A Survey of General-Purpose Computation on Graphics Hardware

John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell

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. This survey should be of particular interest to researchers who are interested in using the latest GPGPU applications in their systems of interest.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Hardware Architecture; D.2.2 [Software Engineering]: Design Tools and Techniques

1. Introduction: Why GPGPU?

Commodity computer graphics chips are probably today's most powerful computational hardware for the dollar. These chips, known generically as Graphics Processing Units or GPUs, have gone from afterthought peripherals to modern, powerful, and programmable processors in their own right. Many researchers and developers have become interested in harnessing the power of commodity graphics hardware for general-purpose computing. Recent years have seen an explosion in interest in such research efforts, known collectively as GPGPU (for "General Purpose GPU") computing. In this State of the Art Report we summarize the motivation and essential developments in the hardware and software behind GPGPU. We give an overview of the techniques and computational building blocks used to map general-purpose computation to graphics hardware and provide a survey of the various general-purpose computing applications 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 NVIDIA GeForce 6800 Ultra ($417 as of June 2005) can achieve a sustained 35.2 GB/sec of memory bandwidth; the ATI X800 XT ($447) can sustain over 63 GFLOPS (compare to 14.8 GFLOPS theoretical peak for a 3.7 GHz Intel Pentium4 SSE unit [Buc04]). GPUs are also on the cutting edge of processor technology; for example, the most recently announced GPU at this writing contains over 300 million transistors and is built on a 110-nanometer fabrication process.

Not only is current graphics hardware fast, it is accelerating quickly. For example, the measured throughput of the GeForce 6800 is more than double that of the GeForce 5900, NVIDIA's previous flagship architecture. 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 outpaces the oft-quoted Moore's Law as applied to traditional microprocessors; compare to a yearly rate of roughly 1.4× for CPU perfor-

John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. "A Survey of General-Purpose Computation on Graphics Hardware." In *Eurographics 2005, State of the Art Reports*, August 2005, pp. 21-51.

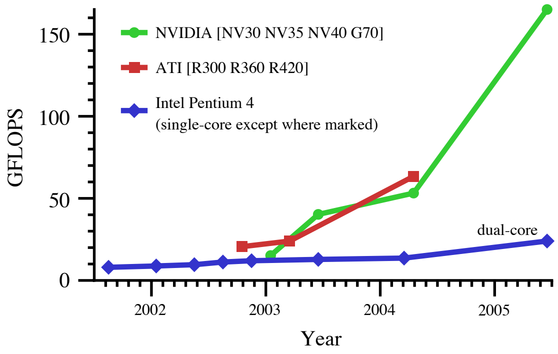


Figure 1: *The programmable floating-point performance of GPUs (measured on the multiply-add instruction as 2 floating-point operations per MAD) has increased dramatically over the last four years when compared to CPUs. Figure courtesy Ian Buck, Stanford University.*

mance [EWN05]. Put another way, graphics hardware performance is roughly doubling every six months (Figure 1).

Why is the performance of graphics hardware increasing more rapidly than that of CPUs? After all, semiconductor capability, driven by advances in fabrication technology, is increasing at the same rate for both platforms. The disparity in performance can be attributed to fundamental architectural differences: CPUs are optimized for high performance on sequential code, so many of their transistors are dedicated to supporting non-computational tasks like branch prediction and caching. On the other hand, the highly parallel nature of graphics computations enables GPUs to use additional transistors for computation, achieving higher arithmetic intensity with the same transistor count. We discuss the architectural issues of GPU design further in Section 2.

This computational power is available and inexpensive; these chips can be found in off-the-shelf graphics cards built for the PC video game market. A typical latest-generation card costs $400–500 at release and drops rapidly as new hardware emerges.

1.2. Flexible and Programmable

Modern graphics architectures have become flexible as well as powerful. Once fixed-function pipelines capable of outputting only 8-bit-per-channel color values, modern GPUs include fully programmable processing units that support vectorized floating-point operations at full IEEE single precision. High level languages have emerged to support the new programmability of the vertex and pixel pipelines [BFH*04, MGAK03, MTP*04]. Furthermore, additional levels of programmability are emerging with every major generation of GPU (roughly every 18 months). Examples of major architectural changes in the current generation (as of this writing) GPUs include vertex texture access, full branching support in the vertex pipeline, and limited branching capability in the fragment pipeline. The next generation is expected to expand on these changes and add "geometry shaders", or programmable primitive assembly, bringing flexibility to an entirely new stage in the pipeline. In short, the raw speed, increased precision, and rapidly expanding programmability of the hardware make it an attractive platform for general-purpose computation.

1.3. Limitations and Difficulties

The GPU is hardly a computational panacea. The arithmetic power of the GPU is a result of its highly specialized architecture, evolved over the years to extract the maximum performance on the highly parallel tasks of traditional computer graphics. The rapidly increasing flexibility of the graphics pipeline, coupled with some ingenious uses of that flexibility by GPGPU developers, has enabled a great 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, which is dominated by memory communication and difficult to parallelize.

Today's GPUs also lack some fundamental computing constructs, such as 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. Finally, while the recent increase in precision to 32-bit floating point has enabled a host of GPGPU applications, 64-bit double precision arithmetic appears to be on the distant horizon at best. The lack of double precision hampers or prevents GPUs from being applicable to many very large-scale computational science problems.

GPGPU computing presents challenges even for problems that map well to the GPU, because despite advances in programmability and high-level languages, graphics hardware remains difficult to apply to non-graphics tasks. The GPU uses an unusual programming model (Section 2.3), so effective GPU programming is not simply a matter of learning a new language, or writing a new compiler backend. Instead, the computation must be recast into graphics terms by a programmer familiar with the underlying hardware, its design, limitations, and evolution. We emphasize that these difficulties are intrinsic to the nature of computer graphics hardware, not simply a result of immature technology. Computational scientists cannot simply wait a generation or two for a graphics card with double precision and a FORTRAN compiler. 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 scientific or engineering domain. But despite the programming challenges, the potential benefits—a leap forward

John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. "A Survey of General-Purpose Computation on Graphics Hardware." In *Eurographics 2005, State of the Art Reports*, August 2005, pp. 21-51.

in computing capability, and a growth curve much faster than traditional CPUs—are too large to ignore.

1.4. GPGPU Today

An active, vibrant community of GPGPU developers has emerged (see `http://GPGPU.org/`), and many promising early applications of GPGPU have appeared already in the literature. We give an overview of GPGPU applications, which range from numeric computing operations such as dense and sparse matrix multiplication techniques [KW03] or multigrid and conjugate-gradient solvers for systems of partial differential equations [BFGS03, GWL*03], to computer graphics processes such as ray tracing [PBMH02] and photon mapping [PDC*03] usually performed offline on the CPU, to physical simulations such as fluid mechanics solvers [BFGS03, Har04, KW03], to database and data mining operations [GLW*04, GRM05]. We cover these and more applications in Section 5.

