Data Parallel Computation on Graphics Hardware

Ian Buck Pat Hanrahan

Stanford University *

Abstract

As the programmability and performance of modern GPUs continues to increase, many researchers are looking to graphics hardware to solve problems previously performed on general purpose CPUs. In many cases, performing general purpose computation on graphics hardware can provide a significant advantage over implementations on traditional CPUs. However, if GPUs are to become a powerful processing resource, it is important to establish the correct abstraction of the hardware; this will encourage efficient application design as well as an optimizable interface for hardware designers.

In this paper, we present a stream processor abstraction for performing data parallel computation using graphics hardware. In this model, computation is expressed as kernels executing over streams of data. We have implemented this abstraction with current programmable hardware in a system called *Brook* and present a variety of applications demonstrating its functionality. Stream computing not only represents an accurate computational model for today's hardware but it also captures the underlying advantage of computing on graphics hardware.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture—Graphics processors

Keywords: Programmable Graphics Hardware, Data Parallel Computing, Stream Computing, Brook

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 [Larsen and McAllister 2001], cellular automata [Harris et al. 2002], and a complete ray tracer [Purcell et al. 2002] 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 [von Neuman 1945] 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.

*{ianbuck, hanrahan}@graphics.stanford.edu

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 [Microsoft 2003b; NVIDIA 2003b]. McCool [2000] 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 [Upstill 1990] and RTSL [Proudfoot et al. 2001]. 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.

The purpose of this paper is to propose a *stream* programming model for general purpose computation on graphics hardware. The main contributions of this paper include:

- We describe the two key reasons that graphics hardware is faster than CPUs: data parallelism and arithmetic intensity (the ratio of computation to bandwidth). The streaming model is built upon these ideas.
- Using these observations as a guide, we present a programming environment called *Brook*. We describe our prototype implementation on todays GPUs.
- We demonstrate how a wide variety of data parallel algorithms can be implemented in Brook and hence run on graphics hardware.
- Finally, we analyze our implementation on current programmable graphics hardware and provide direction for future hardware designs.

2 Background

2.1 Programmable Graphics Hardware

Figure 1 shows a diagram of a modern programmable graphics accelerator such as the ATI Radeon 9700 or the NVidia GeForce FX [ATI 2003; NVIDIA 2003a]. The programmable vertex and fragment processors execute a user specified assembly-level program consisting of 4-way SIMD instructions[Lindholm et al. 2001]. These instructions include standard math operations, such as 3- or 4-component dot products, texture fetch instructions (fragment programs only), and a few special purpose instructions. The Radeon 9700 supports 24-bit arithmetic and the GeForce FX permits fixed point, half-float, and single-precision IEEE floating point arithmetic.

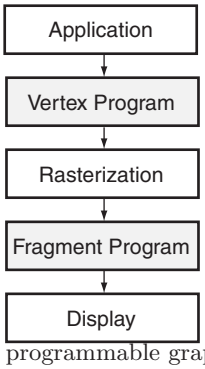

Figure 1: The programmable graphics pipeline.

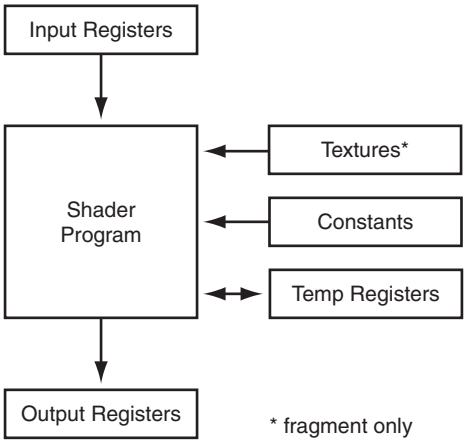

Figure 2: Programming model for current programmable graphics hardware. A shader program operates on a single input element, vertex or fragment, located in the input registers and writes the result into the output registers.

The basic execution model is shown in figure 2. For every vertex or fragment to be processed, the graphics hardware places the element's data fields in the read-only input registers and executes the program. The program then writes the results to the output registers. During execution the program has access to a number of registers as well as constants set by the host application. Although there are several differences in the instruction sets between vertex and fragment programs, the key distinction is that vertex programs may branch, but not fetch from texture (memory), whereas fragment programs may fetch from texture, but not branch. Another difference between fragment programs and vertex processors is their execution speed: since fill rates are faster than triangle submission rates, fragment programs are currently faster.

While this programming model permits general data types and arbitrary user code to be executed on on graphics hardware, it remains very graphics centric and does not provide a clean abstraction for general purpose computing on graphics hardware. Programs operate on vertices and fragments separated by a rasterization stage; memory is divided up into textures and framebuffers; the interface between the hardware and host is through a complex graphics API. This often stands in the way of using the hardware for new applications.

2.2 Programming Abstractions

There were many early programmable graphics systems, but one of the most influential was the UNC PixelPlanes series [Fuchs et al. 1989] culminating in the PixelFlow machine [Molnar et al. 1992]. These systems were based on smart memory, where pixel processors were on the same chip as the embedded framebuffer memory. The pixel processors ran as a SIMD processor. Providing this flexibility has allowed the UNC group to experiment with new graphics algorithms, culminating in Olano and Lastra's implementation of a version of RenderMan on PixelFlow [Olano and Lastra 1998].

