GPUPY: EFFICIENTLY USING A GPU WITH PYTHON

By

BENJAMIN EITZEN

A thesis submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

AUGUST 2007

To the Faculty of Washington State University:

The members of the Committee appointed to examine the thesis of BENJAMIN EITZEN find it satisfactory and recommend that it be accepted.

_____
Chair

_____

_____

ACKNOWLEDGEMENT

I would like to thank the following people, without whom I could not have finished my thesis. Megan, my fiancé, for moving with me to Pullman and patiently waiting, and waiting, for me to finish; Bob, for coming up with the idea for GpuPy and provided constant guidance and much-needed encouragment; Roger and the department, for generously providing me with funding; and finally my parents, for demonstrating the value of having a good education.

GPUPY: EFFICIENTLY USING A GPU WITH PYTHON

Abstract

by Benjamin Eitzen, M.S.
Washington State University
August 2007

Chair: Robert R. Lewis

Originally intended for graphics, a Graphics Processing Unit (GPU) is a powerful parallel processor capable of performing more floating point calculations per second than a traditional CPU. However, the key drawback against the widespread adoption of GPUs for general purpose computing is the difficulty of programming them. Programming a GPU requires non-traditional programming techniques, new languages, and knowledge of graphics APIs. GpuPy attempts to eliminate these drawbacks while still taking full advantage of a GPU. It does this by providing an implementation of an existing numerical API for the Python programming language using a GPU.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

Dedication

To Megan and Oscar, the most wonderful fiancé and dog, respectively, in the world.

Don’t worry, I’m almost done.

CHAPTER ONE

INTRODUCTION

The specialized processors on modern video cards are called *Graphics Processing Units*, or GPUs. For certain algorithms, a GPU can outperform a modern CPU by a substantial factor [15]. The goal of this project is to provide an easy interface for taking advantage of the strengths of a GPU.

GpuPy is an extension to the Python programming language, it provides an interface modeled after the popular NumPy Python extension. Implementing an existing interface on a GPU is beneficial because it eliminates the need to learn a new API and lets existing programs run faster without being rewritten. For some programs, GpuPy provides a drop-in replacement for NumPy; for others, code must be modified.

Chapter 2 provides background information necessary to understand the remainder of this thesis. Chapter 3 gives a high-level description of GpuPy. Chapter 4 details the implementation of GpuPy. Chapter 5 evaluates the accuracy and performance of GpuPy. Chapter 6 concludes, and Chapter 7 details potential future work involving GpuPy.

CHAPTER TWO

BACKGROUND

In order to understand GpuPy, an overview of the underlying technology involved is helpful. The following sections discuss the background information necessary to understand GpuPy.

2.1 GPUs

Almost all modern desktop and laptop computers contain a Graphics Processing Unit (GPU). A GPU is a parallel processor designed to render images. GPUs have evolved rapidly in the last several years; much more so than traditional CPUs such as those manufactured by Intel and AMD. Their degree of parallelism is constantly increasing and they now have a greater number of transistors and are capable of performing more floating point operations per second than traditional CPUs.

2.1.1 OpenGL Rendering Pipeline

GPUs are designed to render complex 3d geometry in real time. Input is passed to a GPU as a collection of vertices, matrices, texture coordinates, texture data, and lighting parameters. A GPU processes the input and produces an image which can then be shown to the user. The sequence of steps by which a GPU produces an image is called the rendering pipeline. Figure 2.1 is a block diagram of the OpenGL rendering pipeline.

When a program uses OpenGL to render an image, it provides the rendering pipeline with a set of vertices and parameters. The vertices are specified in *object coordinates*, which can be thought of as a vertex's location in space. The vertices will then follow a path

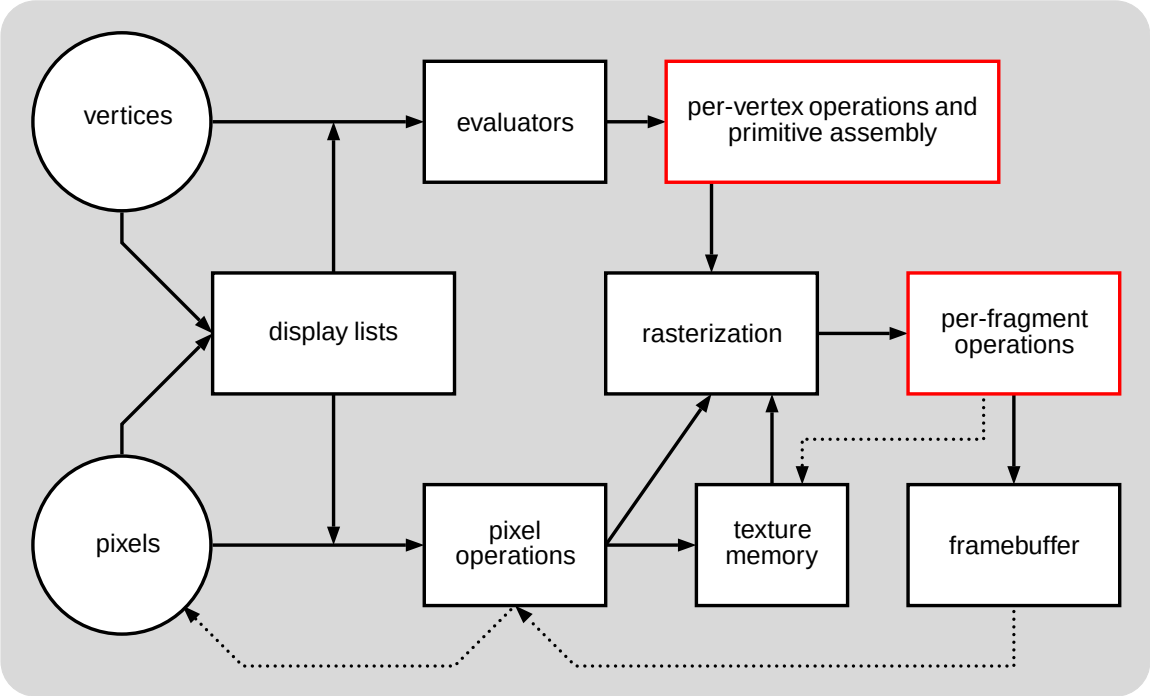

Figure 2.1: The OpenGL Rendering Pipeline. This figure shows how the major components of an OpenGL system fit together. Traditionally, the *per-vertex operations and primitive assembly* and *per-fragment operations* stages have performed fixed functions, but more recent GPUs and APIs allow these stages to be customized via short programs called *shaders*.

through the rendering pipeline, which will eventually output an image to the framebuffer.

The sequence of actions performed by the rendering pipeline to rendering a quadrilateral would be as follows. A program provides the four vertices that make up the corners of the quadrilateral (the *vertices* stage). Each vertex has associated with it a color, a surface normal, and one or more texture coordinates. The *display list* and *evaluator* stages simply provide alternate methods for specifying vertices to OpenGL, and will not be discussed further. The next stage is *per-vertex operations and primitive assembly*. Each vertex is

transformed from object coordinates to eye coordinates using the *model-view* matrix, allowing the vertices to appear as they would if viewed from an arbitrary location. The position and surface normal of a vertex are changed, but the color and texture coordinate(s) remain the same. The vertices are then transformed again, this time by the projection matrix, which maps the vertices to a view volume and possibly adjusts them to account for perspective (more distant objects appear smaller). The vertices are grouped into primitives (points, line segments, or polygons) and any vertices that fall outisde the view volume are discarded, or "clipped." The next stage is *rasterization*, which generates "fragments" for each pixel of the primitives. Fragments are similar to pixels, but contain information in addition to color. Each fragment has a depth value and texture coordinate(s) associated with it. These values are calculated by interpolating the corresponding values from the vertices across the face of the primitive. The resulting fragments are passed to the *per-fragment operations* stage, which performs final processing on the fragments before outputting them to the framebuffer. One common operation performed in this stage is depth buffering. With depth buffering, an incoming fragment only results in a pixel when the fragment's depth value is less than the depth of the existing pixel at the same location. For a more in-depth description of the OpenGL rendering pipeline, see [20].

2.1.2 Texture Mapping

A texture map is a 1-, 2-, or 3-dimensional array of elements, typically containing image data. An individual texture element, called a "texel", has one or more scalar components. These scalar components are typically describe an RGBA color value, but can also be more general 32-bit floating point values. Texture maps are used by OpenGL to "paint" an image onto the surface of a primitive. If texturing is enabled, then the *rasterization* stage

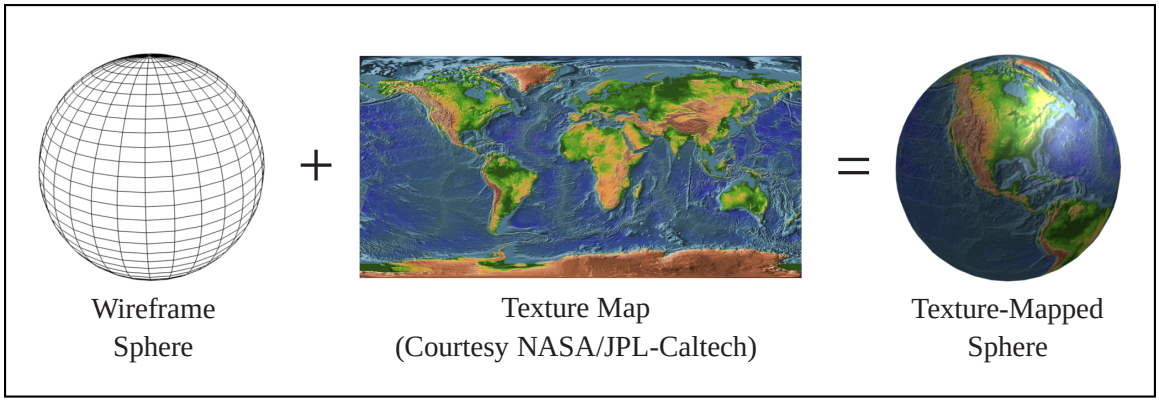

Figure 2.2: Example of Texture Maping. Texture coordinates are specified for each vertex in the sphere. During rasterization, texture coordinates are calculated for each fragment using interpolation. The interpolated texture coordinates are then used to read values from the texture map. This allows images to be "painted" onto primitives.

calculates the color of each fragment using colors from the texture map. The value to use is determined by the interpolated texture coordinates for each fragment. Figure 2.2 shows an example of this process.

2.1.3 Programmable Pipeline

Traditionally, the *per-vertex operations and primitive assembly* and *per-fragment operations* stages performed fixed functions. As a demand for high quality computer graphics emerged, fixed functions were no longer sufficient, and these stages were made more flexible. In modern GPUs, these stages can be fully customized using short programs called "shaders."

