Brook for GPUs: Stream Computing on Graphics Hardware

Ian Buck Tim Foley Daniel Horn Jeremy Sugerman Kayvon Fatahalian Mike Houston Pat Hanrahan

Stanford University

Abstract

In this paper, we present Brook for GPUs, a 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 coprocessor. 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 compute engine compared to the CPU, to determine when the GPU can outperform the CPU for a particular algorithm. We evaluate our system with five applications, the SAXPY and SGEMV BLAS operators, image segmentation, FFT, and ray tracing. For these applications, we demonstrate that our Brook implementations perform comparably to hand-written GPU code and up to seven times faster than their CPU counterparts.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture—Graphics processors D.3.2 [Programming Languages]: Language Classifications—Parallel Languages

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

1 Introduction

In recent years, commodity graphics hardware has rapidly evolved from being a fixed-function pipeline into having programmable vertex and fragment processors. 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. Applications such as linear algebra [Krüger and Westermann 2003], physical simulation, [Harris et al. 2003], and a complete ray tracer [Purcell et al. 2002; Carr et al. 2002] have been demonstrated to run on GPUs.

Originally, GPUs could only be programmed using assembly languages. Microsoft's HLSL, NVIDIA's Cg, and OpenGL's GLslang allow shaders to be written in a high level, C-like programming language [Microsoft 2003; Mark et al. 2003; Kessenich et al. 2003]. However, these languages do not assist the programmer in controlling other aspects of the graphics pipeline, such as allocating texture memory, loading shader programs, or constructing graphics primitives. As a result, the implementation of applications requires extensive knowledge of the latest graphics APIs as well as an understanding of the features and limitations of modern hardware. In addition, the user is forced to express their algorithm in terms of graphics primitives, such as textures and triangles. As a result, general-purpose GPU computing is limited to only the most advanced graphics developers.

This paper presents *Brook*, a programming environment that provides developers with a view of the GPU as a streaming coprocessor. The main contributions of this paper are:

- The presentation of the Brook stream programming model for general-purpose GPU computing. Through the use of streams, kernels and reduction operators, Brook abstracts 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 presentation of a cost model for comparing GPU vs. CPU performance tradeoffs to better understand under what circumstances the GPU outperforms the CPU.

2 Background

2.1 Evolution of Streaming Hardware

Programmable graphics hardware dates back to the original programmable framebuffer architectures [England 1986]. One of the most influential programmable graphics systems was the UNC PixelPlanes series [Fuchs et al. 1989] culminating in the PixelFlow machine [Molnar et al. 1992]. These systems embedded pixel processors, running as a SIMD processor, on the same chip as framebuffer memory. Peercy et al. [2000] demonstrated how the OpenGL architecture [Woo et al. 1999] can be abstracted as a SIMD processor. Each rendering pass implements 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. Thompson et al. [2002] explored the use of GPUs as a general-purpose vector processor by implementing a software layer on top of the graphics library that performed arithmetic computation on arrays of floating point numbers.

SIMD and vector processing operators involve a read, an execution of a single instruction, and a write to off-chip memory [Russell 1978; Kozyrakis 1999]. This results in significant memory bandwidth use. Today's graphics hardware executes small programs where instructions load and store data to local temporary registers rather than to memory. This is a major difference between the vector and stream processor abstraction [Khailany et al. 2001].

The stream programming model captures computational locality not present in the SIMD or vector models through the use of streams and kernels. A *stream* is a collection of records requiring similar computation while *kernels* are

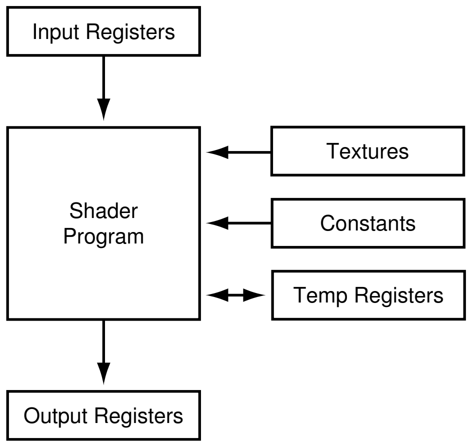


Figure 1: Programming model for current programmable graphics hardware. A shader program operates on a single input element (vertex or fragment) stored in the input registers and writes the execution result into the output registers.

functions applied to each element of a stream. A streaming processor executes a kernel over all elements of an input stream, placing the results into an output stream. Dally et al. [2003] explain how stream programming encourages the creation of applications with high *arithmetic intensity*, the ratio of arithmetic operations to memory bandwidth. This paper defines a similar property called *computational intensity* to compare CPU and GPU performance.