Peercy et al. [2000] demonstrated how the OpenGL architecture [Woo et al. 1999] 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 clearly argued the case for extended precision fragment processing and framebuffers.

The work by Thompson et al. [2002] 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 [Khailany et al. 2001].

Purcell et al. [2002] implemented a ray tracing engine on graphics hardware 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 paper extends the streaming model presented in that paper by considering more general parallel applications and more general programming constructs.

Stream architectures are a topic of great interest in computer architecture. For example, the Imagine stream processor by Kapasi et al. [2002] uses streams to expose locality and parallelism. They were able to show that streaming worked well for a wide range of media applications, including graphics and imaging [Owens et al. 2000]. Related current work includes VLIW media processors and DSPs [Halfhill 2000; Rathnam and Slavenburg 1996], hardwired stream processors and programmable gate arrays [Bove and Watlington 1995; Gokhale and Gomersall 1997], and vector processors [Russell 1978; Kozyrakis 1999].

Over the years, a large number of parallel programming environments have been created, most based on communicating sequential processes, or multithreading. There are also several stream programming models, the most similar to ours being the Imagine StreamC/KernelC programming environment [Kapasi et al. 2002]. StreamC/KernelC, however, exposes the sequential nature of stream computations. In this paper we stress the data parallel nature of streams. A good example of a data parallel language is C* [Hillis and Guy L. Steele 1986; Thinking Machines Corporation 1993], and we base our constructs on those in that language. In short, our programming environment combines ideas from StreamC/KernelC with ideas from C*; for kernel programming we primarily use Cg in a wrapper to integrate it with the rest of our system.

3 Advantages of Stream Architectures

Before defining a programming model for general computing on graphics hardware, it is important to understand why graphics hardware outperforms conventional CPUs. 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 dictate performance. GPUs 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 GPUs exploit these features to maintain peak performance.

3.1 Data Parallelism

Programmable graphics hardware is an example of a data parallel architecture[Hillis 1985]. In this paper, 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. 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 multithreaded 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 [Torborg and Kajiya 1996; Anderson et al. 1997; Igehy et al. 1998]. 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.

3.2 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, or in other words, flops per word transferred. As an example, in today's programmable fragment processors, 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 transfered, four operations can be performed. In the future, based on hardware trends, 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, except when absolutely necessary.

4 Brook Stream Programming Model

An ideal programming model for general purpose computing on graphics hardware should encourage programmers to write data parallel code with high arithmetic intensity. We present *Brook*, a programming environment for general purpose stream computing. Brook abstracts the stream hardware as a co-processor to the host system. In this section, we describe the Brook interface to stream hardware.

4.1 Streams

Streams in Brook are created via the `LoadStream` API call:

```
stream s = LoadStream (float, n, data);
```

where *float* is the element type, n is the number of elements of type *float* in the array *data*. Creating a stream copies data to the stream hardware which can in turn be referenced via the variable *s*. The `StoreStream` function fetches stream values back from the stream hardware.

```
StoreStream (s, data);
```

Streams are collections of records which require similar computation. A record can be of any type supported by the hardware, ranging from a single float value to complex structures. Limits on the length of streams is set by the underlying stream hardware. Streams provide the basic data parallel primitive for a stream processor. In order for a stream processor to operate on the element in parallel, the computation required for each stream record should be largely independent of the other records. Examples of streams include particle positions in a particle simulation or flow vectors for each cell in a fluids simulation. These examples require only a local computation and can be updated independently.

4.2 Kernel Functions

The programmer executes functions, or kernels, over each element of the stream. An example kernel declaration is shown in figure 3. The arguments to a kernel include:

- Input streams specified by the `stream` keyword: These variables contains an element from an input stream.
- Output streams (`stream out`): The result of the kernel computation should be placed in these variables by the kernel program.
- Gather streams (`gather stream`): These stream variables permit indexing into the stream data. Elements are fetched from gather streams via C-like array indexing operation i.e. `array[i]`.

```
kernel void updatepos (stream float3 pos,
       stream float3 vel,
       stream out float3 newpos,
       float timestep) {
  newpos = pos + vel*timestep;
}

#include <brook.h>
int main (void) {
  float pos[N][3], vel[N][3], timestep;
  ... initialize pos, vel ...
  kernel updatepos = LoadKernel(``updatepos'');
  stream s_pos = LoadStream(float3, N, pos);
  stream s_vel = LoadStream(float3, N, vel);
  constant c_timestep =
      LoadConstant(float, &timestep);
  KernelMap (updatepos, s_pos, s_vel,
                        s_pos, c_timestep);
  StoreStream(s_pos, pos);
}
```

Figure 3: A sample Brook kernel which updates a set of positions based on a set of velocities and a timestep. The kernel function is stored in a separate file which is compiled by the Brook kernel compiler. In the main program, the kernel is loaded and the streams and constants created. KernelMap executes the kernel over all the elements of *s_pos* and *s_vel* and places the result into the *s_pos* stream. The positions are then copied from the stream to the array *pos*.