2. Overview of Programmable Graphics Hardware

The emergence of general-purpose applications on graphics hardware has been driven by the rapid improvements in the programmability and performance of the underlying 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 every stage. 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.

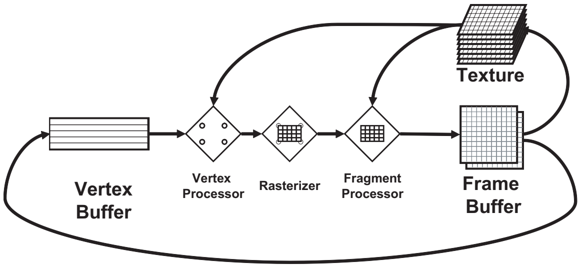


Figure 2: *The modern graphics hardware pipeline. The vertex and fragment processor stages are both programmable by the user.*

The input to the pipeline is a list of geometry, expressed as vertices in object coordinates; the output is an image in a framebuffer. The first stage of the pipeline, the geometry stage, transforms each vertex from object space into screen space, assembles the vertices into triangles, and traditionally performs lighting calculations on each vertex. The output of the geometry stage is triangles in screen space. The next stage, rasterization, both determines the screen positions covered by each triangle and interpolates per-vertex parameters across the triangle. The result of the rasterization stage is a fragment for each pixel location covered by a triangle. The third stage, the fragment stage, computes the color for each fragment, using the interpolated values from the geometry stage. This computation can use values from global memory in the form of *textures*; typically the fragment stage generates addresses into texture memory, fetches their associated texture values, and uses them to compute the fragment color. In the final stage, composition, fragments are assembled into an image of pixels, usually by choosing the closest fragment to the camera at each pixel location. This pipeline is described in more detail in the OpenGL Programming Guide [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 six 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 geometry stage and the fragment stage. In the fixed-function pipeline, the geometry 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

John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. "A Survey of General-Purpose Computation on Graphics Hardware." In *Eurographics 2005, State of the Art Reports*, August 2005, pp. 21-51.

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 2005, 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 NVIDIA GeForce 6800 Ultra and ATI Radeon X800 XT each have 6 vertex and 16 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*04] 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 processor. 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, 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. For more detail on GPU flow control, see Harris and Buck [HB05].

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 NV40 Quadro GPUs. All current GPU fragment processors are SIMD. In SIMD, when evaluation of the branch condition is identical on all active processors, only the taken side of the branch must be evaluated, but 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.

2.4.2.1. 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].

2.4.2.2. 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].

2.4.2.3. 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

John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. "A Survey of General-Purpose Computation on Graphics Hardware." In *Eurographics 2005, State of the Art Reports*, August 2005, pp. 21-51.

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].

2.4.2.4. 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].

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 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++ [MTP*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.

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*04]. Brook can use the GPU as a compilation target. Brook streams are con-

John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. "A Survey of General-Purpose Computation on Graphics Hardware." In *Eurographics 2005, State of the Art Reports*, August 2005, pp. 21-51.

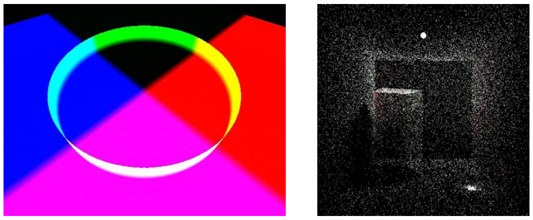


Figure 3: *Examples of fragment program "printf" debugging. The left image encodes ray-object intersection hit points as r, g, b color. The right image draws a point at each location where a photon was stored in a photon map. (Images generated by Purcell et al. [PDC*03].)*

ceptually 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.

Finally, the Glift template library provides a generic template library for a wide range of GPU data structures [LKS*05]. It is designed to be a stand-alone GPU data structure library that helps simplify data structure design and separate GPU algorithms from data structures. The library integrates with a C++, Cg, and OpenGL GPU development environment.

3.3. Debugging Tools

A high-level programming language gives a programmer the ability to create complex programs with much less effort than assembly language writing. With several high-level programming languages available to choose from, generating complex programs to run on the GPU is fairly straightforward. But all good development platforms require more than just a language to write in. One of the most important tools needed for successful platforms is a debugger. Until recently, support for debugging on GPUs was fairly limited.

The needs of a debugger for GPGPU programming are very similar to what traditional CPU debuggers provide, 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.

A GPU debugger has a challenge in that it must be able to provide debug information for multiple vertices or pixels at a time. 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. Drawing data values to the screen for visualization often requires some amount of scaling and biasing for values that don't fit in an 8-bit color buffer (e.g. when rendering floating point data). The ideal GPGPU debugger would automate printf-style debugging, including programmable scale and bias, 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 [Gra05] and GLIntercept [Tre05] 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. There is currently no specific support for debugging shaders. GLIntercept does provide runtime shader editing, but again is lacking in shader debugging support.

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 [App05b] 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 [Bax05] 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 and stepping. Shadesmith works by decomposing a fragment program into multiple independent shaders, one for each assembly instruction in the shader, then adding output instructions to each of these smaller programs. The effects of executing any instruction can be determined by running the right shader. Shadesmith automates the printf debug by running the appropriate shader for a register that is being watched, and drawing the output to an image window. Tracking multiple registers is done by running multiple programs and displaying the results in separate windows. Shadesmith also provides the programmer the ability to write programs to arbitrarily scale and bias the watched registers. 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. GPGPU programs today are generally too complex for assembly level programming. Additionally, Shadesmith only works for OpenGL fragment programs, and provides no support for debugging OpenGL state.

Finally, Duca et al. have 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, 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. Since map is also the most fundamental operation to GPGPU applications, we will describe its GPU implementation in detail. In Section 2.3, we saw how to use the GPU's fragment processor as the computation engine for GPGPU. These five steps are the essence of the map implementation on the GPU. First, the programmer writes a function that gets applied to every element as a fragment program, and stores the stream of data elements in texture memory. The programmer then invokes the fragment program by rendering geometry that causes the rasterizer to generate a fragment for every pixel location in the specified geometry. The fragments are processed by the fragment processors, which apply the program to the input elements. The result of the fragment program execution is the result 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 data in $O(\log n)$ steps using the parallel GPU hardware, compared to $O(n)$ 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*04].

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

John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. "A Survey of General-Purpose Computation on Graphics Hardware." In *Eurographics 2005, State of the Art Reports*, August 2005, pp. 21-51.

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 [Buc05].

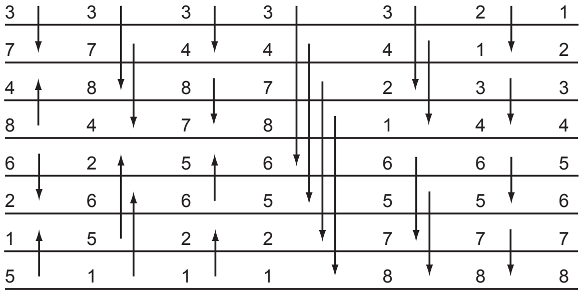
4.1.4. Stream Filtering

Many algorithms require the ability to select a subset of elements from a stream, and discard the rest. This *stream filtering* operation is essentially a nonuniform reduction. These operations can not rely on standard reduction mechanisms, because the location and number of elements to be filtered is variable and not known a priori. Example algorithms that benefit from stream filtering 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 [Hor05b] that implements stream filtering on the GPU. Using a combination of scan [HS86] and search, stream filtering can be achieved in $O(\log n)$ passes.