Stream architectures are a topic of great interest in computer architecture [Bove and Watlington 1995; Gokhale and Gomersall 1997]. For example, the Imagine stream processor [Kapasi et al. 2002] demonstrated the effectiveness of streaming for a wide range of media applications, including graphics and imaging [Owens et al. 2000]. The StreamC/KernelC programming environment provides an abstraction which allows programmers to map applications to the Imagine processor [Mattson 2002]. Labonte et al. [2004] studied the effectiveness of GPUs as stream processors by evaluating the performance of a streaming virtual machine mapped onto graphics hardware. The programming model presented in this paper could easily be compiled to their virtual machine.

2.2 Programming Graphics Hardware

Modern programmable graphics accelerators such as the ATI X800XT and the NVIDIA GeForce 6800 [ATI 2004b; NVIDIA 2004] feature programmable vertex and fragment processors. Each processor executes a user-specified assembly-level shader 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, and a few special-purpose instructions.

The basic execution model of a GPU is shown in figure 1. For every vertex or fragment to be processed, the graphics hardware places a graphics primitive in the read-only input registers. The shader is then executed and the results written to the output registers. During execution, the shader has access to a number of temporary registers as well as constants set by the host application.

Purcell et al. [2002] describe how the GPU can be considered a streaming processor that executes kernels, written as fragment or vertex shaders, on streams of data stored in geometry and textures. Kernels can be written using a variety of high-level, C-like languages such as Cg, HLSL, and GLslang. However, even with these languages, applications must still execute explicit graphics API calls to organize data into streams and invoke kernels. For example, stream management is performed by the programmer, requiring data to be manually packed into textures and transferred to and from the hardware. Kernel invocation requires the loading and binding of shader programs and the rendering of geometry. As a result, computation 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. [2002] 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. [2002; 2004] 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, code written today 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.

3 Brook Stream Programming Model

Brook was a developed as a language for streaming processors such as Stanford's Merrimac streaming supercomputer [Dally et al. 2003], the Imagine processor [Kapasi et al. 2002], the UT Austin TRIPS processor [Sankaralingam et al. 2003], and the MIT Raw processor [Taylor et al. 2002]. We have adapted Brook to the capabilities of graphics hardware, and will only discuss Brook in the context of GPU architectures in this paper. The design goals of the language include:

- **Data Parallelism and Arithmetic Intensity**

  By providing native support for streams, Brook allows programmers to express the data parallelism that exists in their applications. Arithmetic intensity is improved by performing computations in kernels.

- **Portability and Performance**

  In addition to GPUs, the Brook language maps to a variety of streaming architectures. Therefore the language is free of any explicit graphics constructs. We have created Brook implementations for both NVIDIA and ATI hardware, using both DirectX and OpenGL, as well as a CPU reference implementation. Despite the need to maintain portability, Brook programs execute efficiently on the underlying hardware.

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.

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

3.1 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 type `float`, Cg vector types such as `float2`, `float3`, and `float4`, and structures composed of these native types. For example, a stream of rays can be defined as:

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

Support for user-defined memory types, though common in general-purpose languages, is a feature not found in today's graphics APIs. Brook provides the user with the convenience of complex data structures and compile-time type checking.

3.2 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 ...
  streamRead(x, X);            // copy data from mem to stream
  streamRead(y, Y);
  saxpy(a, x, y, result);       // execute kernel on all elements
  streamWrite(result, Result);  // copy data from stream to mem
}
```

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 to 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.

Certain restrictions are placed on kernels to allow data-parallel execution. 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);
  }
}
```

Brook forces 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. As Purcell et al. [2002] observed, 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.

The use of kernels 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.

3.3 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.