- Constants: All non-stream arguments are constants that are set by the host program via the Brook API.

The kernel function is compiled separately from the application using the Brook compiler. The `LoadKernel` function loads a compiled kernel from a file and returns a kernel identifier. `KernelMap` takes as input the kernel, streams, and constants identifiers and executes the kernel on every element of the input streams. The results are placed in the output stream which can be "read back" with the StoreStream command or passed as an input to the next kernel.

Kernel functions are similar to Cg shaders. The body of the kernel is composed of C code. Global memory access is limited to reads inside gather streams, similar to texture access. The kernel execution does not permit any side effects, static variables, or global writes. These restrictions prevent any dependencies between elements in the kernel computation. The streaming hardware can therefore execute multiple copies of the kernel function on different input elements.

Despite these similarities, there are a few key differences from Cg. Kernels operate directly on collections of data from input streams rather than on fragments or vertices. Kernels also make a distinction between data which is streamed as inputs and data gathered with array accesses. This permits the system to manage these streams differently according to their usage patterns. Finally, where Cg only provides a general mechanism for specifying shaders, Brook generalizes the memory management of streams.

Kernel functions are what differentiate stream programming from traditional vector programming. Kernel functions permit arbitrary function evaluation whereas vector operators consist of simple math operations. Kernels capture additional locality of a function by providing local temporary storage and code blocks within the kernel function. A vector operation, in contrast, requires temporaries to always be read and written to a vector register file.

```
reducekernel void matrixmult (
      stream matrix4      left,
      stream matrix4      right,
      stream out matrix4  result) {
  result = left * right;
}

kernel matmult = LoadKernel(``matmult'');
stream s = LoadStream(float[4][4], n, matrices);
float ret[4][4];
KernelReduce (matmult, s, ret);
```

Figure 4: A reduction kernel which performs matrix multiplication. The KernelReduce Brook function executes the matmult reduction over the stream *s* and stores the result in *ret*.

4.3 Reduction

While kernel functions provide a mechanism for applying a function to a set of data, reduction kernels provide a data parallel method to calculate a single value from a set of records. Examples of reduction operations include a simple arithmetic sum, computing the maximum, or more complex operators like matrix multiplication. In order to perform the reduction in parallel, we require the reduction function to be associative: $(a+b)+c = a+(b+c)$. This allows the system to evaluate the reduction in which ever way is best suited for the architecture.

The `KernelReduce` operator takes a reduction kernel and applies it to a single stream to produce a single output. A reduction kernel takes as input two records, which may originate either from the input stream or from partially accumulated results, and produces an updated result. KernelReduce repeatedly applies the reduction until a single element remains.

Figure 4 illustrates an example of a reduction multiplying a stream of matrices together. A reduction kernel declaration is similar to a kernel declaration. The first two stream arguments are the left and right elements to be reduced. The third stream argument is the output value of that reduction. Any other arguments must be constants. To execute the reduction kernel, the host application calls `KernelReduce` function call with the first argument being the stream to reduce.

4.4 Scatter and Gather

The scatter and gather operations provide levels of indirection in reading or 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. Gather operations are supported within the kernel via the `gather stream` keyword. Gather streams are read-only data which can accessed anywhere within the kernel with C-like array access (`a=p[i]`).

The opposite of a gather is a scatter operation. A scatter performs a parallel write operation with reduction (e.g. `p[i]+=a`). The `ScatterOp` function takes four arguments: an index stream specifying where the data is to be scattered, a data stream containing data to scatter, and the destination stream in which the data is written. The final argument is

```
float4 gather4 (texobjRECT t, float i) {
  float2 v;
  v.y = i / WORKSPACE;
  v.x = frac(v.y)*WORKSPACE;
  v.y = floor (v.y);
  return f4texRECT(t, v);
}
```

Figure 5: Cg code for 1D to 2D gather conversion. In order to access 2D textures with a 1D address, the Brook compiler converts a 1D address into a 2D address. The texture width is specified by the constant WORKSPACE.

a user-specified `reducekernel` 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 `KernelReduce` operation. The streaming hardware may reorder the kernel evaluation preserving associativity. If the reducekernel 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 reducekernels, 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 `GatherOp` API call which performs a parallel indirect read with update (e.g. `a=p[i]++`). A user-specified kernel passed to the `GatherOp` function performs an atomic operation on every fetch from a stream. Unlike ScatterOp, the gather kernel is not a reduction operator and cannot reorder gathers from the same location.

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.

5 Implementing on Graphics Hardware

In order to demonstrate that graphics hardware supports the streaming programming model, we have implemented Brook on top of OpenGL and Cg. Our implementation completely hides all references to the graphics API and only exposes the interface outlined in the previous section. The system consists of two components: a kernel compiler, which compiles kernel functions into legal Cg code, and a runtime system built on top of OpenGL which implements the Brook API.

5.1 Streams

Stream data resides in 2D floating point textures. When the application program issues a LoadStream call, the runtime system creates a 2D texture object and copies the data into the texture. The output stream from a kernel is written to a floating point pbuffer and copied back into a texture object. When the host application issues a StoreStream call, the runtime system fetches the texture data from the graphics card into the host memory.

