Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses

David Tarditi   Sidd Puri   Jose Oglesby

Microsoft Research

{dtarditi,siddpuri,joseogl}@microsoft.com

Abstract

GPUs are difficult to program for general-purpose uses. Programmers can either learn graphics APIs and convert their applications to use graphics pipeline operations or they can use stream programming abstractions of GPUs. We describe Accelerator, a system that uses data parallelism to program GPUs for general-purpose uses instead. Programmers use a conventional imperative programming language and a library that provides only high-level data-parallel operations. No aspects of GPUs are exposed to programmers. The library implementation compiles the data-parallel operations on the fly to optimized GPU pixel shader code and API calls. We describe the compilation techniques used to do this. We evaluate the effectiveness of using data parallelism to program GPUs by providing results for a set of compute-intensive benchmarks. We compare the performance of Accelerator versions of the benchmarks against hand-written pixel shaders. The speeds of the Accelerator versions are typically within 50% of the speeds of hand-written pixel shader code. Some benchmarks significantly outperform C versions on a CPU: they are up to 18 times faster than C code running on a CPU.

Categories and Subject Descriptors   D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming; D.3.4 [*Programming Languages*]: Processors—Compilers

General Terms Measurement, Performance, Experimentation, Languages

Keywords   Graphics processing units, data parallelism, just-in-time compilation

1. Introduction

Highly programmable graphics processing units (GPUs) became available in 2001 [10] and have evolved rapidly since then [15]. GPUs are now highly parallel processors that deliver much higher floating-point performance for some workloads than comparable CPUs. For example, the ATI Radeon x1900 processor has 48 pixel shader processors, each of which is capable of 4 floating-point operations per cycle, at a clock speed of 650 MHz. It has a peak floating-point performance of over 250 GFLOPS using single-precision floating-point numbers, counting multiply-adds as two FLOPs. GPUs have an explicitly parallel programming model and

their performance continues to increase as transistor counts increase.

The performance available on GPUs has led to interest in using GPUs for general-purpose programming [16, 8]. It is difficult, however, for most programmers to program GPUs for general-purpose uses.

In this paper, we show how to use data parallelism to program GPUs for general-purpose uses. We start with a conventional imperative language, C# (which is similar to Java). We provide a library that implements an abstract data type providing data-parallel arrays; no aspects of GPUs are exposed to programmers. The library evaluates the data-parallel operations using a GPU; all other operations are evaluated on the CPU. For efficiency, the library does not immediately perform data-parallel operations. Instead, it builds a graph of desired operations and compiles the operations on demand to GPU pixel shader code and API calls.

Data-parallel arrays only provide aggregate operations over entire input arrays. The operations are a subset of those found in languages like APL and include element-wise arithmetic and comparison operators, reduction operations (such as sum), and transformations on arrays. Data-parallel arrays are functional: each operation produces a new data-parallel array. Programmers must explicitly convert back and forth between conventional arrays and data-parallel arrays. The lazy compilation is typically done when a program converts a data-parallel array to a normal array.

Compiling data-parallel operations lazily to a GPU allows us to implement the operations efficiently: the system can avoid creating large numbers of temporary data-parallel arrays and optimize the creation of pixel shaders. It also allows us to avoid exposing GPU details to programmers: the system manages the use of GPU resources automatically and amortizes the cost of accessing graphics APIs. Compilation at run time also allows the system to handle properties and features that vary across GPU manufacturers and models.

We have implemented these ideas in a system called Accelerator. We evaluate the effectiveness of the approach using a set of benchmarks for compute-intensive tasks such as image processing and computer vision, run on several generations of GPUs from both ATI and NVidia. We implemented the benchmarks in hand-written pixel shader assembly for GPUs, C# using Accelerator, and C++ for the CPU. The C# programs, including compilation overhead, are typically within 2× of the speed of the hand-written pixel shader programs, and sometimes exceed their speeds. The C# programs, like the hand-written pixel shader programs, often outperform the C++ programs (by up to 18×).

Prior work on programming GPUs for general-purpose uses either targets the specialized GPU programming model directly or provides a stream programming abstraction of GPUs. It is difficult to target the GPU directly. First, programmers need to learn the graphics programming model, which is specialized to the set of

operations implemented in a typical graphics pipeline. Second, programmers need to learn about pixel shaders. Most of the general-purpose computing power of GPUs is accessible from one part of the graphics pipeline, pixel shading. Pixel shading refers to computing the desired value of an output pixel (*i.e.*, a location in memory). Pixel shaders have a highly restricted parallel programming model in which each element of an output array is computed completely independently of every other element in the array. Third, programmers need to learn specialized graphics APIs such as DirectX or OpenGL, and new programming languages for programming pixel shaders such as Cg [13], HLSL, and the OpenGL Shading Language. Finally, they must implement algorithms using this programming model. The task is complicated by differences in available resources, implemented features, and even versions of the specialized graphics APIs implemented across models and manufacturers of GPUs.

A higher-level approach is to abstract GPUs as stream co-processors that are programmed in tandem with CPUs. This is the approach used by Brook [3]. In Brook, a stream is an array of records. The GPU part of the program is organized as a collection of kernels, each of which takes one or more streams as input and writes one or more streams as output. Kernels are generalizations of pixel shader programs. A kernel is a function that is applied to each element in a set of input streams to produce elements for a set of output streams. Kernels are restricted in their operations to avoid side-effects between computations of different stream elements in an output stream. Kernels are connected to each other in a dataflow fashion.

Programmers use an extension of C that contains new stream types and a new keyword that designates certain functions as kernels. Brook uses a special C compiler to compile the kernels to Cg and also generates C++ code to connect the kernels. Programmers have to divide programs into kernels and handle GPU resource limitations as well. It is possible for the system to generate Cg that will not run on a given GPU.

Accelerator differs from Brook in two ways. First, it provides an even higher-level programming model. Programmers do not have to divide computations into kernels and no aspects of GPUs are exposed to programmers. Accelerator automatically divides the computation into pixel shaders. Accelerator also provides transformations on entire arrays. Second, Accelerator uses just-in-time compilation instead of ahead-of-time compilation. This has two advantages: the system can handle limitations and features that vary by GPU model and it requires no changes to existing tool chains, reducing the barriers to adoption.

The remainder of the paper is organized as follows. Section 2 describes additional related work. Section 3 provides background information on GPUs, the graphics pipeline, and how to program GPUs. Section 4 describes the high-level data-parallel arrays and operations on those arrays. Section 5 describes how to translate those operations to GPU operations. Section 6 provides performance results. Finally, Section 7 concludes.

2. Additional related work

The Sh toolkit [14] is also closely related to Accelerator. Sh allows developers to write GPU applications within one environment without the need to resort to GPU-specific tools like Cg or HLSL. Like Brook, Sh provides a stream programming model for general purpose computation on the GPU. Instead of compiling to Cg, however, Sh provides its own code generators that can target either GPUs or CPUs. Sh provides an interface that makes it possible to extend the system to additional targets. Sh provides an *immediate* mode in which graphics operations are carried out immediately and a *retained* mode in which operations are saved up for later execution. The retained mode in Sh is most similar to Accelerator.

Retained mode makes it possible to build and manipulate stream expressions at a high level.

