact on GPGPU from this new hardware may not be felt for some time as GPGPU developers migrate their applications to the new feature set and become comfortable with the performance aspects of the new hardware and software, but we expect that the following features will be of particular interest for general-purpose computing.

- DX10 introduces a new programmable unit to the pipeline, the *geometry shader*, that is placed after the vertex shader. The input to the GS is an entire primitive. The major difference between the GS and the previous vertex/fragment shaders is that it can output anywhere from 0 to many primitives. This ability to procedurally create new elements is expected to be broadly useful in both graphics tasks (such as shadow volume calculations) and more general-purpose tasks.
- GPGPU application developers have long requested more flexible operations on memory buffers. While operations such as render-to-texture and render-to-vertex-array have partially met these requests, upcoming DX10 hardware promises both greater functionality and greater performance in this area. One new operation in DX10 hardware will be the "stream output", allowing the output of the geometry shader to be directly stored into a memory buffer.
- The new shader model (4.0) associated with DX10 hardware unifies the basic instruction set between the programmable shader units (though each programmable shader still has stage-specific specializations). Along with increases in a variety of shader limits, such as instruction count, register space, and render targets, shader hardware now supports 32-bit integers. This integer capability is expected to both enhance current GPGPU applications (particularly in memory address calculations) as well as enable new ones (such as cryptography). The precision of floating-point computation is also expected to significantly improve in DX10 hardware.

3. Programming Systems

Successful programming for any development platform requires at least three basic components: a high-level language for code development, a debugging environment, and profiling tools. CPU programmers have a large number of well-established languages, debuggers, and profilers to choose from when writing applications. Conversely, GPU programmers have just a small handful of languages to choose from, and few if any full-featured debuggers and profilers.

In this section we look at the high-level languages that have been developed for GPU programming, and the debugging tools that are available for GPU programmers. Code profiling and tuning tends to be a very architecture-specific task. GPU architectures have evolved very rapidly, making profiling and tuning primarily the domain of the GPU manufacturer. As such, we will not discuss code profiling tools in this section.

3.1. High-level shading languages

Most high-level GPU programming languages today share one thing in common: they are designed around the idea that GPUs generate pictures. As such, the high-level programming languages are often referred to as shading languages. That is, they are a high-level language that compiles a shader program into a vertex shader and a fragment shader to produce the image described by the program.

Cg [MGAK03], HLSL [Mic05a], and the OpenGL Shading Language [KBR04] all abstract the capabilities of the underlying GPU and allow the programmer to write GPU programs in a more familiar C-like programming language. They do not stray far from their origins as languages designed to shade polygons. All retain graphics-specific constructs: vertices, fragments, textures, etc. Cg and HLSL provide abstractions that are very close to the hardware, with instruction sets that expand as the underlying hardware capabilities expand. The OpenGL Shading Language was designed looking a bit further out, with many language features (e.g. integers) that do not directly map to hardware available today.

Sh is a shading language implemented on top of C++ [MDP*04]. Sh provides a shader algebra for manipulating and defining procedurally parameterized shaders. Sh manages buffers and textures, and handles shader partitioning

into multiple passes. Sh also provides a stream programming abstraction suitable for GPGPU programming.

Finally, Ashli [BP03] works at a level one step above that of Cg, HLSL, or the OpenGL Shading Language. Ashli reads as input shaders written in HLSL, the OpenGL Shading Language, or a subset of RenderMan. Ashli then automatically compiles and partitions the input shaders to run on a programmable GPU.

3.2. GPGPU languages and libraries

More often than not, the graphics-centric nature of shading languages makes GPGPU programming more difficult than it needs to be. As a simple example, initiating a GPGPU computation usually involves drawing a primitive. Looking up data from memory is done by issuing a texture fetch. The GPGPU program may conceptually have nothing to do with drawing geometric primitives and fetching textures, yet the shading languages described in the previous section force the GPGPU application writer to think in terms of geometric primitives, fragments, and textures. Instead, GPGPU algorithms are often best described as memory and math operations, concepts much more familiar to CPU programmers. The programming systems below attempt to provide GPGPU functionality while hiding the GPU-specific details from the programmer.

The Brook programming language extends ANSI C with concepts from stream programming [BFH*04b]. Brook can use the GPU as a compilation target. Brook streams are conceptually similar to arrays, except all elements can be operated on in parallel. Kernels are the functions that operate on streams. Brook automatically maps kernels and streams into fragment programs and texture memory. Scout is a GPU programming language designed for scientific visualization [MIA*04]. Scout allows runtime mapping of mathematical operations over data sets for visualization.

Accelerator is a system from Microsoft Research that aims to simplify GPGPU programming by providing a high-level data-parallel programming model in a library that is accessible from within traditional imperative programming languages [TPO06]. Accelerator translates data-parallel operations on the fly to GPU pixel shaders, demonstrating significant speedups over C versions running on the CPU.

CGiS is a data-parallel programming language from the Saarland University Compiler Design Lab with similar aims to Brook and Accelerator, but with a slightly different approach [LFW06]. Like Brook, CGiS provides stream data types, but instead of explicit kernels that run on the GPU, the language invokes GPU computation via a built-in data-parallel forall operator.

Finally, the Glift template library provides a generic template library designed to simplify GPU data structure design and separate GPU algorithms from data structures [LKS*06]. Glift defines GPU computation as parallel iteration over the elements of a data structure. The model generalizes the stream computation model and connects GPGPU with CPU-based parallel data structure libraries such as the Standard Template Adaptive Parallel Library (STAPL) [AJR*01]. The library integrates with a C++, Cg, and OpenGL GPU development environment.

Figure 3: *Examples of fragment program "printf" debugging. The left image encodes ray-object intersection hit points as r, g, b color. The right image draws a point at each location where a photon was stored in a photon map. (Images generated by Purcell et al. [PDC*03].)*

3.3. Debugging tools

Until recently, support for debugging on GPUs was fairly limited, and the features necessary for a good GPU debugger were not well defined. The advent of GPGPU programming makes it clear that a GPU debugger should have similar capabilities as traditional CPU debuggers, including variable watches, program break points, and single-step execution. GPU programs often involve user interaction. While a debugger does not need to run the application at full speed, the application being debugged should maintain some degree of interactivity. A GPU debugger should be easy to add to and remove from an existing application, should mangle GPU state as little as possible, and should execute the debug code on the GPU, not in a software rasterizer. Finally, a GPU debugger should support the major GPU programming APIs and vendor-specific extensions.

In many cases, graphically displaying the data for a given set of pixels gives a much better sense of whether a computation is correct than a text box full of numbers would. This visualization is essentially a "printf-style" debug, where the values of interest are printed to the screen. Figure 3 shows some examples of printf-style debugging that many GPGPU programmers have become adept at implementing as part of the debugging process. The ideal GPGPU debugger would automate printf-style debugging, including programmable scale and bias for values outside the display range (e.g. floating point data), while also retaining the true data value at each point if it is needed.

There are a few different systems for debugging GPU programs available to use, but nearly all are missing one or more of the important features we just discussed.

gDEBugger [Gra06] and GLIntercept [Tre06] are tools designed to help debug OpenGL programs. Both are able to capture and log OpenGL state from a program. gDEBugger allows a programmer to set breakpoints and watch OpenGL state variables at runtime, as well as to profile applications using GPU hardware performance signals. There is currently no specific support for debugging shaders, but both support runtime shader editing.

The Microsoft Shader Debugger [Mic05b], however, does provide runtime variable watches and breakpoints for shaders. The shader debugger is integrated into the Visual Studio IDE, and provides all the same functionality programmers are used to for traditional programming. Unfortunately, debugging requires the shaders to be run in software emulation rather than on the hardware. In contrast, the Apple OpenGL Shader Builder [App06b] also has a sophisticated IDE and actually runs shaders in real time on the hardware during shader debug and edit. The downside to this tool is that it was designed for writing shaders, not for computation. The shaders are not run in the context of the application, but in a separate environment designed to help facilitate shader writing.

While many of the tools mentioned so far provide a lot of useful features for debugging, none provide any support for shader data visualization or printf-style debugging. Sometimes this is the single most useful tool for debugging programs. The Image Debugger [Bax06] was among the first tools to provide this functionality by providing a printf-like function over a region of memory. The region of memory gets mapped to a display window, allowing a programmer to visualize any block of memory as an image. The Image Debugger does not provide any special support for shader programs, so programmers must write shaders such that the output gets mapped to an output buffer for visualization.

The Shadesmith Fragment Program Debugger [PS03] was the first system to automate printf-style debugging while providing basic shader debugging functionality like breakpoints, program stepping, and programmable scale and bias for the image printf. While Shadesmith represents a big step in the right direction for GPGPU debugging, it still has many limitations, the largest of which is that Shadesmith is currently limited to debugging assembly language shaders. Additionally, Shadesmith only works for OpenGL fragment programs, and provides no support for debugging OpenGL state.

Finally, Duca *et al.* recently described a system that not only provides debugging for graphics state but also both vertex and fragment programs [DNB*05]. Their system builds a database of graphics state for which the user writes SQL-style queries. Based on the queries, the system extracts the necessary graphics state and program data and draws the appropriate data into a debugging window. The system is build on top of the Chromium [HHN*02] library, enabling debugging of any OpenGL applications without modification to the original source program. This promising approach combines graphics state debugging and program debugging with visualizations in a transparent and hardware-rendered approach.

4. GPGPU Techniques

This section is targeted at the developer of GPGPU libraries and applications. We enumerate the techniques required to efficiently map complex applications to the GPU and describe some of the building blocks of GPU computation.

4.1. Stream operations

Recall from Section 2.3 that the stream programming model is a useful abstraction for programming GPUs. There are several fundamental operations on streams that many GPGPU applications implement as a part of computing their final results: map, reduce, scatter and gather, scan, stream filtering, sort, and search. In the following sections we define each of these operations, and briefly describe a GPU implementation for each.

4.1.1. *Map*

Perhaps the simplest operation, the *map* (or apply) operation operates just like a mapping function in Lisp. Given a stream of data elements and a function, map will apply the function to every element in the stream. A simple example of the map operator is applying scale and bias to a set of input data for display in a color buffer.

The GPU implementation of map is straightforward, and perhaps best illustrated with an example. Assume we have a stream of data with values in the range [0.0 .. 1.0). We would like to convert these values to the range [0..255], perhaps for mapping to a display. A kernel to do this would multiply each element in the stream by 256 and take the *floor* of that value, to produce an output stream in the desired range. This application of a function to an input stream is the essence of the map operation.

4.1.2. *Reduce*

Sometimes a computation requires computing a smaller stream from a larger input stream, possibly to a single element stream. This type of computation is called a *reduction*. For example, a reduction can be used to compute the sum or maximum of all the elements in a stream.

On GPUs, reductions can be performed by alternately rendering to and reading from a pair of textures. On each rendering pass, the size of the output, the computational range, is reduced by one half. In general, we can compute a reduction over a set of n data elements in $O(\frac{n}{p}\log n)$ time steps using the parallel GPU hardware (with p elements processed in one time step), compared to $O(n)$ time steps for a sequential reduction on the CPU. To produce each element of the output, a fragment program reads two values, one from a

corresponding location on either half of the previous pass result buffer, and combines them using the reduction operator (for example, addition or maximum). These passes continue until the output is a one-by-one buffer, at which point we have our reduced result. For a two-dimensional reduction, the fragment program reads four elements from four quadrants of the input texture, and the output size is halved in both dimensions at each step. Buck *et al.* describe GPU reductions in more detail in the context of the Brook programming language [BFH*04b].

4.1.3. *Scatter and gather*

Two fundamental memory operations with which most programmers are familiar are write and read. If the write and read operations access memory *indirectly*, they are called scatter and gather respectively. A *scatter* operation looks like the C code d[a] = v where the value v is being stored into the data array d at address a. A *gather* operation is just the opposite of the scatter operation. The C code for gather looks like v = d[a].

The GPU implementation of gather is essentially a dependent texture fetch operation. A texture fetch from texture d with computed texture coordinates a performs the indirect memory read that defines gather. Unfortunately, scatter is not as straightforward to implement. Fragments have an implicit destination address associated with them: their location in frame buffer memory. A scatter operation would require that a program change the framebuffer write location of a given fragment, or would require a dependent texture write operation. Since neither of these mechanisms exist on today's GPU, GPGPU programmers must resort to various tricks to achieve a scatter. These tricks include rewriting the problem in terms of gather; tagging data with final addresses during a traditional rendering pass and then sorting the data by address to achieve an effective scatter; and using the vertex processor to scatter (since vertex processing is inherently a scattering operation). Buck has described these mechanisms for changing scatter to gather in greater detail [Buc05b].

4.1.4. *Scan*

A simple and common parallel algorithmic building block is the *all-prefix-sums* operation, also known as *scan* [HS86]. For each element in a sequence of elements, prefix-sum computes the sum of all previous elements in the sequence. Blelloch summarized a variety of potential applications of scan [Ble90]. The first implementation of scan on GPUs was presented by Horn and demonstrated for the applications of collision detection and subdivision surfaces [Hor05]. Hensley *et al.* used a similar scan implementation to generate summed-area tables on the GPU [HSC*05]. The algorithms of Horn and Hensley *et al.* were efficient in the number of passes ($O(\log n)$) executed, but required $O(n \log n)$ total work, a factor of $\log n$ worse than the optimal sequential work complexity of $O(n)$. Sengupta *et al.* and Greß *et al.* presented $O(n)$ algorithms for GPUs [SLO06, GGK06]. Greß *et al.* construct a list of potentially intersecting bounding box pairs and utilize scan to remove the non-intersecting pairs. The algorithm of Sengupta *et al.* is notable for its method of switching from a tree-based work-efficient algorithm to Horn's brute-force algorithm as it approaches the root of the tree. This hybrid approach more efficiently uses all of the parallelism provided by the GPU.

4.1.5. *Stream filtering*

Many algorithms require the ability to select a subset of elements from a stream, and discard the rest. The location and number of elements to be filtered is variable and not known a priori. Example algorithms that benefit from this *stream filtering* operation include simple data partitioning (where the algorithm only needs to operate on stream elements with positive keys and is free to discard negative keys) and collision detection (where only objects with intersecting bounding boxes need further computation).

Horn has described a technique called stream compaction [Hor05] that implements stream filtering on the GPU. Using a combination of scan (Section 4.1.4) and search, stream filtering can be achieved in $O(\log n)$ passes.

4.1.6. *Sort*

A *sort* operation allows us to transform an unordered set of data into an ordered set of data. Sorting is a classic algorithmic problem that has been solved by several different techniques on the CPU. Many of these algorithms are data-dependent and generally require scatter operations; therefore, they are not directly applicable to a clean GPU implementation. Recall from Section 2.4 that data-dependent operations are difficult to implement efficiently, and we just saw in Section 4.1.3 that scatter is not implemented for fragment processors on today's GPUs. To make efficient use of GPU resources, a GPU-based sort should be oblivious to the input data, and should not require scatter.

Most GPU-based sorting implementations [BP04, CND03, GZ06, KSW04, KW05a, PDC*03, Pur04] have been based on sorting networks. The main idea behind a sorting network is that a given network configuration will sort input data in a fixed number of steps, regardless of the input data. Additionally, all the nodes in the network have a fixed communication pattern. The fixed communication pattern means the problem can be stated in terms of gather rather than scatter, and the fixed number of stages for a given input size means the sort can be implemented without data-dependent branching. This yields an efficient GPU-based sort, with an overall $O(n \log^2 n)$ computational complexity.