Using textures to store streams of data presents some limitations. The length of a stream is limited by the maximum resolution of 2D textures. Stream types are also restricted to the texture formats supported by the hardware, which consist of floating point vectors of 1 to 4 components specified by the types `float`, `float2`, `float3`, and `float4`.

```
fragout_float main (
        uniform texobjRECT _tex_pos,
        uniform texobjRECT _tex_vel,
        uniform float timestep,
        float4 _pos : WPOS)  {
  fragout_float OUT;
  float3 pos, vel, newpos;

  pos= f3texRECT (_tex_pos, _pos.xy);
  vel= f3texRECT (_tex_vel, _pos.xy);
  {
    newpos = pos + vel*timestep;
  }
  OUT.col.xyz= newpos;
  return OUT;
}
```

Figure 6: A compiled Brook kernel.

Using 1D rather than 2D textures would simplify stream access inside the kernel since accessing gather streams is a 1D address lookup. Graphics hardware, however, limits the size of 1D textures to only a few thousand texels. The Brook kernel compiler introduces code, as shown in figure 5, into the kernel to perform a 1D fetch from within a 2D texture.

Since we are using textures to store streams, all stream access goes through the texture cache. Although Brook distinguishes between input streams and gather streams, graphics hardware makes no distinction between these two texture types. Input streams have a regular access pattern but no reuse since each kernel processes a different input record. Gather streams access patterns are more random but may contain reuse. As Purcell et al. [2002] observed, input stream data can pollute a traditional texture cache which can penalize locality in gather operations.

5.2 Kernel Programs

Kernels compiled using the Brook compiler produce legal Cg code for the NVidia GeForce FX fragment processor. The compiler is built using parser utility Bison and the lexical analyzer Flex [Free Software Foundation, Inc 2003a][Free Software Foundation, Inc 2003b]. These tools parse the Brook kernel code and extract the arguments and code body. These results are passed to a script which outputs a Cg function.

The example output of compiled kernel is shown in figure 6. The compiler wraps the kernel code with Cg code to prepare the variables needed for the kernel body. Both input streams and gather streams arguments are converted into texture fetches. For each input stream, the compiler inserts a texture fetch based on fragment position to get the stream elements for the kernel. For the output stream, the compiler places the output to the color buffer. Constants are declared in the function as `uniform` variables to be set by the runtime environment. The Cg compiler compiles the final assembly output which is loaded by the runtime system.

Using Cg as a compilation target for kernels provides some definite advantages. Kernel programs do not require much modification since the syntax of Cg is based on C. The more difficult compilation phases, like register allocation, are offloaded to the Cg compiler. In addition, the native Cg vector types, like float2, are available to the kernel writer – thus allowing the developer to optimize for graphics hardware's 4-component ALU operations. There are some downsides to

using Cg. Cg does not support all the C native types such as integers and chars, and they are therefore not available within Brook kernels. Also, final performance of the kernel is affected by the quality of the Cg compilation.

The fragment processor provides much of the functionality needed for Brook kernels. Textures provide a fast mechanism for accessing stream data and Cg provides a high level interface for programming. Despite these advantages, there are limitations to using the fragment processor. Branching is not fully supported preventing kernels from performing data dependent looping. Also, there are a limited number of outputs and inputs. On the NVidia GeForce FX, the fragment processor supports up to 16 input textures but only a single output vector. As a result, kernels can only have a single output stream. In addition, kernel size is limited by the number of instructions supported by graphics hardware.

The vertex processor could also be used as a target for kernel functions. Many of the same instructions are available in the vertex processor as well as some additional features such as branching. However, it would prohibit the use of gather streams since texture fetching is not permitted inside of vertex shaders. In addition, the input of a vertex shader must come from the host application. This means that the output of any kernel must be read back to host if it is to be passed to another kernel. As Thompson et al. [2002] observed, this can be a significant performance bottleneck.

In addition, current graphics hardware does not virtualize the hardware resources to allow for larger programs, more local variables, or additional inputs and outputs. On the GeForce FX, fragment programs are limited to 1024 instructions and 32 registers for local storage. Likewise, the number of render outputs and texture inputs to a fragment shader are not extendable. This can severely limit the size and capabilities of kernel functions.

5.3 Kernel Execution

When the application issues a KernelMap call, the runtime system executes the loaded kernel using the fragment processor. The runtime system binds the input textures, sets all the constant variables, and binds the fragment shader corresponding to the kernel. To execute the kernel, the system issues a large quad containing the same number of fragments as elements in the input streams. (An error is raised if the number of elements in the input streams do not match.) The graphics hardware executes the compiled kernel on each fragment using the fragment position to fetch the input element from each input stream texture. The render target is set to an off-screen pbuffer which stores the output result.