The compilation of aggregate array expressions has a long history starting with APL [7]. Later on, it was recognized that these ideas could be applied to vector machines [4]. The advent of data-parallel high-performance computers such as the Connection Machine brought forth new language proposals such as the Paralation model [18]. Over the years many specialized high-level languages have been developed for numerically intensive computation including NESL [1], ZPL [9] and Single Assignment C (SaC) [20]. Accelerator shares the use of high-level aggregate data structures with all of these approaches.

3. Background

GPUs are special-purpose processors designed to render three-dimensional scenes quickly. A scene is divided into triangles and a collection of images that provide the "texture" for each triangle. The triangles are specified by vertices. Each triangle can be transformed independently. The color of each output pixel for a triangle can also be computed independently. This independence allows the use of the GPU as a data-parallel array processor. GPUs also have special-purpose hardware support for graphics operations such as clipping and z-buffering. Even though GPUs are special-purpose processors, they are in almost every desktop system today. Hundreds of millions are sold yearly [15].

3.1 Processor architecture

A simplified block diagram of a GPU is shown in Figure 1. The diagram shows the programmable parts of a GPU and omits some non-programmable, graphics-specific parts. A complete description of the operations available on GPUs can be found in [2]. An array of vertex processors receives the vertices along with optional properties such as their color and normal vectors. The processors compute, via programs called *vertex shaders*, the output positions of the vertices. The transformed vertices are used by the rasterizer to produce the pixel address for each pixel in the triangles. These pixel addresses are used by the pixel processors in programs called *pixel shaders* to compute the final color of each pixel.

As an example, the Nvidia GeForce 6800 has 6 vertex processors and 16 pixel processors. Each vertex or pixel processor has a 4-way vector processing unit which is capable of 16-bit or 32-bit floating-point operations. Each processor has constants and temporary registers. However, there is no temporary memory. That is, a shader cannot write to memory other than by producing a return value.

3.2 Steps in graphics rendering

The steps to rendering an image on the GPU are:

1. Fill a buffer with vertices for the triangles to be rendered.

2. Compile and set the vertex program that is to transform the vertices.

3. Compile and set the pixel shader program that is to compute the value of output pixels.

4. Render:

   (a) Transfer the vertex buffer and shaders to video memory;

   (b) Run the vertex shader given the vertex buffer as input;

   (c) Run the rasterizer on the output of the vertex shader;

   (d) Run the pixel shader on the output of the rasterizer and any specified input textures;

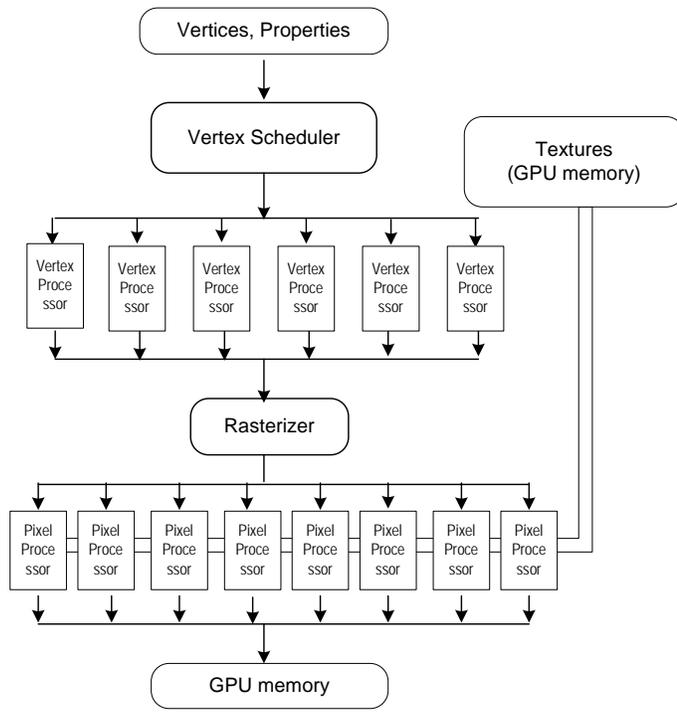   (e) Pass the output value to display memory or texture memory, as specified by the user.

Figure 1. Simplified block diagram for a GPU

A simple use of this graphics architecture is drawing a single triangle with a given image painted on its surface. This is done by storing the three vertices that represent the triangle in a vertex buffer. Each vertex has an x, y, and z coordinate. A simple vertex shader program is written that passes the vertices unaltered to the output register. To apply the texture to the surface of the triangle, a pixel shader program is written that does one texture lookup (from the given image) and returns the fetched value directly to the output register. The inputs to the pixel shader are the coordinates of the output pixel. The system stores the result from the output register at the memory location for the output pixel.

3.3 Programming model

The actual architectures of GPUs are hidden behind abstract virtual machines provided via graphics APIs like DirectX or OpenGL. To program the GPU, one targets those APIs.

DirectX and OpenGL provide processor-independent programming languages for shaders. Both provide an assembly language and higher-level C-like programming language. These languages are just-in-time compiled. Neither virtualizes the resources of a GPU, but requires the user to adhere to the strict register and memory limits of an underlying processor.

Programs are limited in the number of registers they can use and the number of instructions they can contain. Even the length of a chain of memory address calculations can be limited. There was also no looping or branching allowed in the programmable shaders until recently. Shader programs are strict SIMD programs. Many shaders allow only a single output value to be computed per pixel.

For example, in DirectX Pixel Shader 2 there are 12 temporary registers, 8 texture coordinate registers, 32 floating-point constant registers, and 16 sampler registers. A *texture coordinate register* contains a pixel address. These read-only registers are set by the rasterizer from properties of the input vertices. A *sampler register* is associated with an input texture and is used in memory lookup

instructions to indicate which texture to read. In addition to register limits, there are instruction limits. A pixel shader program can have at most 32 texture lookup and 64 arithmetic instructions. The texture lookup instructions are limited further by restrictions on how the address of the lookup is computed using other values read from textures.

As an example, the pixel shader that follows is one that would be run in the example above. It copies the input texture to the output:

```
ps_2_0
dcl_2d s0
dcl t0.xy
texld r0, t0, s0
mov oC0, r0
```

The first line declares the version of DirectX Pixel Shader Assembly language in use. The second line declares that there is an input texture in the sampler register, `s0`, and that it has two dimensions. The third line declares a two-dimensional texture coordinate register, `t0`. The fourth line is a texture load instruction that loads the temporary register, `r0`, with the value of the input texture that is bound to the sampler register `s0` at the location in `t0`. The final line moves the value from `r0` to the write-only output register `oC0`. For each pixel, the rasterizer sets the value of the read-only texture coordinate register `t0` by interpolating the output values of the vertex shader. For this pixel shader, we set up the interpolation process so that `t0` contains the coordinates of the output value. So the above pixel shader means that the value to place in the output location in `t0` is the value in the input texture at that same location.

The number of registers and the number of instructions in pixel shaders has continued to increase over time. For example, DirectX Pixel Shader 3 allows 32 temporary registers, 10 texture coordinate registers, 224 floating-point constant registers, and 16 sampler registers. It also allows 512 or more instructions in a pixel shader (the limit depends on the target GPU). Note that the some of these limits virtualize the underlying hardware and performance may degrade if too many instructions or registers are used in practice.

Accelerator supports DirectX Pixel Shader 2 and DirectX Pixel Shader 3 and uses the most recent version of DirectX pixel shading that is supported by a target GPU card.