4.1.5. 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. Unfortunately, nearly all of the classic sorting methods are not applicable to a clean GPU implementation. The main reason these algorithms are not GPU friendly? Classic sorting algorithms are data-dependent and generally require scatter operations. 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 GPU. To make efficient use of GPU resources, a GPU-based sort should be oblivious to the input data, and should not require scatter.



3	3	3	3	3	2	1
7	7	4	4	4	1	2
4	8	8	7	2	3	3
8	4	7	8	1	4	4
6	2	5	6	6	6	5
2	6	6	5	5	5	6
1	5	2	2	7	7	7
5	1	1	1	8	8	8

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.*

Most GPU-based sorting implementations [BP04, CND03, KSW04, KW05, 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 path. The fixed communication pattern means the problem can be stated in terms of gather rather than a 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 $O(n \log^2 n)$ 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 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

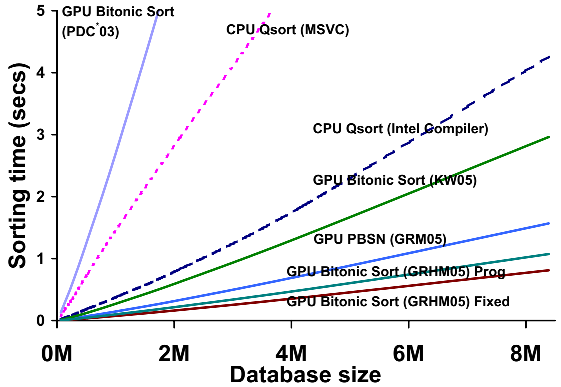


Figure 5: *Performance of CPU-based and GPU-based sorting algorithms on IEEE 16-bit floating point values. The CPU-based Qsort available in the Intel compiler is optimized using hyperthreading and SSE instructions. We observe that the cache-efficient GPU-based sorting network algorithm is nearly 6 times 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.*

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 and is limited on current hardware to IEEE 16-bit floating point values. Alternatively, the limitation to IEEE 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 nearly 1.2 times 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 IEEE 16-bit floating point values using a PC with a 3.4 GHz Pentium 4 CPU and an NVIDIA GeForce 6800 Ultra GPU. A sorting library implementing the algorithm for 16-bit and 32-bit floats is freely available for noncommercial use [GPU05].

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].

4.1.6. 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 (see 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 [Hor05b, 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 we have not yet found a way 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]. 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 candidate neighbors are visited during the search. The details of the kNN-grid implementation are beyond the scope of this paper, and readers are encouraged to review the original papers for more details [PDC*03, Pur04]. The next section of this report 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*05] give a more detailed overview of GPGPU data structures and the GPU memory model.

The GPU Memory Model Before describing GPGPU data structures, we briefly describe the memory primitives with which they are built. 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 (see 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 two reasons. First, the maximum 1D texture size is often too small to be useful and second, 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 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*05] 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*04, LKS*05, MTP*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*04, 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 CPU/GPU interfaces to these structures. Lefohn et al. give code examples for optimized implementations [LKO05].

John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. "A Survey of General-Purpose Computation on Graphics Hardware." In *Eurographics 2005, State of the Art Reports*, August 2005, pp. 21-51.

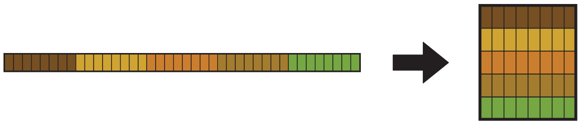


Figure 6: *GPU-based multidimensional arrays usually store data in 2D texture memory. Address translators for N-D arrays generally convert N-D addresses to 1D, then to 2D.*

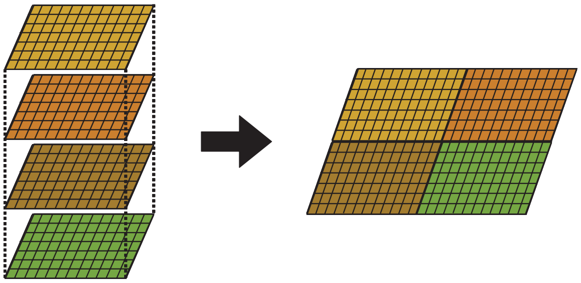


Figure 7: *For the special case of 3D-to-2D conversions or* flat 3D textures, *2D slices of the 3D array are packed into a single 2D texture. This structure maintains 2D locality and therefore supports native bilinear filtering.*

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 et al. also describe a packed texture format for banded sparse matrices. Buck et al. 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. See Section 4.4 for more detail about GPGPU linear algebra.

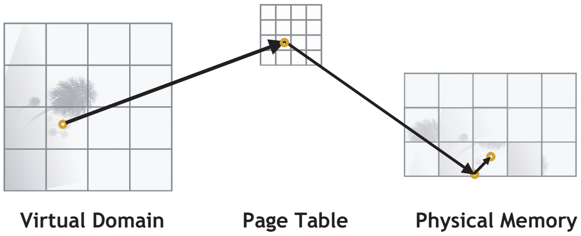


Figure 8: *Page table address data structures can be used to represent dynamic sparse or adaptive GPGPU data structures. For sparse arrays, page tables map only a subset of possible pages to texture memory. Page-table-based adaptive arrays map either uniformly sized physical pages to a varying number 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*05].*

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 stored 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 request by

John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. "A Survey of General-Purpose Computation on Graphics Hardware." In *Eurographics 2005, State of the Art Reports*, August 2005, pp. 21-51.

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].

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 preprocess 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.

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.

Lefohn et al. use a mipmap hierarchy of page tables to define quadtree-like and octree-like dynamic structures [LKS*05, LSK*05]. 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.

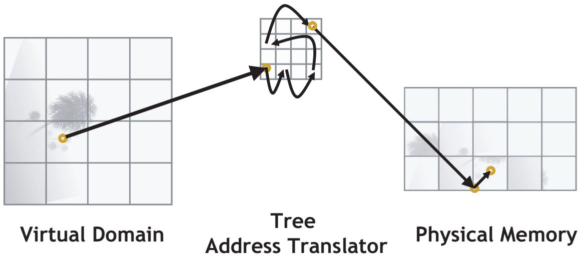


Figure 9: *Tree-based address translators can be used in place of page tables to represent adaptive data structures such as quadtrees, octrees and k-d trees [FS05, LHN05]. Trees consume less memory than page table structures but result in longer access times and are more costly to incrementally update.*

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. Non-Indexable 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*05].