Shaders can be programmed in a variety of languages, the most popular of which are Cg, HLSL, and GLSL. The preceding three languages are all high-level languages whose

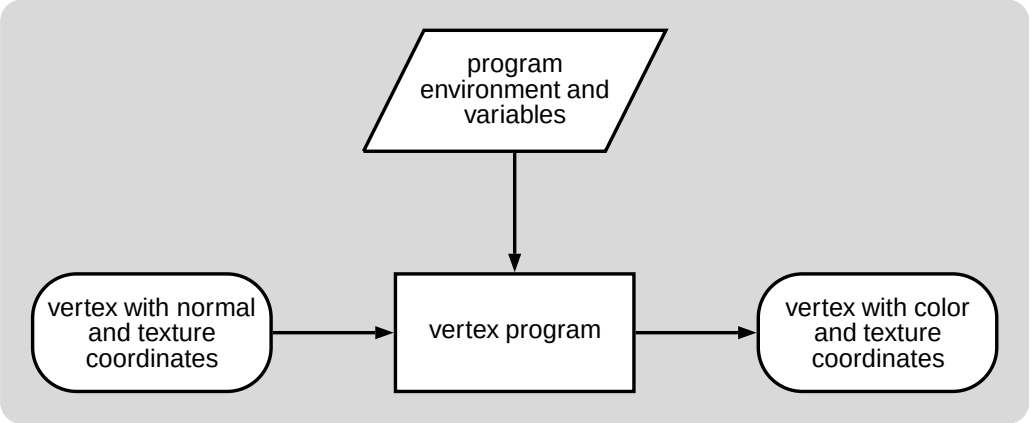

Figure 2.3: Vertex Program Block Diagram.

syntax is similar to other imperative languages. Shaders can also be written using a GPU-specific assembler. Most current pipelines contain two stages that are programmable: vertex shading and fragment shading.

Vertex shaders run during the *per-vertex operations and primitive assembly* stage. They accept a single input vertex and produce a single output vertex. In addition to the input vertex, vertex shaders also have access to global parameters such as light positions and material properties. When a vertex shader is enabled, it is executed once for each input vertex and the output vertex continues through the pipeline as usual. Each vertex is indendent and can therefore be processed in parallel. Figure 2.3 shows the high-level behavior of a vertex shader. Vertex shaders are not currently used by GpuPy.

Fragment shaders run during the *per-fragment operations* stage. They accept a single input fragment and produce a single output pixel. Fragment shaders have access to the same global parameters as vertex shaders, but are also able to read values from texture memory. When a fragment shader is enabled, it is executed once for each input fragment. Like

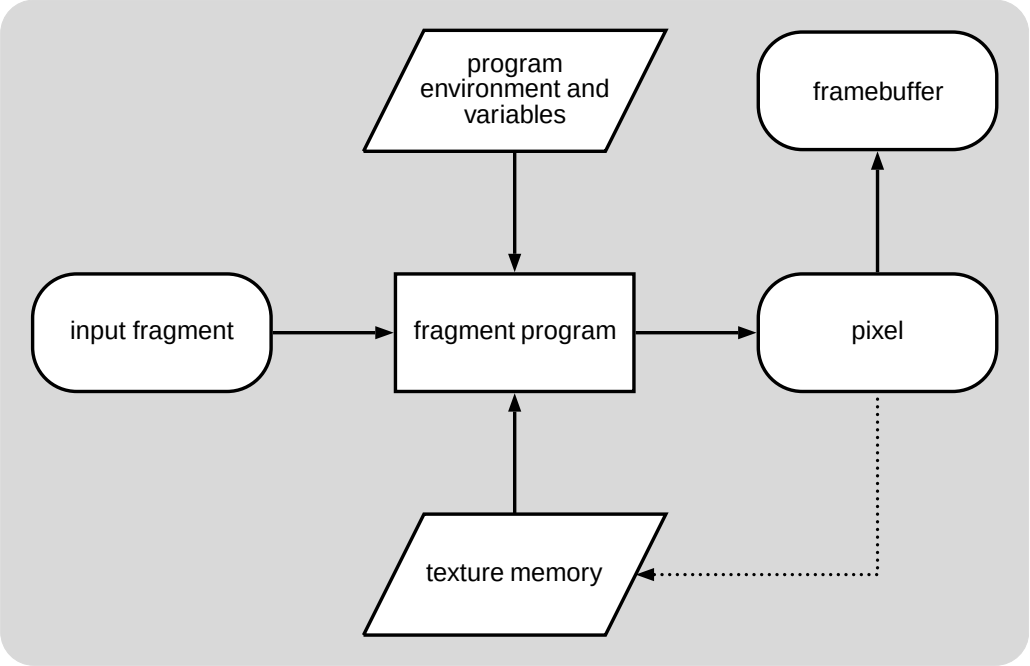

Figure 2.4: Fragment Program Block Diagram. Note that output from a fragment shader can be redirected back to texture memory. This is important when using fragment shaders for general purpose computing.

vertices, fragments can be processed in parallel. Figure 2.4 shows the high-level behavior of a fragment shader.

2.2 Stream Processing

Stream processing is a model of computation in which a "kernel" function is applied to each element in a stream of data. Because each element of the data stream is processed independently, stream processing can easily be done in parallel. Although this can be accomplished to some extent using standard hardware, custom hardware is often used [7]. As will be discussed in the following sections, both GPUs and NumPy fit into the stream

processing model. For examples of stream processing applications, see [8] and [12].

2.3 GPGPU

In the last few years, a significant amount of work has gone into developing ways to use GPUs to perform general purpose computations. GPGPU, which stands for *General Purpose GPU*, is an initiative to study the use of GPUs to perform general purpose computations instead of specialized graphics algorithms [11]. The two most important features to GPGPU are a programmable pipeline and floating point textures.

The rendering pipeline described above can be exploited to act as stream processor [27, 18]. This is done by using texture maps to hold data streams and shaders to implement kernels. For example, when fragment shading is enabled and a texture-mapped quadrilateral is properly rendered, the fragment program will be executed once for each interior fragment of the quadrilateral. The interpolated texture coordinates for each fragment are used to look up values from texture maps. The output of the fragment shader is then written into texture memory using the OpenGL Framebuffer Object Extension [4]. Texture coordinates must be chosen that cause each fragment's interpolated texture coordinates to reference the correct texel. The code in Appendix A.5 renders a quadrilateral with texture coordinates of the four vertices set to the positions of the vertices. This generates interpolated texture coordinates that sample all of the texels in a texture of the same size. Figure 2.5 shows the interpolated texture coordinates generated by the code in Appendix A.5.

There are a number of limitations that must be observed when writing a fragment shader. The output location is fixed for each execution of a shader. This means that a shader chooses the value, but cannot choose the location to which it will be written. Texture also

```
glDrawQuad(0,0,8,4);
```

(0, 0)	(1, 0)	(2, 0)	(3, 0)	(4, 0)	(5, 0)	(6, 0)	(7, 0)
(0, 1)	(1, 1)	(2, 1)	(3, 1)	(4, 1)	(5, 1)	(6, 1)	(7, 1)
(0, 2)	(1, 2)	(2, 2)	(3, 2)	(4, 2)	(5, 2)	(6, 2)	(7, 2)
(0, 3)	(1, 3)	(2, 3)	(3, 3)	(4, 3)	(5, 3)	(6, 3)	(7, 3)

Figure 2.5: Lining Up Texture Coordinates. This figure shows the interpolated texture coordinates resulting from a call to the code in Appendix A.5.

may not be written to if they will be read again while rendering the current primitive. There are also limitations on the resources that can be used during a shader execution. The nature of the resource limits depend on configuration, but generally reflect limitations of the GPU hardware itself such as maximum number of instructions, maximum number of texture instructions, or maximum number of temporary registers. This will be covered in more depth in a later section.

The floating-point values used by most GPUs do not conform to IEEE floating-point standards. Some higher-end GPUs, such as NVIDIA's Quadro product line, support IEEE single-precision values, but others do not. Because of this, GPU algorithms may produce slightly different results than when the same algorithm is run on a CPU. For many applications this is perfectly acceptable but for others it may be problematic. GPUs also support a 16-bit, or half-precision floating point value. If an application can tolerate the reduction in precision, performance may be improved by using 16-bit values. An example of an application that has low precision requirements is ray tracing, and to some extent, anything intended for human visual consumption. Although the current IEEE floating point standard

does not include a 16-bit type, a draft revision of the standard does [14].

2.4 Python

Python is a popular object-oriented programming language that is in wide use throughout the computer industry. Python is an interpreted language like Java or Perl, and like these languages, makes use of a virtual machine. It is designed to be easy to use, yet fully featured. Python is very portable and runs on a variety of platforms [24, 26].

2.4.1 Extending Python

An important feature of Python is that it was carefully designed to be easily extensible. This is accomplished through the use of modules written in C or C++. These modules can be used to extend the functionality of the Python interpreter by adding new functions and object types [25]. An extension module can specify new functions and object types by providing C structs whose members include callback functions and auxiliary data that specify the behavior of the functions or object types under different circumstances.

The callback functions are organized into groups of related operations called protocols. A protocol is essentially a collection of callback functions, and an object type may implement whichever protocols are appropriate for that type. Unneeded callback functions may be left unimplemented. The current version of Python defines the following protocols: object, number, sequence, mapping, iterator, and buffer. The object protocol provides the basic functionality that most objects will implement. The number protocol provides binary operations such as addition and subtraction. The sequence protocol provides operations required to treat objects like arrays. The mapping protocol is similar to the sequence protocol, but allows any Python object to be used as an index, rather than just integers. The

iterator protocol provides a way to visit each member of a container object. The buffer protocol allows the memory containing an object's data to be accessed directly from outside the extension module.

2.4.2 Slicing

Python has a somewhat unique feature called "slicing" that allows subsets of sequences to be selected using the mapping protocol. A slice object is composed of three integers: `start`, `stop`, and `stride`. A slice is represented in Python by three integers separated by colons. Integers omitted take on default values. The default for `start` is 0, the default for `stop` is the length of the sequence being sliced, and the default for `stride` is 1. When a slice object is used to index a sequence object, a new sequence constructed by selecting elements from the original array starting with `start` (i.e., inclusive), ending just before `stop` (i.e., exclusive), and skipping `stride` elements between selections. The three slice arguments can be thought of as the three parameters of a basic `for` loop that produce the desired indices.

2.5 NumPy

NumPy is a Python extension module written by Travis Oliphant and others [17]. It is the successor to Numarray and Numeric, two previous numerical Python extensions. NumPy provides several object types, the most important of which is `NDarray`. This type is used to implement N-dimensional arrays.

NumPy allows mathematical operations to be performed on arrays as though they were scalars. When a mathematical operation is performed on one or more `NDarray` objects, the operation is applied in an element-wise fashion. The result is a new `NDarray` object

whose shape is determined by the shapes of the operands. NumPy's `NDarray` object type implements slicing, but with one minor difference from conventional Python sequence semantics: A slice of an `NDarray` object always refers to the same data as the original object. For instance, if $b = a[:: 2]$, then b contains the elements of a that occur at even indices. Since b refers to the same data as a, changes made to the shared elements will affect both.

2.5.1 Shape, Strides, and Slicing

NumPy describes the contents of an `NDarray` object with a data pointer, the number of dimensions, the shape, and the strides. The *shape* of an array is its size along each dimension and the *strides* of an array describe the distance in linear memory between logically consecutive array elements. For example, an array whose shape is $(2, 3, 4)$ has 24 ($2 \times 3 \times 4$) elements and a contiguous array whose shape is $(2, 3, 4)$ and whose elements are 4 bytes, would have strides of $(48, 16, 4)$.