3.4 Advantages and disadvantages of targeting GPUs

Using GPUs as targets has some key advantages. They are a readily available parallel architecture, with one in almost every desktop system. Because of their explicitly parallel programming model, existing GPU programs are benefiting from the continued increase in transistors due to Moore's law. The performance of GPUs is still increasing substantially and new GPUs come out every 9 to 18 months that deliver significantly higher performance than previous versions. GPUs also have much higher peak floating-point performance than comparable CPUs.

In contrast, most CPU programs use a sequential programming model and are not benefiting from the continued increase in transistors due to Moore's law. They will need to be rewritten or modified substantially to obtain increased performance from new CPUs with multiple cores on a chip. Furthermore, CPU clock speeds have plateaued due to power concerns.

On the other hand, GPU architectures have a number of disadvantages for parallel programming. First, GPUs have a SIMD programming model. Programs that fit the SIMD model map well and other programs map less well. GPUs are moving toward a SPMD model, although use of loops and control-flow instructions in GPU pixel shaders may currently degrade performance, not improve it. Second, the actual architectures of GPUs are hidden behind device drivers that support APIs that implement virtual machines. This abstraction and lack of detail can hamper obtaining the most performance out of a graphics processor. For example, the virtual machines have no model of caches and no easy way for programmers

to indicate how to traverse memory to facilitate memory reuse. This can make it difficult to write parallel programs where memory locality is crucial to performance. In addition, we must target APIs designed to support graphics that introduce extra complexity and overhead. Third, the programmable parts of GPUs have had limited support for primitive types typically found on CPUs. Most notably, they do not support 64-bit floating-point numbers. Current programmable parts of GPUs support 32-bit floating-point numbers and floating-point numbers with lower precision. Integers must be encoded using floating-point numbers. However, the primitive type support is improving: newer GPUs will have built-in support for 8, 16, and 32-bit integers and support for 32-bit floating-point numbers is evolving to be IEEE compliant [2].

4. Data-parallel programming model

The fundamental data type in Accelerator is a parallel array. The parallel array is similar to a conventional array—it has rank and dimensions—however, it does not provide an indexing operation that allows access to an individual element. That is, if A is a parallel array, it is not possible to access $A_{2,3}$ directly from C# without first converting A to a conventional array.

The elements within a parallel array must all have the same primitive type. Each primitive type gives rise to a subtype of parallel array. Currently, Accelerator allows element types to be one of the following: float, int, bool, float4 (a vector of 4 floating-point values).

Operations on parallel arrays are functional in style: the result of any operation is a new array and the parameters are not modified. Operations do not allow any side-effects.

In the C# library, parallel arrays are implemented as an abstract class ParallelArray with a subclass corresponding to each element type. These subclasses are FloatParallelArray, IntParallelArray, BoolParallelArray and Float4ParallelArray.

There are six general classes of operations on data-parallel arrays:

- Construction: a parallel array is constructed from an object of type System.Array. The dimensions of the parallel array and values contained within are identical to those of the parameter System.Array.

- Conversion back to a normal array

- Element-wise operations

- Reductions

- Transformations on arrays

- Linear algebra

The following sections explain the various operations, except for construction and conversion back to a normal array. The operations are summarized in Tables 1 to 4. $R_{i,j}$ denotes the value of the result array at location (i, j), given input arrays A, B, and C as needed. For simplicity, only formulas for 2-dimensional arrays are given.

4.1 Element-wise operations

Many operations are element-wise operations. The value of the nth location in the resulting array is computed by taking the value of the nth location in each parameter array and applying a function to the values. Table 1 shows some of the element-wise operations provided by Accelerator.

4.2 Reductions

A useful array operation is a reduction by an operation across a particular dimension. For example, the reduction of addition across the first dimension of a two dimensional array is a row sum, *i.e.*,

Operation	Definition
Add	$R_{i,j} = A_{i,j} + B_{i,j}$
Subtract	$R_{i,j} = A_{i,j} - B_{i,j}$
Multiply	$R_{i,j} = A_{i,j} \times B_{i,j}$
Divide	$R_{i,j} = A_{i,j} \div B_{i,j}$
Max	$R_{i,j} = \max\left(A_{i,j}, B_{i,j}\right)$
Min	$R_{i,j} = \min\left(A_{i,j}, B_{i,j}\right)$
Select	$R_{i,j} = \begin{cases} B_{i,j} & \text{if } A_{i,j} > 0 \\ C_{i,j} & \text{otherwise} \end{cases}$
Cos	$R_{i,j} = \cos A_{i,j}$
Sqrt	$R_{i,j} = \sqrt{A_{i,j}}$
And	$R_{i,j} = A_{i,j} \wedge B_{i,j}$
Or	$R_{i,j} = A_{i,j} \vee B_{i,j}$
CompareEqual	$R_{i,j} = \begin{cases} \text{true} & \text{if } A_{i,j} = B_{i,j} \\ \text{false} & \text{otherwise} \end{cases}$
CompareGreater	$R_{i,j} = A_{i,j} > B_{i,j}$
CompareGreaterEqual	$R_{i,j} = A_{i,j} \geq B_{i,j}$
CompareLess	$R_{i,j} = A_{i,j} < B_{i,j}$
CompareLessEqual	$R_{i,j} = A_{i,j} \leq B_{i,j}$
Cond	$R_{i,j} = \begin{cases} B_{i,j} & \text{if } A_{i,j} = \text{true} \\ C_{i,j} & \text{otherwise} \end{cases}$

Table 1. Some element-wise operations

Operation	Definition
Sum(1)	$R_i = \sum_j A_{i,j}$
Product(1)	$R_i = \prod_j A_{i,j}$
MaxVal(1)	$R_i = \max_j A_{i,j}$
MinVal(1)	$R_i = \min_j A_{i,j}$
All(1)	$R_i = \bigwedge_j A_{i,j}$
Any(1)	$R_i = \bigvee_j A_{i,j}$

Table 2. Reduction operations

Operation	Definition
Section $(b_i, c_i, s_i, b_j, c_j, s_j)$	$R_{i,j} = A_{b_i + s_i \times i, \ b_j + s_j \times j}$
Shift(s_i, s_j)	$R_{i,j} = A_{i - s_i, \ j - s_j}$
Rotate(s_i, s_j)	$R_{i,j} = A_{(i - s_i) \bmod \text{size}_i, \ (j - s_j) \bmod \text{size}_j}$
Replicate$(\text{size}_i, \text{size}_j)$	$R_{i,j} = A_{i \bmod \text{size}_i, \ j \bmod \text{size}_j}$
Expand(b_i, a_i, b_j, a_j)	$R_{i,j} = A_{(i - b_i) \bmod \text{size}_i, \ (j - b_j) \bmod \text{size}_j}$
Pad(b_i, a_i, b_j, a_j, c)	$R_{i,j} = \begin{cases} A_{i - b_i, \ j - b_j} & \text{if in bounds} \\ c & \text{otherwise} \end{cases}$
Transpose(1, 0)	$R_{i,j} = A_{j,i}$

Table 3. Transformation operations

Operation	Definition
DropDimension(1)	$R_i = A_{i,0}$
AddDimension(1)	$R_{i,j,k} = A_{i,k}$

Table 4. Rank changing operations

it computes the one-dimensional array that contains the sum of all of the elements in each row of the original array. Table 2 shows examples of reducing in the 1st dimension (*i.e.*, along j).

