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 traditional C to include simple data-parallel constructs, enabling the GPU as a general purpose streaming processor. We present a compiler and runtime system that compiles and executes Brook code on the GPU and 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 provide a framework for establishing when and how the GPU can outperform the CPU for a particular algorithm. We demonstrate that applications written in Brook perform comparably to their hand-written GPU counterparts.

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

Keywords: Programmable Graphics Hardware, Data Parallel Computing, Stream 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 [Kruger 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 alleviate some of this burden by allowing 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 configuring other aspects of the graphics pipeline, such as allocating texture memory, loading shader programs, or constructing graphics primitives. As a result, the implementation of general 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 programming 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 include:

- The presentation of the Brook stream programming model for general purpose GPU programming. 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. Using applications written in Brook, we apply the cost model using the latest ATI and NVIDIA graphics hardware.

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.

The SIMD and vector processing steps involve a read, 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 temporary local registers rather than to memory. This is the key difference between the vector processor abstraction and the 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

*{ianbuck, tfoley, danielrh, yoel, kayvonf, mhouston, hanrahan}@graphics.stanford.edu

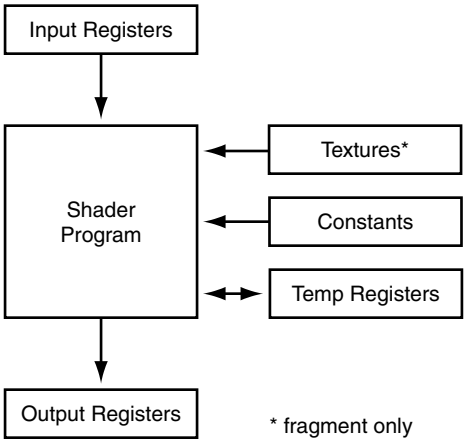


Figure 1: Programming model for current programmable graphics hardware. A shader program operates on a single input element (vertex or fragment) stored in the input registers and writes the execution result into the output registers.

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] explains how stream programming promotes the creation of applications with high *arithmetic intensity*, the ratio of arithmetic operations to required memory bandwidth. This paper applies the idea of arithmetic intensity to a comparison of 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 Radeon 9800 and the NVidia GeForce FX [ATI 2003; NVIDIA 2003] 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 (fragment programs only), 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] describes 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. Instead of raw assembly, Cg, HLSL, and GLslang provide programmers the ability to write kernels in a high level, C-like language. However, even with these languages, applications must still execute explicit graphics API calls to organize data into streams and invoke kernels. For example, the API's texturing system is fully exposed to the programmer, requiring streams 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 length, number of shader outputs, and texture sizes. There has been some work in shading languages which attempts to alleviate some of these constraints. Chan et al. [2002] presented an algorithm to subdivide large shaders automatically into smaller shaders to circumvent shader length and input constraints, but did 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++. However, Sh is intended specifically for the purpose of shading and is not a general-purpose language.

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 general-purpose applications onto graphics hardware.

3 Brook Stream Programming Model

Brook was a developed as a language for streaming processors such as 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 reflect 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 running on both NVIDIA and ATI hardware, using both DirectX with HLSL and OpenGL with Cg, 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/HLSL vector types such as `float2`, `float3`, and `float4`, and structures composed of these native types. For example, we can specify:

```
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 existing high-level graphics languages. 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, primitive types.

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 accessing 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.

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 as result, input stream data can pollute a traditional texture 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. Arithmetic intensity is increased since only the final result of the kernel computation is written back to memory.

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);
  }
}
```

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.

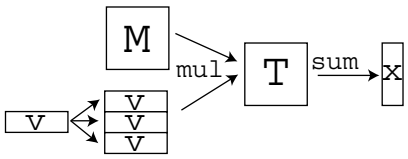
Reductions accept a single input stream and produce as output either a smaller stream of the same type, or a single-element value. Outputs for reductions are specified with the `reduce` keyword. Both reading and writing to the reduce parameter are allowed when computing the reduction of the two values.

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 $x = Mv$.

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

```
float M<50,50>;
float v<1,50>;
float T<50,50>;
float x<50,1>;
...
mul(M,v,T);
sum(T,x);
```



In this example, we first multiply `M` by `v` with the `mul` kernel. Since `v` is smaller than `T` in the first dimension, the elements of `v` 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 2nd dimension of `T` and `x`.

