GpuPy: Transparently and Efficiently Using a GPU for Numerical Computation in Python

BENJAMIN EITZEN and ROBERT R. LEWIS
School of EECS
Washington State University

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 a GPU implementation of *NumPy*, an existing numerical API for the Python programming language.

Categories and Subject Descriptors: C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors); D.3.4 [**Programming Languages**]: Processors; G.4.0 [**Mathematical Software**]: General

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Python, NumPy, array processing, graphics processing unit

1. 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 [Luebke et al. 2004]. The goal of *GpuPy* is to provide a Python interface for taking advantage of the strengths of a GPU.

GpuPy is an extension to the Python programming language which provides an interface modeled after the popular *NumPy* Python extension [van Rossum 2008b]. 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.

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

Contact Author: Robert R. Lewis; School of Electrical Engineering and Computer Science; Washington State University; 2710 University Dr.; Richland, WA 99354. bobl@tricity.wsu.edu

involving *GpuPy*.

2. BACKGROUND

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

2.1 GPUs

Virtually all modern desktop and laptop computers contain a GPU, either integrated into the motherboard or on a separate graphics card. A GPU is a parallel processor designed to render images. Building on graphics accelerator technology, 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 are thus capable of performing increasingly more floating point operations per second than traditional CPUs.

2.1.1 *The OpenGL Rendering Pipeline.* GPUs are designed to render complex 3D geometry in real time. In general, input is passed to a GPU as a collection of vertices, matrices, texture coordinates, textures, and other state variables (lighting parameters, etc.). A GPU processes this input and renders the image into a block of memory called a "frame buffer" which is then displayed on some sort of output device, usually a monitor. This sequence of steps is called the *rendering pipeline*.

For example, the sequence of actions performed by the rendering pipeline to render a quadrilateral would be:

—A program provides the four vertices that make up the corners of the quadrilateral. At this point, coordinates are usually defined in a floating point *object coordinate system*, which disregards both the position and orientation of the object in the overall scene and the point of view of the observer. Each vertex may have associated with it a variety of attributes, such as color, a surface normal, and one or more texture coordinates.

—In the *per-vertex operations and primitive assembly* stage, 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 triangles) and any vertices that fall outisde the view volume are discarded, or "clipped".

—The next stage is *rasterization*, which generates "fragments," which are like pixels, but may contain information besides color for use in the final phase such as transparency, a depth value and texture coordinate(s). These values are usually calculated by interpolating the corresponding values from the vertices across the face of the primitive.

—At last, the resulting fragments are passed to the *per-fragment operations* stage, which performs final processing on the fragments before outputting them to the

frame buffer. 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. This ensures that pixels nearer to the observer conceal pixels further away.

For a more in-depth description of the OpenGL rendering pipeline, see [Segal and Akeley 2006].

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. Each texel typically contains three components describing a red-green-blue (RGB) color, possibly with a fourth opacity or "alpha" (A) component. In the past, the components were represented by 8-bit unsigned integers, but GPUs can now represent components as 32-bit floating point values (which consequently makes them of interest for numerical computation).

Texture maps are used by OpenGL to map an image onto the surface of a primitive. If texturing is enabled, the *rasterization* stage calculates the color or lighting of each fragment using values from one or more texture maps. Since texture coordinates are bound to vertices, texture coordinates for a particular pixel may be (linearly) interpolated from neighboring vertices.

2.1.3 *Programming the Pipeline.* Traditionally, the *per-vertex operations and primitive assembly* and *per-fragment operations* stages performed fixed functions. In modern GPUs, however, these stages are fully programmable using short programs called "shaders".

Like conventional CPUs, shader operation is controlled by a stored program in an underlying machine language which is not generally programmable by humans. From the beginning, individual GPU manufacturers have therefore provided GPU-specific assembler languages with mnemonics for the machine opcodes, macros, and other typical assembler features.

More recently, however, to make GPU programming accessible to a wider audience, higher-level shader languages have been developed which resemble conventional programming languages (typically C), the most popular of which are NVIDIA's Cg [Fernando and Kilgard 2003], Microsoft's High-Level Shading Language (HLSL) [Microsoft 2008], and the OpenGL Shading Language (GLSL) [Rost 2006]. These also have a greater degree of portability between GPU models than the assemblers and provide other conveniences.

Most current pipelines contain three stages that are programmable: fragment shading, vertex shading, and geometry shading.

Vertex shaders run during the early stages of the pipeline. They take a single vertex as input and produce a single vertex as output (there is no way to create or destroy a vertex with a vertex shader). They have access to global parameters such as light and material properties. A vertex shader is executed once for each input vertex. Vertex shaders are independent of each other and can therefore be performed in parallel. A relatively new development, geometry shaders run after vertex shaders but before fragment shaders. They acccept multiple vertices as input and are allowed to create new vertices. The vertices output by a geometry shader

```
void glDrawQuad(int x, int y, int w, int h)
{
    glBegin(GL_QUADS);
    glTexCoord2i(x, y);          glVertex2i(x, y);
    glTexCoord2i(x, y + h);      glVertex2i(x, y + h);
    glTexCoord2i(x + w, y + h); glVertex2i(x + w, y + h);
    glTexCoord2i(x + w, y);      glVertex2i(x + w, y);
    glEnd();
}
```

Fig. 1. OpenGL Code to Draw a Quadrilateral. This code triggers execution of both vertex and fragment shaders in a typical array computation.

continue through the rest of the pipeline as though they were provided explicitly by the controlling program. Neither vertex shaders nor geometry shaders are 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 – including interpolated attributes – but are also able to read and compute with values from texture memory. When a fragment shader is enabled, it is executed once for each input fragment. Like vertices, the fact that fragments are computed independently allows them to be processed in parallel.