4.3 Transformations

4.3.1 Affine transformations

Accelerator provides a class of operations that select or duplicate elements of the given parallel array according to a simple pattern. This pattern can be specified by an affine transformation of the coordinates of the input array and by a definition of the value when the transformed coordinate is out of range. This out-of-range value can be determined in one of three ways:

- A default return value can be specified.
- The value of the input array closest to the out-of-range coordinates can be returned.
- The transformed coordinates can be taken modulo the size of each dimension and the value from the input using the transformed coordinates can be returned.

In Table 3, $size_i$ denotes the extent of the array along the i axis. The Expand and Pad operations enlarge the shape of the array such that, for example, the extent of R along the i axis is $a_i + size_i + b_i$.

4.3.2 Rank-changing operations

There are two helpful rank-changing operations shown in Table 4. The first drops a dimension of an array and the second adds a dimension to an array.

4.4 Linear algebra

Accelerator provides the standard inner and outer products over two arrays.

4.5 Other Operations

4.5.1 Arbitrary Selection

To select a subset of elements of a parallel array that can not be expressed as an affine transformation, Accelerator provides a Gather operation. Given integer-valued data-parallel arrays I and J, the result R of Gather(A, I, J) is given by

$$R_{i,j} = A_{I_{i,j}, J_{i,j}}$$

The result has the same shape as I and J.

4.5.2 Parallel array of indices

Given the dimensions of the output parallel array, a parallel array is constructed where each element is the index along a given dimension.

4.6 Type conversions

Element-wise conversions are provided between parallel arrays with floating-point elements and parallel arrays with integer elements. A FloatParallelArray can be converted to a Float4ParallelArray by replicating the single float value across all 4 components of the corresponding float4 element There are four ways to convert a float4 parallel array to a float parallel array. We can select, for each element, the first, second, third, or fourth component of the corresponding float4 element.

4.7 Syntax and examples

The C# version of Accelerator provides a set of classes that implement parallel arrays for each of the supported element types. The classes provide static functions and operator overloads where appropriate.

```
using Microsoft.Research.DataParallelArrays;

static float[,] Blur(float[,] array, float[] kernel)
{
  float[,] result;
  DFPA parallelArray = new DFPA(array);

  FPA resultX = new FPA(0f, parallelArray.Shape);
  for (int i = 0; i < kernel.Length; i++) {
    int[] shiftDir = new int[] { 0, i};
    resultX += PA.Shift(parallelArray, shiftDir) * kernel[i];
  }

  FPA resultY = new FPA(0f, parallelArray.Shape);
  for (int i = 0; i < kernel.Length; i++) {
    int[] shiftDir = new int[] { i, 0 };
    resultY += PA.Shift(resultX, shiftDir) * kernel[i];
  }

  PA.ToArray(resultY, out result);
  parallelArray.Dispose();
  return result;
}
```

Figure 2. 2-dimensional convolution implemented in Accelerator

Figure 2 shows an example of 2-D convolution implemented in Accelerator. Given an image represented as 2-dimensional array, 2-D convolution computes the value of a pixel in a new image by multiplying the pixel and its neighbors by weights and summing the weighted values. The `Blur` function takes a C# array and 1-dimensional array of weights as arguments. It converts the C# array to a parallel array `parallelArray`. It then computes the weighted values in the x direction by repeatedly shifting the entire original image by `i` pixels and multiplying the shifted image by the appropriate weight. Operator overloading of `*` is used here. This process is then repeated in the y direction. This sample code is equivalent to the code used in the Convolve benchmark.

5. Translation to the GPU

This section describes approaches for translating Accelerator operations into a set of GPU operations. We start by describing a simple approach for translating Accelerator operations. We then refine it in a step-by-step fashion to create a practical translation.

5.1 A simple approach

As described earlier, the GPU pipeline has two programmable components—vertex shaders and pixel shaders. Vertex shaders receive a series of vertices as input and produce a series of vertices as output. Pixel shaders, on the other hand, have easy access to texture memory. A pixel shader can read texture memory as inputs and write to it as outputs, although a pixel shader cannot both read and write to the same texture at the same time.

In Accelerator, we chose to target solely pixel shaders because of their more general memory access properties. We implement all Accelerator arrays as textures that reside in video memory on the GPU. These textures serve as the inputs and outputs to pixel shaders. We implement the Accelerator operators using pixel shaders. The general compilation challenge is to combine multiple operations into one pixel shader, for efficiency reasons that are discussed in Section 5.2. Because not all operations can be combined, we generally convert Accelerator operations into multiple pixel shaders.

Pixel shaders, as discussed earlier, have a highly restricted programming model. Generally, a pixel shader computes the value of each output pixel as a function of the output location. The function is described as a sequential program that may do arithmetic operations and read texture memory. A pixel shader has limited control-flow capabilities, for example, only a certain number of branches,
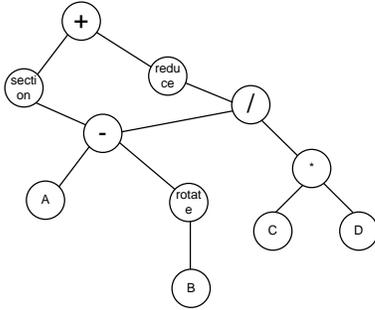
Figure 3. Initial expression DAG

and looping that can be statically unrolled by the compiler. In any case, using control flow may have a severe performance penalty, so it is usually best to avoid it. The penalty arises from the SIMD execution model: all pixel shaders can be considered to be evaluated in lockstep.

More formally, a pixel shader can be modeled the following way:

$$\forall j.A[j] = f(B_1[\phi_1(j)], B_2[\phi_2(j)]\ldots)$$

where A is the output texture, B_1 to B_n are input textures, j is a tuple representing a location in A (a singleton for 1-dimensional arrays and a pair for 2-dimensional arrays), and f is a side-effect-free function. The ϕ_1, ϕ_2 represent transformations that may be applied to the output location to compute an input location in a texture. The transformations are *affine* transformations combined with simple rules for handling input locations that are out of bounds (for example, by subsequently applying a modulus operator to produce a valid input location or by always using a default value). The input location transformations actually have hardware support and may be set up before the pixel shader is evaluated.

Calls to most data-parallel array functions build an expression DAG that represents the requested operation. This allows us to evaluate operations lazily, and to perform operations on the DAG itself. Specifically, each Accelerator operation allocates an expression node that holds the operator and parameter values. The parameter values may be child data-parallel arrays or constants.

Calls to operations that convert a data-parallel array to a normal array trigger evaluation of the expression DAG. The evaluation process will be described shortly.

Calls to operations that convert a normal array to a data-parallel array produce textures. These textures are wrapped in an object so that they can be leaves in an expression DAG. The system eagerly copies the data from the C# array to a texture. This provides a clean semantics for the system: subsequent changes to the C# array do not affect the data-parallel array. The texture memory is reclaimed when the resulting expression object is finalized by the garbage collector. Because finalization is unpredictable and texture memory is quite limited, we recommend that users use an explicit `Dispose` operation or the C# `using` pattern.

Figure 3 shows an example expression DAG. `A`, `B`, `C`, `D` are input arrays (represented as textures). The `-`, `/`, `*` and `+` are element-wise operations on data-parallel arrays. The `rotate`, `reduce`, and `section` operators also operate on entire data-parallel arrays.

