STREAM COMPUTING ON GRAPHICS HARDWARE

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Ian Buck

September 2006

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Patrick M. Hanrahan
(Principal Adviser)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Bill Dally

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Mark Horowitz

Approved for the University Committee on Graduate Studies.

Abstract

The raw compute performance of today's graphics processor is truly amazing. With peak performance of over 60 GFLOPS, the compute power of the graphics processor (GPU) dwarfs that of today's commodity CPU at a price of only a few hundred dollars. As the programmability and performance of modern graphics hardware continues to increase, many researchers are looking to graphics hardware to solve computationally intensive problems previously performed on general purpose CPUs. The challenge, however, is how to re-target these processors from game rendering to general computation, such as numerical modeling, scientific computing, or signal processing. Traditional graphics APIs abstract the GPU as a rendering device, involving textures, triangles, and pixels. Mapping an algorithm to use these primitives is not a straightforward operation, even for the most advanced graphics developers. The results were difficult and often unmanageable programming approaches, hindering the overall adoption of GPUs as a mainstream computing device.

In this dissertation, we explore the concept of stream computing with GPUs. We describe the stream processor abstraction for the GPU and how this abstraction and corresponding programming model can efficiently represent computation on the GPU. To formalize the model, we present Brook for GPUs, a programming system for general-purpose computation on programmable graphics hardware. Brook extends C to include simple data-parallel constructs, enabling the use of the GPU as a streaming co-processor. We present a compiler and runtime system that abstracts and virtualizes many aspects of graphics hardware. In addition, we present an analysis of the effectiveness of the GPU as a streaming processor and evaluate the performance of a collection of benchmark applications in comparison to their CPU implementations.

We also discuss some of the algorithmic decisions which are critical for efficient execution when using the stream programming model for the GPU. For a variety of the applications explored in this dissertation, we demonstrate that our Brook implementations not only perform comparably to hand-written GPU code but also up to seven times faster than their CPU counterparts.

Acknowledgements

TBA

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Data parallel computing is making a comeback on the desktop PC, courtesy of modern programmable graphics hardware. Over the last few years, commodity graphics hardware has been rapidly evolving from a fixed function pipeline into a programmable vertex and fragment processor. While this new programmability was primarily designed for real-time shading, many researchers have observed that its capabilities extend beyond rendering. Applications such as matrix multiply [?], cellular automata [?], and a complete ray tracer [?] have been ported to GPUs. This research exposes the potential of graphics hardware for more general computing tasks. In fact, future GPU architectures may be used for physics-based models, natural phenomena simulation, and AI.

As GPUs evolve to incorporate additional programmability, the challenge becomes to provide new functionality without sacrificing the performance advantage over conventional CPUs. GPUs require a different computational model than the traditional von Neuman architecture [?] used by conventional processors. Otherwise, GPUs will suffer from the same problems faced by modern CPUs: limited instruction level parallelism (ILP), excessive use of caching to capture locality, and diminishing cost-performance returns. A different model must be explored.

Along with the architectural model comes a programming model. Originally, programmable GPUs could only be programmed using assembly language. Recently, both Microsoft and NVIDIA have introduced C-like programming languages, HLSL and Cg respectively, that compile to GPU assembly language [?; ?]. McCool [?] has also described several clever ways to metaprogram the graphics pipeline using his SMASH API. A different approach is taken by domain-specific shading languages such as RenderMan [?] and RTSL [?]. These languages are specialized for shading (e.g. by providing light and surface shaders), and hide details of the graphics pipeline such as the number of stages or the number of passes. A significant advantage of the HLSL and Cg approach over the RenderMan and RTSL approach is that they provide a general mechanism for programming the GPU that is not domain-specific; thus potentially enabling a wider range of applications. However, these languages have only taken a half-step towards generality. They still rely on the graphics library and its abstractions to tie different Cg programs together.

1.1 Contributions

This dissertation explores the potential of commodity GPUs as a streaming processor for computation. We make several contributions in the areas of computer systems, parallel programming, and computer graphics outlined below.

- The motivation and presentation of the stream programming abstraction for GPUs. By analysis of the execution model of today's GPUs as well as a close examination of how GPUs have been able to scale in performance, we describe two main reasons why GPUs have been outperforming CPU architectures: data parallelism and arithmetic intensity (the ratio of computation to bandwidth). The streaming model is built upon these ideas.

- The presentation of the Brook stream programming system for general-purpose GPU computing. Through the use of streams, kernels and reduction operators, Brook formalizes the stream programming abstraction and the GPU as a streaming processor.

- The demonstration of how various GPU hardware limitations can be virtualized or extended using our compiler and runtime system; specifically, the GPU memory system, the number of supported shader outputs, and support for user-defined data structures.

- The evaluation of the costs of using GPU as a streaming processor, specifically the computational intensity (ratio of data transfer time vs. execution time), the kernel call overhead, and the arithmetic intensity.

- The performance evaluation of computing with the GPU compared to the CPU with a variety of benchmark applications implemented in both Brook, hand-coded GPU code, and optimized CPU code.

- The presentation of common algorithmic tradeoffs for more efficient execution of stream programs on GPUs. Specifically, we explore techniques to minimize the need for scatter (indirect write) and the use of branching in kernel functions.

While computing with the GPU is potentially a very broad topic, the purpose of this thesis is to motivate and support the stream programming abstraction for GPUs. There certainly has been other published work using the GPU as a compute engine which exploits domain specific knowledge to map computation onto the graphics pipeline. While we do not dispute that these implementations can be quite efficient, the goal of this work is to target general purpose computation.

1.2 Outline

Before presenting the streaming programming model for GPUs, we must understand the potential for GPUs to perform general computation and understand the aspects of computer graphics which allow them to perform so well. We begin with chapter 2 which provides a brief introduction to GPU architectures and a performance study of their compute potential. We also outline existing work in shading languages and programming models for GPUs as well as similar parallel architectures.

In chapter 3 we present the stream programming model for GPUs and discuss why stream programming is an appropriate model for computation on the GPU. In Chapter 4, we present the Brook stream programming language for GPUs and describe how it relates to other parallel and shading languages. Chapter 5 discusses the mapping of the Brook system onto GPUs. This chapter illustrates how the stream programming model can be efficiently mapped onto the GPU and discusses how we can virtualize many of the hardware constraints which arise when targeting computation on top of a rendering device. Chapter 6 presents an analysis of the effectiveness of the GPU as a stream processor with a performance case study of a variety of benchmark applications on the GPU. In addition, chapter 7 resents a programming model case study illustrating some of the challenges in implementing applications with the streaming programming model and suggests ways to restructure the computation for efficient execution on the GPU. Finally, in chapter **??** we revisit our contributions, highlight some of the lessons learned mapping the stream programming model onto GPUs, and suggest areas of future work.

Chapter 2

The Evolution of the GPU

The work presented in this thesis draws from the evolution of ideas and hardware trends in graphics hardware and stream programming. In this chapter, we discuss the evolution of the GPU from a simple fixed-function rendering device to its current status as a programmable shading processor and discuss its potential for non-rendering compute applications. We also compare the GPU's instruction and memory performance characteristics to its commodity counterpart, the CPU.

This work also draws heavily from existing ideas in data parallel computing and those languages designed for stream processors currently in development within the computer architecture research community. In the second section of this chapter, we present a summary of the existing languages and previous work in stream languages and architecture, highlighting the aspects of these approaches which are applicable to computing with the GPU. We also discuss some of the existing architectural models designed for stream computing.

2.1 Programmable Graphics Hardware

Programmable graphics hardware dates back to the original programmable framebuffer architectures [?]. One of the most influential programmable graphics systems was the UNC PixelPlanes series [?] culminating in the PixelFlow machine [?]. These systems embedded pixel processors, running as a SIMD processor, on the

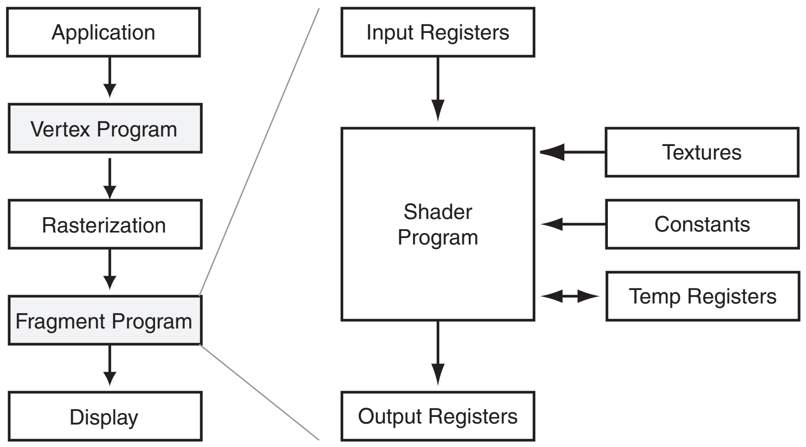


Figure 2.1: The programmable GPU pipeline

same chip as framebuffer memory. SIMD and vector processing operators involve a read, an execution of a single instruction, and a write to off-chip memory [?; ?]. Today's graphics hardware executes small programs where instructions load and store data to local temporary registers rather than to memory.

The basic execution model of modern programmable graphics accelerators, such as the ATI X800XT and the NVIDIA GeForce 6800 [?; ?], is shown in figure 2.1. For every vertex or fragment to be processed, the graphics hardware places a primitive in the read-only input registers. The shader program is then executed and the results written to the output registers. Each processor executes a user-specified assembly-level shader program consisting of 4-way SIMD instructions [?]. These instructions include standard math operations, such as 3 or 4 component dot products, texture-fetch instructions, and a few special-purpose instructions. During execution, the shader has access to a number of temporary registers as well as to constants set by the host application. Memory is accessible via read-only textures which can be fetched with a two dimensional address. Writing to memory within a shader is only possible via the fixed output address of the fragment to be processed, i.e. there is no indirect texture write available in the fragment processor.

One of the key constraints of today's GPUs is the restriction on communication between the shading of different elements. The instruction set deliberately lacks the ability to communicate information from the shading of one fragment or vertex to another within the same pass[1]. This allows graphics hardware vendors to perform the shading of a primitive in parallel since it is impossible for the programmer to introduce dependencies between shading elements. Today's GPUs feature many parallel rendering *pipelines*, each executing the same shader program over multiple fragments or vertices.

2.1.1 Compute Characteristics

While this new programmability was introduced for real-time shading, it has been observed that these processors feature instruction sets general enough to perform computation beyond the domain of rendering. The driving force behind using the GPU as a compute medium is the potential for commodity rendering hardware to outperform the traditional commodity computing platform, the desktop CPU. We compare the performance characteristics of GPUs with Intel's Pentium 4 processor. These performance comparisons illustrate some of the differences between the GPU and CPU architectures. In addition, we highlight some of the important limitations and constraints of the GPU must be taken into consideration.

GFLOPS

One of the primary reasons researchers are so interested in GPUs as a compute platform is the GPU's floating point computing performance advantage. Figure 2.2 and 2.3 illustrate the continuing growth of GPU's GFLOP performance advantage over CPUs. These graphs compare the observed GFLOP performance over time of both the multiply and multiply-add instruction on both the NVIDIA and ATI GPU compared to the peak *theoretical* performance on the Pentium 4 processor. The time of each data point corresponds to release date of the hardware. The GPU performance test involved timing the execution of the largest possible fragment shader consisting solely of MAD or MUL instructions using a single floating point register. The shader was timed by rendering large screen size quad 512 time and computing the observed

[1]The one exception to this is the derivative instruction whose definition is so lax that communication is effectively non-deterministic.

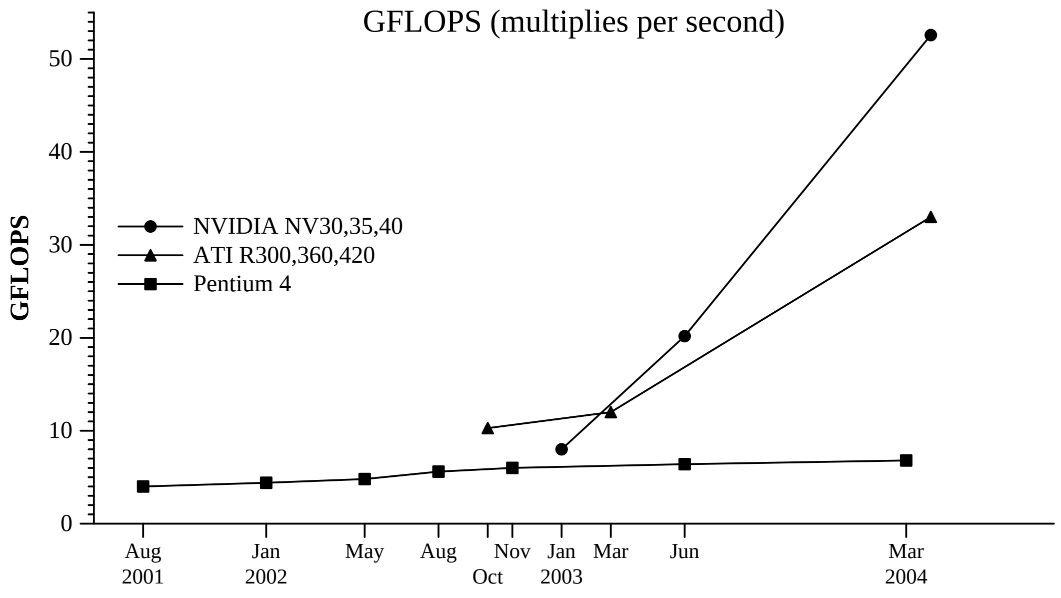


Figure 2.2: GPU vs. CPU MUL performance over time

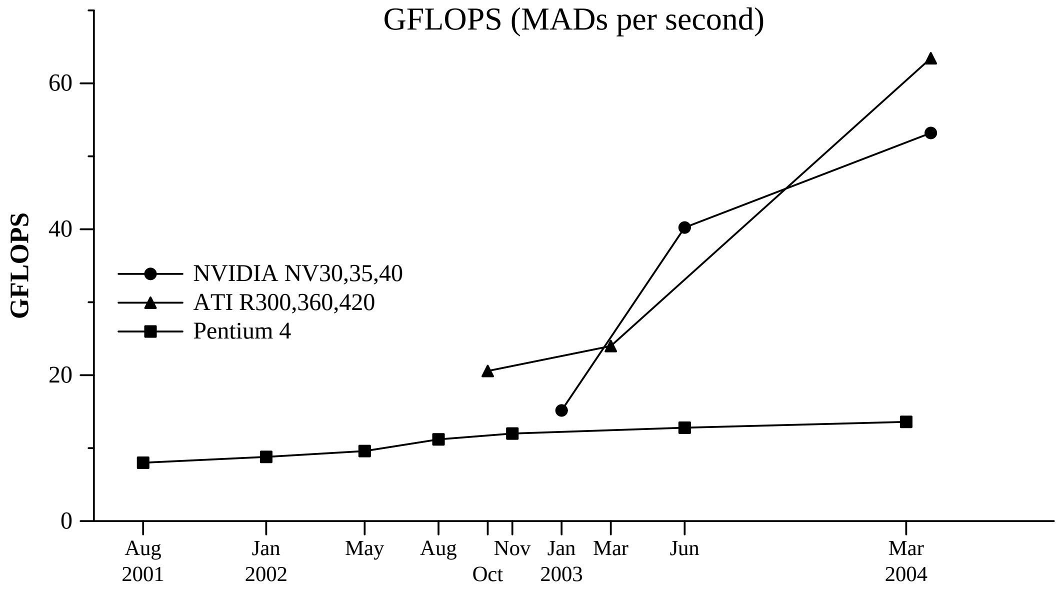


Figure 2.3: GPU vs. CPU MAD performance over time

GFLOP performance. The Pentium GFLOP performance is based on theoretical peak GFLOPS from the Intel Optimization Manual. Current GPUs offer 63 GFLOPS (ATI Radeon X800 MAD performance) compared to 13 GFLOPS on a 3 GHz Pentium 4. Clearly, today's GPUs are almost an order of magnitude faster than CPUs and they continue to get faster.

This performance difference is largely due to the GPU's ability to exploit the parallel nature of programmable shading. As explained above, the instruction set for programmable shading is deliberately constructed to allow for multiple fragments to be rendered in parallel. This allows graphics vendors to spend their growing transistor budgets on additional parallel execution units; the effect of this is observed as additional floating point operations per clock. In contrast, the Pentium4 is designed to execute a sequential program. Adding math units to the processor is unlikely to improve performance since it there is not enough parallelism detectable in the instruction stream to keep these extra units occupied. Therefore, the Pentium 4 performance improvement shown above reflects only clock speed improvements, not additional logic. In contrast, GPU architectures in contrast can easily scale compute performance with additional functional units.

The parallel nature of GPU architectures is clearly a critical aspect of its performance benefit over conventional CPUs. A programming model for computing with the GPU should be designed to reflect this critical aspect of the hardware. While no programming language can automatically convert a serial algorithm into a parallel one, it can encourage the user to construct algorithms which can be executed in parallel with the GPU.