Kipfer *et al.* and Purcell *et al.* implement a bitonic merge sort [Bat68] and Callele *et al.* use a periodic balanced sorting network [DPRS89]. The implementation details of each

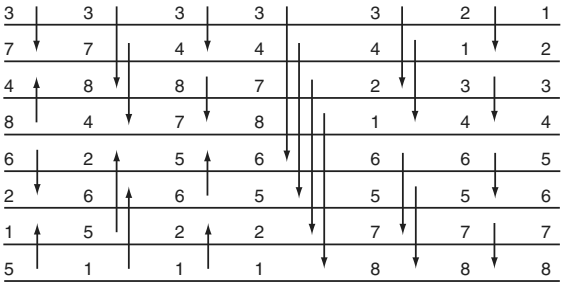

Figure 4: *A simple parallel bitonic merge sort of eight elements requires six passes. Elements at the head and tail of each arrow are compared, with larger elements moving to the head of the arrow.*

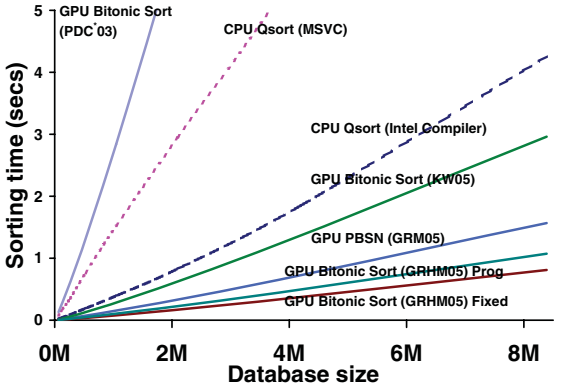

Figure 5: *Performance of CPU-based and GPU-based sorting algorithms on 16-bit floating point values. The CPU-based Qsort available in the Intel compiler is optimized using hyperthreading and SSE instructions. We observe that the cache-efficient GPU-based sorting network algorithm is nearly 6 times faster than the optimized CPU implementation on a 3.4 GHz PC with an NVIDIA GeForce 6800 Ultra GPU. Furthermore, the fixed-function pipeline implementation described by Govindaraju et al. [GRH*05] is nearly 1.2 times faster than their implementation with fragment programs.*

technique vary, but the high-level strategy for each is the same. The data to be sorted is stored in texture memory. Each of the fixed number of stages for the sort is implemented as a fragment program that does a compare-and-swap operation. The fragment program simply fetches two texture values, and based on the sort parameters, determines which of them to write out for the next pass. Figure 4 shows a simple bitonic merge sort.

Sorting networks can also be implemented efficiently using the texture mapping and blending functionalities of the GPU [GRM05]. In each step of the sorting network, a comparator mapping is created at each pixel on the screen and the color of the pixel is compared against exactly one other pixel. The comparison operations are implemented using the blending functionality and the comparator mapping is implemented using the texture mapping hardware, thus entirely eliminating the need for fragment programs. Govindaraju *et al.* [GRH*05] have also analyzed the cache efficiency of sorting network algorithms and presented an improved bitonic sorting network algorithm with a better data access pattern and data layout. The precision of the underlying sorting algorithm using comparisons with fixed-function blending hardware is limited to the precision of the blending hardware. For example, the current blending hardware has 16-bit floating point precision. Alternatively, the limitation to 16-bit values on current GPUs can be alleviated by using a single-line fragment program for evaluating the conditionals, but the fragment program implementation on current GPUs is slightly slower than the fixed-function pipeline. Figure 5 highlights the performance of different GPU-based and CPU-based sorting algorithms on different sequences composed of 16-bit floating point values using a high-end PC. A sorting library implementing the algorithm for 16-bit and 32-bit floats is freely available for noncommercial use [GPU06].

Greß and Zachmann [GZ06] present a novel algorithm, GPU-ABiSort, to further enhance the sorting performance on GPUs. Their algorithm is based on an adaptive bitonic sorting algorithm and achieves an optimal performance of $O(n \log n)$ for any computation time T in the range of $O(\log^2 n) \leq T \leq O(n \log n)$. The algorithm maps well to the GPU and is able to achieve comparable performance to GPUSort [GPU06] on an NVIDIA 7800 GTX GPU.

GPUs have also been used to efficiently perform 1-D and 3-D adaptive sorting of sequences [GHLM05]. Unlike sorting network algorithms, the computational complexity of adaptive sorting algorithms is dependent on the extent of disorder in the input sequence, and work well for nearly-sorted sequences. The extent of disorder is computed using Knuth's measure of disorder. Given an input sequence I, the measure of disorder is defined as the minimal number of elements that need to be removed for the rest of the sequence to remain sorted. The algorithm proceeds in multiple iterations. In each iteration, the unsorted sequence is scanned twice. In the first pass, the sequence is scanned from the last element to the first, and an increasing sequence of elements M is constructed by comparing each element with the current minimum. In the second pass, the sorted elements in the increasing sequence are computed by comparing each element in M against the current minimum in $I - M$. The overall algorithm is simple and requires only comparisons against the minimum of a set of values. The algorithm is, therefore, useful for fast 3D visibility ordering of elements where the minimum comparisons are implemented using the depth buffer [GHLM05].

External memory sorting algorithms are used to organize large terabyte-scale datasets. These algorithms proceed in two phases and use limited main memory to order the data. Govindaraju *et al.* [GGKM06] present a novel external memory sorting algorithm to sort billion-record wide-key databases using a GPU. In the first phase, GPUTeraSort pipelines the following tasks on the CPU, disk controller and GPU: read disk asynchronously, build keys, sort using a GPU, generate runs and write disk. In this phase, GPUTeraSort uses the data parallelism and high memory bandwidth on GPUs to quickly sort large runs. In the second phase, GPUTeraSort uses a similar task pipeline to read, merge and write the runs. GPUTeraSort offloads the compute-intensive and memory-intensive tasks to the GPU; therefore, it is able to achieve higher I/O performance and better memory performance than CPU-only algorithms. In practice, GPUTeraSort outperforms the Indy PennySort[1] record and is able to achieve the best reported price-to-performance on large databases.

4.1.7. *Search*

The last stream operation we discuss, *search*, allows us to find a particular element within a stream. Search can also be used to find the set of nearest neighbors to a specified element. Nearest neighbor search is used extensively when computing radiance estimates in photon mapping (Section 5.4.2) and in database queries (e.g. find the 10 nearest restaurants to point X). When searching, we will use the parallelism of the GPU not to decrease the latency of a single search, but rather to increase search throughput by executing multiple searches in parallel.

Binary Search The simplest form of search is the binary search. This is a basic algorithm, where an element is located in a sorted list in $O(\log n)$ time. Binary search works by comparing the center element of a list with the element being searched for. Depending on the result of the comparison, the search then recursively examines the left or right half of the list until the element is found, or is determined not to exist.

The GPU implementation of binary search [Hor05, PDC*03, Pur04] is a straightforward mapping of the standard CPU algorithm to the GPU. Binary search is inherently serial, so we can not parallelize lookup of a single element. That means only a single pixel's worth of work is done for a binary search. We can easily perform multiple binary searches on the same data in parallel by sending more fragments through the search program.

Nearest Neighbor Search Nearest neighbor search is a slightly more complicated form of search. In this search, we want to find the k nearest neighbors to a given element. On the CPU, this has traditionally been done using a k-d tree [Ben75]. During a nearest neighbor search, candidate elements are maintained in a priority queue, ordered by distance from the "seed" element. At the end of the search, the queue contains the nearest neighbors to the seed element.

Unfortunately, the GPU implementation of nearest neighbor search is not as straightforward. We can search a k-d tree data structure [FS05], but it is difficult to efficiently maintain a priority queue. The important detail about the priority queue is that candidate neighbors can be removed from the queue if closer neighbors are found. Purcell *et al.* propose a data structure for finding nearest neighbors called the kNN-grid [PDC*03, Pur04]. The grid approximates a nearest-neighbor search, but is unable to reject candidate neighbors once they are added to the list. The quality of the search then depends on the density of the grid and the order in which candidate neighbors are visited during the search. The next section of this article discusses GPGPU data structures like arrays and the kNN-grid.

4.2. Data structures

Every GPGPU algorithm must operate on data stored in an appropriate structure. This section describes the data structures used thus far for GPU computation. Effective GPGPU data structures must support fast and coherent parallel accesses as well as efficient parallel iteration, and must also work within the constraints of the GPU memory model. We first describe this model and explain common patterns seen in many GPGPU structures, then present data structures under three broad categories: dense arrays, sparse arrays, and adaptive arrays. Lefohn *et al.* [LKO05, LKS*06] give a more detailed overview of GPGPU data structures and the GPU memory model.

The GPU Memory Model As described in Section 2.3, GPU data are almost always stored in texture memory. To maintain parallelism, operations on these textures are limited to read-only or write-only access within a kernel. Write access is further limited by the lack of scatter support (Section 4.1.3). Outside of kernels, users may allocate or delete textures, copy data between the CPU and GPU, copy data between GPU textures, or bind textures for kernel access. Lastly, most GPGPU data structures are built using 2D textures for three reasons. First, GPU's 2D memory layout and rasterization pattern (i.e., iteration traversal pattern) are closely coupled to deliver the best possible memory access pattern. Second, the maximum 1D texture size is often too small for most problems, and third, current GPUs cannot efficiently write to a slice of a 3D texture.

Iteration In modern C/C++ programming, algorithms are defined in terms of iteration over the elements of a data structure. The stream programming model described in Section 2.3 performs an implicit data-parallel iteration over a stream. Iteration over a dense set of elements is usually accomplished by drawing a single large quad. This is the computation model supported by Brook, Sh, and Scout. Complex structures, however, such as sparse arrays, adaptive arrays, and grid-of-list

[1] http://research.microsoft.com/barc/SortBenchmark

structures often require more complex iteration constructs [BFGS03, KW03, LKHW04]. These range iterators are usually defined using numerous smaller quads, lines, or point sprites.

Generalized Arrays via Address Translation The majority of data structures used thus far in GPGPU programming are random-access multidimensional containers, including dense arrays, sparse arrays, and adaptive arrays. Lefohn *et al.* [LKS*06] show that these virtualized grid structures share a common design pattern. Each structure defines a virtual grid domain (the problem space), a physical grid domain (usually a 2D texture), and an address translator between the two domains. A simple example is a 1D array represented with a 2D texture. In this case, the virtual domain is 1D, the physical domain is 2D, and the address translator converts between them [LKO05, PBMH02].

In order to provide programmers with the abstraction of iterating over elements in the virtual domain, GPGPU data structures must support both virtual-to-physical and physical-to-virtual address translation. For example, in the 1D array example above, an algorithm reads from the 1D array using a virtual-to-physical (1D-to-2D) translation. An algorithm that writes to the array, however, must convert the 2D pixel (physical) position of each stream element to a 1D virtual address before performing computations on 1D addresses. A number of authors describe optimization techniques for pre-computing these address translation operations before the fragment processor [BFGS03, CHL04, KW03, LKHW04]. These optimizations pre-compute the address translation using the CPU, the vertex processor, and/or the rasterizer.

The Brook programming systems provide virtualized interfaces to most GPU memory operations for contiguous, multidimensional arrays. Sh provides a subset of the operations for large 1D arrays. The Glift template library provides virtualized interfaces to GPU memory operations for any structure that can be defined using the programmable address translation paradigm. These systems also define iteration constructs over their respective data structures [BFH*04b, LKS*06, MDP*04].

4.2.1. *Dense arrays*

The most common GPGPU data structure is a contiguous multidimensional array. These arrays are often implemented by first mapping from N-D to 1D, then from 1D to 2D [BFH*04b, PBMH02]. For 3D-to-2D mappings, Harris *et al.* describe an alternate representation, *flat 3D textures*, that directly maps the 2D slices of the 3D array to 2D memory [HBSL03]. Figures 6 and 7 show diagrams of these approaches.

Iteration over dense arrays is performed by drawing large quads that span the range of elements requiring computation. Brook, Glift, and Sh provide users with fully virtualized

Figure 6: *GPU-based multidimensional arrays usually store data in 2D texture memory. Address translators for N-D arrays generally convert N-D addresses to 1D, then to 2D.*

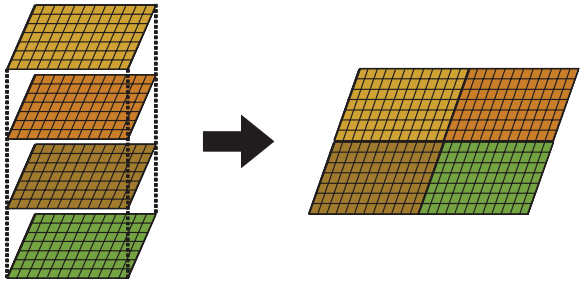

Figure 7: *For the special case of 3D-to-2D conversions or flat 3D textures, 2D slices of the 3D array are packed into a single 2D texture. This structure maintains 2D locality and, therefore, supports native bilinear filtering.*

CPU/GPU interfaces to these structures. Lefohn *et al.* give code examples for optimized implementations [LKO05].

4.2.2. *Sparse arrays*

Sparse arrays are multidimensional structures that store only a subset of the grid elements defined by their virtual domain. Example uses include sparse matrices and implicit surface representations.

Static Sparse Arrays We define *static* to mean that the number and position of stored (non-zero) elements does not change throughout GPU computation, although the GPU computation may update the value of the stored elements. A common application of static sparse arrays is sparse matrices. These structures can use complex, pre-computed packing schemes to represent the active elements because the structure does not change.

Sparse matrix structures were first presented by Bolz *et al.* [BFGS03] and Krüger *et al.* [KW03]. Bolz *et al.* treat each row of a sparse matrix as a separate stream and pack the rows into a single texture. They simultaneously iterate over all rows containing the same number of non-zero elements by drawing a separate small quad for each row. They perform the physical-to-virtual and virtual-to-physical address translations in the fragment stage using a two-level lookup table. In contrast, for random sparse matrices, Krüger *et al.* pack all active elements into vertex buffers and iterate over the structure by drawing a single-pixel point for each element. Each point contains a pre-computed virtual address. Krüger

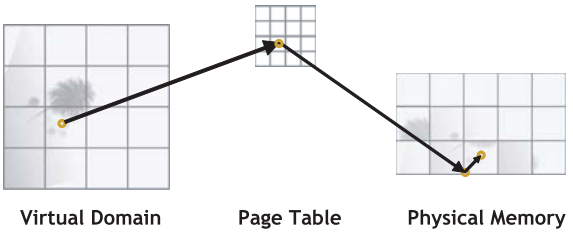

Figure 8: *Page table address data structures can be used to represent dynamic sparse or adaptive GPGPU data structures. For sparse arrays, page tables map only a subset of possible pages to texture memory. Page-table-based adaptive arrays map either uniformly sized physical pages to a varying number of virtual pages or vice versa. Page tables consume more memory than a tree structure but offer constant-time memory accesses and support efficient data-parallel insertion and deletion of pages. Example applications include ray tracing acceleration structures, adaptive shadow maps, and deformable implicit surfaces [LKHW04, LSK*05, PBMH02]. Lefohn et al. describe these structures in detail [LKS*06].*

et al. also describe a packed texture format for banded sparse matrices. Buck *et al.* [BFH*04b] later introduced a sparse matrix Brook example application that performs address translation with only a single level of indirection. The scheme packs the non-zero elements of each row into identically sized streams. As such, the approach applies to sparse matrices where all rows contain approximately the same number of non-zero elements. Section 4.4 contains more detail about GPGPU linear algebra.

Dynamic Sparse Arrays Dynamic sparse arrays are similar to those described in the previous section but support insertion and deletion of non-zero elements during GPU computation. An example application for a dynamic sparse array is the data structure for a deforming implicit surface.

Multidimensional page table address translators are an attractive option for dynamic sparse (and adaptive) arrays because they provide fast data access and can be easily updated. Like the page tables used in modern CPU architectures and operating systems, page table data structures enable sparse mappings by mapping only a subset of possible pages into physical memory. Page table address translators support constant access time and storage proportional to the number of elements in the virtual address space. The translations always require the same number of instructions and are, therefore, compatible with the current fragment processor's SIMD architecture. Figure 8 shows a diagram of a sparse 2D page table structure.

Lefohn *et al.* represent a sparse dynamic volume using a CPU-based 3D page table with uniformly-sized 2D physical pages [LKHW04]. They store the page table on the CPU, the physical data on the GPU, and pre-compute all address translations using the CPU, vertex processor, and rasterizer. The GPU creates page allocations and deletion requests by rendering a small bit-vector message. The CPU decodes this message and performs the requested memory management operations. Strzodka *et al.* use a page discretization and similar message-passing mechanism to define sparse iteration over a dense array [ST04]. Lefebvre *et al.* [LDN04] describe using a page table to implement a virtual texture system.

4.2.3. *Adaptive structures*

Adaptive arrays are a generalization of sparse arrays and represent structures such as quadtrees, octrees, kNN-grids, and *k*-d trees. These structures non-uniformly map data to the virtual domain and are useful for very sparse or multiresolution data. Similar to their CPU counterparts, GPGPU adaptive address translators are represented with a tree, a page table, or a hash table. Example applications include ray tracing acceleration structures, photon maps, adaptive shadow maps, and octree textures.