These four properties are available to the Python programmer, but are primarily used internally by the NumPy extension module. The shape and strides entries both have one entry per dimension. When a slice of an `NDarray` object is created, the new `NDarray` object points to the original array but has its own number of dimensions, shape, and strides.

2.5.2 Broadcasting

In order to allow `NDarray` objects with different but in some sense compatible shapes to be operated on together, NumPy uses a concept called *broadcasting*. It allows the shape of an `NDarray` object to be modified to match the shape of another `NDarray` object. In order for an operation to be valid on `NDarray` objects, all of the operands need to

be broadcast-compatible with each other. Broadcasting is performed on two `NDarray` objects and always converts the shape of the `NDarray` object with fewer dimensions to the shape of the other. Given an `NDarray` object and a target shape, broadcasting tries to find a representation of the `NDarray` object that fits the required shape. If the `NDarray` object being broadcast has fewer than the desired number of dimensions, 1s are repeatedly prepended to its shape until it has the correct number of dimensions. The new shape is then compared to the target shape and must match for the operation to proceed. A shape matches if the corresponding values are identical or at least one of them is equal to 1. Broadcasting can be performed on more than two `NDarray` objects by repeating the process $N-1$ times, where N is the number of `NDarray` objects. For the purposes of broadcasting, scalars are treated like an array with zero dimensions, which makes them broadcast-compatible with any array. A more detailed discussion of broadcasting can be found in [16].

As an example, let us suppose that we have two `NDarray` objects: A and B. If A's shape is $(11, 5, 7)$ and B's shape is $(5, 7)$, then B can be broadcast to correspond to A. The first step would be to prepend 1s to B's shape until it had the same number of dimensions as A, which effectively makes B's shape $(1, 5, 7)$. The second step would be to compare B's effective shape to A's shape. All of the corresponding shape entries match or are equal to 1, therefore A and B are broadcast-compatible. If B's shape were $(11, 5)$, the arrays would no longer be broadcast-compatible.

2.6 Lazy Evaluation

Most programming languages evaluate expressions when they are assigned, or in Python terminology, "bound" to a variable. This is known as *strict* or *eager evaluation*. Instead

of evaluating expressions when they are bound, it is possible to defer evaluation until the value is actually needed. This is known as *lazy evaluation*. The reasoning behind lazy evaluation is that the contents of a variable are irrelevant until the contents are actually needed. Examples of programming languages that use lazy evaluation are Haskell and Miranda [13, 23].

Instead of storing the result of an expression in a variable, lazy evaluation stores the expression itself, which can eventually be evaluated to produce the desired result. An expression may refer to other expressions. The result is a tree containing operators and operands that is evaluated when the result is actually needed. The benefits of lazy evaluation are avoiding unnecessary and redundant calculations. As demonstrated by Tarditi, et al. [22]; there are additional benefits when using a GPU which will be discussed in the following chapters.

One drawback of lazy evaluation is that when a variable is written to, a copy of the existing value must be saved if any unevaluated expressions depend on it. When the existing value has no dependents, the update can be done in place. This is called a destructive update. Destructive updates are usually preferred, since they do not require any copying.

CHAPTER THREE

USING GPUPY

GpuPy provides a Python extension module that interfaces with a GPU. GpuPy interacts closely with NumPy and provides a very similar interface that is able to execute many NumPy programs with minimal changes. GpuPy uses GPU versions of operations whenever possible and delegates to NumPy when a GPU version of the algorithm is not available. The primary goals of GpuPy are to improve performance over NumPy and to require the fewest changes possible to make an existing NumPy program run correctly using GpuPy. Successfully meeting these goals provides a system that can outperform CPU-only software substantially and has almost no learning curve beyond that of NumPy.

Assuming that GpuPy supports the required features for a given NumPy program, translating it to GpuPy is trivial: Change `from numpy` to `from gpupy`, and `float32` to `gpufloat32`. Appendix A.1 contains source code for a simple NumPy program that renders a single shaded sphere. Only three lines need to be changed to convert it into a GpuPy program. Figure 3.1 shows the three lines that need to be changed.

Some features of NumPy, such as writable arrays and advanced slicing [17], are not yet supported by GpuPy. It should be possible in many cases to modify existing programs to avoid these features.

```
 4  from gpupy import *
    ⋮
17  x = fromfunction(lambda x, y: x, (w, h), dtype=gpufloat32)
18  y = fromfunction(lambda x, y: y, (w, h), dtype=gpufloat32)
    ⋮
```

Figure 3.1: Changes Required to Translate the NumPy Program in Appendix A.1 to GpuPy. Only lines requiring changes are shown and the changes are underlined.

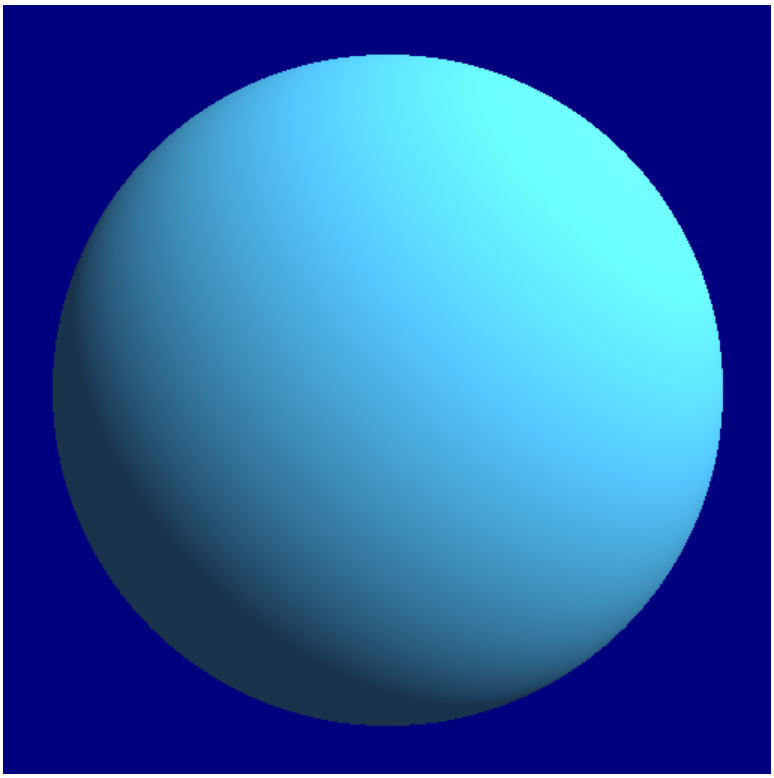

Figure 3.2: Image Produced by the Shaded Sphere Program. This image was produced by the GpuPy version of the program in Appendix A.1. The GpuPy and NumPy versions of the program produce identical images.

CHAPTER FOUR

IMPLEMENTATION

We discuss here the internal (in C) implementation of GpuPy. GpuPy is divided into two layers: the Core Layer and the Driver Layer. The Core Layer is the part of the code that interfaces with Python and NumPy, and the Driver Layer is an implementation of the GpuPy driver model that the Core Layer uses to interface with the GPU. Figure 4.1 illustrates how the various components of GpuPy interacts with eachother.

4.1 The GpuArray Class

The primary class implemented by GpuPy is `GpuArray`. Internally, `GpuArray` objects contain management data and possibly a pointer to an underlying `NDarray` object, which may or may not be `NULL`. The `NDarray` object is what actually contains the array data on the host (when present), the `GpuArray` object stores no array data of its own.

Every `GpuArray` object has a pointer to a `GpuArray` object called the data owner, which may or may not be the same `GpuArray` object. When a `GpuArray` object is not a slice, its data owner is itself and when a `GpuArray` object is a slice, its data owner points to the `GpuArray` object from which the slice was taken. The way GpuPy describes a `GpuArray` object is similar to the way NumPy describes an `NDarray` object, but replaces the data pointer with a pointer to a `GpuArray` object and an offset.

GpuPy therefore represents a `GpuArray` object using five attributes. These attributes are (1) a pointer to a `GpuArray` object, (2) an offset into the object, (3) the number of dimensions in the object, (4) the shape of the object, and (5) the strides of the object. The

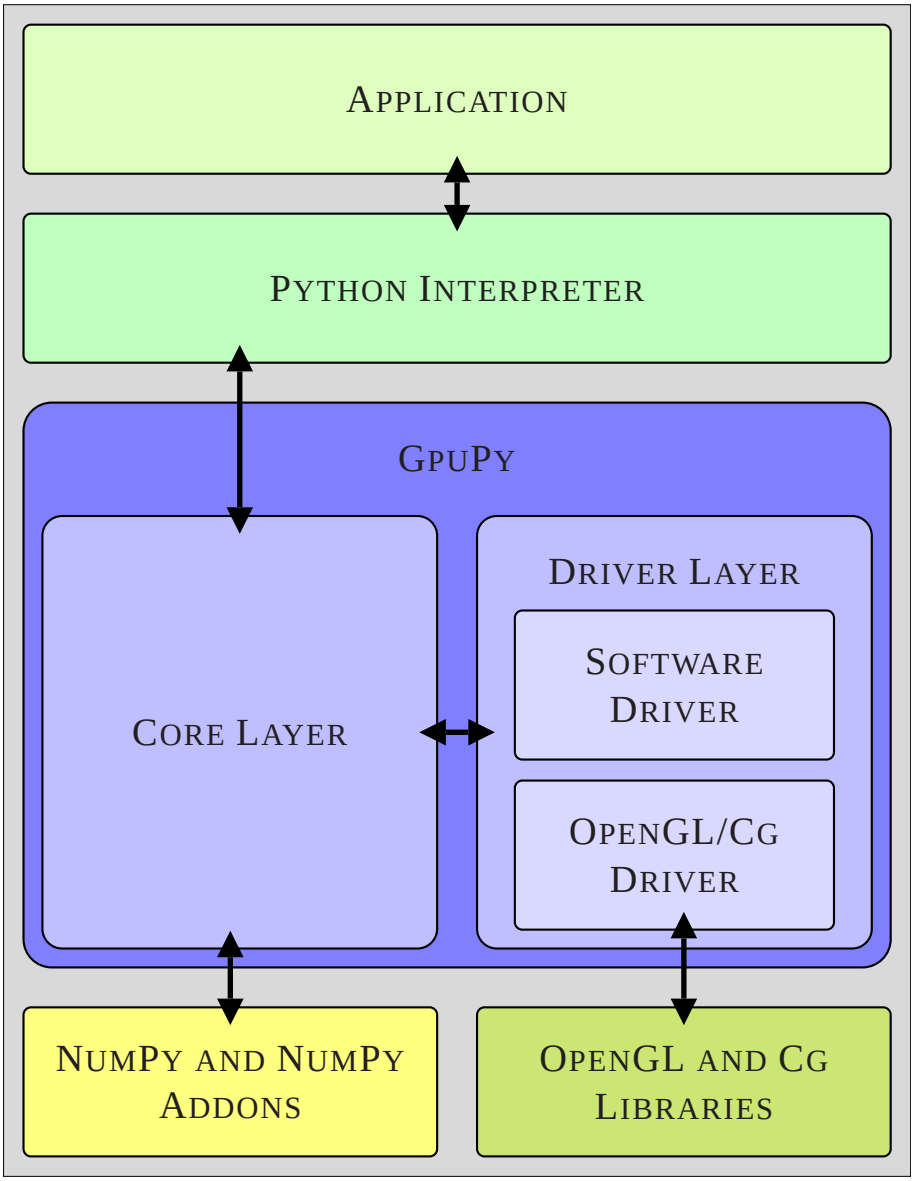

Figure 4.1: Block Diagram of GpuPy. GpuPy is a Python extension module that implements (a subset of) the NumPy API. If a given feature is supported by the Driver Layer, GpuPy will use the Driver Layer, otherwise it will fall back to the NumPy version of the feature. Note that the Driver Layer insulates the rest of GpuPy from the API being used to control the GPU.