2.1.2 Bandwidth Characteristics

With such enormous compute power, we must also examine memory bandwidth characteristics to see how GPUs are going to keep all of these compute units busy. Figure 2.4 shows the observed read performance of cached, sequential, and random access patterns of today's GPUs compared to the Pentium 4. Compared to the CPU, GPUs

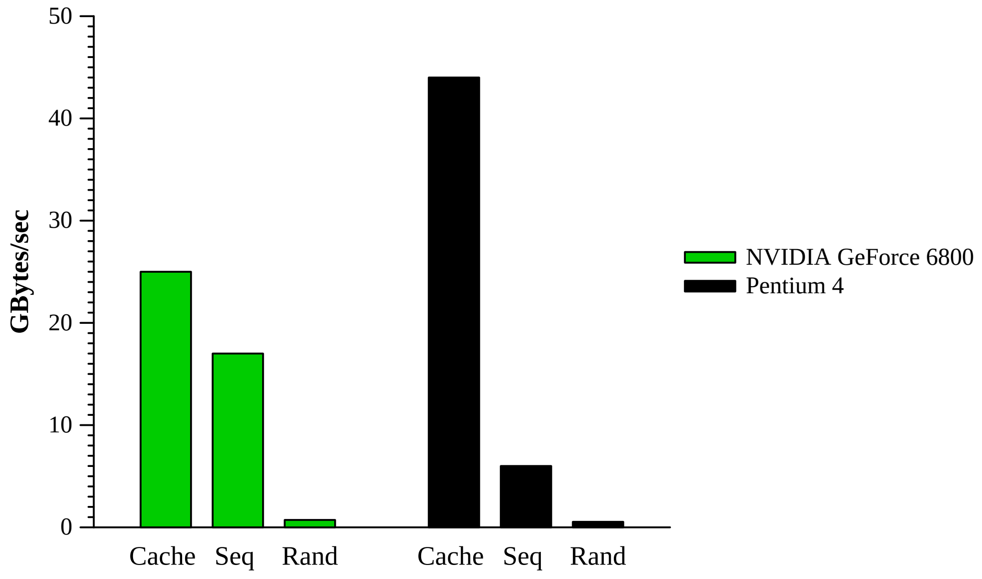


Figure 2.4: Memory read bandwidth: GPU vs. CPU

have significantly more off-chip bandwidth as shown in the sequential bandwidth results. The Pentium 4 uses a 64-bit QDR front side bus operating at 200 MHz yielding 5.96 GB/sec peak sequential bandwidth. The NVIDIA GeForce 6800 memory system consists of a 256-bit DDR memory interface operating at 550 MHz resulting in a theoretical 32.8 GB/sec peak bandwidth. However, we only observe 25 GB/sec of peak read bandwidth since some bandwidth is reserved for writing the framebuffer. Regardless, this 5.5x performance difference reflects the fact that graphics vendors often ship their products with the latest memory technology which is tightly coupled with the GPU. The Pentium 4 must interface with a variety of legacy northbridge memory controllers and commodity memory. The GPU memory system is clearly optimized for sequential access resulting in an order of magnitude difference between sequential and random access. This is not surprising given that rendering requires filling contiguous regions of the screen with data from texture. In the design of our programming model for computing with the GPU, it is clear that structuring computation around sequential memory access is critical for optimal performance.

Another important distinction in memory performance between the GPU and the CPU is the role of the cache. Unlike the cache on the CPU, GPU caches exist

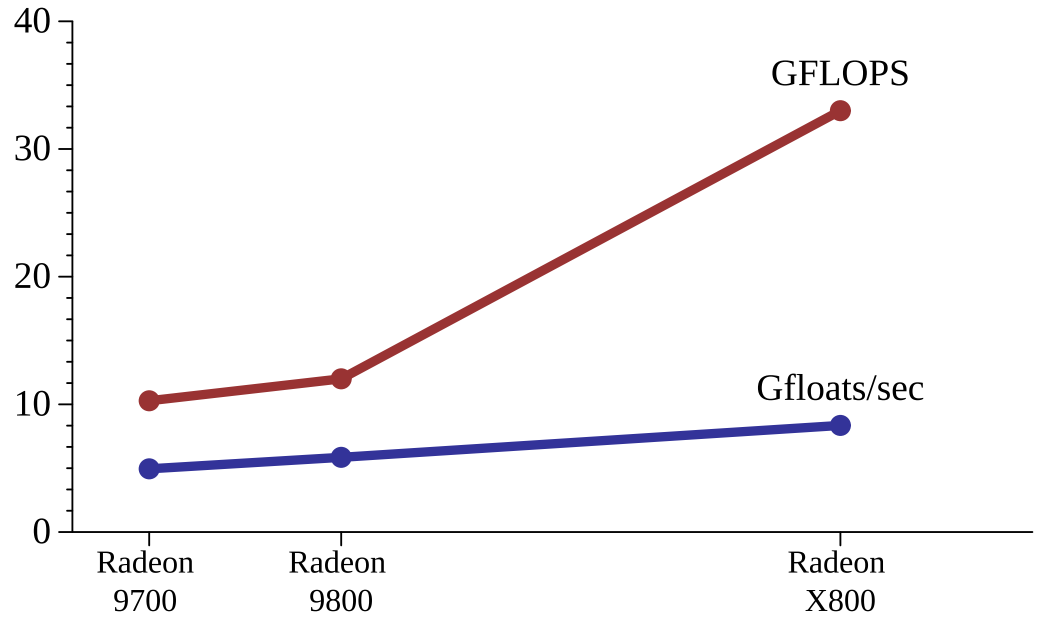


Figure 2.5: The bandwidth gap.

primarily to accelerate texture filtering. As a result, the cache only needs to be as large as the size of the filter kernel for the texture sampler, typically only a few texels which is hardly useful for general purpose computation. This is in contrast to the Pentium 4 which operates at a much higher clock rate and contains megabytes of data. In addition, the Pentium 4 is able to cache both read and write memory operations while the GPU cache is designed for read-only texture data. Any data written to memory, i.e. the framebuffer, is not cached but written out to memory.

2.1.3 The Bandwidth Gap

Despite the GPU's impressive off-chip bandwidth, it is interesting to compare bandwidth with the GPU's compute performance. Figure 2.5 illustrates the growth of peak off-chip bandwidth of the GPU with the growing compute rate over time. The Radeon X800 can multiply two floating point numbers at a rate of 33 GFLOPS, however it can only read floating point data at a rate of 8 Gfloats/second. Clearly, there is a growing gap between the rate at which the GPU can operate on floating point values and the rate at which the memory system can provide the floating point data.

Even though GPUs continue to add more and more transistors, the pin-bandwidth remains limited to 100 Gbit/sec [?] and the limits on the number of pins possible to cram onto a package.

With respect to our programming model design, this 7x performance difference means that it is very likely that applications written for the GPU will be limited by memory operations rather than by computation. Ideally, we would like the reverse for our applications, limited by computation not bandwidth.

While a programming model alone cannot change an algorithm from bandwidth to compute limited, it can encourage the programmer to write code which will have better locality, limiting the bandwidth requirements of the application. In other words, the programming model should encourage applications with a favorable ratio between computation and bandwidth for an algorithm. With the gap between the GPU's compute and bandwidth rates continuing to grow, constructing applications to be compute limited is critical if users are to capitalize on the GPU's potential.

2.2 Programming Languages

There are certainly plenty of high level languages which we can use for expressing computation on the GPU. Today's shading languages already provide a method for expressing graphics-related computation on the GPU. In addition, there are a variety of streaming languages which are being developed at the university level. Finally, there are plenty of lessons learned from data-parallel languages which can be applied to stream computing with GPUs.

2.2.1 Shading Languages

There are a variety of high level shading languages which can be used to program the GPU. These include industry supported languages such as Cg, HLSL, and GLslang as well as the Stanford's RTSL language. However, even with these languages applications must still execute explicit graphics API calls to organize data in texture memory and invoke shaders. Data must be manually packed into textures and transferred to and from the hardware. Shader invocation requires the loading and binding of shader programs as well as the rendering of geometry. As a result, computation on the GPU is not expressed as a set of kernels acting upon streams but rather as a sequence of shading operations on graphics primitives. Even for those proficient in graphics programming, expressing algorithms in this way can be an arduous task.

These languages also fail to virtualize constraints of the underlying hardware. For example, stream elements are limited to natively-supported `float`, `float2`, `float3`, and `float4` types, rather than allowing more complex user-defined structures. In addition, programmers must always be aware of hardware limitations such as shader instruction count, number of shader outputs, and texture sizes. There has been some work in shading languages to alleviate some of these constraints. Chan et al. [?] present an algorithm to subdivide large shaders automatically into smaller shaders to circumvent shader length and input constraints, but do not explore multiple shader outputs. McCool et al. [?; ?] have developed Sh, a system that allows shaders to be defined and executed using a metaprogramming language built on top of C++. Sh is intended primarily as a shading system, though it has been shown to perform other types of computation. However, it does not provide some of the basic operations common in general purpose computing, such as gathers and reductions.

In general, current code written to perform computation on GPUs is developed in a highly graphics-centric environment, posing difficulties for those attempting to map other applications onto graphics hardware.

2.2.2 Data Parallel and Streaming Languages

There have been a variety of stream languages developed in conjunction with novel stream architectures originating out of university research. The StreamC/KernelC programming environment provides an abstraction which allows programmers to map applications to the Imagine processor [?]. Computation is expressed through the kernel function calls over streams. StreamC/KernelC, however, was a language that tightly coupled to that architecture and exposed the sequential nature of how the processor evaluated stream computations. While this can provide an added benefit to programmer, programs written in StreamC/KernelC must be rewritten as architectures in added additional compute units (called clusters in the Imagine processor). GPUs are evolving too rapidly for this to be a viable solution.

The StreamIt language [**?**], originally developed at MIT for the RAW processor [**?**], is another example of a streaming language. StreamIt structures computation as a collection of kernels which can be linked together with a fixed selection of routing connections to stream data from one kernel to the next. While StreamIt structures computation into kernels and streams, it is somewhat constrained by the limited connection types available between kernels.

Both StreamIt and StreamC/KernelC limit the parallel execution of the kernel function over the streams since inputs and outputs can be conditionally *pushed* and *popped* to and from streams. This permits the kernel execution to carry dependencies between different input stream elements. This is in contrast to the GPU where the shader execution model does not permit any state between the inputs.

Data parallel languages are designed intentionally to avoid the limitations present in the existing streaming languages. A good example of a data parallel language is C* [**?**; **?**]. Based on C, execution is expressed as individual operations over collections of data which can be performed in parallel. C* also provides a collection of associative reduction operators which computed in parallel. The Pytolomy project at Berkeley, while primarily designed for DSPs and signal processing, is an example of a programming environment which permits data parallel execution of streams of data. In short, we combine the stream and kernel ideas from StreamC/KernelC with the data parallel ideas from C*. For expressing computation, kernel programming is based on Cg with C providing the remainder of the programming environment.

2.3 GPU Architectural Models

There have been a few studies of mapping the GPU architecture as a compute platform. Labonte et al. [**?**] studied the effectiveness of GPUs as stream processors by evaluating the performance of a streaming virtual machine, SVM, mapped onto graphics hardware. Through this analysis, they showed that the ration of compute to local and global bandwidth was quite similar to other streaming processors developed by university researchers. In addition, the SVM is designed to be a compilation target for a variety of streaming languages and they show how one can implement the SVM on top of existing GPUs.

Peercy et al. [?] demonstrated how the OpenGL architecture [?] can be abstracted as a SIMD processor. Each rendering pass is considered a SIMD instruction that performs a basic arithmetic operation and updates the framebuffer atomically. Using this abstraction, they were able to compile RenderMan to OpenGL 1.2 with imaging extensions. They also argued the case for extended precision fragment processing and framebuffers.

The work by Thompson et al.[?] explores the use of GPUs as a general-purpose vector processor. They implement a software layer on top of the graphics library that provides arithmetic computation on arrays of floating point numbers. The basic problem with the SIMD or vector approach is that each pass involves a single instruction, a read, and a write to off-chip framebuffer memory. The results in significant memory bandwidth use. Today's graphics hardware executes small programs where instructions load and store to temporary internal registers rather than to memory. This is the key difference between the stream processor abstraction and the vector processor abstraction [?].

Purcell et al.[?] implemented a ray tracing engine on graphics hardware by abstracting the hardware as a streaming graphics processor. Their paper divided up the raytracer into streams and kernels in order to map onto a theoretical architecture. They were also able to implement a prototype on the Radeon 9700, although several workarounds were needed. This work extends the streaming model presented in that paper by considering more general parallel applications and more general programming constructs.

2.4 Conclusions

From these observations, we can draw some overall conclusions relevant to developing programming model for computing with the GPU. First, GPUs are able outperform CPUs in terms of compute performance through parallelism. Second, the GPU's memory system is optimized for sequential access. Finally, there is a significant gap between the rate at which we can compute versus the rate at which we can read data into the registers. Any effective programming model for computing with the GPU must respect these characteristics. While a programming model alone cannot change an existing algorithm from sequential to parallel, or change the computation to bandwidth ratio, it can encourage or force the programmer to structure the computation to run efficiently on the GPU.

Chapter 3

Stream Programming Abstraction

In this chapter we define the stream programming abstraction and explain why streaming is an efficient abstraction over traditional SIMD or data parallel programming models for the GPU.

As the graphics processor evolves to include more complete instruction sets, data types, and memory operations, it appears more and more like a general-purpose processor. However, the challenge remains to introduce programmability without compromising performance, otherwise the GPU would become more like the CPU and lose its cost-performance advantages. In order to guide the mapping of new applications to graphics architectures, we propose to view programmable graphics hardware as a *streaming processor*. Stream processing is not a new idea. For example, media processors are able to transform streams of digital information as in MPEG video decode. The IMAGINE processor is an example of a general-purpose streaming processor [**?**] as discussed in chapter 2.

Stream computing differs from traditional computing in that the system reads the data required for a computation as a sequential *stream* of elements. Each element of a stream is a record of data requiring a similar computation, however it is, in general, independent of the other elements. The system executes a program or *kernel* on each element of the input stream then placing the result in an output stream. In this sense, a programmable graphics processor executes a vertex program on a stream of vertices, and a fragment program on a stream of fragments. Since for the most part we are ignoring vertex programs and rasterization, we are basically treating the graphics chip as a streaming fragment processor. We will revisit the vertex processor in chapter 5.

3.1 Advantages of Streaming

Before defining how we map the streaming model into a programming language and the GPU, it is important to understand why streaming is a model for computing with GPUs. While modern VLSI technology permits hardware to contain hundreds of floating point ALUs, there are several challenges to utilizing the hardware efficiently. First, the programmer needs to express enough operations to utilize all of the ALUs every cycle. Second, given the discrepancy in modern hardware between internal clock speeds and off-chip memory speed, memory access rates often dictate performance. Stream programs are better able to overcome these challenges through the use of data parallelism and arithmetic intensity. In this section we define these terms and explain how stream programs exploit these features to maintain peak performance on the GPU.

3.1.1 Data Parallelism

Programmable graphics hardware is an example of a data parallel architecture[**?**]. In this dissertation, we define a data parallel architecture as any architecture where the parallelism is based on applying operators to a collection of data records. Note that this definition encompasses both MIMD and SIMD architectures. Fortunately, in graphics, millions of vertices and billions of fragments are processed per second, so there is abundant data parallelism.

The current GPU programming model for vertex and fragment programs ensures that each program executes independently. The result of a computation on one vertex or fragment cannot effect the computation on another vertex or fragment. This independence has two benefits. First, the system can occupy all of the ALUs by executing multiple parallel copies of the program. Since each stream element's computation is independent from any other, designers can add additional pipelines that process elements of the stream in parallel and achieve immediate computational performance gains.

Second, and more subtly, the latency of any memory read operations can be hidden. In the classic vector processor, loads and stores are pipelined and overlapped with computation. In a multi-threaded implementation, when a memory access occurs, one thread is suspended and another is scheduled. This technique of hiding memory latency was first applied in graphics architectures to hide texture fetch latency [**?**; **?**; **?**]. In summary, the advantage of data parallelism is that it allows the system to use a large number of ALUs and to hide memory latency. Streaming allows for this data parallelism by requiring that the separate stream elements can be operated on independently.

3.1.2 Arithmetic Intensity

Data parallelism alone does not ensure that all of the hundreds of ALU units are occupied. If a program contains mostly memory operations, regardless of the amount of data parallelism, the performance of the program will be limited by the performance of the memory system. However, a program with a significant amount of computation using values in local registers will run efficiently.

In order to quantify this property, we introduce the concept of *arithmetic intensity.* Arithmetic intensity is the ratio of arithmetic operations performed per memory operation, in other words, flops per word transferred. As an example, in today's programmable fragment processor, the memory system permits one billion 128 bit words per second to be read while the computational rate is four times faster, permitting up to 4 billion 128-bit ops per second. In order for a program to maintain a high computation rate, the arithmetic intensity must be greater than four since in the time one word is transferred, four operations can be performed. Based on hardware trends, in the future, the arithmetic intensity required to obtain peak efficiency will increase.

It is important to distinguish the property of arithmetic intensity from the *mechanism* of caching. Caches exploit arithmetic intensity because they reuse values.

However, they are only one mechanism. A large enough register set also provides a mechanism for exploiting locality. The problem with a cache in a streaming architecture is that there is little temporal locality; in a pure stream, every value is used only once. A major difference between a modern CPU and GPU is the ratio of VLSI area devoted to caching vs. ALU's: in a GPU, ALU's dominate, in a CPU, caches dominate. In our programming environment, we want the programmer to expose arithmetic intensity in a way that allows efficient implementation without relying on a cache unless absolutely necessary.

Kernels encourage the programmer to construct programs with high arithmetic intensity. For example, we can compare kernels with traditional vector processing. Kernels perform arbitrary function evaluation whereas vector operators consist of simple math operations. Vector operations always require temporaries to be read and written to a large vector register file. In contrast, kernels capture additional locality by storing temporaries in local register storage. By reducing bandwidth to main memory, arithmetic intensity is increased since only the final result of the kernel computation is written back to memory.

3.1.3 Explicit Communication

Another the key benefits of the stream programming model is that it makes explicit communication in a program at a much courser resolution that traditional languages. We have already discussed one way in which comminication is made explict by the way kernels are restricted from communicating values between stream elements. This allows us to execute the kernel in parallel over the stream elements. In traditional sequencial programming languages, like C, there many possible ways in which one part of a compuation can comminicate with another. Loop-carried state, pointer alaising, global variables are all examples of comminication in C where a compiler must analize when optimizing or parallelizing the code. In stream programming, we restrict the programmer to comunication through streams, which present a much more managable primative for our compiler analize. Additionally, since we have relaxed the exact order with which we read and write the stream values, we can optimize the communication to the most efficient access pattern for the memory system.

In summary, the streaming model which emphasizes data parallelism, arithmetic intensity, and comminucation to allow graphics hardware to exploit parallelism, to utilize bandwidth efficiently and hide memory latency. As shown in chapter 2, these properties allow streaming to scale with the performance trends of GPUs. GPU vendors continue to add more and more computational power in the form of additional parallel pipelines. Data parallelism allows us to keep all of those parallel pipelines busy and continue to utilize any additional pipelines in the future. The significant trend is the growing bandwidth gap. Kernel encourage programmers to write shaders with high arithmetic intensity to promote users be limited by compute performance not bandwidth. As a result, the streaming computing model for graphics hardware makes for efficient model for today's and tomorrow's GPUs.

3.2 Streaming vs. SIMD

The stream programming model captures computational locality not present in the SIMD or vector models through the use of streams and kernels. Many other researchers have proposed a SIMD programming model approach for computing on GPUs. In this section, we contrast the two programming models to emphasize why streaming is an effective model for GPU based computing.

In the SIMD model, the programmer assumes each instruction in a fragment shader is executed over *all* of the data before advancing to the next instruction. All fragments being rendered in a pass are executed simultaneously, stepping through the fragment shader in lock-step. This main benefit of the SIMD programming model is familiar to many programmers who have experience with SIMD architectures like the Connection Machine [?].