Static Adaptive Structures Purcell *et al.* use a static adaptive array to represent a uniform-grid ray tracing acceleration structure [PBMH02]. The structure uses a one-level, 3D page table address translator with varying-size physical pages. A CPU-based pre-process packs data into the varying-size pages and stores the page size and page origin in the 3D page table. The ray tracer advances rays through the page table using a 3D line drawing algorithm. Rays traverse the variable-length triangle lists one render pass at a time. The conditional execution techniques described in Section 2.4 are used to avoid performing computation on rays that have reached the end of the triangle list.

Foley *et al.* recently introduced the first *k*-d tree for GPU ray tracing [FS05]. A *k*-d tree adaptively subdivides space into axis-aligned bounding boxes whose size and position are determined by the data rather than a fixed grid. Like the uniform grid structure, the query input for their structure is the ray origin and direction and the result is the origin and size of a triangle list. In their implementation, a CPU-based pre-process creates the *k*-d tree address translator and packs the triangle lists into texture memory. They present two new *k*-d tree traversal algorithms that are GPU-compatible and, unlike previous algorithms, do not require the use of a stack. Thrane and Simonsen [TS05] also describe and analyze static adaptive ray tracing acceleration structures. They introduce a GPU bounding-volume hierarchy (BVH) structure and compare it to a GPU *k*-d tree.

Dynamic Adaptive Arrays Purcell *et al.* introduced the first dynamic adaptive GPU array, the kNN-grid photon map [PDC*03]. The structure uses a one-level page table with either variable-sized or fixed-sized pages. They update the variable-page-size version by sorting data elements and searching for the beginning of each page. The fixed-page-size variant limits the number of data elements per page but avoids the costly sorting and searching steps.

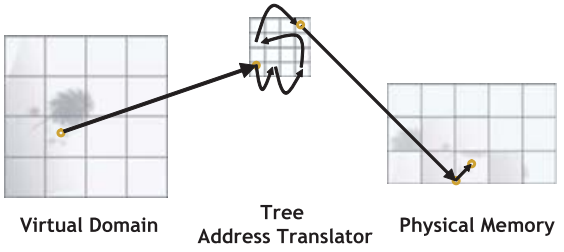

Figure 9: *Tree-based address translators can be used in place of page tables to represent adaptive data structures such as quadtrees, octrees and k-d trees [FS05, LHN05]. Trees consume less memory than page table structures but result in longer access times and are more costly to incrementally update.*

Lefohn *et al.* use a mipmap hierarchy of page tables to define quadtree-like and octree-like dynamic structures [LSK*05, LKS*06]. They apply the structures to GPU-based adaptive shadow mapping and dynamic octree textures. The structure achieves adaptivity by mapping a varying number of virtual pages to uniformly sized physical pages. The page tables consume more memory than a tree-based approach but support constant-time accesses and can be efficiently updated by the GPU. The structures support data-parallel iteration over the active elements by drawing a point sprite for each mapped page and using the vertex processor and rasterizer to pre-compute physical-to-virtual address translations.

In the limit, multilevel page tables are synonymous with N-tree structures. Coombe *et al.* and Lefebvre *et al.* describe dynamic tree-based structures [CHL04, LHN05]. Tree address translators consume less memory than a page table ($O(\log n)$), but result in slower access times ($O(\log n)$) and require non-uniform (non-SIMD) computation. Coombe *et al.* use a CPU-based quadtree translator [CHL04] while Lefebvre *et al.* describe a GPU-based octree-like translator [LHN05]. Figure 9 depicts a tree-based address translator.

4.2.4. *Nonindexable structures*

All the structures discussed thus far support random access and, therefore, trivially support data-parallel accesses. Nonetheless, researchers are beginning to explore non-indexable structures. Ernst *et al.* and Lefohn *et al.* both describe GPU-based stacks [EVG04, LKS*06].

Efficient dynamic parallel data structures are an active area of research. For example, structures such as priority queues (Section 4.1.7), sets, linked lists, and hash tables have not yet been demonstrated on GPUs. While several dynamic adaptive tree-like structures have been implemented, many open problems remain in efficiently building and modifying these structures, and many structures (e.g., *k*-d trees) have not yet been constructed on the GPU. Continued research in understanding the generic components of GPU data structures may also lead to the specification of generic algorithms, such as in those described in Section 4.1.

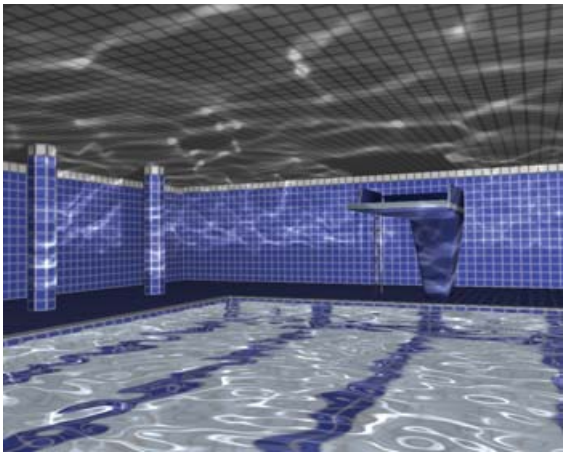

Figure 10: *Solving the wave equation PDE on the GPU allows for fast and stable rendering of water surfaces. (Image generated by Krüger et al. [KBW06].)*

4.3. Differential equations

Differential equations arise in many disciplines of science and engineering. Their efficient solution is necessary for everything from simulating physics for games to detecting features in medical imaging. Typically differential equations are solved for entire arrays of input. For example, physically based simulations of heat transfer or fluid flow typically solve a system of equations representing the temperature or velocity sampled over a spatial domain. This sampling means that there is high data parallelism in these problems, which makes them suitable for GPU implementation.

There are two main classes of differential equations: ordinary differential equations (ODEs) and partial differential equations (PDEs). An ODE is an equality involving a function and its derivatives. An ODE of order n is an equation of the form $F(x, y, \frac{\partial y}{\partial x}, \ldots, \frac{\partial^n y}{\partial x^n}) = 0$ where $\frac{\partial^n y}{\partial x^n}$ is the nth derivative with respect to x. A very simple yet prominent example for ODEs is particle tracing where $\frac{\partial \tilde{x}}{\partial t} = \tilde{v}(\tilde{x}(t), t)$ is to be solved over time (Figure 11). PDEs, on the other hand, are equations involving functions and their partial derivatives, like the wave equation $\frac{\partial^2 \psi}{\partial x^2} + \frac{\partial^2 \psi}{\partial y^2} + \frac{\partial^2 \psi}{\partial z^2} = \frac{\partial^2 \psi}{v^2 \partial t^2}$ (Figure 10). ODEs typically arise in the simulation of the motion of objects, and this is where GPUs have been applied to their solution. Particle system simulation involves moving many point particles according to local and global forces. This results in simple ODEs that can be solved via explicit integration (most have used the well-known Euler, Midpoint, or Runge-Kutta methods). This is relatively simple to implement on the GPU: a simple fragment program is used to update each particle's position and velocity, which

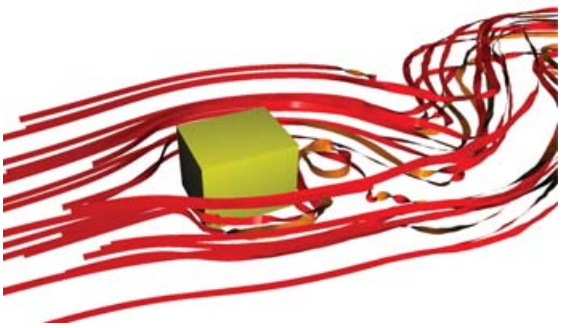

Figure 11: *GPU-computed stream ribbons in a 3D flow field. The entire process from vectorfield interpolation and integration to curl computation, and finally geometry generation and rendering of the stream ribbons, is performed on the GPU [KKKW05].*

are stored as 3D vectors in textures. Kipfer *et al.* presented a method for simulating particle systems on the GPU including inter-particle collisions by using the GPU to quickly sort the particles to determine potential colliding pairs [KSW04]. In simultaneous work, Kolb *et al.* produced a GPU particle system simulator that supported accurate collisions of particles with scene geometry by using GPU depth comparisons to detect penetration [KLRS04]. Krüger *et al.* presented a scientific flow exploration system that supports a wide variety of of visualization geometries computed entirely on the GPU [KKKW05] (Figure 11). A simple GPU particle system example is provided in the NVIDIA SDK [Gre04]. Nyland *et al.* extended this example to add *n*-body gravitational force computation [NHP04]. Related to particle systems is cloth simulation. Green demonstrated a very simple GPU cloth simulation using Verlet integration [Ver67] with basic orthogonal grid constraints [Gre03]. Zeller extended this with shear constraints which can be interactively broken by the user to simulate cutting of the cloth into multiple pieces [Zel05].

When solving PDEs, the two common methods of sampling the domain of the problem are finite differences and finite element methods (FEM). The former has been much more common in GPU applications due to the natural mapping of regular grids to the texture sampling hardware of GPUs. Most of this work has focused on solving the pressure-Poisson equation that arises in the discrete form of the Navier-Stokes equations for incompressible fluid flow. Among the numerical methods used to solve these systems are the conjugate gradient method (Bolz *et al.* [BFGS03] and Krüger and Westermann [KW03]), the multigrid method (Bolz *et al.* [BFGS03] and Goodnight *et al.* [GWL*03]), and simple Jacobi and red-black Gauss-Seidel iteration (Harris *et al.* [HBSL03]).

The earliest work on using GPUs to solve PDEs was done by Rumpf and Strzodka, who mapped mathematical structures like matrices and vectors to textures and linear algebra operations to GPU features such as blending and the OpenGL imaging subset. They applied the GPU to segmentation and non-linear diffusion in image processing [RS01b, RS01a] and used GPUs to solve finite element discretizations of PDEs like the anisotropic heat equation [RS01c]. Recent work by Rumpf and Strzodka [RS05] discusses the use of finite element schemes for PDE solvers on GPUs in detail. Lefohn and Whitaker applied GPUs to the solution of sparse, non-linear PDEs (level-set equations) for volume segmentation [LW02, Lef03].

4.4. Linear algebra

As GPU flexibility has increased over the last decade, researchers were quick to realize that many linear algebraic problems map very well to the pipelined SIMD hardware in these processors. Furthermore, linear algebra techniques are of special interest for many real-time visual effects important in computer graphics. A particularly good example is fluid simulation (Section 5.2), for which the results of the numerical computation can be computed in and displayed directly from GPU memory.

Larsen and McAllister described an early pre-floating-point implementation of matrix multiplies. Adopting a technique from parallel computing that distributes the computation over a logically cube-shaped lattice of processors, they used 2D textures and simple blending operations to perform the matrix product [LM01]. Thompson *et al.* proposed a general computation framework running on the GPU vertex processor; among other test cases they implemented some linear algebra operations and compared the timings to CPU implementations. Their test showed that especially for large matrices a GPU implementation has the potential to outperform optimized CPU solutions [THO02].

With the availability of 32-bit IEEE floating point textures and more sophisticated shader functionality in 2003, Hillesland *et al.* presented numerical solution techniques to least squares problems [HMG03]. Bolz *et al.* [BFGS03] presented a representation for matrices and vectors. They implemented a sparse matrix conjugate gradient solver and a regular-grid multigrid solver for GPUs, and demonstrated the effectiveness of their approach by using these solvers for mesh smoothing and solving the incompressible Navier-Stokes equations. Goodnight *et al.* presented another multigrid solver; their solution focused on an improved memory layout of the domain [GWL*03] that avoids the context-switching latency that arose with the use of OpenGL pbuffers.

Other implementations avoided this pbuffer latency by using the DirectX API. Moravánszky [Mor02] proposed a GPU-based linear algebra system for the efficient representation of dense matrices. Krüger and Westermann took a broader approach and presented a general linear algebra framework supporting basic operations on GPU-optimized

Figure 12: *This image shows a 2D Navier-Stokes fluid flow simulation with arbitrary obstacles. It runs on a staggered 512 by 128 grid. Even with additional features like vorticity confinement enabled, such simulations perform at about 300 fps on current GPUs such as ATI's Radeon X1800 [KW03].*

representations of vectors, dense matrices, and multiple types of sparse matrices [KW03].

Using this set of operations, encapsulated into C++ classes, Krüger and Westermann enabled more complex algorithms to be built without knowledge of the underlying GPU implementation [KW03]. For example, a conjugate gradient solver was implemented with fewer than 20 lines of C++ code. This solver in turn can be used for the solution of PDEs such as the Navier-Stokes equations for fluid flow (Figure 12).

Apart from their applications in numerical simulation, linear algebra operators can be used for GPU performance evaluation and comparison to CPUs. For instance Brook [BFH*04b] featured a spMatrixVec test that used a padded compressed sparse row format.

Galoppo *et al.* [GGHM05] presented an approach to efficiently solve dense linear systems. In contrast to the sparse matrix approaches, they stored the entire matrix as a single 2D texture, allowing them to efficiently modify matrix entries. The results show that even for dense matrices the GPU is able to outperform highly optimized ATLAS implementations.

Instead of focusing on iterative methods, Kass *et al.* recently described a direct tridiagonal linear solver on GPUs used for interactive depth of field simulation [KLO06].

A general evaluation of the suitability of GPUs for linear algebra operations was done by Fatahalian *et al.* [FSH04]. They focused on matrix-matrix multiplication and discovered that these operations are strongly limited by memory bandwidth when implemented on the GPU. They explained the reasons for this behavior and proposed architectural changes to further improve GPU linear algebra performance. To better adapt to such future hardware changes and to address vendor-specific hardware differences, Jiang and Snir presented a first evaluation of automatically tuning GPU linear algebra code [JS05].

The major limitation of all of these approaches is the lack of double precision on current GPUs making them unusable for certain applications. To overcome this issue Göddeke *et al.* [GST06] analyzed native, emulated, and mixed precision approaches for systems of linear equations as they typically arise in the FEM context. They reported speedups of four to five for a mixed precision CPU-GPU over a native CPU implementation. Later Strzodka and Göddeke showed that these iterative refinement methods can be generalized to arbitrary convergent iterative processes [SG06].

4.5. Data queries

In this section, we provide a brief overview of the basic database queries that can be performed efficiently on a GPU [GLW*04].

Given a relational table `T` of m attributes $(a_1, a_2, \ldots, a_m)$, a basic SQL query takes the form

```
Select A
from T
where C
```

where `A` is a list of attributes or aggregations defined on individual attributes and `C` is a Boolean combination of *predicates* that have the form a_i **op** a_j or a_i **op** *constant*. The operator **op** may be any of the following: $=, \neq, >, \geq, <, \leq$. Broadly, SQL queries involve three categories of basic operations: predicates, Boolean combinations, aggregations, and join operations and are implemented efficiently using graphics processors as follows:

Predicates We can use the depth test and the stencil test functionality for evaluating predicates in the form of a_i **op** *constant*. Predicates involving comparisons between two attributes, a_i **op** a_j, are transformed to $(a_i - a_j)$ **op** 0 using the programmable pipeline and are evaluated using the depth and stencil tests.

Boolean combinations A Boolean combination of predicates is expressed in a conjunctive normal form. The stencil test can be used repeatedly to evaluate a series of logical operators with the intermediate results stored in the stencil buffer.

Aggregations These include simple operations such as COUNT, AVG, and MAX. The COUNT query can be implemented using the counting capability of the occlusion queries. The SUM query is computed as $\sum_{i=0}^{k} 2^i$ COUNT(`Select` a_j from T where $a_j = 2^i$) where k is the number of bits in the data representation. Similarly, the AVG and MAX queries are implemented using COUNT operation at each bit-location in the data representation.

Join Operations Join operations combine the records in multiple relations with a common key attribute. They are computationally expensive, and can be accelerated by sorting the records based on the join key. The fast sorting algorithms described in Section 4.1.6 are used to efficiently order the records based on the join key [GM05].

The attributes of each database record are stored in the multiple channels of a single texel, or in the same texel location of multiple textures, and are accessed at run-time to evaluate the queries.

The above queries are key routines used in general relational databases. Bustos *et al.* [BDHK06] presented nearest neighbor search algorithms for point datasets using GPUs. The nearest neighbor queries often arise in multimedia databases, data mining and information retrieval applications. They represent datasets using multiple 2D textures, and at run time, a simple fragment program is used to compute the distance between the dataset vectors and the input vector. Then, the nearest neighbor is computed using a parallel reduction operation on the GPU. In order to handle large high-dimensional data sets, their technique partitions the dataset into blocks. The algorithm is iteratively applied to each block and the nearest neighbor result is readback to the CPU.

5. GPGPU applications

Using many of the algorithms and techniques described in the previous section, in this section we survey the broad range of applications and tasks implemented on graphics hardware.