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 a complete language specification are encouraged to read [Anonymous 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 a 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 must 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: the compiler, `brcc`, a source-to-source translator, 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, and was modified to support Brook language primitives. The compiler builds a parse tree, applies transformations to map Brook kernels into GPU assembly, and emits C++ code which uses 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. BRT and `brcc` currently supports three backends; an OpenGL backend for the NVIDIA GeForceFX, a DirectX 9 backend targeting ATI 9800 hardware, and a reference CPU implementation. Creating a runtime for both DirectX and OpenGL has a number of benefits. NVIDIA's extensions to OpenGL allow us to implement less expensive stream gathers and exceed 64 instructions in a fragment shader, while DirectX allows us to render directly to textures with low overhead. Implementing Brook on two graphics APIs also demonstrates the portability of the language.

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

4.1 Streams

Brook maps streams to floating point textures on the graphics hardware. Some Brook language features have obvious implementations under this mapping; 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 stretching and sub-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.

A greater challenge is posed by 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 virtualize this constraint by wrapping stream data across multiple rows of a texture. This allows the user to allocate streams of up to four dimensions that contain as many elements as texels in the largest 2D texture. Unfortunately, when using this approach the texture coordinates of a stream element no longer coincide with its position in the stream. In order to translate between stream positions and texture coordinates, `brcc` introduces address-translation code into kernels. Specifically, address translation is applied to all gather operations and stream arguments that may differ in shape from the output stream. Cg code used for stream-to-texture address translation is shown below.

```
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. Thus our implementation effectively increases the maximum 1D stream size for a portable Brook program from 2048 to 131072 elements. This limitation points to the need for higher floating-point precision or integer instruction sets in future programmable GPUs.

4.2 Kernels

Kernels are mapped to shaders for the GPU fragment processor. `brcc` transforms the body of a kernel into shader code. Stream arguments are initialized from textures, gather operations are replaced with texture fetches, and non-stream arguments are passed via constant registers. The NVIDIA Cg compiler or the Microsoft HLSL compiler is then applied to the resulting Cg/HLSL 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 target. 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 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 the CG and HLSL compilers to remove any computation that does not contribute to the outputs written in that pass.

Program	Instructions		MFLOPS	Slowdown
	texld	arith		
Mat4Mult4	8	16	8686	
Mat4Mult1	20	16	3884	45%
Cloth4	6	54	6445	
Cloth1	12	102	4031	63%

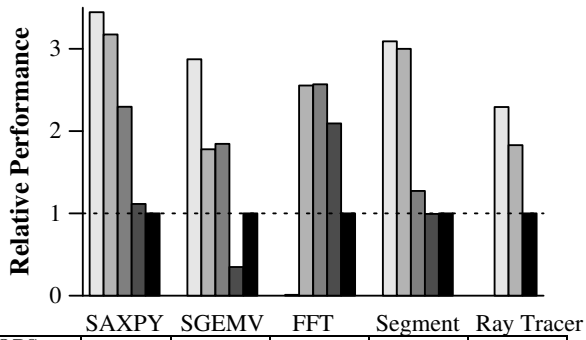
Table 1: Instruction counts and effective throughput with and without hardware support for multiple outputs. The texld and arithmetic instruction counts illustrate the **total** number of instructions to produce all kernel outputs on the DirectX 9 backend. The MFLOPS results are the effective performance of completing all of the operations as specified in the original kernel source. The "slowdown" is the relative performance of the non-multiple output implementation.

To confirm 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 degree of correlation between kernel outputs. For the Mat4Mult kernel, the outputs are not strongly correlated, and the HLSL compiler correctly identified that each row of the output matrix can be computed independently, so 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 both depend on most of the kernel code (a force calculation), and are thus strongly correlated. 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 method similar to Kruger, Westermann [2003]. The reduction is performed in $\log_2$ n passes, where n is the ratio of the sizes of the input and output streams. For each pass, the reduce operation reads pairs of adjacent stream elements, and outputs their reduced values. Since each pass results in half as many values, Brook reductions are a linear-time computation.