The misconception that GPUs also operate in this manner can lead to inefficient computation and performance on the GPU. For example, as discussed above, GPUs are able to hide the latency of texture fetch operation by suspending work on one fragment and switching to another. Therefore, if the arithmetic intensity of a program is high enough, the costs of these texture fetch operations are hidden. Trying to justify this performance characteristic is difficult with the SIMD model since all fragment instructions are modeled as executing simultaneously.

If we think of the GPU as a processor which iterates over the data with a small program, performance optimizations like hiding texture latency start making sense. Other performance differences like random access versus sequential memory access illustrate how the GPU is better characterized as streaming processor than a SIMD processor.

Chapter 4

Brook Stream Language

Brook is an extension of standard ANSI C and is designed to incorporate the ideas of data parallel computing and arithmetic intensity into a familiar and efficient streaming language. Brook was originally developed as a language for streaming processors such as Stanford's Merrimac streaming supercomputer [?], the Imagine processor [?], the UT Austin TRIPS processor [?], and the MIT Raw processor [?]. We have adapted Brook to the capabilities of graphics hardware which will be the focus of this dissertation. However, the same programming model is used for these other architectures.

Brook extends C to include simple data-parallel constructs, enabling the use of the GPU as a streaming co-processor. In comparison with existing high-level languages used for GPU programming, Brook provides the following abstractions.

- Memory is managed via streams: named, typed, and "shaped" data objects consisting of collections of records.
- Data-parallel operations executed on the GPU are specified as calls to parallel functions called kernels.
- Many-to-one reductions on stream elements are performed in parallel by reduction functions.

In this chapter, we explain the design goals surrounding Brook and the features of the language, as well as several examples demonstrating its effectiveness. In the following chapters, we explain how these language primitives are mapped to the hardware and illustrate their performance.

4.1 Design Goals

A variety of design goals are behind the development of the Brook language. The primary goal is to make explicit the stream programming model outlined in chapter **??**. This includes understanding how to map an existing algorithm to streaming hardware through the use of the language primitives as well as allowing us to explore the limits of the stream programming model as a compute platform.

Another goal of Brook is to simplify stream hardware programming, especially for the GPU. The native GPU programming environment abstracts the hardware as a rendering device, steeped with graphics primitives such as texture, frame buffers, pixels, and vertices. The user wishing to use the graphics hardware as a compute platform must be well versed in the latest graphics APIs and must be willing to conceptually map their algorithm to on top of these primitives. Brook abstracts the GPU not as a rendering *device* but as a streaming *co-processor* void of any graphics constructs.

One of the main benefits of implementing an application on streaming hardware is its improved performance over conventional architectures. Brook should not inhibit the programmer from obtaining close to peak performance for a user's application compared to a hand-code low-level implementation.

Important features of the Brook language are discussed in the following sections.

4.2 Streams

A stream is a collection of data which can be operated on in parallel. Streams are declared with angle-bracket syntax similar to arrays, i.e. `float s<10,5>` which denotes a 2-dimensional stream of `float`s. Each stream is made up of *elements*. In this example, `s` is a stream consisting of 50 elements of type float. The *shape* of the stream refers to its dimensionality. In this example, `s` is a stream of shape 10 by 5. Streams are similar to C arrays, however, access to stream data is restricted to kernels (described below) and the `streamRead` and `streamWrite` operators, that transfer data between memory and streams.

Streams may contain elements of any legal C type as well as Cg vector types such as `float2`, `float3`, and `float4`. For example, a stream of rays can be defined as:

```
typedef struct ray_t {
    float3 o;
    float3 d;
    float tmax;
} Ray;
Ray r <100>;
```

Data in streams is stored in "row-major ordering" similar to C

arrays [?]. For example, the elements of the stream `float a<3,2>` is ordered a[0][0], a[0][1], a[1][0], a[1][1], ... This ordering is primarily relevant when using the different stream operators, in particular when transferring user data into and out of streams.

Support for user-defined memory types (i.e. `structs`), though common in general-purpose languages, is a feature not found in today's graphics APIs. In Cg, a programmer can define their own structures which are identical to C structures. However, loading the data from the host processor to the GPU must be done using only the few native types through the Cg binding mechanism and texture loads. For example, a Cg declaration:

```
struct particle {
    float2 pos :  TEXCOORD0;
    float3 vel :  TEXCOORD1;
    float mass :  COLOR0;
}
```

The Cg programmer must manually bind the different native types to the different inputs. Furthermore, there is no type checking of the data. Brook provides the user with the convenience of complex data structures and compile-time type checking.

Many parallel programming languages have introduced first-class arrays similar to streams as a language primitive. In many ways, streams are more restrictive than other array based languages. Brook prevents arbitrary access to array elements requiring separate read and write operators for the data movement. These restrictions, however, greatly assist the compiler and runtime system in managing stream data. Since access to the stream data is limited to bulk reads and writes, the stream data can persist in a separate memory space closer to the stream units, such as the texture memory on the GPU. Furthermore, many array-based parallel language compilers need to detect accesses to array elements since these can introduce synchronization points within the code. By preventing the user from individually access stream elements, we can permit the operations on the streams to proceed independently of the surrounding C code.

4.3 Kernels

Brook kernels are special functions, specified by the `kernel` keyword, which operate on streams. Calling a kernel on a stream performs an implicit loop over the elements of the stream, invoking the body of the kernel for each element. An example kernel is shown below.

```
kernel void saxpy (float a, float4 x<>, float4 y<>,
       out float4 result<>) {
    result = a*x + y;
}

void main (void) {
    float a;
    float4 X[100], Y[100], Result[100];
    float4 x<100>, y<100>, result<100>;

    ...  initialize a, X, Y ...

    // copy data from mem to stream
    streamRead(x, X);
    streamRead(y, Y);

    // execute kernel on all elements
    saxpy(a, x, y, result);

    // Equivalant C code:
    // for (i=0; i<100; i++)
    // result[i] = a*x[i] + y[i];

    // copy data from stream to mem
    streamWrite(result, Result);
}
```

Kernels accept several types of arguments:

- Input streams that contain read-only data for kernel processing.
- Output streams, specified by the `out` keyword, that store the result of the kernel computation. Brook imposes no limit on the number of output streams a kernel may have.
- Gather streams, specified by the C array syntax (`array[]`): Gather streams permit arbitrary indexing to retrieve stream elements. In a kernel, elements are fetched, or "gathered", via the array index operator, i.e. `array[i]`. Like regular input streams, gather streams are read-only.

- All non-stream arguments are read-only constants.

If a kernel is called with input and output streams of differing shape, Brook implicitly resizes each input stream to match the shape of the output. This is done by either repeating (`123` to `111222333`) or striding (`123456789` to `13579`) elements in each dimension. If there are multiple output streams, they must all be of the same size, otherwise, a runtime error is generated.

Certain restrictions are placed on kernels to allow data-parallel execution. Indirect memory access is limited to reads from gather streams, similar to a texture fetch. Operations that may introduce side-effects between stream elements, such as writing static or global variables, are not allowed in kernels. Streams are allowed to be both input and output arguments to the same kernel (in-place computation) provided they are not also used as gather streams in the kernel.

A sample kernel which computes a ray-triangle intersection is shown below.

```
kernel void krnIntersectTriangle(Ray ray<>, Triangle tris[],
       RayState oldraystate<>, GridTrilist trilist[],
       out Hit candidatehit<>) {
    float idx, det, inv_det;
    float3 edge1, edge2, pvec, tvec, qvec;
    if(oldraystate.state.y > 0) {
       idx = trilist[oldraystate.state.w].trinum;
       edge1 = tris[idx].v1 - tris[idx].v0;
       edge2 = tris[idx].v2 - tris[idx].v0;
       pvec = cross(ray.d, edge2);
       det = dot(edge1, pvec);
       inv_det = 1.0f/det;
       tvec = ray.o - tris[idx].v0;
       candidatehit.data.y = dot( tvec, pvec ) * inv_det;
       qvec = cross( tvec, edge1 );
       candidatehit.data.z = dot( ray.d, qvec ) * inv_det;
       candidatehit.data.x = dot( edge2, qvec ) * inv_det;
       candidatehit.data.w = idx;
    } else {
       candidatehit.data = float4(0,0,0,-1);
    }
}
```

Compared to today's GPU graphics languages, kernels are a generalization of shader functions. Instead of operating on input fragments to produce an output color, the kernel operates directly on user data. Brook requires the programmer to distinguish between data streamed to a kernel as an input stream and that which is gathered by the kernel using array access. This distinction permits the system to manage these streams differently. Input stream elements are accessed in a regular pattern but are never reused since each kernel body invocation operates on a different stream element. Gather streams may be accessed randomly, and elements may be reused. Today's graphics hardware makes no distinction between these two memory-access types. As a result, input stream data can pollute a traditional cache and penalize locality in gather operations.

Furthermore, the stream primative encourages programmers to structure their computation is manner which is effecient for stream processor memory systems. As illustrated in chapter 2, the GPU is optimized for sequencial memory access with an order of magnatude performance difference between sequencial verses random memory access.

A kernel function call is similar to the functional programming *map* operation. A map operation applies an operator to each element of a collection[**?**]. Similar to function programming semantics, kernel execution cannot have any side effects since we have forbidden global memory writes and persistent storage between processes stream elements. Therefore calling a kernel function is equivalent to mapping the kernel function to a the elements in a stream.

The use of kernels as functions differentiates stream programming from vector programming. Kernels perform arbitrary function evaluation whereas vector operators consist of simple math operations. Vector operations always require temporaries to be read and written to a large vector register file. In contrast, kernels capture additional locality by storing temporaries in local register storage. By reducing bandwidth to main memory, arithmetic intensity is increased since only the final result of the kernel computation is written back to memory.

It is important to recognize that Brook does not impose any specific limits on kernels such as amount of local state, use of conditionals and control flow, size of a kernel function, or the number of inputs and outputs. Some streaming architectures may be limited in ways which could prevent a straightforward execution of a non-trivial kernel function. It is the job of the Brook compiler to map the kernel function onto the streaming processor, potentially requiring complex transformations of the original specified kernel operation. We explore the challenge of mapping kernels to the stream processor in chapter **??**.

4.4 Reductions

While kernels provide a mechanism for applying a function to a set of data, reductions provide a data-parallel method for calculating a single value from a set of records. Examples of reduction operations include arithmetic sum, computing a maximum, and matrix product. In order to perform the reduction in parallel, we require the reduction operation to be associative: $(a \circ b) \circ c = a \circ (b \circ c)$. This allows the system to evaluate the reduction in whichever order is best suited for the underlying architecture. A simple sum reduction kernels is shown below:

```
reduce sum (float a<>, reduce float r<>) {
    r += c;
}

float r;
float a<100>;
// Compute the sum of all the elements of a
sum (a, r);

Equivalant C code:
    r = a[0];
    for (i=1; i<100; i++)
       r += a[i];
```

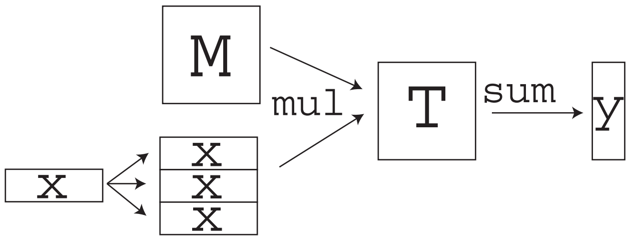
Reductions accept a single input stream and produce as output either a smaller stream of the same type, or a single-element value. Outputs for reductions are specified with the `reduce` keyword. Both reading and writing to the reduce parameter is allowed when computing the reduction of the two values.

If the output argument to a reduction is a single element, it will receive the reduced value of all of the input stream's elements. If the argument is a stream, the shape of the input and output streams is used to determine how many neighboring elements of the input are reduced to produce each element of the output. For example, if the input stream is of type `float<100>` and the reduction argument is a stream of type `float<25>`, the result is a stream that consists of the sums of the every four elements in the input stream:

```
float a<100>;
float r<25>;
// Perform the reduction
sum (a, r);

Equivalant C code:
    for (i=0; i<25; i++) {
       r[i] = a[i*4];
       for (j=1; j<3; j++)
          r[i]+=a[i*4 + j];
    }
```

The example below demonstrates how stream-to-stream reductions can be used to perform the matrix-vector multiplication $y = Ax$.

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

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

float A<50,50>;
float x<1,50>;
float T<50,50>;
float y<50,1>;
...
mul(A,x,T);
sum(T,y);
```



In this example, we first multiply `A` by `x` with the `mul` kernel. Since `x` is smaller than `T` in the first dimension, the elements of `x` are repeated in that dimension to create a matrix of equal size of `T`. The `sum` reduction then reduces rows of `T` because of the difference in size of the second dimension of `T` and `y`.

Today's GPU programming languages do not provide a fully general reduction mechanism similar to reduction kernels. Rarely is it necessary in a graphics application to numerically combine a set of data into a single value. The closest parallel is frame buffer blending in which an incoming fragment is blended with the value stored in the frame buffer often for transparency effects. However, fully programmatic blending operations have only recently become available and shading languages have yet to expose this functionality.

In relation to functional programming, reductions are similar to a *fold* operation. A fold operation consists of applying an operator to set a data to produce a single result, i.e. `r = fold(+, 0, input)` where `+` is the operator to apply, `0` is the initial value, and `input` are the elements to operate on. One significant difference between fold and reduction is that the reduction operator requires that the type of reduction variable is equivalent to the input data while fold relaxes this constraint. For example, assume we wanted to know the number of occurrences of the letter `c` in a string. With a fold operator, this calculation can be expressed as:

```
count = fold(incC, 0, string)
    where:
       int incC (int count, char value) {
          if (value == 'c') return count+1;
          return count;
       }
```

Here, the initial value and return type are integers while the input type is a character. The difficulty with parallelizing the fold operator is that although we can issue multiple copies of the incC operator, the programmer has not specified how to combine the return values to produce a single value. In this simple example, it would be possible for a compiler to extract this information. However, with a more complex reduction operator, this information could be difficult to extract. Furthermore, the programmer can easily insert dependencies which can make compiler generated parallelization impossible. In contrast, reduction operators only define how to combine the partial results in a parallel way. A Brook program performs the above fold operator in two passes.

```
kernel void isC(char s<>, out int v<>) {
    v = (s=='c')?  1 :  0;
}

reduce void sum(int a<>, reduce int b<>) {
    b += a;
}

isC(string, a);
sum(a, count);
```

The first kernel call creates a 0:1 stream based on the string which is then reduced by the sum kernel which yield the total number of elements. Although the fold operator produces the same result with a single operator, Brook's reductions ensure parallel execution.

Reductions are not new to programming languages. The C* language supported a collection of binary associative operators to reduce collection of elements. These included `+=`, `*=`, `/=`, etc. Brook reduction operators are similar to C*'s reduction operations, however, Brook permits the user to define their own reduction operator. By permitting user defined operators, Brook encourages additional arithmetic intensity in user programs rather than only using math operators.

4.5 Indexof and Iterator Streams

It is often useful when writing stream programs for a kernel to know which element of an input stream it is operating on. Typically, this information is relevant when an algorithm relies on the order of the elements. A simple example is a sorting algorithm which can use an element's position to determine which other element to compare against. The `indexof` operator may be called on an input or output stream inside a kernel to obtain the position of the current element within the stream. The return type of the `indexof` operator is always a `float4` type where unused dimensions of the stream have an index of 0.

```
kernel (float a<>, float array[], out float b<>) {
    float4 i = indexof(a);
    b = a + array[i.x];
    }
```

Iterator streams provide another low cost mechanism for generating ordered indices. Iterators are specially-typed streams containing pre-initialized sequential values specified by the user. Constructing an iterator stream uses the `iter` keyword and operator:

```
iter float s<100> = iter(0.0f, 100.0f);
// s initialized with 0.0, 1.0, 2.0, ..., 99.0
```

The first argument of the iter operator is always the initial value of the stream. The second operator is the upper bound of the stream, producing the uniform distribution [initial, upper bound). The step size between stream elements is equal to (initial - upper bound) divided by the number of elements in the stream.

```
iter float s<100> = iter(0.0, 1.0f);
// s:  0.00, 0.01, 0.02, 0.03, ..., 0.99
```

Both the indexof operator and the iterator streams provide convenient mechanisms for algorithms which rely on the ordered stream elements as a data structure. Both of these operators are fairly low overhead. The indexof operator simply accesses the stream address values which has already been computed by the GPU. Iterator streams leverage the existing interpolant hardware used for generating texture coordinates.

4.6 Stream Domain

Executing a stream on a kernel applies the kernel function to all the elements of a stream. However, many algorithms perform operations on only a portion of a data structure. For example, a PDE code may apply a very different operation on the boundary (the edge elements of a stream) than the interior region. While we could test for boundary conditions in the kernel, a more efficient implementation executes separate kernels at the boundries. This avoids the cost of evaluating the conditional as well as maximizing the SIMD execution of the kernel function on the GPU.

Brook provides a *Stream domain* operator to select a contiguous region of the stream to be operated on.

```
float s<100>;
float t<30,20>;

// Execute kernel on elements 3 to 49
mykernel(s.domain(3, 50));

// Execute kernel on 2D sub-region of t
mykernel(t.domain(int2(3,5), int2(15,10))
```

The domain operator returns a reference region of the stream which can be passed to a kernel function. Domain operators are also useful for `streamRead` and `streamWrite` operations for reading or writing portions of the stream back to the host processor.

4.7 Vout: Variable output streams

For each element consumed from the input streams in the basic kernel execution model, a single output element is emitted. This basic operation provides the foundation for most algorithms implemented in Brook. However, there are algorithms which are difficult to implement with one input resulting in exactly one output; specifically, applications which perform either data filtering or amplification. Filtering allows the application developer to decide programmatically whether or not to produce an output on a per element basis. Amplification permits the programmer to produce a data-dependent variable number of output elements for each element consumed.

There are a variety of algorithms for which filtering and amplification are essential. For example, the marching cubes algorithm [?] can produce zero (data filtering) to five triangles (data amplification) per voxel. Other applications which perform filtering and amplification include collision detection and adaptive subdivision which both have significant applications to both scientific and consumer (game) rendering. Some have proposed dedicated hardware for accelerating these algorithms [?; ?; ?].

Variable output streams are specified with the `vout` keyword. First, the kernel function assigns a value to the variable similar to fixed output streams. These assignments, however, do not produce an element onto the stream, nor is the variable automatically placed onto the output stream as per fixed output streams. Rather, the function must perform a `push` operation which copies the contents of the variable to the output stream as shown below.