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 [Ciricescu et al. 2003]. 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 [Dally et al. 2004] and [Gummaraju and Rosenblum 2005].

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. General Purpose GPU (GPGPU) [GPGPU 2008] is an initiative to study the use of GPUs to perform general purpose computations instead of specialized graphics algorithms. The two most important GPU features critical to the success of GPGPU are a programmable pipeline and floating point textures.

The rendering pipeline described above can be exploited to act as stream processor [Venkatasubramanian 2003; Owens et al. 2000]. 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. Instead of a frame buffer, the output of the fragment shader is then written into texture memory using the OpenGL Framebuffer Object Extension[Akeley et al. 2008].

Texture coordinates must be chosen that cause each fragment's interpolated texture coordinates to reference the correct texel. The code in Figure 1 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.

There are a number of limitations that must be observed when writing a fragment shader. The destination 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. A texture also may not be written to if it 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.

While some higher-end GPUs, such as NVIDIA's Quadro product line, support IEEE single-precision values, others do not. Because of this, the results of GPU and CPU algorithm implementations may differ. For many applications this is perfectly acceptable but for others it may be problematic.

GPUs also support a 16-bit "half-precision" floating point format, which supports a sign bit, a ten bit characteristic, and a five bit (biased) exponent. If an application can tolerate the reduction in precision (e.g. anything intended solely human visual consumption), performance and memory usage may be improved by using this. Although the current IEEE floating point standard does not include a 16-bit type, a draft revision of the standard [IEEE P754 2006] does.

2.4 Python

Python [van Rossum 2008b] 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 is fully featured. Python is very portable and runs on a variety of platforms. In this section, we'll cover the features of Python that are most relevant to *GpuPy*.

2.4.1 *Extending Python.* An important feature of Python is that it was carefully designed to be easily extensible [van Rossum 2008a]. This is accomplished by allowing modules written in C or C++ to function equivalently to modules written in Python, including the addition of new data, functions, and classes. This is done with C `struct`s whose members include callbacks (function pointers) and auxiliary data.

The callbacks are organized into groups of related operations called *protocols*, and a new class may implement whichever protocols are appropriate. Unneeded or irrelevant callback functions may be left unimplemented. The current version of

Python defines these protocols:

—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 by indexing into them using non-negative integers.
—The *mapping* protocol is similar to the sequence protocol, but allows any Python object to be used as an index, rather than just nonnegative 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. It is widely used to share data between extension modules.

2.4.2 *Slicing.* Python has a 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 `step`. 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 `step` 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 `step` elements between selections. The three slice arguments can be thought of as the three parameters of a `for` C loop that produce the desired indices:

```
for (i = start; i < stop; i += step)
    /* access element i of the array */
```

2.5 NumPy

NumPy [Oliphant 2007] is a Python extension module written by Travis Oliphant and others. It is the successor to *Numeric* [Hugunin 1995] and incorporates features also developed by *Numarray* [Greenfield et al. 2003], two previous numerical Python extensions. In its C internals, *NumPy* provides several Python classes (in C form), the most important of which is `NDarray`, which is used to implement N-dimensional arrays.

NumPy allows mathematical operations to be performed on arrays as though they were scalars. When an arithmetic operation is performed on one or more `NDarray`s, the operation is applied to corresponding elements (conceptually) in parallel. The result is a new `NDarray` object whose dimensions are determined by the shapes of the operands.

Much of this functionality is incorporated in *NumPy*'s "universal functions" or *ufunc*s, which may resemble fundamental mathematical functions such as *sin()* or *abs()*, but which work for both conventional Python scalars and `NDarray`s. They also allow the user to pass additional arguments to control, for instance, error handling.

`NDarray` extends Python's concept of slicing to multiple dimensions, but with one key difference: Unlike in Python, a slice of an `NDarray` object always refers to

the same data as the original object. For instance, executing `b = a[::2]`, means that `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` with a data pointer, a shape tuple, and a stride tuple. The *shape* of an array is its size along each dimension and the *strides* of an array specify the number of bytes between logically consecutive array elements (note that a `stride` is like a `step`, but uses bytes instead of items). The `shape` and `strides` tuples both have one entry for each dimension of the array.

For contiguous arrays, `strides` follow directly from `shape` and the element size. For example, a 3-dimensional array of 32 bit (4 byte) `float`s, whose shape is $(2, 3, 4)$ has 24 ($2 \times 3 \times 4$) elements. The strides tuple for such an array would be $(48, 16, 4)$. (Recall that C uses row-major order.)

These properties are available to the Python programmer, but are primarily used internally by *NumPy*. In keeping with the aforementioned semantics, when a slice of an `NDarray` is created, a new `NDarray` is created which points to the data of the riginal array but has its own shape and strides.

2.5.2 *Broadcasting.* An array's `strides` tuple may contain zeros. If an array's `strides[n] = 0`, then data in dimensions below `n` will be repeated `shape[n]` times. This leads to one of the key features of *NumPy*, *broadcasting*, which allows operations to be performed on arrays whose `shape` tuples are not identical. In order for an binary operation two `NDarray`s to be valid, both of the operands need to be *broadcast-compatible* with each other.

Broadcasting is, at most, a two-step process. The first step is necessary only if the two arrays differ in their numbers of dimensions. In this case, the size of the `shape` and `strides` tuples of the `NDArray` with fewer dimensions are extended until they are the same length as the other `NDarray`. This is done by prepending 1's to the `shape` tuple and prepending 0's to the `strides` tuple.

The second step is to compare the corresponding elements of the `shape` tuples. Elements match if they are equal to eachother or if at least one of them is equal to 1. The latter case is the origin of the term "broadcasting": the values of the `NDarray` with a 1 in one dimension of its (effective) shape will be broadcast to all elements of the other `NDarray`. If the match is successful, the operation may proceed.