`GpuArray` object specifies the data and the remaining attributes specify how that data is viewed. Because there are many possible ways in which a given `GpuArray` object could be viewed, it is best to think of the five attributes as describing a view of a `GpuArray` object, rather than the object itself.

A `GpuArray` object can be one of three types: `ARRAY`, `CONSTANT`, or `EXPRESSION`. If the type is `ARRAY`, then all of the data is present and there is always an underlying `NDarray` object present. If the type is `CONSTANT`, then there is never an underlying NDarray object present and the value of the constant is stored in the `GpuArray` object. If the type is `EXPRESSION`, then there may or may not be an underlying `NDarray` present. When the type is `EXPRESSION`, the `GpuArray` object contains zero or more pointers to child `GpuArray` objects. For instance, if $a = b + c$, then b and c are a's children. Because `GpuArray` objects of type `EXPRESSION` can have part of their data evaluated and part of it unevaluated, they contain an extra bit for each entry in the array that determines whether the corresponding array element has been evaluated.

4.2 Blocks

The size of the data being processed by GpuPy can be very large. This creates a problem for the GPU because a view may not always fit entirely on the GPU. This means that a GPU cannot necessarily process an entire array at once.

In order to handle views of arbitrary size, GpuPy divides each view into fixed-size blocks. An additional attribute, the block number, is added to the view's description in order to describe a block. The six attributes that make up a block represents a single piece of a view, and more importantly, the contents of a GPU texture. Operations in GpuPy are

always performed on blocks. Before a shader is executed, the blocks upon which it depends are copied onto the GPU. Executing a shader produces a block that may be copied back to the CPU.

A block number is similar to page number in a virtual memory system in that it represents a region of memory that may or may not be present on the GPU at any given time. Each `GpuArray` object can be thought of as a region of virtual memory that is possibly backed by blocks on the GPU. In contrast to an operating system's page frames, largely due to striding, a block in GpuPy can be composed of non-contiguous memory. Adjacent elements in a block may not correspond to adjacent elements in the corresponding `NDarray` [21].

4.3 Caching

GpuPy allocates one block for each texture allocated from the Driver Layer. It uses these blocks to track the contents of the GPU and thereby avoid unnecessary copies between the CPU and GPU.

The blocks are tracked by a hash table hashed by the block's six attributes and by a Least Recently Used (LRU) list. The hash table provides a fast way to know whether a block is already on the GPU and the LRU list maintains a list of blocks reverse-sorted by how recently they were accessed by a shader. When executing a shader which depends on a block that is not present on the GPU, the block must be copied to the GPU. This is analogous to demand paging in virtual memory systems. If GPU memory becomes exhausted, then GpuPy must evict a block from the GPU in order to make room for the new block. The least recently used block is a reasonable first choice for eviction [21].

```
1  from gpupy import *
2
3  # a = [0, 1, 2, 3, 4, 5, 6, 7]
4  a = arange(8, dtype=gpufloat32)
5
6  # b = [8, 9, 10, 11, 12, 13, 14, 15]
7  b = arange(8, 16, dtype=gpufloat32)
8
9  c = 3.5
10 e1 = cos(a)
11 e2 = b + c
12 e3 = e1 * e2
13
14 print e3
```

Figure 4.2: A Simple GpuPy Program. This program was intentionally written with each calculation on its own line, which allows the expressions produced to be identified by the line number.

4.4 Lazy Evaluation

In order for GpuPy to perform calculations on a GPU, blocks must be copied to the GPU and the result block must be copied back. Copying blocks often accounts for the majority of the time spent performing calculations on a GPU. Copying is expensive enough that if only a single binary operation is performed on a GPU, it will typically be slower than performing the same calculation on the CPU. As previously suggested by Tarditi, et al., [22]; GpuPy overcomes this limitation by using lazy evaluation. Lazy evaluation can increase the overall performance of a GPU by allowing more operations per block copied, amortizing the cost of copying data between the CPU and GPU. GpuPy implements lazy evaluation by providing operators that, instead of calculating a result, build an appropriate expression that can be

evaluated at a later time.

Figure 4.2 shows an example GpuPy program. The expression tree produced by this code is shown in Figure 4.3. When execution reaches line 14 in Figure 4.2, the expression tree will be evaluated so that the results may be printed.

In order for lazy evaluation to work, GpuPy needs to keep track of which array elements have been evaluated. As mentioned before, all `GpuArray` objects of type `EXPRESSION` contain an extra bit on the host for each element in the array. This bit is cleared when the `GpuArray` object is created and set when a block containing that element is evaluated and copied to the CPU. As in NumPy, slices in GpuPy refer to the same underlying data and therefore don't require any extra bits.

When an element in an array is requested, GpuPy checks the bit for that element to see if evaluation is necessary. If the bit is not set then the block containing the requested element is evaluated and the appropriate bits are set. GpuPy does not allocate a `GpuArray` object's underlying `NDarray` object until the first bit needs to be set. This need may arise for a number of reasons, but in general, it is when the data needs to be in CPU memory. This can happen when the data needs to be displayed for a user, when the data needs to be converted to another data type, or when an operation that cannot be performed on a GPU is required. Evaluating all of the blocks of a `GpuArray` object is called *flushing* and is done when a complete underlying `NDarray` object is needed. This is necessary, for instance, when the underlying `NDarray` object is going to be used by a method not implemented by GpuPy.

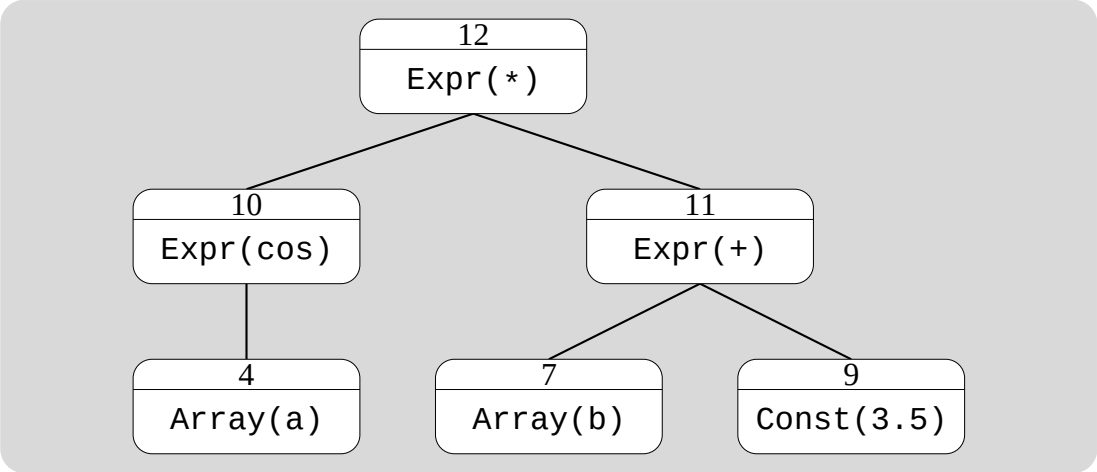

Figure 4.3: Expression Tree Produced from the GpuPy code in Figure 4.2. The number contained in each expression corresponds to the line number in Figure 4.2 that produced it.

4.5 Expression Traversal

When a result is needed, an expression tree must be processed and the correct result produced. Like other trees, a GpuPy expression tree can be processed by performing a depth first traversal. A traversal of a GpuPy expression tree begins with the requested block and recursively calculates its dependencies. The attributes of the blocks encountered during traversal must be propagated to their children. Because attributes propagate to children, the depth-first traversal must perform additional steps each time it visits a `GpuArray` object. For example, if $a = b + c$, and a block describing the even elements of a is requested, then the blocks produced by the traversal should be the even elements of b and c. Figure 4.4 shows how the algorithm used to calculate dependent blocks.

Keeping track of the `GpuArray` object, the number of dimensions, the shape, and the block number are trivial, as each `GpuArray` object contains pointers to its children and

```
1  # Calculate the dependent block
2  def BuildView(block, child):
3      # create a new block that references
4      # the child expression
5      child_block = block(child)
6
7      # the array, number of dimensions, shape,
8      # and block number stay the same
9      child_block.nd = block.nd
10     child_block.shape = block.shape
11     child_block.block_number = block.block_number
12
13     # calculate new strides and offset
14     child_block.offset = child.offset
15     for i in range(block.nd):
16         child_block.strides[i] =
17                 block.strides[i] * child.strides[i]
18         child_block.offset +=
19                 block.array.offsets[i] * child.strides[i]
20
21     return child_block
```

Figure 4.4: The `BuildView` Algorithm. This algorithm constructs a dependent block by combining the attributes of the parent block and the child expression.

the number of dimensions, shape, and block number remain constant during traversal. The offset and strides can change from block to block and therefore present more of a challenge. The proper offset and strides for a block dependency are calculated by combining the parent's view and the child's view as follows: The parent's and child's per-dimension offsets are added together to produce the correct offset and the parent's and child's strides are multiplied together to produce the correct strides for the child's view. A traversal, then, produces a topological ordering of the block dependencies of the requested block.

4.6 Intermediate Representation

In order for code to be generated in a driver independent way, the Core Layer builds an *intermediate representation* (IR) that evaluates to the requested block. IR code is generated by traversing the tree as describe above. For each block encountered during the traversal, an IR instruction for the block is created and added to a list. Cached blocks whose `GpuArray` object is of type `EXPRESSION` are treated as if their type was `ARRAY`. When this traversal completes, the list contains a sequence of operations that can be performed to get the requested block. In order to produce reasonably optimized code, common expressions are eliminated during this process. This is done by maintaining a hash table containing the blocks encountered during the traversal. Whenever an IR instruction is created, the hash table is consulted to see if that block has already been encountered during this traversal. If the block has already been encountered, then the IR instruction from the previous occurrence is used and recursive processing of the block is unnecessary The IR hash table is cleared before each traversal. Figure 4.5 shows the IR produced from the expression tree in Figure 4.3.

For two IR instructions to be recognized as the same, the corresponding blocks must be identical. This means that the `GpuArray` object pointers for the blocks must point to the same `GpuArray` object. GpuPy must do extra work to ensure that this is always the case. When a new `GpuArray` object is created, GpuPy must check to see if an equivalent expression has already been created and if so, return a reference to that object and not to a newly created `GpuArray` object. Depending on the type of the `GpuArray` object, this comparison works differently. If the type is `CONSTANT`, then two `GpuArray` objects are equivalent if they have the same value for the constant. If the type is `EXPRESSION`, the two `GpuArray` objects are equivalent if they have the same opcode and the same operands. If the opcode is symmetric (the order of operands doesn't matter) then the operands are sorted by pointer value before this comparison is performed [3]. When the type is `ARRAY`, no checking is done. This is because comparison of this type can be expensive and it is easy for a programmer to avoid creating equivalent arrays. Because a block always refers to a data owner, slices do not need to be checked for equivalence.