MFLOPS	SAXPY	SGEMV	FFT	Segment	Ray Tracer
ATI Hand	2387	2353	-	8058	10998
ATI Brook	2199	1456	521	7820	8775
NV Hand	1590	1511	524	3318	-
NV Brook	772	285	427	2590	-
CPU	692	818	204	2607	4797

Figure 2: Comparing a variety of applications between the Brook GPU, hand coded GPU, and CPU versions.

We have benchmarked a sum reduction of 2^{20} `float4` elements as taking 21.2 and 6.6 milliseconds, respectively, on our NVIDIA and ATI backends. An optimized CPU implementation performed this reduction in only 14.6 milliseconds. The performance difference between the ATI and NVIDIA implementations is largely due to the cost of copying results from the output Pbuffer to a texture, as described above, after each pass in the OpenGL backend. The proposed Superbuffer specification [Percy 2003], which permits direct render-to-texture functionality under OpenGL, should alleviate this performance penalty.

With our multipass implementation of reduction, the GPU must access 3 times as much memory as 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 on the same stream as above yields theoretical reduction times of 8.1 milliseconds for NVIDIA hardware and 3.6 milliseconds for ATI hardware with our proposed modification.

5 Evaluation and Applications

We now examine the effectiveness of general-purpose computing on GPUs using Brook. For each of our tests, we evaluated Brook using two backends, **ATI**, a DirectX 9 backend running on the ATI Radeon 9800 XT and **NVIDIA**, an OpenGL backend running on the NVIDIA GeForceFX 5900 Ultra. Both systems use a 3 GHz Intel Pentium 4 processor, Intel 875P chipset with 8x AGP, running Windows XP. These same systems were used whenever comparing to the CPU performance unless otherwise noted.

5.1 Applications

We implemented an assortment of real-world algorithms in Brook. The following applications were chosen for three reasons: they are representative of different types of operations 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 perform performance comparisons with our implementations in Brook.

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 routines in the LINPACK Top500 benchmark [2004] uses to establish the top supercomputers in the world. We compare our performance on vectors of length 2048^2 against that of the optimized commercial Intel Math Kernel Library available at [Intel 2004].

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 fast post-processing of images in the framebuffer, as well as scientific applications such as the SETI@home [W.T. Sullivan et al. 1997] project. Our implementation uses three kernels: 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 2003a] project, is equal to $channels{*}5{*}(w{*}h){*}log_2(w{*}h)$. To benchmark Brook against a competitive GPU algorithm, we compare our results with the custom NVIDIA GPU implementation released by Moreland and Angel [2003]. To compare against the CPU, we benchmark the FFTW-3 software library [1] [1998; 2003b].

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 GPU and CPU implementations executed on our test systems. Each iteration of the segmentation evolution kernel requires 30 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.

Ray Tracer 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 Wald [2004] C ray-triangle code which averages 41 million ray-triangle intersections per second for a Pentium 4 3.0GHz processor. We also compare against Purcell's ray-triangle intersection rates presented at SIGGRAPH 2002 which were measured on a Radeon 9700, which has a 28% slower clock rate.

Figure 2 provides a breakdown of the performance of our various test applications. For each application, we compare Brook performance against hand-optimized native

[1] compiled with the Intel C++ compiler [INTEL 2003]

GPU code on both ATI and NVIDIA, as well as against the CPU implementation. We compute the effective MFLOPS of the application based on application timings and the number of floating point operations as specified in the original source, not the final assembly. These results do not include any `streamRead` and `streamWrite` costs. The bars represent the performance normalized by the CPU results.