For the purposes of broadcasting, scalars are treated as an array with zero dimensions, which makes them broadcast-compatible with any array.

As an example, suppose we have two `NDarray`s: `A` and `B`. If `A`'s shape and strides are $(11, 5, 7)$ and $(140, 28, 4)$; and `B`'s shape and strides are $(5, 7)$ and $(56, 8)$. The first step would be to extend the size of `B`'s `shape` and `strides` tuples. They would become $(1, 5, 7)$ and $(0, 56, 8)$. The next step would be to check that the corresponding elements of `A`'s and `B`'s `shape` tuples match, and they do. Therefore, `A` and `B` are broadcast-compatible. Conceptually, `B`'s single $5x7$ array would be repeated 11 times and combined with `A`'s 11 $5x7$ arrays. A more detailed discussion of broadcasting can be found in [Oliphant 2006].

```
 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)
    ⋮
```

Fig. 2. Changes Required to Translate the *NumPy* Program in Appendix A to *GpuPy*. Only lines requiring changes are shown and the changes are underlined.

2.6 Lazy Evaluation

Most programming languages evaluate expressions when they are assigned, or 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. Two examples of programming languages that use lazy evaluation are Haskell [Hudak et al. 2007] and Miranda [Turner 1986].

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 value of the variable is actually needed. The benefits of lazy evaluation are avoiding unnecessary and redundant calculations. As demonstrated by Tarditi et al. [2006]; there are additional benefits when using a GPU which will be discussed in the following sections.

3. USING *GPUPY*

To motivate our discussion of *GpuPy* internals, we first provide a brief overview of how it is used and its capabilities.

GpuPy is a Python extension module that interfaces with a GPU. It 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 allow a GPU to be easily used from Python and to do so using an interface that is similar or identical to an existing API (*NumPy*). Successfully meeting these goals provides a system that can outperform CPU-only programs and requires little knowledge beyond that needed to use *NumPy*.

Translating a *NumPy* program to use *GpuPy* is trivial: Import `gpupy` instead of `numpy` and create `array` elements with type `gpufloat32` instead of any of the (C) types supported by *NumPy*.

Appendix A contains source code for a simple *NumPy* raytracing program that renders a single shaded sphere. Only three lines, shown in Figure 2, need to be changed to convert it to use *GpuPy*. Figure 3 shows the result (identical for both versions). Note that, somewhat paradoxically, apart from the driver code shown in Figure 1, we are not using any OpenGL primitives to do this: It is an analytically-

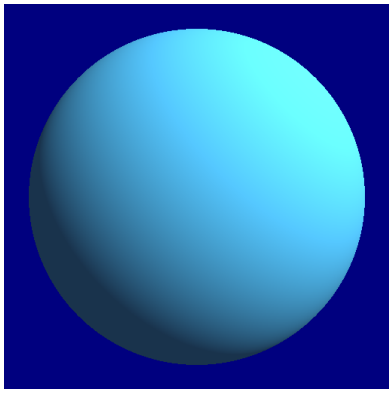

Fig. 3. Image Rendered by the *GpuPy* version of the Shaded Sphere Program Shown in Appendix A. The *NumPy* version is identical.

defined, raytraced sphere, not a polyhedral approximation.

Some features of *NumPy* such as mutable arrays and advanced slicing [Oliphant 2007] are not yet supported by *GpuPy*. We expect to add them soon. In the mean time, it should be possible in many cases to modify existing programs to avoid these features. In the particular case of mutable arrays, treating GPU arrays as immutable will always lead to programs that take better advantage of GPU acceleration.

4. IMPLEMENTATION

We discuss here the internal implementation of *GpuPy*, which is written in C. *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 illustrates how the various components of *GpuPy* interact with each other.

Technically, *GpyPy* implements a subset of *NumPy* functionality. It is important to remember that a major goal of *GpyPy* is transparency: When the user requests functionality that *GpyPy* does not (yet) support (perhaps because of GPU hardware limitations) it will automatically route the request through *NumPy* without user intervention. If the functionality is supported in a later *GpyPy* release, or if the user upgrades their hardware to a GPU that supports it, the user's code runs faster but remains unchanged.

4.1 The GpuArray Class

The primary class implemented by *GpuPy* (in C) is `GpuArray`. Internally, `GpuArrays` contain two pointers, one to an underlying `NDarray` and one to another `GpuArray`, and a set of attributes.

Allocation of the pointed-at `NDarray` is on an as-needed basis. When the pointer is `NULL`, the array's data is either residing in the GPU or has not yet been evaluated. When it is not `NULL`, the pointed-at `NDarray` contains the array's data. In most

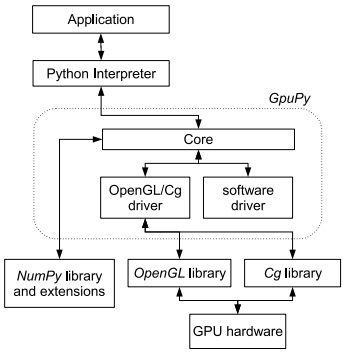

Fig. 4. Block Diagram of *GpuPy*, a Python extension module that implements (a subset of) the *NumPy* API. Note that the Driver Layer insulates the rest of *GpuPy* from the API being used to control the GPU.

cases, therefore, a `GpuArray` itself does not contain any data. The one exception to this is an optimization that avoids allocating the `NDarray` when it would contain only a scalar as part of an expression involving one or more actual arrays.

The `GpuArray` pointer is to a `GpuArray` called the *data owner*. When a `GpuArray` is a slice, its data owner points to the `GpuArray` from which the slice was taken. When a `GpuArray` is not a slice, the `GpuArray`'s data owner is itself. There is always only one level of indirection: A slice of a slice points to the original unsliced array.