5.1. Early work

The use of computer graphics hardware for general-purpose computation has been an area of active research for many years, beginning on machines like the Ikonas [Eng78], the Pixel Machine [PH89], and Pixel-Planes 5 [FPE*89]. Pixar's Chap [LP84] was one of the earliest processors to explore a programmable SIMD computational organization, on 16-bit integer data; Flap [LHPL87], described three years later, extended Chap's integer capabilities with SIMD floating-point pipelines. These early graphics computers were typically graphics compute servers rather than desktop workstations. Early work on procedural texturing and shading was performed on the UNC Pixel-Planes 5 and PixelFlow machines [RTB*92, OL98]. This work can be seen as precursor to the high-level shading languages in common use today for both graphics and GPGPU applications. The PixelFlow SIMD graphics computer [EMP*97] was also used to crack UNIX password encryption [KI99].

The wide deployment of GPUs in the last several years has resulted in an increase in experimental research with graphics hardware. The earliest work on desktop graphics processors used non-programmable ("fixed-function") GPUs. Lengyel *et al.* used rasterization hardware for robot motion planning [LRDG90]. Hoff *et al.* described the use of z-buffer techniques for the computation of Voronoi diagrams [HCK*99] and extended the method to proximity detection [HZLM01]. Bohn used fixed-function graphics hardware in the computation of artificial neural networks [Boh98]. Convolution and wavelet transforms with the fixed-function pipeline were realized by Hopf and Ertl [HE99a, HE99b].

Programmability in GPUs first appeared in the form of vertex programs combined with a limited form of fragment programmability via extensive user-configurable texture addressing and blending operations. While these don't constitute a true ISA, so to speak, they were abstracted in a very simple shading language in Microsoft's pixel shader version 1.0 in Direct3D 8.0. Trendall and Stewart gave a detailed summary of the types of computation available on these GPUs [TS00]. Thompson *et al.* used the programmable vertex processor of an NVIDIA GeForce 3 GPU to solve the 3-Satisfiability problem and to perform matrix multiplication [THO02]. A major limitation of this generation of GPUs was the lack of floating-point precision in the fragment processors. Strzodka showed how to combine multiple 8-bit texture channels to create virtual 16-bit precise operations [Str02], and Harris analyzed the accumulated error in boiling simulation operations caused by the low precision [Har02]. Strzodka constructed and analyzed special discrete schemes which, for certain PDE types, allow reproduction of the qualitative behavior of the continuous solution even with very low computational precision, e.g. 8 bits [Str04].

5.2. Physically based simulation

Early GPU-based physics simulations used cellular techniques such as cellular automata (CA). Greg James of NVIDIA demonstrated the "Game of Life" cellular automata and a 2D physically based wave simulation running on NVIDIA GeForce 3 GPUs [Jam01a, Jam01b, Jam01c]. Harris *et al.* used a Coupled Map Lattice (CML) to simulate dynamic phenomena that can be described by partial differential equations, such as boiling, convection, and chemical reaction-diffusion [HCSL02]. The reaction-diffusion portion of this work was later extended to a finite difference implementation of the Gray-Scott equations using floating-point-capable GPUs [HJ03]. Kim and Lin used GPUs to simulate dendritic ice crystal growth [KL03]. Related to cellular techniques are lattice simulation approaches such as Lattice-Boltzmann Methods (LBM), used for fluid and gas simulation. LBM represents fluid velocity in "packets" traveling in discrete directions between lattice cells. Li *et al.* have used GPUs to apply LBM to a variety of fluid flow problems [LWK03, LFWK05].

Full floating point support in GPUs has enabled the next step in physically based simulation: finite difference and finite element techniques for the solution of systems of partial differential equations (PDEs). Spring-mass dynamics on a mesh were used to implement basic cloth simulation on a GPU [Gre03, Zel05]. Several researchers have also implemented particle system simulation on GPUs (Section 4.3).

Several groups have used the GPU to successfully simulate fluid dynamics. Four papers in the summer

of 2003 presented solutions of the Navier-Stokes equations (NSE) for incompressible fluid flow on the GPU [BFGS03, GWL*03, HBSL03, KW03]. Harris provides an introduction to the NSE and a detailed description of a basic GPU implementation [Har04]. Harris *et al.* combined GPU-based NSE solutions with PDEs for thermodynamics and water condensation and light scattering simulation to implement visual simulation of cloud dynamics [HBSL03]. Recently Hagen *et al.* [HLN06] simulated the dynamics of ideal gases in two and three dimensions described by the Euler equations on the GPU.

Other recent work includes flow calculations around arbitrary obstacles [BFGS03, KW03, LLW04]. Sander *et al.* [STM04] described the use of GPU depth-culling hardware to accelerate flow around obstacles, and sample code that implements this technique is made available by Harris [Har05b]. Rumpf and Strzodka used a quantized FEM approach to solving the anisotropic heat equation on a GPU [RS01c] (Section 4.3). Kolb and Cuntz [KC05] simulated fluids with the SPH method using flattened 3D textures and point sprites on the GPU.

Related to fluid simulation is the visualization of flows, which has been implemented using graphics hardware to accelerate line integral convolution and Lagrangian-Eulerian advection [HWSE99, JEH01, WHE01].

Recently rigid body simulation for computer games has been shown to perform very well on GPUs. Havok [Bon06, GH06] demonstrated an API for rigid body and particle simulation on GPUs, featuring full collisions between rigid bodies and particles, as well as support for simulating and rendering on separate GPUs in a multi-GPU system. Running on a PC with dual NVIDIA GeForce 7900 GTX GPUs and a dual-core AMD Athlon 64 X2 CPU, Havok FX achieves more than a 10x speedup running on GPUs compared to an equivalent, highly optimized multithreaded CPU implementation running on the dual-core CPU alone.

5.3. Signal and image processing

The high computational rates of the GPU have made graphics hardware an attractive target for demanding applications such as those in signal and image processing. Among the most prominent applications in this area are those related to image segmentation (Section 5.3.1) as well as a variety of other applications across the gamut of signal, image, and video processing (Section 5.3.2).

5.3.1. *Segmentation*

The segmentation problem seeks to identify features embedded in 2D or 3D images. A driving application for segmentation is medical imaging. A common problem in medical imaging is to identify a 3D surface embedded in a volume image obtained with an imaging technique such as Magnetic Resonance Imaging (MRI) or Computed Tomograph (CT) Imaging. Fully automatic segmentation is an unsolved image processing research problem. Semi-automatic methods, however, offer great promise by allowing users to interactively guide image processing segmentation computations. GPGPU segmentation approaches have made a significant contribution in this area by providing speedups of more than 10 times and coupling the fast computation to an interactive volume renderer.

Image thresholding is a simple form of segmentation that determines if each pixel in an image is within the segmented region based on the pixel value. Yang and Welch [YW03] used register combiners to perform thresholding and basic convolutions on 2D color images. Their NVIDIA GeForce4 GPU implementation demonstrated a 30% speed increase over a 2.2 GHz Intel Pentium 4 CPU. Viola *et al.* performed threshold-based 3D segmentations combined with an interactive visualization system and observed an approximately 8 times speedup over a CPU implementation [VKG03].

Implicit surface deformation is a more powerful and accurate segmentation technique than thresholding but requires significantly more computation. These *level-set* techniques specify a partial differential equation (PDE) that evolves an initial *seed* surface toward the final segmented surface. The resulting surface is guaranteed to be a continuous, closed surface.

Rumpf and Strzodka were the first to implement level-set segmentation on GPUs [RS01a]. They supported 2D image segmentation using a 2D level-set equation with intensity and gradient image-based forces. Lefohn *et al.* extended that work and demonstrated the first 3D level-set segmentation on the GPU [LW02]. Their implementation also supported a more complex evolution function that allowed users to control the curvature of the evolving segmentation, thus enabling smoothing of noisy data. These early implementations computed the PDE on the entire image despite the fact that only pixels near the segmented surface require computation. As such, these implementations were not faster than highly optimized sparse CPU implementations.

The first GPU-based sparse segmentation solvers came a year later. Lefohn *et al.* [LKHW03, LKHW04] demonstrated a sparse (narrow-band) 3D level-set solver, using the sparse data structure techniques of Section 4.2.2, that provided a speedup of 10–15 times over a highly optimized CPU-based solver [Ins03] (Figure 13). Concurrently, Sherbondy *et al.* presented a GPU-based 3D segmentation solver based on the Perona-Malik PDE [SHN03]. They also performed sparse computation, but had a dense (complete) memory representation. They used the depth culling technique for conditional execution to perform sparse computation. Both of these segmentation systems were integrated with interactive volume renderers.

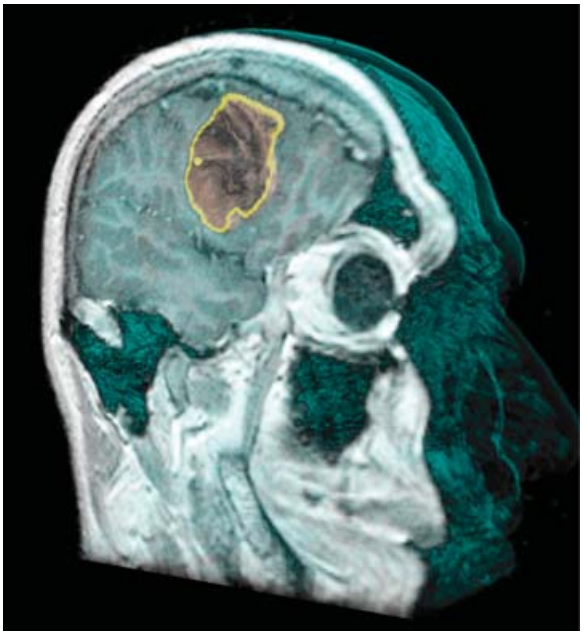

Figure 13: *Interactive volume segmentation and visualization of Magnetic Resonance Imaging (MRI) data on the GPU enables fast and accurate medical segmentations. Image generated by Lefohn et al. [LKHW04].*

Griesser *et al.*'s recent work concentrates specifically on GPU-based foreground/background segmentation for image sequences [GDNV05, Gri05], using an iterative solution run to convergence on a 3×3 neighborhood at each pixel. They incorporate darkness compensation into their algorithm and typically achieve frame times on the order of 4 ms on 640×480 images with an NVIDIA GeForce 6800GT.

5.3.2. *Other signal and image processing applications*

Computer Vision Fung *et al.* use graphics hardware to accelerate image projection and compositing operations [FTM02] in a camera-based head-tracking system [FM04]; their implementation has been released as the open-source OpenVIDIA computer vision library [Ope06], whose website also features a good bibliography of papers for GPU-based computer/machine vision applications.

Yang and Pollefeys used GPUs for real-time stereo depth extraction from multiple images [YP05]. Their pipeline first rectifies the images using per-pixel projective texture mapping, then computes disparity values between the two images, and, using adaptive aggregation windows and cross checking, chooses the most accurate disparity value. Their implementation was more than four times faster than a comparable CPU-based commercial system. Both Geys *et al.* and Woetzel and Koch addressed a similar problem using a plane sweep algorithm. Geys *et al.* compute depth from pairs of images using a fast plane sweep to generate a crude depth map, then use a min-cut/max-flow algorithm to refine the result [GKV04]; the approach of Woetzel and Koch begins with a plane sweep over images from multiple cameras and pays particular attention to depth discontinuities [WK04].

Image Processing The process of image registration establishes a correlation between two images by means of a (possibly non-rigid) deformation. Before the emergence of programmable hardware, Rezk-Salama *et al.*'s 1999 work [RSHGE99] exploited the trilinear interpolation capabilities of 3D texturing hardware for nonlinear image registration in a medical volume rendering application, with the hardware implementation two orders of magnitude faster than the software implementation. The work of Strzodka *et al.* is one of the earliest to use the programmable floating point capabilities of graphics hardware in this area [SDR03, SDR04]; their image registration implementation is based on the multi-scale gradient flow registration method of Clarenz *et al.* [CDR02] and uses an efficient multi-grid representation of the image multi-scales, a fast multi-grid regularization, and an adaptive time-step control of the iterative solvers. They achieve per-frame computation time of under 2 seconds on pairs of 256×256 images.

Strzodka and Garbe describe a real-time system that computes and visualizes motion on 640×480 25 Hz 2D image sequences using graphics hardware [SG04]. Their system assumes that image brightness only changes due to motion (due to the brightness change constraint equation). Using this assumption, they estimate the motion vectors from calculating the eigenvalues and eigenvectors of the matrix constructed from the averaged partial space and time derivatives of image brightness. In the context of video compression, Kelly and Kokaram describe their gradient-based motion estimator [KK04], implemented on the GPU, that uses a hierarchical Wiener-based algorithm that is robust to noise. Both of these papers demonstrate performance improvements over a CPU implementation.

Computed tomography (CT) methods that reconstruct an object from its projections are computationally intensive and often accelerated by special-purpose hardware. Xu and Mueller implement three 3D reconstruction algorithms (Feldkamp Filtered Backprojection, SART, and EM) on programmable graphics hardware, achieving high-quality floating-point 128^3 reconstructions from 80 projections in time frames from seconds to tens of seconds [XM05].

Erra recently introduced fractal image compression to the GPU with a brute-force Cg implementation that achieved a speedup of over 100:1 over a comparable CPU implementation [Err05].

Signal Processing Motivated by the high arithmetic capabilities of modern GPUs, several projects have developed

GPU implementations of the fast Fourier transform (FFT) [BFH*04b, JvHK04, MA03, MAH02, SL05, SW04, Wlo03]. (The *GPU Gems 2* chapter by Sumanaweera and Liu, in particular, gives a detailed description of the FFT and their GPU implementation [SL05].) In general, these implementations operate on 1D or 2D input data, use a Cooley-Tukey radix-2 decimation-in-time approach (with the exception of Jansen *et al.*'s decimation-in-frequency approach [JvHK04]), and require one fragment-program pass per FFT stage. Govindaraju *et al.* suggest that the Stockham formulation of the FFT is better suited for the GPU by avoiding the need for bit reversal [GLGM06]; in their implementation they focus on efficient cache utilization. The real and imaginary components of the FFT can be computed using two components of the 4-vectors in each fragment processor, so many implementations support processing two FFTs in parallel; other implementations use multiple render targets [GLGM06] or clever packing [Hor06] to fully utilize the 4-wide arithmetic units in the fragment processors on a single FFT.

Daniel Horn's open-source "libgpufft" FFT library [Hor06] is one of the performance leaders in GPU FFTs. Horn's implementation is written in Brook, runs efficiently on both NVIDIA and ATI hardware, and is notable for its full use of 4-wide arithmetic on a single FFT in the fragment program and its identical communication pattern across all stages. Owens *et al.*'s analysis of this library [OSH05] showed that repeated computation of 1D FFTs on the GPU had comparable performance to a highly tuned CPU implementation [FJ98]; the GPU was limited by overhead, memory bandwidth, and the lack of write-back GPU caches, while the CPU was limited by computation. GPUs are correspondingly better than CPUs on 2D FFTs because of poorer CPU cache performance on 2D FFTs [Buc05a].

A GPU implementation of the related discrete cosine transform (DCT), used in JPEG and MPEG compression, was recently presented by Green [Gre05]. The discrete wavelet transform (DWT), used in the JPEG2000 standard, is another useful fundamental signal processing operation; a group from the Chinese University of Hong Kong has developed a GPU implementation of the DWT [WWHL04], which has been integrated into an open-source JPEG2000 codec called "JasPer" [Ada05].

Smirnov and Chuieh compared finite-impulse-response (FIR) filter performance for three FIR components from GNU Radio [SC05], finding that GPU FIR performance was superior to CPU performance for very large numbers of FIR taps, and that replacing CPU stages of a GNU radio receiver with GPU stages also improved overall performance.

Infinite impulse response (IIR) filters, unlike FIR filters, have an impulse response with infinite extent. As a result, each evaluation of an infinite impuse response filter depends on the result from the previous sample. This serial dependence has been refactored into a parallel GPU-compatible formulation in two ways. Simon Green described an implementation of separable 2D IIRs by drawing one row or column of an image, synchronizing between each iteration [Gre05]. More recently, Kass *et al.* presented a more efficient parallel formulation of 2D IIRs on the GPU using cyclic reduction implemented with a parallel-prefix scan formulation [KLO06].