5.1.1 Evaluating the expression DAG: take 1

A simple approach to evaluating an expression DAG is to walk the DAG in a post-order fashion, converting every node to a pixel shader as we go. That is, every expression DAG node is converted to a shader that reads the inputs to that node from memory and writes its output to a newly allocated texture in memory. This is highly inefficient but useful for understanding how the basic operations work.

We divide expression operators into three categories and generate (or call) a different kind of pixel shader for each category.

- Element-wise operators (for example, simple scalar operators such as +, *, -, /).

- Operators that perform a linear transformation on memory such as Shift, Rotate, Section, and Expand. These will be referred to as texture coordinate operators.

- Operators that require several render passes or that access their parameters' memory with coordinates that are not an affine transformation of the output coordinates. These include inner product, outer product, gather, and reduction operators. We refer to these informally as "random access" operators.

Element-wise operators are the easiest to handle: they read several input textures (all of the same size) and write an output texture of the same size. For them, the affine ϕ functions are all identity operators, the inputs are simply the textures for the child data-parallel arrays, and the output is simply the result of the operation. For example, $+(B_1, B_2)$ becomes a pixel shader that merely implements

$$\forall j.A[j] = +(B_1[j], B_2[j])$$

This is a straightforward to translate to a pixel shader.

```
ps_2_0
dcl_2d s0        // B_1
dcl_2d s1        // B_2
dcl t0.xy        // j
texld r0, t0, s0 // r0 <- B_1[j]
texld r1, t0, s1 // r1 <- B_2[j]
add r1, r0, r1   // r1 <- B_1[j] + B_2[j]
mov oC0, r1      // output r1
```

Linear transformations are also straightforward to handle. One merely sets the appropriate transformation from the output location to the input location (*i.e.*, a ϕ function) before executing a pixel shader that moves its input to its output.

$$\forall j.A[j] = B_1[\phi(j)]$$

This pixel shader for this is:

```
ps_2_0
dcl_2d s0        // B_1
dcl t0.xy        // φ(j)
texld r0, t0, s0 // r0 <-  B_1[φ(j)]
mov oC0, r0      // output r0
```

Note that one usually must invert the transformation for the original Accelerator operation. For example, a `replicate` may become an identity affine transformation combined with a modulus operation on the output location (to handle the typical case where the input array is smaller than the output array). The hardware implements the modulus operation.

Finally, random-access operators are also easy to handle. They simply call a set of pre-built shaders, using textures corresponding to the operator parameters as their inputs. Accelerator has pre-built shaders for inner product (matrix multiply), outer product, and reduce. Note that matrix vector multiplication does not need a pre-built shader because it can be expressed reasonably efficiently using other Accelerator operations.

5.2 Combining multiple element-wise operations into one pixel shader

The approach of using a pixel shader per expression node is highly inefficient. It incurs the overhead of at least one render pass for every operator in the expression, creates at least one intermediate texture value for the result of every operator, and incurs a great deal of memory traffic. In a system with limited memory, such an evaluation scheme may also quickly exhaust GPU memory. There is a significant per-pass overhead associated with executing a shader. For example, a pixel shader must be compiled on the CPU and transferred to GPU memory. In addition, there is time for setting up the graphics pipeline (on the order of several hundred thousand cycles). The memory traffic is problematic because GPUs are optimized to write output values to frame buffers—not to textures. This makes it expensive to use the output of one pixel shader as the input to another.

For example, consider evaluating the expression DAG in Figure 3 this way. This DAG would yield at least seven shaders and seven arrays in GPU memory. A simple floating-point operation such as multiply would require loading two values from GPU memory, writing the value to GPU memory, and a full render pass with control starting and ending on the CPU. Any speed gained by the parallelism of the GPU and fast floating-point operations would be overwhelmed by the pass and memory overhead.

Thus, it is desirable to try to combine multiple data-parallel operations into one pixel shader. This yields a new set of problems, though. Recall from Section 3.3 that pixel shaders have a number of limitations, including the number and types of registers, the length of chains of memory address calculations, and instruction limits.

5.2.1 Representing combined operations

We first describe how to represent the combination of operations. We do this by making it explicit where a new shader program should be created when converting an expression DAG to a set of shaders. (In Section 5.1.1 every node was converted to a shader.)

We add a new kind of expression node called a *break node*. A break node has exactly one child, the root of the subexpression that requires evaluation as a separate shader. This implies that the result of the subexpression will be placed in memory. The parents of the break node are all operation nodes that use the subexpression as a parameter.

The approach in Section 5.1.1 can be modeled by inserting a break node before every interior node in the expression graph.

5.2.2 Combining element-wise operations

It is straightforward to combine a tree (or DAG) of element-wise operators into one pixel shader (ignoring resource constraints). This can be defined in an inductive fashion.

Given $A = \mathrm{op}_1(B, C)$ and $D = \mathrm{op}_2(A, E)$, where op_1 and op_2 are element-wise operators, it is straightforward to combine them into one pixel shader and avoid computing an intermediate array A.

$$\forall j. A[j] = \mathrm{op}_1(B[j], C[j])$$

$$\forall j. D[j] = \mathrm{op}_2(A[j], E[j])$$

can be turned into:

$$\forall j. D[j] = f(B[j], C[j], E[j])$$

where

$$f(t_1, t_2, t_3) = \mathrm{op}_2(\mathrm{op}_1(t_1, t_2), t_3)$$

We can implement this partitioning by doing a pre-order traversal of the expression DAG. Whenever a non-element-wise operation is encountered, we insert breaks between the operation and all of its non-leaf children, forcing the children to be evaluated to memory. Figure 4 shows the initial expression DAG with breaks inserted according to this algorithm.
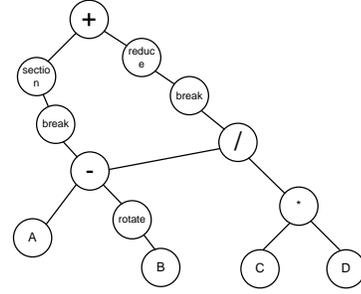

Figure 4. Expression DAG with shader breaks marked

5.2.3 Generating a DAG of valid shaders

Resource limits are the primary obstacle to combining all of the element-wise operations in a subgraph whose boundaries are marked by shader breaks into one shader. We present a recursive algorithm for traversing an expression DAG to produce a DAG of valid shaders, taking into account resource constraints.

We first describe register allocation in a pixel shader, because register limits are one of the constraints on adding operations to pixel shader. There are four kinds of registers available: temporary registers, texture coordinate registers, constant registers and sampler registers.

Allocation of temporary registers is straightforward. Because we can not spill values to memory and there is no control flow in our pixel shaders, the temporary registers can be allocated with a simple algorithm that assigns a register on first definition and frees a register on last use. Certain pixel shader restrictions such as limits to dependent memory operations sometimes prevent a register from being added to the free list.

Texture coordinate registers, constant registers and sampler registers are defined outside of the pixel shader and are read-only. They are simply assigned in order of use and cannot be re-allocated while building a pixel shader.

The recursive algorithm for traversing an expression DAG produces a shader DAG and a register. A shader DAG is represented as a shader together with child shader DAGs that provide inputs (*i.e.*, textures) to the shader. The shader is incomplete. Specifically, to complete the shader, we would need to move the value in the distinguished register to the output register for the shader. When the algorithm hits a shader break, it completes the current shader. In order to return a shader and register, it creates a new shader that has the output of the newly completed shader as an input. More specifically, it adds a sampler register for the output of the completed shader. The new shader loads the value of this sampler register at the default location into a register. It returns this new shader and register as its result. We will refer to this method of marking a shader as complete and starting a new shader that loads its output as *finishing* a shader.