```
void kernel foo (int a<>,
    vout int c<>) {
    int i;
    for (i=0; i<a; i++) {
       c = i;
       push(c);
    }
}
```

In the above example, the variable output stream c results in a variable number of output elements based on the contents of the stream a. The contents of the argument are not modified by the push and the kernel is free to read and write from the variable with the same caveats as fixed output streams. The push operator can only be used inside kernels and on variable output streams. It is the responsibility of the programmer not to overflow the number of declared elements in the output stream. The values of the unused elements of the stream are undefined.

There are a number of ways to define the ordering semantics of the final collection of output elements. On one end of the design spectrum outputs must be strictly ordered by the index of the fragments that created them. On the other end lies the completely unordered case: pushed values are packed in the stream but completely unordered with respect to one another. In the middle is another possibility - allowing the pushed values per input element to remain contiguous but unordered with respect to a larger input element ordering. Many algorithms which require vout functionality do not rely on the element ordering. However, useful procedures could exist that depend on proper ordering. For instance, given a list of translucent triangles sorted by depth to be tessellated, the programmer may wish the output of the tessellation to be ordered similarly for blending purposes. Brook currently maintains the order of the variable output stream elements. Where in the ordering design spectrum the optimal variable output ordering constraint lies remains an open question.

4.8 ScatterOp and GatherOp

The scatter and gather operations provide levels of indirection in reading and writing data, similar to the scatter/gather capability of the first vector machines. Brook purposefully separates scatter operations from gathers in order to maintain data parallelism. If we permitted writes and reads to arbitrary elements inside of kernels, we would introduce dependencies between stream elements. Allowing multiple gathers inside of kernel functions permits the programmer the flexibility to walk complex data structures and traverse data arbitrarily. Simple gather operations are supported within the kernel via the array syntax.

The opposite of a gather is a scatter operation. A scatter performs a parallel write operation with reduction (e.g. `p[i]+=a`). Brook provides a native operator for scatter operations:

streamScatterOp(dst, index, data, func);

The `streamScatterOp` function takes four arguments: a data stream containing data to scatter, an index stream specifying where the data is to be scattered, and the destination stream in which the data is written. The final argument is a user-specified `reduce` function used for combining the data to be written with the data present at destination. This includes collisions within the index stream. For example, a scatter to the same location is identical to a reduction operation. The streaming hardware is free reorder the kernel evaluation preserving associativity. If the reduce kernel argument is `NULL`, a simple "replace" reduction is performed, where only the last element to be scattered to a location is written. We also provide predefined reduce kernels, such as `SCATTER_ADD`, which have been optimized for the underlying streaming hardware.

In addition to gathers inside of kernels, the Brook API also includes `streamGatherOp` operator which performs a parallel indirect read with update (e.g. `a=p[i]++`). The arguments are similar to the `streamScatterOp` operator.

streamGatherOp(dst, index, data, func);

where `index` contains the indices to fetch from the source stream `data` which are placed into the `dst` stream. Here the `func` reduction is applied to each element fetched and atomically updates the element in the source `data` stream. GatherOp is quite useful when working data structures with a streaming processor.

Graphics hardware supports similar reductions and atomic gather operations in the read-modify-write portion of the fragment pipeline. The stencil buffer, z-buffer, and blending units perform simplified versions of these operators. Utilizing these graphics features for computation, however, can be awkward and indirect and do not allow user-specified code. ScatterOp and GatherOp offer a generalized version of this functionality.

4.9 Summary

In this chapter, we had described the Brook stream programming language. The main contribution of the Brook language is to make explicit the stream programming as described in chapter 3. By structuring computation with kernel and streams, we capture the main benefits of the stream program abstraction: data parallelism, artihmetic intensity, and explicit communication. By expressing computation in kernel functions which are restricted from introducing dependancies between stream elements, we permit the streaming processor to execute the kernel functions in a data parallel fasion.

Secondly, where traditional data parallel languages specify parallel operations at the level of individual math operations, Brook programs specifiy entire functions to operate in parallel. This encourages the programmer to write code with high arithemetic intensity since the operations and memory local to the computation can be performed locally on-chip, minimizing global memory traffic.

Finally, Brook makes requires the programmer to make explicit communication within an algorith. Communication that poses challenges for traditional sequencial languages such as pointer aliasing and loop carried dependencies are not present in Brook due to the usage of streams and the restrictions placed on kernel functions. In addition, Brook distinguishes between memory which is accessed sequencially verses randomly. This encourages programmers to structure computation around the optimal communication pattern for streaming processors.

The other main contribution of the Brook language is that it completely abstracts the GPU as a streaming co-procesor rather than a rendering device. Executing code on the GPU is as simple as calling a kernel function. This allows developers to think less about how to map their algorithm to graphics primatives such as textures, fragments, and framebuffers, rather focus on how it can be mapped to the more fundamental stream programming model.

Chapter 5

Targeting Stream Computing to GPUs

In this chapter, we demonstrate how we can map the streaming programming model presented in the Brook programming language to modern GPUs. Controlling a GPU today is done through the graphics API, primarily DirectX or OpenGL. These APIs expose the GPU as a rendering device communicating via primitives such as textures, triangles, and pixels. Mapping an algorithm to use these primitives is not a straightforward operation, even for the most advanced graphics developers. One practical benefit for implementing Brook on GPUs is to abstract the underlying graphics primitives and operations so that any programmer familiar with C can write a GPU-based application.

The primary goal of implementing Brook on GPUs is to demonstrate that today's GPUs are streaming processors disguised as rendering devices. This chapter illustrates how we can efficiently map Brook's computation primitives, such as streams and kernels, on top of existing GPU mechanisms. Furthermore, there are hardware limitations and constraints which present some challenges when exposing the GPU as a general purpose compute platform. We illustrate how our Brook compiler and runtime system can virtualize many of these constraints to present a generalized compute platform.

5.1 System Outline

The Brook compilation and runtime system maps the Brook language onto existing programmable GPU APIs. The system consists of two components: `brcc`, a source-to-source compiler, and the Brook Runtime (BRT), a library that provides runtime support for kernel execution and stream management. The compiler is based on cTool [?], an open-source C parser, which was modified to support Brook language primitives. The compiler maps Brook kernels into Cg shaders which are translated by vendor-provided shader compilers into GPU assembly. Additionally, `brcc` emits C++ code which uses the BRT to invoke the kernels. Appendix **??** provides a before-and-after example of a compiled kernel.

BRT is an architecture-independent software layer which provides a common interface for each of the backends supported by the compiler. Brook currently supports three backends: an OpenGL and DirectX backend and a reference CPU implementation. Creating a cross-platform implementation provides three main benefits. First, we demonstrate the portability of the language by allowing the user to choose the best backend for the hardware. Second, we can compare the performance of the different graphics APIs for GPU computing. Finally, we can optimize for API-specific features, such as OpenGL's support of 0 to n texture addressing and DirectX's render-to-texture functionality.

The following sections describe how Brook maps the Brook language primitives onto the GPU.

5.2 Mapping Streams and Kernels

Brook represents streams as floating point textures on the graphics hardware. With this representation, the `streamRead` and `streamWrite` operators upload and download texture data, gather operations are implemented as dependent texture reads, and the implicit repeat and stride operators are achieved with texture sampling.

With stream data stored in textures, Brook uses the GPU's fragment processor to execute a kernel function over the stream elements. `brcc` compiles the body of a kernel into a Cg shader. Stream arguments are initialized from textures, gather operations are replaced with texture fetches, and non-stream arguments are passed via constant registers. The NVIDIA or Microsoft shader compiler is then applied to the resulting Cg code to produce GPU assembly. A stub function replaces the original kernel prototype which performs all of the BRT function calls to load and issue the compiled shader.

To execute a kernel, the BRT issues a single quad containing the same number of fragments as elements in the output stream. The kernel outputs are rendered into the current render targets. The DirectX backend renders directly into the textures containing output stream data. OpenGL, however, does not provide a lightweight mechanism for binding textures as render targets. OpenGL Pbuffers provide this functionality, however, as Bolz et al.[**?**] discovered, switching between render targets with Pbuffers can have significant performance penalties. Therefore, our OpenGL backend renders to a single floating-point Pbuffer and copies the results to the output stream's texture. The proposed Superbuffer specification [**?**], which permits direct render-to-texture functionality under OpenGL, should alleviate this restriction.

5.3 Virtualizing Hardware constraints

There are numerous hardware limits of today's graphics hardware that impose significant constraints on the Brook programming environment. These include limits on the sizes and formats of textures, the number of inputs and outputs supported by the fragment processor, numerical precision, and the length of fragment programs. In many cases, these constraints, while reasonable in a graphics application, seem arbitrary and overbearing in a general computing context. For example, today's graphics cards support textures of up to 4096 texels in each dimension [1]. It is rare that a texture for a graphics application exceeds this size. However, since Brook represents streams as textures, a 1D stream cannot have a length longer than 4K, which is clearly constraining. In this section, we explore ways to virtualize many of these constraints in the current graphics hardware.

[1]on current NVIDIA hardware

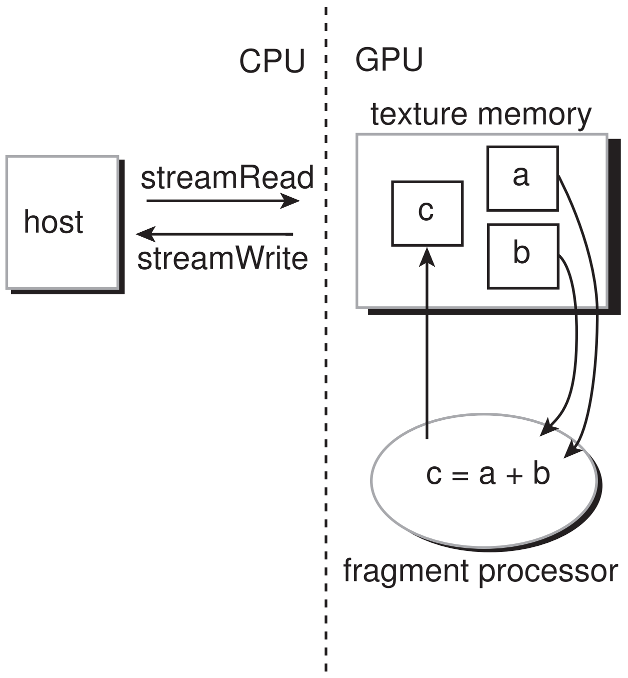


Figure 5.1: A block diagram

5.3.1 Stream Shape and Length

Floating-point textures are limited to two dimensions and a maximum size of 4096 by 4096 on NVIDIA or 2048 by 2048 on ATI hardware. As mentioned above, if we directly map stream shape to texture shape, then Brook programs cannot create streams of more than two dimensions or 1D streams of more than 2048 or 4096 elements.

To address this limitation, `brcc` provides a compiler option to wrap the stream data across multiple rows of a texture. This permits arbitrarily-sized streams assuming the total number of elements fits within a single texture. In order to access an element by its location in the stream, `brcc` inserts code to convert between the stream location and the corresponding texture coordinates. The Cg code shown below is used for stream-to-texture address translation and allows for streams of up to four dimensions containing as many elements as texels in a maximum sized 2D texture.

```
float2 calculatetexpos( float4 streamIndex,
    float4 linearizeConst, float2 reshapeConst ) {
    float linearIndex = dot( streamIndex, linearizeConst );
    float texX = frac( linearIndex );
    float texY = linearIndex - texX;
    return float2( texX, texY ) * reshapeConst;
}
```

Our address-translation implementation is limited by the precision available in the graphics hardware. In calculating a texture coordinate from a stream position, we convert the position to a scaled integer index. If the unscaled index exceeds the largest representable sequential integer in the graphics card's floating-point format (16,777,216 for NVIDIA's s23e8 format, 131,072 for ATI's 24-bit s16e7 format) then there is not sufficient precision to uniquely address the correct stream element. For example, our implementation effectively increases the maximum 1D stream size for a portable Brook program from 2048 to 131072 elements on ATI hardware. Ultimately, these limitations in texture addressing point to the need for a more general memory addressing model in future GPUs.

5.3.2 User-Defined Stream Types

Brook permits the user to define their own stream types via C's struct syntax. Graphics APIs, however, only provide `float`, `float2`, `float3` and `float4` texture formats. There are a variety of different ways we can represent user-defined stream types natively on the GPU. The Brook runtime could store the stream elements in a single packed texture (often referred to as the arrays of structs storage method), segmenting the different struct members into a single texture (struct of arrays), or maintaining separate textures for each struct member (also struct of arrays). The three options are shown in figure 5.3.2.

For a variety of reasons, Brook implements the multiple texture method for storing streams of structures. GPUs require that individual textures must consist of texels of the same type, i.e. all `float4` or `float2` etc, which can complicate the structures consisting of differing types. To implement the first two options, Brook could store the stream in single component float texture and perform multiple texture fetches for each

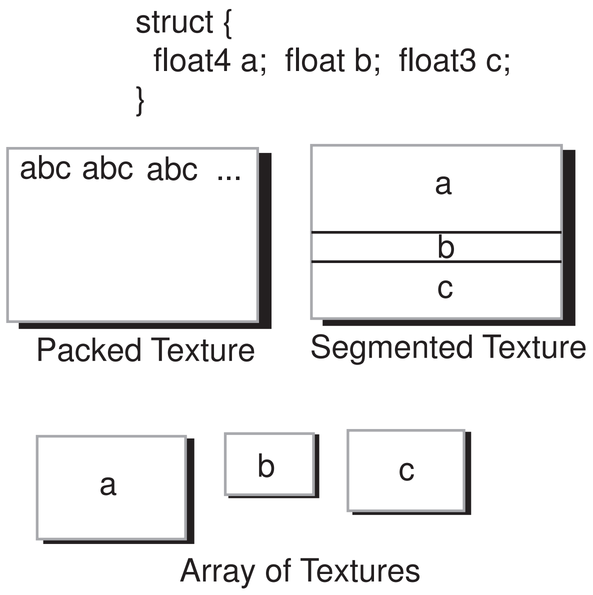


Figure 5.2: Different ways to do structs

component of a struct element. However, this approach would amplify the limited texture size constraint since the number of texels required to store the stream would be the total number of stream elements times the number of floats per element. The alternative is to pack struct elements into the larger texels (`float4`s) and introduce code into the shader to unpack the stream element. However, this approach is likely to additional fetch data not part of the stream element and waste bandwidth.

A further difficulty with packed and segmented textures is writing to these streams. Current graphics hardware only supports writing up to 4 separate textures per rendering pass with only a single `float4` write per texture. There is no direct method to perform multiple writes to the same texture within a shader. Writing to these packed or segmented textures would requires additional passes or copies to update all of the structure elements.

The array of textures approach does not suffer from the above limitations. To fetch the elements of the structure, we simply fetch from each texture. The texture coordinates of a stream element are the same for all of the textures. This approach does not require any packing or padding of the data and writing to the stream can be done directly as long as there are no more than 4 structure members (we address output virtualization in the next section). One limitation of this approach, however, is that we must do additional work during streamRead and streamWrite, since the user data presented to these operators is in C's array of structure layout. Additionally, by using multiple textures per stream, we are consuming additional texture inputs in the fragment program. An NVIDIA GeForce 6800 supports up to 16 texture inputs per fragment program. By splitting the stream into multiple textures, we are more likely to run out of texture inputs.

5.3.3 GPU Shader Constraints

The task of mapping kernels to fragment shaders is complicated by the fragment processors constraints. One of the most common limitations experienced by GPU programmers is the limited number of shader outputs available in today's hardware. While there is no inherent limit on the number of outputs of a kernel function, it is certainly possible to exceed the number of shader outputs supported. Also, as discussed in the previous section, stream outputs of structures must be broken into outputs for each struct member further consuming shader outputs. Today's GPUs typically only support up to four `float4` outputs. This output limit is often the most constraining resource within the GPU.

To alleviate this constraint, `brcc` splits kernels into multiple passes in order to compute all of the outputs. For each pass, the compiler produces a complete copy of the kernel code, but only assigns a subset of the kernel outputs to the shader outputs. To execute the kernel, the runtime system executes the set of shaders in separate passes. We take advantage of the aggressive dead-code elimination performed by today's shader compilers to remove any computation that does not contribute to the outputs written in that pass.

To test the effectiveness of our pass-splitting technique, we applied it to two kernels: Mat4Mult, which multiplies two streams of 4x4 matrices, producing a single 4x4 matrix (4 `float4`s) output stream; and Cloth, which simulates particle-based cloth with spring constraints, producing updated particle positions and velocities.

We tested two versions of each kernel. Mat4Mult4 and Cloth4 were compiled with hardware support for 4 `float4` outputs, requiring only a single pass to complete. The Mat4Mult1 and Cloth1 were compiled for hardware with only a single output, forcing the runtime to generate separate shaders for each output.

As shown in Table **??**, the effectiveness of this technique depends on the amount of shared computation between kernel outputs. For the Mat4Mult kernel, the computation can be cleanly separated for each output, and the shader compiler correctly identified that each row of the output matrix can be computed independently. Therefore, the total number of arithmetic operations required to compute the result does not differ between the 4-output and 1-output versions. However, the total number of texture loads does increase since each pass must load all 16 elements of one of the input matrices. For the Cloth kernel, the position and velocity outputs share much of the kernel code (a force calculation) which must be repeated if the outputs are to be computed in separate shaders. Thus, there are nearly twice as many instructions in the 1-output version as in the 4-output version. Both applications perform better with multiple-output support, demonstrating that our system efficiently utilizes multiple-output hardware, while transparently scaling to systems with only single-output support.

As shown above, this technique is effective for kernels where there is little dependence between shader outputs. The dead code elimination cannot eliminate the execution of code shared between outputs. Certainly a manual decomposition of the kernel could either group outputs which share code, or generate passes of intermediate data to minimize duplicate computation. These approaches have been explored by the graphics community with respect to shader decomposition for complex rendering shaders and have since been implemented in the Brook runtime system as presented in Foley et. al. [**?**]. This approach, called MRDS, performs a search of the different kernel decompositions from specific points in the shader DAG called dominators. Each pass can generate either outputs or temporaries to be used in future passes. By applying a cost function to potential splits, we can compute the optimal decomposition of the kernel. This MRDS algorithm can be applied to many of the other constraints present in shaders, including the number of instructions, input streams, and constant arguments.

5.3.4 Virtualization with Next Generation GPUs

There are some shader limitations which we do not virtualize. Currently, shaders do not support integer data types nor integer arithmetic. Brook programs targeted for the GPU must represent all data types with floating point data types and compute with floating point arithmetic only. While it is conceivably possible to emulate integer arithmetic with a floating-point only processor, native support for integers is expected to be available on next-generation GPUs. Likewise, not all of today's GPUs support branching inside of Shaders, however, this too is anticipated with next-generation hardware. The variety of branching techniques is explored in chapter **??**. Efforts spent on virtualizing these limitations would be quickly deemed irrelevant.

One limitation which Brook cannot solve through virtualization is total stream memory. Virtual memory is commonplace with today's CPUs with the hard drive providing additional storage when main memory becomes limited. Today's GPUs perform a limited version of virtual memory where as entire textures can be swapped in and out of video memory. However, this operation, performed by the graphics driver, is only effective at the granularity of an entire texture, i.e. if a texture is not used in a rendering pass, the texture may be bumped from the graphics card memory. This does not solve the problem of virtualizing the upper limit of the size of an individual texture and therefore stream size. Techniques which could resolve this constraint with existing hardware is left to future work. One hope is that future hardware will support exception handling which is the basis for many CPU virtual memory systems.

5.4 Reductions

Current graphics hardware does not have any capabilities to perform reduction directly. Furthermore, given the limited number of instructions allowed in a single shader, it is unlikely we can perform a reduction of an entire stream in a single pass. This section discusses the implementation and effectiveness of our multipass approach.

Consider a simple sum reduction of a 1D stream of length n to a single scalar result. The equivalent C code is:

```
r = a[0];
for (i = 1; i < n; i++)
    r = r + a[i];
```

A straightforward decomposition of this loop into multiple passes on the GPU would perform each iteration as a separate pass. Each pass would read the reduction value r and the next value from the a stream, perform the sum, and write the result. This yields a total of n passes, for a total of $2n$ reads, n writes, and n sum operations. One simple optimization would be to partially unroll the for loop to reduce the number of reads and writes:

```
for (i = 0; i < n; i+=k) {
    r = r + a[i];
    r = r + a[i+1];
    r = r + a[i+2];
    ...
    r = r + a[i+k];
}
```

In this example, we unroll the loop k times, where the loop body consists of the maximum number of operations allowed inside of a shader. Here we perform a total of n/k passes, for a total of $n + n/k$ reads, n/k writes, and n sum operations.

The downside to this approach is that each iteration of the loop is dependent on the previous iterations results. To implement this loop on the GPU, we must perform each loop iteration as a separate rendering pass, where each pass consists of rendering a single point, fetching the previous `r` value from texture, updating `r`, and writing the result. While this is a valid implementation, it does not take advantage of the parallel shading pipelines present in modern GPUs as discussed in chapter 2.

To benefit from the GPU's parallel architecture, we can restructure the computation to perform the reduction in parallel, with a method similar to Kruger and Westermann [?]. The reduction can be performed in $O(\log n)$ passes, where n is the ratio of the sizes of the input and output streams. For each pass, we reduce adjacent stream elements and output a new stream with fewer values.

