IMPLEMENTING PERFORMANCE LIBRARIES ON GRAPHICS HARDWARE

MATTHEW M TRENTACOSTE

Abstract. We propose a simple method to implement floating-point vector math operations and matrix multiplication on graphics hardware, focusing on identification of details, in both software and hardware, which affect performance and ease of use. Before widespread adoption of the graphics processing unit (GPU) as another computation processor, we must address the need of application interfaces (APIs) that abstract away the details of the implementation. We focus on providing an interface to the hardware that utilizes high level interfaces that hide the specifics of implementing the functionality on the GPU, while maintaining performance. We then use this interface to implement non-negative matrix factorization, used for performing feature extraction, to demonstrate the strengths of the library when run on current graphics hardware.

Contents

List of Tables

List of Figures

Acknowledgements

I would like to acknowledge Doug James, Kayvon Fatahalian, and John Ketchpaw for their assistance on this project, and ATI for hardware.

1. Introduction

It has been widely accepted for some time that commodity graphics processing units (GPUs) are capable of performing significant amounts of computation. Until recently, little work has been done on using graphics hardware for purposes other than generating imagery. This is largely because, for the majority of its existence, graphics hardware has not been very programmable. Before the introduction of shaders into the render pipeline, mapping algorithms to graphics hardware was tedious, if not impossible, requiring complicated abstractions. When paired with floating point datatypes to accurately represent the range of values that the shaders can potentially generate, the GPU is capable of accurately implementing a wide range of functions. This versatility, which extends beyond the scope of merely rendering, provides much of the basis needed for a processor to be usable for general computation.

Numerous algorithms have been implemented on graphics hardware as a result of this change in the pipeline. Methods such as real-time computation of caustics by Trendall and Steward [40], Perlin Noise by Hart[9], and more complicated operations such as ray tracing by Purcell et al.[33] have been possible. There has also been work in relation to solvers for specific problems such as interactive physical simulation by Harris[8], diffusion modelling by Diewald et al.[5] and Rumpf & Strzodka[37], Voronoi computation by Hoff[10], level set solvers by Lefohn & Whitaker[20], Lattice Boltzmann computations by Li et al.[21], and general-purpose solvers by Bolz[2] and Kruger & Westermann[16].

All of these works are implemented through an existing graphics API, notably OpenGL or Direct3D. While GPUs perform many general operations, both OpenGL and Direct3D only expose graphics-related abstractions of the underlying general functionality. An example of this is that all the methods use textures to store data between the multiple passes required for a given operation as there is no other way. As a result, most general computation solvers on graphics hardware, with the exception of Kruger and Westermann[16], exist as singular closed objects to perform a specific task. While the solver performs multiple steps, due to the difficulty associated with working within the graphics-oriented abstraction of the API, those individual steps are not exposed to the user. As the solvers are generally operating only on data they had previously operated on, there are also compatibility issues between most solvers due to the representations of data they assume. Often data must be transferred back to the central processing unit (CPU) to convert to new representation at substantial cost to performance.

Our current work attempts to create a matrix library to act as a sort of "glue" between various other solvers, providing matrix and vector functions to perform computations that would otherwise require the data to be moved back to the CPU. Instead of complex routines, we expose an efficient set of simple operators which can easily be combined to perform arbitrarily complex computations that would otherwise require data to be moved back to the CPU.

In the remainder of this paper, we first determine a representation of data to work as efficiently as possible within the architectural basis of graphics hardware, paying special attention to how the fragment shader performs texture fetches. Second, we implement two general functions based on that representation, a method to perform an arbitrary operation to each element of a vector and a method for performing matrix/vector products. Lastly, we implement non-negative matrix factorization as a combination of these operators to demonstrate the utility of the library.

2. Graphics Hardware

Fundamentally, modern graphics hardware is an implementation of depth-buffered rendering and broken into two stages: geometry and rasterization. Currently, geometry is represented as collections of triangles defined by triples of vertices, the vertices are transformed by some process, rasterized into fragments, and then outputted by another process operating per fragment. The outputs of the fragment program are often compared to the values currently residing in the final buffer (usually based on depth into the scene) and are then blended with the current value based on some test. Figure 1 contains a diagram of the current graphics pipeline. The dotted line of blending stage indicates that it does not always apply as will be described later.

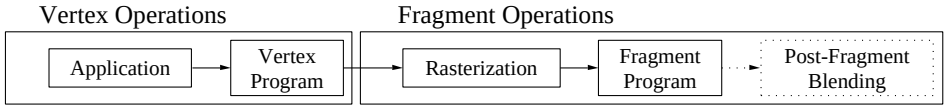

Figure 1. High-level model of programmable graphics pipeline

The GPU is separate from the CPU and operates asynchronously, resulting in the majority of calls to a graphics API to return immediately as the CPU is no longer involved in the computational load dispatched to the GPU. This creates a relationship between processors much like that of threads where semaphores are necessary to ensure that each has the correct available data on which to perform computations. Even under very heavy rendering workloads, the CPU is free to perform other operations. There is an overhead cost in dispatching data to the GPU, so CPU free cycles increase rapidly as the amount of GPU work per load increases.

The concept of *shaders* moves the vertex and fragment operations away from a state machine representation to a more programmable model[1], allowing the execution of user-defined code that is more versatile than setting state on some predefined implementation. This integration of programmable operations into the graphics pipeline provides a greater scope of possible outputs from the vertex and

[1]**Programmable shaders :** These shaders are small programs comprised of assembly code that operate on their respective elements though a simpler processor. They perform the exact same function as their fixed-function counterparts, but have the option to do more.

fragment stages than previously allowed. The classic example of this is the explosion in various lighting models that have appeared in hardware rendering methods in recent years as compared to previous implementations that relied on the fact that the graphics card computed the standard Ambient/Diffuse/Specular lighting per vertex.

While shaders allow more varied methods of processing vertices and fragments, they only expand the functionality available at various stages, not alter the entire pipeline. Triples of vertices are transformed per-vertex, then taken as a triple and rasterized, and the results of that operation are computed per-fragment. The individual elements at each stage are evaluated separate from other elements. This restriction is significant not only in how it affects the scope of possible operations of the GPU, but how the GPU obtains considerable performance improvements over a CPU of comparable complexity.

2.1. **Stream Processing.** More formally, the system architecture paradigm of forcing elements of a collection to be operated on individually, on which the GPU is based is known as *stream processing*. The motivation for such a design is to try to alleviate the ever-increasing disparity between the computation speed of a processor and the transfer speed of the caches it utilizes to store data. Figure 2 shows a comparison of the flow models of a streaming processor and a regular CPU. The *stream processing model* has two main components, a *stream* of data elements and a *kernel* that operates on the elements of that *stream*.

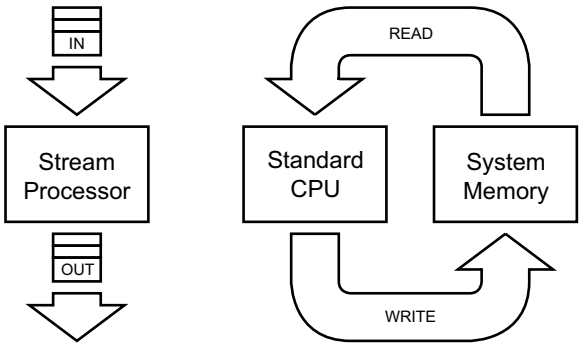

FIGURE 2. Comparison of stream processor and CPU architectures
The fundamental difference is that even though the IN and OUT for the stream processor are part of main memory, all it can see is the current element. A kernel is loaded on the stream processor and processes all elements going through the processor until a new kernel is set. A standard CPU has free access to memory, reading and writing instructions and data as needed.

A *stream* of elements is a simple collection of (potentially) complex elements, and can most easily be viewed as an array of `structs`. This differs from existing vector processors in that the elements can consist of multiple datatypes. If a vector processor was performing i operations, such as $A * B + C$, on three floats A, B, C, there would be a vector of length i for A, a vector of length i for B, and a vector of

length i for C. If a *stream processor* was performing the same operation, it would have one stream of length i in which each element would contain A,B, and C.

The *kernel* that operates upon those elements is some program that takes an element of some kind as input and produces an element of some kind as output. The *kernel* enforces the key paradigm of stream computing where no element can perform operations dependant on other elements. In other words, stream processing *kernels* explicitly do not have the ability to *gather* input elements or *scatter* output elements. This forces an inherent parallelism into the model of the stream processor which is the basis for the performance gains that graphics hardware has over the CPU.