5.4 Reduce

Graphics hardware does not have a native method for reductions. The Brook runtime implements reduction via a multi-pass method similar to reduction networks in data parallel architectures [Reynolds et al. 1992]. The reduction is performed in a tree requiring $lg(n)$ parallel steps, where n is the number of elements to reduce. In each pass, the reduce kernel reads two adjacent stream elements, performs the reduction, and outputs a single reduced value. Each pass results in half as many values to reduce. This is repeated until a single element remains which is then read from the pbuffer and returned to the user.

Using this algorithm, we benchmarked reduction performance on the ATI Radeon 9700. To reduce 2^{20} floating point numbers (requiring 20 passes) with a simple sum `reducekernel` took 12 milliseconds. The compiled reduce kernel computes the position of the two stream elements to reduce, performs the 1D to 2D conversion for 2D texture lookup, sums the values and stores the output. One simple extension to graphics hardware would be to provide an internal mechanism to perform reductions. Each parallel processor could maintain its own reduction value in a "reduction register" and perform the final reduction on the host. This would require a single pass over the stream data.

To evaluate potential speedup of this feature, we assume the cost of accessing a reduction register is the same as accessing any other register in the fragment program. To simulate the performance of the reduction kernel used above, we benchmark a simple shader which performs a texture fetch (fetching the stream) and `ADD` instruction. Note that the 1D to 2D conversion is not required since no gather is performed. For the same test case used above, the simulated reduction performance is 1.7 milliseconds.

One could use existing blending hardware available in the fragment pipeline to perform the reduction in a similar manner. However, these units are not programmable nor do they support high precision floating point in current graphics hardware.

5.5 Scatter and Gather

Gather operations are supported natively by the programmable fragment unit. The Brook compiler converts gather array accesses into calls to the Brook `gather` function. The gather function converts the 1D access into a 2D texture lookup and returns the result as shown in figure 5.

ScatterOp and GatherOp operations are not natively supported by the fragment hardware. Brook implements scatter by rendering points into the destination stream as follows. First the destination stream is rendered to an off-screen pbuffer. Next, the system fetches the index stream from the 2D texture. Using the index data, the system renders OpenGL points positioned corresponding to the index value. With the user specified ScatterOp kernel bound, the point fragment performs the reduction by fetching the two values to reduce from texture and writes the result into the pbuffer. Once all of the points have been rendered, the data has been scattered into the destination stream present in the pbuffer. 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 performance drawbacks. 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. Fetching the index stream from the graphics card to host memory is a slow operation. The readback rate for the NVidia GeForce4 Ti4200 on a PIII 800Mhz PC is only 151 MB/sec. This limits the scatter rate to a theoretical peak of only 37.7 Mscatters/second.

Future graphics hardware may soon remove this readback limitation. The recently released DirectX 9 API [Microsoft 2003a] includes functionality for displacement mapping, permitting the texture data to directly control the point's location. Another option may be to allow the fragment processor to set the location of a fragment inside the shader. This would remove the requirement to render OpenGL points and the dependence on the point rendering rate.

The implementation of GatherOp and ScatterOp is fur-

ther complicated by index streams with multiple references to the same stream element. Similar to KernelReduce, the GatherOp and ScatterOp operators cannot perform a reduction in a single pass. To solve this, 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 to 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.

6 Streaming Applications

Using Brook, we have implemented a variety of applications including an FFT, a singular value decomposition kernel, a multi-grid fluid solver, and a finite element PDE solver. In this section, we present a few sample applications which bring out some issues related to graphics hardware.

6.1 Sparse Matrix Multiply

This application performs multiplication of a large sparse matrix with a dense vector. The matrix representation and the kernel are shown figure 7. Each call to the sparse matrix kernel performs a multiplication of the next non-zero element in a row with the corresponding element of the vector. We call the kernel k times, where k is the maximum number of non-zero elements in a row.

To test the effectiveness of sparse matrix multiply on graphics hardware, we examined the instruction usage of the compiled kernel. The compiled Brook kernel was compiled with the NVidia CG compiler for the `NV_FRAGMENT_PROGRAM` fragment program supported by GeForce FX. After hand optimizing the results from the compiler, the 16 total instructions broke down as follows: 6 texture fetch instructions, 4 math instructions corresponding to math expressed in the kernel, 6 instructions for two 1D to 2D conversion for the gathers.

This example demonstrates some of the inefficiencies for storing streams in 2D textures. More instructions are used in computing the 2D conversion than in performing the math operations specified in the kernel. Removing this 3-instruction penalty for gather operations would make this kernel quite compact.

Sparse matrix multiply, while it does provide an interesting application suffers from two main drawbacks. First, the kernel function does not have high arithmetic intensity, only performing 4 math operations but reading 6 words. Second, the code is largely scalar and does not take advantage of the 4-way SIMD operators.