4.7 Driver Model

In order to provide a more extensible system, GpuPy implements a driver layer that abstracts the GPU capabilities needed by the Core Layer. Most importantly, this allows drivers for different GPU architectures to be easily implemented and tested. It also allows the Core Layer to be indifferent to the precise method used for programming the GPU. GpuPy's driver model is similar to an operating system's I/O driver model, but provides additional features specific to GpuPy.

The Driver Layer specifies 14 essential functions that each driver implementation must

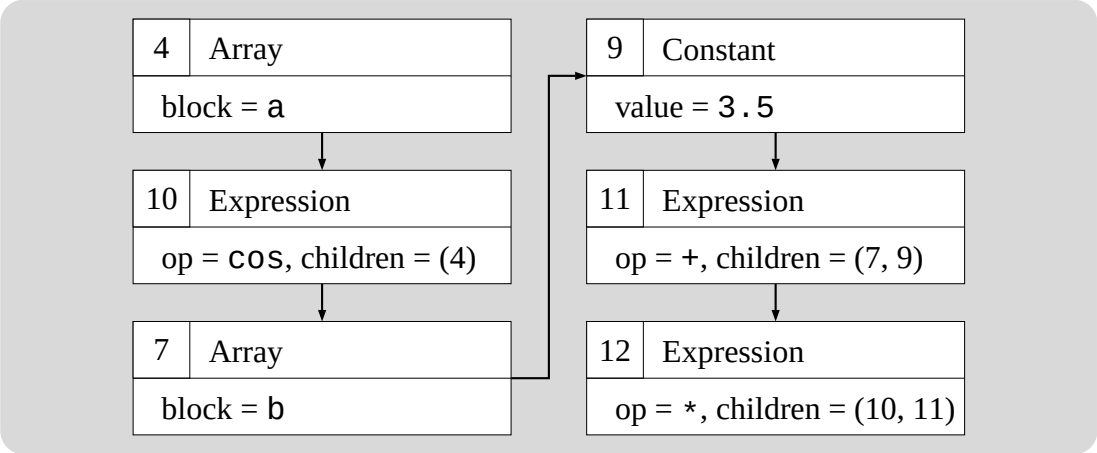

Figure 4.5: Intermediate Representation (IR) for the Expression in Figure 4.3. The numbers in this figure correspond to the numbers in Figure 4.3. GpuPy uses IR to describe shader code in a driver-independent way.

provide. These functions are `init()`, `cleanup()`, `get_method()`, `block_alloc()`, `block_free()`, `block_read()`, `block_write()`, `check_ir()`, `eval_ir()`, `resource_get()`, `resource_zero()`, `resource_add()`, `resource_sub()`, and `resource_check()`. In addition to the 14 essential functions, a driver also provides functions for any of the NumPy functionality that it knows how to reproduce. These additional functions depend on the type of GPU being used and are chosen from the set of all methods known to NumPy.

4.7.1 Infrastructure

Infrastructure driver functions are the most basic functions that a driver must provide. They are responsible for initialization, cleanup, and describing driver capabilities to the Core Layer.

- `init()`:

  Allocates memory, initializes required APIs (e.g., OpenGL), and returns an opaque pointer to the driver's private data structure. The private data structure is provided to all remaining Driver Layer functions. If this function succeeds, then the remaining Driver Layer functions are ready to be called. If this function fails, then the driver could not be initialized and GpuPy will not be able to perform any calculations.

- `cleanup()`:

  Reverses any actions performed by the `init()` function. This function is not currently used by GpuPy, but is included for completeness and the future possibility of dynamically changing drivers. This function must not fail.

4.7.2 Block-Related

Block-related functions allow the Core Layer to allocate, free, and control the contents of GPU textures.

- `block_t *block_alloc(int type)`:

  Allocates and a GPU texture and returns a pointer to it. If there are no more textures available, `NULL` is returned and the Core Layer knows that it must free a GPU texture before it can allocate more. The `type` argument specifies what type of block to allocate. GpuPy currently only supports a single block type (`gpufloat32`), but will probably be expanded to support other types.

- `void block_free(block_t *block)`:

  Frees a GPU texture allocated by a call to `block_alloc()`.

- `int block_read(block_t *block, float *buf)`:

  Copies the contents of a block from the Driver Layer to the CPU. The `block` parameter specifies the block to read and `buf` is the location to which to copy the data. This function returns 0 on success and < 0 on failure.

- `int block_write(block_t *block, float *buf)`:

  Copies the contents of a buffer provided by the Core Layer into the specified block. The `block` parameter specifies the block to write and `buf` is the location from which to copy the data. This function returns 0 on success and < 0 on failure.

4.7.3 IR-Related

IR-related functions allow the Core Layer to control shader execution on a GPU without actually knowing anything about how the driver interfaces with the GPU.

- `int check_ir(ir_list_t *list, resource_t *rsrc)`:

  Determines whether the provided IR code can be compiled into a single shader. The `list` parameter contains the IR code to be checked and the `rsrc` parameter is filled in with the resources required to evaluate the result. This function returns > 0 if the IR can be evaluated by a single shader, 0 if it cannot, and < 0 if an error occurs.

- `int eval_ir(ir_list_t *list, block_t *dst)`:

  Evaluates the provided IR code and saves the result in the specified block. The `list` parameter contains the IR code to be evaluated and the `dst` parameter specifies the location to store the result. This function returns 0 on success and < 0 on failure.

4.7.4 Resource-Related

Resource-related functions allow the Core Layer to process expressions based on the resource usage of the current driver. Since the explicit resources may differ between GPU architectures, it is important that the Core Layer be able to handle resources in a generic way. Each driver provides an opaque resource counter type that can be used to estimate the resource requirements.

- `void resource_get(GpuArray *gpa, resource_t *rsrc)`:
  Provides an upper bound of the resources required by the driver to perform the given operation. The Core Layer uses this to produce a resource counter that is an upper bound for an entire expression.

- `void resource_zero(resource_t *rsrc)`:
  Clears the specified resource counter.

- `void resource_add(resource_t *rsrc1, resource_t *rsrc2)`:
  Adds a resource counter to another. This function adds `rsrc1` to `rsrc2` and stores the result in `rsrc1`. It is used by the Core Layer to accumulate resources as an expression is processed.

- `void resource_sub(resource_t *rsrc1, resource_t *rsrc2)`:
  Subtracts a resource counter from another. This function subtracts `rsrc2` from `rsrc1` and stores the result in `rsrc1`. This is used by the Core Layer to refine resource estimates during expression processing.

- `int resource_check(resource_t *rsrc)`:

Checks whether the driver can handle the specified number of resources using a single shader. If the resources specified by `rsrc` can be handled by a single shader, 1 is returned. Otherwise 0 is returned.

We will discuss these in greater detail in Section 5.8.

4.7.5 Function-related

Functionality-related functions describe the capabilities of the current driver.

- `callback_t get_method(int opcode)`:

  Returns a pointer to the function that implements the requested operation. The `opcode` parameter specifies the requested oepration. This is how driver-specific support is implemented. If a driver does not support the requested function, it returns `NULL` and the Core Layer will know that it must fall back to the NumPy version.

All remaining driver functions provide driver-specific implementations of NumPy functionality such as `add()`, `subtract()`, `sin()`, and `exp()`. These functions return a `GpuArray` object representing the appropriate expression tree.

4.8 Partitioning

As mentioned in the background chapter, GPUs place strict limits on the resource usage of shaders. This means that a GpuPy expression may not be able to be evaluated with a single shader. GpuPy must therefore partition its expressions into subexpressions that can be evaluated separately and combined to produce the correct result. In order to partition an expression, GpuPy must select which blocks to evaluate. Once a block has been evaluated,

it can be used as an operand in the next shader, allowing an arbitrary expression to be broken up into a sequence of valid shaders.

The partitioning algorithm works by performing a depth-first traversal of the expression tree. As when building the IR code, cached blocks whose `GpuArray` object is of type `EXPRESSION` are treated as arrays. For each block encountered, the partitioning algorithm decides whether or not it can be evaluated by a single shader. If the block can be processed in a single shader, partitioning continues. If the block cannot be evaluated by a single shader, then the block's children are evaluated and cached one at a time until it can. If a block cannot be evaluated by a single shader after all of its children have been collapsed, then the partitioning algorithm fails.

The simplest method for determining whether a block can be evaluated by a single shader is to build IR code for it and query the driver using the Driver Layer's `check_ir()` function. Unfortunately, this approach requires IR code to be built for every single block dependency of the requested block. Ideally, the partitioning algorithm could determine whether a block can be evaluated by a single shader. Because of the optimizations made while building IR code, the sequence of blocks visited will not be the same as during partitioning. This prevents partitioning from having an easy way to know exactly what the resource usage will be before the IR code is built. Algorithm 4.6 shows the simple partitioning algorithm.

A compromise between building IR code for every block and having a partitioning algorithm that can perfectly count resources is to force the partitioning algorithm keep an upper bound on the resource limit and only build IR code and query the driver when the upper bound is exceeded. In order for this to work, the partitioning algorithm needs to

```
1  def Partition(block):
2      # only process non-leaves
3      if block.type != EXPRESSION or checkCache(block):
4          return SUCCESS
5      # recursively process all subexpressions
6      for child in block.array.children:
7          child_block = BuildView(block, child)
8          Partition(child_block)
9      # can code for this block be compiled?
10     if BuildIrCode(block):
11         return SUCCESS
12     else:
13         # couldn't build code, need to collapse
14         # subexpressions
15         for child in block.array.children:
16             # collapse child...
17             child_block = BuildView(block, child)
18             Evaluate(child_block)
19             # ...and try building the code again
20             if BuildIrCode(block):
21                 return SUCCESS
22         # collapsed all children and still can't
23         # build code
24         return FAILURE
```

Figure 4.6: Simple Partitioning Algorithm. This algorithm recursively partitions an expression into subexpressions that can be evaluated by a GPU. In GpuPy, this algorithm is actually implemented in C, but is shown here in Python for added simplicity.

have some way to track Driver Layer resources. Because the Core Layer does not know anything about the limits imposed by Driver Layer, it uses functions provided by the Driver Layer in order to keep track of resources. The Driver Layer functions listed in 5.7.4; `resource_zero()`, `resource_get()`, `resource_add()`, `resource_sub()`, and `resource_check()`; are used by the partitioning algorithm to maintain an upper bound on the resource usage of an expression.

The Driver Layer provides an upper limit for each possible IR instruction. A reasonable upper bound on resource usage for a list of IR instructions is the sum of the upper bounds of the instructions.

As an example, imagine a traversal that encounters five blocks whose type is ARRAY. When the IR code is built and common subexpressions are eliminated, some of the five blocks may be eliminated because they are redundant.

In compiler terminology, a variable is said to be "live" if its value will be used in the future. When a variable is live, its value must be preserved and therefore the register containing it cannot be overwritten. The number of variables that are live at a given point in the execution of a program determines how many registers are free for other uses. The same holds true for the temporary registers in a GPU. The number of temporary registers needed can also change when IR code is built because the liveness of a variable may change due to common subexpression elimination.