Efficient dynamic parallel data structures are an active area of research. For example, structures such as priority queues (see Section 4.1.6), 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.

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.

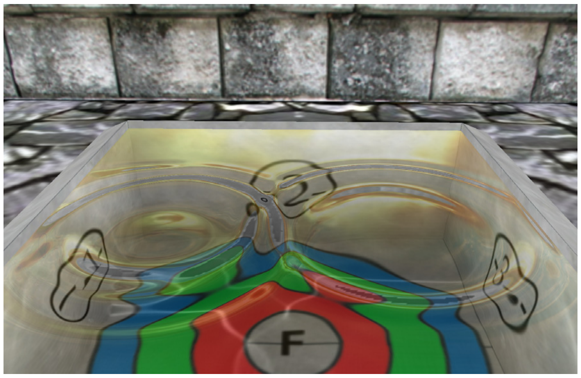


Figure 10: *Solving the wave equation PDE on the GPU allows for fast and stable rendering of water surfaces. (Image generated by Krüger et al. [KW03])*

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 $F(x, y, y', \cdots, y^{(n)}) = 0$ where $y^{(n)}$ is the nth derivative with respect to x. PDEs, on the other hand, are equations involving functions and their partial derivatives, like the wave equation $\frac{\partial^2\psi}{\partial x^2} + \frac{\partial^2\psi}{\partial y^2} + \frac{\partial^2\psi}{\partial z^2} = \frac{\partial^2\psi}{v^2\partial t^2}$ (see 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 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] (see 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].

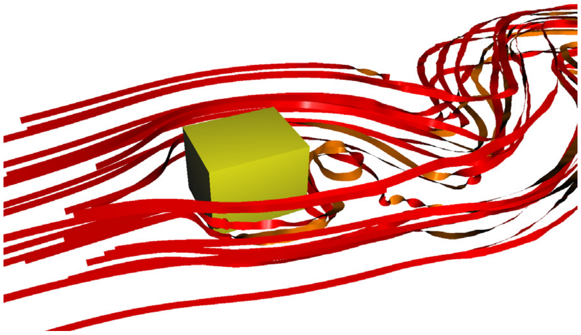


Figure 11: *GPU-computed stream ribbons in a 3D flow field. The entire process from vectorfield interpolation and integration to curl computation, and finally geometry generation and rendering of the stream ribbons, is performed on the GPU [KKKW05].*

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 [GV96] (Bolz et al. [BFGS03] and Krüger and Westermann [KW03]), the multigrid method [BHM00] (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 et al. 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 representations of vectors, dense matrices, and multiple types of sparse matrices [KW03]. Their implementation was based on a 2D texture representation for vectors in which a vector is laid out into the RGBA components of a 2D texture. A matrix was composed of such vector textures, either split column-wise for dense matrices or diagonally for banded sparse matrices. With this representation a component-wise vector-vector operation—add, multiply, and so on—requires rendering only one quad textured with the two input vector textures with a short shader that does two fetches into the input textures, combines the results (e.g. add or multiply), and outputs this to a new texture representing the result vector. A matrix-vector operation in turn is executed as multiple vector-vector operations: the columns or diagonals are multiplied with the vector one at a time and are added to the result vector. In this way a five-banded matrix—for instance, occurring in the Poisson equation of the Navier-Stokes fluid simulation—can be multiplied with a vector by rendering only five quads. The set of basic operations is completed by the reduce operation, which computes single values out of vectors e.g., the sum of all vector elements (Section 4.1.2).

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 (see Figure 12).

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 200 fps on current GPUs such as ATI's Radeon X800 [KW03].*

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*04] featured a spMatrixVec test that used a padded compressed sparse row format.

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].

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 $(a_1, a_2, ..., a_m)$, 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, and can be implemented using the counting capability of the occlusion queries.

Join Operations Join operations combine the records in multiple relations using a common join 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 4.1.5 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.

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 their method to proximity detection [HZLM01]. Bohn et al. 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 (see 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]. 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] (see Section 4.3).

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].

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× 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 et al. [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× 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 et al. 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 that provided a speedup of 10–15× over a highly optimized CPU-based

John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. "A Survey of General-Purpose Computation on Graphics Hardware." In *Eurographics 2005, State of the Art Reports*, August 2005, pp. 21-51.

solver [Ins03] (Figure 13). They used a page table data structure to store and compute only a sparse subset of the volume on the GPU. Their scheme used the CPU as a GPU memory manager, and the GPU requested memory allocation changes by sending a bit vector message to the CPU. 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 the segmentation systems presented by Lefohn et al. and Sherbondy et al. were integrated with interactive volume renderers. As such, users could interactively control the evolving PDE computation. Lefohn et al. used their system in a tumor segmentation user study [LCW03]. The study found that relatively untrained users could segment tumors from a publicly available MRI data set in approximately six minutes. The resulting segmentations were more precise and equivalently accurate to those produced by trained experts who took multiple hours to segment the tumors manually. Cates et al. extended this work to multi-channel (color) data and provided additional information about the statistical methods used to evaluate the study [CLW04].

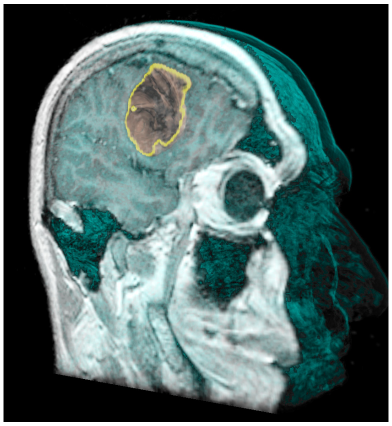


Figure 13: *Interactive volume segmentation and visualization of Magnetic Resonance Imaging (MRI) data on the GPU enables fast and accurate medical segmentations. Image generated by Lefohn et al. [LKHW04].*

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 [Ope05], 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 computed 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 at 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. 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. Their system is 4.5 times faster than a CPU equivalent (as of May 2004), and they expect the additional arithmetic capability of newer graphics hardware will allow the use of more advanced estimation models (such as estimation of brightness changes) in real time.

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 timeframes from seconds to tens of seconds [XM05].

Signal Processing Motivated by the high arithmetic capabilities of modern GPUs, several projects have developed GPU implementations of the fast Fourier transform (FFT) [BFH*04, JvHK04, MA03, SL05]. (The *GPU Gems 2* chapter by Sumanaweera and Liu, in particular, gives a

John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. "A Survey of General-Purpose Computation on Graphics Hardware." In *Eurographics 2005, State of the Art Reports*, August 2005, pp. 21-51.

detailed description of the FFT and their GPU implementation [SL05].) In general, these implementations operate on 1D or 2D input data, use a radix-2 decimation-in-time approach, and require one fragment-program pass per FFT stage. The real and imaginary components of the FFT can be computed in two components of the 4-vectors in each fragment processor, so two FFTs can easily be processed in parallel. These implementations are primarily limited by memory bandwidth and the lack of effective caching in today's GPUs, and only by processing two FFTs simultaneously can match the performance of a highly tuned CPU implementation [FJ98].