This parallelism is the basis of all graphics hardware companies touting their performance increases "Moore's Law cubed", stating that their hardware doubles in performance every six months as opposed to every 18 months as claimed by CPU manufacturers. Requiring that every kernel operate without *gather* of input elements or *scatter* of output elements, allows all available silicon to be filled with parallel pipelines. Given an abstraction that guarantees the availability of multiple parallel pipelines of computation, and that every element is capable of being computed separately, it becomes apparent that if an element is in the cache, it can be operated upon without any restrictions by definition of the stream processing abstraction. The main advantage of using a stream processing architecture is the inherent ability to hide the memory latency resulting from the cache hierarchy. Any elements residing in the line that is loaded into the cache along with a requested element are capable of being operated on, yielding that maximum possible ratio of cache hits to cache misses.

2.2. **Current Hardware.** Current graphics hardware adheres to the stream processing model with some added specializations for rendering images. The most important deviation from the simple streaming model described is the inclusion of a *gather* operator in the fragment program portion of the pipeline. Opposed to other shading systems such as Renderman who provide more functionality for procedural generation of surfaces, the majority of detail in current realtime graphics is obtained through the use of texture lookups. The fragment programs still map one input fragment to one output fragment, but have the ability to look up elements from other specific streams (textures).

Textures are basically arrays with the added ability to look up non-integer elements via blending the surrounding entries. They are used to store a variety of data including standard diffuse maps, lookup tables[2] for functions and intermediate results of computation. The pervasive use of texturing in modern graphics hardware

[2] **Texture lookup tables :** There are many examples of encoding functions in textures, but the most common in the normalization cubemap. For those unfamiliar with the cubemap, it is a unit cube with a different texture for each face. Given XYZ texture coordinates, it looks up the values from the respective side of the cube. The normalization cubemap, instead of containing color information, it contains values such that if the texture coordinates represent the end point of a vector touching the cube at that point, the RGB values fetched are the XYZ values of that vector normalized. This was the only method of normalizing a vector as needed in per-pixel lighting in pre-DirectX 9 hardware.

has placed pressure on graphics hardware manufacturers to provide as much performance in fetching textures as possible. These optimizations manifest themselves in the filtering and storage methods of textures. Filtering involves sampling multiple texture elements (texels) of the texture and blending between them based on the values of the texture coordinate compared to the coordinates of those textures, and is beyond the scope of this paper. The other method deals with the ordering of texels in memory or the *swizzle* of the texture. Textures must have good locality of values[3] in regards to both the horizontal and vertical dimensions. *Swizzled* textures reorder squares of texels into rows in memory, so when a given texel is accessed, the three texels beside and below it are loaded into the texture cache instead of just the texel immediately beside it. This greatly increases the percentage of texture fetches that are already in cache for normal access patterns when compared to an *unswizzled* texture.

We targeted our implementation at the ATI 9700 Pro GPU, and will discuss some of the specifics of the card as they pertain to this body of work. In terms of parallelism, the GPU contains 4 pipelines for vertex transformation and 8 fragment pipelines for fragment processing. For the fragment pipelines, the 9700 Pro yields a theoretical performance of

$$4_{floats/pixel} * 8_{pixels/clock} * 350Mhz \quad = \quad 11.2GFlops \tag{1}$$

For reasons discussed later, the actual performance is significantly lower. The fragment pipelines also are fully-programmable, presenting more available operations, compared to the register combiners of previous GPUs which were more akin to the fixed-function pipeline. Both pipelines are specifically designed to be parallelized and simple to optimize. All assembly instructions execute in one clock cycle (some texture lookups are an exception), ensuring that there are no wasted cycles in preserving the pipelines operating in lockstep. Furthermore, with the exception of the most complicated instructions (`exp`, `log`, `sqrt`, and `rsqrt`), operations in vertex and fragment programs occur simultaneously on all elements of vector-4s of data.

The 9700 Pro has the ability to render to offscreen buffers in GPU memory and use those buffers as textures to perform lookups into; necessary for storing computations between steps. The most significant of the changes between the 9700 Pro and previous GPUs is that the 9700 Pro supports 24-bit floating point throughout the pipeline where previously lower-precision floating point was implemented in the vertex pipeline and fixed point was implemented in the fragment pipeline.

While floating point is a large improvement in terms of robustness of the computations performed in the vertex and fragment programs, for these high-precision computations to be of use, there needs to be high precision datatypes to store them intermittently. The 9700 Pro has support for floating point textures, but it comes

[3]**Texture swizzle :** For the normal case of texture coordinates being interpolated over a triangle, if pixel(i, j) maps to texel(i, j), then pixel$(i+1, j)$ maps to texel$(i+1, j)$ and pixel$(i, j+1)$ maps to texel$(i, j+1)$.

Format	Mantissa	Exponent	Notes
16 bit	10 bits	5 bits	Texture format
24 bit	16 bits	7 bits	Pipeline precision
32 bit	23 bits	8 bits	Texture format

TABLE 1. ATI 9700 Pro floating-point precisions

at a cost. The increased complexity of circuitry needed to perform arithmetic and conversion on floating point data prevents certain operations from occurring in current hardware, most notably in the features available for texture filtering and post-fragment blending (as indicated by the dotted box in Figure 1). Table 1 shows the floating point formats available and where they are used. For our library we were only interested in 32-bit floating point data as we seek to offer precision of computation as close to the CPU as possible. Texture filtering is not supported for floating point textures, but was not relevant because we used textures as look-up tables and did not want any modification of values based on offsets in the texture coordinates. On the other hand, the lack of post-fragment blending plays a very large role in the observed performance of our library.

3. Mathematical Kernels

Mapping traditional computational operations to graphics hardware is not always a straightforward process. Even if the mapping of a given operation is fairly simple, the restrictions imposed through the graphics API and the hardware of the GPU present an entirely different set of performance pitfalls that must be considered. The two main functions of our library are to perform user-defined vector math kernels on a per-element basis and to multiply two matrices together. We discuss both the instructions pertaining to the operations and the methods of storing and manipulating data implemented in the library.

Inv	Div	Sqrt	InvSqrt
Cbrt	InvCbrt	Pow	Exp
Ln	Log10	Cos	Sin
Sincos	Tan	Acos	Asin
Atan	Atan2	Cosh	Sinh
Tanh	Acosh	Asinh	Atanh

TABLE 2. Intel VML functions

3.1. **Per-Element Kernels.** There is nothing revolutionary about the concept of per-element kernels in itself. It is done every time a value in an array is read or written. When performance becomes a consideration, the concept becomes more formal. Most math libraries have a set of highly optimized (usually transcendental) functions for operating on vectors of data. Intel Vector Math Library[14] (VML) is one such set of routines. The library aims for not only efficiency in mathematical computation, but also memory access. Each element is stated to be separate from

every other element in its vector and only related to the respective element in another vector. The functions of the library know that any other elements that are loaded into the cache, along with a given element, are available to be used. Just like in the stream processing model, the library prevents excess cache misses, increasing the efficiency of memory access, and effectively hiding the memory hierarchy from the application calling the library with respect to those operations. Table 2 contains a listing of the functions provided by the library.

Instruction	Description	Cycles
`exp`	Full precision exponent (2^x)	1
`log`	Full precision $\log_2 x$	1
`pow`	Full precision x^y	3*
`rcp`	Reciprocal	1
`rsq`	Reciprocal square root	1
`sincos`	Sin and cosine	8*

* Performed through macros of instructions

TABLE 3. Pixel assembly instructions

There are aspects specific to the architecture of the GPU that must be taken into consideration when deciding on methods of representation. The internal workings of the 9700 Pro become more complicated when `exp`, `log`, `sqrt`, and `rsqrt` operations from are being utilized in the shader or if floating point textures are loaded. The previous statement that instructions in the fragment pipeline operate on vector-4s in parallel isn't completely true. The pipelines for the 9700 Pro consist of two separate arithmetic logic units (ALUs); one operating on 3-tuples and one operating on scalars. Instructions from Table 3 are only implemented in the scalar ALU of the pipeline. Texture fetches are the only fragment program assembly instructions that don't always take one clock cycle to complete. Texture fetches on the 9700 Pro take 1 clock per 32 bits of data for the texture format. A 1-channel 32-bit floating point texture takes 1 clock to load while a 4-channel 32-bit floating point texture takes 4 clocks to load. This number varies even more depending on whether the data is in the texture cache, or has to be requested from GPU memory.