```
t = a;
for (i = n; i > 1; i /= 2)
    for (j = 0 ; j < i/2; j++)
       t[j] = t[j*2]+t[j*2+1];
r = t[0];
```

Here, the inner `j` loop is updating multiple memory locations without any loop carried dependencies. We can parallelize this loop across multiple fragments by rendering a large quad similar to kernel execution. The outer `i` loop is implemented with separate passes. Compared to the previous techniques, we perform many fewer passes, $log_2 n$ rather than n passes. In the first pass, we read n elements, and write $n/2$ elements, decreasing the number of elements to be read in the next pass by a half. This yields a total of $2n$ reads, n writes, n total sum operations. Therefore, despite the logarithmic number of passes, these parallel reductions are a linear-time computation.

One final optimization is to unroll the loops similar as before to perform a larger amount of the reduction per pass:

```
t = a;
for (i = n; i > 1; i /= k)
    for (j = 0 ; j < i/k; j++) {
       r = t[j*k];
       r += t[j*k+1];
       r += t[j*k+2];
       ...
       r += t[j*k+(k-1)];
       t[j] = r
    }
r = t[0];
```

This further reduces the computation to $n+n/(k-1)$ reads, $n/(k-1)$ writes, and n total sum operations. A breakdown of the four implementations is shown below.

Implementation	Reads	Writes	Ops	Passes
Optimal	n	1	n	1
Simple loop	$2n$	n	n	n
Unrolled loop	$n + n/k$	n/k	n	n/k
Parallel loop	$2n$	n	n	$\log_2 n$
Unrolled parallel loop	$n + n/(k-1)$	$n/(k-1)$	n	$\log_k n$

As illustrated in this table, the main benefit of parallel reductions is the reduced number of passes. The cost of issuing a pass on the graphics hardware is relatively expensive since the CPU needs to set up texture coordinates, bind textures, and issue geometry. Furthermore, with OpenGL, we incur additional cost in copying the data from the Pbuffer to an input texture for reading in the next pass as described above. Unrolling the parallel loop yields the minimum number of reads, writes, and passes.

The unrolled parallel reduction code shown assumes that the input stream to be reduced is an integer power of k. To handle reductions of streams of arbitrary lengths, the Brook compiler generates shaders which can perform between 2-way and 8-way reductions in a single shader. We then apply the 8-way reduction on the first 8^w elements where w is $log_8 n$. The elements not reduced are carried over to the next pass and the process is repeated until there are less than 8 elements remaining, applying the smaller n-way reduction kernel to compute the final value. For multi-dimensional reductions, we perform the reduction in one of the dimensions followed by the next dimension. Of course, to perform these reductions in parallel, the reduction operator must be an associative operation, `(a op b) op c = a op (b op c)`. This is a requirement of the language, although there is no compile-time enforcement.

We have benchmarked computing the sum of 2^{20} `float4` elements as taking 2.4 and .79 milliseconds, respectively, on our NVIDIA and ATI DirectX backends and 4.1 and 1.3 milliseconds on the OpenGL backends. An optimized CPU implementation performed this reduction in 14.6 milliseconds. The performance difference between the DirectX and OpenGL implementations is largely due to the cost of copying results from the output Pbuffer to a texture, as described above. Though the GPU performs many more reads and writes than the optimal CPU implementation, the GPU memory bandwidth far exceeds the Pentium 4 memory bandwidth as discussed in chapter 2.

With our multipass implementation of reduction, the GPU must access significantly more memory than an optimized CPU implementation to reduce a stream. If graphics hardware provided a persistent register that could accumulate results across multiple fragments, we could reduce a stream to a single value in one pass. We simulated the performance of graphics hardware with this theoretical capability by measuring the time it takes to execute a kernel that reads a single stream element, adds it to a constant and issues a fragment kill to prevent any write operations. Benchmarking this kernel with DirectX on the same stream as above yields theoretical reduction times of .41 and .18 milliseconds on NVIDIA and ATI hardware respectively.

5.5 Stream Operators

In this section, we review some of the implementation issues regarding support for the Brook stream operators. We have already discussed the `streamRead` and `streamWrite` operators which copy data to and from host memory and textures. Below we discuss the implementation of the `domain`, `GatherOp`, and `ScatterOp` stream operators.

Domain

Implementation of the `domain` operator is relatively straightforward in the common case. To issue a kernel on a region of an output stream, we simply modify the render geometry to cover only the portion of the stream's texture which was specified with the domain operator. Likewise, to operate on the region of an input stream, we modify the interpolant values to correspond to the correct portion of the stream texture. In general, the domain operator is relatively straightforward to implement since it always specifies a contiguous region of a stream and therefore its corresponding texture.

The domain operator is somewhat complicated when combined with address translation. Although the specified region of the stream to operate over is contiguous, the

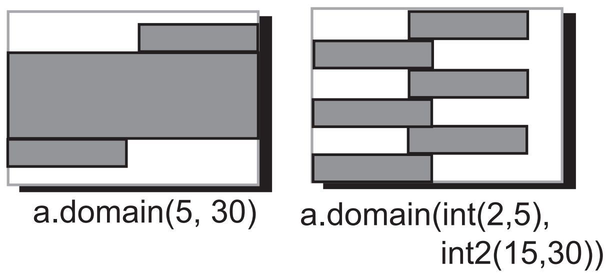


Figure 5.3: Address translation with Domain

region does not necessarily correspond to a contiguous region of the source texture. For input streams, we applied the address translation before fetching as before. The solution for output streams is a bit more complex since the GPU natively can only output to contiguous 2D regions. One solution is to apply the kernel function for all stream elements and predicate the computation if the domain operator does not include that region of the stream. While this solution is fairly simple to implement, it does incur the performance penalty of executing the kernel over unnecessary elements.

An alternative solution is to divide the computation into contiguous regions and issue the corresponding geometry. For large 1D streams, we can segment the computation into up to three regions to cover all the elements of the domain. While this approach works for 1D streams, virtualized n-D streams can have many regions which could make this approach limited by the geometry issue rate as shown in figure 5.5.

For simplicity and correctness, the current Brook implementation performs predication by evaluting the kernel function over all of the stream elements and aborts the computation with `kill` instruction is the fragment is found to be outside of the domain region. Efficient domain implementation in the presence of address translation is left to future work.

ScatterOp and GatherOp

The ScatterOp and GatherOp operators in Brook require atomic read/modify/write semantics for the destination and source streams respectively. We cannot support these semantics on current hardware as neither NVIDIA nor ATI supports programmable blending to floating-point render targets. Our current system implements ScatterOp and GatherOp by reading stream data back to host memory and performing the operation sequentially with the CPU. ScatterOp is, however, a generalization of the fixed-function blending capabilities of today's hardware and easily could be implemented with future hardware.

An alternative solution is to use the vertex pipeline to perform the scatter operation. First, the destination stream is placed into an off-screen Pbuffer. Next, the system fetches the index stream from the 2D texture. Using the index data, we render OpenGL points positioned corresponding to the index value. With the user specified ScatterOp kernel bound as a fragment shader, the point performs the reduction by fetching the two values to reduce from texture and writes the result into the Pbuffers. Once all of the points have been rendered, the data is scattered into the destination stream present in the Pbuffers. A GatherOp works in a similar manner except it requires additional passes: one to write the gathered value and a further pass to execute the gather kernel. Graphics hardware that supports multiple render outputs per pass can avoid this extra cost. This method does have some significant performance drawbacks which make it unattractive for today's hardware. First, the index stream must be read back to host memory, a costly operation. Also, the graphics system must issue a point per index element and can be limited by the geometry issue rate of the card.

The implementation of GatherOp and ScatterOp is further complicated by index streams with multiple references to the same stream element. Current GPUs do not allow reading and writing to the same Pbuffer. If multiple references exist to the same location, GatherOp and ScatterOp cannot execute in a single pass. Rather, the index stream is divided into portions which do not contain multiple references. These separate streams are then executed in different passes.

This inefficiency could be eliminated with the introduction of programmable blending. Current blending already supports similar semantics for resolving writes to the same location. Existing OpenGL ordering semantics requires that blending operations are applied in the order in which the primitives were issued by the application.

By extending blending operations to be programmable, the ScatterOp kernel could perform the reduction in the blending hardware, requiring only a single pass.

5.6 Conclusion

The main contribution of this section is to demonstrate how we can implement the streaming programming model with today's GPUs and to discuss some of the challenges and solutions in doing so. The fragment processor provides a compute platform which can read from memory both sequentially and randomly, execute our kernel and reduction code, and output to contiguous regions of memory into our output streams. There are some challenges with using the fragment processor as a compute device, mainly the limits on instruction count, number input and output streams, and data-types. Most of these constraints can be virtualized as long as we can decompose our language primitives into operations which write into contiguous regions of memory. Representing streams as textures is a natural fit, although there are some constraints which require virtualization. Those few aspects of the GPU which cannot be virtualized limit the GPUs capabilities as a general purpose streaming processor. Improvements to the hardware may provide the optimal solution for overcoming these limitations. Specific features may include integer data types, bit operations, and exception handling for better stream virtualization. Other GPU limitations can be hidden through CPU emulation, including ScatterOp, GatherOp, and domain with address translation. The language primitives which are quite fundamental to streaming highlight the GPU's need for improvement in more flexible memory access capabilities and programmable blending. However, with CPU emulation, we can explore the effectiveness of these features assuming future hardware support.

Chapter 6

Evaluating Stream Computing on GPUs

In this chapter we explore the effectiveness of the GPU as a streaming processor. Our analysis consists of three main components. First we explore some of the basic properties pertaining to the GPU as a streaming processor. Second, we provide a GPU vs. CPU performance case study ofa collection of applications. From these performance results, we can observe properties which will contribute to an algorithm executing efficiently on the GPU. Finally, in the next chapter, we provide an algorithmic case study illustrating some of the challenges and solutions for implementing applications on GPUs with the streaming programming model.

6.1 GPUs as a streaming processor

As we have shown in earlier chapters, we can map the streaming processor computing model on top of existing GPU architectures. In this section, we explore the effectiveness of the GPU as a streaming machine and compare it to its commodity competitor, the desktop CPU. As discussed in chapter **??**, CPUs and GPUs present very different programming abstractions as well as performance characteristics. However, if we examine these differences closely with respect to streaming, we can draw some general conclusions about computing with the GPU.

6.1.1 Computational Intensity

The general structure of many Brook applications consists of copying data to the GPU with `streamRead`, performing a sequence of kernel calls, and copying the result back to the CPU with `streamWrite`. Executing the same computation on the CPU does not require these extra data transfer operations. Considering the cost of the transfer can affect whether the GPU will outperform the CPU for a particular algorithm. In general, applications which spend a significant amount of time computing on the GPU compared to the time of transferring data to and from the host system should favor the GPU in performance. We call this property the *computational intensity* of the algorithm.

To study this effect, we consider a program which downloads n records to the GPU, executes a kernel on all n records, and reads back the results. The time taken to perform this operation on the GPU and CPU is:

$$\begin{aligned} T_{gpu} &= n(T_r + K_{gpu}) \\ T_{cpu} &= nK_{cpu} \end{aligned}$$

where T_{gpu} and T_{cpu} are the running times on the GPU and CPU respectively, T_r is the transfer time associated with downloading and reading back a single record, and K_{gpu} and K_{cpu} are the times required to execute a given kernel on a single record. This simple execution time model assumes that at peak, kernel execution time and data transfer speed are linear in the total number of elements processed / transferred. The GPU will outperform the CPU when $T_{gpu} < T_{cpu}$. Using this relationship, we can show that:

$$T_r < K_{cpu} - K_{gpu}$$

As shown by this relation, the performance benefit of executing the kernel on the GPU ($K_{cpu} - K_{gpu}$) must be sufficient to hide the data transfer cost (T_r).

From this analysis, we can make a few basic conclusions about the types of algorithms which will benefit from executing on the GPU. First, the relative performance of the two platforms is clearly significant. The *speedup* is defined as time to execute a kernel on the CPU relative to the GPU, $s \equiv K_{cpu}/K_{gpu}$. The greater the speedup for a given kernel, the more likely it will perform better on the GPU. Secondly, an algorithm which performs a significant amount of computation relative to the time spent transferring data is likely to be dominated by the computation time. This relationship is the *computational intensity*, $\gamma \equiv K_{gpu}/T_r$, of the algorithm. The higher the computational intensity of an algorithm, the better suited it is for computing on the GPU. By substituting into the above relation, we can derive the relationship between speedup and computational intensity.

$$\gamma > \frac{1}{s-1}$$

The idea of computational intensity is similar to arithmetic intensity, defined by Dally et al. [?] to be the number of floating point operations per word read in a kernel. Computational intensity differs in that it considers the entire cost of executing an algorithm on a device versus the cost of transferring the data set to and from the device. Computational intensity is quite relevant to the GPU which generally does not operate in the same address space as the host processor.

6.1.2 Kernel Call Overhead

For our cost model, we assume that the parameters K_{gpu} and T_r are independent of the number of stream elements n. In reality, this assumption does not hold for short streams. GPUs are more efficient at transferring data in mid to large sized amounts. More importantly, there is overhead associated with issuing a kernel. Every kernel invocation incurs a certain fixed amount of CPU time to setup and issue the kernel on the GPU. With multiple back-to-back kernel calls, this setup cost on the CPU can overlap with kernel execution on the GPU. For kernels operating on large streams, the GPU will be the limiting factor. However, for kernels which operate on short

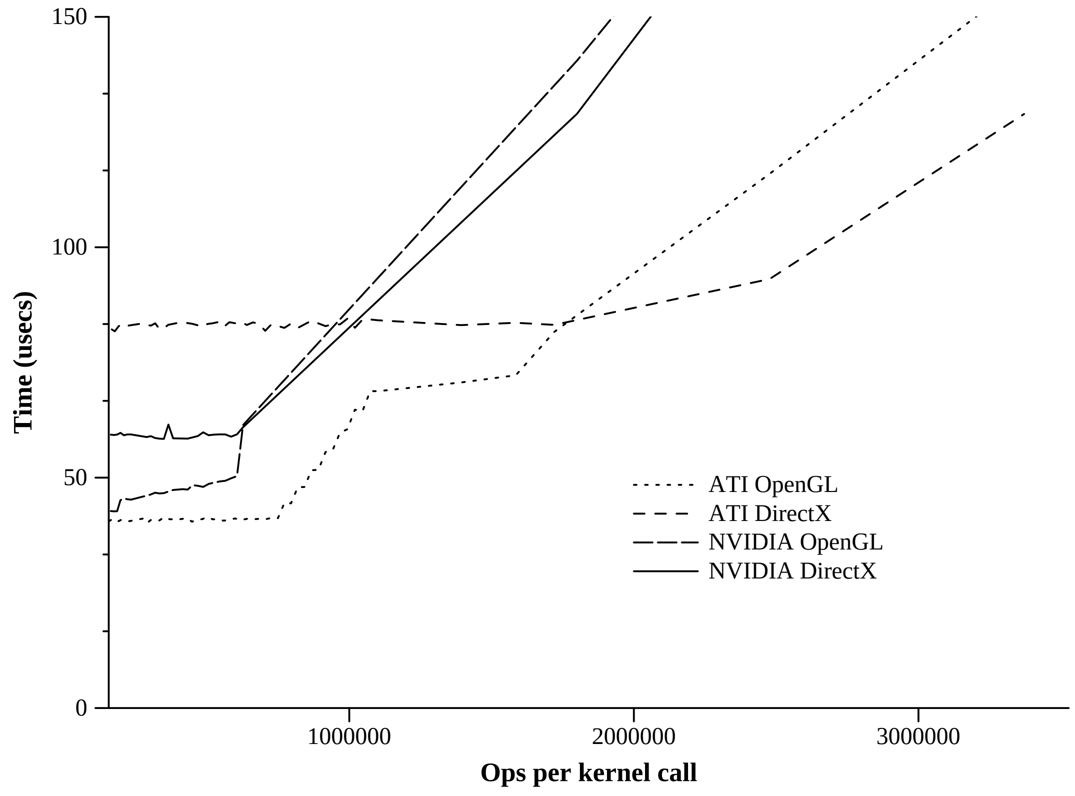


Figure 6.1: Kernel call overhead

The average cost of a kernel call for various stream lengths with our synthetic kernel. At small sizes, the fixed CPU cost to issue the kernel dominates total execution time. The stair-stepping is assumed to be an artifact of the rasterizer.

streams, the CPU may not be able to issue kernels fast enough to keep the GPU busy. Figure 6.1 shows the average execution time of 1,000 iterations of a synthetic kernel with the respective runtimes. As expected, both runtimes show a clear *knee* where issuing and running a kernel transitions from being limited by CPU setup to being limited by the GPU kernel execution. For our synthetic application which executes 43 MAD instructions, the ATI runtime crosses above the knee when executing over 750K and 2M floating point operations and NVIDIA crosses around 650K floating point operations for both OpenGL and DirectX.

6.1.3 Arithmetic Intensity

One disadvantage of this model is that it assumes K_{cpu} and K_{gpu} are known. Currently, there are no well established heuristics for predicting the execution time of a shader on the GPU other than benchmarking of the actual shader. With a better understanding of the performance model of today's GPUs, we could use our model to predict whether an algorithm will perform better on the GPU verses the CPU.

One critical aspect of establishing K_{gpu} is the *arithmetic intensity* of the algorithm. As discussed in chapter 2, the arithmetic intensity is ratio of the computation performed in an algorithm relative to the amount of memory read and written. Arithmetic intensity is similar to computational intensity however, arithmetic intensity examines the math verses memory instructions executed where computational intensity addresses the total compute time verses communication time with the GPU as a device. The stream programming model promotes high arithmetic intensity with the kernel function declaration. In general, applications with high arithmetic intensity are expected to perform well on the GPU due to its high computational performance.

To explore the performance impacts of arithmetic intensity, we created a synthetic workload to easily explore the trade-offs between computation and memory operations:

```
kernel void bench(float4 inStream[][],
    out float4 outStream<>) {
       // Perform memory fetch operations
       float4 a = inStream[...];
       float4 b = inStream[...];
       float4 c = inStream[...];

       // Perform MAD instructions
       a += b * c;
       a += a * a;
       a += a * a;
       ...
       outStream = a;

    }
```

`bench` is a simple kernel that perform a sequence of fetch operations followed by a sequence of math operations. By varying the number of fetch operations in relation to the number of math operations, we can examine the effects of increasing arithmetic intensity observing a performance shift from bandwidth to computation limited.

To simplify the analysis, we minimize the effects of computational intensity by not including time spent during streamRead and streamWrite. Furthermore, we execute our shader on a large enough stream to limit the effect of kernel call overhead.

INSERT GRAPH HERE.

6.1.4 Conclusions

Our analysis shows that there are three key aspects to using the GPU as a stream processor. First, GPUs do not operate within the same memory space as the host system. To counter the cost of transferring data to and from the GPU, the computational intensity of the algorithm, γ, required to outperform the CPU must be inversely proportional to the speedup achieved by the GPU. Second, the amount of word done per kernel call should be large enough to hide the setup cost required to issue the kernel. Third, in order to achieve the best possible speedup from stream computing on the GPU, our algorithms must exhibit high arithmetic intensity which is the ratio to computation to communication in a kernel, such that we are limited by the impressive compute rate of the GPU, not by the memory system. We anticipate that while the specific numbers presented in this thesis may vary with newer hardware, the computational intensity, arithmetic intensity, and kernel overhead will continue to dictate effectiveness of the GPU as a streaming processor.

6.2 Performance Case Studies

We now examine the performance of implementing applications with the streaming programming model on the GPU. We have implemented a few representative but unique scientific applications on GPUs using Brook. The following applications were chosen for three reasons: they represent a variety of algorithms performed in numerical

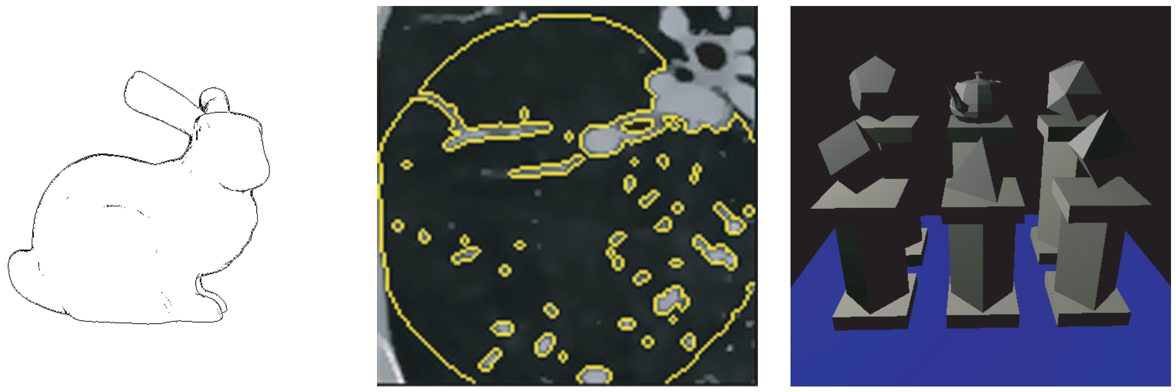


Figure 6.2: These images were created using the Brook applications FFT, Segment, and Ray Tracer.

applications which can benefit from the GPU's floating point computation power; they are important algorithms used widely both in computer graphics and general scientific computing; they are optimized CPU- or GPU-based implementations which are available to make performance comparisons with our implementations in Brook.

The goal of exploring these applications is not just to compare performance between the GPU and the CPU but also to understand the algorithmic properties which impacts the performance of streaming on the GPU. These properties include memory access patterns, arithmetic intensity, and general workloads. In addition, by running our Brook implementations with both the DirectX and OpenGL backends, we can measure the effects of the two APIs.

6.2.1 Linear algebra: SAXPY and SGEMV

The BLAS (Basic Linear Algebra Subprograms) library is a collection of low-level linear algebra subroutines [?]. **SAXPY** performs the vector scale and sum operation, $y = ax + y$, where x and y are vectors and a is a scalar. **SGEMV** is a single-precision dense matrix-vector product followed by a scaled vector add, $y = \alpha Ax + \beta y$, where x, y are vectors, A is a matrix and α, β are scalars. Matrix-vector operations are critical in many numerical applications, and the double-precision variant of SAXPY is a core computation kernel employed by the LINPACK Top500 benchmark [?] used to rank the top supercomputers in the world. We compare our performance against that of the optimized commercial Intel Math Kernel Library[?] for SAXPY and the ATLAS BLAS library[?] for SGEMV, which were the fastest public CPU implementations we were able to locate. For a reference GPU comparison, we implemented a hand-optimized DirectX version of SAXPY and an optimized OpenGL SGEMV implementation. For these tests, we use vectors or matrices of size 1024^2.

6.2.2 Image Segmentation

Segment performs a 2D version of the Perona and Malik [?] nonlinear, diffusion-based, seeded, region-growing algorithm, as presented in Sherbondy et al. [?], on a 2048 by 2048 image. Segmentation is widely used for medical image processing and digital compositing. We compare our Brook implementation against hand-coded OpenGL and CPU implementations executed on our test systems. Each iteration of the segmentation evolution kernel requires 32 floating point operations, reads 10 floats as input and writes 2 floats as output. The optimized CPU implementation is specifically tuned to perform a maximally cache-friendly computation on the Pentium 4.

6.2.3 Fast Fourier Transform

Our Fourier transform application, performs a 2D Cooley-Tukey fast Fourier transform (**FFT**) [?] on a 4 channel 1024 by 1024 complex signal. The fast Fourier transform algorithm is important in many graphical applications, such as post-processing of images in the framebuffer, as well as scientific applications such as the SETI@home project [?]. Our implementation uses three kernels: a horizontal and vertical 1D FFT, each called 10 times, and a bit reversal kernel called once. The horizontal and vertical FFT kernels each perform 5 floating-point operations per output value. The total floating point operations performed, based on the benchFFT [?] project, is equal to $5 \cdot w \cdot h \cdot channels \cdot \log_2(w \cdot h)$. To benchmark Brook against a competitive GPU algorithm, we compare our results with the custom OpenGL implementation available from ATI at [?]. To compare against the CPU, we benchmark the heavily optimized FFTW-3 software library compiled with the Intel C++ compiler [?].

6.2.4 Ray

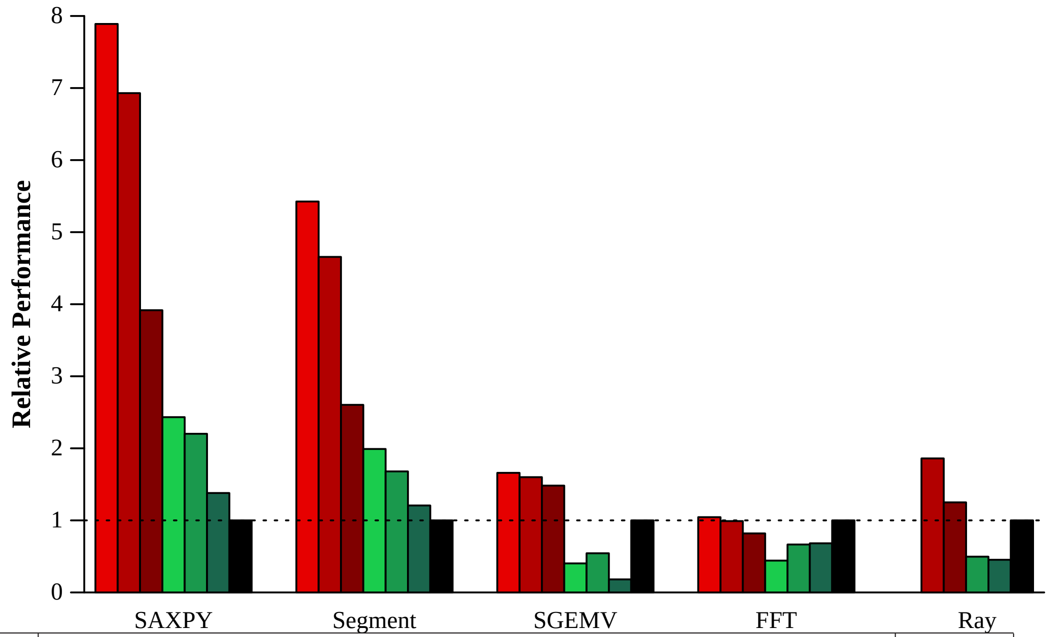
Ray is a simplified version of the GPU ray tracer presented in Purcell et al. [?]. This application consists of three kernels, ray setup, ray-triangle intersection (shown in section **??**), and shading. For a CPU comparison, we studied the ray-triangle intersection rate compared against the published results of Wald's [?] hand-optimized assembly which can achieve up to 100M rays per second on a Pentium 4 3.0GHz processor.

6.3 Performance Results

For each test, we evaluated Brook using the OpenGL and DirectX backends on both an ATI Radeon X800 XT Platinum running version 4.4 drivers and a pre-release[1] NVIDIA GeForce 6800 running version 60.80 drivers, both running Windows XP. For our CPU comparisons, we used a 3 GHz Intel Pentium 4 processor with an Intel 875P chipset running Windows XP, unless otherwise noted.

Figure 6.3 provides a breakdown of the performance of our various test applications. We show the performance of each application running on ATI (shown in red), NVIDIA (green), and the CPU (black). For each GPU platform, the three bars show the performance of the reference native GPU implementation and the Brook version executing with the DirectX and OpenGL backends. The results are normalized by the CPU performance. The table provides the effective MFLOPS observed based on the floating point operations as specified in the original source. For the ray tracing code, we report ray-triangle tests per second. In all of these results, to minimize the effects of computational intensity and kernel call overhead, we do not include the `streamRead` and `streamWrite` costs and have chosen significantly large dataset sizes.

[1]Running 350MHz core and 500Mhz memory



	MFLOPS				RT/sec
ATI					
🟥 Reference	4923	14171	2335	1278	-
🟥 BrookDX	4324	12163	2251	1212	186
🟥 BrookGL	2444	6800	2086	1003	125
NVIDIA					
🟩 Reference	1518	5200	567	541	-
🟩 BrookDX	1374	4387	765	814	50
🟩 BrookGL	861	3152	255	897	45
⬛ CPU	624	2616	1407	1224	100

Figure 6.3: Comparing the relative performance of our test applications between a reference GPU version, a Brook DirectX and OpenGL version, and an optimized CPU version. Results for ATI are shown in red, NVIDIA are shown in green. The bar graph is normalized by the CPU performance as shown by the dotted line. The table lists the observed MFLOPS for each application. For the ray tracer, we list the ray-triangle test rate.

In general, we observe that the GPU implementations perform well against their CPU counterparts. The Brook DirectX ATI versions of SAXPY and Segment performed roughly 7 and 4.7 times faster than the equivalent CPU implementations. However, FFT was our poorest performing application relative to the CPU. In the next few sections, we draw some general observations about the kinds of applications which will perform well as stream applications on GPU.

6.3.1 Suitability to Streaming

The SAXPY application illustrates that even a kernel executing only a single MAD instruction is able to out-perform the CPU. This application is fairly well suited to streaming since it takes as two input stream of values performs a multiply-add operation and produces a single output stream. This predictive, sequential memory access pattern is quite suitable to the streaming programming model and therefore streaming hardware which is optimized for sequential access, as discussed in chapter 2.

In contrast, the FFT performed relatively poorly. The Brook implementation is only .99 the speed of the CPU version. The Cooley-Tukey algorithm is a multi-pass algorithm, requiring a gather of the previous pass data in a bit-reversed order. The heavy use of the non-sequential access pattern stresses the memory system of the GPU. Secondly, the FFTW CPU implementation blocks the computation to make use of the Pentium4 large L2 cache. The CPU cache allows FFTW to reuse the write data. The GPU does not have any write level caching and therefore all writes are written to off-chip memory. As a result, the memory bandwidth difference between Pentium4 L2 cache, 44 GB/sec, and the NVIDIA GeForce 6800 Ultra, 36 GB/sec, clearly favors the CPU when there is heavy reuse of the data. However, despite this blocking, Brook is able to roughly match the performance of the CPU. Other stream processors have explored adding an additional levels of on-chip storage to correct this deficiency. These are discussed in chapter **??**.

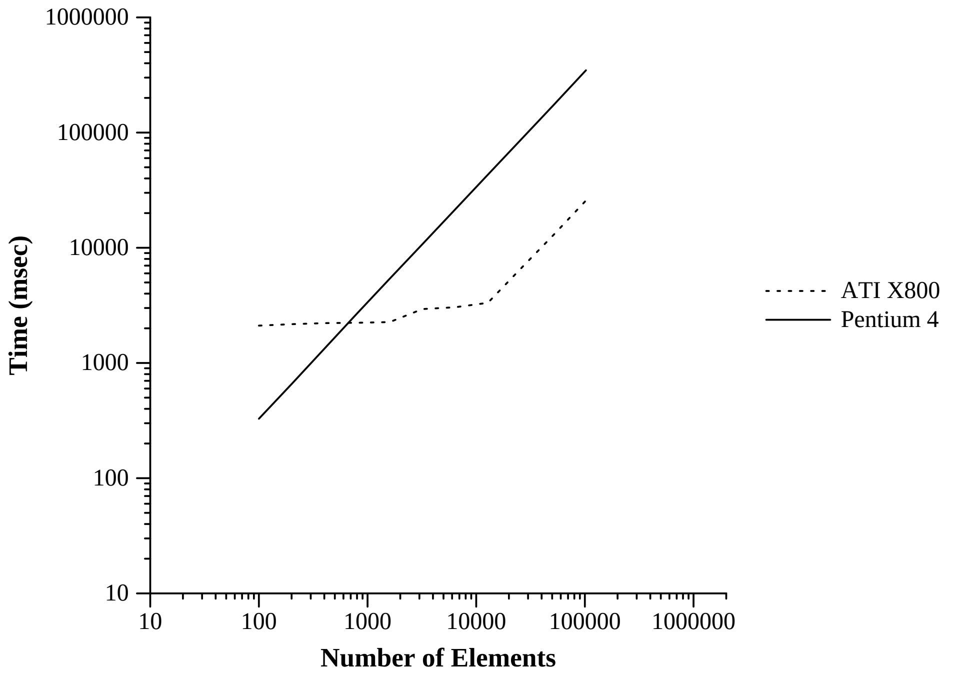


Figure 6.4: The performance of the Neohookean application with increasing matrix size.

6.3.2 Arithmetic Intensity

Another significant factor is the arithmetic intensity of the application. Consider the Segment and the SGEMV application which were able to obtain 14.2 GFLOPS and 2.3 GFLOPS respectively on the GPU. If we calculate the number of floating point math operations compared to the number of memory words transferred, we find that Segment has an average of 3.7 floating point operations per word transferred. This is due largely to the rather complex diffusion operator applied to the image. This is in contrast to the SGEMV application which performs on average 1/3 floating point operations per word. Clearly Segment, with its higher arithmetic intensity, is able to benefit from the GPU's significant floating point performance.

6.3.3 Efficiency of a high-level language

In some cases, our Brook implementations outperform the reference GPU implementations. For the NVIDIA FFT results, Brook performs better than the reference

OpenGL FFT code provided by ATI. We also outperform Moreland and Angel's NVIDIA specific implementation [?] by the same margin. A similar trend is shown with SGEMV, where the DirectX Brook implementation outperforms hand-coded OpenGL. We assume these differences are due to the relative performance of the DirectX and OpenGL drivers.

We can draw some general conclusions about the cost of using a high-level language like Brook instead of hand coding the fragment assembly and programming the graphics API directly. Since Brook is compiled to these lower level primitives, the concern is that the additional overhead introduced by the Brook compiler and runtime system could limit the overall peak performance for these applications. However, as we can see from these results, Brook is generally within 80% of the hand coded version.

In general, the same cannot be said for CPU based languages. Though languages like C map fairly directly to CPU assembly, to obtain peak performance on a modern processor like the Pentium4, we need to use the SEE instruction sets, which few compilers support natively, as well as explicitly blocking for the internal CPU caches. While

writing C code which maps to SSE instructions and takes full advantage of the processor caches is possible, these types of code optimizations are generally considered fairly advanced and are only undertaken with careful observance of the final assembly which is generated by the compiler. All of the CPU implementations used in this performance study were either hand assembled or explicitly optimized code for the Pentium4.

For example, consider the FFT application. To achieve peak performance on the CPU, FFTW performs an exhaustive search over a suite of known FFT implementations and chooses the fastest based on which is able to best utilize the CPU caches. It then reports the timing for the fastest implementation. On our test Pentium4 system, the CPU is able to achieve 1224 MFLOPS. If we instruct FFTW to use its reference C implementation of FFT, this performance drops to 204 MFLOPS, a 6x performance difference. This is in contrast to the Brook implementation. The only optimization made in the Brook code was to use 4-vector math operations which are native to the GPU and available within Brook. The Brook version is within 95% of the hand optimized GPU implementation which we obtained directly from the GPU vendor.

In general, we have found that users porting code from unoptimized C to Brook can see significant performance improvements mainly due to the fact that even straightforward C rarely produces the most optimal CPU code. Neohookean is an application which performs a matrix assembly for finite-element calculation and is used in solid mechanics hydrocodes. Figure 6.4 illustrates that Brook is able to achieve over a 14x speedup for dataset sizes with favorable computational intensity. This C code is currently in use by the mechanical engineering department for hyrocode simulations.

6.3.4 DirectX vs. OpenGL

We can also compare the relative performance of the DirectX and the OpenGL backends. DirectX is within 80% of the performance of the hand-coded GPU implementations. The OpenGL backend, however, is much less efficient compared to the reference implementations. This was largely due to the need to copy the output data from the OpenGL pbuffer into a texture. As discussed in chapter 5, OpenGL does not provide a direct render-to-texture functionality which we use to store our stream data. Instead, we must copy the kernel output from a temporary buffer into the texture representing the output stream. This copy operation is not necessary with DirectX as it supports direct render-to-texture. In many cases, the GPU reference implementations either were written in DirectX or used application-specific knowledge to avoid the copy with OpenGL.

6.3.5 NVIDIA vs. ATI

For these applications we observe that ATI generally performs better than NVIDIA. We believe this may be due to higher floating point texture bandwidth on ATI. We observe 1.2 Gfloats/sec of floating point texture bandwidth on NVIDIA compared to ATI's 4.5 Gfloats/sec when reading float4 texture data. This was confirmed to be an issue with the Geforce 6800 cache when reading from float4 textures. Despite the observable compute performance favoring NVIDIA with 40 billion multiplies per second versus ATI's 33 billion, for many of these applications this bandwidth penalty dominates performance.

6.4 Conclusions

From our application performance study, we can see that an application's arithmetic intensity and memory access patterns are critical if our Brook implementations are to outperform an optimized CPU implementation. We also see further speedups if we compare our straightforward Brook implementations to equivalently unoptimized C code. Furthermore, we have shown that the overhead using the high-level Brook language does not significantly impact performance compared to a hand-coded GPU implementation.[AI1]

Our analysis shows that there are three key aspects to using the GPU as a stream processor. First, GPUs do not operate within the same memory space as the host system. To counter the cost of transferring data to and from the GPU, the computational intensity of the algorithm, γ, required to outperform the CPU is inversely proportional to the speedup achieved by the GPU. Second, the amount of work done per kernel call should be large enough to hide the setup cost required to issue the kernel. Third, in order to achieve the best possible speedup from stream computing on the GPU, our algorithms must exhibit high arithmetic intensity which is the ratio of computation to communication in a kernel, such that we are limited by the impressive compute rate of the GPU, not by the memory system. We anticipate that while the specific numbers presented in this thesis may vary with newer hardware, the computational intensity, arithmetic intensity, and kernel overhead will continue to dictate effectiveness of the GPU as a streaming processor.

Chapter 7

Stream Programming Case Studies

In addition to these performance studies, we have observed some general programming challenges when porting applications to the stream programming model on the GPU. We have implemented over 50 different applications to Brook ranging from small test benchmarks to large numerical applications. In the following sections, we highlight three of these applications: sorting, molecular dynamics, and line of sight. We discuss some of the general lessons learned from implementing these application in Brook and more common challenges encountered and general principles for working around them.

7.1 Sorting on the GPU

Implementing sorting under the stream programming model on the GPU presents particular challenges. Most of the more common sorting algorithms are by nature sequential operations. Though many of them adopted to take advantage of task parallelism, few are suitable for stream programming model. As discussed previously, we must choose a sorting algorithm which is data parallel. Secondly, as discussed in chapter 5, the GPU does not natively support scatter, or indirect write, operations. Therefore, we need a sorting algorithm which minimizes indirect writes.

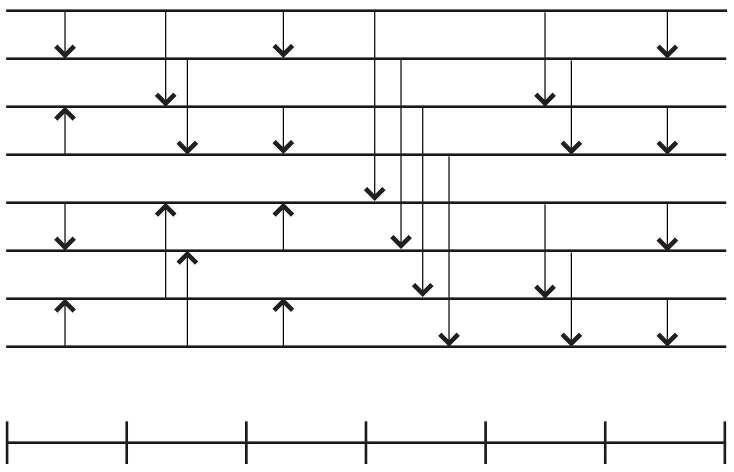


Figure 7.1: Bitonic Sorting Network for 8 Elements

Each vertical arrow represents a comparison and swap operation between two elements in the stream. If the values do not obey the direction of the arrow, the two elements are swapped. This example requires 6 kernel executions, specified by the hash marks, which perform the comparison and swap in parallel. In general, bitonic sorting requires O($\log^2$ n) kernel executions to sort n numbers. The kernel code to perform the sort is shown below:

```
kernel void bitonic(iter float idx1<>, float input[ARRAYSIZE],
    out float output<>, float stageWidth,
    float offset, float twoOffset) {

    float idx2;
    float sign, dir;
    float min, max;

    // either compared with element above or below
    sign = (fmod(idx1, twoOffset) < offset) ?  1.0 :  -1.0;

    // "arrow" direction in the bitonic search algorithm
    dir = (fmod(floor(idx1/stageWidth), 2) == 0) ?  1.0 :  -1.0;

    // comparing elements idx1 and idx2
    idx2 = idx1 + sign*offset;

    min = (input[idx1] < input[idx2]) ?  input[idx1] :  input[idx2];
    max = (input[idx1] > input[idx2]) ?  input[idx1] :  input[idx2];

    output = (sign == dir) ?  min :  max;
}
```

7.1.1 Batcher Bitonic Sort

The streaming sort application performs a Batcher bitonic sort [?] to sort n numbers. This technique was first employed on GPUs by Purcell et al.[?] and further described in [?] and also implemented on the Imagine stream processor [?]. Bitonic sorting is efficient for graphics hardware because it is both data parallel and is performed in place, requiring only gather operations. Bitonic sorting operates with a static comparison network where two elements are compared and switched if out of order as shown in figure 7.1.

Due to the lack of scatter capabilities, the swap operation is difficult. To emulate swap, the comparison evaluation is performed twice, once for either side of the arrow in figure 7.1. Based on the position in the stream and the pass number, the kernel computes which element to examine and the direction of the comparison before writing the output. Bitonic sorting would also benefit from a more complete instruction set. Computing which element to compare with requires a modulus 2 operation which is not available on current graphics hardware. This is emulated by a `fmod` instruction macro.

7.1.2 Complexity of Stream Applications

The main lesson from implementing sorting on the GPU is that the streaming programming model may change the algorithmic complexity of an algorithm. Unlike many of the its sequential relatives, the bitonic sort algorithm has $O(n \log^2 n)$ complexity rather than $O(n \log n)$. We must execute the kernel function $\log^2 n$ times, each time over the n elements. In general, it has been proven [?] that parallel network sorts have $O(n \log^2 n)$ complexity. Because on of the tenets of the stream programming model is a data parallelism, migrating programs from the sequential CPU domain to the streaming GPU may incur additional costs in algorithmic complexity.

7.2 Eliminating Scatter in Molecular Dynamics

As part of the streaming programming model, Brook encourages programmers to structure computation around sequential memory access. However, many applications naturally require indirect memory access. For reading data, Brook permits gather operators inside of kernels which array indexing. For indirect write operations, Brook offers an external scatter operator. However, as discussed above, this scatter operator cannot be implemented efficiently on the GPU. Therefore, Brook developers should consider alternatives to scatter. In this section, we'll discuss some of the common cases where we can eliminate scatter operations from our code.

There are a variety of methods to minimize scatter in an algorithm. One solution is to restructure the computation to eliminate the scatter operation completely. Another possible solution is to convert scatters to gather operations. While this cannot be done in all cases, it can often be the simplest solution to eliminating scatter. Both of these transformations have benefits and drawbacks which will be discussed in context of an application which relied heavily on scatter operations.

Gromacs

Gromacs is a numerical simulation software package for simulating the motion of atoms in for predicting and understanding how proteins assemble themselves, or *fold*, which determines how they operate in biological systems. Current numerical simulations may take on the order of days to simulate a few nanoseconds of motion which is long enough for simple proteins to fold. However average proteins may take up to hundreds of milliseconds to complete folding, requiring years worth of simulation time.

The majority of Gromacs' simulation time is spent computing the non-bonded forces between atoms. In order to only evaluate the force function between atoms which are reasonably close to each other, Gromacs computes a *neighbor list* consisting of atoms pairs which fall within a given cutoff radius. Updates to the neighbor list are performed at a lower frequency compared to the rest of the simulation as not to dominate the computation, typically every 10 time steps.

The pseudo-code for the non-bonded force calculation is:

```
for each atom i
    for each atom j in i's neighbor list
       f = F(i, j);
       forces[i] += f;
       forces[j] -= f;
```

where `f` is the computed vector force between atoms i and j. These forces are computed from the electrostatics and Lennard-Jones components between two molecules given by the equation below [**?**].

$$F\left(i,j\right)=\left(\frac{1}{4\pi\epsilon_0}\frac{q_iq_j}{\epsilon_r r_{ij}^2}+12\frac{C_{12}}{r_{ij}^{12}}-6\frac{C_6}{r_{ij}^6}\right)\frac{\mathbf{r}_{ij}}{r_{ij}} \tag{7.1}$$

7.2.1 Implementing the Force Function

Converting this sequential code to using streams and kernels is initially straightforward. The force kernel takes as input the stream of atom i's and evaluating the inner for loop inside the kernel function and gathering the correct atom j's from the neighbor list. The difficulty arises however in determining what the kernel output should be since the results of the force calculation are scattered into the forces array. Since the kernel input is a stream of atom i's, we could output a stream of i forces but this does not facilitate the j forces:

```
kernel void computeForce(atom i<>, force forces[], out force fi<>)
    fi = 0;
    for each atom j in i's neighbor list
       fetch j;
       f = F(i, j);
       fi += f;
       forces[j] -= f; // illegal
```

One alternative would be to convert the indirect write operations into an indirect read operation (i.e. gather) which the GPU is capable of performing. The kernel outputs all computed fi force vectors to memory. With an additional pass, a second kernel could then use the neighbor list data to gather all of the forces for a particular atom. The pseudo code for the two kernels is shown below.

```
kernel genforce(atom i<>, atom j<>, force fij<>)
    fij = F(i,j);

kernel sumforce(atom i<>, force fij[], force fi<>)
    fi = 0; for each atom j in i’s neighbor list
       fi += F[..];
```

The downside to this approach is that it wastes bandwidth. The kernel code showed above is at least able to perform some local summations before outputting the fi force to memory. The alternative is to perform each force calculation twice, once for the i atom and again for the j atom. This modification allows us to simplify the pseudo code to eliminate the indirect write inside the inner loop:

```
kernel void computeForce(atom i<>, out force fi<>)
    fi = 0; for each atom j in i’s neighbor list
       fetch j;
       f = F(i, j);
       fi += f;
```

Since both atom i and j have each other present in their respective neighbor lists, we will compute both `F(i,j)` and `F(j,i)` despite their equality. However, this modification permits to express the computation as a kernel function.

The obvious disadvantage of this modification is that we perform twice as much computation. The GPU will have double the number of interactions and therefore read twice as many atom positions. However, there are significant global bandwidth savings. A kernel will only output a single force vector per atom. The original method reads half as many atom positions, however it scatters a force vector per interaction which it then reads again in a separate pass. A break-down of the cost of these two approaches in table 7.1.

Deciding which technique is most beneficial for eliminating a scatter operation is a function of the arithmetic intensity of the algorithm. If a technique has high

	Reads	Writes	Total
computeForce	$n + 2m$	n	$2n + 2m$
genforce	$2m$	m	$3m$
sumForce	$2m$	n	$n + 2m$
gen+sum Total	$4m$	$m + n$	$n + 5m$

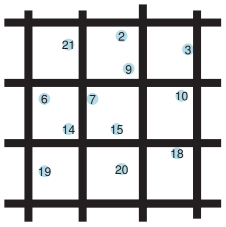
Table 7.1: Memory operations for emulating scatter in Gromacs

This table outlines the memory costs of the two methods to emulate the scatter operation in Gromacs. n is the number of atoms in the simulation and m are the total non-duplicate interactions which must be evaluated. For a typical simulation $m \approx 200n$. The first method only uses the computeForce kernel while the second requires both genForce and sumForce.

arithmetic intensity, it may be better to perform the calculation only once. In the case of the Gromacs force evaluation, the arithmetic intensity is not significantly high, only requiring approximately 12 flops per force calculation. Clearly performing twice this amount of computation for a 2.5x bandwidth improvement is beneficial.

7.2.2 Address Sorting in Gromacs

In some cases, it is not possible replace a scatter operation with computation nor gathers. Particularly this occurs when the scatter address is not fixed but computed by the kernel function itself. An example of this in occurs Gromacs when working the acceleration data structure. Gromacs divides the simulation space into cells. These cells are used for generating the neighbor lists so that only atoms are close enough to have a significant effect are evaluated by the force function. Typically we want to compute the properties of the cell such as the number of particles in cell to update the data structure. A simple CPU implementation shown in figure 7.2 would increment the count of an atom's corresponding cell based on atom position, a computed address. This most basic scatter operation is trivial for a CPU implementation however poses a challenge for the GPU. It is simple to build a kernel which generates a stream of the computed cell addresses per atom, however, what is needed is a stream consisting of which atoms are in each cell for a kernel to compute the net value of the cell. This



```
for all particles p
  voxels[p.pos] += p.vel
```

Figure 7.2: Scatter used to compute net atom count per cell.

Though this scatter operation is simple for a CPU implementation, this scatter operation is quite difficult for the GPU.

inversion was simple in the above force calculation example since the scatter index was known a priori. However, this time the address is computed

The solution to implementing this scatter operation is to sort the data to scatter by the scatter addresses. Bitonic sorting is used since it is a network sort which has static memory access patterns such that any scatter operations can be easily converted into gathers. The resulting stream is ordered such that all of the data to be scattered is in contiguous scatter address locations. We can then execute a kernel over all of the output elements, where for each element we perform a simple binary search over the sorted stream, gathering the final result. In the case of Gromacs, this technique can be simplified to only sort the addresses since we only care about the number of atoms in each cell. An overview of this technique is outlined in figure 7.3.

7.2.3 General Techniques for Eliminating Scatter

By reworking the force evaluation portion of Gromacs, we have shown that it is sometimes possible to eliminate scatter operations by either performing additional computation or converting them into gather operations. In general, the nature of the computation will dictate whether these techniques can be applied. In the case of the Gromacs force evaluation, we were able to take advantage of Newton's third law to replace the scatter with double the computation.

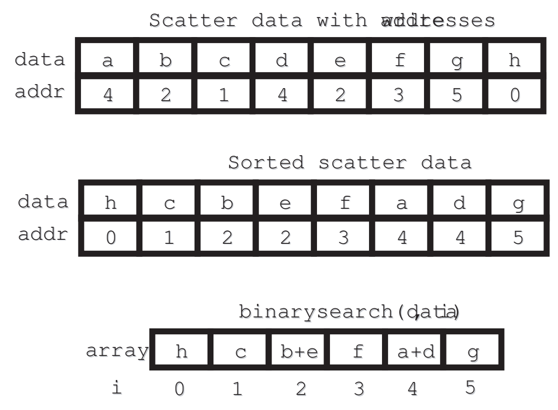


Figure 7.3: Using address sorting to implement scatter.

Address sorting is a two step process. First we sort the scatter data using the scatter address as the key. This produces a sorted list of data by address. The next step is to execute a kernel over the output stream, performing a binary search of the sorted scatter data looking for elements which match the array address value.

In general, it is also possible to replace scatter operations with gather operations whenever the scatter address is static. In Gromacs, the address used to scatter the computed force into the forces array is not data-dependent. Rather, it is based on which atom is being evaluated. This fact allowed us to easily generate the *inverse* address used by the gather operation which replaces the scatter. Computations which occur over fixed mesh topologies is another example of where scatters to static addresses can be replaced by gathers operations. Where conventional CPU program may compute values at the nodes and scatter the results to adjacent nodes, we can convert these operations into gathers by writing out the scattered values and gathering them in a second pass.

Finally where the scatter address is computed, we can implement scatter s by sorting the scatter data based on the scatter address. This technique is clearly the most costly due to the number of passes involved. However, may be the only technique possible to perform the scatter on the GPU without involving the CPU.

7.2.4 Conclusions

In this section, we explored three techniques to implement scatter solely with the GPU. Due to the GPU's limited capability to perform scatter operations, applications which naturally perform scatter operations challenging to implement on the GPU. Choosing between these techniques is a based on the computation. In the case of Gromacs, we have shown that it is better to double the computation than take the additional bandwidth penalty of than the additional passes required to convert the scatter into gathers. The address sorting technique is clearly the most costly solution requiring many passes over the scatter data however may be the only approach not involving the CPU if the scatter addresses are not static.

As GPUs continue to evolve, it is likely that future GPUs will provide some native capabilities for scatter operation. Already, the vertex processor is capable of setting the x,y position of a vertex which could be used to scatter data. Historically, it has been difficult to get data from texture memory to the vertex processor though this capability has only just been added to the GPU. While it is possible to use this new functionality to perform scatters, ideally, scatter capabilities are needed at the fragment level.

7.3 Branching in Computing Line of Sight

While the previous section discussed how to restructure computation around the memory system, this section deals with structuring computation to make the most efficient use of the computational units. The stream programming model enforces data parallel execution of the kernel functions over the stream elements by prohibiting any communication which would introduce dependencies between the elements. This allows the GPU to execute multiple version of the kernel in parallel. Today's GPUs execution units have limited support for branching in their instruction sets since the parallel fragment processor which execute the kernel operate as a SIMD parallel processor. In the SIMD model, the processors all execute the same instruction with different data. As a result, branching, or data-dependent changes in code

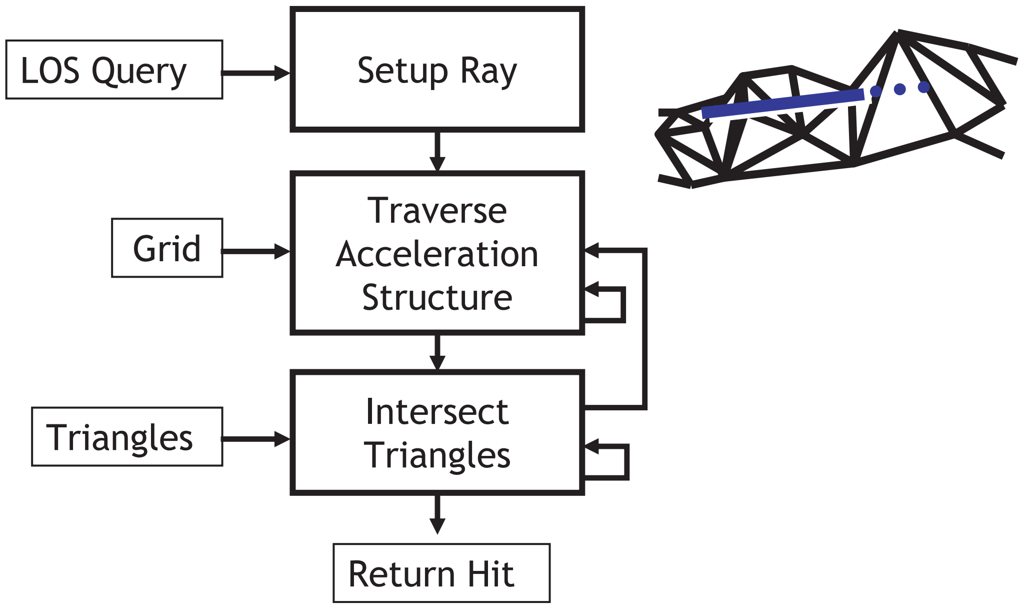


Figure 7.4: Block diagram of streaming line of sight computation.

The line of sight computation in computed with three separate kernels. First we setup the ray to traverse through the scene. The second kernel walks the regular grid acceleration structure. The final kernel insects the ray with all of the triangles in a grid, either returning a hit or returning the ray to the traverser state.

flow are generally prohibited. In this section, we discuss how this limitation can impact streaming application on the GPU and highlight some ways to work around these challenges.

7.4 Computing Line of Sight

Computing the line of sight in a virtual world is a problem akin to ray tracing. A typical simulation, we have simulation entities scattered over a terrain whose behavior is based on the their perception of environment. Computing the visibility of different entities as well as portions of the terrain is often one of the most compute intensive part of simulating virtual worlds. In the OneSAF combat simulation system, over 70% of the compute budget is reserved for line of sight computation.

Implementing the light of sight calculation in Brook is largely based on our ray-tracing implemention which originated from the work by Purcell et al.[?]. Figure 7.4 provides a block diagram of the line of sight computation. The first kernel takes as input a stream of queries which are converted into rays which are to be walked through the a regular grid acceleration structure. We then iterate over the stream of rays, walking them through the computation grid acceleration structure until a ray enters a grid cell containing triangles. These queries are processed by the intersect triangle kernel which tests the ray against all of the triangles in the grid cell, returning a ray hit if found.

There are a variety of branching constructs used in the line of sight code. These include basic if-then-else conditionals as well as data dependent looping. In the following section, we discuss how these constructs are implemented on the GPU as well as some of the performance implications.

7.4.1 Predication

As we walk our query rays through the grid, we need to make sure that we do not continue to process queries which have already resulted in a hit. To do this we maintain a ray state field which records whether a ray is presently walking the grid, inside of a grid cell with untested triangles, or has completed either with a hit or miss (the ray's t-max has been exceeded). A basic Brook implementation of line of sight executes either the traverse or intersect kernel over all of the queries, updating the ray state as the computation progresses. The beginning of both of these kernels first test the ray state to test if kernel should process the ray. For example, the pseudo code for the traversal kernel is as follows:

```
kernel void traverse(ray r<>, out ray result<>, ...)
    if (r.state == NeedsTraverse)
       result = ...  // perform traversal
    else
       result = r; // do nothing
```

Though this Brook program will compile to the GPU fragment shader, the compiler will predicate computation. In the absence of native branching instructions, the compiler will evaluate both sides of the conditional and discard one of the results based on the value of the Boolean. For example:

```
Shader Code
if (a)
    b = 1/sqrt(c);
else
    b = 1/c;

Compiled Assembly
rsq r1.x, c
rcp r2.x, c
cmp b, -a, r1.x, r2.x
```

Here we can see that even though we would expect to only execute either the inverse square root or the reciprocal operation, instead both operations are performed and the GPU chooses one of these results. So even though predication correctly implements the branch, the disadvantage of this approach is that we are executing both sides of the branch. In the case of the line of sight code, this means we will be always be incurring the performance cost of executing all of the kernel instructions even if the code path implies these instructions are skipped. Clearly we must look for ways of preventing a branch from executing unnecessarily.

7.4.2 Writemask Operators in Brook

The main disadvantage to predication is that it evaluates both sides of the branch. It is possible however to use traditional Z-buffering to prevent the GPU from executing code not taken as part of the branch. This method can be better than the predication mechanism since in many cases the GPU can prevent the execution of the kernel by performing the depth test before the execution of a kernel program on a fragment processor. If we could store the ray state into Z buffer, we can only execute the kernel body on rays which the computation.

To test the effectiveness of use the Z-buffer to filter the computation, we added a few additional stream operators to Brook for filtering computation. These include:

```
streamSetWriteMask( maskStream );

streamEnableWriteMask();

streamDisableWriteMask();
```

`streamSetWriteMask` takes as input a stream of boolean values (represented as floats) to use a filter subsequent kernel executions. `streamEnableWriteMask` and `streamDisableWriteMask` turn on and off the this feature. Where ever the mask stream is set to one, the computation proceeds as normal for the corresponding element in the output stream. Where the mask stream is zero the kernel function is not evaluated and the output stream element is not modified. The implementation of these operators simply update, enable, and disable to GPU's z buffer to filter the computation.

To evaluate the effectiveness of using the writemask operators for line of sight calculations, set to write mask such that the the kernel only performs the computation for queries which are in the corresponding state. We tested the Brook line of sight application with a 1Mtri terrain dataset. This real-world data consisted of a non-uniform triangular mesh which included regions of high and low resolution. The input stream was 10,000 randomly distributed queries with a 48% terrain hit rate. As shown in Figure 7.5, the write masking improved the overall performance of the execution time of the queries by a factor of 7.

These results illustrate that although the GPU executes kernel functions as a SIMD processor with limited support for branching, we can bypass kernel execution completely for some stream elements through the use of traditional z buffering. This provides a significant performance improvement over traditional predication Further experimentation however shows that this optimization is applied by the hardware at a courser resolution than each stream element. Figure 7.6 compares the performance of evaluating a kernel function with a write mask with high versus low spatial locality.

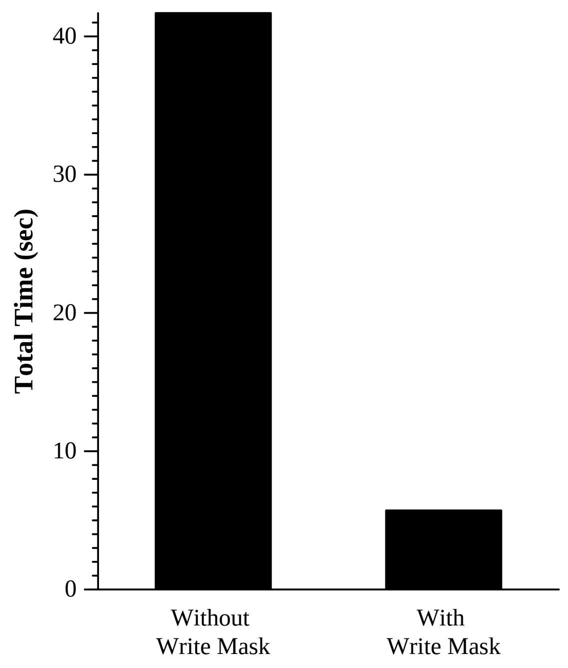


Figure 7.5: Line of sight performance both with and without writemasking.

The total query time for 10,000 randomly distributed queries over a 1Mtri terrain. Using write masking improved performance by a factor of 7.

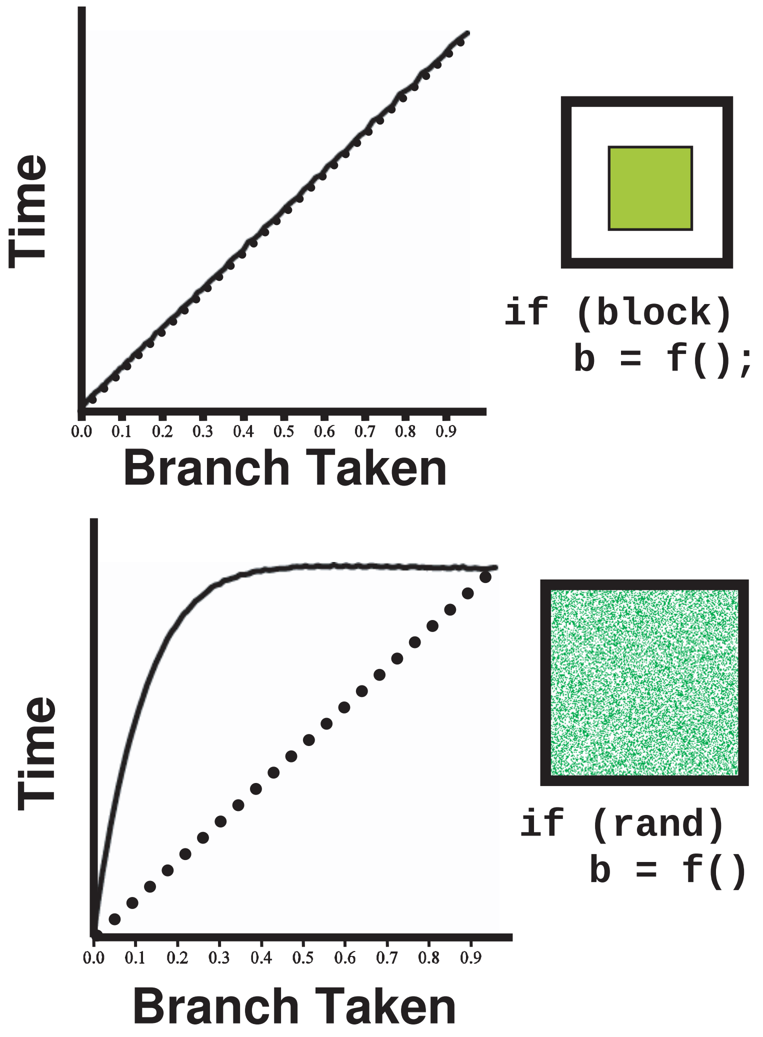


Figure 7.6: Spatial locality of writemask operator

These graphs illustrate the spatial locality of using the writemask operator on the NVIDIA GeForce 6800. We compare the time of executing a program versus probability of taking a branch with a random and block Boolean condition. The diagonal line indicates the ideal execution time assuming the Boolean always prevents f() from executing.

From these results, we can see that the writemask optimization is clearly most effective if there is locality in the branch with neighboring stream elements. Otherwise, this optimization is not much more better than predication.

7.4.3 Using Shader Model 3 Branching Instructions

We also evaluated the potential for using the Shader Model 3 branching instructions for line of sight. Native branching instructions only became available recently with the release of the GeForce 6 series GPUs. These instructions provide native branching, including looping, at the fragment level:

```
MOVC CC, R0; IF GT.x; MOV R0, R1; # executes if R0.x > 0 ELSE; MOV R0, R2; # ex
```

With this new functionality it is possible to execute our entire line of sight evaluation in a single kernel function. We do this by looping each ray through the grid cells, intersecting any triangles along the way. The pseudo code for the traversal is:

```
setup traversal while no hit while no triangle in cell traverse grid for each t
```

The resulting code required only lines 92 lines of Brook code and compiled to 186 instructions using the NVIDIA fragment program 2 extension.

Unfortunately, the shader model 3 version of the line of sight code was unable to compute the correct result for even our small test cases due to limitations in the instruction set, and further study revealed that the performance of instruction level branching is lackluster.

First, the branching instructions are limited to a maximum of 255 loop iterations in the kernel code. This means that any while loop exceeding this loop count will abort the computation. Secondly, there is a maximum 64K instruction execution limit per kernel invocation. Any line of sight ray computation which executes this will abort. Finally, given that by evaluating the entire line of sight computation in a single kernel call, it may a significant amount of time to complete. If the GPU takes more than a few seconds to complete the kernel call, Windows assumes that the hardware as stopped responding and shuts down the device driver.

Even despite these limitations, further study revealed that there remain performance issues with the branching instructions. Figure **??** illustrates how these instructions are sensitive to spatial locality issues, similar to the writemask operator. From these results, we conclude that the GPU is likely evaluating the branching instructions with the parallel SIMD execution units. Furthermore the branching instructions suffer from even further spatial locality issues than the writemasking operators. As our line of sight computation deviate between rays, it is likely that our overall performance will quickly drop.

7.4.4 Choosing a Branching Mechanism

From our line of sight study, we can draw some general conclusions about choosing an effective branching mechanism for a stream programming on the GPU. Choosing between the different branching mechanisms is primarily based on the size of the conditional and amount of state present. For short branches, i.e. two to four operations, predication is preferred since the overhead for either write masking or the branching instructions will negate any benefits. One larger branches, one key benefit of the branching instructions is that we preserve all of our program state across the branch operation. Write masking requires that we save all of the necessary program state, execute the branch in a separate pass, and restore the state of our program to continue. For large programs, these saves and restores can make write masking inefficient. However, if we can effectively isolate the branch component of our program, write masking can provide the best performance and does not have any of the instruction limitations sighted above. In the future, it is certainly possible that GPUs may evolve into more MIMD style processor which would eliminate many of the disadvantages of the branching instructions.

Chapter 8

Discussion

This dissertation makes several contributions to the area of computer systems, parallel programming, and computer graphics. In this chapter, we review these contributions and highlight some areas for future work.

8.1 Contributions

8.1.1 Stream programming model

We have presented the stream programming abstraction for GPUs. Through an analysis of the execution model of today's GPUs as well as a close examination of how GPUs have been able to scale in performance, we have identified two main reasons why GPUs have been outperforming CPU architectures: data parallelism and arithmetic intensity. The stream model is built upon these ideas. As we have shown, applications which have limited arithmetic intensity tend to be restricted by the memory performance rather than by the computation rate. Additionally, Gromacs has demonstrated that in many cases, artificially increasing the arithmetic intensity of an algorithm can reduce the memory bandwidth of an application. As GPUs continue to increase in performance, these properties are likely to become more significant for efficient GPU utilization.

8.1.2 Brook programming system

We have presented the Brook stream programming system for general-purpose GPU computing. Through the use of streams, kernels, and reduction operators, Brook formalizes the stream programming abstraction and the GPU as a streaming processor. The basic stream and kernel primitives are critical for structuring computation such as organizing predictable sequential memory access, capturing the locality of computation, and encouraging programmers to implement their algorithms with high arithmetic intensity.

We have demonstrated how to compile Brook programs to the GPU's native assembly and graphics API. As GPUs continue to evolve, Brook may need to evolve as well. For example, in the Imagine processor, the stream register file is exposed just as a vector register file is exposed in a vector processor. This additional level in a memory hierarchy has many advantages but makes it much more difficult to write an optimizing compiler. Another interesting issue is how to combine multiple streaming processors, or multiple GPUs, into a larger machine. Such a machine might have a huge cost-performance advantage over existing supercomputers.

8.1.3 Extending and virtualizing of the GPU limitations

We have demonstrated a system of virtualizing and extending various GPU hardware limitations using our compiler and runtime system; specifically, the GPU memory system, the number of supported shader outputs, and support for user-defined data structures. Virtualization of hardware constraints can also bring the GPU closer to a fully general streaming processor. Many of the constraints eliminated by theses techniques are an artifact of rendering and are not inherent to basic architecture. For example, the fact that existing GPUs only support a 2D segmented memory system could be easily relaxed in future architectures. Additionally, support for exception mechanisms would be of great benefit to further GPU virtualization techniques.

8.1.4 Evaluting GPU as a streaming processor

We have evaluated the costs of using the GPU as a streaming processor, specifically the computational intensity, the kernel call overhead, and the arithmetic intensity. Our computational intensity analysis demonstrated that read/write bandwidth is important for establishing the types of applications that perform well on the GPU. Ideally, future GPUs will perform the read and write operations asynchronously with the computation. This solution changes the GPU execution time to be max of T_r and K_{gpu}, a much more favorable expression. It is also possible that future streaming hardware will share the same memory as the CPU, eliminating the need for data transfer altogether.

As GPUs continue to evolve, it is likely that the bandwidth gap will continue to grow between instruction and memory rates. This trend will continue to put pressure on application writers to structure their algorithms with even higher arithmetic intensity to capture the added compute performance. Streaming processors developed in the research community have alleviated this problem by adding additional levels to the memory hierarchy with on-chip memories which narrow this bandwidth gap. Assuming future GPUs adopt similar ideas, further research is needed to see how these additional memories can be used by the compiler and runtime system.

8.1.5 Computing the GPU versus CPU

We have evaluated the performance of computing with the GPU compared to CPU with a variety of benchmark applications implemented in both Brook, hand-coded GPU code, and optimized CPU code. In general, we have shown the potential for up to a 7x performance improvement over optimized CPU implementations depending on how well the application maps to the stream programming model and the amount of arithmetic intensity present in the computation.

8.1.6 Tradeoffs for Stream Programming on GPUs

We have presented a collection of common algorithmic tradeoffs for more efficient execution of stream programs on GPUs. Specifically, we explore techniques to minimize the need for scatter and the use of branching in kernel functions.

8.2 Last words

In summary, the Brook programming environment provides a simple but effective tool for computing on GPUs. Brook for GPUs has been released as an open-source project [?] and our hope is that this effort will make it easier for application developers to capture the performance benefits of stream computing on the GPU for the graphics community and beyond. By providing easy access to the computational power within consumer graphics hardware, stream computing has the potential to redefine the GPU as not just a rendering engine, but the principle compute engine for the PC.

Chapter 9

Appendix: Brook compilation example

This is where the code example goes...