The main attributes in a `GpuArray` are

—an offset (index) into its data owner,

—its number of dimensions,

—its shape,

—its strides,

—its type, and

—an "evaluated" bitmap.

Because there are many possible ways in which a given data owner could be viewed (i.e., sliced or reshaped), it is best to think of the attributes as describing a "view" of a `GpuArray`, rather than the object itself.

A `GpuArray` can be one of three types:

—If the type is `ARRAY`, then all of the data is present and there is always an underlying `NDarray` present.

—If the type is `CONSTANT`, then there is never an underlying `NDarray` present and the value of the constant is stored in the `GpuArray`.

—If the type is `EXPRESSION`, then there may or may not be an underlying `NDarray` present. In this case, the `GpuArray` contains zero or more pointers to child `GpuArray`s.

For example, after the assignment `a = b + c`, `a` has the `EXPRESSION` type and `b` and `c` are its children.

`EXPRESSION GpuArray`s can have part of their data evaluated and part of it unevaluated: There is then a bit in the "evaluated" bitmap 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 attributes that make up a block represent a single piece of a view, and more importantly, the contents of a single GPU texture. Operations in *GpuPy* are always performed on blocks. Before a shader is executed, the blocks upon which it depends must reside on the GPU. Executing a shader produces a block that may be copied back to the CPU.

A block number is similar to a page number in a virtual memory system [Tanenbaum 1999] in that it represents a region of memory that may or may not be present on the GPU at any given time. Each `GpuArray` can be thought of as a region of virtual memory, some part of which is backed by blocks on the GPU.

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 in a hash table and a least-recently-used (LRU) list. The hash table is used to quickly determine whether a block is already present on the GPU, and the LRU list is used to choose an appropriate candidate for eviction from the GPU if its memory becomes full. 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 [Tanenbaum 1999]. If GPU memory becomes exhausted, then *GpuPy* must evict a block from the GPU in order to make room for the new block. We believe LRU to be a reasonable algorithmic choice, but further research is called for. For example, it may be better to aggressively allocate textures and simply rely on the GPU's device driver to take care of the details.

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 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. [2006], *GpuPy* overcomes this limitation by using lazy evaluation. Lazy evaluation can increase the overall performance of a

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

Fig. 5. A Simple *GpuPy* Program. This program was intentionally written with each operation on its own line, which allows the expressions produced to be identified by the line number.

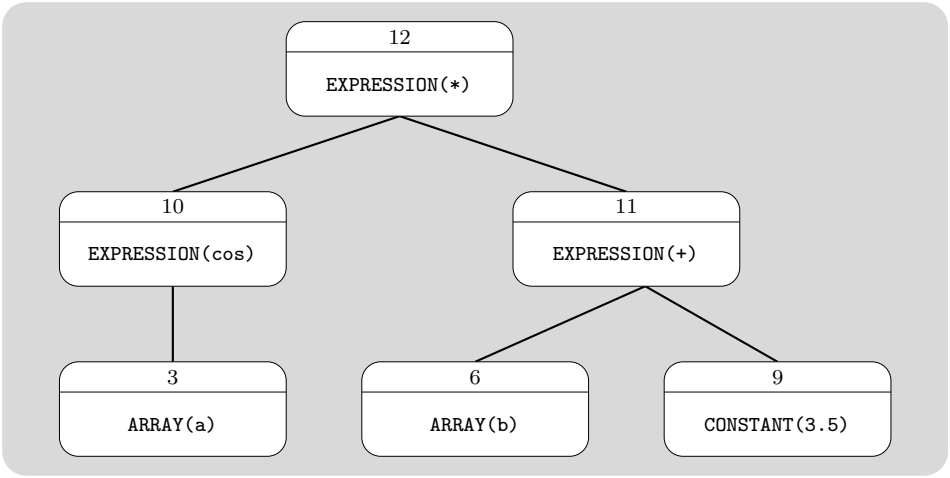

Fig. 6. Expression Tree Produced from the *GpuPy* code in Figure 5. The number contained in each expression corresponds to the line number in Figure 5 that produced it.

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 5 shows an example *GpuPy* program. The expression tree produced by this code is shown in Figure 6. When execution reaches line 14, 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 `EXPRESSION` `GpuArray`s contain an bit in a bitmap on the host for each element in the array. This bit is

cleared when the `GpuArray` is created (in the GPU) and set when a block containing that element is evaluated and copied to the CPU. This is somewhat analogous to the "dirty" bit used in a virtual memory system [Silberschatz et al. 2005]. As in *NumPy*, slices in *GpuPy* refer to the same underlying data and therefore don't require any additional bits.

GpuPy does not allocate a `GpuArray`'s underlying `NDarray` until the first bit needs to be set. When an element in an array is needed, *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 all of its corresponding bits (including the original one) are set.

The need may arise for a number of reasons, but in general it is when the element's 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 (at least at the current level of *GpuPy*'s development) is required.

Evaluating all of the blocks of a `GpuArray` is called *flushing* and is done when a complete underlying `NDarray` is needed. This is necessary, for instance, when the `NDarray` is going to be passed to some function not controlled by *GpuPy* that expects an `NDarray`.

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 7 shows the dependent blocks calculation algorithm.

Keeping track of the `GpuArray`'s attributes is not difficult, as most of them 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 The Driver Layer

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.