Tex. Fetch Clocks	Arithmetic Inst	Total Operations	Ops/Inst
1	1	1	1/2
1	10	10	10/11
4	1	4	4/5
4	10	40	40/14

TABLE 4. Computational work vs. execution time

These two aspects dictate some foresight into the nature of the library's usage patterns of the library when deciding which method of storing to employ on the GPU. Depending on the nature of operations being performed on the data, choosing to pack 4 consecutive elements of a vector into the 4-channels of a texture might increase or decrease performance. Consider the work accomplished by performing one arithmetic operation verses 10 arithmetic operations one a 1 and 4 channel

texture shown in Table 4. We see that packing is only advantageous for long kernels using simple operations and relatively few texture lookups. The overhead for texture fetches increases rapidly in the 4-channel case when two or three textures must be fetched to compute the output as in complex per-element kernels and matrix multiplications. Also, since complex instructions can only be executed in the scalar ALU, there is no concurrent execution advantage. The data must be repacked depending on whether the matrix is the first or second operand in the multiplication.

3.2. **Matrix Multiplication.** In our library, matrix multiplication is performed in a similar method to the one described by Larsen and McAllister in [17]. Their implementation was designed to operate on 8-bit fixed point datatypes present on the GeForce 3 using the fixed-function pipeline. In order to extend their basic method to work on the floating-point datatypes available on the ATI 9700 Pro, we must decompose their method into the operations performed and adjust them accordingly to the changes in the hardware.

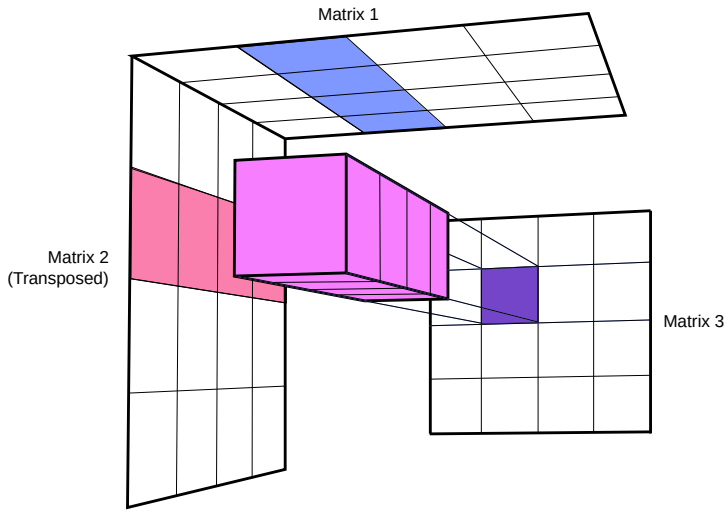

FIGURE 3. Computing an element of a matrix multiplication

Both implementations are based one of the simplest methods of computing a matrix multiplication on a system of distributed processors. The matrix multiplication is visualized as a cube of processors with the three dimensions of the cube identical to the three directions of the matrices being multiplied. The first matrix is lying on the top face of the cube and its values are replicated throughout all of the respective processors. The result is that each horizontal slice of the cube is a copy of the first matrix. Likewise, the second matrix is transposed and distributed across one of the side faces of the cube, so that each vertical slice of the cube is a copy of the second matrix. Now each processor in the cube has a value from both the first matrix and the second matrix. If those values appearing in the same position when viewed from the front of the matrix are multiplied and summed together, the multiplication occurs.

The slices are represented by fullscreen axis-aligned quads with appropriately chosen texture coordinates. When this arrangement is viewed with an orthogonal projection (to ensure alignment of respective elements) and rendered to a buffer with the same dimensions as the output matrix, the GPU can implement the same functionality as the matrix multiplication. All of the quads can be issued in a single render and the GPU just works to complete the computation until it has processed all the quads.

While Larsen and McAllister implemented their work on the fixed-function pipeline and did not explicitly state so, they employed two separate kernels through available functionality to compute the multiplication of two matrices. The element-multiplication kernel performed a texture fetch with the proper coordinates into each of the input matrices and multiplied them together using the `modulate` texture operation. The accumulation kernel summed the results of all the individual element-multiplication kernels using post-fragment blending. Equation 2 is the closest arithmetic description of the method they implement for $C = AB$ for matrices A, B and C.

$$C_{m,p} = \sum_{i=1}^{n} A_{m,i} B_{i,p} \tag{2}$$

This method is not directly applicable to floating point textures in latest generation graphics hardware because it utilizes post-fragment blending to implement the accumulation kernel, which is not currently available for floating point render targets. Accumulation is still possible, but must be done iteratively. The method that is implemented is similar to the method shown in Equation 3 and subsequent steps. The matrix X is of the same dimensions as C and contains the summation up to that step. The element-multiplication and accumulation kernels used previously must be merged into a single multiply-add kernel that performs the multiplication of two elements and adds it to a value that has accumulated all of the element-multiplications up to that point. Each element-multiplication operation is manually accumulated into a temporary buffer, which is then read as a texture to provide the prior work.

$$\begin{aligned} C_{m,p} &= (A_{m,1}B_{1,p}) + (A_{m,2}B_{2,p}) + \cdots + (A_{m,n}B_{n,p}) \qquad (3) \\ \\ X_{m,p} &= 0 \\ C_{m,p} &= X_{m,p} + (A_{m,1}B_{1,p}) + (A_{m,2}B_{2,p}) + \cdots + (A_{m,n}B_{n,p}) \\ C_{m,p} &= X_{m,p} + (A_{m,2}B_{2,p}) + (A_{m,3}B_{3,p}) + \cdots + (A_{m,n}B_{n,p}) \\ &\vdots \\ C_{m,p} &= X_{m,p} + (A_{m,n}B_{n,p}) \end{aligned}$$

This small conceptual change requires moderate alteration of the process of rendering a multiplication in terms of the API usage and impacts performance greatly. First, regardless of any API or storage issues, the fact that floating point textures require 4 times as much space and take 4 times as long to load compared to fixed-point textures is a significant performance hit. Additionally, there are two aspects of the setup that cause performance deterioration: the texture storage representation the API uses when a buffer is being both written and read, and the additional work required by the CPU to guide the rendering process.

Neither the GPU's drivers nor the graphics API expose any means of specifying the storage method of the texture. This is problematic because of the choices the driver makes in regards to fragment buffers acting as both renderable targets and textures. While the *swizzling* of textures on the GPU speeds up read access times, it slows down write access times. When a texture is flagged as a renderable target, the drivers decide to *unswizzle* the texture to speed up writes, and slowing down reads as the bi-directional locality is lost and more cache thrashing occurs. This would not be a concern if there was some way that the user could specify the order in which the fragments being generated by a triangle being rasterized, but that is not the case. For many usage patterns, this is an acceptable decision to be made for the user. For our work, where all textures being fetched are *unswizzled* and that there are many more reads than writes, this configuration is the most pathological condition. The already slow fetches of floating-point textures are further slowed down by the resultant texture cache thrashing due to the *unswizzling* of the textures.

Because API calls don't always match one-to-one with the instruction set of the hardware, every state change incurs a significant overhead due to the drivers having to translate API instructions to driver instructions. Whereas Larsen and McAllister set up all state ahead of time, we are forced to make changes in between the rendering of each slice to perform the accumulation operation through the API. In addition, there is a delay of retransmitting data over the AGP bus, further adding to the overhead of a state change. In order to preform the manual accumulation, two temporary buffers are required. At each step, one buffer serves as the accumulation of previous work, and the other buffer receives the new stage of element-multiplication plus the accumulation of previous work. Then, the purposes of the buffers are switched for the next stage, and the operation is repeated. This requires us to render a given slice and manually change the state regarding the utilization of the temporary buffers before rendering the next slice.