Daniel Horn maintains an open-source optimized FFT library based on the Brook distribution [Hor05a]. 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 DFT [WWHL05], which has been integrated into an open-source JPEG2000 codec called "JasPer" [Ada05].

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 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 [Bio05].

Image/Video Processing Frameworks Apple's Core Image and Core Video frameworks allow GPU acceleration of image and video processing tasks [App05a]; the open-source framework Jahshaka uses GPUs to accelerate video compositing [Jah05].

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 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 brute-force-test every ray against every surface for intersection, so there are several data structures that reduce the total number of surfaces rays need to test against (called acceleration structures). 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.

Some of the early 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].

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.

John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. "A Survey of General-Purpose Computation on Graphics Hardware." In *Eurographics 2005, State of the Art Reports*, August 2005, pp. 21-51.

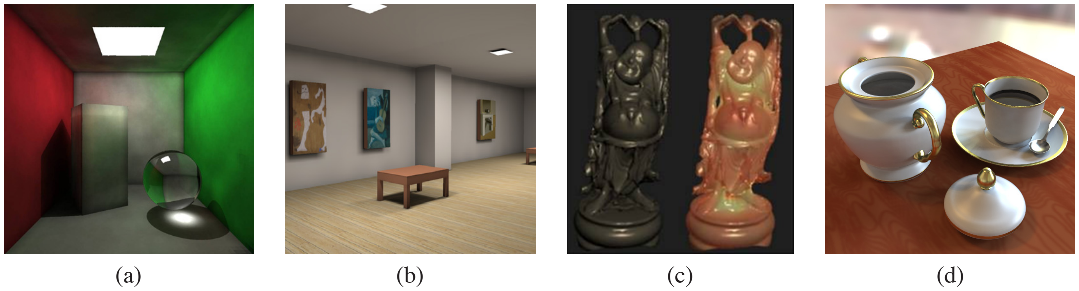


Figure 14: *Sample images from several global illumination techniques implemented on the GPU. (a) Ray tracing and photon mapping [PDC*03]. (b) Radiosity [CHL04]. (c) Subsurface scattering [CHH03]. (d) Final gather by rasterization [Hac05].*

5.4.2. Photon Mapping

Photon mapping [Jen96] is a two-stage global illumination algorithm. The first stage consists of emitting photons out 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 at each surface point that was contributed 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 discussed previously. Constructing the photon map and indexing the map to find good energy estimates at each image point are much more difficult on the GPU. Ma and McCool proposed a low-latency photon lookup algorithm based on hash tables [MM02]. Their algorithm was never implemented on the GPU, and construction of the hash table is not currently amenable to a GPU implementation. 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] (see Sections 4.1.6 and 4.2 for some implementation details). Figure 14a shows an image rendered with this system. Finally, 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].

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 techniques with global illumination effects and combine the results. 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]. Each of these scene rasterizations is accumulated to form an estimate of the indirect illumination at each visible point. This indirect illumination estimate is combined with direct illumination computed with a traditional rendering technique like photon mapping. 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.

Finally, 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.

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, RR86, GMTF89, SLJ98, GKMV03].

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, HTG03, HTG04, HCK*99, KP03, MOK95, RMS92, SF91, VSC01, GRLM03]. 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].

Shadow Generation Shadows correspond to the regions visible to the eye and not visible to the light. Popular techniques include variations of shadow maps [SKv*92, HS99, BAS02, SD02, Sen04, LSK*05] and shadow volumes [Cro77, Hei91, EK02]. Algorithms have also been proposed to generate soft shadows [BS02, ADMAM03, CD03].

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 and a better layout effectively utilizes the caches on GPUs such as vertex caches. 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× 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 marketing 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, semi-linear 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*04, 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 approaching something like maturity. 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 in GPGPU 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*04] 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 data-parallel processors are beginning to appear in the mass market, most notably the Cell processor produced by IBM, Sony, and Toshiba [PAB*05]. The tiled architecture of the 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 potentially better suited for general-purpose computing. As GPUs grow more general, low-level programming is supplanted by high-level languages and toolkits, and new contenders such as the Cell chip emerge, GPGPU researchers face the challenge of transcending their computer graphics roots and developing computational idioms, techniques, and frameworks for desktop data-parallel computing.

Acknowledgements

Thanks to Ian Buck, Jeff Bolz, 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. http://www.ece.uvic.ca/~mdadams/jasper/, 2005.

[ADMAM03] ASSARSSON U., DOUGHERTY M., MOUNIER M., AKENINE-MÖLLER T.: An optimized soft shadow volume algorithm with real-time performance. In *Graphics Hardware 2003* (July 2003), pp. 33–40.

[App05a] Apple Computer Core Image. http://www.apple.com/macosx/tiger/coreimage.html, 2005.

[App05b] Apple Computer OpenGL shader builder / profiler. http://developer.apple.com/graphicsimaging/opengl/, 2005.

[BAS02] BRABEC S., ANNEN T., SEIDEL H.-P.:

Shadow mapping for hemispherical and omnidirectional light sources. In *Advances in Modelling, Animation and Rendering (Proceedings of Computer Graphics International 2002)* (July 2002), pp. 397–408.

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

[Bax05] Baxter B.: The image debugger. `http://www.billbaxter.com/projects/imdebug/`, 2005.

[Ben75] Bentley J. L.: Multidimensional binary search trees used for associative searching. *Communications of the ACM 18*, 9 (Sept. 1975), 509–517.

[BFGS03] Bolz J., Farmer I., Grinspun E., Schröder P.: Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. *ACM Transactions on Graphics 22*, 3 (July 2003), 917–924.

[BFH*04] Buck I., Foley T., Horn D., Sugerman J., Fatahalian K., Houston M., Hanrahan P.: Brook for GPUs: Stream computing on graphics hardware. *ACM Transactions on Graphics 23*, 3 (Aug. 2004), 777–786.

[BHM00] Briggs W. L., Henson V. E., McCormick S. F.: *A Multigrid Tutorial: Second Edition*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[Bio05] BionicFX. `http://www.bionicfx.com/`, 2005.

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

[BP03] Bleiweiss A., Preetham A.: Ashli—Advanced shading language interface. *ACM SIGGRAPH Course Notes* (2003). `http://www.ati.com/developer/SIGGRAPH03/AshliNotes.pdf`.

[BP04] Buck I., Purcell T.: A toolkit for computation on GPUs. In *GPU Gems*, Fernando R., (Ed.). Addison Wesley, Mar. 2004, pp. 621–636.

[BS02] Brabec S., Seidel H.-P.: Single sample soft shadows using depth maps. In *Graphics Interface* (May 2002), pp. 219–228.

[BSAE04] Bandi N., Sun C., Agrawal D., El Abbadi A.: Hardware acceleration in commercial databases: A case study of spatial operations. pp. 1021–1032.