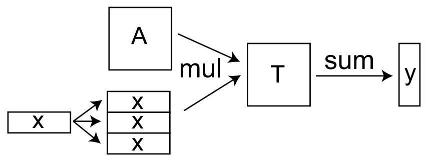
Reductions accept a single input stream and produce as output either a smaller stream of the same type, or a single-element value. Outputs for reductions are specified with the `reduce` keyword. Both reading and writing to the reduce parameter are allowed when computing the reduction of the two values.

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.

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`.

3.4 Additional language features

In this section, we present additional Brook language features which should be mentioned but will not be discussed further in this paper. Readers who are interested in more details are encouraged to read [Buck 2004].

- 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.
- *Iterator streams* are streams containing pre-initialized sequential values specified by the user. Iterators are useful for generating streams of sequences of numbers.
- The Brook language specification also provides a collection of high-level stream operators useful for manipulating and reorganizing stream data, such as grouping elements into new streams and extracting subregions of streams and explicit operators to stride, repeat, and wrap streams. These operators can be implemented on the GPU through the use of iterator streams and gather operations. Their use is important on streaming platforms which do not support gather operations inside kernels.
- The Brook language provides parallel indirect read-modify-write operators called *ScatterOp* and *GatherOp* which are useful for building and manipulating data structures contained within streams. However, due to GPU hardware limitations, we currently perform these operations on the CPU.

4 Implementation on Graphics Hardware

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. The compiler is based on cTool [Flisakowski 2004], 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 A 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. Secondly, 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 direct render-to-texture functionality.

The following sections describe how Brook maps the stream, kernel, and reduction language primitives onto the GPU.

4.1 Streams

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 expressed as dependent texture reads, and the implicit repeat and stride operators are achieved with texture sampling. Current graphics APIs, however, only provide `float`, `float2`, `float3` and `float4` texture formats. To support streams of user-defined structures, BRT stores each member of a structure in a different hardware texture.

Many application writers may wish to visualize the result of a Brook computation. The BRT provides a C++ interface which allows the user to bind Brook streams as native graphics API textures which can be interactively rendered in a traditional graphics application. This option requires that Brook make streams available in a fixed, documented texture layout. By default, streams are stored as a texture with the same dimensions as the stream shape.

A greater challenge is posed by the hardware limitations on texture size and shape. Floating-point textures are limited to two dimensions, and a maximum size of 4096 by 4096 on NVIDIA and 2048 by 2048 on ATI hardware. If we directly map stream shape to texture shape, then Brook programs can not 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 arbitrary-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.

4.2 Kernels

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.

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.[2003] 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 [Percy 2003], which permits direct render-to-texture functionality under OpenGL, should alleviate this restriction.

Program	Instructions		MFLOPS	Slowdown
	texld	arith		
Mat4Mult4	8	16	3611	
Mat4Mult1	20	16	1683	53%
Cloth4	6	54	5086	
Cloth1	12	102	2666	47%

Table 1: This table demonstrates the performance cost of splitting kernels which contain more outputs than supported by the hardware. Included are the instruction counts and observed performance of the matrix multiply and cloth kernels executing on both 4-output hardware and 1-output hardware using the NVIDIA DirectX backend. The slowdown is the relative drop in performance of the non-multiple output implementation.

The task of mapping kernels to fragment shaders is complicated by the limited number of shader outputs available in today's hardware. When a kernel uses more output streams than are supported by the hardware (or uses an output stream of structure type), `brcc` splits the kernel 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 shader outputs. 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 1, 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.

4.3 Reductions

Current graphics hardware does not have native support for reductions. BRT implements reduction via a multipass

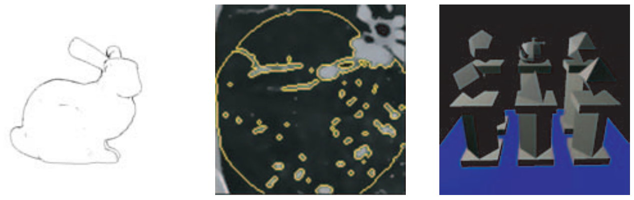


Figure 2: These images were created using the Brook applications FFT, Segment, and Ray Tracer

method similar to Kruger and Westermann [2003]. The reduction is performed in $O(\log n)$ passes, where n is the ratio of the sizes of the input and output streams. For each pass, the reduce operation reads up to 8 adjacent stream elements, and outputs their reduced values. Since each pass produces between 2 and 8 fewer values, Brook reductions are a linear-time computation. The specific size of each reduction pass is a function of the size of the stream and reduction kernel.