4. Non-Negative Matrix Factorization

Non-negative matrix factorization[19] (NMF) is an iterative machine learning method of identifying parts of a set of data, such as a collection of images to be used for facial recognition. Similar to Principle Component Analysis (PCA), it seeks to create a set of basis functions to describe a set of data. These constraints are modelled after the property that the firing rate of neurons is never negative, and thus unable to change sign. This leads to a parts-based method of identifying local structures in the set of data compared to more traditional approaches like PCA,

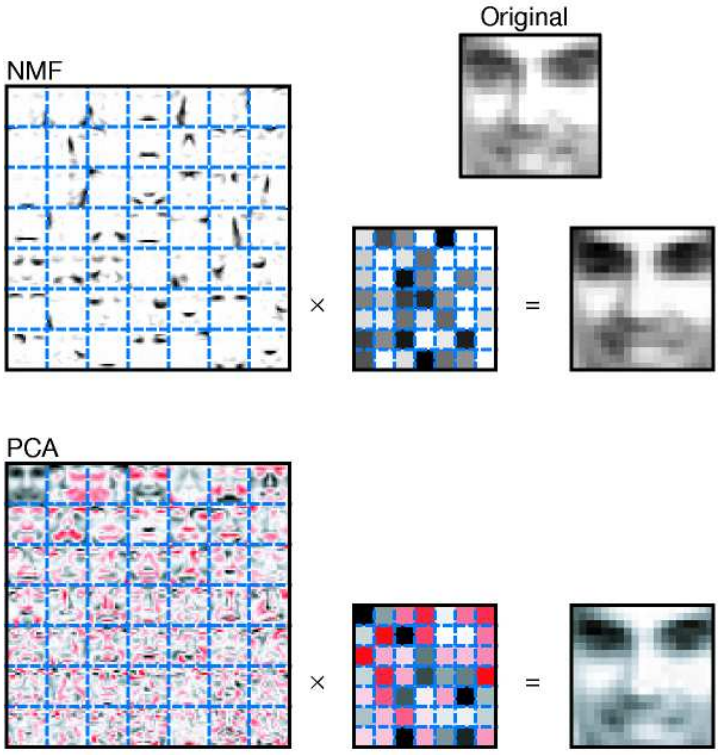

FIGURE 4. Image Representation with NMF and PCA basis functions Both PCA and NMF use the linear combination of their basis functions to approximate the topmost image. Positive values are illustrated by black pixels and negative values are illustrated by red pixels. Image courtesy of Daniel D. Lee and H. Sebastian Seung, **Nature**

whose basis functions operate on a more global level and generate representations best described as "eigenfaces", or distorted versions of whole faces.

To understand better how this method works and how its representation is different from the holistic representation of PCA. The database of images is regarded as an $m \times n$ matrix F, each row contains n non-negative pixel entries of one of the m face images. Both representations construct the approximation of the form

$$\tilde{F} \approx UV^T \tag{4}$$

that is best fit of F in the least squares sense, where U is an $m \times k$ matrix and V is an $n \times k$ matrix. The rank k is chosen such that UV^T is a compressed form of the data in F.

The difference in results is determined by the constraints placed on the matrix factors U and V. PCA constrains the columns of U to be orthonormal and the columns of V to be orthonormal to each other. While this method is statistically interpreted as the directions of the largest variance, the results rarely have an obvious visual interpretation. Because PCA allows the entries of U and V to have arbitrary sign, linear combinations generally involve complex cancellations between positive and negative sections of elements, the individual eigenfaces generated often lack intuitive meaning. NMF does not allow arbitrary sign. As only positive entries are allowed, there is no subtraction performed in the linear combination of elements as in PCA. Each basis may only add something separate to the final image, as no other basis can subtract from what it adds. This is a much closer match to the intuitive notion of combining parts to assemble the whole, and how NMF learns a parts-based representation.

$$U_{ia} \leftarrow U_{ia} \sum_{\mu} \frac{F_{i\mu}}{(UV^T)_{i\mu}} V_{\mu a} \tag{5}$$

$$U_{ia} \leftarrow \frac{U_{ia}}{\sum_j U_{ja}} \tag{6}$$

$$V_{a\mu} \leftarrow V_{a\mu} \sum_{i} \frac{F^T_{i\mu}}{(VU^T)_{i\mu}} U_{ia} \tag{7}$$

There are various sets of update rules that preserve the non-negativity constraint of NMF, minimizing various error metrics. The theory behind choosing one set over another is beyond the scope of this paper, and a full discussion by Lee and Seung can be found in [18]. We choose to implement the conventional metric of reducing least squares error. Equations 5, 6, and 7 describe the iterative process. The three steps are repeatedly applied until UV^T converges to F.

5. Implementation

Our implementation is built around the DirectX 9 API, and targets the ATI 9700 Pro, but is compatible with any current or future piece of hardware that supports floating-point texture formats. The primary goal of the library is to provide the user with an intuitive task-oriented interface that obscures the details of the hardware implementation without sacrificing performance. The main performance issue dependant on API usage as opposed to the underlying hardware is the flow of data to and from the GPU. A standard BLAS interface returns a pointer to the result in system memory. This operation should be avoided whenever possible as GPU readback is currently very slow. Only when the user wants to directly access elements, should the data be read back to system memory. To accomplish this, we model the interface after a standard matrix algebra library, and add specialized element accessors.

5.1. **Interface.** Our interface consists of two objects that wrap collections of DirectX API objects related to data storage and manipulation, the `linAlgShader` and `lasMatrix` objects. They perform the high-level functionality of operator and storage respectively. These operations could be stream-lined further to closer match existing matrix algebra APIs without costing the user any control options, but offer a better interface for observation of the operation of the GPU. Between them, per-element operations and matrix multiplication are supported with the option to transpose all input matrices. Figure 5 contains a short code sample on how to use the API to square a matrix.

```
linAlgShader las;                       // Create controller
PLASMATRIX p;                           // Create matrices
PLASMATRIX q;

las.NewMatrix ( &p );                   // Allocate matrices
las.NewMatrix ( &q );

p->CreateMatrix ( S, S );               // Set size
q->CreateMatrix ( S, S );

for ( k = 0; k < S; k++ )               // Set data for p
    for ( j = 0; j < S; j++ )
        p->_w(k,j) = f[(k*S)+j];

las.Multiply ( &q, &p, &p );            // q = p * p

cout << q->_r(0,0);                     // read and print a value
```

FIGURE 5. Example usage of API

`linAlgShader` is a controller object, responsible for creating and maintaining the overall state of the objects required to render through the graphics API. It creates a rendering context, queries the capabilities of the hardware to decide the best method of rendering and representation, and manages all of the objects such as shaders and vertex data for slices to perform operations. It also maintains a list of all the matrices currently instantiated, as it must be responsible for GPU memory management, evicting unused matrices to system memory to make room for matrices required by the current operation. We design our API with a specialization in iterative algorithms, as the GPU is best at continually processing the same data without CPU intervention. `linAlgShader` caches the intermediate accumulation matrices required for rendering a multiplication to save the overhead cost of creating and deleting these resources every multiply. They exist for the duration of the controller's instantiation, unless room is needed on the GPU, as they are the first objects to be evicted.

`lasMatrix` is a collection of Direct3D objects responsible for representing a matrix stored in textures. It contains the GPU buffer containing the matrix data as well as objects to bind the buffer to be read as a texture and written as a render target. It also contains a system copy of the matrix, and tracks updates to both the CPU and GPU copies, synchronizing between them as needed, and ensuring that

data is not transferred across the AGP bus unless necessary. To aid this process, the object exposes separate read and write accessors to the data, so the synchronization flag is not set unnecessarily.

The native high-level shading language of DirectX 9 is utilized for allowing user definition of per-element functions. Currently, functions are written directly into a file that allows the API to bind high-level variables to the matrices being represented. The DirectX 9 high-level shading language contains much of the same functionality as the Intel VML as well as many simpler arithmetic functions. In a high-level shader function, the elements are loaded through explicit texture fetches, then combined in some set of operations, and return a value. The name of the function is specified when the CPU function to perform the operation is called. Figure 6 contains the high-level code to compute $D = (A * \cos(B))/\sqrt{C}$. The function takes a struct that contains texture coordinates for fetching the respective elements of the vectors. The three elements are explicitly loaded into variables and then the result of operations on them is returned. This method is very crude and separates the GPU code from that of the native CPU code, somewhat breaking the abstraction built into our library, but has the advantage that per-element operations can be changed without requiring the program to be recompiled.

```
float4 PS ( PS_INPUT In ) : COLOR
{
    // read texture
    float A = tex2D ( Matrix1, In.Tex1 ).r;
    float B = tex2D ( Matrix2, In.Tex2 ).r;
    float C = tex2D ( Matrix3, In.Tex3 ).r;

    // pass value through
    return ( (A*cos(B)/sqrt(C) ).rrrr;
}
```

FIGURE 6. Example high-level per-element operation

The semantics of this language are geared towards general graphics operations and expose functionality that could confuse the user if they are not already familiar with the shading language. The `.r` and ṙrrr describing the replication of values across the 4 channels of the native datatypes are used to specify the number of elements to provide the compiler with a better understanding of the desired operation. `.rrrr` is required as the compiler expects a 4-channel output even if the buffer being written to has less channels. The `COLOR` tag specifies that the output will be written to a buffer.