The Brook ATI versions of all the applications perform roughly 2-3 times faster than the equivalent CPU implementations. In addition, Brook achieves performance close to that of the hand-coded GPU implementations (92% SAXPY (ATI), 82% FFT (NVIDIA), 97% Segment (ATI)). The two outlier applications were SAXPY and SGEMV with the NVIDIA backend. This was largely due to the need to copy the output data from the OpenGL pbuffer into a texture (refer to 4.2. The hand coded versions use application specific knowledge to avoid this copy).

The NVIDIA OpenGL and the ATI DirectX Brook implementations significantly differ in performance, despite both cards being the latest models from their respective vendors. In addition to the previously mentioned OpenGL copy issue, the NVIDIA implementation seems to perform poorly on the large kernels used in Segment. Labonte et al. [2004] illustrates how the number of live registers in a kernel can severely impact performance on NVIDIA hardware. Segment compiles to use 8 32-bit floating point registers which, according to Labonte, can limit the NVIDIA hardware to a peak performance of only 4,000 MFLOPS. Given the number of gather operations required for the Segment kernel, our results are in line with his results.

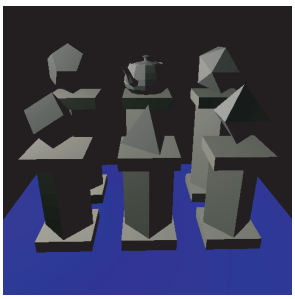
In addition, we derive some conclusions from analysis of the individual applications. SAXPY illustrates that even a kernel executing only a single MAD instruction is able to out-perform the CPU due to the additional internal bandwidth available on the GPU. Segment demonstrates that a straightforward Brook implementation of an algorithm can outperform a time-consumingly hand-optimized CPU version.

The Brook FFT implementation benchmarks comparably to Mooreland and Angel's GPU FFT implementation and outperforms the native FFTW rate shown in Figure 2. However, the FFTW implementation experiences enormous gain when configured to benchmark the CPU and choose the most cache-friendly read/write patterns. FFTW includes a large repository of hand-optimized assembly code and benchmarks the hardware to select the fastest path. This "superoptimization" can raise the performance of FFTW from 204 MFLOPS to 1,224 MFLOPS, which outperforms both GPU implementations. A similar trend is observed when we use the hand-optimized assembly of Wald's ray tracer code, which can achieve up to 100M rays per second (13,500 effective MFLOPS). These two applications indicate that, in some cases, hand-optimized assembly code for today's CPUs can still outperform the Brook GPU code. It is possible that through expert knowledge of the GPU architecture, we could apply the same labor intensive optimizations to our GPU code and outperform these applications. This points to possible future work revolving around automatically benchmarking and optimizing GPU kernels based on texture cache and kernel call patterns, which are needed to better compare against hand-optimized CPU code.

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 at which the overall performance using the GPU exceeds that of the CPU.

FFT: Edge Detect

Segment: Lung CT



Ray Tracer: Glassner

Figure 3: These images were created using the Brook applications FFT, Segment, and Ray Tracer

5.2 GPU vs. CPU

The general structure of many GPU 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. A 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.

$$\begin{aligned} T_{gpu}(i, l_r, l_w) &= l_r/R + i/K_{gpu} + l_w/W \\ T_{cpu}(i) &= i/K_{cpu} \end{aligned}$$

where T_{gpu} and T_{cpu} are running times on the GPU and CPU respectively, R and W are the bandwidth rates for `streamRead` and `streamWrite`, l_r and l_w are the number of floats transfered, i is the total number of instructions executed, and K_{cpu} and K_{gpu} are the execution rates.

The GPU will outperform the CPU when $T_{gpu} < T_{cpu}$, i.e. when the work performed per float transferred is sufficient that the GPU's computational advantage hides the data transfer cost. This relationship is the *arithmetic intensity*, $\alpha \equiv i/l$, of the algorithm. The higher the arithmetic intensity of an algorithm, the better suited it is for computing on the GPU.

We created a synthetic workload to easily explore the parameters of the model. It is designed as follows:

```
float4 inStream<length, length>, outStream<length, length>;

streamRead(inStream, inData);
```

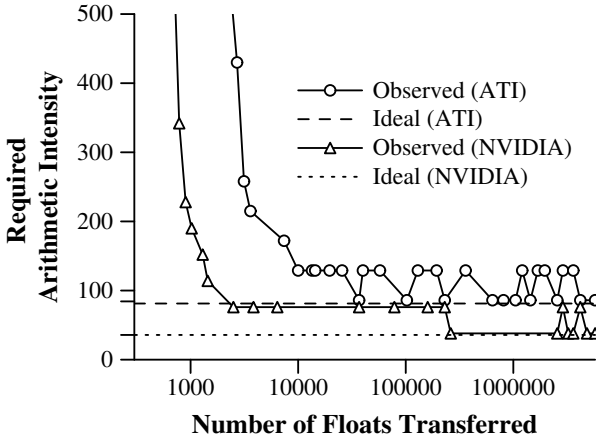


Figure 4: The minimum observed arithmetic intensity required for the GPU to complete faster than the CPU (including data transfer). The horizontal lines are the calculated expected arithmetic intensities.

```
for (i = 0; i < numIters, i++) {
   runKernel(float4(1.0f, 1.0f, 1.0f, 1.0f), inStream, outStream);
}
streamWrite(outStream, outData);
```

`runKernel` is a simple kernel that executes many floating point operations on a stream. We benchmarked against the same number of SSE instructions on the CPU.

With our synthetic workload, l_r and l_w are equal ($l = l_r = l_w$) and i is the product of l * (kernel length) * (number of iterations). We derive the arithmetic intensity where we expect the GPU to outperform the CPU as:

$$\alpha \quad > \quad \frac{R^{-1} + W^{-1}}{{K_{cpu}}^{-1} - {K_{gpu}}^{-1}}$$

With a 2^{20} element stream and 1000 kernel iterations, we experimentally determined T_{cpu}, T_{gpu}, W, R, K_{cpu}, and K_{gpu}. [2] [3]

	ATI	NVIDIA	CPU
R Mfloats/sec	138.2	151.8	–
W Mfloats/sec	13.5	44.1	–
K MFLOPS	10384	4148	943
α FLOPs/float	84.3	35.7	–

We also experimentally obtained the arithmetic intensity required for $T_{gpu} < T_{cpu}$ by increasing the number of iterations for varying l and comparing times. Figure 4 shows the required arithmetic intensity against the line $y = \alpha$.

Both runtimes trend, albeit noisily[4], towards the values predicted by the model. This indicates that our simple linear model is reasonable for evaluating the CPU versus the GPU with large datasets.

[2]The ATI W and R timings were particularly noisy. As a result, we chose the median value over 20 different test runs.

[3]The MFLOPS numbers reflect this particular workload, not the peak hardware performance. The NVIDIA numbers in particular suffer dramatically from using multiple distinct registers per calculation.

[4]The noise is due to the correlation between the transfer cost, a function of the stream length, and the kernel execution time, which is also dependent on the stream length.

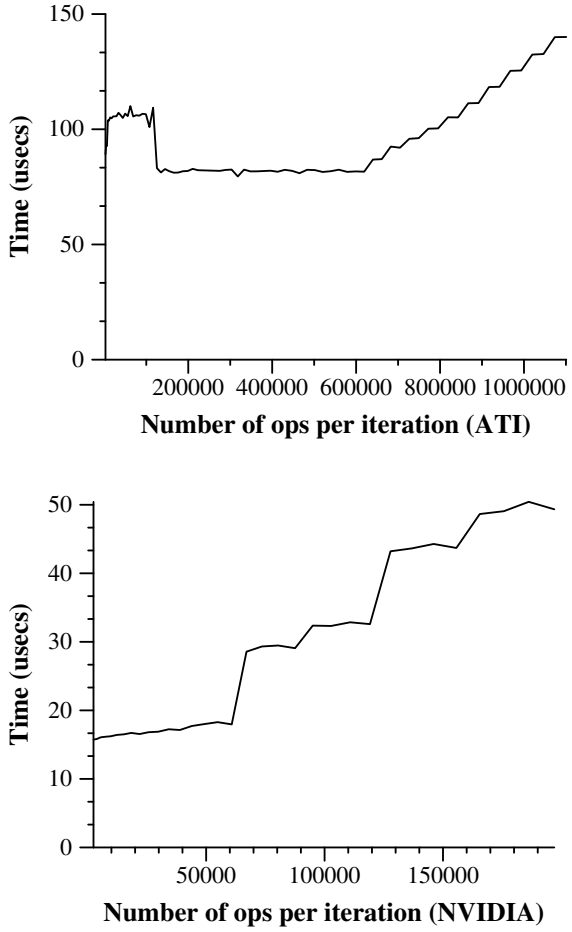


Figure 5: The average cost of a kernel call for various stream lengths and 1000 iterations with out synthetic kernel. At small sizes, the fixed CPU cost to issue the kernel dominates total execution time. The stair-stepping found in both graphs is assumed to be an artifact of the rasterizer.

There are two reasons why our cost model does not work for short streams. The first, and less significant, is that `streamRead` and `streamWrite` are not linear. GPUs are more efficient at transferring data in mid to large sized amounts. The major factor in the degradation is that the overhead of issuing a kernel limits the overall GPU's performance, preventing the GPU from outperforming the CPU. Every kernel invocation incurs a certain non-trivial 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 be done in parallel 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 5 shows the average execution time of 1,000 iterations of our 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, the NVIDIA runtime crosses above the knee when executing around 61,000 floating point operations, and ATI, using DirectX, crosses around 670,000 floating point operations.

Our analysis shows that there are two key application

properties necessary for effective utilization of the GPU. First, the arthimetic intensity, the amount of work performed compared to the amount of data transfered, must be high enough to outperform the CPU (35.7 vs. 84.3 floating point operations per float transfered for our NVIDIA and ATI test hardware). Second, the amount of work done per kernel call should be large enough to hide the setup cost required to issue the kernel (670K vs. 61K floating point operations per kernel call for NVIDIA and ATI). We anticipate that while the specific numbers may vary with newer hardware, the arthimetic intensity and kernel overhead will continue to dictate effective GPU utilization.

6 Discussion