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.

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 Evaluation and Applications

We now examine the performance of several scientific applications on GPUs using Brook. 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.

5.1 Applications

We implemented an assortment of algorithms in Brook. The following applications were chosen for three reasons: they are representative of different types of algorithms performed in numerical applications; they are important algorithms used widely both in computer graphics and general scientific computing; optimized CPU- or GPU-based implementations are available to make performance comparisons with our implementations in Brook.

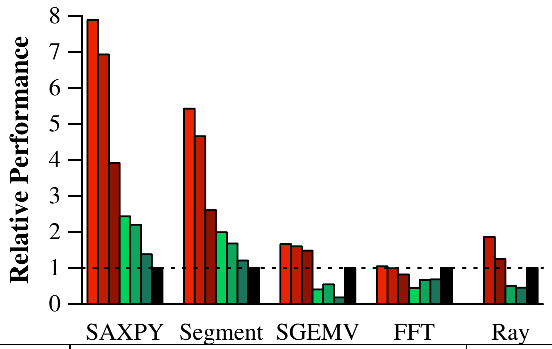
[1]Running 350MHz core and 500Mhz memory

BLAS **SAXPY** and **SGEMV** routines: The BLAS (Basic Linear Algebra Subprograms) library is a collection of low-level linear algebra subroutines [Lawson et al. 1979]. 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 [2004] used to rank the top supercomputers in the world. We compare our performance against that of the optimized commercial Intel Math Kernel Library[Intel 2004] for SAXPY and the ATLAS BLAS library[Whaley et al. 2001] 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.

Segment performs a 2D version of the Perona and Malik [1990] nonlinear, diffusion-based, seeded, region-growing algorithm, as presented in Sherbondy et al. [2003], 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.

FFT: Our Fourier transform application performs a 2D Cooley-Tukey fast Fourier transform (FFT) [1965] 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 [Sullivan et al. 1997]. 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 [Frigo and Johnson 2003] 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 [ATI 2004a]. To compare against the CPU, we benchmark the heavily optimized FFTW-3 software library compiled with the Intel C++ compiler [INTEL 2003].

Ray is a simplified version of the GPU ray tracer presented in Purcell et al. [2002]. This application consists of three kernels, ray setup, ray-triangle intersection (shown in section 3), and shading. For a CPU comparison, we compare against the published results of Wald's [2004] hand-optimized assembly which can achieve up to 100M rays per second on a Pentium 4 3.0GHz processor.



	MFLOPS				RT/sec
	SAXPY	Segment	SGEMV	FFT	Ray
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 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.

Figure 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, we do not include the `streamRead` and `streamWrite` costs.

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. SAXPY illustrates that even a kernel executing only a single MAD instruction is able to out-perform the CPU due to the additional internal bandwidth available on the GPU. FFT was our poorest performing application relative to the CPU. The Brook implementation is only .99 the speed of the CPU version. FFTW blocks the memory accesses to make very efficient use of the processor cache. (Without this optimization, the effective CPU MFLOPS drops to from 1224 to 204.) Despite this blocking, Brook is able to roughly match the performance of the CPU.

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 (refer to 4.2). This is particularly evident with SGEMV test which must perform a multipass reduction operation. The hand coded versions use application specific knowledge to avoid this copy.

We observe that, for these applications, 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, while the peak, observable compute performance favors NVIDIA with 40 billion multiplies per second versus ATI's 33 billion.

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 [2003] 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.

These applications provide perspective on the performance of general-purpose computing on the GPU using Brook. The performance numbers do not, however, include the cost of `streamRead` and `streamWrite` operations to transfer the initial and final data to and from the GPU which can significantly affect the total performance of an application. The following section explores how this overhead affects performance and investigates the conditions under which the overall performance using the GPU exceeds that of the CPU.

5.2 Modeling Performance

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.