Whenever there is a common subexpression, a temporary register will be used to hold the result between the first time it is used and the last time it is used, this affects the number of temporaries used and therefore must be considered by the Driver Layer when providing the upper bounds. Algorithm 4.7 shows the improved partitioning algorithm.

```
1  def Partition(block):
2      # get upper bound resource usage for block
3      resources = block.GetUpperBound()
4      # only process non-leaves
5      if block.type != EXPRESSION or checkCache(block):
6          return resources
7      # recursively process all subexpressions
8      # and accumulate upper bound
9      for child in block.array.children:
10         child_block = BuildView(block, child)
11         resources += Partition(child_block)
12     # Has the upper bound been exceeded?
13     if !CheckUpperBound(resources):
14         # get exact resources by building code
15         resources = BuidIrCode()
16         if resources != FAILURE:
17             return resources
18         else:
19             # couldn't build code, need to collapse
20             # subexpressions
21             for child in block.array.children:
22                 # collapse child...
23                 child_block = BuildView(block, child)
24                 Evaluate(child_block)
25                 # ...and try building the code again
26                 resources = BuildIrCode(block):
27                 if resources != FAILURE:
28                     return resources
29             # collapsed all children and still can't
30             # build code
31             return FAILURE
32     # return upper bound for this block
33     return resources
```

Figure 4.7: Improved Partitioning Algorithm. Like the simple partitioning algorithm, this algorithm recursively partitions an expression into manageable-sized subexpressions. The improved algorithm, however, eliminates many of the calls to `BuildIrCode`, which improves the performance of partitioning.

4.9 NumPy Compatibility

One of the goals of GpuPy is to be able to execute existing NumPy scripts with minimal modification. In order to accomplish this, GpuPy knows how to fall back to the NumPy versions of operations when they are not implemented by the Driver Layer. This is accomplished by intercepting all module member methods provided by NumPy and dispatching them appropriately. When a method is called, GpuPy's `dispatch()` method is what actually gets called. The two things considered by the `dispatch()` method are the types of the arguments and whether or not NumPy and GpuPy implement the intercepted method. When GpuPy implements the method for that type, it is invoked. Otherwise, the NumPy version is invoked. When the NumPy version is used, any arguments that are `GpuArray` objects are flushed and the underlying NDarray objects are used in place of the original arguments.

4.10 OpenGL/Cg Driver

GpuPy's primary driver uses OpenGL and NVIDIA's Cg library [28, 1] to send shader information to the GPU. It implements the most commonly used NumPy operations. Most of the code in the Cg driver is used for generating Cg code from IR code and for evaluating and retrieving the results. Figure 4.8 shows Cg code generated from the IR in Figure 4.5.

Aside from the actual code generation, the most important part of the Cg driver is proving estimates of the resource usage of a `GpuArray` object. It does this by assuming that all expressions are unique and that each intermediate result is stored in a new temporary register.

`check_ir()` causes the Cg driver's register allocation algorithm to run and determine

```
1  float main(float2 p : TEXCOORD0,
2             uniform samplerRECT a0,
3             uniform samplerRECT a1) : COLOR
4  {
5      float t0, t1;
6      float c0 = 3.5;
7      t0 = texRECT(a0, p);
8      t0 = cos(t0);
9      t1 = texRECT(a1, p);
10     t1 = t1 + c0;
11     t0 = t0 * t1;
12     return t0;
13 }
```

Figure 4.8: Cg code generated by the OpenGL/Cg Driver from the intermediate representation in Figure 4.5.

the precise number of resources required by the shader. The resources that the Cg driver considers are the resources specified in the `ARB_fragment_program` documentation [5]. Specifically, the resources considered by Cg are:

- total instructions: The total number of instructions needed by the shader.
- ALU instructions: The number of ALU instructions needed by the shader. ALU instructions perform arithmetic operations such as addition and subtraction.
- texture instructions: Texture instructions are used to read values from a texture.
- texture indirections: Texture indirections occur when a value read from a texture is used as an argument to a subsequent read from a texture. GpuPy does not currently use texture indirections, but will in the future.

- temporary registers: The number of registers needed to execute the shader. This depends on the structure of the program and the number of common subexpressions.
- parameters: Parameters are used to pass constant values to shaders. GpuPy uses parameters to represent constants that appear in expression trees and to pass extra required information to the shader.
- attributes: Attributes are things like texture coordinates and other OpenGL state information.

4.11 Software Driver

GpuPy also contains a software driver that doesn't use a GPU at all. It implements some basic operations but is mostly used for testing. Most of the software driver's functions perform no work and return default values. It allows only a single operation to be performed per evaluation. This is useful for testing the Core Layer's algorithms because advanced behaviors such as partitioning and block eviction can be triggered using small, easy to understand programs. Unlike the Cg driver, the software driver has no external dependencies, and therefore allows GpuPy's basic functionality to be tested on systems where no GPU is present.

CHAPTER FIVE

EVALUATION

Since GpuPy attempts to accelerate NumPy, it is important that it produces results that are as close as possible to NumPy and does so more quickly than NumPy. GpuPy is evaluated for accuracy and performance using a collection of Python programs that attempt to exploit the various features of GpuPy. These programs are used to evaluate how well GpuPy performs and how accurate the results produced by GpuPy are.

Performance is measured using execution time on a quiescent system and accuracy is measured using relative error. Relative error is a widely used metric for the accuracy of a floating point value [6, 21, 10]. It is calculated by dividing the absolute difference between the value in question and the correct answer by the correct answer. For our purposes, we consider the value produced by NumPy to be the correct answer. The relative errors for each element in the array are calculated and their mean, standard deviation, and maximum departure are used to evaluate the overall behavior of GpuPy.

5.1 Basic Functionality

Basic functionality is tested using a program that performs a large number of simple operations. For each basic operation tested, random arrays are generated for each operand and the operation is performed using both NumPy and GpuPy. The basic test produces one set of results for each operation it tests. Table 5.1 shows the results of running the basic test program found in Appendix A.2 using an NVIDIA GeForce 7800 GT GPU. This table shows that for most operations, the GPU produces results that are close to those produced

by NumPy.

Machine epsilon is a measurement of the precision of a machine's floating-point implementations. When a real number is represented using a floating-point number, the relative error of this representation is bounded by the machine epsilon [10]. Machine epsilon can be determined mathematically or experimentally and is about 1.19×10^{-7} for IEEE single-precision values. Table 5.1 shows that for most operations, GpuPy's relative error is indeed bounded by the machine epsilon and can therefore be considered accurate. The lack of accuracy for some of the operations is likely indicative of the GPU using approximations for these operations.

5.2 Distance Map

The distance map test generates a grayscale image produced by calculating the distances between randomly generated points. More formally, suppose there is a set S of randomly chosen points and an $M \times M$ grid of pixels. For each grid point p, calculate

$$d(p) = \min_{q \in S} |p - q|$$

and linearly map the result so that the minimum value of $d()$ corresponds to 0 and the maximum value of $d()$ corresponds to 1. The distance map test produces two files, the first is a grayscale image created by using the linearly mapped values of $d(p)$ for each pixel p. The second file stores the raw linearly mapped values of $d()$. Figure 5.1 shows an image produced by the distance map test.

Three versions of the distance map test exist: NumPy, GpuPy, and C. A helper script

runs the three versions and compares their performance and results. Each version is run

(NumPy) Function	Mean	Standard Deviation	Maximum Departure
absolute	0	0	0
add	31	44	88
arccos	28981	13771	28981
arcsin	507330	5255030	129580895
arctan	2703	1382	2703
arctan2	2048	1470	2433
cos	103	70	260
cosh	72	62	184
divide	24	38	106
equal	0	0	0
exp	0	0	0
fabs	0	0	0
fmod	691	6318	183620
log	205	757	11485
log10	198	755	11490
maximum	0	0	0
multiply	13	33	106
pow	57	59	414
power	57	59	414
sin	604	3912	85211
sinh	293	1255	27246
sqrt	32	40	119
subtract	9	28	109
tan	652	3906	85277
tanh	361	2133	37517

Table 5.1: Relative Errors in Basic Functionality Results. All values are scaled by 10^9. These were collected using a GeForce7800 GT. The source code for this test can be found in Appendix A.2

several times, with increasing sizes of S. The image files produced can be visually compared by a human and the raw data files produced can be evaluated using the mean, standard deviation, and maximum departure of the relative error.

Figure 5.2 compares the performance of the three versions of the distance map test on two different systems. The running time of each version of the program is plotted in relation to the size of S. Table 5.2 provides details about each item plotted in figure 5.2. The plot shows that for this test, running times increase more slowly in relation to the size

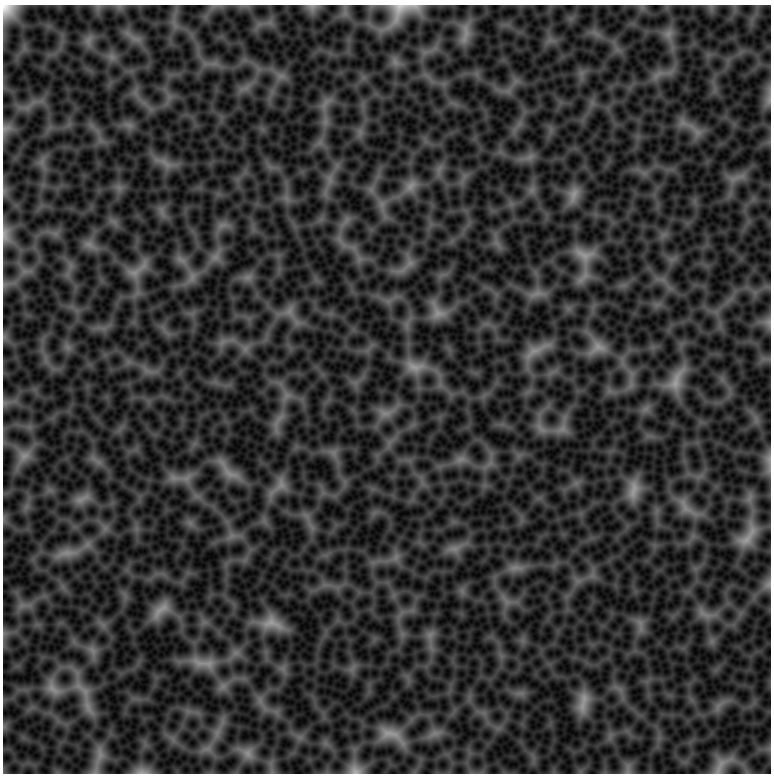

Figure 5.1: Image Produced by the Distance Map Test. This image was generated using a set S of randomly chosen points on a 512 x 512 grid. Each pixel's intensity is set according to the distance from it to the nearest point in S. For this image, $|S| = 5000$.

of S for GpuPy than for NumPy or C. When the size of S is large, GpuPy outperforms NumPy and C by around a factor of 10.

