GENERAL PURPOSE PROGRAMMING ON MODERN GRAPHICS HARDWARE

Robert Fleming, B.Sc.

Thesis Prepared for the Degree of

MASTER OF SCIENCE

UNIVERSITY OF NORTH TEXAS

May 2008

Fleming, Robert. General Purpose Programming on Modern Graphics Hardware. Master of Science (Computer Science), May 2008, 90 pp., 1 table, 3 figures, references, 124 titles.

I start with a brief introduction to the graphics processing unit (GPU) as well as general-purpose computation on modern graphics hardware (GPGPU). Next, I explore the motivations for GPGPU programming, and the capabilities of modern GPUs (including advantages and disadvantages). Also, I give the background required for further exploring GPU programming, including the terminology used and the resources available. Finally, I include a comprehensive survey of previous and current GPGPU work, and end with a look at the future of GPU programming.

To Wanda and Cheesepuff, my partners in crime.

TABLE OF CONTENTS

LIST OF TABLES AND ILLUSTRATIONS

CHAPTER 1

MOTIVATION

Early graphics processing units of the 1980s and 1990s were built as 2D accelerators. Common operations included bit blitting, which is a combination of two bit maps using a raster op (for example: AND, OR). Today, GPUs are a cheap commodity solution to having a data-parallel co-processor alongside the CPU. In their evolution as processors, they became fully programmable in three stages of the graphics pipeline (§ 2.2). In addition, they specialize in massively parallel scalar processing; modern GPUs have up to 128 independent processors (NVIDIA 2006), whereas CPUs have, at the desktop level, only reached 8 cores.

GPUs are also outpacing CPUs in terms of raw processing horsepower. At the time of this writing, Nvidia's G80 is capable of a theoretical peak performance of 340 Gflops, compared to the almost 40 Gflops of Intel's Core 2 Duo processor (NVIDIA 2008, *NVIDIA CUDA*). GPU pixel processing is increasing at a rate of 1.7 times per year, and vertex processing is increasing at a rate of 2.3 times per year, contrasted with a 1.4 multiplier for CPU performance (Owens et al. 2005).

Recently, there has been an increasing interest in *general purpose* computation on graphics hardware. The ability to work independently alongside the CPU as a co-processor is interesting but not motivating enough to learn how to apply problems to the graphics domain. However, with full programmability and the computational power of GPUs outpacing CPUs (in terms of a price/performance ratio), the GPU has moved to a place of being the primary processing unit for certain applications, with the CPU managing data and direction of the GPU. This is motivation enough to learn how to apply certain problems to the domain of computer graphics, thus we achieve general purpose computation on graphics hardware.

1.1 What Kinds of Computation Suit the GPU?

GPUs were built primarily as graphics processing units, and so we know that they excel at computer graphics. To see what kinds of problems can be mapped well to GPUs, we simply abstract computer graphics computation. Two key ideas in graphics processing are data parallelism and independence. That is, GPUs efficiently operate on streams of data where different elements of the stream are processed simultaneously (parallelism). Also, each piece of data in the stream typically has little or no dependence upon other data elements in the stream. Any problem that benefits from parallelization and has mostly independent data would map well to GPU hardware.

We also know that GPUs operate almost exclusively on vectors of floating point numbers. Any arithmetically intense problems would do well to take advantage of the fact that the cost of GPU computation is decreasing more rapidly than the cost of communication (memory bandwidth, caches) (Harris 2005). A good example of this is finding the solution of a system of linear equations (Krüger and Westermann 2005). Solving systems of linear equations maps well to GPU hardware because the ratio of arithmetic operations to words transferred (where a word is the word-size of the architecture; i.e. typically 32 bits) is very high.

1.2 GPU Limitations

Not all computation maps well to GPU programming, however. Until very recently, GPUs lacked integer operations such as bit-shifts and bitwise logical operations (AND, XOR). As a result they are not well-suited to fixed-point computation (though this is typically not a downside as floating-point computation is available) and other integer-heavy arithmetic. GPUs also lack efficient branching operations; only recently are they capable of branching in the fragment processor and previously the vertex processor. Typically, GPUs evaluate both sides of the branch and discard one (Owens et al 2005), whereas CPUs historically excel at branch prediction.

Caches are typically small on GPUs, and memory-intensive computation is not well-suited for mapping onto graphics hardware. While memory bandwidth may be high, GPU vertex processors can only cache approximately 24 vertices[1], in constrast with the 1-2 megabyte caches of newer CPUs. This is not to say GPUs don't access large pieces of data in memory, but they typically don't read/write to memory in frequent iterations (in fact, typical GPU programs read several times and write only once at the end). The Nvidia Geforce 8800 GTX has 768 MB of on-card RAM and a peak memory bandwidth of 86.4 GBps (NVIDIA 2006), clearly demonstrating apt memory computation ability, but due to the structure of GPU programming, it is not suited for fast, successive read/writes.

GPUs are further limited in floating point precision, with only the most recent GPUs having full 32-bit IEEE 754 single-precision floats, whereas historically they have done computation in 16 or 24-bit precision. Double-precision 64-bit floats are rumored for an upcoming revision of Nvidia's Tesla line of GPUs, which will be built specifically to address GPGPU (NVIDIA 2007). There is no indication of double-precision floats reaching commodity graphics hardware any time soon. The current lack of double-precision floats limits the adoption of GPU programming for many areas of scientific computing and numerical computation.

Lastly, GPU programming is limited in that the problem of interest must be mapped to the graphics programming model, with only a few abstractions available to programmers familiar with the CPU model. Achieving good performance in GPU programming requires in-depth knowledge of not only graphics programming, but also modern hardware trends. This further limits the adoption of general purpose GPU programming as new problem domains usually require an expert in the domain as well as an expert in graphics programming.

1.3 What This Document Contains

This thesis introduces and covers all introductory aspects of general-purpose computation on modern graphics hardware and provide instructions on how to further obtain information on any specific or general sub-domain thereof. Chapter 1 provides the motivation and impetus for GPGPU, followed by Chapter 2, which covers the description and methodology for general-purpose computation on GPUs. Chapter 3 is a comprehensive look at all current and previous general-purpose work done on GPUs, with an emphasis on current work descriptions. Chapter 4 analyzes the future of the industry.

CHAPTER 2

BACKGROUND MATERIAL

As an introduction to general-purpose computation on graphics hardware, this section provides the necessary prerequisites. Section 2.1 is a quick history lesson, followed by Section 2.2, which details the traditional graphics pipeline. Section 2.3 introduces the GPU programming model. Section 2.4 is a current look at all the tools available for GPGPU programming. Finally, section 2.5 details an example GPGPU program.

2.1 A Brief History of GPU Programmability

The Nvidia Geforce 256 was the first card to offload vertex transform and lighting (T&L) from the CPU and onto the GPU[2]. This generation of card also introduced a programmable method of combining textures and colors by way of such OpenGL extensions as EXT_texture_env_combine and NV_register_combiners. While this introduced greater flexibility than previous generations, it was not considered truly programmable, because the number of combinations remained limited.

The first true programmability arrived with the DirectX 8 generation of graphics cards. Cards in this generation include the Geforce 3, ATI Radeon 8500, and the Geforce 4. On the OpenGL side of things, the programmability was first exposed via vendor-specific extensions such as NV_vertex_program, but it was later available via the multi-vendor ARB_vertex_program. This extension allowed full control over the vertex transform stage of the graphics pipeline (§ 2.2). Around this time, the term *shader* became prevalent as a way of denoting a program that runs on the graphics card. A pixel shader (or fragment program, in OpenGL) is a program that runs on each individual fragment processed, and a vertex shader (or vertex program) runs on each vertex processed.

From this generation on, the programmability of GPUs was denoted with what shader model they were capable of. The shader model is closely tied with what DirectX version the GPU is capable of rendering in full hardware mode (without CPU assistance), as shown in Table 2.1. Typically, with each generation of GPUs the two main independent hardware vendors (IHV), ATI and Nvidia, would release OpenGL extensions exposing the same functionality as DirectX.

With each shader model release, the programmability was typically increased by several factors. For example, the maximum number of instruction slots changed. In pixel shader 1.1 the maximum number of texture fetch instructions was 4, and the maximum number of arithmetic instructions was 8. In pixel shader 2.0 this was increased to 32 texture fetches and 64 arithmetic operations. Pixel shader 3.0 increased the instruction slots to a minimum of 512, and with loops and branching taken into account, the number of executed instructions can be as high as 2^16. Pixel shader 4.0 increased the available instruction slots to 2^16, with the executed instructions effectively unlimited. Each generation was a huge improvement over the previous generation. Also, new available registers and new instructions became available with every shader model release. For instance, vertex shader 3.0 introduced the ability to do texture fetches in the vertex shader, which enabled a whole range of new possibilities in GPGPU as memory fetches so early in the pipeline were previously impossible.

With the introduction of the Geforce 8 and shader model 4.0, the programmability of GPUs reached a whole new level with what is known as the unified shader model. Under shader model 4.0, all three shader types, pixel, vertex, geometry, (§ 2.2) enjoy the same instruction set. This is possible with the new scalar architecture of shader model 4.0 hardware, where there is no longer specialized processors for shading pixels and other processors for transforming vertices. Instead, there are general purpose processors that can be scheduled to run any type of shading program. In the past, when rendering a geometry-heavy scene, the pixel processors were mostly idle; likewise when rendering a heavily post-processed scene, the vertex processors were underutilized. Under the unified architecture, all resources can be used regardless of the type of scene. This is discussed extensively in (NVIDIA 2006). Shader model 4.0 also introduced geometry shaders, a previously nonprogrammable stage of the graphics pipeline, as well as true integer computation support on GPUs (including bitwise operations).

DirectX Version	Vertex Shader Version	Pixel Shader Version	Geometry Shader Version
8.0	1.0[3], 1.1	1.0, 1.1	n/a
8.1	1.2, 1.3, 1.4	1.0, 1.1	n/a
9.0	2.0	2.0	n/a
9.0a	2a, 2b	2.x	n/a
9.0c	3.0	3.0	n/a
10.0	4.0	4.0	4.0
10.1	4.1	4.1	4.1

Table 2.1: How Shader Models Relate to DirectX Versions

2.2 The Graphics Pipeline

There are many resources available that explain the graphics rendering pipeline in more detail, such as (Akenine-Möller and Haines 2002) and (Shreiner et all 2007), but a brief explanation is included here for completeness. The following describes the graphics pipeline as implemented in the OpenGL API. Illustration 2.1 should be used as a guide when reading this section.

First, vertices are specified by the application, either with conventional attributes such as position, color, and texture coordinates or by generic numbered attributes. Next, they are transformed by either the fixed-function pipeline or by a vertex shader. Modern graphics drivers usually implement the fixed-function pipeline via a generated vertex shader. The fixed-function vertex transformation includes multiplying the vertex position by the modelview, and projection matrices, as well as texture coordinate generation and lighting calculations. A custom vertex shader has the option to do all or none of these calculations.

The next step is primitive assembly, where vertices are assembled into primitives specified by the application, such as triangles, points, or quads. If a geometry shader is specified, it is executed once on each input primitive. The geometry shader is specified with an output primitive type, which may or may not be different than the input primitives. The geometry shader can emit new vertices, which are then assembled into the output primitive type.

Optionally, the vertex colors are clamped to the range [0, 1], and flat shading can be applied. Flat shading applies just one color to the entire primitive, as opposed to smooth shading which interpolates the colors between each vertex.

Vertices are then clipped to the view volume in clip coordinates, and then perspective division converts them to normalized device coordinates. The viewport transformation step then converts vertices to window coordinates. Final color processing refers to converting color values to fixed-point values. See the OpenGL specification § 2.14.9 for further details (Segal and Akeley 2006). The primitive is determined to be either front or back facing, which determines whether the front or back color of a vertex is chosen.

Rasterization converts the primitive into a two-dimensional image, the output of which is a fragment and an associated depth value. The fragment can be thought of as a 'potential pixel,' as technically a pixel value is what gets written to the framebuffer, and a fragment may be killed before it reaches the framebuffer. A pixel shader, or fragment shader, is executed for each fragment generated. The shader may change any value associated with that fragment, such as color or depth, except the final position in the framebuffer is already determined at this point in the pipeline (final fragment position may be manipulated earlier in the pipeline indirectly such as via a vertex shader). Some final fragment operations such as the depth test, alpha test, stencil test, and blending are done before the fragment is finally written to the framebuffer. Also, it is worth noting that the "framebuffer" may or may not be the actual backbuffer (or frontbuffer, i.e. the visible window) of the application, and indeed, many GPGPU programs are run without rendering an image to the backbuffer. With modern hardware, it is possible to render directly to texture memory, the results of which may be used in subsequent rendering passes. This concept is critical to GPGPU programming.

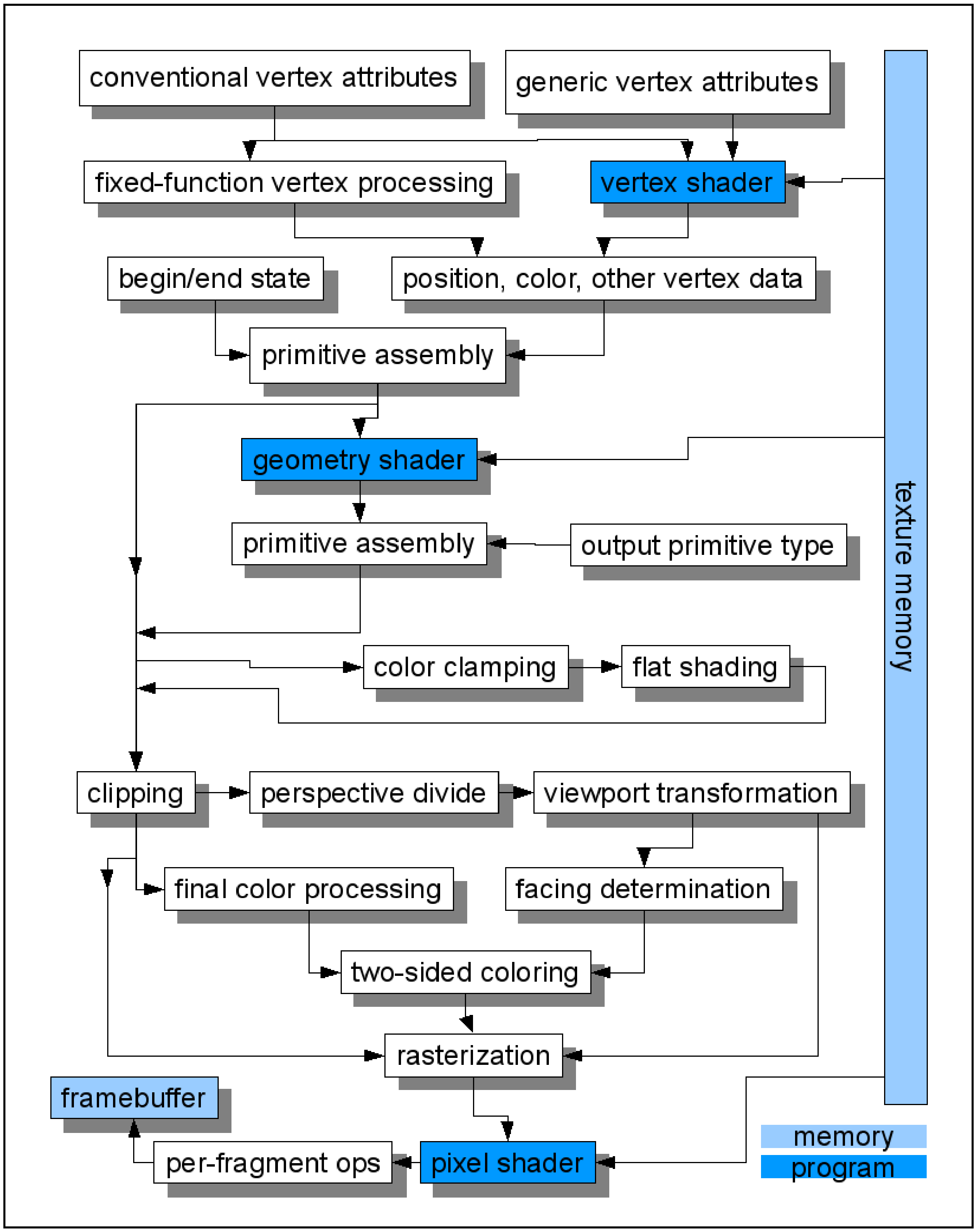

Illustration 2.1: The OpenGL Graphics Rendering Pipeline

2.3 The Programming Model

The GPU programming model is a subset of a broader field called *stream processing*. In stream processing, a set of input data (called a stream) is applied to a series of operations (called a kernel), once for each element in the stream. The central idea is that each element in the stream is independent from every other element, allowing each element to be processed in parallel. When only one kernel at a time is applied to the entire stream, it is called uniform streaming, which is a form of single-instruction-multiple-data (SIMD), as opposed to multiple-instruction-multiple-data (MIMD), or single-instruction-single-data (SISD), the latter of which is the execution model of a typical single-core CPU. Modern CPUs have vector processors that utilize the streaming SIMD extensions (SSE) instruction set, and are capable of, for example, multiplying four floating point numbers against another four floating point numbers in a single instruction. While this is SIMD, stream processing is the same concept at a much larger level, with typical streams being millions of elements large, and kernels consisting not just of a single multiply, but possibly hundreds of operations. For this reason, GPU programming is sometimes called single-program-multiple-data, or SPMD.

Stream processing is a data-parallel methodology, as opposed to a task-parallel methodology. An example of task parallelism is the CPU threading model: each task is broken up into a thread of execution which may have completely different instructions than any other thread, as well as possibly different data. In stream processing, the instructions for each element of the stream are fixed (branching and looping aside); only the data is different. Unlike task parallelism, which has the overhead of synchronization issues to deal with (race conditions, locks, etc.), it is easy to demonstrate that the effectiveness of a data-parallel device like the GPU can be increased simply by adding more stream processing elements.