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,

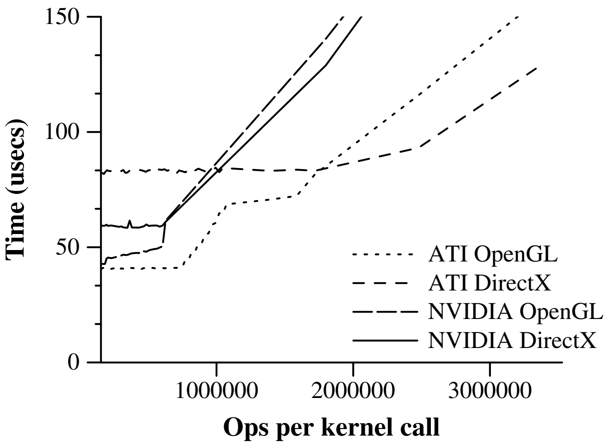


Figure 4: The average cost of a kernel call for various stream lengths with our synthetic kernel. At small sizes, the fixed CPU cost to issue the kernel dominates total execution time. The stair-stepping is assumed to be an artifact of the rasterizer.

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.

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

The idea of computational intensity is similar to arithmetic intensity, defined by Dally et al. [2003] 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.

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, we find this generally not to be the case 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 streams, the CPU may not be able to issue kernels fast enough to keep the GPU busy. Figure 4 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.

Our analysis shows that there are two key application properties necessary for effective utilization of the GPU. First, in order to outperform the CPU, the amount of work performed must overcome the transfer costs which is a function of the computational intensity of the algorithm and the speedup of the hardware. Second, the amount of work done per kernel call should be large enough to hide the setup cost required to issue the kernel. We anticipate that while the specific numbers may vary with newer hardware, the computational intensity, speedup, and kernel overhead will continue to dictate effective GPU utilization.

6 Discussion

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.

Virtualization of hardware constraints can also bring the GPU closer to a streaming processor. Brook virtualizes two aspects which are critical to stream computing, the number of kernel outputs and stream dimensions and size. Multiple output compilation could be improved by searching the space of possible ways to divide up the kernel computation to produce the desired outputs, similar to a generalization of RDS algorithm proposed by Chan et al.[2002]. This same algorithm would virtualize the number of input arguments as well as total instruction count. We have begun incorporating such an algorithm into `brcc` with promising results.

In addition, several features of Brook should be considered for future streaming GPU hardware. Variable outputs allow a kernel to conditionally output zero or more data for each input. Variable outputs are useful for applications that exhibit data amplification, e.g. tessellation, as well as applications which operate on selected portions of input data. We are currently studying these applications and adding this capability into Brook through a multipass algorithm. It is conceivable that future hardware could be extended to include this functionality thus enabling entirely new classes of streaming applications. Secondly, stream computing on GPUs will benefit greatly from the recent addition of vertex textures and floating point blending operations. With these capabilities, we can implement Brook's parallel indirect read-modify-write operators, ScatterOp and GatherOp, which are useful for working with and building data structures stored in streams. One feature which GPUs support that we would like to expose in Brook is the ability to predicate kernel computation. For example, Purcell et al. [2002] is able to accelerate computation by using the GPU's depth test to prevent the execution of some kernel operations.

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 [Brook 2004] 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.

7 Acknowledgments

We would like to thank ATI and NVIDIA for providing access to their hardware, specifically Mark Segal and Nick Triantos. Tim Purcell provided our Brook ray tracing implementation. Bill Mark provided invaluable feedback on our submission. We would also like to thank the following people for their help in the design of the Brook language: Bill Dally, Mattan Erez, Tim Knight, Jayanth Gummaraju, Francois Labonte, Eric Darve, and Massimiliano Fatica from Stanford; Tim Barth and Alan Wray from NASA; Peter Mattson, Ken Mackenzie, Eric Schweitz, Charlie Garrett, Vass Litvinov, and Richard Lethin from Reservoir Labs.

The GPU implementation of Brook is supported by DARPA. Additional support has been provided by ATI, IBM, NVIDIA and SONY. The Brook programming language has been developed with support from Department of Energy, NNSA, under the ASCI Alliances program (contract LLL-B341491), the DARPA Smart Memories Project (contract MDA904-98-R-S855), and the DARPA Polymorphous Computing Architectures Project (contract F29601-00-2-0085). Additional support is provided by the NVIDIA fellowship, Rambus Stanford Graduate fellowship, and Stanford School of Engineering fellowship programs.

A BRCC Code Generation

The following code illustrates the compiler before and after for the SAXPY Brook kernel. The `__fetch_float` and `_stype` macros are unique to each backend. `brcc` also inserts some argument information in the end of the compiled Cg code for use by the runtime. The DirectX assembly and CPU implementations are not shown.