6.2 Sorting

The streaming sort application performs a Batcher bitonic sort [Batcher 1968] to sort n numbers. This sort 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 8.

$$\begin{pmatrix} 3 & 0 & 0 & 2 \\ 0 & 0 & 0 & 1 \\ 0 & 4 & 0 & 0 \\ 6 & 0 & 0 & 8 \end{pmatrix} \begin{pmatrix} 8 \\ 3 \\ 6 \\ 2 \end{pmatrix}$$

elem: (3 2 1 4 6 8)
ipos: (0 3 3 1 0 4)
start: (0 2 3 4)
len: (2 1 1 2)
v: (8 3 6 2)

```
kernel void sparsematrix (stream float prev,
       stream float start, stream float len,
       gather stream float elem,
       gather stream float ipos,
       gather stream float v,
       float pass,
       stream out float result) {
  float offset = start + pass;
  if (pass < len)
    result = prev + elem[offset]*v[ipos[offset]];
  else
    result = prev;
}
```

Figure 7: Streaming sparse matrix multiply requires four streams to represent the matrix, one gather stream for the dense vector, and an input stream of the partial result. *elem* contains the ordered list of non-zero elements of the matrix. *ipos* stores the column index for each element. *start* indicates the position in the elem stream where each row begins. *len* is the number of non-zero elements in that row. The kernel computes which non-zero element to gather from start index for the row plus the pass number. The element is multiplied with the corresponding element from v, unless the pass number is higher than the number elements in the row.

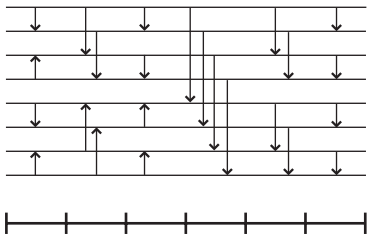

Figure 8: Bitonic Sorting network for 8 elements. Each vertical arrow represents a comparison and swap operation between two elements in the stream. If the values do not obey the direction of the arrow, the two elements are swapped. This example requires 6 kernel executions, specified by the hash marks, which perform the comparison and swap in parallel. In general, bitonic sorting requires $O(lg^2 n)$ kernel executions to sort n numbers.

Because current graphics hardware natively supports only one stream output without an efficient scatter mechanism, performing a swap operation is difficult. To emulate swap, the comparison evaluation is performed twice, once for either side of the arrow in figure 8. 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.

This implementation was ported to the ATI Radeon 9700 using the `ARB_FRAGMENT_PROGRAM` extension [OpenGL ARB 2003]. Sorting a one million element stream consisting of `float4`s with the x component used as a key took 2.5 seconds. If graphics hardware natively supported multiple outputs and scatter, we could improve sorting performance significantly. The kernel could output both elements of the

comparison and place the results in the correct position in the output stream for the next pass. This could potentially double the sorting performance as each pass would require only $n/2$ kernel executions.

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 three instruction macro.

6.3 Scan

Traditional data-parallel computing operators can be implemented with the streaming model. For example, the scan operator, also known as the parallel-prefix, is widely discussed in data parallel computing literature [Ladner and Fischer 1980][Belloch 1986] and used extensively in the Connection Machine architecture [Hillis 1985]. A scan operation computes n partial reductions where each element is the reduction result *up to* that element in the stream. For example a +-scan computes the running sum of a list of numbers. Since the reduction is associative, computing a scan operation requires $lg(n)$ passes over the list of elements.

Scan can be implemented via gather streams and multiple kernel passes using the same algorithm described by Hillis and Steele [1986]. To demonstrate streaming scan, we compute a summed-area table used for image filtering [Crow 1984]. To compute a 1024x1024 area table, we initialize the grid to contain the area of a single cell. We then perform 10 kernel executions ($lg(1024)$) for the horizontal sum scan which computes the partial sums in the horizontal direction. We complete the table by performing and 10 additional kernel executions to scan in the vertical scan.

6.4 Gromacs

Gromacs [Lindahl et al. 2001] is an open source molecular dynamics simulator package widely used in protein folding research. As a test of the generality of streaming computing, we have ported to streaming the more compute intensive Gromacs operations, namely the non-bonded molecular interactions. These forces are computed from the electrostatics and Lennard-Jones components between two molecules given by the equation below [van der Spoel et al. 2002].

$$F_i(\mathbf{r}_{ij}) = \left(\frac{1}{4\pi\epsilon_0} \frac{q_i q_j}{\epsilon_r r_{ij}^2} + 12\frac{C_{12}}{r_{ij}^{12}} - 6\frac{C_6}{r_{ij}^6} \right) \frac{\mathbf{r}_{ij}}{r_{ij}} \qquad (1)$$

To compute forces, Gromacs generates an interaction list indicating for each molecule which other molecules are close enough to impose a significant force. The molecular interaction kernel takes as input the stream of molecules and the interaction list; other gather streams contain molecule properties such as type, charge, and position. The kernel evaluates the force function for the input molecule with each molecule in the interaction list. The output of the kernel is the net force vector on the molecule and the force potential imparted on the molecule.

In order to optimize the computation for the 4-way SIMD architecture of GPUs, we compute the force contributions of four molecules simultaneously. We repeat this computation for the next four molecules. Because of the instruction limit for GPUs, we only compute the force interaction for up to eight molecular pairs per molecule. If the interaction list includes more than eight, we repeat the kernel execution, adding the remaining forces. Gromacs also computes the total force potential in the system. We use the `KernelReduce` operator with a simple summation kernel to stream the potential values returned from the force calculation to a single value.