5.2. **Representation.** Currently, we represent matrices as one single texture. This restricts the dimensions of the matrix to those of the maximum texture size for the GPU (2048x2048 texels for 9700 Pro). While blocking schemes that store a matrix in a collection of textures chosen to provide better cache usage yield better results, the asynchronous relation of the GPU and CPU make such methods less desirable for benchmarking operations and identifying bottlenecks of the hardware.

For our implementation, we use only 1-channel textures to store vectors and matrices in our library based on the logic that per-element kernels often require the use of instructions only available in the scalar ALU and matrix multiply kernels are short. We believe this to be the most versatile of packing representations across the entire range of operations available, based on cache usage performance and instruction utilization. Since matrices can be either the first or second matrix in the multiplication, if row or column packing is used, they have to potentially be repacked depending on which order. Methods to manually encode the swizzle of the matrix in the channels of the texture also costs more overall than rendering with one-channels. We find that the overhead to potentially perform this operation even half the time is sufficient to use single-channel textures as they experience the best overall performance.

6. Performance

6.1. **Methodology.** We test the performance of our library with three separate sets of operations. First, we compare per-element function performance to that of the Intel VML. Second, we compare single matrix multiplication to that of the Intel BLAS library. Finally, we compare implementations on non-negative matrix factorization using our library and the Intel Math Kernel Library.

We do not discuss the implications of the time to transfer data to and from the GPU over the AGP bus. Our implementation focuses on providing support for iterative algorithms and makes the assumption that once the data has been moved to the GPU, the amount of computation will be significant enough that the transfer time over the AGP bus will be small in comparison. Data is transferred over the bus only as requested by the user, so any iterative algorithm implemented through the library would only transfer data to the GPU at the beginning and retrieve it at the end. Also, as we strive to provide a means of connecting separate specialized computational kernels on the GPU through a set of general purpose routines, the goal is to keep the data on the GPU for as much of the computation as possible.

6.2. **Vector Math.** For testing vector math performance we benchmark the performance of implementations of all of the functions provided in the Intel VML against the native functions over a range of vector sizes. Revising the predictions of Equation 1 to take into account our implementation details, such as 1 channel textures, we have

$$1_{floats/pixel} * 8_{pixels/clock} * 350Mhz = 2.8GFlops$$

potential performance. Considering the simplest per-element operation involves one arithmetic operation and one texture fetch to provide data for it, the realistic performance maximum would be half that value: 1.4 GFlops.

A summary of the average performance for each function is included in Table 5. The MFlops reported include the dual fragment rasterization overhead as well as

Function	CPU MFlops	GPU MFlops	Ratio	Count
InvSqrt	92.07	1245.19	13.53	3
Inv	136.28	1258.83	9.24	3
Log10	98.52	1003.73	10.19	4
Exp	63.39	1021.16	16.11	4
Div	102.41	729.11	7.12	5
Sqrt	101.04	729.15	7.22	5
Pow	3.54	669.58	189.21	6
Sinh	2.01	394.74	196.11	8
Cosh	43.74	391.10	8.94	8
Tanh	37.70	300.45	7.97	10
Cos	54.40	269.07	4.95	11
Sin	55.74	268.34	4.82	11
Tan	35.68	190.04	5.33	15
Acos	0.72	189.73	263.21	16
Asin	0.63	176.28	281.30	16
Atan	27.09	100.82	3.72	28
Atan2	23.51	87.42	3.72	32

TABLE 5. Vector math performance of CPU and GPU

CPU overhead. The dual fragment occurs because the two triangles that comprise the quad for rendering the operation both generate fragments for pixels that overlap that edge, so $WH + \max(W, H)$ operations occur to process WH elements. CPU overhead is a result of setting all of the API state and rendering the per-element function. The rightmost column shows the instruction counts for per-element functions. The simplest functions, `Inv`, `InvSqrt`, `Exp`, and `Log10`, all come quite close to the maximum possible performance of the GPU for our current implementation. The observed performances are closely correlated to the number of instructions required to perform the operation. The stream processing model maximizes the cache usage and effectively hides the memory hierarchy, ensuring the amortized time of a texture fetch is one clock.

While the CPU overhead to render a function is small, usually on the order of 0.0003 seconds, it makes a noticeable impact on the observed performance. In terms of performance in MFlops, the CPU overhead is uniformly distributed across all of the elements on which it operates, so the number of elements must be large enough such that the CPU overhead per element is much smaller than the amount of time required to compute the result for that element. Figure 7 displays the performance of CPU and GPU in regards to the number of elements operated upon. As expected the CPU performance is nearly constant over any number of elements. GPU performance, on the hand, starts out near zero, increases rapidly, then levels off. The CPU overhead becomes sufficiently small with respect to the time to perform the operation that it is effectively zero, occurring at approximately 5×10^5 elements.

While somewhat noticeable in unary functions of Figure 7, the outliers in binary functions are very obvious. While the same outliers are consistent between operations and over multiple sets of similar operations, they change in regards to the

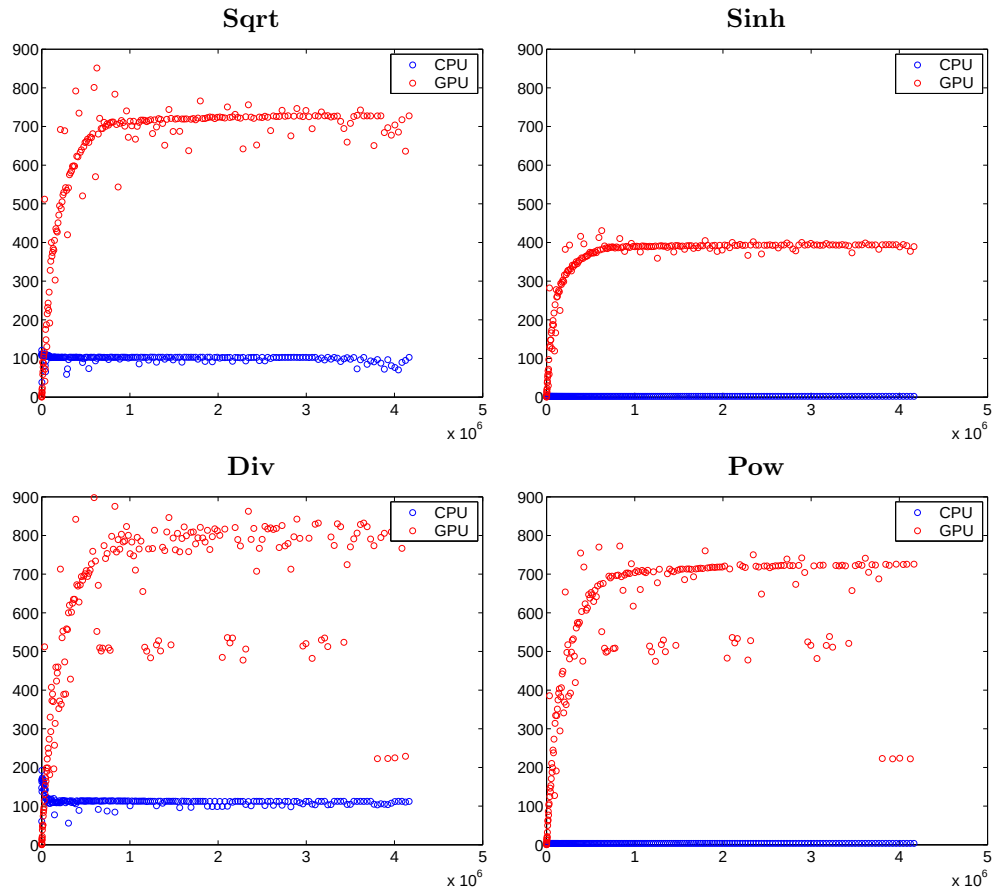

FIGURE 7. Comparison of CPU and GPU performance
The top row contains results for two unary per-element functions, (a) `Sqrt`, (b) `Sinh`. The bottom row contains results for two binary per-element functions, (c) `Div` and (d) `Pow`.