Original Brook code:

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

Intermediate Cg code:

```
void  saxpy (float  alpha, float4  x, float4 y, out float4 result) {
  result = alpha * x + y;
}
void main (uniform float  alpha : register (c1),
           uniform _stype _tex_x : register (s0),
           float2 _tex_x_pos : TEXCOORD0,
           uniform _stype _tex_y : register (s1),
           float2 _tex_y_pos : TEXCOORD1,
           out float4 __output_0 : COLOR0) {
  float4  x; float4  y; float4  result;
  x = __fetch_float4(_tex_x, _tex_x_pos );
  y = __fetch_float4(_tex_y, _tex_y_pos );
  saxpy(alpha, x, y, result );
  __output_0 = result;
}
```

Final C++ code:

```
static const char* __saxpy_fp30[] = {
"!!FP1.0\n"
"DECLARE alpha;\n"
"TEX R0, f[TEX0].xyxx, TEX0, RECT;\n"
"TEX R1, f[TEX1].xyxx, TEX1, RECT;\n"
"MADR o[COLR], alpha.x, R0, R1;\n"
"END \n"
"##!!BRCC\n"
"##narg:4\n"
"##c:1:alpha\n"
"##s:4:x\n"
"##s:4:y\n"
"##o:4:result\n"
"##workspace:1024\n"
"##!!multipleOutputInfo:0:1:\n"
"",NULL};
void  saxpy (const float  alpha,
             const ::brook::stream& x,
             const ::brook::stream& y,
             ::brook::stream& result) {
  static const void *__saxpy_fp[] = {
     "fp30", __saxpy_fp30, "ps20", __saxpy_ps20,
     "cpu", (void *) __saxpy_cpu, NULL, NULL };
  static __BRTKernel k(__saxpy_fp);
  k->PushConstant(alpha);
  k->PushStream(x);
  k->PushStream(y);
  k->PushOutput(result);
  k->Map();
}
```

References

ATI, 2004. Hardware image processing using ARB_fragment_program. http://www.ati.com/developer/sdk/RadeonSDK/Html/Samples/OpenGL/HW_Image_Processing.html.

ATI, 2004. Radeon X800 product site. http://www.ati.com/products/radeonx800.

Bolz, J., Farmer, I., Grinspun, E., and Schröder, P. 2003. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Trans. Graph. 22*, 3, 917–924.

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

Brook, 2004. Brook project web page. http://brook.sourceforge.net.

Buck, I. 2004. Brook specification v.0.2. Tech. Rep. CSTR 2003-04 10/31/03 12/5/03, Stanford University.

Carr, N. A., Hall, J. D., and Hart, J. C. 2002. The Ray Engine. In *Proceedings of Graphics hardware*, Eurographics Association, 37–46.

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 Graphics hardware*, Eurographics Association, 69–78.

Cooley, J. W., and Tukey, J. W. 1965. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation 19* (April), 297–301.

Dally, W. J., Hanrahan, P., Erez, M., Knight, T. J., Labont, F., Ahn, J.-H., Jayasena, N., Kapasi, U. J., Das, A., Gummaraju, J., and Buck, I. 2003. Merrimac: Supercomputing with Streams. In *Proceedings of SC2003*, ACM Press.

Dongarra, J. 2004. Performance of various computers using standard linear equations software. Tech. Rep. CS-89-85, University of Tennessee, Knoxville TN.

England, N. 1986. A graphics system architecture for interactive application-specific display functions. In *IEEE CGA*, 60–70.

Flisakowski, S., 2004. cTool library. http://ctool.sourceforge.net.

Frigo, M., and Johnson, S. G., 2003. benchFFT home page. http://www.fftw.org/benchfft.

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 *Computer Graphics (Proceedings of ACM SIGGRAPH 89)*, 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.

Harris, M. J., Baxter, W. V., Scheuermann, T., and Lastra, A. 2003. Simulation of cloud dynamics on graphics hardware. In *Proceedings of Graphics hardware*, Eurographics Association, 92–101.

INTEL, 2003. Intel software development products. http://www.intel.com/software/products/compilers.

Intel, 2004. Intel math kernel library. http://www.intel.com/software/products/mkl.