The Gromacs acceleration structure is divided into a regular 3D grid. The dimensions of each cell is chosen such that molecules only impart a force on other molecules in that cell and neighboring cells. To build the acceleration structure in Brook, we first compute which cell a molecule resides in based on its position. Next, we sort the molecules by grid cell using the bitonic sorting method shown in figure 8. This places the molecules in same grid cell next to each other in the stream. Next, we compute the starting position of each grid cell with a binary search kernel. This kernel takes as input the grid number and performs multiple gathers within the sorted molecule stream searching for the first molecule contained in that grid cell. The output of the search kernel contains the start offset for each grid cell.

Gromacs demonstrates how a large application with lots of potential data parallelism benefit from streaming hardware. The molecular force kernel in particular is ideally for streaming hardware. Data parallelism exists the different force calculations allowing the different computations to happen in parallel. Also, the force kernel has high arithmetic intensity. The compiled Cg force kernel contains a total of 426 fragment instructions with only 49 texture fetch instructions to compute eight molecular interactions. This computation, which typically comprises of over 90% of work performed in Gromacs, can make effective usage of streaming hardware.

7 Discussion

In this paper we have described a programming environment for writing applications for next-generation programmable GPUs. This environment was designed by surveying a large number of data parallel applications and articulating future hardware constraints. Many programs can be run on current hardware, although still not efficiently. However, as the floating point instruction issue rates rapidly increase, there will be strong motivation to port programs to this platform.

Our study began as an attempt to characterize the types of applications that would not map well to graphics hardware. Those without data parallelism, or with low arithmetic intensity will not run well. There is also a set of applications where the parallel algorithms are less efficient than the sequential algorithms. For example, parallel sort and scan requires an extra $\log n$ operations than their sequential versions. This factor means that the GPU is at a disadvantage over the CPU.

One of the biggest issues in GPU programs is that certain kernels cannot be run on the hardware because of resource constraints. These include the number of instructions, the number of registers, the number of vertex interpolants, and the number of outputs. In Chan et al. [2002], an algorithm was developed to subdivide large kernels automatically into smaller kernels. That algorithm handled many resource constraints, but does not work for multiple outputs. Solving this problem is a high priority for future work.

The programming environment we proposed is very clean and general. The major changes in current hardware that are needed to support this model are: (1) The notion of streaming inputs and output of stream records of larger size. Using textures for stream inputs is inefficient, and not enough outputs are available in current hardware. (2) Reduction is poorly supported. (3) Gather is efficiently solved using existing texturing methods (except that the texture caches

are optimized for traditional texture mapping applications). Scatter is currently not implemented efficiently.

Looking further towards the future, many questions remain. There are interesting micro-architectural questions that deserve further research. 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 has 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. Providing such computational power within consumer graphics hardware has the potential redefine the GPU as not just rendering engine, but the principal compute engine for the PC.

References

Anderson, B., Stewart, A., MacAulay, R., and Whitted, T. 1997. Accommodating memory latency in a low-cost rasterizer. In *Proceedings of the 1997 SIGGRAPH/Eurographics workshop on Graphics hardware*, ACM Press, 97–101.

ATI, 2003. RADEON 9700 product web site. http://mirror.ati.com/products/pc/radeon9700pro/index.html.

Batcher, K. E. 1968. Sorting networks and their applications. *Proceedings of AFIPS Spring Joint Computing Conference 32*, 307–314.

Belloch, G., 1986. Parallel prefix versus concurrent memory access.

Bove, V., and Watlington, J. 1995. Cheops: A reconfigurable data-flow system for video processing. In *IEEE Trans. Circuts and Systems for Video Technology*, 140–149.

Chan, E., Ng, R., Sen, P., Proudfoot, K., and Hanrahan, P. 2002. Efficient partitioning of fragment shaders for multipass rendering on programmable graphics hardware. In *Proceedings of the conference on Graphics hardware 2002*, Eurographics Association, 69–78.

Crow, F. C. 1984. Summed-area tables for texture mapping. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, 207–212.

Free Software Foundation, Inc, 2003. Gnu bison web page. http://www.gnu.org/software/bison.

Free Software Foundation, Inc, 2003. Gnu flex web page. http://www.gnu.org/software/flex.

Fuchs, H., Poulton, J., Eyles, J., Greer, T., Goldfeather, J., Ellsworth, D., Molnar, S., Turk, G., Tebbs, B., and Israel, L. 1989. Pixel-planes 5: a heterogeneous multiprocessor graphics system using processor-enhanced memories. In *Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, ACM Press, 79–88.

Gokhale, M., and Gomersall, E. 1997. High level compilation for fine grained fpgas. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, 165–173.

Halfhill, T. 2000. TI cores accelerate DSP arms race. In *Microprocessor Report*. March, 22–28.

Harris, M., Coombe, G., Scheuermann, T., and Lastra, A. 2002. Physically-Based visual simulation on graphics hardware. In *Grahpics Hardware*, 1–10.

Hillis, W. D., and Guy L. Steele, J. 1986. Data parallel algorithms. *Communications of the ACM 29*, 12, 1170–1183.

Hillis, D., 1985. The connection machine. MIT Press.