The algorithm does a post-order traversal, evaluating the children and then attempting to combine the children with the shader for the parent node. In the absence of resource limits, the parent node would concatenate the code for the child shaders to its code. It would also merge the sets of registers in the child shaders. Finally, it would append instructions that compute its value from the registers for the children.

In the presence of resource limits, we attempt to do the above in a greedy fashion. If we reach a node where all parameters and the parent can not be put into the same shader, we restrict the combination as follows. If a child shader can not be combined with the parent, the child shader is finished. Finishing creates a new

```
if (node is not a leaf or break node)
{
  shaderOut = new Shader()
  N = number of parameters to node
  // create the child shaders
  for i = 1 to N
  {
    ShaderRegister(i) = CreateShader(child(i))
  }
  // combine as many children as will fit with the parent
  while (Shader(i) can be combined with Shader(node))
  {
    Append Shader(i) to shaderOut
    i = i + 1
  }
  // finish off any remaining child shaders
  if (i <= N)
  {
    while (i <= N)
    {
      Finish(ShaderRegister(i).Shader)
      ShaderRegister(i).Shader.OutputRegister = ShaderRegister(i).Register
      Append(Shader(i).Shader) as a Sampler of shaderOut
    }
  }
  // add the parent code
  AppendInstructionsFor(shaderOut, node, {ShaderRegister(i)})
}
if (node is a break node)
{
  Finish the parameter shader
  Create a new shader that loads the output of the break node
  into a register
}
if (node is a leaf)
{
  Create a new shader that loads the output of the leaf node
  into a register
}
```

Figure 5. Pseudo-code for building a shader DAG

```
ps_2_0            ps_2_0            ps_2_0            ps_2_0
dcl_2d s0         dcl_2d s0         dcl_2d s0         dcl_2d s0
dcl_2d s1         dcl_2d s1         dcl t0.xy         dcl_2d s1
dcl t0.xy         dcl_2d s2         dcl t1.xy         dcl t0.xy
dcl t1.xy         dcl t0.xy         dcl t2.xy         dcl t1.xy
texld r0, t0, s0  texld r0, t0, s0  dcl t3.xy         texld r0, t0, s0
texld r1, t1, s1  texld r1, t0, s1  texld r0, t0, s0  texld r1, t1, s1
sub r1, r0, r1    texld r2, t0, s2  texld r1, t1, s0  add r1, r0, r1
mov oC0, r1       mul r2, r1, r2    add r0, r0, r1    mov oC0, r1
                  rcp r2, r2.x      texld r1, t2, s0
                  mul r2, r0, r2    add r0, r0, r1
                  mov oC0, r2       texld r1, t3, s0
                                    add r0, r0, r1
                                    mov oC0, r0
```

Figure 6. Actual pixel shader code generated for the example graph in Figure 3.

shader that simply loads the output of the completed child shader into a register. This new, small shader is combined with the parent.

We attempt to combine child shaders with the parent instead of with each other because many GPUs support only one output. If more than one child were combined into a single shader that was separate from the parent, the combined shader would need an output for each child—the parent requires one input for every child in that shader.

To determine if a child shader can be combined with a parent we estimate the number of additional registers and lines of code that the parent would require and check if the combined shader is within limits. If a register limit is exceeded, we actually run our register allocator to get an exact register count and combine the shaders if this new count is within the limits.

Pseudo-code for this technique is shown in Figure 5. The set of shaders generated from Figure 4 is shown in Figure 6. Note that column 3 contains the shader for the reduction operation, which is actually executed multiple times to perform the log-height reduction.

5.2.4 Evaluating the DAG of shaders

This step evaluates each shader on the GPU, invoking the target graphics API to compile and run each pixel shader. It traverses the shader DAG in a bottom-up fashion and evaluates each node (*i.e.*, shader) in the DAG. Evaluating a node requires a full render pass that uses the entire graphics pipeline. The first step is triangle setup, that is, sending a list of vertices for the output rectangle. Accelerator sets up a vertex buffer with a single rectangle (two triangles) and sets a vertex shader that simply copies its input to the output register. The pixel processors simply receive as input the locations of the pixels (or array elements) to be computed. The next step is to set the textures that hold the results from child nodes as input textures for the shader. Finally, the render pass is executed which runs the pixel shader, and the resulting texture is returned.

5.3 Combining transformation operations with other operations into one pixel shader

Transformation operators such as Rotate, Section, and Expand are different from other operators: they perform no actual computation on the values loaded from an input texture. They merely transform the output location to an input location and load a value from the input texture.

Some of these operators are distributive with respect to element-wise operations. Also, some of these operators can be composed with other transformation operators. As an example of distributivity, consider a Rotate of the sum of two arrays. This is equivalent to a sum of the Rotate of the two arrays.

In general, transformation operations consist of an affine transformation from output locations to input locations *and* a specification of how out-of-bounds input locations are to be treated. For example, if one shifts a 5×5 array to the left by 2 elements, there are several choices as to how to fill in the 2 columns on the right of the resulting array. The possibilities included in Accelerator are:

- Wrap–wrap the values from the left hand side of the array
- Default–fill in a specified default value such as 0
- Clamp–duplicate the value in the rightmost column

The treatment of out-of-bounds input locations can interfere with distributivity of transformations with respect to simple operations. Wrapping and clamping distribute with element-wise operations. A transformation using a default value distributes only when the default value is the identity element for the per-element operation. For example, if the operation is addition, then a shift with a default of 0 distributes with respect to the additions, but a default of any other value does not.

We take advantage of the distributivity of some transformation operations with respect to element-wise operations and the composability of some transformation operations with other operations to reduce the number of shader breaks inserted in the expression DAG.

We subdivide the pass that inserts shader breaks into two passes. The first pass inserts shader breaks only for random-access operations. The second pass is a top-down pass over the expression DAG that handles transformation operations.

First, it attempts to push transformation operations (with respect to element-wise operations) to leaf nodes and break nodes. These nodes are data nodes whose results are read from memory. For transformation operations that do not distribute with respect to element-wise operations, it simply inserts a break node between the transformation operation and the element-wise operation. The transformation information is applied when reading the values for the break node from memory and the transformation operation is removed from the expression DAG.

Second, when it encounters two adjacent transformation nodes (*i.e.*, transformation nodes with a parent-child relationship), it attempts to combine those transformations. The only restrictions to this composition are when the handling of the boundary values does not compose. If the transformation nodes do not compose, it again inserts a break node between the two transformation nodes.

5.3.1 Refinements of the texture coordinate operation pass

The advantage to moving a transformation operation as far down the DAG as possible is that it produces fewer shaders, which leads to fewer render passes and, in general, faster performance. There is one case when it is best to leave the transformation operation in place, *i.e.*, add a break node after the operation. This is when the transformation increases the size of the array and the expression below the texture coordinate operation is expensive to compute.

For example, consider a replicate that tiles a 100×100 array 10 times in each direction to produces an array that is 1000×1000. If the expression below the operation is the sum of two 100×100 arrays, by leaving the operator in place we perform 20,000 memory accesses and 10,000 additions. Then we tile—which costs the overhead of one render pass and the cost of writing a new array of 1,000,000 elements. If we push the tiling to the data nodes, we will perform 2,000,000 memory accesses and 1,000,000 additions. If the cost of a render pass is less than the cost of the extra memory accesses and additions, we are better off leaving the texture coordinate operation in place.