[Buc04] Buck I.: GPGPU: General-purpose computation on graphics hardware—GPU computation strategies & tricks. *ACM SIGGRAPH Course Notes* (Aug. 2004).

[Buc05] Buck I.: Taking the plunge into GPU computing. In *GPU Gems 2*, Pharr M., (Ed.). Addison Wesley, Mar. 2005, ch. 32, pp. 509–519.

[BW03] Baciu G., Wong W. S. K.: Image-based techniques in a hybrid collision detector. *IEEE Transactions on Visualization and Computer Graphics 9*, 2 (Apr. 2003), 254–271.

[CCWG88] Cohen M. F., Chen S. E., Wallace J. R., Greenberg D. P.: A progressive refinement approach to fast radiosity image generation. In *Computer Graphics (Proceedings of SIGGRAPH 88)* (Aug. 1988), vol. 22, pp. 75–84.

[CD03] Chan E., Durand F.: Rendering fake soft shadows with smoothies. In *Eurographics Symposium on Rendering: 14th Eurographics Workshop on Rendering* (June 2003), pp. 208–218.

[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* (2002), vol. 313, AMS, pp. 67–84.

[CH05] Coombe G., Harris M.: Global illumination using progressive refinement radiosity. In *GPU Gems 2*, Pharr M., (Ed.). Addison Wesley, Mar. 2005, ch. 39, pp. 635–647.

[CHH02] Carr N. A., Hall J. D., Hart J. C.: The ray engine. In *Graphics Hardware 2002* (Sept. 2002), pp. 37–46.

[CHH03] Carr N. A., Hall J. D., Hart J. C.: GPU algorithms for radiosity and subsurface scattering. In *Graphics Hardware 2003* (July 2003), pp. 51–59.

[CHL04] Coombe G., Harris M. J., Lastra A.: Radiosity on graphics hardware. In *Proceedings of the 2004 Conference on Graphics Interface* (May 2004), pp. 161–168.

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

[CLW04] CATES J. E., LEFOHN A. E., WHITAKER R. T.: GIST: An interactive, GPU-based level-set segmentation tool for 3D medical images. *Medical Image Analysis 10*, 4 (July/ Aug. 2004), 217–231.

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

[Cro77] CROW F. C.: Shadow algorithms for computer graphics. In *Computer Graphics (Proceedings of SIGGRAPH 77)* (July 1977), vol. 11, pp. 242–248.

[DNB*05] DUCA N., NISKI K., BILODEAU J., BOLITHO M., CHEN Y., COHEN J.: A relational debugging engine for the graphics pipeline. *ACM Transactions on Graphics 24*, 3 (Aug. 2005). To appear.

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

[EK02] EVERITT C., KILGARD M.: Practical and robust stenciled shadow volumes for hardware-accelerated rendering. *ACM SIGGRAPH Course Notes 31* (2002).

[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* (Aug. 1997), pp. 57–68.

[Eng78] ENGLAND J. N.: A system for interactive modeling of physical curved surface objects. In *Computer Graphics (Proceedings of SIGGRAPH 78)* (Aug. 1978), vol. 12, pp. 336–340.

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

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

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

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

[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* (May 1998), vol. 3, pp. 1381–1384.

[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* (May 2004), vol. 5, pp. 93–96.

[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 *Computer Graphics (Proceedings of SIGGRAPH 89)* (July 1989), vol. 23, pp. 79–88.

[FS05] FOLEY T., SUGERMAN J.: KD-Tree acceleration structures for a GPU raytracer. In *Graphics Hardware 2005* (July 2005). To appear.

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

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

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

[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* (Apr. 2005), pp. 49–56.

[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 Transactions on Graphics 24*, 3 (Aug. 2005). To appear.

[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* (Apr. 2003), pp. 177–180.

[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* (Sept. 2004), pp. 534–541.

[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* (Nov. 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* (Mar. 2005), pp. 59–66.

[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* (June 2004), pp. 215–226.

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

[GMTF89] Goldfeather J., Molnar S., Turk G., Fuchs H.: Near real-time CSG rendering using tree normalization and geometric pruning. *IEEE Computer Graphics & Applications 9*, 3 (May 1989), 20–28.

[GPU05] GPUSort: A high performance GPU sorting library. `http://gamma.cs.unc.edu/GPUSORT/`, 2005.

[Gra05] Graphic Remedy gDEBugger. `http://www.gremedy.com/`, 2005.

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

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

[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.

[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 2003* (July 2003), pp. 25–32.

[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* (2005), pp. 611–622.

[GTGB84] Goral C. M., Torrance K. E., Greenberg D. P., Battaile B.: Modelling the interaction of light between diffuse surfaces. In *Computer Graphics (Proceedings of SIGGRAPH 84)* (July 1984), vol. 18, pp. 213–222.

[GV96] Golub G. H., Van Loan C. F.: *Matrix Computations, Third Edition.* The Johns Hopkins University Press, Baltimore, 1996.

[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* (July 2003), pp. 102–111.

[GWWH03] Goodnight N., Wang R., Woolley C., Humphreys G.: Interactive time-dependent tone mapping using programmable graphics hardware. In *Eurographics Symposium on Rendering: 14th Eurographics Workshop on Rendering* (June 2003), pp. 26–37.

[Hac05] Hachisuka T.: High-quality global illumination rendering using rasterization. In *GPU*

Gems 2, Pharr M., (Ed.). Addison Wesley, Mar. 2005, ch. 38, pp. 615–633.

[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*, Fernando R., (Ed.). Addison Wesley, Mar. 2004, pp. 637–665.

[Har05a] HARRIS M.: Mapping computational concepts to GPUs. In *GPU Gems 2*, Pharr M., (Ed.). Addison Wesley, Mar. 2005, ch. 31, pp. 493–508.

[Har05b] HARRIS M.: NVIDIA fluid code sample. `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*, Pharr M., (Ed.). Addison Wesley, Mar. 2005, ch. 34, pp. 547–555.

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

[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 99* (Aug. 1999), Computer Graphics Proceedings, Annual Conference Series, pp. 277–286.

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

[HE99a] HOPF M., ERTL T.: Accelerating 3D convolution using graphics hardware. In *IEEE Visualization '99* (Oct. 1999), pp. 471–474.

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

[Hei91] HEIDMANN T.: Real shadows real time. *IRIS Universe*, 18 (Nov. 1991), 28–31.

[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 Transactions on Graphics 21*, 3 (July 2002), 693–702.

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

[HK93] HANRAHAN P., KRUEGER W.: Reflection from layered surfaces due to subsurface scattering. In *Proceedings of SIGGRAPH 93* (Aug. 1993), Computer Graphics Proceedings, Annual Conference Series, pp. 165–174.

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

[Hor05a] HORN D.: libgpufft. `http://sourceforge.net/projects/gpufft/`, 2005.

[Hor05b] HORN D.: Stream reduction operations for GPGPU applications. In *GPU Gems 2*, Pharr M., (Ed.). Addison Wesley, Mar. 2005, ch. 36, pp. 573–589.