There are two important tenets in stream processing. First, compute intensity is the number of arithmetic operations per memory access. Second, data parallelism means that each stream element can be processed independent of any other. A high level of data parallelism and compute intensity both indicate a good match for GPU programming, because, as mentioned earlier, the cost of computation is decreasing more rapidly than the cost of communication.

On GPUs, input streams typically mean a set of vertices sent by the application. Output streams are usually 2D textures, or the framebuffer itself, which can be thought of as a 2D texture. Kernels can mean either the vertex shader, geometry shader, or pixel shader, depending on the desired number of output stream elements. The typical GPGPU setup involves drawing a fullscreen quad such that the number of pixels in the quad equals the number of desired times to execute the kernel body, which in this case is implemented via a pixel shader. Optionally rendering the result into a texture to save the results for future computations, it is possible to manipulate the viewport to ensure a one-to-one mapping of output pixels to texels. Alternatively, the quad could be rendered to the framebuffer and then copied to a texture, for architectures that do not support render-to-texture (RTT). Regardless of the destination, texture or framebuffer, it is essential to realize that, due to the structure of the graphics pipeline, the output memory is write-only, and all other texture memory is read-only. This has led to a technique called *ping-ponging*, where the result of one computation is used as the input for the next computational pass, with the source and destination textures swapping at each iteration.

2.3.1 Stream Processing Technology

There are a few common techniques in stream processing, the simplest of which is called a *map* operation. A map simply applies the kernel function to each element in the stream. As before, using a pixel shader as the kernel, a pixel is generated for each element in the stream, producing the output stream.

A *reduce* operation produces a smaller output stream from a larger set. Typically, the larger set is input for the reduction kernel, where multiple elements are read, but only one element is written. This is usually implemented via a pixel shader that reads from multiple textures before writing a single value. A reduce operation can be repeated to condense a stream down to a desired size, potentially down to a single stream element.

The two methods of stream communication are *scatter* and *gather*. While the cost of communication is high compared to the cost of computation, not all algorithms are possible with pure computation, and sometimes communication is necessary. Gather is basically a random-access read, which in our case is typically sampling a texture. Gather was previously only possible in the fragment processor, but in the most recent GPU architectures it is available in all programmable stages of the graphics pipeline (§ 2.2). Scatter is a random-access write into memory. The fragment processor is incapable of scatter, as the location in memory of the output pixel is fixed at the time of the shader's execution. To implement scatter on the GPU, a vertex or geometry shader is written to manipulate the vertex's position, or generate new vertices on-the-fly.

2.4 Languages

The most important advancement towards GPGPU thus far has been the advent of high-level shading languages. Previously, shaders had to be written in assembly that, while standardized, still changed with each shading model iteration. High-level languages enabled more complex algorithms to be expressed in easier-to-understand syntax, just as in traditional CPU programming languages. The three primary high-level shading languages are the OpenGL Shading Language (GLSL), Nvidia's C for Graphics (Cg), and Microsoft's High Level Shading Language (HLSL). GLSL is OpenGL specific, and HLSL is Direct3D specific, but Nvidia's Cg can be used with either API. All three have slight syntactic differences, but with an understanding of one language, it is relatively easy to transition to another shading language, assuming an existing understanding of the API it is used in (either D3D or OGL). Indeed, Cg and HLSL were developed by close collaboration between Nvidia and Microsoft, and the languages remain nearly identical.

Emerging from the high-level shading languages, several efforts have been made to program GPUs without the need to learn a graphics API, but instead using traditional C or C++. While still in its infancy, these efforts, usually in the form of a library rather than a new language (to take advantage of existing tools and technologies), have come a long way towards making GPGPU programming more available and approachable. First, the IHV specific solutions are Nvidia's Compute Unified Device

Architecture (CUDA) (NVIDIA 2008, *NVIDIA CUDA*), and ATI/AMD's Close-to-Metal (CTM) (Advanced Micro Devices 2006). ATI is also releasing an evolution of CTM called Compute Abstraction Layer (CAL), which promises to be higher-level than CTM, which is a very low-level interface to the GPU. On the open-source side of things, there is BrookGPU (Standford University Graphics Lab 2008), or simply Brook, which is a combination of the Brook stream programming language, and the Brook runtime library. Programs written in the Brook language (which is very similar to ANSI C) can be compiled down to run on a number of backends, including Direct3D, OpenGL, and CTM. The Brook compiler takes Brook language files (*.br) and compiles them into C++, which (after compiled) then calls into the Brook runtime. Another open-source project is LibSh (McCool and DuToit 2004), simply Sh, which embeds itself inside C++. Instead of compiling a stream programming language into C++, the GPU programs in Sh are simply C++ programs. These programs make calls into the Sh runtime, which then dynamically takes the Sh operations and compiles them down first into an intermediate language, then finally into GPU code that the Sh runtime handles transparently in the backend. Both Brook and Sh are exposed as C or C++ libraries that provide an abstraction to the GPU-specific languages. Brook has, until most recently, been an inactive project for several years, and LibSh was abandoned as of August 2006. Most development in GPGPU/stream programming languages (as opposed to GPU languages, such as Cg/GLSL) has been in vendor-specific solutions, such as CUDA and CTM, that are nevertheless available for free to encourage development. Commercial stream programming languages remain few, with PeakStream (a spinoff of Brook) acquired by Google in June 2007, and Rapidmind (RapidMind 2008, *RapidMind*

Homepage), a commercialization of LibSh. Rapidmind takes the idea of Sh even further, with the backend runtime capable of executing on not only GPUs but multi-core CPUs as well as STI's (an alliance of Sony, Toshiba, and IBM) Cell architecture. Lastly, it is worth noting other stream programming languages such as MIT's StreamIt and projects such as Intel's Ct that are built mainly for multi-core CPUs and do not seem to take advantage of the extreme parallelism available in GPUs.

2.5 Debuggers, Profilers, and Authoring Tools

While stream programming languages are still developing, high-level shading languages have been available and production-ready for years. What limits their adoption, however, is not only the need to utilize a graphics API alongside the shading language, to take advantage of the GPU, but also the availability of tools such as debuggers and profilers.

Microsoft provides PIX, which allows profiling, optimizing, and debugging Direct3D code. Unfortunately, when debugging vertex and pixel shaders, the code is executed on the Direct3D reference rasterizer, which executes on the CPU. This is not only slower but not the exact environment the shader code will eventually run under, and relies instead on the quality of the reference rasterizer.

On the OpenGL side of things, there are many projects designed to trace and log OpenGL calls, and generally intercept rendering commands and capture OpenGL state. Examples include BunGLe, Graphic Remedy's gDEBugger, GLIntercept, spyGLass, and Nvidia's GLExpert. While some of these, such as gDEBugger, allow setting breakpoints and debugging, none of them allow debugging shaders, which is of primary interest to

GPGPU programming. Two projects, Shadesmith and glslDevil (Strengert, Klein, and Ertl 2007), allow debugging of shaders, and both execute on the target hardware instead of software emulation. Shadesmith, however, only debugs fragment programs. GlslDevil retrieves information directly from the hardware pipeline, and does so transparently to the application (in other words, it does not require recompilation or relinking of the executable being debugged). In addition, it supports geometry shaders as well as vertex and fragment shaders.

Profiling information is readily available from IHV's, with Nvidia's PerfHUD/NVPerfGraph, and ATI's GPU PerfStudio. NV's PerfHUD only supports Direct3D, while ATI's PerfStudio supports both OpenGL and Direct3D. There is also Nvidia's ShaderPerf and ATI's GPU ShaderAnalyzer for profiling shaders specifically, with the former also only supporting D3D and the latter supporting both API's. OpenGL profiling information can be acquired through some of the previously mentioned OpenGL debuggers, such as gDEBugger.

Tools written specifically to help author shaders include Apple's OpenGL Shader Builder, TyphoonLab's OpenGL Shader Designer, Nvidia's FX Composer, and ATI's Rendermonkey. These tools provide a closed environment where shaders can be applied to various geometries without the need to worry about the external API calls (such as OpenGL or Direct3D), allowing quick turn-around between editing a shader and seeing it's effects on the output. However, in the context of GPGPU programming, these tools are not ideal, as many GPGPU programs aren't necessarily interested in rendering output, and many require complicated setup aside from just rendering geometry with textures.

While the number of debuggers and profilers may seem numerous, many have shortcomings that show the immaturity of GPU and GPGPU programming tools. For instance, many profilers aren't available for both graphics API's, and many debuggers are not capable of debugging shaders, or are not capable of debugging on the actual hardware. In addition, the relative infancy of GPGPU and stream programming languages means that existing debuggers and profilers can not yet be taken advantage of. It is expected this will change rapidly with time, with the eventual sophistication of GPU programming tools reaching that of CPU programming tools.

2.6 Matrix Multiplication Example

As an example of GPGPU programming, we explain how to do matrix multiplication on the GPU using two sample implementations: one using the OpenGL shading language, GLSL, and another using Nvidia's CUDA. We define the product of two matrices **A** and **B**, where **A** is an *m* by *n* matrix, and **B** is a *n* by *p* matrix, to be **C = A * B**, where **C** is an *m* by *p* matrix. The entry $\boldsymbol{c_{i,j}}$ denotes the value of **C** at position (*i, j*), where

$$C_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j} = a_{i,1} b_{1,j} + a_{i,2} b_{2,j} + \ldots + a_{i,n} b_{n,j}$$

for each pair *i, j* where $0 \leq i \leq m$ and $0 \leq j \leq p$. To simplify things, we assume square matrices of size [*dim*, *dim*], and therefore $m = n = p = dim$. Consider the following C++ code for computing **C** on the CPU:

```
void matrixMul_cpu(float* C, float* A, float* B, const int dim) {
    for (int i = 0; i < dim; ++i) {
        for (int j = 0; j < dim; ++j) {
            float sum = 0;
            for (int k = 0; k < dim; ++k) {
                sum += A[i * dim + k] * B[k * dim + j];
            }
            C[i * dim + j] = sum;
        }
    }
}
```

Here we store **A**, **B**, and **C** in one-dimensional arrays where the element $c_{y,x}$ is accessed via **C**[*y* * *dim* + *x*]. Now we demonstrate how to perform this same computation using the GPU.

2.6.1 GLSL Implementation

The prerequisite before any OpenGL commands can be passed to the GPU is, first, a valid OpenGL context must be acquired. This is window-system specific, but many libraries exist to program this in a cross-platform way; for our purposes we used the GLFW library (GLFW Project 2008). The full source code for our GLSL sample, including initialization code and any functions not explicitly defined here, can be found in Appendix A. Our GLSL implementation will compute **C** on the GPU by storing **A**, **B**, and **C** as textures, and then render to **C** with a GLSL fragment shader that computes each element $c_{i,j}$. We first define our matrix dimensions and then allocate space on the GPU for **A**, **B**, and **C** as textures:

```
const int dim = 7;
GLuint A_gpu_data = createTexture(dim);
GLuint B_gpu_data = createTexture(dim);
GLuint C_gpu_data = createTexture(dim);
```

The function **createTexture()** returns a single-channel[4], floating-point texture of size [*dim*, *dim*] that is suitable for storing our matrices. We use the variable **C_gpu_data** to refer to **C** in texture memory, and the variable **C_cpu_data** to refer to **C** in main memory. Next, we allocate space on the CPU for our matrices and randomly assign values to **A** and **B**:

```
float* A_cpu_data = (float*) malloc(dim * dim * sizeof(float));
float* B_cpu_data = (float*) malloc(dim * dim * sizeof(float));
float* C_cpu_data = (float*) malloc(dim * dim * sizeof(float));
random_fill(A_cpu_data, dim * dim);
random_fill(B_cpu_data, dim * dim);
```

We next transfer the data of **A** and **B** from main memory to texture memory:

```
glBindTexture(GL_TEXTURE_RECTANGLE_ARB, A_gpu_data);
glTexSubImage2D(GL_TEXTURE_RECTANGLE_ARB, 0, 0, 0, dim, dim, GL_LUMINANCE,
    GL_FLOAT, A_cpu_data);
glBindTexture(GL_TEXTURE_RECTANGLE_ARB, B_gpu_data);
glTexSubImage2D(GL_TEXTURE_RECTANGLE_ARB, 0, 0, 0, dim, dim, GL_LUMINANCE,
    GL_FLOAT, B_cpu_data);
```

The zeros in the function **glTexSubImage2D()** refer to mipmap levels and x-, y-offsets within the array, which we would set if we did not want to transfer the entire array. For more details about this and any other OpenGL function refer to the OpenGL man pages (Khronos Group 2007). Note that **A_cpu_data** and **B_cpu_data** are no longer needed, and can be released using **free()** should we choose to. In order to render to the texture **C_gpu_data** we need to create a framebuffer object, which can be thought of as an off-screen window, and attach our texture as its main color buffer:

```
GLuint fbo = 0;
glGenFramebuffersEXT(1, &fbo);
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fbo);
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT0_EXT,
    GL_TEXTURE_RECTANGLE_ARB, C_gpu_data, 0);
```

In the OpenGL shading language, shaders are created and attached to *program objects*, which can be thought of as containers for shader programs (for example, holding one of each of a vertex shader, and a fragment shader). In our case, we need only a single fragment shader, and opt to use the fixed-function vertex processing. Shaders are passed to OpenGL as text strings, which can be read from disk, or included in the source directly. We store our fragment shader source in the string **fShaderSource**, which we will discuss later.

```
const char* fShaderSource = //discussed later
GLuint fShader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fShader, 1, &fShaderSource, NULL);
glCompileShader(fShader);
GLuint program = glCreateProgram();
glAttachShader(program, fShader);
glLinkProgram(program);
```

A key idea in GPGPU computing when rendering to textures is to spawn the fragment shader once for each destination element; in our case, once for each texel of **C**. Thus, since a fragment shader is executed once for each input pixel, we must generate one pixel per texel of **C**. The following code creates this 1:1 mapping:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0.0, dim, 0.0, dim);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glViewport(0, 0, dim, dim);
```

Next, we must tell OpenGL where we are rendering to (the framebuffer object we attached **C** to), what textures we would like access to while rendering, and what program object (that contains our fragment shader) we would like to be active. We also pass some information to our fragment shader; namely, *dim* and the locations of **A** and **B**, by specifying them as uniform parameters. In GLSL, uniform parameters are variables which do not change per primitive; in our case, they never change.

```
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fbo);
glDrawBuffer(GL_COLOR_ATTACHMENT0_EXT);
glClear(GL_COLOR_BUFFER_BIT);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_RECTANGLE_ARB, A_gpu_data);
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_RECTANGLE_ARB, B_gpu_data);
glUseProgram(program);
glUniform1i(glGetUniformLocation(program, "A"), 0);
glUniform1i(glGetUniformLocation(program, "B"), 1);
glUniform1i(glGetUniformLocation(program, "dim"), dim);
```

Finally, to invoke our fragment shader once for every texel in **C** we draw a screen-sized quad:

```
glBegin(GL_QUADS);
    glTexCoord2f(0, 0);     glVertex2f(0, 0);
    glTexCoord2f(dim, 0);   glVertex2f(dim, 0);
    glTexCoord2f(dim, dim); glVertex2f(dim, dim);
    glTexCoord2f(0, dim);   glVertex2f(0, dim);
glEnd();
```

Our quad is of size [*dim*, *dim*] and is screen-sized due to the viewport specified above (which defines the size of our OpenGL screen). Note the texture coordinates of the quad are unnormalized and are of the range [0, *dim*]. Normalized texture coordinates are of the range [0, 1]. Now we examine our fragment shader, which we stored in **fShaderSource**, which performs the matrix multiplication:

```
#extension GL_ARB_texture_rectangle : enable
uniform sampler2DRect A;
uniform sampler2DRect B;
uniform int dim;
void main() {
    float sum = 0.0;
    for (int i = 0; i < dim; ++i) {
        float a = texture2DRect(A, vec2(i, gl_TexCoord[0].t)).r;
        float b = texture2DRect(B, vec2(gl_TexCoord[0].s, i)).r;
        sum += a * b;
    }
    gl_FragColor.r = sum;
}
```

We used the OpenGL extension ARB_texture_rectangle to allow our matrices to be any dimension in size; usually, OpenGL texture sizes are required[5] to be a power of two. Another benefit of using texture rectangles is it allows us to sample textures using unnormalized texture coordinates. **A** and **B** are of type **uniform sampler2DRect**, which means they are two-dimensional texture rectangles whose values do not change. They are *sampled* (i.e. read from texture memory) via the function **texture2DRect()**, which takes as parameters the texture name, and, for two-dimensional textures, a floating-point pair (*s*, *t*). The function returns four floating-point numbers in the form of a **vec4** (a GLSL data type), from which we are only interested in the first element (accessed[6] with the **.r**). The **texture2DRect()** function always returns four floats; it is up to the user to access the valid members that are returned. In our case, since **A** and **B** are single-channel textures, only the first element is valid; the rest are undefined. The texture coordinates that we generated for our quad are accessed natively in the shader via the variable **gl_TexCoord[0]**. These coordinates are automatically interpolated across the face of the quad, with **gl_TexCoord[0].s** varying in the *x* direction of the texture, and **gl_TexCoord[0].t** varying in the *y* direction. Our fragment shader calculates a single value of **C**; namely, $\boldsymbol{c_{i,j}}$ where (i, j) is (s, t). Our loop runs from 0 to *dim* and sums the products of the elements of **A** and **B**, with **A** being sampled along the current row, and **B** being sampled along the current column. Note that the shader is executed once per pixel, with the current pixel in column *s* and row *t*. Therefore, the variable **gl_TexCoord[0].st** is a constant unique to each pixel during the shader's execution. Our resultant texture, **C**, is also a single-channel texture; therefore we store the result in the first channel of the output color (indicated by **gl_FragColor.r**).

We usually wish to access the results of our computations back on the CPU in main memory. In order to transfer **C** back to the CPU we do the following:

```
glReadBuffer(GL_COLOR_ATTACHMENT0_EXT);
glReadPixels(0, 0, dim, dim, GL_LUMINANCE, GL_FLOAT, C_cpu_data);
```

The result of the matrix multiplication can be indexed at (y, x) via the array **C_cpu_data**[*y* * *dim* + *x*].

Our implementation has a few limitations; namely, it is limited in that *dim* * *dim* must be a valid texture size. On current hardware, this limits us to matrices of size 8192 by 8192. Also, our implementation rendered to a single-channel floating-point texture (specifically, a GL_LUMINANCE texture, see Appendix A for details), which must be a valid render target under both the video card and the driver in order to work. This is not widely supported (our test system used an Nvidia 8800 GT under Linux). Fortunately, rendering to a four-channel floating-point texture is more widely supported, and our matrix multiplication example can be optimized by packing two by two submatrices of **A** and **B** into the four color channels red, green, blue, alpha (RGBA) of their respective textures. This approach and more optimizations can be found in the work of Hall et al.

(Hall, Carr, and Hart 2003). Our implementation is intended for didactic purposes only. The full listing can be found in Appendix A.

2.6.2 CUDA Implementation

Our CUDA implementation performs the same computation as our GLSL implementation: **C = A * B**, for matrices of size [*dim*, *dim*]. CUDA introduces some new terminology: the *host* is referring to the CPU, and *host memory* refers to main memory, or RAM. The *device* refers to the GPU, and *device memory* refers to texture memory, or sometimes called global memory when using CUDA[7]. Instead of a fragment shader performing the multiplication, our kernel is written in a dialect of C++. Each kernel invocation is executed over a *grid*, which can be thought of as our computational domain and range. This maps well to our example, which has both a domain and range of size *dim*. The grid is broken up into *blocks*, sometimes called thread blocks, because each block is further broken up into *threads*. Each thread can be thought of as a pixel, on each of which our fragment shader executed on. In this case, each thread calculates an element $c_{i,j}$ of **C**. We refer to the size of each block as **BLOCK_SIZE**, which is optimally chosen to be a multiple of the number of symmetric multiprocessors (SM) on the GPU[8]. By breaking up our computational domain into a multiple of the number of multiprocessors, we ensure that each SM is utilized; splitting it up less, for instance, would not utilize the full capability of the GPU.

We start by defining our matrix dimensions, the size in memory of a matrix, and allocating space on the GPU for each matrix:

```
const int BLOCK_SIZE = 14;
const int dim = BLOCK_SIZE;
const int mem_size = dim * dim * sizeof(float);
float *A_gpu_data, *B_gpu_data, *C_gpu_data;
cudaMalloc((void**) &A_gpu_data, mem_size);
cudaMalloc((void**) &B_gpu_data, mem_size);
cudaMalloc((void**) &C_gpu_data, mem_size);
```

Note that **C_gpu_data** is not a texture in this example, but simply a one-dimensional array of floats, exactly like its CPU counterpart. Next, we allocate space on the CPU for our matrices and randomly assign values to **A** and **B**:

```
float* A_cpu_data = (float*) malloc(mem_size);
float* B_cpu_data = (float*) malloc(mem_size);
float* C_cpu_data = (float*) malloc(mem_size);
random_fill(A_cpu_data, dim * dim);
random_fill(B_cpu_data, dim * dim);
```

We upload **A** and **B** to the GPU:

```
cudaMemcpy(A_gpu_data, A_cpu_data, mem_size, cudaMemcpyHostToDevice);
cudaMemcpy(B_gpu_data, B_cpu_data, mem_size, cudaMemcpyHostToDevice);
```

Before we execute our kernel, we need to specify the number of threads per block, and the number of blocks to split our grid into. This is done using a CUDA-specific datatype called **dim3**. One approach might be to just execute one block with *dim* by *dim* threads:

```
dim3 threadsPerBlock( dim, dim );
dim3 blocksPerGrid( 1, 1 );
```

Note that the third component of a **dim3** is not used in our case. This does not take advantage of all the multiprocessors we have available, however, as each block is executed on one SM. Luckily, CUDA limits the number of threads per block to 512, which would limit us to matrices of size 512 by 512, which we would rather avoid.

Alternatively, we could execute *dim* by *dim* blocks each with one thread:

```
dim3 threadsPerBlock( 1, 1 );
dim3 blocksPerGrid( dim, dim );
```

This relieves the 512 by 512 limitation, and allows us to implement matrix multiplication on the GPU using the exact same syntax as the CPU implementation:

```
__global__ void matrixMul_gpu(float* C, float* A, float* B, const int dim) {
    for (int i = 0; i < dim; ++i) {
        for (int j = 0; j < dim; ++j) {
            float sum = 0;
            for (int k = 0; k < dim; ++k) {
                sum += A[i * dim + k] * B[k * dim + j];
            }
            C[i * dim + j] = sum;
        }
    }
}
```

The keyword **__global__** refers to a function which is *called* from the host, but *executed* on the device. This function is called using the following syntax:

```
matrixMul_gpu<<< blocksPerGrid, threadsPerBlock >>>(
    C_gpu_data, A_gpu_data, B_gpu_data, dim
);
```

Unfortunately, by only using one thread per block, we are not utilizing all 8 stream processors available to each symmetric multiprocessor. In order to take full advantage of the hardware we need to break up the problem into appropriate chunks for each SM as well as each of the stream processors available per SM. We choose the number of threads to be a multiple of the number of SM, which we indicated earlier as

BLOCK_SIZE, and we choose the number of blocks by dividing our computational domain by the size of each block:

```
dim3 threadsPerBlock( BLOCK_SIZE, BLOCK_SIZE );
dim3 blocksPerGrid( dim/BLOCK_SIZE, dim/BLOCK_SIZE );
```

We execute our kernel with the same syntax as above (we have only changed **threadsPerBlock** and **blocksPerGrid**) with the following function call:

```
matrixMul_gpu<<< blocksPerGrid, threadsPerBlock >>>(
    C_gpu_data, A_gpu_data, B_gpu_data, dim
);
```

Using the same syntax as the CPU implementation for our GPU implementation no longer works. Instead, we define it as follows:

```
__global__ void matrixMul_gpu(float* C, float* A, float* B, const int dim) {
    float sum = 0.0;
    for (int i = 0; i < dim; ++i) {
        float a = A[dim * blockDim.y * blockIdx.y + dim * threadIdx.y + i];
        float b = B[      blockDim.x * blockIdx.x + dim * i + threadIdx.x];
        sum += a * b;
    }
    C[dim * blockDim.y  * blockIdx.y +
          blockDim.x  * blockIdx.x +
      dim * threadIdx.y + threadIdx.x] = sum;
}
```

The keyword **__global__** indicates this function is executed on the device, which means the one-dimensional arrays **C**, **A**, and **B** are actually in GPU memory. The variable **blockDim** refers to the dimensions of the block, and is a special variable accessible inside CUDA kernels (much like the aforementioned **gl_TexCoord[0]** accessed inside our fragment shader). In our case, **blockDim.x** and **blockDim.y** are both of size **BLOCK_SIZE**. The variable **blockIdx** refers to the index of the current block being executed; similarly, the variable **threadIdx** refers to the current thread being executed. Our loop sums the products of **a** and **b** by indexing **A** along the current row and **B** along the current column. Unlike our GLSL sample which indexed into **A** and **B** using texture coordinates we passed to the shader, we have full access to these two-dimensional matrices stored in one-dimensional arrays. We also have full write access to global memory; although we don't utilize it, it is possible to write to **A** and **B** just as we write to **C**. Note the only difference between **C** and the other two matrices is the assignment of random values before launching the kernel; no special setup was required to write to **C**. In our GLSL sample, both **A** and **B** were read-only. Illustration 2.2 shows the computation for a single thread block and a single thread.

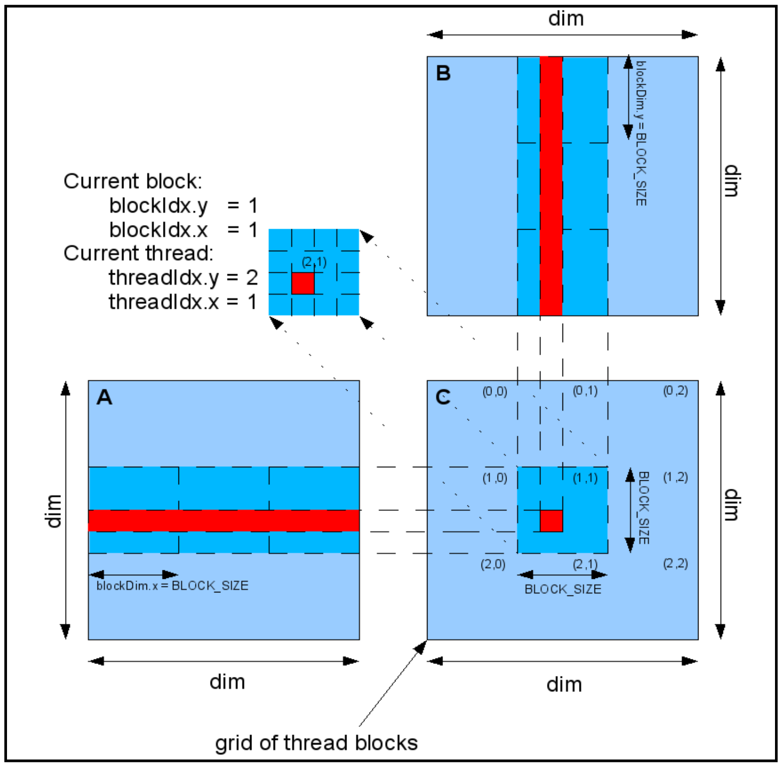

Illustration 2.2 Computation C = A * B

Our CUDA implementation also faces several limitations. Matrix dimensions are required to be a multiple of **BLOCK_SIZE** in order for our algorithm to work. Necessarily, it also requires the latest Nvidia hardware (G80+), as CUDA was just released within the last year. Our implementation does not take advantage of the 16kb shared memory available to each block (which is faster than accessing global memory); however, an example in the CUDA SDK does so, and it also extends to non-square matrices. This shared memory is only exposed via CUDA; it is not available to the graphics APIs. Just as with our GLSL implementation, this example is for illustration purposes only. For high-performance matrix multiplication, Nvidia provides CUBLAS: an implementation of the Basic Linear Algebra Subprograms (BLAS) on top of CUDA. More information on CUBLAS, the CUDA SDK, and the CUDA toolkit can be found on Nvidia's website (NVIDIA 2008, *Nvidia Cuda Zone*), and the CUDA programming guide (NVIDIA 2008, *NVIDIA CUDA*). The full listing for our CUDA example can be found in Appendix B.

CHAPTER 3

SURVEY

In this section we provide a comprehensive overview of all current work in general-purpose computation on graphics hardware. In section 3.1 we take a look at the classic GPGPU problem: real-time rendering and visualization. In section 3.2 we look at image processing algorithms which naturally map to the GPU. In section 3.3 we find some unusual uses of the GPU with audio processing, and in section 3.4 we look at computational geometry. Section 3.5 is a combination of both low-level mathematical methods used in general-purpose GPU programming as well as high-level data structures, the combination of which provides a solid algorithmic background for many of the other applications in this survey. Section 3.6 explores many simulations currently being done on graphics cards, some but not all of which also have a visual component. In section 3.7 we finish the survey with directions to more information that this paper might not have covered, as well as previous surveys conducted.

3.1 GPGPU Rendering

It seems odd to generalize computation on graphics hardware and then turn around and use it for real-time rendering, but that's exactly what some of the earliest GPGPU applications did. As rendering became more sophisticated the necessity to model reality more exactly became computationally costly. Utilizing the GPU's stream processing power to do expensive lighting and shadowing calculations became an obvious evolutionary step. The closeness of the simulation and the rendering, in that they are both on the GPU, became an advantage as well. As GPGPU languages evolve and they become capable of accessing the hardware in more efficient ways than the traditional graphics pipeline (at least when it comes to general computation), it is expected that graphics APIs will be relegated to rendering the result of more expensive computations done by more suitable, more general languages.

3.1.1 Global Illumination

Global illumination refers to a group of methods used to simulate realistic lighting effects. It usually refers to a combination of direct illumination and indirect illumination, where the former is the contribution of light coming directly from a light source, and the latter taking into account light which has been reflected from other objects in the scene. There are many algorithms that can be used to approximate global illumination; of notable interest are those approximations which can be computed in real-time on the GPU.

3.1.1.1 Radiosity

In traditional radiosity as described in computer graphics, the scene is broken up into patches, each with coefficients (called form factors) describing how well each patch can see each other patch. These coefficients are used in a series of linear equations which, when solved, describe the diffuse energy leaving each patch. Radiosity is view-independent and does not describe effects such as specular lighting and reflections, which depend on the viewer's position. Once the energy (or resultant color, if visualized) is calculated, the scene may be viewed from any position without the need to recalculate the radiosity. An introduction to radiosity can be found in (Goral et al. 1984).

Coombe et al. (Coombe, Harris, and Lastra 2004) were the first to implement the entire radiosity algorithm on the GPU. Utilizing a progressive radiosity solution, which computes the coefficients on the fly to avoid O(n^2) storage space of the equations, Coombe et al. modify the traditional radiosity algorithm in a few subtle ways to make it suitable for the SIMD nature of GPU computing. The scene is not subdivided into patches, but instead textures are used to store the radiosity component for each piece of geometry. Normally, in a subdivided scene the patches would be tested for visibility and radiosity (effectively the diffuse energy of the surface) would be assigned to each visible polygon. Instead, the texels of each polygon are tested for visibility, thus mapping well to a fragment program on the GPU.

In addition, a company called Geomerics claims to have an implementation of a real-time radiosity simulation on the GPU that includes not just a model of diffuse energy off of surfaces but also bounced specular highlights (Geometrics 2007).

3.1.1.2 Ray-tracing

While radiosity is a view-independent algorithm that models diffuse energy, ray-tracing is very much view-dependent. Classical Whitted ray-tracing is an extension of ray-casting, where rays are cast from the eye-point into the scene, one per pixel, and, once intersected with an object, the shading of that pixel is determined by the interaction between the surface material and the lights in the scene. Whitted ray-tracing extends the algorithm by casting three additional types of rays once they hit an object in the scene: refraction, reflection, and shadow rays. Since the primary rays originate from the viewer's position, ray-tracing is not a view-independent algorithm, and the shading of each pixel must be recalculated if the view changes. This is made up for by the fact that shadows, reflections, and refractions are much easier to calculate than with other rendering methods.

Ray-tracing on the GPU has been a difficult topic to implement, as typically it is very branch-heavy computation, and also the classic algorithm is presented using recursion, which is not possible on modern GPUs. Purcell was the first to implement a ray-tracer entirely on the GPU, with no part of the algorithm done by the host processor (Purcell 2004). Purcell accelerated the ray intersection computations by utilizing a uniform 3D grid. Each ray was tested for intersection only against geometry in its current grid cell, before being extended to the next, nearby cell. This model was later followed by Christen (Christen 2005), and Karlsson & Ljungstedt (Karlsson and Ljungstedt 2004). Both Christen and Karlsson & Ljungstedt found their GPU implementations to be comparable but not yet faster than professional CPU implementations, such as Mental Images' "Mental Ray," now owned by Nvidia. Christen notes, at the time of his writing, the limited total memory available on graphics cards might have been a limiting factor.

In an attempt to improve the GPU ray-tracing algorithm, Thrane and Simonsen (Thrane and Simonsen 2005) investigated alternatives to the uniform 3D grid partitioning scheme. They tested the uniform grid, kd-tree, and BVH (bounding-volume hierarchy) implementations. They found that the traditional uniform grid was the slowest, with the kd-tree showing some improvement, but the BVH proving the fastest when all three are traversed on the GPU. The reason for this is because the BVH traversal is simpler than the others, as it contains only one branch (and GPUs were not very efficient at branches at the time), and that this promotes low divergence in the execution paths of the various kernels (fragment programs), and therefore has good cache utilization.

Building upon Thrane and Simonsen's BVH work, Carr et al. (Carr et al. 2006) rebuild the threaded hierarchy for each frame entirely on the GPU, thus allowing for dynamic geometry and dynamic level-of-detail (LOD). The term threaded hierarchy refers to a technique in which two pointers are added to each node: a hit pointer referring to the first child, and a miss pointer referring to the next sibling. This avoids the traditional traversal reliance on branches and a stack, allowing it to be efficiently streamed to the GPU. Carr et al. conclude that their BVH stored in a geometry image accelerates GPU-based ray-tracing to where it is competitive with other high-performance ray-tracers.

Recent work by Robert et al. (Robert, Schoepke, and Bieri 2007) shows a performance increase of almost a factor of two compared to a CPU-based kd-tree accelerated ray-tracer by utilizing a hybrid technique combining the GPU and CPU. They introduce the *object intersection buffer* (OIB) which circumvents the need for an acceleration structure (such as the aforementioned BVH or uniform grid) for reducing the number of intersection tests for primary rays. Secondary rays such as reflection and refraction rays are computed on the CPU which exploits the parallelism between the GPU acting as a co-processor and the CPU. Shadow rays are avoided altogether and shadow mapping, commonly used in non-ray-traced rendering, is used as an alternative.

Szirmay-Kalos et al. (Szirmay-Kalos et al. 2005) approximate reflection and refraction rays by utilizing the distance stored in the texels of an environment map, and thus are able to simulate ray-tracing without actually tracing rays. The results produce multiple reflections and refractions comparable to ray-traced scenes, and the implementation is suitable for real-time simulations such as games. Additional information can be found in (Szirmay-Kalos, Aszodi, and Lazányi 2006).

3.2 Image Processing

Image processing in our context refers to digital image processing, and can involve simple operations such as image resizing and scaling, or more complicated algorithms such as principle components analysis (PCA). Digital image processing techniques are used for a variety of things such as image classification, pattern recognition, and feature extraction. Recent work involves utilizing the GPU for real-time video image processing, but the speedup provided by hardware-acceleration is also helpful in offline computations, particularly for large data sets.

An example of such real-time video processing is the work by Fischer (Fischer 2006), whose dissertation contributes to the area of augmented reality in three specific ways. Augmented reality systems display computer-generated information on top of images from the real world, likely captured by a camera or webcam. The *ARGUS* framework is an augmented reality system used in the medical field for surgical navigation. Fischer extends this framework with a hybrid tracking system, a user interaction library, and a method for calculating the occlusion of rendered objects by a patients' anatomy. Using this extension, it is possible to render a virtual chair partially occluded by a real-world table in front of it. The second contribution involves something coined *stylized* augmented reality, in which, after the real-world scene is rendered, optionally with virtual objects next to real objects (hence augmented), the scene is post-processed on the GPU to create effects such as cartoon rendering, or the impression that the scene was painted with a paint brush. Edge detection filters are also possible as a post-process. The third contribution involves a new rendering method for the display of hidden structures in iso-surfaces and polygonal models.

Static image processing remains an important topic, with one of the oldest possibly being image denoising. In (Kharlamov and Podlozhnyuk 2007), Kharlamov & Podlozhnyuk utilize Nvidia's GPGPU language CUDA to implement two popular denoising algorithms: K Nearest Neighbors (KNN) and Non Local Means (NLM). They found that while KNN maps well to GPU hardware, NLM is memory-bound where memory accesses are typically implemented as texture fetches on GPUs. They propose a solution that utilizes CUDA's shared memory model that assumes weights within blocks of pixels are constant, for small pixel blocks (such as 8x8). This reduces the number of texture fetches so that (modified) NLM approaches the speed of KNN, and can actually be combined with KNN to achieve even better visual results. Even despite the decreased performance of the original NLM, all three algorithms are capable of running as a real-time filter (used on video, for example) on modern graphics hardware, specifically Nvidia's Geforce 8.

Mathematically, a convolution is an operator that takes two functions and returns the amount of overlap. In image processing, convolutions are useful for certain algorithms such as edge detection, image smoothing, or image blurring. Podlozhnyuk

(Podlozhnyuk 2007, *Image Convolution with CUDA*) implements a separable convolution filter, also using CUDA. The results show that by utilizing CUDA's shared memory, they achieve twice the performance versus an implementation using standard texture fetches. This is a good example of GPGPU languages advancing past general-purpose GPU processing done using traditional GPU languages. For 2D convolution kernels, a separable approach tends to be less than optimal. Instead, FFTs are used to calculate the discrete fourier transformation more efficiently in (Podlozhnyuk 2007, *FFT-based 2D Convolution*). An additional implementation of FFTs on GPUs can be found in (Marichal-Hernandez, Rodriguez-Ramos, and Rosa 2007), where Marichal-Hernandez et al. use FFTs for wave-front reconstruction in adaptive optics systems.

Another possible filtering kernel is the Gaussian, which is computed on the GPU in (Turkowski 2007). Turkowski's implementation is analogous to polynomial forward differencing as he computes the polynomial coefficients on the fly and incrementally. With each state update the coefficients are recalculated with just one vector instruction.

3.2.1 Volume Reconstruction

Volumetric reconstruction involves the creation of 3D models from 2D source material, such as images or pictures. Hornung & Kobbelt introduce a GPU-based reconstruction method (Hornung and Kobbelt 2006) that solves many of the restrictions of previous methods, such as the matching of surface patches, and biased consistency estimation. The improvement over a CPU-based software approach is massive, with an example 126 image set with 16 million voxels reconstructed in 856 minutes on the CPU compared to just 82 minutes on the GPU. The results showed that the data transfer to and from the GPU was the main bottleneck, and predicted an estimated four times performance increase for texture sizes of 4096^2, with the maximum available at the time being only 2048^2. Since then, the current maximum texture size has increased to 8192^2. For comparison, a recent CPU-based approach is explained in (Habbecke and Kobbelt 2007).

3.3 Audio

Sound cards have transformed from external plug-in cards to being integrated into most modern motherboards, and have not followed the exponential growth curve that graphics cards have enjoyed. Yet, processing the acoustics of a complex virtual 3D scene remains computationally expensive. A few novel approaches using the GPU for audio processing and acoustic simulations have emerged.

Audio processing is a specialized form of digital signal processing (DSP), and using the GPU as a digital signal processor to process audio was seen in Gallo & Tsingos (Gallo and Tsingos 2004) and Whalen (Whalen 2005). Whalen notes that many audio processing algorithms, such as bandpass, multi-tap delay, and reverb are iterative methods that require output from previous computation, and are therefore unsuitable for parallelization, or at the very least must be implemented using multiple passes.

Ray-tracing techniques have been adopted for computing room acoustics and sound wave propagation, as seen in Jedrzejewski (Jedrzejewski and Marasek 2004) and Röber et al. (Röber, Kaminski, and Masuch 2007). Jedrzejewski traced 2^{14} rays uniformly distributed on the source sphere with all rays propagating until reaching 10 reflections. The CPU implementation took 0.5 seconds to compute, whereas the GPU implementation took only 32 milliseconds, with half of the time spent uploading transferring textures to the GPU. Röber et al. subdivided the scene into a uniform 3D grid to reduce the number of sound ray-object intersections. They note future improvements could possibly include different acceleration structures to reduce the number of intersections even further. Techniques involving different acceleration structures for GPU-based ray-tracing can be found in § 3.1.1.2.

3.4 Computational Geometry

Computational geometry is the study of problems stated in terms of geometry (i.e. points, lines, etc) and can be divided into two main categories: combinatorial computational geometry, sometimes called algorithmic geometry, and numerical computational geometry, sometimes called geometric modeling. Problems in combinatorial computational geometry are many; some examples include convex hull calculation, Delaunay triangulation, line segment intersection, and point-in-polygon tests. Geometric modeling is also called computer-aided geometric design (CAGD), and involves curve and surface modeling. Since many applications of computational geometry involve visualizing the result of computation, it is only natural to attempt to move that computation over to the GPU where it is closer to being rendered, and to benefit from the extreme parallelism available.

Early GPU work in computational geometry involved mesh tessellation. Boubekeur & Schlick (Boubekeur and Schlick 2005) implement an on-the-fly tesselator by linearly interpolating between three vertices. Their implementation of the curved point-normal (PN) triangle involves a constant amount of data to be transferred to the

GPU, regardless of the target refinement. The main impetus for their algorithm is the lack of dynamically generated vertices on the GPU. Since then, with the introduction of the geometry shader, that is no longer the case. However, the geometry shader is not primarily a tessellation unit, with the maximum number of floating-point dynamic outputs at the time of this writing being only 1024 (with 8 bytes per vertex this only allows 128 vertices generated) per geometry shader invocation. In the future, it is expected that dedicated tessellation hardware will allow much more dynamic output generated entirely on the GPU.

Rong & Tan (Rong and Tan 2006) compute an approximation to the Voronoi diagram of a set of two-dimensional seeds and claim the difference is hardly visible to the naked eye. They implement an algorithm called *jump flooding* on the GPU which they use to compute the Voronoi diagram approximation, as well as an approximation to the distance transform of a set of 2D seeds, in constant time. They give a CPU implementation of their jump flooding algorithm for higher dimensions, with the reason given that the GPU cannot render into 3D textures. Since then, with the advent of shader model 4 and Direct3D 10, this has also changed. This makes a jump flooding in three dimensions implementation on the GPU likely with future work.

Real-time surface deformation and manipulation remains a difficult problem that has recently seen progress thanks to modern GPU technology. Marinov et al (Marinov, Botsch, Kobbelt 2007) perform multiresolution deformations of large meshes almost completely independent of CPU performance, thus taking advantage of the faster advancement of GPUs in relation to CPUs. The method involves approximating the normal field of the deformed mesh using a small number of attributes passed per vertex.

It is stated that simple recomputation of vertex normals by local averaging is impossible due to lack of neighboring vertex information, but with the advent of the geometry shader this approach is possible. In Zhou et al. (Zhou et al. 2007) a solution is proposed for interactive manipulation of subdivision surfaces, including surfaces with geometric textures and displaced subdivision surfaces. The heart of their algorithm involves a shell deformation solver, which replaces the common iterative solver for handling the highly nonlinear data of deformed subdivision surfaces. The algorithm is stable and uses only local operations, making it suitable for GPU implementation. The accompanying video demonstrates the instability of previous solvers, and shows the GPU algorithm being fed motion-capture data that dynamically deforms the models in real-time.

Aiger & Kedem (Aiger and Kedem 2008) set out to solve a geometric pattern matching problem called the *largest common pointset* (LCP) problem. The LCP problem reduces to searching for the depth in a polytopes arrangement, which they compute using depth peeling on the GPU. They demonstrate an application of the algorithm with real-time object recognition in which a given geometry model can be detected in a gray-scale image regardless of translation and scaling of the model.

3.5 GPU Algorithms and Data Structures

Many GPGPU techniques are attempts at mapping traditional algorithms and data structures to GPUs so that the computational horsepower and extreme parallelism can be taken advantage of. In section 3.5.1 we take a look at general mathematical problems as well as specific subdomains like cryptography and financial calculations. In section 3.5.2 we touch upon higher-level data structures being mapped to the GPU, as opposed to low level data structures such as arrays (textures). Section 3.5.3 looks at sorting structures on GPUs.

3.5.1 Mathematics

General mathematics on the GPU covers a wide range, from abstract computation to more practical computation. Banterle & Giacobazzi (Banterle and Giacobazzi 2007) implement the *octagon abstract domain* efficiently on the GPU, which they then used to create an abstract program analyzer. The octagon abstract domain provides a way to represent a system of inequalities of the form $\pm x \pm y \leq c$, where x and y are program variables and c is a constant (which can be real, rational, or integer) automatically inferred. Their implementation involves difference bound matrices (DBM) which map one-to-one with 2D textures. The static analyzer performs many operators on the GPU, including union, intersection, assignment, widening, test guard operator, and emptiness test. Many tests see a speedup compared to the CPU implementation, but they state their main bottleneck is the reduction check when performing a widening or emptiness test operation. They state this bottleneck could be alleviated with random writes, which (scatter) can be done with the vertex processor or with a GPGPU language such as CUDA or CAL. Another optimization desired is to increase the size of octagons by using hierarchal techniques such as page textures, which can be implemented using a dependent texture lookup.

In (Harris, Sengupta, and Owens 2007), Harris et al. present a parallel implementation of the *all-prefix-sums* operation, commonly known as *scan*. The all-prefix-sums operation takes a binary associative operator $\otimes$ with identity I, and an array of n elements $[x_0, x_1, \ldots, x_{(n-1)}]$ and returns the array $[I, x_0, (x_0 \otimes x_1), \ldots, (x_0 \otimes x_1 \otimes \ldots \otimes x_{(n-2)})]$. For example, with $\otimes$ being multiplication, a scan on the array [2, 5, 3, 1, 8] would return [0, 2, 10, 30, 30]. It is said that there are many applications for scan, including sorting, polynomial evaluation, lexical analysis, and building histograms. In particular, Harris et al. demonstrate scan in use for summed-area tables, stream compaction, and radix sort. Their parallel algorithm is work-efficient in that it does asymptotically no more work than a sequential scan. They benchmark several implementations of the parallel scan, and aside from the GPU speedup over the CPU implementation, the CUDA version outperforms the OpenGL version by up to a factor of seven. This is because CUDA exposes on-chip shared memory, thread synchronization, and scatter writes into memory which are not available in fragment programs (though scatter is at least possible in a vertex program).

Another application of scan is seen in (Lessig 2007), where Lessig computes eigenvalues of a tridiagonal symmetric matrix using a bisection algorithm. Lessig pays particularly close attention to which areas of memory certain operations take place in with two different implementations for different sized matrices: one for matrices of size 512^2 which can fit entirely into CUDA's shared memory cache of 16 KB (for the G80 series), and one for arbitrarily large matrices.

3.5.1.1 Random Number Generation

There are many types of pseudo-random number generators (PRNG) and distributions on which to generate the numbers, the classical combination being the linear congruential generator (LCG) on a uniform distribution. Many PRNGs require high-precision integer arithmetic which, up until recently, GPUs have lacked (usually only obtaining 24 bits of integer precision in a 32 bit floating point number, because integers were not natively supported). Thus, Pang et al. (Pang, Wong, and Heng 2006) proposed a cellular automata (CA) based PRNG that does not require high-precision integer arithmetic or bitwise operations. Testing their CA based PRNG on the GPU against an LCG PRNG on the CPU they achieved on average a ten times speedup, with better results when more numbers are generated (due to GPU-CPU bandwidth, a single random number generated is less efficient than a CPU approach).

With the advent of natively supported integer computation on GPUs, Howes & Thomas (Howes and Thomas 2007) make heavy use of bitwise operations with their hybrid LCG + Tausworthe generator. The period of a generator is how long until the generator must loop and start repeating itself; with their hybrid approach they achieve a period of 2^{121}, compared to a period of 2^{32} for a typical LCG. Howes & Thomas also demonstrate how to map uniformly distributed random numbers to other distributions, such as a Gaussian distribution. They use a Box-Muller transform for this task, which, while typically not used in CPU implementations, maps well to the GPU due to no branches, looping, or memory fetches, but instead is very math-heavy. A Wallace Gaussian generator is also demonstrated for generating Gaussian distributed random numbers directly without first generating them on a uniform distribution and then transforming them. Finally, these PRNGs are applied to two different pricing option models: Asian and Lookback (more GPU financial calculations in § 3.5.1.5). For raw random number generation, they found the GPU implementation on average achieved a

26x speedup, with up to a 59x speedup for the Asian pricing option, and a 23x speedup for the Lookback option.

Another popular generator option is the Mersenne twister, which was not used in (Howes and Thomas 2007) due to having a large state that must be updated sequentially. However, Podlozhnyuk (Podlozhnyuk 2007, *Parallel Mersenne Twister*) parallelizes the Mersenne twister by launching several instances with different initial states, computed using (Matsumoto and Nishimura 1998), and sharing the common state between all threads. They also transform the uniformly distributed random numbers using a Box-Muller transformation, but into a normal distribution instead of a Gaussian one.

3.5.1.2 Cryptography

Cryptography on the graphics card was introduced by Cook et al. (Cook et al. 2005), which concluded that symmetric key encryption was unsuitable for implementation on GPUs. The experiment implemented the block cipher AES algorithm using OpenGL. Since then, GPUs have evolved and a number of architectural improvements have surfaced.

Moss et al. (Moss, Page, and Smart 2007) chose to implement the public-key (a.k.a. asymmetric key) RSA algorithm using a residue number system (RNS). Despite most researchers ignoring expensive public-key algorithms and instead opting for (usually) more speedy symmetric key algorithms, Moss et al. managed to get a 3x speedup over the CPU implementation. They note that the various overheads associated with GPU computation (such as transferring data to the GPU) requires a large number of parallel exponentiations in order to break-even or surpass the speed of a CPU implementation. This is a common realization in GPGPU programming.

With the introduction of shader model 4.0 and Nvidia's G80 we saw a number of improvements that fix the problems concluded by Cook et al, notably the introduction of natively supported integer stream processing. Yamanouchi (Yamanouchi 2007) implemented the AES symmetric key algorithm using a number of Nvidia-specific OpenGL extensions that are nevertheless expected to eventually become multi-vendor extensions. The key extension used in their algorithm is the transform feedback extension (Woolley and Carter 2008), which allows the output of the geometry shader to be redirected into buffer objects as vertex attributes, and optionally not computing the rest of the pipeline (§ 2.2).

3.5.1.3 Floating-point

Current GPUs do not conform exactly to the IEEE 754 floating-point standard, but come extremely close; the most modern of which differ only in their use of specials (NaNs, Inf) (Cebenoyan 2005). Even still, GPUs have only recently achieved full single-precision floating-point arithmetic, and double precision is still only on the horizon. A number of attempts at emulating higher precision floats using the GPU have arrived as a result: Göddeke et al. (Göddeke, Strzodka, and Turek 2007), Thall (Thall 2006), Graça & Defour (Da Graça and Defour 2006). Emulating extended precision floats goes back as far as Dekker (Dekker 1971). Of particular note, Göddeke et al. conclude that using mixed-precision calculations on the GPU for finite element discretizations can achieve the same accuracy as doing the entire calculation in double precision, but with a speedup of 4-5x. The same calculations done using just single precision proved instable.

3.5.1.4 Histograms

A histogram is a table of frequencies; in image processing a histogram usually represents the number of times a pixel value fits within a certain range, defined by that bin, for *n* bins. Histograms are often used in image post-processing (§ 3.2), such as tone mapping operators for high-dynamic range images. To calculate a histogram sequentially on the CPU for a 1D image of *m* pixels for *n* bins, consider the following pseudocode:

```
for (int i = 0; i < n; ++i)
    histogram[i] = 0;

for (int i = 0; i < m; ++i)
    histogram[image[i]]++;
```

As we know, the GPU does data-parallel processing, so this algorithm does not efficiently map directly to the GPU. There are a number of approaches to re-mapping it to the GPU. Early work in Green (Green*05) involves using occlusion queries to count the number of pixels in a bin, and as such requires *n* passes for *n* bins. The resultant histogram is stored in system memory since the final summation is done on the CPU. In addition, it introduces a trade-off in which you choose between either synchronization between the CPU and the GPU in order to read-back the query, or a lag of several frames before the query result is available, neither of which is ideal. Fluck et al. (Fluck et al. 2006) divide the input image into *n* tiles for *n* bins, where each pixel of a given tile must read values from every other pixel in the same tile. A reduction operation (§ 2.3.1) is then performed to sum up local histogram tiles, which takes O(log *m*) passes for an input image of size *m*.

The previous methods used a *gather* approach, where the input image is read in several places before summed in each bin. Scheuermann & Hensley (Scheuermann and Hensley 2007) introduce a *scatter* approach to GPU histogram collection. First they read the input image using either vertex texture fetch (VTF) or render-to-vertex-buffer (R2VB) depending on the IHV (at the time, Nvidia hardware supported VTF while ATI hardware supported R2VB, but neither supported the alternative). Then, using the scatter capability of modern vertex processors, the vertex position is altered depending on where to bin the input pixel, and then they use additive blending to accumulate the resulting histogram in the output render target. This approach builds the histogram using an arbitrary number of bins in a single rendering pass. Run time was constant with regards to histogram size; instead, the performance was bounded by input image size. It also has an advantage in that the histogram is computed and collected entirely on the GPU, and can perform post-processing effects without first transferring any information back to the CPU.

Another similar scatter approach by Diard (Diard 2007) uses the geometry shader to emit vertices which result in fragments being rendered at the correct location to bin a certain input texel, also using blending to accumulate the result in a single pass. The approach is flexible in that a single geometry shader invocation can process the input image, but to further take advantage of the parallelism available the input image is divided into tiles, with each tile being processed by a geometry shader. The size of each tile can be optimized depending on the desired number of geometry shader invocations, with careful attention paid to the number of maximum scalar outputs available in a GS (1024 at the time of this writing).

An approach using CUDA, similar to Fluck et al., is explored in (Podlozhnyuk 2007, *Histogram Calculation in CUDA*). Since additive blending is not available care was taken to use atomic writes for the final histogram collection. When atomic writes were not available, a final accumulation of sub-histograms pass was necessary.

3.5.1.5 Economics and Finance

Computational finance is an area of programming for financial risk management; it uses numerical methods to aid in the prediction of option prices as well as other financial and economic decisions. Option pricing on the GPU was introduced by Kolb & Pharr (Kolb and Pharr 2005). Nvidia's CUDA SDK has several examples of using the GPU for specific option pricing methods. Among these are Black-Scholes (Podlozhmyuk 2007, *Black-Scholes Option Pricing*), the binomial option pricing model (Podlozhnyuk 2007, *Binomial Option Pricing Model*), and Monte Carlo option pricing (Podlozhnyuk and Harris 2007), the latter of which uses a parallel version of the Mersenne Twister (§ 3.5.1.1). The implementations noted careful memory usage on the G80 was necessary for optimal performance, such as fitting as much as possible into shared memory versus global memory.

3.5.2 Data Structures

While we have discussed mapping simple data structures to the GPU, such as mapping arrays to textures, we have not yet discussed higher-level data structures. The most comprehensive work in this area is Lefohn's dissertation (Lefohn 2006), and prior work (Lefohn, Kniss, and Owens 2005). Lefohn introduces *Glift*, a generic, abstract C++ template library for creating, accessing, and traversing parallel random-access data structures on the GPU. It is modeled after the standard template library (STL), a part of the C++ Standard Library, in that it provides a separation between algorithms and data structures, allowing interchangeable and reusable code. Glift abstracts GPU data structures into four areas: physical memory, programmable address translation (i.e. abstracting a 2D texture and indexing into it as if it was a 4D array), virtual memory, and iterators. Using policy-based C++ template design (Alexandrescu 2001) Glift allows useful abstractions that perform almost as efficiently as hand-coded alternatives in OpenGL and Cg, but allows much more succinct code than the hand-written alternative. Lefohn describes GPU versions of a stack, octree, and quadtree in terms of generic Glift components. They then demonstrate four specific applications that use complex data structures: octree 3D paint, adaptive shadow maps, resolution-matched shadow maps, and a novel depth-of-field algorithm. Constructing these applications is made easier using the Glift abstraction, and Glift itself is one step closer to an efficient, high-level C++ standard library for GPUs.

3.5.3 Sorting

Sorting algorithms are usually a first step to structures that are then used for searching or other database operations. Previous work involved in GPU-based sorting can be found in (Govindaraju et al. 2004), (Govindaraju, Raghuvanshi, and Manocha 2005), (Gamma Research Group, GPUSort), and (Kipfer and Westermann 2005).

Recent work by Gress & Zachmann (Gress and Zachmann 2006) presents GPU-ABiSort: an adaptive bitonic merge sorting algorithm that achieves an optimal time complexity of O((n log n)/p) for sorting n values with p stream processing units. The algorithm uses efficient linear stream memory accesses and is scalable up to n / log n stream processing units. Comparing their implementation versus both a CPU reference implementation as well as the GPUSort system mentioned above Gress & Zachmann manage a speedup of over 2.5 times the CPU implementation. For an n of 1048576 GPUSort manages 173 ms compared to a time of 165 ms for GPU-ABiSort. They used the ping-ponging technique (§ 2.3) and describe an efficient mapping of 1D sorting structures to 2D textures to circumvent the size limitation provided by 1D textures at the time.

Govindaraju et al. (Govindaraju et al. 2006) break the PennySort price-performance benchmark with their recent update of GPUSort dubbed GPUTeraSort. The PennySort benchmark is a competition for how many 100-byte records you can sort for a penny, with GPUTeraSort managing a record 60GB. The competition is further broken down into the Daytona and Indy categories, for which GPUTeraSort qualifies for both, and achieves a 4 times performance improvement, and 1.4 times performance improvement, respectively. Govindaraju et al. improve upon their previous bitonic sorting algorithm with a hybrid radix-bitonic sorting algorithm that outperforms previous GPU-based sorting algorithms by 3 to 6 times. For comparison with GPU-ABiSort, GPUTeraSort has a computational complexity of O((n $\log^2 n$) / 2). Some of the limitations of GPUTeraSort are that it prefers fixed-sized keys, as opposed to variable-sized keys, since it is based on radix sort, and that it does not perform as well as adaptive algorithms (such as GPU-ABiSort) on almost sorted data.

3.6 Simulation

Simulation is a natural choice for GPGPU applications. In many cases, it is desirable to only visualize the result of complex physical simulations. In these cases, moving the computation over to the GPU where it is closer to being rendered eliminates the typical bottleneck of data transfer and can also readily take advantage of the parallelism available. In the following sections we examine GPU-powered simulations of cloth, crowds and automatons, fluid dynamics, and physical simulations. We also take a look at some large, multi-GPU applications in section 3.6.5, and finish up with a look at the specific domain of bioinformatics.

3.6.1 Cloth

Early work for cloth simulations on the GPU can be found in (Wloka 2001), (Green 2003, Nvidia cloth sample), (Green 2003, *Stupid OpenGL Shader Tricks*), and (Meggs 2005). With the introduction of shader model 3.0, Zeller (Zeller 2005) and (Zeller 2006) modeled cloth as a network of particles subject to external forces (gravity, wind, drag) held together by a series of constraints including spring and collision constraints. Particle positions and normals were stored in floating-point textures which were updated by a series of pixel shader passes and rendered dynamically using a VTF in the vertex shader. The particle motion was updated using Verlet integration and the constraints were resolved by a relaxation step at each iteration. The system was flexible enough to describe fixed and free-motion particles and thus allowed the cloth to be "cut" dynamically. Zeller updated their work in (Zeller 2007) when shader model 4.0 became available. The primary differences in the new model are particle positions are now stored in vertex buffers instead of textures, and are updated using vertex and geometry shaders instead of pixel shaders. This is possible due to the stream-out functionality of Direct3D 10, which works similarly to the transform feedback (Khronos Group 2007) OpenGL extension mentioned previously in § 3.5.1.2 Cryptography.

3.6.2 Crowd

Recently there has been an interest in crowd simulation and agent-based modeling on the GPU in order to take advantage of the increased flops compared to CPUs. Zioma (Zioma 2006) manages to compute path-finding information on the GPU, but does not compute any artificial intelligence; it is strictly for navigation. Millán et al. (Millán, Hernandez, and Rudomin 2006) compute character behavior on the GPU by using finite-state machines (FSM). They store three different types of texture maps: agent-space maps, which contain agent information (position, state), world-space maps, which contain information on the environment so that characters may interact with it, and FSM maps, which contain information on each state and transitions between states. These texture maps are sampled in a fragment shader and then updated each iteration. To render the agents a combination of level-of-detail (LOD), imposters (basically rendering an image of the agent instead of the agent itself), and OpenGL pseudo-instancing was used. They achieved up to 256 thousand agents simulated and animated interactively at a rate between 6 and 15 frames per second. They claim the main disadvantage with the approach is editing the FSM texture map must be done by hand in order to program agent behavior, and that a tool built to automate this procedure would help immensely.

D'Souza et al. (D’Souza, Lysenko, and Rahmani 2007) simulate and visualize agents at a rate of over 50 updates per second for a population of just over 2 million agents. Instead of encoding agent behavior in a FSM texture map, it is programmed directly using fragment shaders. Texture maps for agent and environment information are still used. Agent interaction with the environment uses a scatter approach, which they accomplish with a VTF in the vertex shader. In all, agent movement, growback, replacement, mating, and pollution formation/diffusion are modeled using shaders. They claim their prototype system outperforms high-performance computing (HPC) clusters as the 2 million agent simulation was done using a single graphics workstation with an Nvidia Geforce 8800.

3.6.3 Fluid

Computational fluid dynamics (CFD) is a vast and difficult subject involving solving the *Navier-Stokes* equations (NSE), a set of nonlinear partial differential equations which describe single-phase fluid flow. An introduction to CFD can be found in (Anderson 1995). Solving these equations is computationally expensive and approximations formed by simplifying the equations (by removing terms) are common. An unconditionally stable solver for the NSE was introduced in (Stam 1999), with previous methods requiring very small time steps to remain stable. This permitted larger time steps which pushed CFD into the realm of real-time simulations. Fluid simulations on the GPU are numerous, some early work of which can be found in (Foster and Fedkiw 2001), (Y. Liu, X. Liu, and Wu 2004), and (Harris 2004) and many others. A few of the most recent are described here.

Sander et al. (Sander, Tatarchuk, and Mitchell 2006) utilize early-z culling to efficiently reduce the amount of work necessary to solve simplified (no viscosity term) NSE in two dimensions. Early-z culling involves taking advantage of the z-buffer to skip costly pixel shaders. An extension of the algorithm to three dimensions is described. Rapidmind (RapidMind 2008, *Real-time Fluid Simulation*) has a demonstration of a water volume based on (Maes, Fujimoto, and Chiba 2006), in which Maes et al. describe an optimization to heightfield-based water simulations. In these simulations the water is animated in three dimensions but the fluid flow is not calculated in all three dimensions.

The Navier-Stokes equations are modeled under an assumption that the fluid is incompressible, and viscous. Removing the viscosity term yields the Euler equations, which are compressible and inviscid. In computer graphics, fluids are often still held to be incompressible (volume does not change) even when solving the Euler equations. Brandvik & Pullan (Brandvik and Pullan 2008) recently implemented two and three dimensional Euler solvers on the GPU using Brook and CUDA (respectively). For a grid size of 300,000 nodes they achieved a speedup of 29 times the CPU implementation for the 2D equations, and a 16 times speedup for 3D.

A comprehensive introduction to the NS equations in terms of computer graphics and game programming can be found in (Bridson and Müller-Fischer 2007). Bridson & Müller-Fischer describe the equations, provide an overview of numerical simulation and advection algorithms, and provide methods for making the fluid incompressible. They describe different kinds of fluid effects that can be achieved such as smoke and water, and introduce the different real-time water simulations used in computer graphics, such as procedural water, heightfield-based water, and smoothed particle hydrodynamics.

Finally, Crane et al. (Crane, Llamas, and Tariq 2007) simulate and render 3D fluids in real-time, including animation and simulation specifics for water, fire, and smoke. Their GPU implementation discretizes space using a Eulerian discretization, and they map grid cells in the computational domain directly to voxels in a 3D texture. Slices of three-dimensional textures can be rendered into and updated using a pixel shader since shader model 4.0. Fluid interaction with solid objects (such as the boundary encapsulating the fluid) are dealt with by denoting solid voxel cells, and modifying the boundary conditions of the fluid equations so that fluid is allowed to flow along the surface of a solid but not through it. Boundary information is stored in 3D textures that can be updated for dynamic obstacles (solids). Voxelization of solid obstacles is updated by rendering the input mesh into slices of a 3D texture using an orthogonal projection and near plane values equal to the depth of the current slice. This allows dynamic solid obstacle updates each frame. Additional terms are kept depending on whether the fluid modeled is smoke, fire, or water. For smoke, density and temperature values are kept and computed using the same advection equation used for velocity. Fire requires a quantity called a *reaction coordinate* which is described by an equation from (Nguyen, Fedkiw, and Jensen 2002). With water, the interaction between the air and the water surface is modeled using the *level set method* described in (Sethian 1999), which only requires an additional scalar value at each grid cell, mapping well to the GPU.

Storage requirements for the simulation are 41 bytes per cell and rendering requires an additional 20 bytes per pixel, much of which can be shared amongst multiple simultaneous fluid simulations. Rendering of the fluid is done by ray-marching through the 3D texture representing the fluid and accumulating densities before compositing with the final scene.

3.6.4 Physics

More sophisticated physical simulations are starting to see increased demand lately as the market demands more than just visual realism but interaction as well. While the subject of physical simulation is as large and difficult as fluid dynamics, in this section we focus primarily on Newtonian dynamics.

Traditionally the domain of CPUs, adoption of physics on the GPU has been slow for a number of reasons. First, the game industry drives a large part of the research done in graphics technology and graphics chipsets, and while graphics in games can scale depending on the end user's hardware, the gameplay is largely the same on every platform because it is difficult to scale. Thus, many physical effects in games have largely been relegated to eye-candy and visual results that don't affect gameplay (which might negatively affect sales), as seen with the release of Havok FX. However, Havok FX was quietly canceled as Havok was acquired by Intel. Second, and more importantly, the computational power required for intensive real-time rigid body simulations was simply not available until recently. Combined with the market demand of increased interaction as well as visual realism, we fully expect physics on the GPU to increase in momentum in the coming years[9].

There are a number of recent *n*-body simulations on the GPU (Nyland, Prins, and Harris 2007), (Stock and Gharakhani 2008), (Laeuchli 2008). An *n*-body simulation is a system in which each body interacts with all other bodies. For example, in astrophysics, an *n*-body simulation might be a series of particles, each representing a star cluster, planet, or galaxy, and affect each other via gravitational forces. Nyland et al. solve the all-pairs *n*-body problem using a Geforce 8 and CUDA. The all-pairs problem refers to calculating all O(n^2) interactions between the bodies, as opposed to approaches which approximate the contribution of large numbers of particles that are far away, which can reduce the complexity to O(*n* log(*n*)) (Darden and Pedersen 1993). The GPU is more memory constrained than arithmetically limited, so to evaluate the matrix of all n^2 force interactions they parallelize one axis of the matrix (*n* rows) and compute sequentially the other axis (*n* columns). To further reduce memory usage computation is done on tiles, which are square regions of the n^2 matrix of size p^2 (for $p << n$), in which only space for 2*p* particle descriptions is needed to describe p^2 interactions by reusing *p* descriptions in later computations. Each tile is computed by a thread block which is further broken down into a thread per row. In effect, Nyland et al. manage to keep memory usage down (to keep from being bandwidth constrained) and by constantly feeding the arithmetic units data they achieve nearly the peak theoretical performance of the Geforce 8800 GTX. At 128 processors of 1.35 GHz each with one flop per cycle per processor the GTX is capable of 172.8 gigaflops, but for multiply-add instructions (MAD) the theoretical peak doubles as two flops per clock cycle is possible. In their all-pairs *n*-body simulation, Nyland et al. manage over 200 gigaflops for *n* of size 2^{14}, claiming a performance improvement of over 50 times an optimized CPU implementation, and 250 times faster than their own reference CPU implementation. They contrast with their previous *n*-body simulation attempt (Nyland, Prins, and Harris 2004) that used Cg and OpenGL, in which all data was either read-only or write-only (see ping-ponging, § 2.3), and required O(n^2) memory.

Harada (Harada 2007) built a rigid-body simulation capable of rendering 16,384 chess pieces at a rate of 44 frames per second. Initially, the method involves generating particles from rigid-bodies by discretizing space around the rigid body and then extracting the solid voxels by way of depth-peeling. These particles are then inserted into a uniform 3D grid in order to compute particle collisions. By using a grid voxel size capable of holding at most one particle, the number of checks a single particle must do to calculate collisions is optimum at 3^3. The number of particles per rigid body is adjustable, thus allowing for a tradeoff between speed (low number of particles) and accuracy depending on the application. Collision reaction between particles is computed using the discrete element method (DEM), which is a repulsive force modeled by a linear spring, and a damping force modeled by a dashpot (Mishra 2003). After the reactions are computed, linear momentum of the rigid body is computed by summing the forces acting on the particles of that rigid body. Angular momentum is the same, but requires the positions of the particles relative to the center of mass of the body. Position and rotation (using a quaternion) of the rigid body are then calculated by integrating from momentum. Finally, new particle positions are calculated relative to the new rigid body position and quaternion, and the simulation repeats. The uniform grid, rigid body positions and quaternions, as well as particle data are all stored using 2D textures. Using a particle method approach to rigid-body dynamics has a number of advantages, notably that collisions are easy and fast, and map well to the GPU. The disadvantage is that the uniform grid structure takes up GPU memory whether particles occupy the space or not (with the trade-off obviously being speed of collision calculations). Harada briefly mentions how the rigid body particle simulation could be combined with a smoothed particle hydrodynamics simulation in which rigid bodies and fluid particles can interact with each other in the same unified system.

3.6.4.1 Collision Detection

An important, and costly, part of any physical simulation is the collision detection. In the previous section we saw a particle-based collision scheme, but it is expensive to compute accurately in that the more complex an object gets the more particles it requires to accurately represent it. Sometimes we would wish to compute collisions for arbitrary convex objects. Previous work in collision detection on the GPU includes many works by UNC's Gamma group (Gamma Research Group, *Collision Detection*), in which many are image-based algorithms that rely on screen-space overlap tests between objects. Sathe (Sathe 2006) uses another approach using cube maps to compute collision detection, but only returns a binary (true/false) result as to whether a collision occurred or not. Ideally, we would like to compute not only collision detection, but also point of contact (or penetration depth) and collision response, that may later be used by a physical simulation.

Collision detection is commonly broken up into two phases: the broad phase, and the narrow phase. The goal of the broad phase is to quickly cull non-colliding pairs using cheap intersection tests, whether by use of spatial subdivision, or cheaper, more coarse collision-only meshes (with the fine mesh reserved for rendering purposes), so that the potentially colliding set is minimized. During the narrow phase collision tests find the exact time and point of contact, which is much more computationally costly, with the idea being that the broad phase culled the majority of potential collisions, leaving the narrow phase the job of calculating only necessary collisions.

Le Grand implemented the broad phase step using CUDA in (Le Grand 2007). They describe the sort and sweep (Baraff and Witkin 1997) and spatial subdivision approaches to the broad phase, and further showed how to parallelize spatial subdivision. The collision cells are then sorted using a parallel version of radix sort. Traversing the sorted list and doing a bounding volume overlap test results in a list of potential collisions suitable for the narrow phase. The results tested the algorithm with Gauss-Seidel physics with 30,720 potentially colliding objects. The algorithm reduced 450,000,000 potential collisions down to 203,000 on average.

Kipfer (Kipfer 2007) chose to accelerate the narrow phase of collision detection by implementing a parallel version of Lemke's algorithm for solving the *linear complementarity problem* (LCP; not to be confused with largest common pointset, § 3.4). First, an introduction to physics is given followed by an explanation of determining and resolving contact points. Then, linear programming, LCP, and quadratic programming are briefly covered. Convex distance calculation is touched upon before introducing Lemke's algorithm and its parallel implementation in CUDA. The results are compared against an optimized CPU LCP solver capable of 21,000 distance queries per second, with the GPU implementation capable of 69,000 queries per second on a Geforce 8800.

3.6.5 GPUs in High-performance Computing, Networks, and Clusters

The idea of incorporating GPUs into cluster computing was demonstrated as early as (Fan et al. 2004), where Fan et al. incorporate 32 GPUs into a CPU cluster, increasing its theoretical peak performance by 512 gigaflops. By comparison, a mere four years later, a single Nvidia G80 is capable of a theoretical peak of 340 gigaflops by itself (Chapter 1). To this end, both major graphics card IHVs have introduced headless GPUs in an attempt to enter the HPC market.

Seamans & Alexander (Seamans and Alexander 2007) built a prototype virus scanning system to see if the GPU could act as a cheap commodity version of a high-end network processor. They drew many comparisons of a modern GPU to Intel's IXP2800 network processor, namely the highly data-parallel algorithms and high speed memory subsystems. The prototype lacked the features of many mature virus scanning products and libraries, such as regular-expression signatures and MD5 hash comparisons. They compared their prototype against the open source ClamAV library and omitted the features their prototype lacked before doing performance benchmarks. The results show a 27x speedup when no matches were found versus up to an 11x speedup in the worst case when all possible signatures were matches, stating that the reason for the performance difference was the time taken to transfer information back to the CPU in the case of a match.

Deschizeaux & Blanc (Deschizeaux and Blanc 2007) also built a prototype but this time for seismic processing of the earth's subsurface. The oil and gas industry processes terabytes of image data from seismic surveys, and the prototype was built to estimate the feasibility of GPUs in clusters. The sound wave propagation is modeled through a finite-difference algorithm applied in the frequency domain. They found a speedup ranging from 8x to 15x for their CUDA kernels compared to an optimized CPU version. They state that the main drawback is the transfer speed between the GPU and the CPU over the PCIe bus, and conclude that GPUs have the potential to disrupt the price/performance of typical CPU clusters.

Göddeke et al. (Göddeke et al. 2008) analyzed the addition of GPU code to a parallel multigrid solver used for finite element simulations. They took the finite element package FEAST and found that by modifying less than 1% of the code they achieved a speedup of two to three times for large problems. Rather than replace the solver they simply added additional paths depending on if the node in the cluster has a capable GPU, allowing for heterogeneous high performance computing. They found that system bandwidth is the most important performance factor for linear system solvers. While the GPU they used had only single-precision floating point numbers available they used a mixed precision emulation approach that allowed them to achieve the same accuracy as the CPU solvers using double-precision (§ 3.5.1.3).

Finally, it is worth noting the increasing prevalence of GPUs in distributed computing projects, such as the Folding@Home project, which released its GPU client in October of 2006 (Pande 2006). Unfortunately, the GPU client is only available for Windows computers with ATI graphics cards at the time of this writing. They found that the average GPU donated provides 60 times the performance of the average CPU donated (Elsen et al. 2006) (also see Illustration 3.1 below).

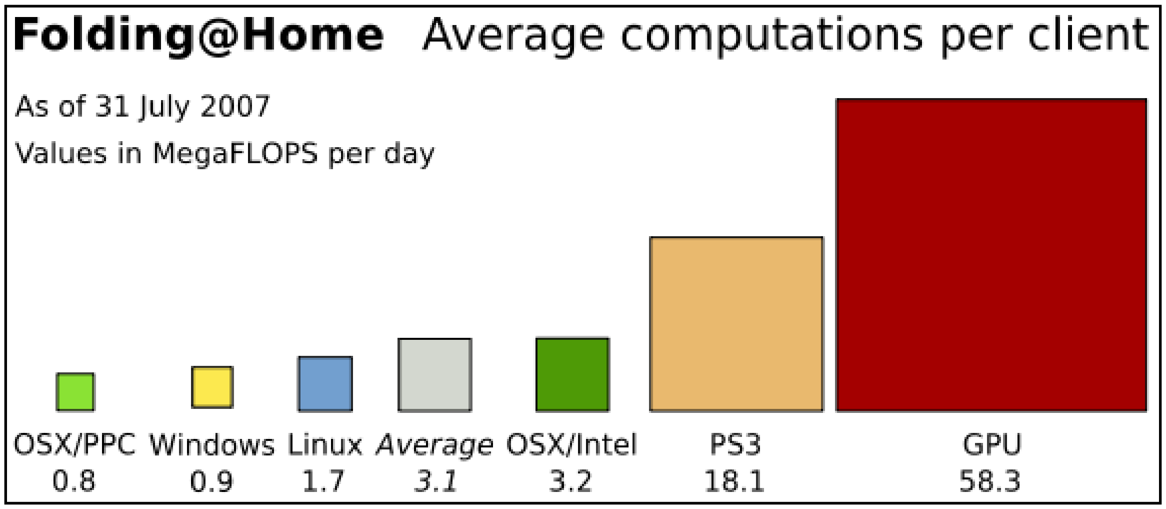

Illustration 3.1: Folding@Home Average Computations per Client[10]

3.6.6 Bioinformatics

While Folding@Home is technically under the umbrella of bioinformatics and computational biology, in this section we focus on single workstation projects. Charalambous et al. (Charalambous, Trancoso, and Stamatakis 2005) did some early work translating parts of RAxML, a program for phylogenetic tree inference, to the GPU. They profiled the OpenMP version of RAxML and found a single loop that accounted for over 50% of the execution time. After converting this loop over to the GPU they found a 3x speedup for the loop itself and a 1.2x speedup for the overall application, which shows promising results for porting the rest of the computationally heavy loops over to the GPU.

Recent work by Manavski & Valle (Manavski and Valle 2008) used CUDA to accelerate the Smith-Waterman sequence alignment algorithm. Unlike the heuristic approaches used in SSEARCH and BLAST they manage to implement the exact Smith-Waterman algorithm with no approximations. In addition, after extensive benchmarking they found that their CUDA implementation was 18x faster than an OpenGL implementation, 3x faster than an optimized SIMD version, 30x faster than SSEARCH and 2.4x faster than BLAST.

3.7 Previous Work

Aside from the information presented here one can find many resources on general-purpose computation on GPUs at GPGPU.org, as well as the websites of both Nvidia and ATI. Other surveys of GPGPU work include the seminal work of Owens et al. (Owens et al. 2005), which was later updated in (Owens et al. 2007).

CHAPTER 4

CONCLUSION

We have introduced the concept of general-purpose computation on graphics hardware, and provided the tools and information for further investigation. In our example, we illustrated the difficulties involved in implementing a simple algorithm on the GPU using modern GPU and GPGPU programming methods. Until the tools and libraries available for GPU programming reach the sophistication of those in CPU programming, these difficulties of implementation will remain. In exchange for this difficulty, the computational horsepower and parallel capabilities of GPUs provide an order of magnitude more performance. In our survey, we detailed research which either outperforms its CPU counterpart or is impossible to implement using the CPU. Many of these benefit simply by using parallel algorithms which are inappropriate for traditional serial processors. However, CPUs are becoming more parallelized every year. Intel is rumored to have plans for a discrete GPU, code named Larrabee, which is predicted to compete with Nvidia and ATI's offerings. It will use a derivative of the x86 instruction set for its shader cores, as opposed to a custom-made instruction set that attempts to abide by the IEEE 754 standard for floating-point arithmetic (currently employed by Nvidia and ATI). Simultaneously, it is expected that GPUs will increase their programmability; possibly exposing more parts of the graphics pipeline to shader programs. As a third side, STI's Cell architecture combines several lightweight processing cores with a GPU and connects them with a high-bandwidth 20GB/s bus.

Currently, GPUs have emerged as the dominant data-parallel coprocessor by achieving the highest performance (in terms of Gflops) as well as the best price/performance ratio. However, with the increased generality of GPUs and the parallelization of CPUs it is uncertain which will surface as the dominant stream processor. We predict that stream processing languages will become more mature and will become the preferred programming platform for both desktop computing and supercomputing. Forward looking platforms, such as the Rapidmind framework, abstract all three architectures (GPU, CPU, Cell). Will GPGPU form the basis for future stream processing in light of the parallel programming revolution?

NOTES

1 The vertex cache size can be queried via the Direct3D API.

2 The term GPU was first coined by Nvidia upon introducing of the Geforce 256.

3 Pixel and vertex shader version 1.0 are reference models and were never actually implemented. Versions 1.1 were the first actual implementations.

4 Single-channel refers to a texture with only one color channel; for example, red, instead of red, green, blue, and alpha (RGBA). However, here we are storing floating-point numbers instead of color information in this channel.

5 It is possible to have non-power-of-two textures without using texture rectangles, but they are indexed via normalized texture coordinates; here, we would prefer coordinates in the range [0, *dim*], and thus we use texture rectangles.

6 A **vec4**'s elements may be accessed via **.r**, **.g**, **.b**, or **.a** or any combination thereof, including **.rgba** or **.rrgg**, the latter of which accesses its elements multiple times (a technique called swizzling).

7 CUDA also supports sampling textures, and therefore texture filtering, but by default reads from global memory are unfiltered and treated as floating-point values.

8 In our case, the 8800 GT has 14 multiprocessors with 8 stream processors each, for a total of 112 stream processors.

9 Pun intended.

10 Obtained from http://en.wikipedia.org/wiki/Image:F%40H_FLOPS_per_client.svg. Licensed under the Creative Commons license which can be found at http://creativecommons.org/licenses/by-sa/2.5/legalcode.

APPENDIX A

matrixMul_glsl.cpp