The Driver Layer requires 14 essential functions that each driver implementation

```
# Calculate the dependent block
def BuildView(block, child):
    # create a new block that references
    # the child expression
    child_block = block(child)

    # the array, number of dimensions, shape,
    # and block number stay the same
    child_block.nd = block.nd
    child_block.shape = block.shape
    child_block.block_number = block.block_number

    # calculate new strides and offset
    child_block.offset = child.offset
    for i in range(block.nd):
        child_block.strides[i] =
                block.strides[i] * child.strides[i]
        child_block.offset +=
                block.array.offsets[i] * child.strides[i]

    return child_block
```

Fig. 7. The `BuildView` Algorithm. This algorithm constructs a dependent block by combining the attributes of the parent block and the child expression.

must provide. In addition to these functions, a driver also provides functions for any of the *NumPy* functionality that it knows how to reproduce. These additional functions depend on the model of GPU being used and are chosen from the set of all methods known to *NumPy*.

4.6.1 *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.

4.6.2 *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 others.

—`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 a negative value 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.

We will discuss these in greater detail in Section 4.7.

4.6.3 *Evaluation*

—`block_t *evaluate_expr(PyGpuArrayObject *gpa, int bnumber)`:
Evaluates block `bnumber` for the expression contained in `gpa` and returns the `block_t` that contains the result, or `NULL` if the expression can not be evaluated.

4.6.4 *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 operation. 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` representing the appropriate expression tree.

4.7 Partitioning

As mentioned in the Section 2, 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. Partitioning is currently handled by the driver layer.

4.8 The OpenGL/Cg Driver

GpuPy's primary driver uses OpenGL and NVIDIA's Cg library [Woo et al. 1999; CgS 2008] to send shader information to the GPU. It implements the most commonly used *NumPy* operations.

The Cg driver works in three stages. During the first stage, a depth-first traversal of the expression tree is performed and stored in a linked list. The traversal always expands larger sub-trees first, so that when code is generated, temporary values will be used as soon as possible rather than consuming a temporary register. The linked list produced here is processed by the remaining two stages to evaluate the expression.

The second stage is the most complex of the three. It must walk through the list generated in the first stage and generate shader code that can be run on the GPU. The complex part of this process is that due to resource constraints placed on shaders, the operations stored in the linked list may not all fit into a single shader. Below is a list ways in which a shader is limited.

—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.

The second stage iterates over the list and divides it into a series of sub-lists that fit into a single shader. Shaders can only write a limited number of outputs, the boundaries between sub-lists must occur when the number of temporary values is less than or equal to this limit (1 in the current version of *GpuPy*). During iteration, the second stage always remembers the most recent place in the list where this condition is met. This is the most recent safe place at which the list may be divided. At each step, the resource usage is calculated and if this is greater than the capabilities of the GPU, then the list is divided at the last waypoint and the algorithm begins again at the node following the last safe waypoint. Each sub-list produced in stage two is passed to stage three where it is evaluated on the GPU. If stage three fails, then stage two will back up to the next-to-last waypoint and try stage three again. This process repeats until stage three is successful or the front of the list is reached, in which case the algorithm fails.

Stage three is relatively simple: given a sub-list, emit shader code and evaluate this code into a newly allocated `GpuArray`. If this process fails, then stage two

operation	source	P_{50}	P_{75}	P_{97}	P_{98}	P_{99}	P_{100}
add	*NumPy*	0.00	0.00	5.00e-1	5.00e-1	5.00e-1	5.00e-1
	GpuPy	0.00	0.00	5.00e-1	5.00e-1	5.00e-1	5.00e-1
subtract	*NumPy*	0.00	0.00	5.00e-1	5.00e-1	5.00e-1	5.00e-1
	GpuPy	0.00	0.00	5.00e-1	5.00e-1	5.00e-1	5.00e-1
multiply	*NumPy*	2.43e-1	3.71e-1	4.88e-1	4.94e-1	4.99e-1	5.00e-1
	GpuPy	2.43e-1	3.71e-1	4.88e-1	4.94e-1	4.99e-1	5.00e-1
divide	*NumPy*	2.50e-1	3.67e-1	4.91e-1	4.99e-1	5.00e-1	5.00e-1
	GpuPy	3.13e-1	5.42e-1	9.58e-1	1.00	1.11	1.50

Table I. The error in ULPs for basic operations for *NumPy* and *GpuPy*. P_N is the N^{th} percentile.

adjusts the sub-list and tries again. The code generation in stage three is very simple and does not perform any optimizations. After it produces the code, it compiles and evaluates the code using Cg library functions.

4.9 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.

5. EVALUATION

We evaluate both the quality and the performance of *GpuPy*'s calculations. Quality is measured by how close *GpuPy* is to the arbitrary-precision result, and performance is measured by running a test program under both *GpuPy* and *NumPy*.

5.1 Quality

One way to measure error in floating point numbers is Units in the Last Place (ULPs). For a given floating point number, the ULP is the quantity represented by the least significant digit of the floating point number. This quantity depends on the exponent and the number of digits of precision. Consider the floating point value 2.71828×10^3. For this number, the base $B = 10$, the precision $p = 6$, and the exponent $e = 3$. The ULP in this case is 0.00001, or B^{p-e+1}. In general, a floating point number can represent any real number in its range to within 0.5 ULPs. For a detailed discussion of floating point numbers, see [Goldberg 1991].