Tone Mapping Tone mapping is the process of mapping pixel intensity values with high dynamic range to the smaller range permitted by a display. Goodnight *et al.* implemented an interactive, time-dependent tone mapping system on GPUs [GWWH03]. In their implementation, they chose the tone-mapping algorithm of Reinhard *et al.* [RSSF02], which is based on the "zone system" of photography, for two reasons. First, the transfer function that performs the tone mapping uses a minimum of global information about the image, making it well-suited to implementation on graphics hardware. Second, Reinhard *et al.*'s algorithm can be adaptively refined, allowing a GPU implementation to trade off efficiency and accuracy. Among the tasks in Goodnight *et al.*'s pipeline was an optimized implementation of a Gaussian convolution. On an ATI Radeon 9800, they were able to achieve highly interactive frame rates with few adaptation zones (limited by mipmap construction) and a few frames per second with many adaptation zones (limited by the performance of the Gaussian convolution).

Audio Gallo and Tsingos characterized GPUs for two audio rendering kernels [GT04], analyzing the performance of variable-delay-line and filtering operations. Their implementation was 20% slower than a CPU implementation, and they identified floating-point texture resampling and long 1D texture support as desirable features for improving performance. Jędrzejewski used ray tracing techniques on GPUs to compute echoes of sound sources in highly occluded environments [Jęd04]. BionicFX has developed commercial "Audio Video Exchange" (AVEX) software that accelerates audio effect calculations using GPUs [Bio06].

Image/Video Processing Frameworks Apple's Core Image and Core Video frameworks allow GPU acceleration of image and video processing tasks [App06a]; the open-source framework Jahshaka uses GPUs to accelerate video compositing [Jah06].

5.4. Global illumination

Perhaps not surprisingly, one of the early areas of GPGPU research was aimed at improving the visual quality of GPU-generated images. Many of the techniques described below accomplish this by simulating an entirely different image generation process from within a fragment program (e.g. a ray tracer). These techniques use the GPU strictly as a computing engine. Other techniques leverage the GPU to perform most of the rendering work, and augment the resulting image with

Figure 14: *Sample images from several global illumination techniques implemented on the GPU. (a) Ray tracing and photon mapping [PDC*03]. (b) Radiosity [CHL04]. (c) Subsurface scattering [CHH03]. (d) Final gather by rasterization [Hac05].*

global effects. Figure 14 shows images from some of the techniques we discuss in this section.

5.4.1. *Ray tracing*

Ray tracing is a rendering technique based on simulating light interactions with surfaces [Whi80]. It is nearly the reverse of the traditional GPU rendering algorithm: the color of each pixel in an image is computed by tracing rays out from the scene camera and discovering which surfaces are intersected by those rays and how light interacts with those surfaces. The ray-surface intersection serves as a core for many global illumination algorithms. Perhaps it is not surprising, then, that ray tracing was one of the earliest GPGPU global illumination techniques to be implemented.

Ray tracing consists of several types of computation: ray generation, ray-surface intersection, and ray-surface shading. Generally, there are too many surfaces in a scene to test every ray against every surface for intersection, so special data structures (called *acceleration structures*) are used to reduce the total number of surfaces rays need to be tested against. Ray-surface shading generally requires generating additional rays to test against the scene (e.g. shadow rays, reflection rays, etc.) The earliest GPGPU ray tracing systems demonstrated that the GPU was capable of not only performing ray-triangle intersections [CHH02], but that the entire ray tracing computation including acceleration structure traversal and shading could be implemented entirely within a set of fragment programs [PBMH02, Pur04]. Section 4.2 enumerates several of the data structures used in this ray tracer.

Nearly all of the major ray tracing acceleration structures have been implemented in some form on the GPU: uniform grids [PBMH02, Pur04], *k*-d trees [FS05], and bounding volume hierarchies [TS05]. Detailed comparisons between GPU implementations of these three acceleration structures can be found in Thrane and Simonsen's masters thesis [TS05]. All of these structures are limited to accelerating ray tracing of static scenes. The efficient implementation of dynamic ray tracing acceleration structures is an active research topic for both CPU and GPU-based ray tracers. Recent work by Carr *et al.* [CHCH06] describes a dynamic bounding volume hierarchy acceleration structure, though the current implementation is limited to accelerating only a single mesh.

Some of the early GPU-based ray tracing work required special drivers, as features like fragment programs and floating point buffers were relatively new and rapidly evolving. There are currently open-source GPU-based ray tracers that run with standard drivers and APIs [Chr05, KL04].

Finally, Weiskopf *et al.* have implemented nonlinear ray tracing on the GPU [WSE04]. Nonlinear ray tracing is a technique that can be used for visualizing gravitational phenomena such as black holes, or light propagation through media with a varying index of refraction (which can produce mirages). Their technique builds upon the linear ray tracing discussed previously, and approximates curved rays with multiple ray segments.

5.4.2. *Photon mapping*

Photon mapping [Jen96] is a two-stage global illumination algorithm. The first stage consists of emitting photons from the light sources in the scene, simulating the photon interactions with surfaces, and finally storing the photons in a data structure for lookup during the second stage. The second stage in the photon mapping algorithm is a rendering stage. Initial surface visibility and direct illumination are computed first, often by ray tracing. Then, the light contributed to each surface point by the environment (indirect) or through focusing by reflection or refraction (caustic) is computed. These computations are done by querying the photon map to get estimates for the amount of energy that arrived from these sources.

Tracing photons is much like ray tracing (Section 5.4.1). Constructing the photon map and indexing the map to find good energy estimates at each image point are much more difficult on the GPU than the CPU. Purcell *et al.* implemented two different techniques for constructing the photon

map and a technique for querying the photon map, all of which run at interactive rates [PDC*03] (Sections 4.1.7 and 4.2 contain some implementation details). Figure 14a shows an image rendered with this system. Larsen and Christensen load-balance photon mapping between the GPU and the CPU and exploit inter-frame coherence to achieve very high frame rates for photon mapping [LC04].

Finally, caustics rendering can be performed interactively on the GPU using image-space techniques. Wyman and Davis [WD06] describe a caustics rendering technique that works by rendering the scene from the light source, and storing the x, y, z location for the nearest diffuse surface. They then evaluate indirect illumination by directly drawing these photons, or by performing an image-space nearest neighbor search. Shah *et al.* [SKP06] describe a caustics rendering technique which works by splatting photons into a caustics map via point rendering. Caustics are then rendered by projecting visible scene points into the light source coordinate system and indexing into the caustic map, much like current shadow mapping techniques.

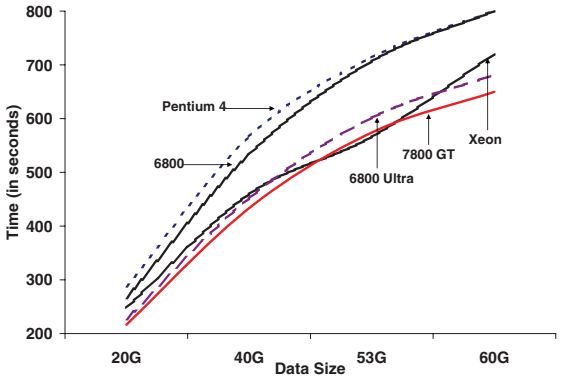

Figure 15: *Performance of GPUTeraSort on a low-end PC with a $250 commodity GPU and an optimized CPU-based algorithm on high-end CPUs costing $1500. GPUTeraSort is able to achieve 3 times better performance-price and also scales better on gigabyte-scale databases. Result from Govindaraju et al. [GGKM06].*

5.4.3. *Radiosity*

At a high level, radiosity works much like photon mapping when computing global illumination for diffuse surfaces. In a radiosity-based algorithm, energy is transferred around the scene much like photons are. Unlike photon mapping, the energy is not stored in a separate data structure that can be queried at a later time. Instead, the geometry in the scene is subdivided into patches or elements, and each patch stores the energy arriving on that patch.

The classical radiosity algorithm [GTGB84] solves for all energy transfer simultaneously. Classical radiosity was implemented on the GPU with an iterative Jacobi solver [CHH03]. The implementation was limited to matrices of around 2000 elements, severely limiting the complexity of the scenes that can be rendered.

An alternate method for solving radiosity equations, known as progressive radiosity, iterates through the energy transfer until the system reaches a steady state [CCWG88]. A GPU implementation of progressive radiosity can render scenes with over one million elements [CHL04, CH05]. Figure 14b shows a sample image created with progressive refinement radiosity on the GPU.

5.4.4. *Subsurface scattering*

Most real-world surfaces do not completely absorb, reflect, or refract incoming light. Instead, incoming light usually penetrates the surface and exits the surface at another location. This subsurface scattering effect is an important component in modeling the appearance of transparent surfaces [HK93]. This subtle yet important effect has also been implemented on the GPU [CHH03]. Figure 14c shows an example of GPU subsurface scattering. The GPU implementation of subsurface scattering uses a three-pass algorithm. First, the amount of light on a given patch in the model is computed. Second, a texture map of the transmitted radiosity is built using precomputed scattering links. Finally, the generated texture is applied to the model. This method for computing subsurface scattering runs in real time on the GPU.

5.4.5. *Hybrid rendering*

Finally, several GPGPU global illumination methods that have been developed do not fit with any of the classically defined rendering techniques. Some methods use traditional GPU rendering in unconventional ways to obtain global illumination effects. Others combine traditional GPU rendering with global illumination effects. We call all of these techniques *hybrid* global illumination techniques.

The Parthenon renderer generates global illumination images by rasterizing the scene multiple times, from different points of view [Hac05]. These scene rasterizations are accumulated to form an estimate of the indirect illumination at each visible point. This indirect illumination estimate is combined with direct illumination computed by traditional GPU rendering techniques. A sample image from the Parthenon renderer is shown in Figure 14d. In a similar fashion, Nijasure computes a sparse sampling of the scene for indirect illumination into cubemaps [Nij03]. The indirect illumination is progressively computed and summed with direct lighting to produce a fully illuminated scene.

Szirmay-Kalos *et al.* demonstrate how to approximate ray tracing on the GPU by localizing environment maps [SKALP05]. They use fragment programs to correct reflection map lookups to more closely match what a ray tracer would compute. Their technique can also be used to generate

multiple refractions or caustics, and runs in real time on the GPU.

Finally, Gautron *et al.* describe a technique for accelerating irradiance caching using the GPU [GKBP05]. They use the GPU to compute irradiance contributions (via splatting) and geometric data, the CPU to traverse the irradiance data, and the GPU to combine direct and indirect illumination for the final image. Their technique shows well over an order of magnitude speedup over a traditional irradiance cache implementation.

5.5. Geometric computing

GPUs have been widely used for performing a number of geometric computations. These geometric computations are used in many applications including motion planning, virtual reality, etc. and include the following.

Constructive Solid Geometry (CSG) operations CSG operations are used for geometric modeling in computer-aided design applications. Basic CSG operations involve Boolean operations such as union, intersection, and difference, and can be implemented efficiently using the depth test and the stencil test [GHF86, GMTF89, GKMV03, RR86, SLJ98].

Distance Fields and Skeletons Distance fields compute the minimum distance of each point to a set of objects and are useful in applications such as path planning and navigation. Distance computation can be performed either using a fragment program or by rendering the distance function of each object in image space [HCK*99, SOM04, SPG03, ST04].

Collision Detection GPU-based collision detection algorithms rasterize the objects and perform either 2D or 2.5-D overlap tests in screen space [BW03, GRLM03, HTG03, HTG04, HCK*99, KP03, VSC01]. Furthermore, visibility computations can be performed using occlusion queries and used to compute both intra- and inter-object collisions among multiple objects [GLM05].

Transparency Computation Transparency computations require the sorting of 3D primitives or their image-space fragments in a back-to-front or a front-to-back order and can be performed using depth peeling [Eve01] or by image-space occlusion queries [GHLM05].

Particle Tracing Particle tracing—and in general generation of vector-field visualizing primitives—has been an active field of research, particularly since the availability of geometry creation and modification features on GPUs. Recent applications make use of either the copy-to-vertex-buffer [KLRS04,KC05], the render-to-vertex-buffer [KSW04, KKKW05] or the vertex-texture-fetch [KW05b, KKW05] functionality to displace primitives.

Compression and LOD Techniques Quite often CPU-based LOD techniques that are able to efficiently handle large amounts of data suffer from the problem that the speedup due to the reduced geometry is often neutralized by the time for the bus transfer of the updated geometry. Therefore, GPU-based LOD [DVS03, BS05, JWLL05] and compression techniques [KSW05] minimize the transfer by expanding the data on the GPU only during rendering.

These algorithms perform computations in image space, and require little or no pre-processing. Therefore, they work well on deformable objects. However, the accuracy of these algorithms is limited to image precision, and can be an issue in some geometric computations such as collision detection. Recently, Govindaraju *et al.* proposed a simple technique to overcome the image-precision error by sufficiently "fattening" the primitives [GLM04]. The technique has been used in performing reliable inter- and intra-object collision computations among general deformable meshes [GKJ*05].

The performance of many geometric algorithms on GPUs is also dependent upon the layout of polygonal meshes; a better layout more effectively utilizes the vertex caches on GPUs. Recently, Yoon *et al.* proposed a novel method for computing cache-oblivious layouts of polygonal meshes and applied it to improve the performance of geometric applications such as view-dependent rendering and collision detection on GPUs [YLPM05]. Their method does not require any knowledge of cache parameters and does not make assumptions on the data access patterns of applications. A user constructs a graph representing an access pattern of an application, and the cache-oblivious algorithm constructs a mesh layout that works well with the cache parameters. The cache-oblivious algorithm was able to achieve 2–20 times improvement on many complex scenarios without any modification to the underlying application or the run-time algorithm.

5.6. Databases and data mining

Database Management Systems (DBMSs) and data mining algorithms are an integral part of a wide variety of commercial applications such as online stock market trading and intrusion detection systems. Many of these applications analyze large volumes of online data and are highly computation- and memory-intensive. As a result, researchers have been actively seeking new techniques and architectures to improve the query execution time. The high memory bandwidth and the parallel processing capabilities of the GPU can significantly accelerate the performance of many essential database queries such as conjunctive selections, aggregations, semilinear queries and join queries. These queries are described in Section 4.5. Govindaraju *et al.* compared the performance of SQL queries on an NVIDIA GeForce 6800 against a 2.8 GHz Intel Xeon processor. Preliminary comparisons indicate up to an order of magnitude improvement for the GPU over a SIMD-optimized CPU implementation [GLW*04].

GPUs are highly optimized for performing rendering operations on geometric primitives and can use these capabilities

to accelerate spatial database operations. Sun *et al.* exploited the color blending capabilities of GPUs for spatial selection and join operations on real world datasets [SAA03]. Bandi *et al.* integrated GPU-based algorithms for improving the performance of spatial database operations into Oracle 9I DBMS [BSAE04].

Recent research has also focused attention on the effective utilization of graphics processors for fast stream mining algorithms. In these algorithms, data is collected continuously and the underlying algorithm performs *continuous* queries on the data stream as opposed to *one-time* queries in traditional systems. Many researchers have advocated the use of GPUs as stream processors for compute-intensive algorithms [BFH*04b, FF88, Man03, Ven03]. Recently, Govindaraju *et al.* have presented fast streaming algorithms using the blending and texture mapping functionalities of GPUs [GRM05]. Data is streamed to and from the GPU in real time, and a speedup of 2–5 times is demonstrated on online frequency and quantile estimation queries over high-end CPU implementations. The high growth rate of GPUs, combined with their substantial processing power, are making the GPU a viable architecture for commercial database and data mining applications.

6. Conclusions: Looking Forward

The field of GPGPU computing is maturing. Early efforts were characterized by a somewhat ad hoc approach and a "GPGPU for its own sake" attitude; the challenge of achieving non-graphics computation on the graphics platform overshadowed analysis of the techniques developed or careful comparison to well optimized, best-in-class CPU analogs. Today researchers seeking to publish GPGPU work typically face a much higher bar, set by careful analyses such as Fatahalian *et al.*'s examination of matrix multiplication [FSH04]. The bar is higher for novelty as well as analysis; new work must go beyond simply "porting" an existing algorithm to the GPU, to demonstrating general principles and techniques or making significantly new and non-obvious use of the hardware. Fortunately, the accumulated body of knowledge on general techniques and building blocks surveyed in Section 4 means that GPGPU researchers need not continually reinvent the wheel. Meanwhile, developers wishing to use GPUs for general-purpose computing have a broad array of applications to learn from and build on. GPGPU algorithms continue to be developed for a wide range of problems, from options pricing to protein folding. On the systems side, several research groups have major ongoing efforts to perform large-scale GPGPU computing by harnessing large clusters of GPU-equipped computers. The emergence of high-level programming languages provided a huge leap forward for GPU developers generally, and languages like BrookGPU [BFH*04b] hold similar promise for non-graphics developers who wish to harness the power of GPUs.