```
// Compilation requirements:
// - GLFW
// - GLEW
// Graphics card requirements:
// - ARB_texture_rectangle
// - ARB_texture_float
// - GLSL capable
// - EXT_framebuffer_object
// - GL_LUMINANCE_32F_ARB must be a supported render target.
//   See the EXT_framebuffer_object specification for details.
//
// Compile with the command:
// g++ matrixMul_glsl.cpp -lglfw –lGLEW
#include <GL/glew.h>
#include <GL/glfw.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

GLuint createTexture(const int dim);
void random_fill(float* data, int size);
void print_matrix(float* data, const int dim);
void matrixMul_cpu(float* C, float* A, float* B, const int dim);

// Computes C = A * B using a GLSL fragment shader,
// where C, A, and B are square matrices of size [dim, dim].
int main(int argc, char* argv[]) {
  srand(2008);

  // Create an OpenGL context; see the GLFW documentation for details.
  glfwInit();
  glfwOpenWindow(2, 2, 8,8,8,8, 32, 0, GLFW_WINDOW);
  glewInit();

  // Matrix dimensions are [dim, dim].
  const int dim = 7;

  // Allocate space on the GPU for A, B, and C as textures.
  GLuint A_gpu_data = createTexture(dim);
  GLuint B_gpu_data = createTexture(dim);
  GLuint C_gpu_data = createTexture(dim);

  // Allocate space on the CPU.
  float* A_cpu_data = (float*) malloc(dim * dim * sizeof(float));
  float* B_cpu_data = (float*) malloc(dim * dim * sizeof(float));
  float* C_cpu_data = (float*) malloc(dim * dim * sizeof(float));

  // Assign random floats to A and B.
  random_fill(A_cpu_data, dim * dim);
  random_fill(B_cpu_data, dim * dim);
```

```

// Upload A and B to the GPU.
glBindTexture(GL_TEXTURE_RECTANGLE_ARB, A_gpu_data);
glTexSubImage2D(GL_TEXTURE_RECTANGLE_ARB, 0, 0, 0, dim, dim, GL_LUMINANCE,
  GL_FLOAT, A_cpu_data);
glBindTexture(GL_TEXTURE_RECTANGLE_ARB, B_gpu_data);
glTexSubImage2D(GL_TEXTURE_RECTANGLE_ARB, 0, 0, 0, dim, dim, GL_LUMINANCE,
  GL_FLOAT, B_cpu_data);

// Create a framebuffer object and attach C so that we can render to it.
GLuint fbo = 0;
glGenFramebuffersEXT(1, &fbo);
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fbo);
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT0_EXT,
  GL_TEXTURE_RECTANGLE_ARB, C_gpu_data, 0);

// Our fragment shader does the actual matrix multiplication.
const char* fShaderSource = " \
  #extension GL_ARB_texture_rectangle : enable\n \
  uniform sampler2DRect A; \
  uniform sampler2DRect B; \
  uniform int dim; \
  void main() { \
    float sum = 0.0; \
    for (int i = 0; i < dim; ++i) { \
      float a = texture2DRect(A, vec2(i, gl_TexCoord[0].t)).r; \
      float b = texture2DRect(B, vec2(gl_TexCoord[0].s, i)).r; \
      sum += a * b; \
    } \
    gl_FragColor.r = sum;\
  }";

// Create our GLSL program and fragment shader.
GLuint fShader = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fShader, 1, &fShaderSource, NULL);
glCompileShader(fShader);
GLuint program = glCreateProgram();
glAttachShader(program, fShader);
glLinkProgram(program);

// create a 1:1 mapping of pixels to texels
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0.0, dim, 0.0, dim);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glViewport(0, 0, dim, dim);

// Render to C using A and B as textures
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fbo);
glDrawBuffer(GL_COLOR_ATTACHMENT0_EXT);
glClear(GL_COLOR_BUFFER_BIT);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_RECTANGLE_ARB, A_gpu_data);
glActiveTexture(GL_TEXTURE1);
```

```
  glBindTexture(GL_TEXTURE_RECTANGLE_ARB, B_gpu_data);
  glUseProgram(program);
  glUniform1i(glGetUniformLocation(program, "A"), 0);
  glUniform1i(glGetUniformLocation(program, "B"), 1);
  glUniform1i(glGetUniformLocation(program, "dim"), dim);

  // Render a screen-sized quad which generates a pixel for every texel in C,
  // and runs our fragment shader for each pixel.  Note the unnormalized
  // texture coordinates in the range [0, dim],
  // which are accessed in our shader.
  glBegin(GL_QUADS);
    glTexCoord2f(0, 0);     glVertex2f(0, 0);
    glTexCoord2f(dim, 0);   glVertex2f(dim, 0);
    glTexCoord2f(dim, dim); glVertex2f(dim, dim);
    glTexCoord2f(0, dim);   glVertex2f(0, dim);
  glEnd();

  // Read back C from the GPU
  glReadBuffer(GL_COLOR_ATTACHMENT0_EXT);
  glReadPixels(0, 0, dim, dim, GL_LUMINANCE, GL_FLOAT, C_cpu_data);

  // Print the results.  The result now resides in the array C_cpu_data.
  // C(y, x) is now accessible via C_cpu_data[y * dim + x]
  // for 0 <= y <= dim, and 0 <= x <= dim.
  printf("A on cpu\n");
  print_matrix(A_cpu_data, dim);
  printf("B on cpu\n");
  print_matrix(B_cpu_data, dim);
  printf("C computed on gpu\n");
  print_matrix(C_cpu_data, dim);

  // Recompute C on the cpu for verification.
  free(C_cpu_data);
  C_cpu_data = (float*) malloc(dim * dim * sizeof(float));
  matrixMul_cpu(C_cpu_data, A_cpu_data, B_cpu_data, dim);
  printf("C computed on cpu\n");
  print_matrix(C_cpu_data, dim);

  // Cleanup.
  free(A_cpu_data);
  free(B_cpu_data);
  free(C_cpu_data);
  glDeleteFramebuffersEXT(1, &fbo);
  glDeleteTextures(1, &A_gpu_data);
  glDeleteTextures(1, &B_gpu_data);
  glDeleteTextures(1, &C_gpu_data);
  glfwTerminate();
}