Igehy, H., Eldridge, M., and Proudfoot, K. 1998. Prefetching in a texture cache architecture. In *Proceedings of the 1998 EUROGRAPHICS/SIGGRAPH workshop on Graphics hardware*, ACM Press, 133–ff.

Kapasi, U., Dally, W. J., Rixner, S., Owens, J. D., and Khailany, B. 2002. The imagine stream processor. *Proceedings of International Conference on Computer Design* (September).

Khailany, B., Dally, W. J., Rixner, S., Kapasi, U. J., Mattson, P., Namkoong, J., Owens, J. D., Towles, B., and Chan, A. 2001. IMAGINE: Media processing with streams. In *IEEE Micro*. IEEE Computer Society Press.

Kozyrakis, C. 1999. A media-enhance vector architecture for embedded memory systems. Tech. Rep. UCB/CSD-99-1059, Univ. of California at Berkeley.

Ladner, R. E., and Fischer, M. J. 1980. Parallel prefix computation. *Journal of the ACM (JACM) 27*, 4, 831–838.

Larsen, E. S., and McAllister, D. 2001. Fast matrix multiplies using graphics hardware. In *Supercomputering 2001*.

Lindahl, E., Hess, B., and van der Spoel, D. 2001. GROMACS 3.0: a pacakge for molecular simulation and trajectory analysis. *Journal of Molecular Modeling*.

Lindholm, E., Kligard, M. J., and Moreton, H. 2001. A user-programmable vertex engine. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, ACM Press, 149–158.

McCool, M. 2000. Smash: A next-generation api for programmable graphics accelerators. Tech. Rep. CS-2000-14, University of Waterloo, August.

Microsoft, 2003. Directx home page. http://www.microsoft.com/windows/directx/default.asp.

Microsoft, 2003. High-level shader language. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/directx/graphics/reference/Shaders/HighLevelShaderLanguage.asp.

Molnar, S., Eyles, J., and Poulton, J. 1992. PixelFlow: High-speed rendering using image composition. In *Computer Graphics (Proceedings of ACM SIGGRAPH 92)*, 231–240.

NVIDIA, 2003. GeForce FX: Product overview. http://www.nvidia.com/docs/lo/2416/SUPP/TB-00653-001_v01_Overview_110402.pdf.

NVIDIA, 2003. NVIDIA Cg language specification. http://developer.nvidia.com/view.asp?IO=cg_specification.

Olano, M., and Lastra, A. 1998. A shading language on graphics hardware: the pixelflow shading system. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, ACM Press, 159–168.

OpenGL ARB, 2003. Arb_fragment_program extension specification. http://oss.sgi.com/projects/ogl-sample/registry/ARB/fragment_program.txt.

Owens, J. D., Dally, W. J., Kapasi, U. J., Rixner, S., Mattson, P., and Mowery, B. 2000. Polygon rendering on a stream architecture. In *Proceedings 2000 SIGGRAPH/EUROGRAPHICS workshop on on Graphics hardware*, ACM Press, 23–32.

Peercy, M. S., Olano, M., Airey, J., and Ungar, P. J. 2000. Interactive multi-pass programmable shading. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., 425–432.

Proudfoot, K., Mark, W. R., Tzvetkov, S., and Hanrahan, P. 2001. A real-time procedural shading system for programmable graphics hardware. *ACM Transactions on Graphics* (August).

Purcell, T. J., Buck, I., Mark, W. R., and Hanrahan, P. 2002. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics 21*, 3 (July), 703–712. ISSN 0730-0301 (Proceedings of ACM SIGGRAPH 2002).

Rathnam, S., and Slavenburg, G. 1996. An architectural overview of the programmable multimedia processor, TM-1. In *Proceedings of Compcon*, 319–326.

Reynolds, P., Pancerella, C. M., and Srinivasan, S. 1992. Making parallel simulations go fast. In *Proceedings of the 24th conference on Winter simulation*, ACM Press, 646–656.

Russell, R. 1978. The Cray-1 computer system. In *Comm. ACM*, 63–72.

Thinking Machines Corporation, 1993. C* reference manual, May.

Thompson, C. J., Hahn, S., and Oskin, M. 2002. Using modern graphics architectures for general-purpose computing: A framework and analysis. *International Symposium on Microarchitecture*.

Torborg, J., and Kajiya, J. T. 1996. Talisman: commodity realtime 3d graphics for the pc. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, ACM Press, 353–363.

Upstill, S., 1990. The renderman companion. Addison-Wesley.

van der Spoel, D., van Buuren, A. R., Apol, E., Meulenhoff, P. J., Tieleman, D. P., alfons L.T.M. Sijbers, Hess, B., Feenstra, K. A., Lindahl, E., van Drunen, R., and Berendsen, H. J. 2002. GROMACS user manual. *Department of Biophysical Chemistry, University of Groningen.*.

von Neuman, J. 1945. First draft of a report on the EDVAC. Tech. Rep. W-670-ORD-492, Moore School of Electrical Engineering, Univ. of Penn., Philadelphia., June.

Woo, M., Neider, J., Davis, T., Shreiner, D., and OpenGL Architecture Review Board, 1999. OpenGL programming guide.