For each operation tested we measure the error of the *GpuPy* result and the error of the *NumPy* result. A large number of trials are run over a range of input values and the statistics are collected. The percentiles for the error are calculated because they illustrate the general behavior of GPU arithmetic. The fact that the highest percentiles are much larger than the lower ones indicates that the worst errors are outliers and occur for only a limited number of inputs. The first group of functions evaluated is the basic arithmetical operations: *add*, *subtract*, *multiply* and *divide*.

operation	source	P_{50}	P_{75}	P_{97}	P_{98}	P_{99}	P_{100}
arccos	*NumPy*	2.04e-1	3.38e-1	4.65e-1	4.78e-1	4.90e-1	4.95e-1
	GpuPy	2.07e2	4.04e2	5.47e2	5.61e2	5.67e2	7.05e2
arcsin	*NumPy*	2.41e-1	3.59e-1	4.85e-1	4.89e-1	4.94e-1	4.98e-1
	GpuPy	5.63e2	1.65e3	9.09e3	1.25e4	1.25e4	9.30e6
arctan	*NumPy*	2.34e-1	3.17e-1	4.78e-1	4.86e-1	4.86e-1	4.89e-1
	GpuPy	3.44e1	4.63e1	6.69e1	6.72e1	6.83e1	7.22e1
cos	*NumPy*	2.48e-1	3.75e-1	4.86e-1	4.91e-1	4.95e-1	5.00e-1
	GpuPy	2.06	5.55	5.57e1	8.02e1	1.58e2	1.42e4
cosh	*NumPy*	2.43e-1	3.89e-1	4.86e-1	4.87e-1	4.98e-1	4.98e-1
	GpuPy	7.98e-1	1.16	1.94	2.02	2.10	2.24
exp	*NumPy*	2.18e-1	3.43e-1	4.82e-1	4.83e-1	4.87e-1	4.89e-1
	GpuPy	1.09	1.45	2.32	2.42	2.65	2.75
fmod	*NumPy*	0.00	0.00	0.00	0.00	0.00	0.00
	GpuPy	1.00	2.00	1.89e5	3.15e5	1.05e6	1.61e7
log	*NumPy*	2.38e-1	3.74e-1	4.67e-1	4.77e-1	4.92e-1	4.96e-1
	GpuPy	5.52e-1	8.88e-1	1.53	1.68	2.09	4.30
log10	*NumPy*	2.59e-1	3.79e-1	4.78e-1	4.82e-1	4.96e-1	4.97e-1
	GpuPy	3.22e-1	5.55e-1	1.04	1.16	1.34	4.39
power	*NumPy*	2.40e-1	3.71e-1	4.83e-1	4.87e-1	4.93e-1	5.00e-1
	GpuPy	3.68	7.55	1.88e1	2.06e1	2.35e1	3.47e1
sin	*NumPy*	2.34e-1	3.78e-1	4.69e-1	4.83e-1	4.87e-1	4.96e-1
	GpuPy	1.68	4.99	3.42e1	3.54e1	8.00e1	7.86e6
sinh	*NumPy*	3.37e-1	5.94e-1	9.83e-1	1.15	1.18	1.20
	GpuPy	1.73	2.47	7.58	1.50e1	2.62e1	6.70e1
sqrt	*NumPy*	2.16e-1	3.74e-1	4.79e-1	4.82e-1	4.90e-1	4.92e-1
	GpuPy	3.29e-1	5.38e-1	1.00	1.00	1.00	1.32
tan	*NumPy*	2.37e-1	3.67e-1	4.82e-1	4.86e-1	4.88e-1	4.96e-1
	GpuPy	7.27	1.71e1	7.67e1	2.05e2	8.39e6	1.68e7
tanh	*NumPy*	1.81e-1	3.41e-1	4.66e-1	4.73e-1	4.85e-1	4.95e-1
	GpuPy	1.53	2.24	1.10e1	1.69e1	1.86e1	2.01e1

Table II. The error in ULPs for functions for *NumPy* and *GpuPy*. P_N is the N^{th} percentile.

Table I shows that *NumPy*'s single-precision floating point values are always within 0.5 ULP of the actual value. This can be viewed as evidence that the basic operations are performed faithfully by *NumPy*. This level of precision is expected of most (if not all) modern CPUs that support floating point calculations. In general, the same cannot be said for GPUs. *GpuPy*'s floating point values are all within 0.5 ULP for addition, subtraction, and multiplication, but some are greater than 0.5 ULPs for divide. This means that calculations performed by the GPU do not necessarily produce the floating point value closest to the actual value. The divide operation's greater error can be explained by the fact that division is implemented using recipricol and multiply. Although not as precise as the values calculated using *NumPy*, *GpuPy* does a reasonably good job.

The next group are functions that are implemented by *NumPy* and *GpuPy*. These functions are *arcsin*, *arccos*, *arctan*, *cos*, *cosh*, *sin*, *sinh*, *tan*, and *tanh*.

Table II shows that *NumPy*, as expected, produces good results. With the exception of *sinh*, all functions are within 0.5 ULP of the actual value. *GpuPy* clearly does not perform as well as *NumPy* for these functions. This is because the full 24 bits of precision offered by single-precision numbers is simply not needed for most

graphics applications. Another contributing factor is that only some of the operations in Table II are implemented in hardware and the remainder are composed of those.

Several other operations are implemented by *GpuPy*, but they are uninteresting because the result is derived directly from the input without performing any real calculations. The uninteresting operations are: *absolute*, *ceil*, *equal*, *fabs*, *floor*, *greater*, *greater_equal*, *less*, *less_equal*, *maximum*, *minimum*, and *not_equal*. For all of these operations, *NumPy* and *GpuPy* produce the same results with zero error.

5.2 Performance

To test performance, we run the same program on both *NumPy* and *GpuPy* and compare the results. The test program generates a grayscale image produced by calculating the minumum distances between each pixel and a given set of points. More formally, given a set S of randomly chosen points and an $M \times M$ grid of pixels. For each grid point p, calculate:

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

Intuitively, the projection of the "ridges" of $d()$ onto the image plane is the Voronoi diagram [de Berg et al. 2000] of S.

This algorithm is an especially useful one for testing *GpuPy*, as we can scale the size of the image to adapt to GPUs with larger or smaller amounts of texture memory and we can scale the size of S to increase the depth of the expression tree to exercise lazy evaluation. To display the results, we linearly map them so that the minimum value of $d()$ corresponds to 0 and the maximum value of $d()$ corresponds to 1. Figure 8 shows an image produced by the distance map test.