5.4 Common subexpressions

Computing a common subexpression once and reusing the value can be faster than recomputing the expression at each use. We handle CSEs by introducing a pass over the expression DAG that identifies CSEs for which reuse is cheaper than recomputation. The pass inserts break nodes between those subexpressions and their parents. It is trivial to identify some CSEs: they are expression DAG nodes with multiple parents.

A heuristic for when to break at a CSE was developed experimentally. It takes into account the size of the expression, the number of reuses, and the size of the inputs.

5.5 Managing graphics resources

GPU memory can be a limited resource when operating over large data sets. A typical high-end GPU has 256 MBytes or 512 MBytes of GPU memory and a 1000×1000 color image can use 16 MBytes of memory when represented in red-green-blue-alpha format using 32-bit floating-point numbers. If memory is not managed carefully during evaluation, it is possible to run out of memory quickly. In addition, requesting large blocks of memory can be costly. Because of this, Accelerator manages a small pool of textures explicitly, instead of always requesting textures from the graphics API. Intermediate textures created during the evaluation are reference counted and returned to the pool as soon as possible.

Accelerator also caches compiled pixel shaders to avoid unnecessary compilation by the target graphics API. The graphics APIs take pixel shader programs as strings. Accelerator keeps a hash table from these strings to compiled pixel shader programs. It uses the hash table to see if a program has already been compiled by the graphics API.

6. Performance evaluation

We evaluated the performance of Accelerator by measuring the execution times of various programs, each written in three versions: C# using Accelerator, hand-written Pixel Shader 3.0 assembly code, and C++ running on a CPU. Figure 5 describes the benchmarks. In the benchmarks, including the Accelerator and pixel shader as-

sembly code versions, all floating-point numbers used are single-precision (32-bit) floating-point numbers.

The C# programs were compiled with Visual Studio 2005. For the C++ versions of Sum, Matrix-vector multiplication, and Matrix-matrix multiplication, we used BLAS routines from Intel's Math Kernel Library 7.0. We hand-optimized the other C++ programs with respect to memory access patterns, but did not tune them to take full advantage of SSE or hyper-threading. We compiled the C++ programs with either Visual Studio 2005 (using `/Ox /fp:fast`) or the Intel C++ Compiler 9.0 for Windows (using `/Ox`), whichever was faster in each case. For each benchmark, the three versions produce the same result. For the resulting arrays A and A',

$$(\forall i) \ \frac{\left|A[i] - A'[i]\right|}{\max_j \left|A[j]\right|} < 10^{-6}.$$

As a reference machine, we used a Dell Optiplex GX280 with a 3.2GHz Pentium 4 CPU, 16KB of L1 cache, 1MB of L2 cache, 1GB of 400ns memory, and a PCI Express bus. The machine ran Windows XP with Service Pack 2, DirectX 9.0 (June 2005 update), and DirectX for Managed Code 1.0.2902.0.

We measured four GPUs, spanning two GPU generations

- NVIDIA GeForce 6800 Ultra, 256MB RAM, eVGA
- NVIDIA GeForce 7800 GTX, 256MB RAM, eVGA (newer)
- ATI x850 XT Platinum Edition, 256 MB RAM
- ATI x1800 (newer)

The Accelerator measurements include just-in-time compilation overhead and other execution costs of the library (on average about 9% of the running time). The results for Accelerator and the hand-coded pixel shaders do not include the transfer times for getting the initial data onto the GPU and reading the final result back. For long-running benchmarks, this makes little difference, *e.g.*, $< 2\%$ for motion estimation, $< 20\%$ for stereo matching; the short-running benchmarks are meant to model small pieces of a computation, and the communication time should be amortized over the whole program.

Figure 7 shows the performance difference between Accelerator and hand-coded pixel shader code, as speedups versus C++ running on the CPU. The graph uses a logarithmic scale; higher bars means better performance. Speedups of less than 1 imply that the CPU outperforms the other systems. On six benchmarks, Accelerator is within a factor of two of the performance of hand-coded pixel shader code.

There are cases where there is room for further optimization work. The most striking differences are in Rotate and Motion Estimation. These benchmarks involve non-uniform memory access (gather), where out-of-bounds access needs to be treated specially. While GPUs have direct hardware support for this operation, Accelerator cannot access it through DirectX C# graphics APIs.

On the smaller benchmarks (Sum, Matrix-Vector and Matrix-Matrix multiplication), we have had promising results when we change our model for texture lookup, allowing us to elide the vertex data that we currently send to the GPU before invoking the generated pixel shader. We speculate that this is the main source of discrepancy between the timing results for Accelerator and the hand-coded shaders.

Figure 8 shows the speedup of the benchmarks running in Accelerator versus those written in C++ and running on the CPU. We show the speedup using each GPU. For the Life, Demosaic, and Convolve benchmarks, where data access is regular and local, Accelerator on the GeForce 7800 GTX is 10–18×faster than the C++ version. For Rotate, Corner Detection, and Stereo Matching, with less localized data access, Accelerator is 3–4×faster. For the Sum and Matrix-Vector multiplication benchmarks, we find that Accel-

Benchmark Name	Size	C++ Time	Description
Sum	Primitive	1ms	Compute the sum of absolute values of a 1000×1000 matrix. Corresponds to the SASUM primitive of BLAS.
Matrix-vector multiplication	Primitive	1ms	Multiply a 1000×1000 matrix by a vector. Corresponds to the SGEMV primitive of BLAS.
Matrix-matrix multiplication	Primitive	303ms	Multiply two 1000×1000 matrices. Corresponds to the SGEMM primitive of BLAS.
Life	Method	45ms	Compute one iteration of Conway's "Game of Life" [6] on a 1000×1000 grid. Cell values are represented by single-precision floating-point numbers to model a general cellular automaton.
Demosaic	Method	94ms	Compute an RGB image from a 1000×1000 pixel Bayer pattern. The RAW file produced by a typical CCD contains information for only one color at each pixel, in what is known as a Bayer pattern. Reconstruction of the full-color image, called "demosaicing," is done by interpolation. We use a recent high-quality algorithm [12].
Convolve	Method	86ms	Convolve a 1000×1000 monochromatic image with a 5×5 Gaussian filter. A Gaussian filter is separable, allowing us to optimize this operation into a convolution with a 5×1 filter, followed by a convolution with a 1×5 filter.
Rotate	Method	129ms	Rotate a 1000×1000 color image by $10°$, with bilinear inter-pixel interpolation and cropping.
Corner detection	Module	152ms	Find corner-like features in a 1000×1000 monochromatic image, using the KLT algorithm [11, 21, 5]. The computation has these steps: 1. Convolve with a 5×5 Gaussian filter to reduce noise. 2. Compute C, the Hessian of the image, at each point. 3. Convolve C with a 9×9 Gaussian filter. 4. Find the eigenvalues λ_1, λ_2 of C. 5. The "cornerness" is $\gamma\lambda_1 + \lambda_2$, where $\gamma = 0.5$.
Motion estimation	Module	69ms	Perform MPEG-style hierarchical motion estimation on two 512×512 monochromatic images, with quarter-pixel accuracy [17]. For each 16×16 pixel "macroblock," estimate a motion vector that takes the block in the first image to the most similar one in the second image. Start by searching a 7×7 pixel region in a 4×downsampled image. Refine this result by searching a smaller neighborhood in a 2×downsampled image, then in the original image, and finally in a 2×and a 4×subsampled image.
Stereo matching	Module	702ms	Given two pictures, taken by cameras separated by a small horizontal distance, compute the distance at each pixel [19]. This is a brute-force image registration solver. At each pixel, find the offset that produces the least SSD over a 7×7 pixel neighborhood.