More broadly, GPUs may be seen as the first generation of commodity data-parallel coprocessors. Their tremendous computational capacity and rapid growth curve, far outstripping traditional CPUs, highlight the advantages of domain-specialized data-parallel computing. We can expect increased programmability and generality from future GPU architectures, but not without limit; neither vendors nor users want to sacrifice the specialized performance and architecture that have made GPUs successful in the first place. The next generation of GPU architects face the challenge of striking the right balance between improved generality and ever-increasing performance.

At the same time, other desktop parallel machines are beginning to appear in the mass market. CPU vendors are aggressively pursuing multicore designs, including a heterogeneous example in the Cell processor produced by IBM, Sony, and Toshiba [PAB*05]. The tiled architecture of Cell provides a dense computational fabric well suited to the stream programming model discussed in Section 2.3, similar in many ways to GPUs but oriented toward running fewer threads with more available resources than the very large number of fine-grained, lightweight threads on the GPU.

Perhaps the most important challenge is to find the right high-level programming model for aggressively multithreaded parallel computation. Heterogenous systems such as the Sony Playstation 3, which joins a Cell processor and a modern GPU with a high-bandwidth 20 GB/s bus, present even more interesting opportunities and challenges. Current graphics APIs, and GPGPU languages layered on top of graphics APIs, hide much of the complexity of parallel execution (such as scheduling, synchronization, locks, and so on) with an "implicitly parallel" programming model. For example, pixel computations must be completely independent (e.g., no communication with neighboring pixels) and do not include scatter operations that would enable multiple fragment processors to write to the same pixel. GPUs are growing more general, low-level GPGPU programming is being supplanted by high-level languages and toolkits, and new general-purpose data-parallel contenders such as the Cell chip have emerged. Can the GPGPU research community build on its experience with successful, high-level parallel programming models to transcend its computer graphics roots and develop the computational idioms, techniques, and frameworks for the desktop parallel computing environment of the future?

Acknowledgments

*This article is an updated and extended version of the authors' 2005 State of the Art Report, presented at Eurographics 2005 [OLG*05].*

Thanks to David Blythe, Ian Buck, Jeff Bolz, Dominik Göddeke, Daniel Horn, Marc Pollefeys, and Robert Strzodka

for their thoughtful comments, and to the anonymous reviewers for their helpful and constructive criticism.

References

[Ada05] ADAMS M.: JasPer project. Available at http://www.ece.uvic.ca/~mdadams/jasper/, 2005.

[AJR*01] AN P., JULA A., RUS S., SAUNDERS S., SMITH T., TANASE G., THOMAS N., AMATO N., RAUCHWERGER L.: STAPL: An adaptive, generic parallel C++ library. In *Workshop on Languages and Compilers for Parallel Computing*, pp. 193–208, August 2001.

[App06a] Apple Computer Core Image. Available at http://www.apple.com/macosx/tiger/coreimage.html, 2006.

[App06b] Apple Computer OpenGL shader builder/profiler. Available at http://developer.apple.com/graphicsimaging/opengl/, 2006.

[Bat68] BATCHER K. E.: Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computing Conference*, vol. 32, pp. 307–314, April 1968.

[Bax06] BAXTER B.: The image debugger. Available at http://www.billbaxter.com/projects/imdebug/, 2006.

[BDHK06] BUSTOS B., DEUSSEN O., HILLER S., KEIM D.: A graphics hardware accelerated algorithm for nearest neighbor search. In *Proceedings of the 6th International Conference on Computational Science, vol. 3994 of Lecture Notes in Computer Science*. Springer, pp. 196–199, May 2006.

[Ben75] BENTLEY J. L.: Multidimensional binary search trees used for associative searching. *Commun. ACM* 18(9):509–517, September 1975.

[BFGS03] BOLZ J., FARMER I., GRINSPUN E., SCHRÖDER P.: Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. *ACM Trans. Graph. (Proceedings of ACM SIGGRAPH)* 22(3). 917–924, July 2003.

[BFH04a] BUCK I., FATAHALIAN K., HANRAHAN P.: GPUBench: Evaluating GPU performance for numerical and scientific applications. In *2004 ACM Workshop on General-Purpose Computing on Graphics Processors*, p. C–20, August 2004.

[BFH*04b] BUCK I., FOLEY T., HORN D., SUGERMAN J., FATAHALIAN K., HOUSTON M., HANRAHAN P.: Brook for GPUs: Stream computing on graphics hardware. *ACM Trans. Graph.* 23(3):777–786, August 2004.

[Bio06] Bionic FX.: Available at http://www.bionicfx.com/, 2006.

[Ble90] BLELLOCH G.: *Vector Models for Data-Parallel Computing*. MIT Press, 1990.

[Bly06] BLYTHE D.: The Direct3D 10 system. *ACM Trans. Graph.* 25(3):724–734, August 2006.

[Boh98] BOHN C. A.: Kohonen feature mapping through graphics hardware. In *Proceedings of the Joint Conference on Information Sciences*, vol. II, pp. 64–67, 1998.

[Bon06] BOND A.: Havok FX: GPU-accelerated physics for PC games. In *Proceedings of Game Developers Conference 2006*. Available at http://www.havok.com/content/view/187/77/, March 2006.

[BP03] BLEIWEISS A., PREETHAM A.: Ashli—Advanced shading language interface. *ACM SIGGRAPH Course Notes*. Available at http://www.ati.com/developer/SIGGRAPH03/AshliNotes.pdf, July 2003.

[BP04] BUCK I., PURCELL T.: A toolkit for computation on GPUs. In *GPU Gems*, Fernando R., editor. Addison Wesley, pp. 621–636, March 2004.

[BS05] BOUBEKEUR T., SCHLICK C.: Generic mesh refinement on GPU. In *Graphics Hardware 2005*, pp. 99–104, July 2005.

[BSAE04] BANDI N., SUN C., AGRAWAL D., EL ABBADI A.: Hardware acceleration in commercial databases: A case study of spatial operations. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases*, pp. 1021–1032, September 2004.

[Buc05a] BUCK I.: GPGPU: General-purpose computation on graphics hardware—high level languages for GPUs. *ACM SIGGRAPH Course Notes*, August 2005.

[Buc05b] BUCK I.: Taking the plunge into GPU computing. In *GPU Gems 2*, M. Pharr, editor. Addison Wesley, chapter 32, pp. 509–519, March 2005.

[BW03] BACIU G., WONG W. S. K.: Image-based techniques in a hybrid collision detector. *IEEE Trans. Vis. Comput. Graph.* 9(2): 254–271, April 2003.

[CCWG88] COHEN M. F., CHEN S. E., WALLACE J. R., GREENBERG D. P.: A progressive refinement approach to fast radiosity image generation. In *Comput. Graph. (Proceedings of SIGGRAPH)*, vol. 22, pp. 75–84, August 1988.

[CDR02] CLARENZ U., DROSKE M., RUMPF M.: Towards fast non-rigid registration. In *Inverse Problems, Image Analysis and Medical Imaging, AMS Special Session Interaction of Inverse Problems and Image Analysis*, vol. 313, AMS, pp. 67–84, 2002.

[CH05] COOMBE G., HARRIS M.: Global illumination using progressive refinement radiosity. In *GPU Gems 2*,

Pharr M., editor. Addison Wesley, chapter 39, pp. 635–647, March 2005.

[CHCH06] CARR N. A., HOBEROCK J., CRANE K., HART J. C.: Fast GPU ray tracing of dynamic meshes using geometry images. In *Proceedings of the 2006 Conference on Graphics Interface*, pp. 203–209, June 2006.

[CHH02] CARR N. A., HALL J. D., HART J. C.: The ray engine. In *Graphics Hardware 2002*, pp. 37–46, September 2002.

[CHH03] CARR N. A., HALL J. D., HART J. C.: GPU algorithms for radiosity and subsurface scattering. In *Graphics Hardware*, pp. 51–59, July 2003.

[CHL04] COOMBE G., HARRIS M. J., LASTRA A.: Radiosity on graphics hardware. In *Proceedings of the 2004 Conference on Graphics Interface*, pp. 161–168, May 2004.

[Chr05] CHRISTEN M.: *Ray Tracing on GPU*. Master's thesis, University of Applied Sciences Basel, 2005.

[CND03] CALLELE D., NEUFELD E., DE LATHOUWER K.: Sorting on a GPU. Available at http://www.cs.usask.ca/faculty/callele/gpusort/gpusort.html, 2003.

[DNB*05] DUCA N., NISKI K., BILODEAU J., BOLITHO M., CHEN Y., COHEN J.: A relational debugging engine for the graphics pipeline. *ACM Trans. Graph.* 24(3):453–463, August 2005.

[DPRS89] DOWD M., PERL Y., RUDOLPH L., SAKS M.: The periodic balanced sorting network. *J. ACM* 36(4):738–757, October 1989.

[DVS03] DACHSBACHER C., VOGELGSANG C., STAMMINGER M.: Sequential point trees. *ACM Trans. Graph.* 22(3):657–662, July 2003.

[EMP*97] EYLES J., MOLNAR S., POULTON J., GREER T., LASTRA A., ENGLAND N., WESTOVER L.: PixelFlow: The realization. In *1997 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pp. 57–68, August 1997.

[Eng78] ENGLAND J. N.: A system for interactive modeling of physical curved surface objects. In *Comput. Graph. (Proceedings of SIGGRAPH 1978)*, vol. 12, pp. 336–340, August 1978.

[Err05] ERRA U.: Toward real time fractal image compression using graphics hardware. In *International Symposium on Visual Computing 2005, vol. 3804 of Lecture Notes in Computer Science*, Springer-Verlag, pp. 723–728, December 2005.

[Eve01] EVERITT C.: *Interactive Order-Independent Transparency*. Tech. rep., NVIDIA Corporation. Available at http://developer.nvidia.com/object/Interactive_Order_Transparency.html, May 2001.

[EVG04] ERNST M., VOGELGSANG C., GREINER G.: Stack implementation on programmable graphics hardware. In *Proceedings of Vision, Modeling, and Visualization*, pp. 255–262, November 2004.

[EWN05] EKMAN M., WARG F., NILSSON J.: An in-depth look at computer performance growth. *ACM SIGARCH Comput. Archit. News* 33(1):144–147, March 2005.

[FF88] FOURNIER A., FUSSELL D.: On the power of the frame buffer. *ACM Trans. Graph.* 7(2):103–128, 1988.

[FJ98] FRIGO M., JOHNSON S. G.: FFTW: An adaptive software architecture for the FFT. In *Proceedings of the 1998 International Conference on Acoustics, Speech, and Signal Processing*, vol. 3, pp. 1381–1384, May 1998.

[FM04] FUNG J., MANN S.: Computer vision signal processing on graphics processing units. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 5, pp. 93–96, May 2004.

[FPE*89] FUCHS H., POULTON J., EYLES J., GREER T., GOLDFEATHER J., ELLSWORTH D., MOLNAR S., TURK G., TEBBS B., ISRAEL L.: Pixel-Planes 5: A heterogeneous multiprocessor graphics system using processor-enhanced memories. In *Comput. Graph. (Proceedings of SIGGRAPH)*, vol. 23, pp. 79–88, July 1989.

[FS05] FOLEY T., SUGERMAN J.: KD-Tree acceleration structures for a GPU raytracer. In *Graphics Hardware 2005*, pp. 15–22, July 2005.

[FSH04] FATAHALIAN K., SUGERMAN J., HANRAHAN P.: Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. In *Graphics Hardware 2004*, pp. 133–138, August 2004.

[FTM02] FUNG J., TANG F., MANN S.: Mediated reality using computer graphics hardware for computer vision. In *6th International Symposium on Wearable Computing*, pp. 83–89, October 2002.

[GDNV05] GRIESSER A., DE ROECK S., NEUBECK A., VAN GOOL L.: GPU-based foreground-background segmentation using an extended colinearity criterion. In *Proceedings of Vision, Modeling, and Visualization*, pp. 319–326, November 2005.

[GGHM05] GALOPPO N., GOVINDARAJU N. K., HENSON M., MANOCHA D.: LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, p. 3, November 2005.

[GGK06] GREß A., GUTHE M., KLEIN R.: GPU-based collision detection for deformable parameterized surfaces. *Comput. Graph. Forum.* 25(3):497–506, September 2006.

[GGKM06] GOVINDARAJU N. K., GRAY J., KUMAR R., MANOCHA D.: GPUTeraSort: High performance graphics coprocessor sorting for large database management. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pp. 325–336, June 2006.

[GH06] GREEN S., HARRIS M.: Game physics simulation on NVIDIA GPUs. In *Proceedings of Game Developers Conference 2006*. Available at http://www.havok.com/content/view/187/77/, March 2006.

[GHF86] GOLDFEATHER J., HULTQUIST J. P. M., FUCHS H.: Fast constructive-solid geometry display in the Pixel-Powers graphics system. In *Comput. Graph. (Proceedings of SIGGRAPH 86)*, vol. 20, pp. 107–116, August 1986.

[GHLM05] GOVINDARAJU N. K., HENSON M., LIN M. C., MANOCHA D.: Interactive visibility ordering of geometric primitives in complex environments. In *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*, pp. 49–56, April 2005.

[GKBP05] GAUTRON P. K., KŘIVÁNEK J., BOUATOUCH K., PATTANAIK S. N.: Radiance cache splatting: A GPU-friendly global illumination algorithm. In *Eurographics Symposium on Rendering*, pp. 55–64, June 2005.

[GKJ*05] GOVINDARAJU N. K., KNOTT D., JAIN N., KABUL I., TAMSTORF R., GAYLE R., LIN M. C., MANOCHA D.: Interactive collision detection between deformable models using chromatic decomposition. *ACM Trans. Graph.* 24(3):991–999, August 2005.

[GKMV03] GUHA S., KRISHNAN S., MUNAGALA K., VENKATASUBRAMANIAN S.: Application of the two-sided depth test to CSG rendering. In *2003 ACM Symposium on Interactive 3D Graphics*, pp. 177–180, April 2003.

[GKV04] GEYS I., KONINCKX T. P., VAN GOOL L.: Fast interpolated cameras by combining a GPU based plane sweep with a max-flow regularisation algorithm. In *Proceedings of the 2nd International Symposium on 3D Data Processing, Visualization and Transmission*, pp. 534–541, September 2004.

[GLGM06] GOVINDARAJU N. K., LARSEN S., GRAY J., MANOCHA D.: A memory model for scientific algorithms on graphics processors. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. p. 89, November 2006.

[GLM04] GOVINDARAJU N. K., LIN M. C., MANOCHA D.: Fast and reliable collision culling using graphics hardware. In *Proceedings of ACM Virtual Reality and Software Technology*, pp. 2–9, November 2004.

[GLM05] GOVINDARAJU N. K., LIN M. C., MANOCHA D.: Quick-CULLIDE: Efficient inter- and intra-object collision culling using graphics hardware. In *Proceedings of IEEE Virtual Reality*, pp. 59–66, March 2005.

[GLW*04] GOVINDARAJU N. K., LLOYD B., WANG W., LIN M., MANOCHA D.: Fast computation of database operations using graphics processors. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pp. 215–226, June 2004.

[GM05] GOVINDARAJU N. K., MANOCHA D.: Efficient relational database management using graphics processors. In *ACM SIGMOD Workshop on Data Management on New Hardware*, pp. 29–34, June 2005.

[GMTF89] GOLDFEATHER J., MOLNAR S., TURK G., FUCHS H.: Near real-time CSG rendering using tree normalization and geometric pruning. *IEEE Comput. Graph. Appl.* 9(3):20–28, May 1989.

[GPU06] GPUSort: A high performance GPU sorting library. Available at http://gamma.cs.unc.edu/GPUSORT/, 2006.

[Gra06] Graphic Remedy gDEBugger. Available at http://www.gremedy.com/, 2006.

[Gre03] GREEN S.: NVIDIA cloth sample. Available at http://download.developer.nvidia.com/developer/SDK/Individual_Samples/samples.html#glsl_physics, 2003.

[Gre04] GREEN S.: NVIDIA particle system sample. Available at http://download.developer.nvidia.com/developer/SDK/Individual_Samples/samples.html#gpu_particles, 2004.