We ran the distance map program in *GpuPy* and *NumPy*. A helper script runs the two versions and compares their performance and results. Each version is run several times, with increasing sizes of S.

Figure 9 compares the running trials of a Python program that implements the distance map algorithm. Each trial involves running the program a number of times with an increasing number of points. Two trials were run in *NumPy* mode on different CPUs and two trials were run in *GpuPy* mode on different GPUs.

6. 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. In addition, *GpuPy* supports arrays larger than can fit on the GPU.

GpuPy may allow many existing *NumPy* programs to be run using a GPU making only trivial changes. 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*.

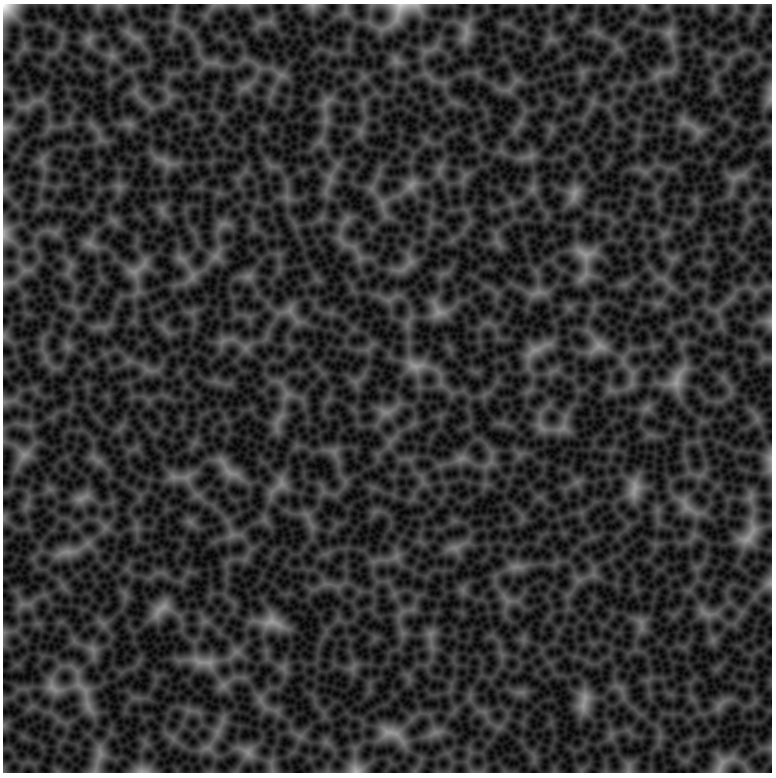

Fig. 8. Distance Map Image. This was generated using a set S of 5000 randomly chosen points on a 512×512 grid. Each pixel's intensity is set according to the distance from it to the nearest point in S.

7. 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, MLab (MATLAB$^{\text{TM}}$compatibility), and MA (masked arrays) 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 those used for geometry and image-processing, are more easily described using vectors rather than scalars. The sphere rendering done in Section 3, 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* [Tarditi et al. 2006]. 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.

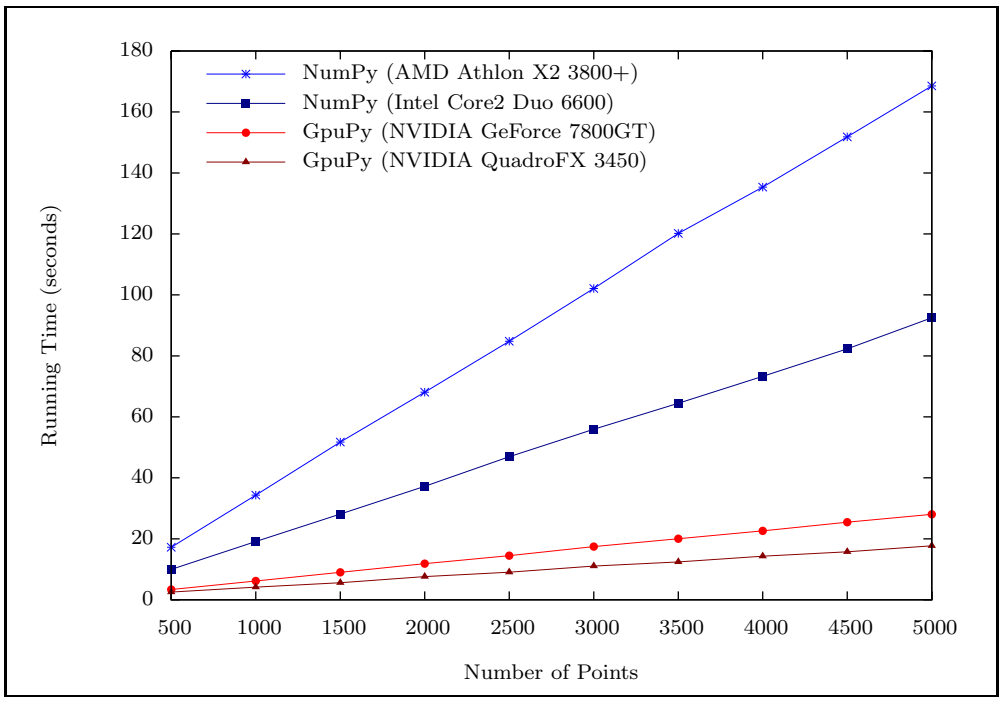

Fig. 9. Distance Map Performance Comparison. This plot compares the performance of *GpuPy* and *NumPy* on two different systems.

—Multiple render buffers: Newer 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 [Peercy et al. 2006] and NVIDIA's CUDA [CUD 2008]. Drivers could also potentially be written that use something other than a GPU to perform calculations. An Ethernet-connected GPU cluster was described in [Fan et al. 2004]. The GPU cluster outperformed CPU-based solutions for a flow simulation.