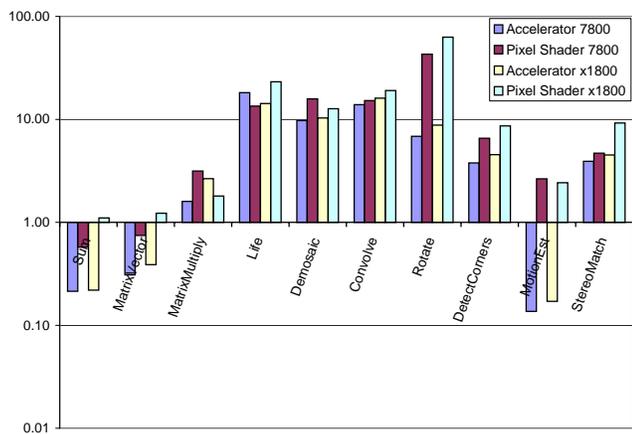**Table 5.** Benchmark descriptions

Figure 7. Performance of Accelerator versus hand-coded pixel shader programs on a GeForce 7800 GTX and an ATI x1800. Performance is shown as speedup relative to the C++ versions of the programs.

Figure 8. Speedup of Accelerator programs on various GPUs compared to C++ programs running on a CPU

erator is 4–5×slower than the CPU. This is because the GPU computation model does not allow accumulation of results across different pixels, forcing us instead to use a log-height reduction, *i.e.*, the number of render passes is logarithmic in the size of the input.

7. Conclusion

We have demonstrated how to translate high-level data-parallel operations to efficient GPU programs. The data-parallel operations
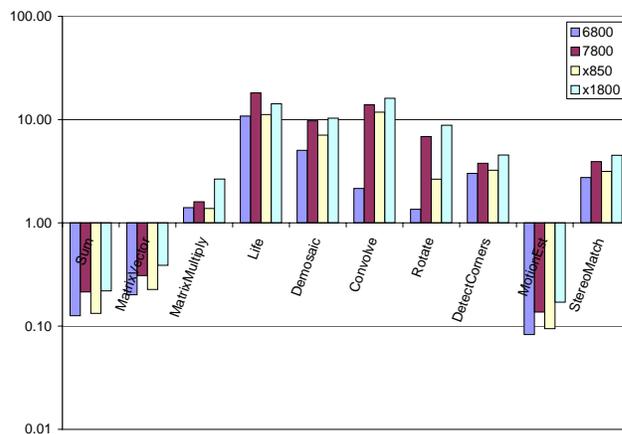
are provided as a data type in a conventional imperative language, making them easily accessible to programmers. The operations are translated on the fly to GPU pixel shader code and graphics API calls. We have developed a set of techniques to partition the operations into GPU pixel shader programs. Due to the high overhead of running a pixel shader compared to the cost of floating-point computation, minimizing these boundaries can be key to performance. The techniques include combining element-wise operations, pushing transformation operations across element-wise operations to leaves, combining transformation operations, and iden-

tifying common subexpressions. The performance of some benchmarks is comparable to hand-written pixel shader programs.

Our work demonstrates that it is possible to compile high-level data-parallel language extensions to mass-market parallel processors that are available today. It suggests a number of interesting directions to pursue. First, there is further work to be done on improving compilation of programs to current GPU programming models. Second, it is worthwhile to explore compiling language extensions for data parallelism to GPUs. This would allow ahead-of-time compilation and the use of generalized reduction and map operations. Finally, it suggests that it is worth investigating changing instruction sets used for distributing programs to represent parallelism explicitly at a higher level than today's processor ISAs, for example, by using virtual machine ISAs.

References

[1] BLELLOCH, G. E. NESL: A Nested Data-Parallel Language. Tech. Rep. CMU-CS-93-129, April 1993.

[2] BLYTHE, D. The Direct3D 10 System. *Transactions on Graphics 25*, 3 (Aug. 2006), 724–734.

[3] BUCK, I., FOLEY, T., HORN, D., SUGERMAN, J., FATAHALIAN, K., HOUSTON, M., AND HANRAHAN, P. Brook for GPUs: Stream computing on graphics hardware. *Transactions on Graphics 23*, 3 (Aug. 2004).

[4] BUDD, T. A. An APL compiler for a vector processor. *ACM Transactions on Programming Languages and Systems 6*, 3 (July 1984), 297–313.

[5] E.TRUCCO, AND VERRI, A. *Introductory Techniques for 3-D Computer Vision*. Prentice Hall, 1998.

[6] GARDNER, M. The fantastic combinations of John Conway's new solitaire game "life". *Scientific American 223* (1970), 120–123.

[7] GUIBAS, L. J., AND WYATT, D. K. Compilation and delayed evaluation in APL. In *POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (New York, NY, USA, 1978), ACM Press, pp. 1–8.

[8] LASTRA, A., LIN, M., AND MONOCHA, D., Eds. *2004 ACM Workshop on General-Purpose Computing on Graphics Processors* (August 2004). http://www.cs.unc.edu/Events/Conferences/GP2/.

[9] LIN, C., AND SNYDER, L. ZPL: An array sublanguage. In *Languages and Compilers for Parallel Computing* (1993), pp. 96–114.

[10] LINDHOLM, E., KILGARD, M. J., AND MORETON, H. A user-programmable vertex engine. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques* (2001), ACM, pp. 149–158.

[11] LUCAS, B., AND KANADE, T. An iterative image registration technique with an application to stereo vision. In *IJCAI81* (1981), pp. 674–679.

[12] MALVAR, H. S., WEI HE, L., AND CUTLER, R. High-quality linear interpolation for demosaicing of bayer-patterned color images. In *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)* (2004).

[13] MARK, W. R., GLANVILLE, R. S., AKELEY, K., AND KILGARD, M. J. Cg: A system for programming graphics in a c-like language. *Transactions on Graphics 22*, 3 (2003), 896–907.

[14] MCCOOL, M., AND TOIT, S. D. *Metaprogramming GPUs with Sh*. A K Peters, 2004.

[15] MONTRYM, J., AND MORETON, H. The GeForce 6800. *IEEE Micro* (March—April 2005), 41—51.

[16] PHARR, M., AND FERNANDO, R., Eds. *GPUGems2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley, 2005.

[17] RICHARDSON, I. E. G. *Video Codec Design*. John Wiley & Sons, 2002.

[18] SABOT, G. W. *The Paralation Model : Architecture-Independent Parallel Programming (Artificial Intelligence)*. The MIT Press, 1988.

[19] SCHARSTEIN, D., SZELISKI, R., AND ZABIH, R. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms, 2001.

[20] SCHOLZ, S.-B. Single assignment C - Functional programming using imperative style. In *Proceedings of the 6th International Workshop on Implementation of Functional Languages (IFL94)* (1994), pp. 21.1–21.13.

[21] SHI, J., AND TOMASI, C. Good features to track. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR'94)* (Seattle, June 1994).