The raw floating-point values produced by the distance map test are evaluated for accuracy using the same techniques as the basic functionality test. Table 5.3 shows the results of these tests. Table 5.3 shows that, in general, as the size of S increases, GpuPy's relative error increases. This is expected and is caused by compounding of the per-operation relative errors shown in Table 5.1.

symbol	description
C1	C version on Intel Pentium IV 2.00GHz
N1	NumPy version on Intel Pentium IV 2.00GHz
G1	GpuPy version on NVIDIA GeForce FX 5500 (NV34)
C2	C version on AMD Athlon 2.00GHz
N2	NumPy version on AMD Athlon 2.00GHz
G2	GpuPy version on NVIDIA GeForce 7800 GT (NV44)

Table 5.2: Key to Figure 5.2

$\|S\|$	mean	standard deviation	max departure
10	6000	6000	40000
20	9000	7000	36000
50	9000	8000	37000
100	12000	10000	49000
200	21000	16000	71000

Table 5.3: Relative Errors in Raw Distance Map Test Results. All values are scaled by 10^9. Note that these errors are not large enough to make a difference in the final image, since each floating point value is mapped to an 8-bit value.

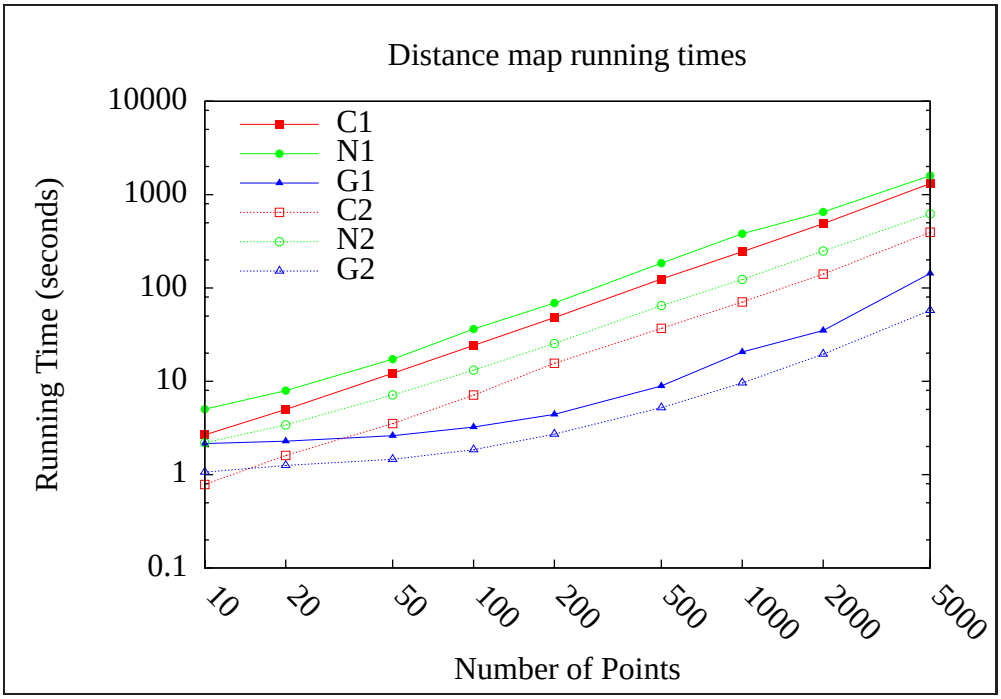

Figure 5.2: Distance Map Performance Comparison. This plot compares the performance of GpuPy, NumPy and C implementations of the distance map test. For large numbers of points, GpuPy outperforms C and NumPy by about a factor of 10.

CHAPTER SIX

CONCLUSIONS

GpuPy shows a significant performance improvement over NumPy and C versions of the test application. GpuPy outperforms NumPy and C by around a factor of 10 for some tests. The lazy evaluation and tree partitioning algorithms work well enough to allow a GPU to be used efficiently without requiring any direct programming of the GPU. GpuPy also allows some existing NumPy programs to be run using a GPU without making any changes to the original program aside from importing the GpuPy module. This provides an easy way to use a GPU for general purpose calculations. GpuPy's design makes it easy to iteratively add support for new GPUs or other parallel computing architectures and provides almost seamless integration with NumPy.

CHAPTER SEVEN

FUTURE WORK

There are many options for future work on GpuPy. Some possibilities are listed and discussed below.

- Better NumPy support: The eventual goal of GpuPy is to be a drop-in replacement for NumPy. There are a large number of features that need to be added before this can happen. Reductions, sorting, mutable arrays, and advanced slicing are all examples of features the current implementation lacks. Support for NumPy extensions like Linear Algebra and MLab (MATLAB^{TM}compatibility) may also benefit from GPU acceleration.

- Improved mapping to GPU: Using fixed-size blocks is less than ideal. It requires that all arrays be rounded up to the next multiple of the block size, even if the array is small and the block size is large. Removing this limitation would allow GpuPy to scale better, especially for arrays whose size is less than one block. Going further than this, having a more advanced block scheme could allow features such as broadcasting and striding to be moved entirely onto the GPU, which would improve performance.

- Vector data types: GpuPy currently allows elements of an array to be only scalars, but GPUs also have native representations of 2-, 3-, and 4-vectors of floating point values. Certain algorithms, such as ones dealing with image-processing, are more easily described using vectors rather than scalars. The sphere rendering test, for

example, would benefit from vector data types.

- Shader caching: Performance can be improved by implementing a shader-caching algorithm such as the one described in Accelerator [22]. Each view could have associated with it a shader that evaluates to it. This would be especially useful when all of the blocks of an expression are being evaluated, since different blocks from the same expression have identical shader code, but different blocks. When a cache hit occurred, the cost of partitioning and building the code would be eliminated.

- Python's `compiler` package: Using Python's `compiler` package to build expression trees may have advantages over the interpretive technique. It would allow GpuPy to have more complete information and it would not have to guess about things like which array elements would be requested. This would allow GpuPy to more efficiently perform calculations since unneeded elements would never be evaluated. GPUs can also perform conditional branching, which could be taken advantage of using the `compiler` package.

- Multiple render buffers: New GPUs have the ability to write to multiple render buffers from a single shader. Taking advantage of this feature could allow GpuPy to evaluate a tree more efficiently because it would not be limited to a single sub-expression. It could work on up to N sub-expressions at a time, where N is the number of render buffers allowed by the underlying hardware. Currently, shaders that do not exhaust single-shader resources may need to be run because the entire sub-expression does not fit. Allowing multiple write buffers would remove the requirement that an entire subexpression be evaluated at once and allow multiple partial

subexpressions to be evaluated together.

- More drivers: GpuPy drivers should be written to take advantage of the different alternatives to Cg. Examples are ATI's DPVM API [19] and NVIDIA's CUDA [2]. Drivers could also potentially be written that use something other than a GPU to perform calculations. An Ethernet-connected GPU cluster was described in [9]. The GPU cluster outperformed CPU-based solutions for a flow simulation.

BIBLIOGRAPHY

[1] Cg language specification. http://developer.nvidia.com/object/cg_toolkit.html.

[2] Nvidia cuda compute unified device architecture: Programming guide. http://developer.nvidia.com/object/cuda.html.

[3] Alfred V. Aho and Jeffrey D. Ullman. *The theory of parsing, translation, and compiling*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1972.

[4] Kurt Akeley, Jason Allen, Bob Beretta, Pat Brown, Matt Craighead, Alex Eddy, Cass Everitt, Mark Galvan, Michael Gold, Evan Hart, Jeff Juliano, Mark Kilgard, Dale Kirkland, Jon Leech, Bill Licea-Kane, Barthold Lichtenbelt, Kent Lin, Rob Mace, Teri Morrison, Chris Niederauer, Brian Paul, Paul Puey, Ian Romanick, John Rosasco, R. Jason Sams, Jeremy Sandmel, Mark Segal, Avinash Seetharamaiah, Folker Schamel, Daniel Vogel, Eric Werness, and Cliff Woolley. Ext_framebuffer_object. http://www.opengl.org/registry/specs/EXT/framebuffer_object.txt.

[5] Bob Beretta, Pat Brown, Matt Craighead, Cass Everitt, Evan Hartand, Jon Leech, Bill Licea-Kane, Bimal Poddar, Jeremy Sandmel, Jon Paul Schelter, Avinash Seetharamaiah, and Nick Triantos. Arb_fragment_program. http://oss.sgi.com/projects/ogl-sample/registry/ARB/fragment_program.txt, September 2002.

[6] Richard L. Burden and Douglas J. Faires. *Numerical Analysis, 7th ed.* Brooks Cole, January 2005.

[7] Silviu Ciricescu, Ray Essick, Brian Lucas, Phil May, Kent Moat, Jim Norris, Michael Schuette, and Ali Saidi. The reconfigurable streaming vector processor (rsvptm). In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 141, Washington, DC, USA, 2003. IEEE Computer Society.

[8] William J. Dally, Ujval J. Kapasi, Brucek Khailany, Jung Ho Ahn, and Abhishek Das. Stream processors: Progammability and efficiency. *Queue*, 2(1):52–62, 2004.

[9] Zhe Fan, Feng Qiu, Arie Kaufman, and Suzanne Yoakum-Stover. GPU cluster for high performance computing. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 47, Washington, DC, USA, 2004. IEEE Computer Society.

[10] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, 1991.

[11] General-purpose computation using graphics hardware. `http://www.gpgpu.org`.

[12] Jayanth Gummaraju and Mendel Rosenblum. Stream programming on general-purpose processors. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 343–354, Washington, DC, USA, 2005. IEEE Computer Society.

[13] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: being lazy with class. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–1–12–55, New York, NY, USA, 2007. ACM Press.

[14] Draft standard for floating-point arithmetic P754. `http://grouper.ieee.org/groups/754`, October 2006.

[15] David Luebke, Mark Harris, Jens Krüger, Tim Purcell, Naga Govindaraju, Ian Buck, Cliff Woolley, and Aaron Lefohn. GPGPU: general purpose computation on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Course Notes*, New York, NY, USA, 2004. ACM Press.

[16] Travis E. Oliphant. Guide to NumPy. `http://www.numpy.org`, December 2006.

[17] Travis E. Oliphant. Python for scientific computing. *Computing in Science and Engg.*, 9(3):10–20, 2007.

[18] John D. Owens, William J. Dally, Ujval J. Kapasi, Scott Rixner, Peter Mattson, and Ben Mowery. Polygon rendering on a stream architecture. In *HWWS '00: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 23–32, New York, NY, USA, 2000. ACM Press.

[19] Mark Peercy, Mark Segal, and Derek Gerstmann. A performance-oriented data parallel virtual machine for gpus. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Sketches*, page 184, New York, NY, USA, 2006. ACM Press.

[20] Mark Segal and Kurt Akeley. The opengl graphics system: A specification. `http://opengl.org/documentation/specs/`, December 2006.

[21] Andrew S. Tanenbaum. *Structured Computer Organization, 4th ed.*, chapter 6. Prentice Hall, 4 edition, 1999.

[22] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: using data parallelism to program GPUs for general-purpose uses. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 325–335, New York, NY, USA, 2006. ACM Press.

[23] D Turner. An overview of miranda. *SIGPLAN Not.*, 21(12):158–166, 1986.

[24] Guido van Rossum. Python programming language. `http://www.python.org`, 1990 - 2007.

[25] Guido van Rossum. Extending and embedding the python interpreter. `http://docs.python.org/ext/ext.html`, September 2006.

[26] Guido van Rossum. Python 2.5 documentation. `http://docs.python.org`, September 2006.