// Create a single-channel floating-point texture of size [dim, dim].
GLuint createTexture(const int dim) {
  GLuint tex = 0;
  GLenum texture_target = GL_TEXTURE_RECTANGLE_ARB;
  GLenum texture_format = GL_LUMINANCE;
  GLenum internal_format = GL_LUMINANCE32F_ARB;

  glGenTextures(1, &tex);
  glBindTexture(texture_target, tex);
  glTexParameteri(texture_target, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
  glTexParameteri(texture_target, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
  glTexParameteri(texture_target, GL_TEXTURE_WRAP_S, GL_CLAMP);
  glTexParameteri(texture_target, GL_TEXTURE_WRAP_T, GL_CLAMP);
  glTexImage2D(texture_target, 0, internal_format, dim, dim, 0,
    texture_format, GL_FLOAT, 0);
  return tex;
}

void random_fill(float* data, const int size) {
  for (int i = 0; i < size; ++i) {
    data[i] = rand() / (float)RAND_MAX;
  }
}

void print_matrix(float* data, const int dim) {
  for (int i = 0; i < dim; ++i) {
    for (int j = 0; j < dim; ++j)
      printf("%g ", data[i * dim + j]);
    printf("\n");
  }
}

// Computes C = A * B on the CPU.
void matrixMul_cpu(float* C, float* A, float* B, const int dim) {
  for (int i = 0; i < dim; ++i)
    for (int j = 0; j < dim; ++j) {
      float sum = 0;
      for (int k = 0; k < dim; ++k) {
        sum += A[i * dim + k] * B[k * dim + j];
      }
      C[i * dim + j] = sum;
    }
}
```

APPENDIX B

matrixMul_cuda.cu

```
// Compilation requirements:
// - CUDA toolkit
// - CUDA SDK is not required.
// Graphics card requirements:
// - Nvidia Geforce 8 or higher.
//
// Compile with the command:
// nvcc matrixMul_cuda.cu -run –lcudart
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <cuda.h>

const int BLOCK_SIZE = 14;

void random_fill(float* data, const int size);
void print_matrix(float* data, const int dim);
void matrixMul_cpu(float* C, float* A, float* B, const int dim);
__global__ void matrixMul_gpu(float* C, float* A, float* B, const int dim);

// Computes C = A * B using CUDA,
// where C, A, and B are square matrices of size [dim, dim].
int main(int argc, char* argv[]) {
  srand(2008);

  // Matrix dimensions are [dim, dim].
  // Must be a multiple of BLOCK_SIZE.
  const int dim = BLOCK_SIZE;

  // Size in memory of a matrix.
  const int mem_size = dim * dim * sizeof(float);

  // Allocate space on the CPU.
  float* A_cpu_data = (float*) malloc(mem_size);
  float* B_cpu_data = (float*) malloc(mem_size);
  float* C_cpu_data = (float*) malloc(mem_size);

  // Allocate space on the GPU.
  float *A_gpu_data, *B_gpu_data, *C_gpu_data;
  cudaMalloc((void**) &A_gpu_data, mem_size);
  cudaMalloc((void**) &B_gpu_data, mem_size);
  cudaMalloc((void**) &C_gpu_data, mem_size);

  // Assign random floats to A and B.
  random_fill(A_cpu_data, dim * dim);
  random_fill(B_cpu_data, dim * dim);

  // Upload A and B to the GPU.
  cudaMemcpy(A_gpu_data, A_cpu_data, mem_size, cudaMemcpyHostToDevice);
  cudaMemcpy(B_gpu_data, B_cpu_data, mem_size, cudaMemcpyHostToDevice);

  dim3 threadsPerBlock( BLOCK_SIZE, BLOCK_SIZE );
  dim3 blocksPerGrid( dim/threadsPerBlock.x, dim/threadsPerBlock.y );

  // One grid of thread blocks is launched with the specified number of
  // threads per block.
  matrixMul_gpu<<< blocksPerGrid, threadsPerBlock >>>(
    C_gpu_data, A_gpu_data, B_gpu_data, dim
  );

  // Read back C from the GPU.
  cudaMemcpy(C_cpu_data, C_gpu_data, mem_size, cudaMemcpyDeviceToHost);

  // Print the results.  The result now resides in the array C_cpu_data.
  // C(y, x) is now accessible via C_cpu_data[y * dim + x]
  // for 0 <= y <= dim, and 0 <= x <= dim.
  printf("A\n");
  print_matrix(A_cpu_data, dim);
  printf("B\n");
  print_matrix(B_cpu_data, dim);
  printf("C computed on gpu\n");
  print_matrix(C_cpu_data, dim);

  // Recompute C on the cpu for verification.
  free(C_cpu_data);
  C_cpu_data = (float*) malloc(mem_size);
  matrixMul_cpu(C_cpu_data, A_cpu_data, B_cpu_data, dim);
  printf("C computed on cpu\n");
  print_matrix(C_cpu_data, dim);

  // Cleanup.
  free(A_cpu_data);
  free(B_cpu_data);
  free(C_cpu_data);
  cudaFree(A_gpu_data);
  cudaFree(B_gpu_data);
  cudaFree(C_gpu_data);
}

// Computes C = A * B on the GPU.
__global__ void matrixMul_gpu(float* C, float* A, float* B, const int dim) {
  float sum = 0.0;
  for (int i = 0; i < dim; ++i) {
    float a = A[dim * blockDim.y * blockIdx.y + dim * threadIdx.y + i];
    float b = B[      blockDim.x * blockIdx.x + dim * i + threadIdx.x];
    sum += a * b;
  }
  C[dim * blockDim.y  * blockIdx.y +
        blockDim.x  * blockIdx.x +
    dim * threadIdx.y + threadIdx.x] = sum;
}

void random_fill(float* data, const int size) {
  for (int i = 0; i < size; ++i) {
    data[i] = rand() / (float)RAND_MAX;
  }
}

void print_matrix(float* data, const int dim) {
  for (int i = 0; i < dim; ++i) {
    for (int j = 0; j < dim; ++j)
      printf("%g ", data[i * dim + j]);
    printf("\n");
  }
}