The Brook programming environment enables programmers to use the GPU as a streaming coprocessor. As the hardware continues to advance, it will be interesting to examine potential GPU modifications to improve its effectiveness as a streaming coprocessor. Our arithmetic intensity analysis demonstrated that read/write bandwidth is critical 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 l_r/R, i/K_{gpu}, and l_w/W, a much more favorable expression.

Virtualization of hardware constraints can also bring the GPU closer to a general purpose 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. However, there are some improvements which can only be made in hardware, specifically improved floating point precision for ATI hardware and support for integer operations in general.

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. Currently, we support this capability in Brook through a multipass algorithm, but 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 could benefit greatly from added support for vertex textures and floating point blending operations. With these capabilities, we could 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 and 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 general purpose computing on GPUs. Brook for GPUs has been released as an open source project and our hope is that this effort will make it easier for application developers to capture the performance benefits of stream computing on the GPU for the graphics community and beyond. By providing easy access to the computational power within consumer graphics hardware, stream computing has the potential to redefine the GPU as not just a rendering engine, but the principle compute engine for the PC.

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 9 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

Anonymous. 2004. Brook specification v.0.2. Tech. rep.

ATI, 2003. Radeon 9800 technical specification. http://www.ati.com/products/radeon9800/radeon9800pro/specs.html.

Bolz, J., Farmer, I., Grinspun, E., and Schroder, 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. In *IEEE Trans. Circuts and Systems for Video Technology*, 140–149.

Carr, N. A., Hall, J. D., and Hart, J. C. 2002. The ray engine. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on 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 the conference on Graphics hardware 2002*, 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 Supercomputing 2003*, 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. 1998. FFTW: An adaptive software architecture for the FFT. In *Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing*, IEEE, vol. 3, 1381–1384.

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

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

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.

Harris, M. J., Baxter, W. V., Scheuermann, T., and Lastra, A. 2003. Simulation of cloud dynamics on graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on 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.

Kruger, 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. Submitted to ISCA 2004.

Lawson, C. L., Hanson, R. J., Kincaid, D. R., and Krogh, F. T. 1979. Basic Linear Algebra Subprograms for Fortran usage. *ACM Transactions 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 the 28th annual conference on Computer graphics and interactive techniques*, ACM Press, 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 Transactions on Graphics 22*, 3 (July), 896–907.

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

McCool, M. D., Qin, Z., and Popa, T. S. 2002. Shader metaprogramming. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, Eurographics Association, 57–68. revised version.

McCool, M. D., Moule, K., and Toit, S. D., 2004. Sh: Embedded metapgramming language. http://libsh.sourceforge.net/.

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.

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

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

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.

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 Transactions 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 Transactions on Graphics 21*, 3 (July), 703–712. ISSN 0730-0301 (Proceedings of ACM SIGGRAPH 2002).

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*.

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.

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

W.T. Sullivan, I., 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.