[27] Suresh Venkatasubramanian. The graphics card as a stream computer. In *SIGMOD-DIMACS Workshop on Management and Processing of Data Streams*, 2003.

[28] Mason Woo, Davis, and Mary Beth Sheridan. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

APPENDIX

APPENDIX ONE
SOURCE CODE

A.1 Shaded Sphere Source Code

(by Robert R. Lewis)

```
import sys
from PIL import Image

from numpy import *

# parameter settings
(w,h)        = (512, 512)      # image dimensions
r            = 0.4 * min(w, h) # sphere radius
(vx, vy, vz) = (w/2, h/2,   w) # viewer position
(lx, ly, lz) = (-1,   1,   1)  # light direction
bg           = (0.0, 0.0, 0.5) # background color
ka           = (0.1, 0.2, 0.3) # ambient sphere color
kd           = (0.2, 0.5, 0.6) # diffuse sphere color
(cx, cy, cz) = (w/2, h/2, 0)   # sphere position

# Start with pixel coordinates.
x = fromfunction(lambda x, y: x, (w, h), dtype=float32)
y = fromfunction(lambda x, y: y, (w, h), dtype=float32)
z = 0 # on the image plane

(dx, dy, dz) = (x - vx, y - vy, z - vz) # viewing direction

# Solve the quadratic equation for each pixel
# (note: no explicit iteration)
a = dx**2 + dy**2 + dz**2
b = 2 * dx * (vx - cx) + 2 * dy * (vy - cy) + 2 * dz * (vz - cz)
c = cx**2 + cy**2 + cz**2 + vx**2 + vy**2 + vz**2 \
    - 2 * (cx*vx + cy*vy + cz*vz) - r**2
disc = b*b - 4*a*c # discriminant

t = (-b - sqrt(disc)) / (2 * a)  # the ray parameter
```

```
32
33 # intersection
34 (ix, iy, iz) = (vx + t * dx, vy + t * dy, vz + t * dz)
35
36 # normal to sphere at intersection (guaranteed of unit length)
37 (nx, ny, nz) = ((ix - cx) / r, (iy - cy) / r, (iz - cz) / r)
38
39 # dot product of sphere normal and light normal
40 # (for diffuse shading)
41 nDotL = nx*lx + ny*ly + nz*lz
42
43 # Where the ray hits the sphere, set to the shaded diffuse
44 # color, otherwise set to the background color.
45 channels = [ 255 *
46              where(disc > 0,
47              where(nDotL > 0, ka_i + nDotL * kd_i, ka_i),
48              bg_i) for (bg_i, ka_i, kd_i) in zip(bg, ka, kd) ]
49
50 # Convert the array to an image and write it as a PNG file.
51 imgs = [ Image.frombuffer("F", (w, h), c,
52                           "raw", "F", 0, 1).convert("L")
53          for c in channels ]
54 Image.merge("RGB", imgs).save("shaded_sphere.png")
```

A.2 Basic Functionality Test

```
1  from numpy import *
2  from gpupy import *
3
4  # calculate relative error for each array element
5  # and then calculate the mean, standard deviation,
6  # and maximum departure
7  def check_result(value, actual):
8      ab = re = 0.0
9      count = len(value)
10
11     # calculate the mean
12     re_sum = 0.0
13     for i in range(0, count):
14         if actual[i] != 0:
15             re = abs((value[i] - actual[i]) / actual[i])
16         else:
17             re = abs((value[i] - actual[i]))
18         re_sum += re
19     re_mean = re_sum / count
20
21     # calculate stddev and max departure
22     max_dev = 0.0
23     stddev = 0.0
24     for i in range(0, count):
25         if actual[i] != 0:
26             re = abs((value[i] - actual[i]) / actual[i])
27         else:
28             re = abs((value[i] - actual[i]))
29
30         dev = fabs(re - re_mean)
31         if dev > max_dev:
32             max_dev = dev
33         stddev += dev * dev
34     stddev = sqrt(stddev / count)
35
36     # return a tuple containing the results
```

```
    return (re_mean, stddev, max_dev)

# test the specified unary function
# using provided data
def test_unary(fcn, n, data):
    a = array(data, dtype=float32)
    ga = array(data, dtype=gpufloat32)

    x = fcn(a)
    xa = fcn(ga)

    return check_result(xa, x)

# test the specified binary function
# using provided data
def test_binary(fcn, n, data1, data2):
    a = array(data1, dtype=float32)
    b = array(data2, dtype=float32)

    ga = array(data1, dtype=gpufloat32)
    gb = array(data2, dtype=gpufloat32)

    x = fcn(a, b)
    xa = fcn(ga, gb)

    return check_result(xa, x)

# unary operations to test
unary_ops = (absolute, arccos, arcsin, arctan,
             cos, cosh, exp, fabs, log, log10,
             sin, sinh, sqrt, tan, tanh)

# binary operations to test
binary_ops = (add, arctan2, divide, fmod, maximum,
              multiply, pow, power, subtract, equal)

# size of arrays to test
size = 1024
```

```
75
76 # factor to scale random data by
77 # changing this value leaves the results mostly
78 # unchanged, which is why relative error is
79 # used. This is set to 1 because some of the
80 # functions have a limited domain (arcsin, etc.)
81 scale = 1
82
83 # place for storing results
84 results = {}
85
86 # get some random data
87 data1 = random.rand(size) * scale
88 data2 = random.rand(size) * scale
89
90 # test everything with the random data
91 for op in unary_ops:
92     results[op.__name__] = test_unary(op, size, data1)
93
94 for op in binary_ops:
95     results[op.__name__] = test_binary(op, size, data1, data2)
96
97 # sort results by operation
98 keys = sort(results.keys())
99
100 # output results
101 # (formatted for latex)
102 maxlen = 0
103 max_val = [0, 0]
104
105 for k in keys:
106     if len(k) > maxlen:
107         maxlen = len(k)
108     for i in range(0, len(max_val)):
109         if len(str(results[k][i])) > max_val[i]:
110             max_val[i] = len(str(results[k][i]))
111
112 maxwidth = 0
```

```
for k in keys:
    if len(str(k)) > maxwidth:
        maxwidth = len(str(k))

# scale results to something more reasonable
scale_factor = 1.0e9

for k in keys:
    print "%-25s & %-10d & %-10d & %-10d \\\\\n\\hline" % \
          ('\\texttt{' + k.replace('_', '\\_') + '}',
           round(results[k][0] * scale_factor),
           round(results[k][1] * scale_factor),
           round(results[k][2] * scale_factor))
```

A.3 Distance Map Program

(by Robert R. Lewis)

```
from PIL import Image
import sys
from getopt import getopt
from random_values import *

nPt = 8 # default
expo = 2 # default distance exponent (Euclidean)
nGrid = 512 # image size (in each dimension)
useGpupy = False

imgOutput = False
rawOutput = False
img_name = "distmap.png"
raw_name = "distmap.dat"

(optsvals, args) = getopt(sys.argv[1:], 'e:go:r:p:s:w:')
for (opt, val) in optsvals:
    if opt == '-e':
        expo = float(val)
    elif opt == '-g':
        useGpupy = True
    elif opt == '-o':
        img_name = str(val)
        imgOutput = True
    elif opt == '-r':
        raw_name = str(val)
        rawOutput = True
    elif opt == '-p':
        nPt = int(val)
    elif opt == '-s':
        myrandseed(int(val));
    elif opt == '-w':
        nGrid = int(val)

if useGpupy:
```

```
    from gpupy import *
    dtype = gpufloat32
else:
    from numpy import *
    dtype = float32

# generate nPt random points within the grid
pt = []
for k in range(nPt):
    x = nGrid * myrandf()
    y = nGrid * myrandf()
    pt.append((x, y))

def dist((x, y), (x0, y0)):

    """Returns the Euclidean distance between two (2D) points."""

    return ((x-x0)**expo + (y-y0)**expo)**(1.0/expo)

def dist_array((x0, y0)):

    """Returns an array whose elements are the distance (in units of
    rows and columns) to a given point (x0, y0)."""

    f = lambda x, y, x0=x0, y0=y0: dist((x, y), (x0, y0))
    return fromfunction(f, (nGrid, nGrid), dtype=dtype)

# distMin[i,j] is the distance from pixel (i,j) to the closest 'pt'.
distMin = dist_array(pt[0])
for i in range(1, nPt):
    distMin = minimum(dist_array(pt[i]), distMin)

pxlMin = 0
pxlMax = max(distMin.flat)

# scale and offset distances to lie between 0 and 1
distMinScaled = (distMin - pxlMin) / (pxlMax - pxlMin)

```

```
# (gray) pixel values lie between 0 and 255
pxls = 255 * distMinScaled

if imgOutput:
    # bug: GpuPy doesn't correctly handle astype
    if useGpupy:
        imgdata = pxls
    else:
        imgdata = pxls.astype(float32)

    img = Image.frombuffer("F", (nGrid, nGrid), imgdata,
                           "raw", "F", 0, 1).convert("L")
    img.save(img_name)

if rawOutput:
    f = open(raw_name, "wb")
    if useGpupy:
        f.write(buffer(pxls))
    else:
        f.write(buffer(pxls.astype(float32)))
    f.close()
```

A.4 Simplified OpenGL/Cg Driver Excerpt

```
1  int
2  cg_block_read(block_t *blk, float *buf)
3  {
4      block_descr_t *descr;
5      int type;
6      int count, rows, elements;
7      int i;
8
9      /* Attach requested texture to a framebuffer object. */
10     glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
11                               GL_COLOR_ATTACHMENT0_EXT,
12                               GL_TEXTURE_RECTANGLE_ARB,
13                               blk->texid,
14                               0);
15
16     /* Tell GL we want to read this attachment. */
17     glReadBuffer(GL_COLOR_ATTACHMENT0_EXT);
18
19     /* How big is the array this block comes from? */
20     count = 1;
21     for(i = blk->nd - 1; i >= 0; i--)
22         count *= blk->dimensions[i];
23
24     /* If it is at least the size of a block, then we
25        need to read an entire block. */
26     if (count >= GPUPY_BLOCK_SIZE)
27         count = GPUPY_BLOCK_SIZE;
28
29     /* How many complete scanline? How many remaining
30        elements? */
31     rows = count / cg_typemap[type].block_w;
32     elements = count % cg_typemap[type].block_w;
33
34     if (rows){
35         /* Read rows. */
36         glReadPixels(0, 0,
```

```
                    BLOCK_WIDTH, rows,
                    GL_RED, GL_FLOAT, &buf[0]);
    }

    if (elements){
        /* Read remaining elements. */
        glReadPixels(0, rows,
                    elements, 1,
                    GL_RED, GL_FLOAT,
                    &buf[rows * BLOCK_WIDTH]);
    }

    return 0;
}
```

A.5 Textured Quadrilateral Code

```
1  void
2  glDrawQuad(int x, int y, int w, int h)
3  {
4      glBegin(GL_QUADS);
5          glTexCoord2i(x, y);
6          glVertex2i(x, y);
7
8          glTexCoord2i(x, y + h);
9          glVertex2i(x, y + h);
10
11         glTexCoord2i(x + w, y + h);
12         glVertex2i(x + w, y + h);
13
14         glTexCoord2i(x + w, y);
15         glVertex2i(x + w, y);
16     glEnd();
17 }
```