ordering of operations. While the `linAlgShader` object handles high level memory management, it plays no roll in determining the layout of textures and buffers on the GPU. Depending on the order that objects are created and destroyed, the texture fetch performance varies greatly. Depending on the size of textures, performing the same sequence of operations in forward and reverse, can alter the performance by almost a factor of 2. This is further complicated by the binary functions shown in Figure 7, as they require fetches to two separate texture to perform the given operation and cause a greater percentage of cache misses. Comparing `Div` and `Pow` from Figure 7, we see that the same outliers appear in both, but are somewhat less divergent in `Pow`. Looking back on the number of instructions we see that `Div` takes 5 instructions to complete, while `Pow` takes 6. The difference in texture fetch times decreases as the ratio of arithmetic to texture fetch instructions increases. While for any algorithms that can fit all data and temporary matrices and most other

usage patterns, the performance matches the trend, we include this to emphasize that intelligent memory allocation patterns are still beneficial.

The asynchronous nature of the GPU makes the task of benchmarking performance, or simply determining how long an operation took to complete, a complicated task. We have verified our results for per-element functions in several ways. The more straightforward method is the utilization of a DirectX 9 Query object to request from the graphics card how many pixels were rendered from a given set of API calls. The card keeps track of the rasterization process and returns the number upon completion of the task. Thus, the time to execute a given operation is the time to render with the query minus the time to use a query without rendering any geometry. This makes assumptions in how the driver operates, and while it yields accurate results, is not guaranteed to do so.

The other method is based on some assumptions of the architecture of the hardware. Even if a function call returns immediately, the GPU still has to work for some period of time to complete the requested operation. Taking advantage of the pipeline architecture of the hardware, we can selectively create bottlenecks to determine performance. We need to arrange our task in a way that the bottleneck occurs in the part of the pipeline we wish to benchmark. In this case, that fragment shader is the obvious bottleneck as only 4 vertices are needed for a full-screen quad that will generate thousands of fragments. Below a certain point, the overhead of the API takes more time than the operation. As the rate and magnitude of operations issued increases, so does the apparent performance of the GPU until it reaches the point where the bottleneck is no longer the CPU, but the fragment shader to perform the computation. We can divide the time to issue a render by the number of operations completed in that span of time to get the performance of the GPU. This asynchronous operation has a major benefit in the fact that it enables the creation of semaphores to allow the CPU to continue working on other computations concurrent to the GPU. While the current library does not offer semaphore functionality, if being used interactive at a (relatively) slow rate such as by a human, the library appears to perform computations instantaneously because functions return at once so.

6.3. **Matrix Multiplication.** Conceptually, a matrix multiplication is nothing more than a per-element function that multiplies two numbers and adds them to a third repeatedly. The difference is that the texture fetches required for each step of a matrix multiply are not as cache-friendly as those used by the vector math function. We implement matrix multiplication as the application of z per-element functions on input matrices of $x \times z$ and $z \times y$. In addition to the performance considerations mentioned in regards to per-element operations, matrix multiplication has more details that affect the observed performance.

The primary goal of the stream computing model is to hide cache latency. There are a collection of reasons why matrix multiplication either breaks this model, or adversely affects performance. The fact that the buffer can be used as a render target currently requires the texture to be stored unswizzled, causing texture fetches to slow down and the bi-directional locality to be lost. Matrix multiplication requires

$3n^3$ reads and n^3 writes in it most naive form when the manual accumulation is included. As mentioned in Section 3.2, the method of rendering the multiply requires that one of the matrices be transposed, meaning that one of the textures will be accessed down one column at a time while its cache lines only extend sideways, so it will have a cache miss when it fetches a texel.

There is also the added performance concern that the CPU has to manually perform the state changes of the accumulation as opposed to having the post-fragment blending functionality perform them. This requires more room on the GPU, causes more cache misses, and requires significant CPU work. The CPU must change the state and render the next step, which requires considerable work for each message sent to the GPU. The CPU is tied up most of the time managing the state of multiply render when it should be free to work on other tasks.

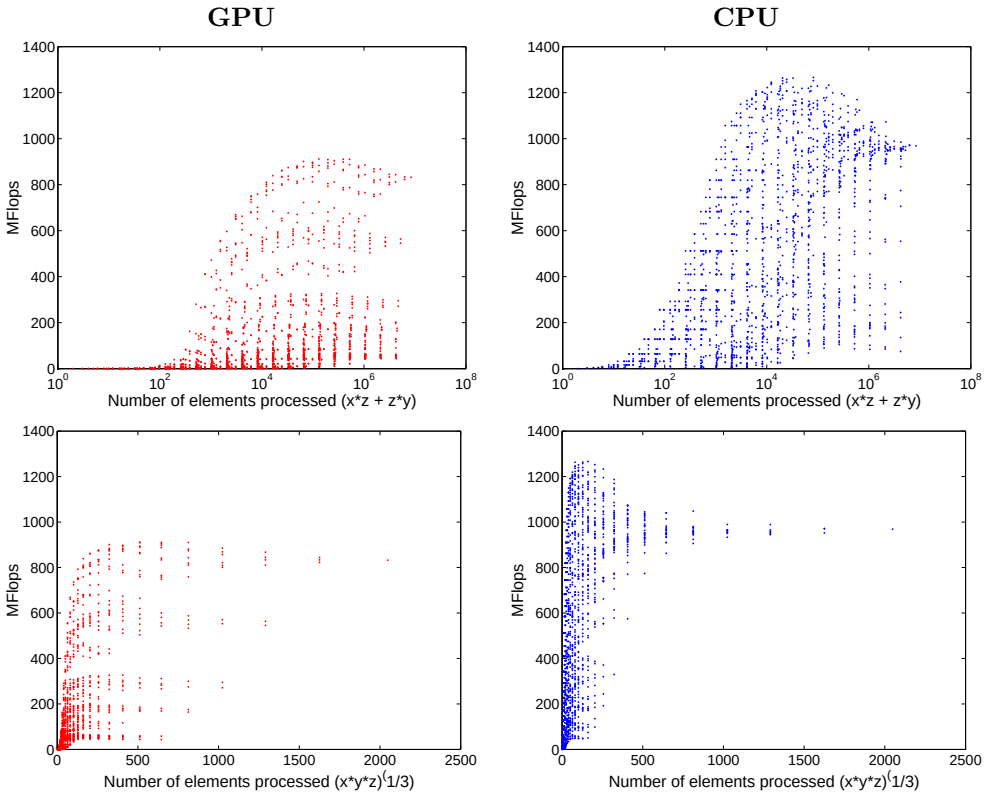

FIGURE 8. CPU and GPU matrix multiplication performance
The results are displayed in two formats, with the top row showing (a) GPU and (b) CPU performance as a variable of the number of input elements, and the bottom row showing (c) GPU and (d) CPU performance as a variable of total computational. *Note the logarithmic scaling of the x-axis in (a) and (b).*

There are several means of alleviating these problems. The first is to render less passes with more texture fetches per pass. While this results in longer shaders, the

accumulation texture fetch only occurs once per shader, so its cost is amortized across all of the texels rasterized that step. With less passes to be rendered, the CPU would be free to perform other tasks. A more ideal solution would be possible if the swizzle of the buffers used as textures and rendertargets was user-controllable. All of the user matrices would be swizzled for fast reads, and all of the temporary textures would be unswizzled for fast writes. If more than one set of texels was being fetched, the accumulation texture could be fetched first and other fetches could occur until all of the data is there, effectively hiding the longer read time of an unswizzled texture, but still being able to perform a fast write to the buffer. Then, completing the render with a render to the destination buffer. These variations are all attempts at reducing the constant factors regarding various aspects of the process. A more robust solution would include altering the data representation and method of rendering, which will be discussed in detail later.