[Gre05] GREEN S.: Image processing tricks in OpenGL. In *Proceedings of Game Developers Conference 2005*. Available at http://download.nvidia.com/developer/presentations/2005/GDC/OpenGL_Day/OpenGL_Image_Processing_Tricks.pdf, March 2005.

[GRH*05] GOVINDARAJU N. K., RAGHUVANSHI N., HENSON M., TUFT D., MANOCHA D.: *A Cache-Efficient Sorting Algorithm for Database and Data Mining Computations using Graphics Processors*. Tech. Rep. TR05-016, University of North Carolina, 2005.

[Gri05] GRIESSER A.: *Real-Time, GPU-based Foreground-Background Segmentation*. Tech. Rep. BIWI-TR-269, Computer Vision Lab, ETH Zürich, August 2005.

[GRLM03] GOVINDARAJU N. K., REDON S., LIN M. C., MANOCHA D., CULLIDE: Interactive collision detection between complex models in large environments using graphics hardware. In *Graphics Hardware*, pp. 25–32, July 2003.

[GRM05] GOVINDARAJU N. K., RAGHUVANSHI N., MANOCHA D.: Fast and approximate stream mining of quantiles and frequencies using graphics processors. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pp. 611–622, June 2005.

[GST07] GÖDDEKE D., STRZODKA R., TUREK S.: Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations. *Int. J. Parallel, Emerg. Distrib. Syst*. To appear, 2007.

[GT04] GALLO E., TSINGOS N.: Efficient 3D audio processing with the GPU. In *2004 ACM Workshop on General-Purpose Computing on Graphics Processors*, p. C–42, August 2005.

[GTGB84] GORAL C. M., TORRANCE K. E., GREENBERG D. P., BATTAILE B.: Modelling the interaction of light between diffuse surfaces. In *Comput. Graph. (Proceedings of SIGGRAPH 1984)*, vol. 18, pp. 213–222, July 1984.

[GWL*03] GOODNIGHT N., WOOLLEY C., LEWIN G., LUEBKE D., HUMPHREYS G.: A multigrid solver for boundary value problems using programmable graphics hardware. In *Graphics Hardware 2003*, pp. 102–111, July 2003.

[GWWH03] GOODNIGHT N., WANG R., WOOLLEY C., HUMPHREYS G.: Interactive time-dependent tone mapping using programmable graphics hardware. In *Proceedings of the 14th Eurographics Workshop on Rendering*, pp. 26–37, June 2003.

[GZ06] GREß A., ZACHMANN G.: GPU-ABiSort: Optimal parallel sorting on stream architectures. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium*, April 2006.

[Hac05] HACHISUKA T.: High-quality global illumination rendering using rasterization. In *GPU Gems 2*, PHARR M., (Ed.). Addison Wesley, chapter 38, pp. 615–633, March 2005.

[Har02] HARRIS M. J.: *Analysis of Error in a CML Diffusion Operation*. Tech. Rep. TR02-015, University of North Carolina, 2002.

[Har04] HARRIS M.: Fast fluid dynamics simulation on the GPU. In *GPU Gems*, R. Fernando, editor. Addison Wesley, pp. 637–665, March 2004.

[Har05a] HARRIS M.: Mapping computational concepts to GPUs. In *GPU Gems 2*, M. Pharr, editor. Addison Wesley, chapter 31, pp. 493–508, March 2005.

[Har05b] HARRIS M.: NVIDIA fluid code sample. Available at http://download.developer.nvidia.com/developer/SDK/Individual_Samples/samples.html#gpgpu_fluid, 2005.

[HB05] HARRIS M., BUCK I.: GPU flow control idioms. In *GPU Gems 2*, M. Pharr, editor. Addison Wesley, chapter 34, pp. 547–555, , March 2005.

[HBSL03] HARRIS M. J., BAXTER III W., SCHEUERMANN T., LASTRA A.: Simulation of cloud dynamics on graphics hardware. In *Graphics Hardware 2003*, pp. 92–101, July 2003.

[HCK*99] HOFF III K., CULVER T., KEYSER J., LIN M., MANOCHA D.: Fast computation of generalized Voronoi diagrams using graphics hardware. In *Proceedings of SIGGRAPH 1999, Computer Graphics Proceedings, Annual Conference Series*, pp. 277–286, August 1999.

[HCSL02] HARRIS M. J., COOMBE G., SCHEUERMANN T., LASTRA A.: Physically-based visual simulation on graphics hardware. In *Graphics Hardware 2002*, pp. 109–118, September 2002.

[HE99a] HOPF M., ERTL T.: Accelerating 3D convolution using graphics hardware. In *IEEE Visual. 1999*, pp. 471–474, October 1999.

[HE99b] HOPF M., ERTL T.: Hardware based wavelet transformations. In *Proceedings of Vision, Modeling, and Visualization*, pp. 317–328, November 1999.

[HHN*02] HUMPHREYS G., HOUSTON M., NG R., FRANK R., AHERN S., KIRCHNER P., KLOSOWSKI J.: Chromium: A stream-processing framework for interactive rendering on clusters. *ACM Trans. Graph*. 21(3):693–702, July 2002.

[HJ03] HARRIS M. J., JAMES G.: Simulation and animation using hardware accelerated procedural textures. In *Proceedings of Game Developers Conference 2003*, March 2003.

[HK93] HANRAHAN P., KRUEGER W.: Reflection from layered surfaces due to subsurface scattering. In *Proceedings of SIGGRAPH 1993, Computer Graphics Proceedings, Annual Conference Series*, pp. 165–174, August 1993.

[HLN06] HAGEN T. R., LIE K.-A., NATVIG J. R.: Solving the Euler equations on graphics processing units. In *Proceedings of the 6th International Conference on Computational Science, vol. 3994 of Lecture Notes in Computer Science*. Springer, pp. 220–227, May 2006.

[HMG03] HILLESLAND K. E., MOLINOV S., GRZESZCZUK R.: Nonlinear optimization framework for image-based modeling on programmable graphics hardware. *ACM Trans. Graph.* 22(3):925–934, July 2003.

[Hor05] HORN D.: Stream reduction operations for GPGPU applications. In *GPU Gems 2*, M. Pharr, editor. Addison Wesley, chapter 36, pp. 573–589, March 2005.

[Hor06] HORN D.: libgpufft. Available at http://sourceforge. net/projects/gpufft/, 2006.

[HS86] HILLIS W. D., STEELE JR. G. L.: Data parallel algorithms. *Comm. ACM* 29(12):1170–1183, December 1986.

[HSC*05] HENSLEY J., SCHEUERMANN T., COOMBE G., SINGH M., LASTRA A.: Fast summed-area table generation and its applications. *Comput. Graph. Forum* 24(3):547–555, September 2005.

[HTG03] HEIDELBERGER B., TESCHNER M., GROSS M.: Real-time volumetric intersections of deforming objects. In *Proceedings of Vision, Modeling, and Visualization*, pp. 461–468, November 2003.

[HTG04] HEIDELBERGER B., TESCHNER M., GROSS M.: Detection of collisions and self-collisions using image-space techniques. *J. of WSCG* 12(3):145–152, February 2004.

[HWSE99] HEIDRICH W., WESTERMANN R., SEIDEL H.-P., ERTL T.: Applications of pixel textures in visualization and realistic image synthesis. In *1999 ACM Symposium on Interactive 3D Graphics*, pp. 127–134, April 1999.

[HZLM01] HOFF K. E. III, ZAFERAKIS A., LIN M. C., MANOCHA D.: Fast and simple 2D geometric proximity queries using graphics hardware. In *2001 ACM Symposium on Interactive 3D Graphics*, pp. 145–148, March 2001.

[Ins03] The Insight Toolkit. Available at http://www.itk. org/, 2003.

[Int06] Intel processors product list. Available at http://www.intel.com/products/processor, 2006.

[Jah06] JAHSHAKA: Jahshaka image processing toolkit. Available at http://www.jahshaka.org/, 2006.

[Jam01a] JAMES G.: NVIDIA game of life sample. Available at http://download.developer.nvidia.com/developer/SDK/Individual_Samples/samples.html#GL_GameOfLife, 2001.

[Jam01b] JAMES G.: NVIDIA water surface simulation sample. Available at http://download.developer. nvidia. com/developer/SDK/Individual_Samples/samples.html#WaterInteraction, 2001.

[Jam01c] JAMES G.: Operations for hardware-accelerated procedural texture animation. In *Game Programming Gems 2*, M. Deloura, editor. Charles River Media, pp. 497–509, 2001.

[Jęd04] JĘDRZEJEWSKI M.: *Computation of Room Acoustics on Programmable Video Hardware*. Master's thesis, Polish-Japanese Institute of Information Technology, Warsaw, Poland, 2004.

[JEH01] JOBARD B., ERLEBACHER G., HUSSAINI M. Y.: Lagrangian-Eulerian advection for unsteady flow visualization. In *IEEE Visual.*, pp. 53–60, October 2001.

[Jen96] JENSEN H. W.: Global illumination using photon maps. In *Rendering Techniques '96 (Proceedings of the 6th Eurographics Rendering Workshop)*, pp. 21–30, June 1996.

[JS05] JIANG C., SNIR M.: Automatic tuning matrix multiplication performance on graphics hardware. In *Proceedings of the Fourteenth International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pp. 185–196, September 2005.

[JVRHK04] JANSEN T., VON RYMON-LIPINSKI B., HANSSEN N., KEEVE E.: Fourier volume rendering on the GPU using a Split-Stream-FFT. In *Proceedings of Vision, Modeling, and Visualization*, pp. 395–403, November 2004.

[JWLL05] JI J., WU E., LI S., LIU X.: Dynamic LOD on GPU. In *Proceedings of Computer Graphics International 2005*, pp. 108–114, June 2005.

[KBR04] KESSENICH J., BALDWIN D., ROST R.: The OpenGL Shading Language version 1.10.59. Available at http://www.opengl.org/documentation/oglsl.html, April 2004.

[KBW06] KRÜGER J., BÜRGER K., WESTERMANN R.: Interactive screen-space accurate photon tracing on GPUs. In *Eurographics Symposium on Rendering*, pp. 319–329, June 2006.

[KC05] KOLB A., CUNTZ N.: Dynamic particle coupling for GPU-based fluid simulation. In *Proceedings of the 18th Symposium on Simulation Technique*, pp. 722–727, September 2005.

[KF05] KILGARIFF E., FERNANDO R.: The GeForce 6 series GPU architecture. In *GPU Gems 2*, M. Pharr, editor. Addison Wesley, chapter 30, pp. 471–491, March 2005.

[KI99] KEDEM G., ISHIHARA Y.: Brute force attack on UNIX passwords with SIMD computer. In *Proceedings of the 8th USENIX Security Symposium*, pp. 93–98, August 1999.

[KK04] KELLY F., KOKARAM A.: Graphics hardware for gradient based motion estimation. In *Embedded Processors for Multimedia and Communications, vol. 5309 of Proceedings of the SPIE*, pp. 92–103, April 2004.

[KKKW05] KRÜGER J., KIPFER P., KONDRATIEVA P., WESTERMANN R.: A particle system for interactive visualization of 3D flows. *IEEE Trans. on Visual. Comput. Graph.* 11(6):744–756, November/December 2005.

[KKW05] KONDRATIEVA P., KRÜGER J., WESTERMANN R.: The application of GPU particle tracing to diffusion tensor field visualization. In *IEEE Visual.*, pp. 73–78, October 2005.

[KL03] KIM T., LIN M. C.: Visual simulation of ice crystal growth. In *2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pp. 86–97, August 2003.

[KL04] KARLSSON F., LJUNGSTEDT C. J.: *Ray tracing fully implemented on programmable graphics hardware*. Master's thesis, Chalmers University of Technology, 2004.

[KLO06] KASS M., LEFOHN A., OWENS J.: *Interactive Depth of Field*. Tech. Rep. #06-01, Pixar. Available at http://graphics.pixar.com/DepthOfField/, 2006.

[KLR04] KOLB A., LATTA L., REZK-SALAMA C.: Hardware-based simulation and collision detection for large particle systems. In *Graphics Hardware*, pp. 123–132, August 2004.

[KP03] KNOTT D., PAI D. K.: CInDeR: Collision and interference detection in real-time using graphics hardware. In *Proceedings of the 2003 Conference on Graphics Interface*, pp. 73–80, June 2003.

[KSW04] KIPFER P., SEGAL M., WESTERMANN R.: UberFlow: A GPU-based particle engine. In *Graphics Hardware 2004*, pp. 115–122, August 2004.

[KSW05] KRÜGER J., SCHNEIDER J., WESTERMANN R.: DuoDecim—a structure for point scan compression and rendering. In *Proceedings of the Symposium on Point-Based Graphics 2005*, pp. 99–108, June 2005.

[KW03] KRÜGER J., WESTERMANN R.: Linear algebra operators for GPU implementation of numerical algorithms. *ACM Trans. Graph.* 22(3):908–916, July 2003.

[KW05a] KIPFER P., WESTERMANN R.: Improved GPU sorting. In *GPU Gems 2*, M. Pharr, editor. Addison Wesley, chapter 46, pp. 733–746, March 2005.

[KW05b] KRÜGER J., WESTERMANN R.: GPU simulation and rendering of volumetric effects for computer games and virtual environments. *Comput. Graph. Forum* 24(3):685–693, September 2005.

[LC04] LARSEN B. D., CHRISTENSEN N. J.: Simulating photon mapping for real-time applications. In *Eurographics Symposium on Rendering*, pp. 123–132, June 2004.

[LDN04] LEFEBVRE S., DARBON J., NEYRET F.: *Unified Texture Management for Arbitrary Meshes*. Tech. Rep. 5210, INRIA, May 2004.

[Lef03] LEFOHN A. E.: *A Streaming Narrow-Band Algorithm: Interactive Computation and Visualization of Level-Set Surfaces*. Master's thesis, University of Utah, December 2003.

[LFW06] LUCAS P., FRITZ N., WILHELM R.: The CGiS compiler. In *Proceedings of the 15th International Conference on Compiler Construction, vol. 3923 of Lecture Notes in Computer Science*. Springer, pp. 105–108, March 2006.

[LFWK05] LI W., FAN Z., WEI X., KAUFMAN A.: GPU-based flow simulation with complex boundaries. In *GPU Gems 2*, M. Pharr, editor. Addison Wesley, chapter 47, pp. 747–764, March 2005.

[LHN05] LEFEBVRE S., HORNUS S., NEYRET F.: Octree textures on the GPU. In *GPU Gems 2*, M. Pharr, editor. Addison Wesley, chapter 37, pp. 595–613, March 2005.

[LHPL87] LEVINTHAL A., HANRAHAN P., PAQUETTE M., LAWSON J.: Parallel computers for graphics applications. *ACM SIGOPS Oper. Syst. Rev.* 21(4):193–198, October 1987.

[LKHW03] LEFOHN A. E., KNISS J. M., HANSEN C. D., WHITAKER R. T.: Interactive deformation and visualization of level set surfaces using graphics hardware. In *IEEE Visual.*, pp. 75–82, October 2003.

[LKHW04] LEFOHN A. E., KNISS J. M., HANSEN C. D., WHITAKER R. T.: A streaming narrow-band algorithm: Interactive computation and visualization of levelset surfaces. *IEEE Trans. Visual. Comput. Graph.* 10(4):422–433, July/August 2004.

[LKM01] LINDHOLM E., KILGARD M. J., MORETON H.: A user-programmable vertex engine. In *Proceedings of ACM SIGGRAPH 2001, Computer Graphics Proceedings, Annual Conference Series*, pp. 149–158, August 2001.

[LKO05] LEFOHN A., KNISS J., OWENS J.: Implementing efficient parallel data structures on GPUs. In *GPU Gems 2*, M. Pharr, editor. Addison Wesley, chapter 33, pp. 521–545, March 2005.

[LKS*06] LEFOHN A. E., KNISS J., STRZODKA R., SENGUPTA S., OWENS J. D.: Glift: An abstraction for generic, efficient GPU data structures. *ACM Trans. on Graphics* 26(1):60–99, January 2006.

[LLW04] LIU Y., LIU X., WU E.: Real-time 3D fluid simulation on GPU with complex obstacles. In *Proceedings of Pacific Graphics 2004*, pp. 247–256, October 2004.

[LM01] LARSEN E. S., MCALLISTER D.: Fast matrix multiplies using graphics hardware. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, p. 55, November 2001.

[LP84] LEVINTHAL A., PORTER T.: Chap – a SIMD graphics processor. In *Computer Graphics (Proceedings of SIGGRAPH 84)*, vol. 18, pp. 77–82, July 1984.