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

[HS99] HEIDRICH W., SEIDEL H.-P.: Realistic, hardware-accelerated shading and lighting. In *Proceedings of SIGGRAPH 99* (Aug. 1999), Computer Graphics Proceedings, Annual Conference Series, pp. 171–178.

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

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

[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* (Apr. 1999), pp. 127–134.

[HZLM01] HOFF III K. E., ZAFERAKIS A., LIN M. C., MANOCHA D.: Fast and simple 2D geomet-

John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. "A Survey of General-Purpose Computation on Graphics Hardware." In *Eurographics 2005, State of the Art Reports*, August 2005, pp. 21-51.

ric proximity queries using graphics hardware. In *2001 ACM Symposium on Interactive 3D Graphics* (Mar. 2001), pp. 145–148.

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

[Jah05] Jahshaka: Jahshaka image processing toolkit. `http://www.jahshaka.com/`, 2005.

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

[Jam01b] James G.: NVIDIA water surface simulation sample. `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*, Deloura M., (Ed.). Charles River Media, 2001, pp. 497–509.

[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 Visualization 2001* (Oct. 2001), pp. 53–60.

[Jen96] Jensen H. W.: Global illumination using photon maps. In *Eurographics Rendering Workshop 1996* (June 1996), pp. 21–30.

[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)* (Sept. 2005). To appear.

[JvHK04] 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* (Nov. 2004), pp. 395–403.

[KBR04] Kessenich J., Baldwin D., Rost R.: The OpenGL Shading Language version 1.10.59. `http://www.opengl.org/documentation/oglsl.html`, Apr. 2004.

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

[KKKW05] Krüger J., Kipfer P., Kondratieva P., Westermann R.: A particle system for interactive visualization of 3D flows. *IEEE Transactions on Visualization and Computer Graphics* (2005). To appear.

[KL03] Kim T., Lin M. C.: Visual simulation of ice crystal growth. In *2003 ACM SIGGRAPH / Eurographics Symposium on Computer Animation* (Aug. 2003), pp. 86–97.

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

[KLRS04] Kolb A., Latta L., Rezk-Salama C.: Hardware-based simulation and collision detection for large particle systems. In *Graphics Hardware 2004* (Aug. 2004), pp. 123–132.

[KP03] Knott D., Pai D. K.: CInDeR: Collision and interference detection in real-time using graphics hardware. In *Graphics Interface* (June 2003), pp. 73–80.

[KSW04] Kipfer P., Segal M., Westermann R.: UberFlow: A GPU-based particle engine. In *Graphics Hardware 2004* (Aug. 2004), pp. 115–122.

[KW03] Krüger J., Westermann R.: Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics 22*, 3 (July 2003), 908–916.

[KW05] Kipfer P., Westermann R.: Improved GPU sorting. In *GPU Gems 2*, Pharr M., (Ed.). Addison Wesley, Mar. 2005, ch. 46, pp. 733–746.

[LC04] Larsen B. D., Christensen N. J.: Simulating photon mapping for real-time applications. In *Rendering Techniques 2004: 15th Eurographics Workshop on Rendering* (June 2004), pp. 123–132.

[LCW03] Lefohn A. E., Cates J. E., Whitaker R. T.: Interactive, GPU-based level sets for 3D brain tumor segmentation. In *Medical Image Computing and Computer Assisted Intervention (MICCAI)* (2003), pp. 564–572.

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

[LFWK05] Li W., Fan Z., Wei X., Kaufman A.: GPU-based flow simulation with complex boundaries. In *GPU Gems 2*, Pharr M., (Ed.). Addison Wesley, Mar. 2005, ch. 47, pp. 747–764.

[LHN05] Lefebvre S., Hornus S., Neyret F.: Octree textures on the GPU. In *GPU Gems 2*, Pharr M., (Ed.). Addison Wesley, Mar. 2005, ch. 37, pp. 595–613.

[LHPL87] Levinthal A., Hanrahan P., Paquette M., Lawson J.: Parallel computers for graphics applications. *ACM SIGOPS Operating Systems Review 21*, 4 (Oct. 1987), 193–198.

[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 Visualization 2003* (Oct. 2003), pp. 75–82.

[LKHW04] Lefohn A. E., Kniss J. M., Hansen C. D., Whitaker R. T.: A streaming narrow-band algorithm: Interactive computation and visualization of level-set surfaces. *IEEE Transactions on Visualization and Computer Graphics 10*, 4 (July/Aug. 2004), 422–433.

[LKM01] Lindholm E., Kilgard M. J., Moreton H.: A user-programmable vertex engine. In *Proceedings of ACM SIGGRAPH 2001* (Aug. 2001), Computer Graphics Proceedings, Annual Conference Series, pp. 149–158.

[LKO05] Lefohn A., Kniss J., Owens J.: Implementing efficient parallel data structures on GPUs. In *GPU Gems 2*, Pharr M., (Ed.). Addison Wesley, Mar. 2005, ch. 33, pp. 521–545.

[LKS*05] Lefohn A. E., Kniss J., Strzodka R., Sengupta S., Owens J. D.: Glift: Generic, efficient, random-access GPU data structures. *ACM Transactions on Graphics* (2005). To appear.

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

[LM01] Larsen E. S., McAllister D.: Fast matrix multiplies using graphics hardware. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing* (New York, NY, USA, 2001), ACM Press, p. 55.

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

[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 90)* (Aug. 1990), vol. 24, pp. 327–335.

[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* (Aug. 2005). To appear.

[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* (2003), vol. 19, pp. 444–456.

[MA03] Moreland K., Angel E.: The FFT on a GPU. In *Graphics Hardware 2003* (July 2003), pp. 112–119. `http://www.cs.unm.edu/~kmorel/documents/fftgpu/`.

[Man03] Manocha D.: Interactive geometric and scientific computations using graphics hardware. *ACM SIGGRAPH Course Notes*, 11 (2003).

[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 Transactions on Graphics 22*, 3 (July 2003), 896–907.

[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 Visualization 2004* (Oct. 2004), pp. 171–178.

[Mic05a] Microsoft high-level shading language. `http://msdn.microsoft.com/library/default.asp?url=`

/library/en-us/directx9_c/directx/graphics/reference/hlslreference/hlslreference.asp, 2005.

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

[MM02] Ma V. C. H., McCool M. D.: Low latency photon mapping using block hashing. In *Graphics Hardware 2002* (Sept. 2002), pp. 89–98.

[MOK95] Myszkowski K., Okunev O. G., Kunii T. L.: Fast collision detection between complex solids using rasterizing graphics hardware. *The Visual Computer 11*, 9 (1995), 497–512.

[Mor02] Moravánszky A.: Dense matrix algebra on the GPU. In *Direct3D ShaderX2*, Engel W. F., (Ed.). Wordware Publishing, 2002.

[MTP*04] McCool M., Toit S. D., Popa T., Chan B., Moule K.: Shader algebra. *ACM Transactions on Graphics 23*, 3 (Aug. 2004), 787–795.