REFERENCES

2008. Cg Language Specification. `http://developer.nvidia.com/object/cg_toolkit.html`.

2008. Cuda Zone. `http://developer.nvidia.com/object/cuda.html`.

Akeley, K., Allen, J., Beretta, B., Brown, P., Craighead, M., Eddy, A., Everitt, C., Galvan, M., Gold, M., Hart, E., Juliano, J., Kilgard, M., Kirkland, D., Leech, J., Licea-Kane, B., Lichtenbelt, B., Lin, K., Mace, R., Morrison, T., Niederauer, C., Paul, B., Puey, P., Romanick, I., Rosasco, J., Sams, R. J., Sandmel, J., Segal, M., Seetharamaiah, A., Schamel, F., Vogel, D., Werness, E., and Woolley, C. 2008. Ext_framebuffer_object. `http://www.opengl.org/registry/specs/EXT/framebuffer_object.txt`.

Ciricescu, S., Essick, R., Lucas, B., May, P., Moat, K., Norris, J., Schuette, M., and Saidi, A. 2003. The reconfigurable streaming vector processor (rsvptm). In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, Washington, DC, USA, 141.

Dally, W. J., Kapasi, U. J., Khailany, B., Ahn, J. H., and Das, A. 2004. Stream processors: Progammability and efficiency. *Queue 2,* 1, 52–62.

de Berg, M., van Kreveld, M., Overmars, M., and Schwarzkopf, O. 2000. *Computational Geometry: Algorithms and Applications*, 2nd. ed. Springer-Verlag.

Fan, Z., Qiu, F., Kaufman, A., and Yoakum-Stover, S. 2004. GPU cluster for high performance computing. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, Washington, DC, USA, 47.

Fernando, R. and Kilgard, M. J. 2003. *The Cg Tutorial*. Addison Wesley.

Goldberg, D. 1991. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv. 23,* 1, 5–48.

GPGPU 2008. General-Purpose Computation Using Graphics Hardware. `http://www.gpgpu.org`.

Greenfield, P., Miller, J. T., Hsu, J.-c., and White, R. 2003. numarray: A New Scientific Array Package for Python. In *PyCon Proceedings*.

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

Hudak, P., Hughes, J., Jones, S. P., and Wadler, P. 2007. A history of haskell: being lazy with class. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*. ACM Press, New York, NY, USA, 12–1–12–55.

Hugunin, J. 1995. The Python Matrix Object: Extending Python for Numerical Computation. In *The Third Python Workshop*. http://www.python.org/workshops/1995-12/papers/hugunin.html.

IEEE P754 2006. Draft Standard for Floating-Point Arithmetic P754. `http://grouper.ieee.org/groups/754`.

Luebke, D., Harris, M., Krüger, J., Purcell, T., Govindaraju, N., Buck, I., Woolley, C., and Lefohn, A. 2004. GPGPU: general purpose computation on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Course Notes*. ACM Press, New York, NY, USA.

Microsoft. 2008. HLSL. urlhttp://msdn.microsoft.com/en-us/library/bb509561

Oliphant, T. E. 2006. Guide to NumPy. `http://www.numpy.org`.

Oliphant, T. E. 2007. Python for scientific computing. *Computing in Science and Engg. 9,* 3, 10–20.

Owens, J. D., Dally, W. J., Kapasi, U. J., Rixner, S., Mattson, P., and Mowery, B. 2000. Polygon rendering on a stream architecture. In *HWWS '00: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*. ACM Press, New York, NY, USA, 23–32.

Peercy, M., Segal, M., and Gerstmann, D. 2006. A performance-oriented data parallel virtual machine for gpus. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Sketches*. ACM Press, New York, NY, USA, 184.

Rost, R. J. 2006. *OpenGL Shading Language*, 2nd. ed. Addison Wesley.

Segal, M. and Akeley, K. 2006. The opengl graphics system: A specification. `http://opengl.org/documentation/specs/`.

Silberschatz, A., Gagne, G., and Galvin, P. B. 2005. *Operating System Concepts*, 7th ed. Wiley.

Tanenbaum, A. S. 1999. *Structured Computer Organization, 4th ed.*, 4 ed. Prentice Hall, Chapter 6.

Tarditi, D., Puri, S., and Oglesby, J. 2006. 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*. ACM Press, New York, NY, USA, 325–335.

Turner, D. 1986. An overview of miranda. *SIGPLAN Not. 21,* 12, 158–166.

van Rossum, G. 2008a. *Extending and Embedding the Python Interpreter*, 2.5.2 ed. Python Software Foundation.

van Rossum, G. 2008b. *Python Reference Manual*, 2.5.2 ed. Python Software Foundation.

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

Woo, M., Davis, and Sheridan, M. B. 1999. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

APPENDIX

A. SHADED SPHERE SOURCE CODE

```
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

 # intersection
(ix, iy, iz) = (vx + t*dx, vy + t*dy, vz + t*dz)

# normal to sphere at intersection
# (this is guaranteed to be of unit length)
(nx, ny, nz) = ((ix-cx)/r, (iy-cy)/r, (iz-cz)/r)

# dot product of sphere normal and light normal
# (for diffuse shading)
nDotL = nx*lx + ny*ly + nz*lz

```

```
# Where the ray hits the sphere, set to the shaded
# diffuse color, otherwise set to black.
channels = [ 255 * where(disc > 0,
             where(nDotL > 0, ka_i + nDotL * kd_i, ka_i),
             bg_i) for (bg_i, ka_i, kd_i) in zip(bg, ka, kd) ]

# Convert the array to an image and write it as a PNG file.
imgs = [ Image.frombuffer(
           "F", (w, h), c, "raw", "F", 0, 1).convert("L")
         for c in channels ]
Image.merge("RGB", imgs).save("shaded_sphere.png")
```