// Computes C = A * B on the CPU.
void matrixMul_cpu(float* C, float* A, float* B, const int dim) {
  for (int i = 0; i < dim; ++i)
    for (int j = 0; j < dim; ++j) {
      float sum = 0;
      for (int k = 0; k < dim; ++k) {
        sum += A[i * dim + k] * B[k * dim + j];
      }
      C[i * dim + j] = sum;
    }
}
```

REFERENCES

Advanced Micro Devices, Inc. 2006. *ATI CTM Guide: Technical reference manual*, vol. 1.01. Advanced Micro Devices, Inc. Available at: http://ati.amd.com/companyinfo/researcher/documents/ATI_CTM_Guide.pdf (accessed May 2008).

Aiger, Dror, and Klara Kedem. 2008. Applying graphics hardware to achieve extremely fast geometric pattern matching in two and three dimensional transformation space. *Information Processing Letters*, vol. 105, no.6 (March): 224-230.

Akenine-Möller, Tomas, and Eric Haines. 2002. *Real-time Rendering*. 2nd edition. Natick, Massachusetts: A K Peters, Ltd.

Alexandrescu, Andrei. 2001. *Modern C++ Design*. Boston, Massachusetts: Addison-Wesley Professional.

Anderson, John D. 1995. *Computational Fluid Dynamics*. 1st ed. McGraw-Hill Science/Engineering/Math.

Banterle, Francesco, and Roberto Giacobazzi. 2007. A Fast Implementation of the Octagon Abstract Domain on Graphics Hardware. Paper presented at the 14th International Static Analysis Symposium (SAS), Kongens Lyngby, Denmark, August 22-24, 2007.

Baraff, David, and Andrew Witkin. 1997. *An Introduction to Physically Based Modeling: Rigid Body Simulation II – Nonpenetration Constraints.* Lecture Notes, Robotics Institute, Carnegie Mellon University, Siggraph Course. Available at: http://www.cs.cmu.edu/~baraff/sigcourse/notesd2.pdf (accessed May 2008).

Boubekeur, Tamy, and Christophe Schlick. 2005. *Generic Mesh Refinement on GPU.* Lecture Notes, ACM SIGGRAPH/Eurographics Graphics Hardware. Available at: http://iparla.labri.fr/publications/2005/BS05/GenericMeshRefinementOnGPU.pdf (accessed May 2008).

Brandvik, Tobias, and Graham Pullan. 2008. Acceleration of a 3D Euler Solver Using Commodity Graphics Hardware. Paper presented at the 46th AIAA Aerospace Sciences Meeting and Exhibit, Reno, Nevada, January 07-10, 2008.

Bridson, Robert, and Matthias Müller-Fischer. 2007. Fluid Simulation: SIGGRAPH 2007 course notes. Paper presented at the International Conference on Computer Graphics and Interactive Techniques, San Diego, California, August 2007.

Carr, Nathan A., Jared Hoberock, Keenan Crane, and John C. Hart. 2006. Fast GPU Ray Tracing of Dynamic Meshes using Geometry Images. *ACM International Conference Proceeding Series*, vol. 137, Proceedings of Graphics Interface 2006 (June): 203-209.

Cebenoyan, Cem. 2005. *Nvidia Technical Brief: Floating point specials on the GPU*. Nvidia Corporation. Available at: ftp://download.nvidia.com/developer/Papers/2005/FP_Specials/FP_Specials.pdf (accessed May 2008).

Charalambous, Maria, Pedro Trancoso, and Alexandros Stamatakis. 2005. Initial Experiences Porting a Bioinformatics Application to a Graphics Processor. Paper presented at the 10th Panhellenic Conference in Informatics (PCI), Greece, November 11-13, 2005.

Christen, Martin. 2005. Ray Tracing on GPU. Diploma Thesis, University of Applied Sciences Basel, Switzerland. Available at: http://gpurt.sourceforge.net/DA07_0405_Ray_Tracing_on_GPU-1.0.5.pdf (accessed May 2008).

Cook, Debra L., John Ioannidis, Angelos D. Keromytis, and Jake Luck. 2005. CryptoGraphics: Secret Key Cryptography Using Graphics Cards. Paper Presented at the RSA Conference, Cryptographer's Track. Available at: http://www1.cs.columbia.edu/~angelos/Papers/2004/gc_ctrsa.pdf (accessed May 2008).

Coombe, Greg, Mark J. Harris, and Anselmo Lastra. 2004. Radiosity on Graphics Hardware. Paper presented at Graphics Interface 2004, Ontario, Canada, May 17-19, 2004. Available at: http://www.cs.unc.edu/~coombe/research/radiosity/coombe04radiosity.pdf (accessed May 2008).

Crane, Keenan, Ignacio Llamas, and Sarah Tariq. 2007. Chapter 30: Real-Time Simulation and Rendering of 3D Fluids. In *GPU Gems 3*, ed. Hubert Nguyen, 633-675. Upper Saddle River, New Jersey: Addison-Wesley Professional.

Da Graça, Guillaume, and David Defour. 2006. Implementation of float-float operators on graphics hardware. Paper presented in the 7th Conference on Real Numbers and Computers (RNC'7), Nancy, France, July 10-12, 2006.

Darden, T., D. York, and L. Pedersen. 1993. Particle mesh Ewald: An N-log(N) method for Ewald sums in large systems. *Journal of Chemical Physics*, vol. 98, no. 12, (June): 10089-10092.

Dekker, T. J. 1971. A floating point technique for extending the available precision. *Numerische Mathematik*, vol. 18, no. 3 (June): 224-242.

Deschizeaux, Bernard, and Jean-Yves Blanc. 2007. Chapter 38: Imaging Earth's Subsurface Using CUDA. In *GPU Gems 3*, ed. Hubert Nguyen, 831-850. Upper Saddle River, New Jersey: Addison-Wesley Professional.

Diard, Franck. 2007. Chapter 41: Using the Geometry Shader for Compact and Variable-Length GPU Feedback. In *GPU Gems 3*, ed. Hubert Nguyen. Upper Saddle River, New Jersey: Addison-Wesley Professional.

D'Souza, Roshan M., Mikola Lysenko, and Keyvan Rahmani. 2007. SugarScape on Steroids: simulating over a million agents at interactive rates. Paper presented at the Agent2007 Conference, Chicago, Illinois, November 15-17, 2007.

Elsen, Erich, Mike Houston, V. Vishal, Eric Darve, Pat Hanrahan, Vijay Pande. 2006. N-Body Simulation on GPUs. Paper presented at the 2006 ACM/IEEE Conference on Supercomputing, Tampa, Florida, November 11-17, 2006.

Fan, Zhe, Feng Qiu, Arie Kaufman, and Suzanne Yoakum-Stover. GPU Cluster for High Performance Computing. Paper presented at the 2004 ACM/IEEE Conference on Supercomputing, Pittsburgh, Pennsylvania, November 06-12, 2004.

Fischer, Jan T. 2006. Rendering Methods for Augmented Reality. PhD. diss., University of Tübingen. Available at: http://www.janfischer.com/pub_pages/pub-fischer06-diss.html (accessed May 2008).

Fluck, Oliver, Shmuel Aharon, Daniel Cremers, Mikael Rousson. 2006. GPU Histogram Computation. Poster presented at the International Conference on Computer Graphics and Interactive Techniques, ACM SIGGRAPH 2006 Research Poster, Boston, Massachusetts. July 30-August 03, 2006.

Foster, Nick, and Ronald Fedkiw. 2001. Practical Animation of Liquids. Paper presented at the 28th annual conference on Computer graphics and interactive techniques 2001. Available at: http://physbam.stanford.edu/~fedkiw/papers/stanford2001-02.pdf (accessed May 2008).

Gallo, Emmanuel, and Nicolas Tsingos. 2004. Efficient 3D Audio Processing with the GPU. Poster presented at GP2, ACM Workshop on General Purpose Computing on Graphics Processors, Los Angeles, August 2004.

Gamma Research Group. Collision Detection. University of North Carolina, Department of Computer Science. http://www.cs.unc.edu/~geom/collide/ (accessed May 2008).

Gamma Research Group. GPUSort: High Performance Sorting using Graphics Processors. University of North Carolina, Department of Computer Science, 2003. http://gamma.cs.unc.edu/GPUSORT/ (accessed May 2008).

Geomerics: Technology. Geomerics, Ltd., 2007. http://www.geomerics.com/technology.htm (accessed May 2008).

GLFW Project. SourceForge.net. http://glfw.sourceforge.net/ (accessed May 2008).

Göddeke, Dominik, Robert Strzodka, and Stefan Turek. 2007. Performance and Accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations. *International Journal of Parallel, Emergent and Distributed Systems*, vol. 22, no. 4 (January): 221-256.

Göddeke, Dominik, Robert Strzodka, Jamaludin Mohd-Yusof, Patrick McCormick, Hilmar Wobker, Christian Becker, and Stefan Turek. 2008. Using GPUs to Improve Multigrid Solver Performance on a Cluster. Accepted for publication in the *International Journal of Computational Science and Engineering*. Available at: http://www.mpi-inf.mpg.de/~strzodka/papers/public/GoStMo_08GPUcluster.pdf (accessed May 2008).

Goral, Cindy M., Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile. 1984. Modeling the Interaction of Light Between Diffuse Surfaces. *ACM SIGGRAPH Computer Graphics*, vol. 18, no. 3 (July): 213-222.

Govindaraju, Naga K., Brandon Lloyd, Wei Wang, Ming Lin, and Dinesh Manocha. 2004. Fast Computation of Database Operations Using Graphics Processors. Paper presented at the 2004 ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004.

Govindaraju, Naga K., Nikunj Raghuvanshi, and Dinesh Manocha. 2005. Fast and Approximate Stream Mining of Quantiles and Frequencies using Graphics Processors. Paper presented in the 2005 ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, June 14-16, 2005.

Govindaraju, Naga K., Jim Gray, Ritesh Kumar, and Dinesh Manocha. 2006. GPUTeraSort: High Performance Graphics Co-processor Sorting for Large Database Management. Paper presented in the 2006 ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, June 27-29, 2006.

Green, Simon. 2003. *Nvidia cloth sample*. Nvidia Corporation.

_____. 2003. Stupid OpenGL Shader Tricks. Nvidia Corporation, Inc. http://developer.nvidia.com/docs/IO/8230/GDC2003_OpenGLShaderTricks.pdf (accessed May 2008).

_____. 2005. Image Processing Tricks in OpenGL. Paper presented at the Game Developer's Conference (GDC) 2005, San Francisco, California, March 07-11, 2005.

Gress, Alexander, and Gabriel Zachmann. 2006. GPU-ABiSort: Optimal Parallel Sorting on Stream Architectures. Paper presented at the 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS). Available at: http://cg.in.tu-clausthal.de/papers/gpu-abisort-ipdps-2006/gpu-abisort-ipdps-2006.pdf (accessed May 2008).

Habbecke, Martin, and Leif Kobbelt. 2007. A Surface-Growing Approach to Multi-View Stereo Reconstruction. Paper presented at the 2007 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR), Minneapolis, Minnesota, June 18-23, 2007.

Hall, Jesse D., Nathan A. Carr, and John. C. Hart. 2003. *Cache and Bandwidth Aware Matrix Multiplication on the GPU*. University of Illinois. Available at: http://graphics.cs.uiuc.edu/~jch/papers/UIUCDCS-R-2003-2328.pdf (accessed May 2008).

Harada, Takahiro. 2007. Chapter 29: Real-time Rigid Body Simulation on GPUs. In GPU *Gems 3*, ed. Hubert Nguyen. Upper Saddle River, New Jersey: Addison-Wesley Professional.

Harris, Mark J. 2004. Chapter 38: Fast Fluid Dynamics Simulation on the GPU. In *GPU Gems*, ed. Randima Fernando. Boston, Massachusetts: Addison-Wesley Professional.

_____. 2005. Chapter 31: Mapping Computational Concepts to GPUs. In *GPU Gems 2*, ed. Matt Pharr and Randima Fernando. Upper Saddle River, New Jersey: Addison-Wesley Professional.

Harris, Mark J., Shubhabrata Sengupta, and John D. Owens. 2007. Chapter 39: Parallel Prefix Sum (Scan) with CUDA. In *GPU Gems 3*, ed. Hubert Nguyen. Upper Saddle River, New Jersey: Addison-Wesley Professional.

Hornung, Alexander, and Leif Kobbelt. 2006. Robust and Efficient Photo-Consistency Estimation for Volumetric 3D Reconstruction. *European Conference on Computer Vision (ECCV), LNCS*, pub. Springer Berlin, vol. 3952, p. 179-190.

Howes, Lee, and David Thomas. 2007. Chapter 37: Efficient Random Number Generation and Application using CUDA. In *GPU Gems 3*, ed. Hubert Nguyen. Upper Saddle River, New Jersey: Addison-Wesley Professional.

Jedrzejewski, Marcin, and Krzysztof Marasek. 2004. Computation of Room Acoustics Using Programmable Video Hardware. Paper presented at the 2004 International Conference on Computer Vision and Graphics (ICCVG), Warsaw, Poland, September 2004.

Karlsson, Filip, and Carl Jonan Ljungstedt. 2004. Ray Tracing Fully Implemented on Programmable Graphics Hardware. Master's thesis, Chalmers University of Technology. Available at: http://www.ce.chalmers.se/edu/proj/raygpu/downloads/raygpu_thesis.pdf (accessed May 2008).

Kharlamov, Alexander, and Victor Podlozhnyuk. 2007. *Image Denoising*. Nvidia Corporation. Available at:

developer.download.nvidia.com/compute/cuda/sdk/website/projects/imageDenoising/doc/imageDenoising.pdf (accessed May 2008).

Khronos Group. OpenGL 2.1 Reference Pages. OpenGL Software Development Kit, Khronos Group, 2007. http://www.opengl.org/sdk/docs/man/ (accessed May 2008).

Kipfer, Peter, and Rüdiger Westermann. 2005. Chapter 46: Improved GPU Sorting. In *GPU Gems 2*, ed. Matt Pharr and Randima Fernando. Upper Saddle River, New Jersey: Addison-Wesley Professional.

Kipfer, Peter. 2007. Chapter 33: LCP Algorithms for Collision Detection Using CUDA. In *GPU Gems 3*, ed. Hubert Nguyen. Upper Saddle River, New Jersey: Addison-Wesley Professional.

Kolb, Craig, and Matt Pharr. 2005. Chapter 45: Option Pricing on the GPU. In *GPU Gems 2*, ed. Matt Pharr and Randima Fernando. Upper Saddle River, New Jersey: Addison-Wesley Professional.

Krüger, Jens, and Rüdiger Westermann. 2005. Chapter 44: A GPU Framework for Solving Systems of Linear Equations. In *GPU Gems 2*, ed. Matt Pharr and Randima Fernando. Upper Saddle River, New Jersey: Addison-Wesley Professional.

Laeuchli, Jesse. 2008. Chapter 9.4: N-Body Simulations on the GPU. In *ShaderX^6: Advanced Rendering Techniques*, ed. Wolfgang Engel. Charles River Media.

Lefohn, Aaron E., Joe Kniss, and John Owens. 2005. Chapter 33: Implementing efficient parallel data structures on GPUs. In *GPU Gems 2*, ed. Matt Pharr and Randima Fernando. Upper Saddle River, New Jersey: Addison-Wesley Professional.

Lefohn, Aaron E. 2006. Glift: Generic Data Structures for Graphics Hardware. PhD. diss., University of California Davis. Available at: http://graphics.cs.ucdavis.edu/~lefohn/work/dissertation/ (accessed May 2008).

Le Grand, Scott. 2007. Chapter 32: Broad-phase Collision Detection with CUDA. In *GPU Gems 3*, ed. Hubert Nguyen. Upper Saddle River, New Jersey: Addison-Wesley Professional.

Lessig, Christian. 2007. *Eigenvalue Computation with CUDA*. Nvidia Corporation. Available at: developer.download.nvidia.com/compute/cuda/1_1/Website/projects/eigenvalues/doc/eigenvalues.pdf (accessed May 2008).

Liu, Youquan, Xuehui Liu, and Enhua Wu. 2004. Real-Time 3D Fluid Simulation on GPU with Complex Obstacles. Paper presented at the Computer Graphics and Applications, 12th Pacific Conference, October 06-08, 2004.

Maes, Marcelo M., Tadahiro Fujimoto, and Norishige Chiba. 2006. Efficient Animation of Water Flow on Irregular Terrains. Paper presented at the 4th international conference on Computer graphics and interactive techniques in Australasia and Southeast Asia, Kuala Lumpur, Malaysia, November 29-December 02, 2006.

Manavski, Svetlin A., and Giorgio Valle. 2008. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC Bioinformatics*, vol. 9, suppl. 2 (March). Available at: http://www.manavski.com/downloads/SWcuda01-11-pics.pdf. (accessed May 2008).

Marichal-Hernandez, José Gil, José Manuel Rodriguez-Ramos, and Fernando Rosa. 2007. Modal Fourier wavefront reconstruction using Graphics Processing Units. *Journal of Electronic Imaging*, vol. 16, no. 2 (April-June): 9 pages.

Marinov, Martin, Mario Botsch, and Lief Kobbelt. 2007. GPU-Based Multiresolution Deformation Using Approximate Normal Field Reconstruction. *ACM Journal of Graphics Tools*, vol. 12, no. 1, p. 27-46.

Matsumoto, Makoto, and Takuji Nishimura. 2008. Dynamic Creation of Pseudorandom Number Generators. Paper presented at the 1998 Monte Carlo and Quasi-Monte Carlo Methods (MCQMC), Claremont. Available at: http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/DC/dgene.pdf (accessed May 2008).

McCool, Michael, and Stefanus DuToit. 2004. *Metaprogramming GPUs with Sh*. A K Peters, Ltd.

Meggs, Andrew. 2005. Parachute Pants and Denim Dresses: Taking Real-Time Cloth Beyond Curtains. Paper presented at the Game Developers Conference (GDC) 2005, San Francisco, California, March 07-11, 2005.

Millán, Erik, Benjamín Hernández, and Isaac Rudomin. 2006. Chapter 9.1: Large crowds of autonomous animated characters using fragment shaders and level of detail. In *ShaderX^5: Advanced Rendering Techniques*, ed. Wolfgang Engel. Charles River Media.

Mishra, B. K. 2003. A Review of Computer Simulation of Tumbling Mills by the Discrete Element Method: Part 1 – Contact Mechanics. *International Journal of Mineral Processing*, vol. 71, no. 1-4, p. 73-93.

Moss, Andrew, Dan Page, and Nigel Smart. 2007. Toward Acceleration of RSA Using 3D Graphics Hardware. *Cryptography and Coding*, (December): 369-388.

Nguyen, Duc Quang, Ronald Fedkiw, and Henrik Wann Jensen. 2002. Physically Based Modeling and Animation of Fire. Paper presented at the 29th annual conference on Computer graphics and interactive techniques, San Antonio, Texas, July 23-26, 2002.

NVIDIA. 2006. *Technical Brief: Nvidia Geforce 8800 GPU Architecture Overview.* Nvidia Corporation. Available at: www.nvidia.com/object/IO_37100.html (accessed May 2008).

_____. 2007. *NVIDIA Tesla: GPU Computing Technical Brief*. Vol. 1.0. Nvidia Corporation. Available at: http://www.nvidia.com/docs/IO/43395/Compute_Tech_Brief_v1-0-0_final__Dec07.pdf (accessed May 2008).

_____. 2008. *NVIDIA CUDA: Compute Unified Device Architecture: Programming Guide.* Nvidia Corporation. Available at: http://developer.download.nvidia.com/compute/cuda/2.0-Beta2/docs/Programming_Guide_2.0beta2.pdf (accessed May 2008).

_____. 2008. Nvidia Cuda Zone. NVIDIA Corporation. http://www.nvidia.com/cuda (accessed May 2008).

Nyland, Lars, Jan Prins, and Mark Harris. 2004. The Rapid Evaluation of Potential Fields Using Programmable Graphics Hardware. Paper presented at the 2004 ACM Workshop on General-Purpose Computing on Graphics Processors, Los Angeles, California, August 07-08, 2004.

_____. 2007. Chapter 31: Fast N-Body Simulation with CUDA. In *GPU Gems 3*, ed. Hubert Nguyen. Upper Saddle River, New Jersey: Addison-Wesley Professional.

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

_____. 2007. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, vol. 26, no. 1, (March): 80-113.

Pande, Vijay. Folding@home Distributed Computing. Vijay Pande and Stanford University, 2000-2008. http://folding.stanford.edu/ (accessed May 2008).

Pang, Wai-Man, Tien-Tsin Wong, and Pheng-Ann Heng. 2006. Chapter 9.8: Implementing High-Quality PRNG on GPU. In *ShaderX^5: Advanced Rendering Techniques*, ed. Wolfgang Engel. Charles River Media.

Podlozhnyuk, Victor. 2007. *Binomial Option Pricing Model.* Nvidia Corporation. Available at: http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/binomialOptions/doc/binomialOptions.pdf (accessed May 2008).

_____. 2007. *Black-Scholes Option Pricing.* Nvidia Corporation. Available at: http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/BlackScholes/doc/BlackScholes.pdf (accessed May 2008).

_____. 2007. *FFT-based 2D Convolution.* Nvidia Corporation. Available at: http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/convolutionFFT2D/doc/convolutionFFT2D.pdf (accessed May 2008).

_____. 2007. *Histogram Calculation in CUDA.* Nvidia Corporation. Available at: http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/histogram256/doc/histogram.pdf (accessed May 2008).

_____. 2007. *Image Convolution with CUDA.* Nvidia Corporation. Available at: http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/convolutionSeparable/doc/convolutionSeparable.pdf (accessed May 2008).

_____. 2007. *Parallel Mersenne Twister.* Nvidia Corporation. Available at: http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/MersenneTwister/doc/MersenneTwister.pdf (accessed May 2008).

Podlozhnyuk, Victor, and Mark Harris. 2007. *Monte Carlo Option Pricing.* Nvidia Corporation. Available at: http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/MonteCarlo/doc/MonteCarlo.pdf (accessed May 2008).

Purcell, Timothy J. 2004. Ray Tracing on a Stream Processor. PhD. diss., Stanford University. Available at: http://graphics.stanford.edu/papers/tpurcell_thesis/ (accessed May 2008).

RapidMind. RapidMind Homepage. RapidMind Inc., 2008. http://www.rapidmind.com/ (accessed May 2008).

_____. Real-time Fluid Simulation. RapidMind Inc., 2008. http://www.rapidmind.com/case-fluids.php (accessed May 2008).

Röber, Niklas, Ulrich Kaminski, and Maic Masuch. 2007. Ray Acoustics Using Computer Graphics Technology. Paper presented at the 10th International Conference on Digital Audio Effects (DAFx-07), Bordeaux, France, September 10-15, 2007.

Robert, Philippe C. D., Severin Schoepke, and Hanspeter Bieri. 2007. Hybrid Ray Tracing: Ray Tracing Using GPU-Accelerated Image-Space Methods. Paper presented at the 2nd International Conference on Computer Graphics Theory and Applications (GRAPP 2007), Barcelona, Spain, March 08-11, 2007.

Rong, Guodong, and Tiow-Seng Tan. 2006. Jump Flooding in GPU with Applications to Voronoi Diagram and Distance Transform. Paper presented at the ACM Symposium on Interactive 3D Graphics and Games (I3D '06), Redwood City, California, March 14-17, 2006.

Sander, Pedro V., Natalya Tatarchuk, and Jason L. Mitchell. 2006. Chapter 9.6: Explicit Early-Z Culling for Efficient Fluid Flow Simulation. In *ShaderX^5: Advanced Rendering Techniques*, ed. Wolfgang Engel. Charles River Media.

Sathe, Rahul. 2006. Chapter 9.4: Collision Detection Shader Using Cube-Maps. In *ShaderX^5: Advanced Rendering Techniques*, ed. Wolfgang Engel. Charles River Media.

Scheuermann, Thorsten, and Justin Hensley. 2007. Efficient Histogram Generation Using Scattering on GPUs. Paper presented at the 2007 Symposium on Interactive 3D Graphics and Games (I3D '07), Seattle, Washington, April 30-May 02, 2007.

Seamans, Elizabeth, and Thomas Alexander. 2007. Chapter 35: Fast Virus Signature Matching on the GPU. In *GPU Gems 3*, ed. Hubert Nguyen. Upper Saddle River, New Jersey: Addison-Wesley Professional.

Segal, Mark, and Kurt Akeley. 2006. *The OpenGL Graphics System: A Specification.* Version 2.1. Ed. Chris Frazier, Jon Leech, Pat Brown. Pub. Silicon Graphics, Inc. Available at: http://www.opengl.org/documentation/specs/version2.1/glspec21.pdf (accessed May 2008).

Sethian, J. A. 1999. *Level Set Methods and Fast Marching Methods: Evolving Interfaces in Computational Geometry, Fluid Mechanics, Computer Vision, and Materials Science.* Cambridge, U.K.; New York: Cambridge University Press.

Shreiner, Dave, Mason Woo, Jackie Neider, and Tom Davis. 2007. *OpenGL Programming Guide: The Official Guide to Learning OpenGL*. Version 2.1, 6th edition, Addison-Wesley Professional.

Stam, Jos. 1999. Stable Fluids. Paper presented at the 26th annual conference on Computer graphics and interactive techniques. Available at: http://www.dgp.toronto.edu/people/stam/reality/Research/pdf/ns.pdf (accessed May 2008).

Standford University Graphics Lab. BrookGPU. Merrimac: Standford Streaming Supercomputer Project. http://graphics.stanford.edu/projects/brookgpu (accessed May, 2008).

Stock, Mark J., and Adrin Gharakhani. 2008. Toward efficient GPU-accelerated N-body simulations. Paper presented at the 46th AIAA Aerospace Sciences Meeting and Exhibit, Reno, Nevada, January 07-10, 2008.

Strengert, Magnus, Thomas Klein, and Thomas Ertl. 2007. A Hardware-Aware Debugger for the OpenGL Shading Language. Paper presented at the 22nd ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware, San Diego, California, August 04-05, 2007.

Szirmay-Kalos, László, Barnabás Aszodi, István Lazányi, and Mátyás Premecz. 2005. *Approximate Ray-Tracing on the GPU with Distance Imposters*. Department of Control Engineering and Information Technology, Technical University of

Budapest, Hungary. Available at: http://www.fsz.bme.hu/~szirmay/ibl_link.htm (accessed May 2008).

Szirmay-Kalos, László, Barnabás Aszodi, and István Lazányi. 2006. Chapter 6.2: Ray Tracing Effects without Tracing Rays. In *ShaderX⁴: Advanced Rendering Techniques*, ed. Wolfgang Engel. Charles River Media.

Thall, Andrew. 2006. Extended-Precision Floating-Point Numbers for GPU Computation. Poster presented at the International Conference on Computer Graphics and Interactive Techniques, SIGGRAPH research poster, Boston, Massachusetts, July 30-August 03, 2006.

Thrane, Niels, and Lars Ole Simonsen. 2005. A Comparison of Acceleration Structures for GPU Assisted Ray Tracing. Master's thesis, University of Aarhus. Available at: http://larsole.com/files/GPU_BVHthesis.pdf (accessed May 2008).

Turkowski, Ken. 2007. Chapter 40: Incremental Computation of the Gaussian. In *GPU Gems 3*, ed. Hubert Nguyen. Upper Saddle River, New Jersey: Addison-Wesley Professional.

Whalen, Sean. 2005. *Audio and the Graphics Processing Unit.* Available at: www.node99.org/projects/gpuaudio/gpuaudio.pdf (accessed May 2008).

Wloka, Matthias. 2001. *Interactive Cloth Simulation.* Nvidia Corporation. Available at: http://developer.nvidia.com/docs/IO/1279/ATT/ClothSim.pdf (accessed May 2008).

Woolley, Cliff, and Nick Carter. 2008. *NV_transform_feedback*. Available at: http://www.opengl.org/registry/specs/NV/transform_feedback.txt (accessed May 2008).

Yamanouchi, Takeshi. 2007. Chapter 36: AES Encryption and Decryption on the GPU. In *GPU Gems 3*, ed. Hubert Nguyen. Upper Saddle River, New Jersey: Addison-Wesley Professional.

Zeller, Cyril. 2005. Cloth Simulation on the GPU. Sketch presented at the 2005 ACM SIGGRAPH International Conference on Computer Graphics and Interactive Techniques, Los Angeles, California, July 31-August 04, 2005.

_____. 2006. Chapter 1.2: Practical Cloth Simulation on Modern GPUs. In *ShaderX⁴: Advanced Rendering Techniques*, ed. Wolfgang Engel. Charles River Media.

_____. 2007. *White Paper: Cloth Simulation.* Nvidia Corporation. Available at: developer.download.nvidia.com/whitepapers/2007/SDK10/Cloth.pdf (accessed May 2008).

Zhou, Kun, Xin Huang, Weiwei Xu, Baining Guo, and Heung-Yeung Shum. 2007. Direct Manipulation of Subdivision Surfaces on GPUs. *ACM Transactions on Graphics*, article 91, vol. 26, no. 3, (July).

Zioma, Renaldas. 2006. Chapter 6.4: GPU powered path-finding using precomputed Navigation Mesh approach. In *ShaderX^4: Advanced Rendering Techniques*, ed. Wolfgang Engel. Charles River Media.