[LRDG90] LENGYEL J., REICHERT M., DONALD B. R., GREENBERG D. P.: Real-time robot motion planning using rasterizing computer graphics hardware. In *Computer Graphics (Proceedings of ACM SIGGRAPH 1990)*, vol. 24, pp. 327–335, August 1990.

[LSK*05] LEFOHN A., SENGUPTA S., KNISS J., STRZODKA R., OWENS J. D.: Dynamic adaptive shadow maps on graphics hardware. In *ACM SIGGRAPH 2005 Conference Abstracts and Applications*, August 2005.

[LW02] LEFOHN A. E., WHITAKER R. T.: *A GPU-Based, Three-Dimensional Level Set Solver with Curvature Flow*. Tech. Rep. UUCS-02-017, University of Utah, 2002.

[LWK03] LI W., WEI X., KAUFMAN A.: Implementing lattice Boltzmann computation on graphics hardware. In *The Visual Computer*, vol. 19, pp. 444–456, December 2003.

[MA03] MORELAND K., ANGEL E.: The FFT on a GPU. In *Graphics Hardware*, pp. 112–119. Available at http://www.cs.unm.edu/∼kmorel/documents/fftgpu/, July 2003.

[MAH02] MITCHELL J. L., ANSARI M. Y., HART E.: Advanced image processing with DirectX® 9 pixel shaders. In *ShaderX2: Shader Programming Tips and Tricks with DirectX 9.0*,W. F. Engel, editor. Wordware Publishing, pp. 457–464, 2002.

[Man03] MANOCHA D.: Interactive geometric and scientific computations using graphics hardware. *ACM SIGGRAPH Course Notes*, July 2003.

[MDP*04] MCCOOL M., DU TOIT S., POPA T., CHAN B., MOULE K.: Shader algebra. *ACM Trans. on Graphics* 23(3):787–795, August 2004.

[MGAK03] MARK W. R., GLANVILLE R. S., AKELEY K., KILGARD M. J.: Cg: A system for programming graphics hardware in a C-like language. *ACM Trans. on Graphics* 22(3):896–907, July 2003.

[MIA*04] MCCORMICK P. S., INMAN J., AHRENS J. P., HANSEN C., ROTH G.: Scout: A hardware-accelerated system for quantitatively driven visualization and analysis. In *IEEE Visual.*, pp. 171–178, October 2004.

[Mic05a] Microsoft high-level shading language. Available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/directx/graphics/reference/hlslreference/hlslreference.asp, 2005.

[Mic05b] Microsoft shader debugger. Available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/directx/graphics/Tools/ShaderDebugger.asp, 2005.

[MM05] MONTRYM J., MORETON H.: The GeForce 6800. *IEEE Micro* 25(2):41–51, March/April 2005.

[Mor02] MORAVÁNSZKY A.: Dense matrix algebra on the GPU. In *ShaderX2: Shader Programming Tips and Tricks with DirectX 9.0*,W. F. Engel, editor. Wordware Publishing, pp. 352–380, 2002.

[NHP04] NYLAND L., HARRIS M., PRINS J.: N-body simulations on a GPU. In *2004 ACM Workshop on General-Purpose Computing on Graphics Processors*, p. C–37, August 2004.

[Nij03] NIJASURE M.: *Interactive Global Illumination on the Graphics Processing Unit*. Master's thesis, University of Central Florida, 2003.

[OL98] OLANO M., LASTRA A.: A shading language on graphics hardware: The PixelFlow shading system. In *Proceedings of SIGGRAPH 1998*, Computer Graphics Proceedings, Annual Conference Series, pp. 159–168, July 1998.

[OLG*05] OWENS J. D., LUEBKE D., GOVINDARAJU N., HARRIS M., KRÜGER J., LEFOHN A. E., PURCELL T.: A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, pp. 21–51, September 2005.

[Ope03] OpenGL Architecture Review Board. ARB fragment program. Revision 26. Available at http://oss.sgi.com/projects/ogl-sample/registry/ARB/fragment_program.txt, 22 August 2003.

[Ope04] OpenGL Architecture Review Board. ARB vertex program. Revision 45. Available at http://oss.sgi.com/projects/ogl-sample/registry/ARB/vertex_program.txt, 27 September 2004.

[Ope06] OpenVIDIA. GPU-accelerated computer vision library. Available at http://openvidia.sourceforge.net/, 2006.

[OSH05] OWENS J. D., SENGUPTA S., HORN D.: *Assessment of Graphic Processing Units (GPUs) for Department of Defense (DoD) Digital Signal Processing (DSP) Applications*. Tech. Rep. ECE-CE-2005-3, Department of Electrical and Computer Engineering, University of California, Davis. Available at http://www.ece.ucdavis.edu/cerl/techreports/2005-3/, October 2005.

[OSW*03] OpenGL Architecture Review Board, SHREINER D., WOO M., NEIDER J., DAVIS T.: *OpenGL Programming Guide: The Official Guide to Learning OpenGL*. Addison-Wesley, 2003.

[Owe05] OWENS J.: Streaming architectures and technology trends. In *GPU Gems 2*, M. Pharr, editor. Addison Wesley, chapter 29, pp. 457–470, March 2005.

[PAB*05] PHAM D., ASANO S., BOLLIGER M., DAY M. N., HOFSTEE H. P., JOHNS C., KAHLE J., KAMEYAMA A., KEATY J., MASUBUCHI Y., RILEY M., SHIPPY D., STASIAK D., WANG M., WARNOCK J., WEITZEL S., WENDEL D., YAMAZAKI T., YAZAWA K.: The design and implementation of a first-generation CELL processor. In *Proceedings of the International Solid-State Circuits Conference*, pp. 184–186, February 2005.

[PBMH02] PURCELL T. J., BUCK I., MARK W. R., HANRAHAN P.: Ray tracing on programmable graphics hardware. *ACM Trans. Graph.* 21(3):703–712, July 2002.

[PDC*03] PURCELL T. J., DONNER C., CAMMARANO M., JENSEN H. W., HANRAHAN P.: Photon mapping on programmable graphics hardware. In *Graphics Hardware*, pp. 41–50, July 2003.

[PH89] POTMESIL M., HOFFERT E. M.: The Pixel Machine: A parallel image computer. In *Comput. Graph. (Proceedings of SIGGRAPH 1989)*, vol. 23, pp. 69–78, July 1989.

[POAU00] PEERCY M. S., OLANO M., AIREY J., UNGAR P. J.: Interactive multi-pass programmable shading. In *Proceedings of ACM SIGGRAPH 2000, Computer Graphics Proceedings, Annual Conference Series*, pp. 425–432, July 2000.

[PS03] PURCELL T. J., SEN P.: Shadesmith fragment program debugger. Available at http://graphics.stanford.edu/projects/shadesmith/, 2003.

[Pur04] PURCELL T. J.: *Ray Tracing on a Stream processor.* PhD thesis, Stanford University, March 2004.

[RR86] ROSSIGNAC J. R., REQUICHA A. A. G.: Depth-buffering display techniques for constructive solid geometry. *IEEE Comput. Graph. Appl.* 6(9):29–39, September 1986.

[RS01a] RUMPF M., STRZODKA R.: Level set segmentation in graphics hardware. In *Proceedings of the IEEE International Conference on Image Processing (ICIP '01)*, vol. 3, pp. 1103–1106, October 2001.

[RS01b] RUMPF M., STRZODKA R.: Nonlinear diffusion in graphics hardware. In *Data Visualization 2001 (Proceedings of the EG/IEEE VisSym)*, pp. 75–84, May 2001.

[RS01c] RUMPF M., STRZODKA R.: Using graphics cards for quantized FEM computations. In *Proceedings of VIIP 2001*, pp. 193–202, 2001.

[RS05] RUMPF M., STRZODKA R.: Graphics processor units: New prospects for parallel computing. In *Numerical Solution of Partial Differential Equations on Parallel Computers, vol. 51 of Lecture Notes in Computational Science and Engineering*. Springer-Verlag, pp. 89–134, 2005.

[RSHGE99] REZK-SALAMA C., HASTREITER P., GREINER G., ERTL T.: Non-linear registration of pre- and intraoperative volume data based on piecewise linear transformations. In *Proceedings of Vision, Modeling, and Visualization*, pp. 365–372, November 1999.

[RSSF02] REINHARD E., STARK M., SHIRLEY P., FERWERDA J.: Photographic tone reproduction for digital images. *ACM Trans. Graph.* 21(3):267–276, July 2002.

[RTB*92] RHOADES J., TURK G., BELL A., STATE A., NEUMANN U., VARSHNEY A.: Real-time procedural textures. In *1992 Symposium on Interactive 3D Graphics*, vol. 25, pp. 95–100, March 1992.

[SAA03] SUN C., AGRAWAL D., ABBADI A. E.: Hardware acceleration for spatial selections and joins. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pp. 455–466, June 2003.

[SC05] SMIRNOV A., CHIUEH T.: *An Implementation of a FIR Filter on a GPU*. Tech. rep., Experimental Computer Systems Lab, Stony Brook University. Available at http://www.ecsl.cs.sunysb.edu/fir/, 2005.

[SDR03] STRZODKA R., DROSKE M., RUMPF M.: Fast image registration in DX9 graphics hardware. *J. Med. Inform. Techn.* 6:43–49, November 2003.

[SDR04] STRZODKA R., DROSKE M., RUMPF M.: Image registration by a regularized gradient flow: A streaming implementation in DX9 graphics hardware. *Computing* 73(4):373–389, November 2004.

[SG04] STRZODKA R., GARBE C.: Real-time motion estimation and visualization on graphics cards. In *IEEE Visual.*, pp. 545–552, October 2004.

[SG06] STRZODKA R., GÖDDEKE D.: Mixed precision methods for convergent iterative schemes. In *Proceedings of the Workshop on Edge Computing Using New Commodity Architectures*, pp. D-59–60, May 2006.

[SHN03] SHERBONDY A., HOUSTON M., NAPEL S.: Fast volume segmentation with simultaneous visualization using programmable graphics hardware. In *IEEE Visual 2003*, pp. 171–176, October 2003.

[SALP05] SZIRMAY-KALOS L., ASZÓDI B., LAZÁNYI I., PREMECZ M.: Approximate ray-tracing on the GPU with distance imposters. *Comput. Graph. Forum* 24(3): 685–704, September 2005.

[SKP07] SHAH M. A., KONTTINEN J., PATTANAIK S. N.: Caustics mapping: An image-space technique for real-time caustics. *IEEE Trans. Visual. Comput. Graph.* 13(2):272–280, March/April 2007.

[SL05] SUMANAWEERA T., LIU D.: Medical image reconstruction with the FFT. In *GPU Gems 2*, M. Pharr, editor. Addison Wesley, chapter 48, pp. 765–784, March 2005.

[SLJ98] STEWART N., LEACH G., JOHN S.: An improved Z-buffer CSG rendering algorithm. In *1998 SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pp. 25–30, August 1998.

[SLO06] SENGUPTA S., LEFOHN A. E., OWENS J. D.: A work-efficient step-efficient prefix sum algorithm. In *Proceedings of the Workshop on Edge Computing Using New Commodity Architectures*, pp. D-26–27, May 2006.

[SOM04] SUD A., OTADUY M. A., MANOCHA D.: DiFi: Fast 3D distance field computation using graphics hardware. *Comput. Graph. Forum* 23(3):557–566, September 2004.

[SPG03] SIGG C., PEIKERT R., GROSS M.: Signed distance transform using graphics hardware. In *IEEE Visual.*, pp. 83–90, October 2003.

[ST04] STRZODKA R., TELEA A.: Generalized distance transforms and skeletons in graphics hardware. In *Proceedings of EG/IEEE TCVG Symposium on Visualization (VisSym 2004)*, pp. 221–230, 2004.

[STM04] SANDER P., TATARCHUK N., MITCHELL J. L.: *Explicit Early-Z Culling for Efficient Fluid Flow Simulation and Rendering*. Tech. rep., ATI Research. Available at http://www.ati.com/developer/techreports/ATITechReport_EarlyZFlow.pdf, August 2004.

[Str02] STRZODKA R.: Virtual 16 bit precise operations on RGBA8 textures. In *Proceedings of Vision, Modeling, and Visualization*, pp. 171–178, November 2002.

[Str04] STRZODKA R.: *Hardware Efficient PDE Solvers in Quantized Image Processing*. PhD thesis, University of Duisburg-Essen, 2004.

[SW04] SCHIWIETZ T., WESTERMANN R.: GPU-PIV. In *Proceedings of Vision, Modeling, and Visualization*, pp. 151–158, November 2004.

[THO06] THOMPSON C. J., HAHN S., OSKIN M.: Using modern graphics architectures for general-purpose computing: A framework and analysis. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 306–317, November 2006.

[TPO06] TARDITI D., PURI S., OGLESBY J.: Accelerator: Using data-parallelism to program GPUs for general-purpose uses. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 325–335, October 2006.

[Tre06] TREBILCO D.: GLIntercept. Available at http://glintercept.nutty.org/, 2006.

[TS00] TRENDALL C., STEWART A. J.: General calculations using graphics hardware, with applications to interactive caustics. In *Rendering Techniques 2000: Proceedings of the 11th Eurographics Rendering Workshop*, pp. 287–298, June 2000.

[TS05] THRANE N., SIMONSEN L. O.: *A Comparison of Acceleration structures for GPU Assisted Ray Tracing*. Master's thesis, University of Aarhus, August 2005.

[Ups90] UPSTILL S.: *The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics*. Addison-Wesley, 1990.

[Ven03] VENKATASUBRAMANIAN S.: The graphics card as a stream computer. In *SIGMOD-DIMACS Workshop on Management and Processing of Data Streams*, 2003.

[Ver67] VERLET L.: Computer "experiments" on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules. *Phys. Rev.*, 159, 98–103, July 1967.

[VKG03] VIOLA I., KANITSAR A., GRÖLLER M. E.: Hardware-based nonlinear filtering and segmentation using high-level shading languages. In *IEEE Visual.*, pp. 309–316, October 2003.

[VSC01] VASSILEV T., SPANLANG B., CHRYSANTHOU Y.: Fast cloth animation on walking avatars. *Computer Graphics Forum* 20(3):260–267, September 2001.

[WD06] WYMAN C., DAVIS S.: Interactive image-space techniques for approximating caustics. In *SI3D '06: Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, pp. 153–160, March 2006.

[WHE01] WEISKOPF D., HOPF M., ERTL T.: Hardware-accelerated visualization of time-varying 2D and 3D vector fields by texture advection via programmable per-pixel operations. In *Proceedings of Vision, Modeling, and Visualization*, pp. 439–446, November 2001.

[Whi80] WHITTED T.: An improved illumination model for shaded display. *Communications of the ACM* 23(6):343–349, June 1980.

[WK04] WOETZEL J., KOCH R.: Multi-camera realtime depth estimation with discontinuity handling on PC graphics hardware. In *Proceedings of the 17th International Conference on Pattern Recognition*, pp. 741–744, August 2004.

[Wlo04] WLOKA M.: Interactive geometric and scientific computations using graphics hardware—implementing a GPU-efficient FFT. *ACM SIGGRAPH Course Notes*. Presented by John Spitzer, July 2003.

[WSE04] WEISKOPF D., SCHAFHITZEL T., ERTL T.: GPU-based nonlinear ray tracing. *Comput. Graph. Forum* 23(3):625–633, September 2004.

[WWHL04] WANG J., WONG T.-T., HENG P.-A., LEUNG C.-S.: Discrete wavelet transform on GPU. Available at http://www.cse.cuhk.edu.hk/∼ttwong/software/dwtgpu/dwtgpu.html, August 2004.

[XM05] XU F., MUELLER K.: Accelerating popular tomographic reconstruction algorithms on commodity PC graphics hardware. *IEEE Trans. on Nuclear Science* 52(3):654–663, June 2005.

[YLPM05] YOON S.-E., LINDSTROM P., PASCUCCI V., MANOCHA D.: Cache-oblivious mesh layouts. *ACM Trans. Graph.* 24(3):886–893, August 2005.

[YP05] YANG R., POLLEFEYS M.: A versatile stereo implementation on commodity graphics hardware. *Real-Time Imaging* 11(1):7–18, February 2005.

[YW03] YANG R., WELCH G.: Fast image segmentation and smoothing using commodity graphics hardware. *J. Graph. Tools.* 7(4):91–100, 2003.

[Zel05] ZELLER C.: Cloth simulation on the GPU. In *ACM SIGGRAPH 2005 Conference Abstracts and Applications*, August 2005.