Figure 8 compares performance on matrices of dimensions $x \times z$ and $z \times y$ for all combinations of powers of two in the possible range of texture sizes, $x, y, z = 2^n : n \in [1, 2048]$. The set of data is very complex with large changes in the relative performance of CPU and GPU. Yet, several key differences between implementations can be observed.

First, the GPU requires an order of magnitude more input elements than the CPU before it begins to exhibit noticeable performance. This is due to the setup work required to validate the data and perform all the necessary state setup. This is very similar to the behavior observed with vector math functions, where performance was low until a sufficiently large number of operations occurred to absorb the overhead work.

Second, we can see from the distribution of values, that at larger numbers of elements; the CPU performance converges to a singular value, but the GPU exhibits several strata of performance levels. These correspond to different ratios of oblong matrices. This is a result from the number of cache misses for a given ratio of input matrix dimensions. The access pattern of the naive multiplication implementation results in good cache usage accessing the first matrix as it is read row-wise, same as the data alignment, resulting in $\frac{n-1}{n}$ cache hit ratio for n texels loaded per memory access. The access pattern for the second matrix yields poor cache usage as it is read column-wise, resulting in $\frac{1}{n}$ cache hit ratio. As x or y increases in relation to the other, the number of cache misses increases also. This causes the observed performance of combinations of dimensions resulting in the same total number of computations to be sorted based on how close the ratio of $x : y$ is to $1 : 1$.

Third, due to the overhead of state switching required to perform a multiplication, increases in z have a larger effect on performance than changes in x and y. Changing x or y by a factor of 2 results in a factor of 2 change in performance, while changing z by a factor of two results in a factor of 4 change in performance.

All of these considerations can been observed in the following example, the GPU has a $6.8\times$ the performance of the CPU when a 2048×2 by 2×2048 multiplication is performed compared to $.08\times$ the performance of the CPU when a 2×2048 by

2048×2048 multiplication is performed compared to $1.49\times$ the performance of the CPU when a 2048×2048 by 2048×2 multiplication is performed.

Kruger and Westermann[16] provide a library with similar functionality to that of our own. They differ on how to represent matrices, opting to store an $N \times N$ matrix Q as N separate textures each containing one of the diagonals of Q. Each of the diagonals is blocked into a square texture to improve cache locality. This works well for the sparse case because they can simply not store or render empty diagonals, but requires them to perform N state changes for a dense matrix. While they do not implement matrix-matrix multiplication, the implications of such a storage scheme can be observed when comparing times. For a 4096×4096 matrix-vector multiply, their library took .23 seconds. Our implementation performed the same operation in .35 seconds when the shared dimension of the matrices was used and 4096 state changes were made. It makes sense that our performance be lower since we store a vector a long texture, so the cache locality on accessing that is poor. On the other hand, we achieved a time of .022 seconds for the case requiring only one state change. This shows that state changes to take up a considerable amount of the total rendering time, and should be used only when they offer the possibility to save work other places, such as in the sparse case.

6.4. **Non-Negative Matrix Factorization.** The process below identifies the actual steps implemented in both on the CPU and on our GPU library to perform non-negative matrix factorization. It performs the exact same steps as Equations 5, 6, 7, but it written in a more iterated form.

Initialize $\mathbf{U} \leftarrow m \times k$ matrix of strictly positive random entries
Initialize $\mathbf{V} \leftarrow n \times k$ matrix of strictly positive random entries

repeat

$$
\begin{aligned}
\mathbf{U}_{n_1} &= [u_{n_1}(i,j)] \leftarrow \mathbf{FU} \\
\mathbf{U}_{n_2} &= [u_{n_2}(i,j)] \leftarrow \mathbf{UV^T U} \\
\mathbf{U}_{n} &= [u_{n}(i,j)] \leftarrow \left[\frac{u_{n_1}(i,j)}{u_{n_2}(i,j)}\right] \\
\mathbf{V}_{n_1} &= [v_{n_1}(i,j)] \leftarrow \mathbf{F^T U} \\
\mathbf{V}_{n_2} &= [v_{n_2}(i,j)] \leftarrow \mathbf{VU^T U} \\
\mathbf{V}_{n} &= [v_{n}(i,j)] \leftarrow \left[\frac{v_{n_1}(i,j)}{v_{n_2}(i,j)}\right] \\
\mathbf{U} &= [u(i,j)] \leftarrow [u_n(i,j) \times u_n(i,j)] \\
\mathbf{V} &= [v(i,j)] \leftarrow [v_n(i,j) \times v_n(i,j)] \\
\mathbf{U} &= [u(i,j)] \leftarrow \left[\frac{u(i,j)}{\sum_{r=1}^{M} u(r,j)}\right]
\end{aligned}
$$

until U and V converge

FIGURE 9. Iterative NMF algorithm

Complex repeating operations such as these are the rationale for caching temporary matrices. NMF must run many times and there is no need to create and destroy temporary objects used for a specific step if they can be saved and used

for the next step. Every texture in an iterative step is either $M \times K$, $N \times K$, or $M \times N$ with the exception of a single $K \times K$. The same step can use a cached matrix every iteration, as well as multiple steps of the same iteration sharing the same cached matrices. Caching to reduce overhead also works to prevent the observed fragmentation in GPU memory that was adversely affecting vector math performance.

To test performance, we chose a subset of one of several publicly available collections of faces for evaluating facial recognition algorithms. M,N,K were chosen to be 361,1024,144, which is to say we have 1024 images, each with 361 pixels (a resolution of 19×19) and seek to represent the subset we chose using 144 basis functions, each of 19×19 pixels.

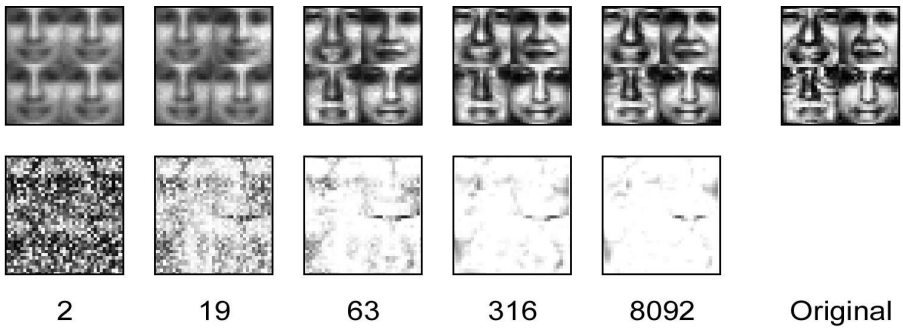

FIGURE 10. Evolution of UV^T approximation of images
The top images of 4 faces, with exception of the rightmost image, are linear combination of that part of UV^T that contains those faces after the given number of intervals. The bottom images are 4 adjacent basis images and how they evolve over time from random noise to representing specific details.

The observed performance over $10,000$ iterations was that the CPU implementation using Intel MKL libraries took 162 ms per iteration totaling 1606.8 secs, while our library took 100 ms per iteration totaling 994.2 secs. The time to execute an iteration remained constant for both CPU and GPU implementations as the workload did not vary from iteration to iteration. Figure 10 depicts the progression of some of the basis matrices and the approximation yields compared with the original of those faces. Even though our library implementation is more sensitive to changes in access patterns than more complicated representations and often likely to perform worse than the CPU, we show that it is still possible to obtain respectable results.

7. Discussion

We summarize our findings in a set of succinct evaluations of the current state of graphics and hardware and suggestions and guidelines for improving upon it.

7.1. **Representation.** We specifically chose the simplest representation of matrices for our library for straightforward identification of performance bottlenecks of the hardware, fully aware of the bottlenecks that would result from that choice. This was especially pertinent with regards to the implications of the representation on texture fetch cache misses. The primary goal of the stream processor is to hide the memory hierarchy, and whereever it fails at this goal the performance gains over a standard computing model suffer.

The ideal choice for a representation would ensure both good cache usage for all input matrices and require as little intervention from the CPU as possible. Unfortunately in practice these two are mutually exclusive as representations of matrices with better cache coherency usually rely on blocking the data into sub-matrices. So, if any given matrix is represented by N sub-matrices, a matrix multiplication will require on the order of N^2 instructions by the CPU to perform the multiplication. While no one solution can handle all cases, it appears that increasing CPU cycles to lessen GPU memory bandwidth requirements is the better performing choice over the majority of tests. Also, packing the diagonals of matrix into a set of sub-matrices seems to be quite effective in terms of cache utilization.