[NHP04] Nyland L., Harris M., Prins J.: N-body simulations on a GPU. In *Proceedings of the ACM Workshop on General-Purpose Computation on Graphics Processors* (Aug. 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 98* (July 1998), Computer Graphics Proceedings, Annual Conference Series, pp. 159–168.

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

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

[Ope05] OpenVIDIA: GPU-accelerated computer vision library. http://openvidia.sourceforge.net/, 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*, Pharr M., (Ed.). Addison Wesley, Mar. 2005, ch. 29, pp. 457–470.

[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* (Feb. 2005), pp. 184–186.

[PBMH02] Purcell T. J., Buck I., Mark W. R., Hanrahan P.: Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics 21*, 3 (July 2002), 703–712.

[PDC*03] Purcell T. J., Donner C., Cammarano M., Jensen H. W., Hanrahan P.: Photon mapping on programmable graphics hardware. In *Graphics Hardware 2003* (July 2003), pp. 41–50.

[PH89] Potmesil M., Hoffert E. M.: The Pixel Machine: A parallel image computer. In *Computer Graphics (Proceedings of SIGGRAPH 89)* (July 1989), vol. 23, pp. 69–78.

[POAU00] Peercy M. S., Olano M., Airey J., Ungar P. J.: Interactive multi-pass programmable shading. In *Proceedings of ACM SIGGRAPH 2000* (July 2000), Computer Graphics Proceedings, Annual Conference Series, pp. 425–432.

[PS03] Purcell T. J., Sen P.: Shadesmith fragment program debugger. http://graphics.stanford.edu/projects/shadesmith/, 2003.

[Pur04] Purcell T. J.: *Ray Tracing on a Stream Processor*. PhD thesis, Stanford University, Mar. 2004.

[RMS92] Rossignac J., Megahed A., Schneider B.-O.: Interactive inspection of solids: Cross-sections and interferences. In *Computer Graphics (Proceedings of SIGGRAPH 92)* (July 1992), vol. 26, pp. 353–360.

[RR86] Rossignac J. R., Requicha A. A. G.: Depth-buffering display techniques for constructive solid geometry. *IEEE Computer Graphics & Applications 6*, 9 (Sept. 1986), 29–39.

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

[RS01b] Rumpf M., Strzodka R.: Nonlinear diffusion in graphics hardware. In *Proceedings of EG/IEEE TCVG Symposium on Visualization (VisSym '01)* (2001), Springer, pp. 75–84.

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

[RS05] Rumpf M., Strzodka R.: Graphics processor units: New prospects for parallel computing. In *Numerical Solution of Partial Differential Equations on Parallel Computers*. Springer, 2005. To appear.

[RSSF02] Reinhard E., Stark M., Shirley P., Ferwerda J.: Photographic tone reproduction for digital images. *ACM Transactions on Graphics 21*, 3 (July 2002), 267–276.

[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* (Mar. 1992), vol. 25, pp. 95–100.

[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* (June 2003), pp. 455–466.

[SD02] Stamminger M., Drettakis G.: Perspective shadow maps. *ACM Transactions on Graphics 21*, 3 (July 2002), 557–562.

[SDR03] Strzodka R., Droske M., Rumpf M.: Fast image registration in DX9 graphics hardware. *Journal of Medical Informatics and Technologies 6* (Nov. 2003), 43–49.

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

[Sen04] Sen P.: Silhouette maps for improved texture magnification. In *Graphics Hardware 2004* (Aug. 2004), pp. 65–74.

[SF91] Shinya M., Forgue M. C.: Interference detection through rasterization. *The Journal of Visualization and Computer Animation 2*, 4 (1991), 131–134.

[SG04] Strzodka R., Garbe C.: Real-time motion estimation and visualization on graphics cards. In *IEEE Visualization 2004* (Oct. 2004), pp. 545–552.

[SHN03] Sherbondy A., Houston M., Napel S.: Fast volume segmentation with simultaneous visualization using programmable graphics hardware. In *IEEE Visualization 2003* (Oct. 2003), pp. 171–176.

[SKALP05] Szirmay-Kalos L., Aszódi B., Lazányi I., Premecz M.: Approximate ray-tracing on the GPU with distance imposters. *Computer Graphics Forum 24*, 3 (Sept. 2005). To appear.

[SKv*92] Segal M., Korobkin C., van Widenfelt R., Foran J., Haeberli P.: Fast shadows and lighting effects using texture mapping. In *Computer Graphics (Proceedings of SIGGRAPH 92)* (July 1992), vol. 26, pp. 249–252.

[SL05] Sumanaweera T., Liu D.: Medical image reconstruction with the FFT. In *GPU Gems 2*, Pharr M., (Ed.). Addison Wesley, Mar. 2005, ch. 48, pp. 765–784.

[SLJ98] Stewart N., Leach G., John S.: An improved Z-buffer CSG rendering algorithm. In *1998 SIGGRAPH / Eurographics Workshop on Graphics Hardware* (Aug. 1998), pp. 25–30.

[SOM04] Sud A., Otaduy M. A., Manocha D.: DiFi: Fast 3D distance field computation using graphics hardware. *Computer Graphics Forum 23*, 3 (Sept. 2004), 557–566.

[SPG03] Sigg C., Peikert R., Gross M.: Signed distance transform using graphics hardware. In *IEEE Visualization 2003* (Oct. 2003), pp. 83–90.

[ST04] Strzodka R., Telea A.: Generalized distance transforms and skeletons in graphics hardware. In *Proceedings of EG/IEEE*

TCVG Symposium on Visualization (VisSym '04) (2004), pp. 221–230.

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

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

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

[THO02] 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* (2002), pp. 306–317.

[Tre05] TREBILCO D.: GLIntercept. `http://glintercept.nutty.org/`, 2005.

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

[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 (July 1967), 98–103.

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

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

[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* (2001), pp. 439–446.

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

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

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

[WWHL05] WANG J., WONG T.-T., HENG P.-A., LEUNG C.-S.: Discrete wavelet transform on GPU. `http://www.cse.cuhk.edu.hk/~ttwong/software/dwtgpu/dwtgpu.html`, 2005.

[XM05] XU F., MUELLER K.: Accelerating popular tomographic reconstruction algorithms on commodity PC graphics hardware. *IEEE Transactions on Nuclear Science* (2005). To appear.

[YLPM05] YOON S.-E., LINDSTROM P., PASCUCCI V., MANOCHA D.: Cache-oblivious mesh layouts. *ACM Transactions on Graphics 24*, 3 (Aug. 2005). To appear.

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

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

[Zel05] ZELLER C.: Cloth simulation on the GPU. In *ACM SIGGRAPH 2005 Conference Abstracts and Applications* (Aug. 2005). To appear.

John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. "A Survey of General-Purpose Computation on Graphics Hardware." In *Eurographics 2005, State of the Art Reports*, August 2005, pp. 21-51.