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).

Kessenich, J., Baldwin, D., and Rost, R., 2003. The OpenGL Shading Language. http://www.opengl.org/documentation/oglsl.html.

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.

Krüger, J., and Westermann, R. 2003. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Trans. Graph. 22*, 3, 908–916.

Labonte, F., Horowitz, M., and Buck, I., 2004. An evaluation of graphics processors as stream co-processors. Unpublished.

Lawson, C. L., Hanson, R. J., Kincaid, D. R., and Krogh, F. T. 1979. Basic Linear Algebra Subprograms for Fortran usage. *ACM Trans. on Mathematical Software 5*, 3 (Sept.), 308–323.

Lindholm, E., Kligard, M. J., and Moreton, H. 2001. A user-programmable vertex engine. In *Proceedings of SIGGRAPH 2001*, ACM Press/Addison-Wesley Publishing Co., 149–158.

Mark, W. R., Glanville, R. S., Akeley, K., and Kilgard, M. J. 2003. Cg: A system for programming graphics hardware in a C-like language. *ACM Trans. Graph. 22*, 3, 896–907.

Mattson, P. 2002. *A Programming System for the Imagine Media Processor.* PhD thesis, Stanford University.

McCool, M. D., Qin, Z., and Popa, T. S. 2002. Shader metaprogramming. In *Proceedings of Graphics hardware*, Eurographics Association, 57–68.

McCool, M., Du Toit, S., Popa, T., Chan, B., and Moule, K. 2004. Shader algebra. *ACM Trans. Graph.*.

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 *Proceedings of ACM SIGGRAPH 92*, ACM Press, 231–240.

Moreland, K., and Angel, E. 2003. The FFT on a GPU. In *Proceedings of Graphics hardware*, Eurographics Association, 112–119.

NVIDIA, 2004. GeForce 6800: Product overview. http://nvidia.com/page/geforce_6800.html.

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 of 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 ACM SIGGRAPH 2000*, ACM Press/Addison-Wesley Publishing Co., 425–432.

Percy, J., 2003. OpenGL Extensions. http://mirror.ati.com/developer/SIGGRAPH03/ Percy_OpenGL_Extensions_SIG03.pdf.

Perona, P., and Malik, J. 1990. Scale-space And Edge Detection Using Anisotropic Diffusion. *IEEE Trans. on Pattern Analysis and Machine Intelligence 12*, 7 (June), 629–639.

Purcell, T. J., Buck, I., Mark, W. R., and Hanrahan, P. 2002. Ray tracing on programmable graphics hardware. *ACM Trans. Graph.*, 703–712.

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

Sankaralingam, K., Nagarajan, R., Liu, H., Huh, J., Kim, C., D.Burger, Keckler, S., and Moore, C. 2003. Exploiting ILP, TLP, and DLP using polymorphism in the TRIPS architecture. In *30th Annual International Symposium on Computer Architecture (ISCA)*, 422–433.

Sherbondy, A., Houston, M., and Napel, S. 2003. Fast volume segmentation with simultaneous visualization using programmable graphics hardware. *IEEE Visualization*.

Sullivan, W., Werthimer, D., Bowyer, S., Cobb, J., Gedye, D., and Anderson, D. 1997. A new major SETI project based on Project Serendip data and 100,000 personal computers. In *Astronomical and Biochemical Origins and the Search for Life in the Universe, Proceedings of the Fifth International Conference on Bioastronomy*, Editrice Compositori, C. Cosmovici, S. Bowyer, and D. Wertheimer, Eds.

Taylor, M. B., Kim, J., Miller, J., Wentzlaff, D., Ghodrat, F., Greenwald, B., Hoffmann, H., Johnson, P., Lee, J.-W., Lee, W., Ma, A., Saraf, A., Seneski, M., Shnidman, N., Strumpen, V., Frank, M., Amarasinghe, S., and Agarwal, A. 2002. The raw microprocessor: A computational fabric for software circuits and general purpose programs. In *IEEE Micro*.

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*.

Wald, I. 2004. *Realtime Ray Tracing and Interactive Global Illumination.* PhD thesis, Saarland University.

Whaley, R. C., Petitet, A., and Dongarra, J. J. 2001. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing 27*, 1–2, 3–35.

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