If the decision has been made to block matrices to increase cache performance, then it is only logical to choose methods for other parts of the multiply operation that can alleviate more bottlenecks. Divide-and-conquer algorithms are fundamentally based on the separation of large chunks of data into smaller portions. Many matrix-multiplication algorithms exist that have a lower asymptotic running time than $O(n^3)$, but the best known is Strassen's Algorithm. It starts with the idea that two $N \times N$ matrices are multiplied by splitting them each into 4 parts, multiplying the respective parts, and recursing the 4-way split onto each of those multiplications. The results are summed together by N^2 additions. This yields a recurrence equation of

$$T(n) = 8T(n/2) + \Theta(n^2) \tag{8}$$

which has the solution of $O(n^3)$, no better than the naive implementation already in place. What Strassen discovered was there a way to accomplish the same thing using more additions but only 7 recursive multiplications, yielding the recurrence

$$\begin{aligned} T(n) &= 7T(n/2) + \Theta(n^2) \\ T(n) &= \Theta(n^{\log 7}) \\ T(n) &= O(n^{2.81}) \end{aligned} \tag{9}$$

We omit the details of how this is accomplished, but will say that through some rearrangement of terms, we can trade a matrix multiplication for several vector addition and subtraction operations which are much less expensive than a full $(n/2) \times (n/2)$ multiplication.

The bottom line is that there are methods which offer substantial performance gains over our implementation. It was never our aim to provide the implementation offering the best performance, merely demonstrate that the GPU is robust enough to offer performance benefits over the CPU without relying on it for all of the computation of connecting various kernels.

7.2. **CPU vs. GPU.** The question all of this comes down to, "Is it worth performing computation on the GPU?" One could argue that there is currently less precision available, less control over the internal workings, and a only a small subset of operations which are currently applicable on graphics hardware. And should one choose to use graphics hardware, there is only a marginal benefit in performance for what will probably be an ordeal to port code.

Some might argue, that at this time, the answer to implementing general computation on graphics hardware does not provide the performance gains to justify the large switch in current computational methods. While this may be true, few will argue that it will remain true for long. The *stream processor* model fuels the Moore's Law Cubed increase in performance touted by every graphics card manufacturer. As silicon space doubles, a stream processor can double the number of parallel pipelines and get double performance while a standard CPU can add more cache and get a slight 15% speed increase. While the GPU and CPU are roughly comparable in performance at this moment for most kernels, in a year graphics hardware will be $4\times$ faster compared to CPU's which will be $1.6\times$ faster. This disparity of performance will continue to increase, and as graphics hardware gains more functionality the implementation of other algorithms on the GPU will not only be feasible, but desirable.

7.3. **Future Work.** Future work stemming from this research can grouped into two main categories: expanding the functionality of the matrix library, and developing better interfaces for use in programs that use graphics hardware for general computation.

There are many avenues down which we can expand the functionality of the matrix library. A better storage representation would be a starting point using a divide and conquer multiplication step. The high-level shading language elements can be better integrated into the API. There is also a need for optimized support for sparse matrices and other specific forms. Gather and scatter operators can be implemented through dependent texture reads. An unforseen problem arises if the library is used in an interactive setting such as a plug-in for a mathematical computation package. During heavy matrix multiplication loads, the workstation would be begin to lag. Our first thought was that the resource management for issuing state changes were consuming all of the CPU's power. As it turns out, the lag appears to be caused by the matrix using the entirety of the memory bandwidth on GPU memory thus significantly slowing down the rate of blitting data to the screen. Something akin to thread priority in an operating system kernel needs to be included in the library otherwise it will consume all bandwidth to the detriment of the user.

The interfaces through which to program the GPU are fairly inadequate for our purposes. Graphics APIs such as DirectX and OpenGL perform the desired task, but require a great deal of abstraction. There needs to be a new kind of API, not designed for graphics but for general computation that can be extended for graphics. The API and drivers assume that all use of graphics hardware will be for real-time graphics. While simplifing the assumptions of developers, it makes certain operations using graphics hardware more obfuscated. While adding OpenGL extensions specifically designed for easier general computation would be a plus, we still think it is missing the big picture. The GPU is no longer a piece of equipment merely for generating imagery. It has the opportunity to become a full-fledged co-processor alongside the CPU, and should be programmed as such. Implementing general libraries on top of existing graphics APIs is more problematic in the long run. It makes more sense to think of a texture as a kind of array, as opposed to an array as a kind of texture.

The difficulty in benchmarking specific features is due to: the asynchronous operation of the GPU and the rapid rate at which new hardware is released. Together these issues make detailed material concerning performance aspects of graphics hardware is almost non-existent. What the real-time graphics community as a whole needs is a utility that can automatically benchmark a piece of graphics hardware. This would be different from any multimedia benchmarking utility currently available in that it would not try to assign a number based on performance of rendering a scene, but would try to create a relational table mapping between API flags and hardware performance by specifically generating cases to cause bottlenecks to map the GPU architecture. Therefore once a programmer had run a utility on a piece of hardware, they could see the effects various changes had in the performance of the card and eliminate bottlenecks. This would give programmers the information they need to optimize for hardware; information they would otherwise have no means of obtaining about other than by trial and error.

8. Conclusion

In this work, we have described a simple framework for the implementation of numerical algorithms on graphics hardware. Forgoing one of several potentially more efficient ways to represent data when stored and operate upon it, we chose a simple method as a means of observing the performance displayed under various sets of circumstance without the need to sort through the idiosyncracies of a more complicated implementation.

It is plain to see that while the GPU is a streaming processor, the fragment shader's ability to perform random look into textures breaks the architecture to a degree. The goal of a streaming processor is to hide the memory hierarchy and depending on the access patterns performed by fragment shader. This is more of concern to the user now that dependant texture reads are possible and any value can be issued at the coordinates for a textures fetch. The new-found freedom in what new computation options are available comes at the cost of needing to be more cautious about the performance ramifications they carry.

Even though high precision floating point available in the shader pipelines and storage are a new addition to commodity graphics hardware, they have opened up many new means in which to utilize the hardware. There is a lack of much publicly available information concerning the design specifics of modern graphics processors, and even less information explaining how a chain of events can cause the performance bottlenecks we observed. While we have gained much insight in regards to the internal workings of the GPU and how it relates to the API with regards to fragment shades, texturing, and memory access, we have must gain experience with new cards through trial an error due to the lack of a a clear and logical approach for gaining such an experience.

References

1. Anthony A. Apodaca and M. W. Mantle, *Renderman: Pursuing the future of graphics*, IEEE Computer Graphics & Applications **10** (1990), no. 4, 44–49.
2. Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schrder, *The gpu as numerical simulation engine*, SIGGRAPH 2003 Conference Proceedings, Annual Conference Series, ACM Press/ACM SIGGRAPH, 2003.
3. Cem Cebenoyan and Matthias Wloka, *Optimizing the graphics pipeline*, Presentation, nVidia, March 2003.
4. Wei-Chao Chen, Jean-Yves Bouguet, Michael H. Chu, and Radek Grzeszczuk, *Light field mapping: Efficient representation and hardware rendering of surface light fields*, SIGGRAPH 2002 Conference Proceedings (John Hughes, ed.), Annual Conference Series, ACM Press/ACM SIGGRAPH, 2002, pp. 447–456.
5. U. Diewald, T. Preusser, M. Rumpf, and R. Strzodka, *Diffusion models and their accelerated solution in computer vision applications*, 2001.
6. Matteo Frigo and Steven G. Johnson, *FFTW: An adaptive software architecture for the FFT*, Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing (Seattle, WA), vol. 3, May 1998, pp. 1381–1384.
7. David Guillamet and Jordi Vitria, *Classifying faces with non-negative matrix factorization.*
8. Mark J. Harris, Greg Coombe, Thorsten Scheuermann, and Anselmo Lastra, *Physically-based visual simulation on graphics hardware*, Proceedings of the conference on Graphics hardware 2002, Eurographics Association, 2002, pp. 109–118.
9. John C. Hart, *Perlin noise pixel shaders*, Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on on Graphics hardware, ACM Press, 2001, pp. 87–94.
10. Kenneth E. Hoff III, John Keyser, Ming Lin, Dinesh Manocha, and Tim Culver, *Fast computation of generalized Voronoi diagrams using graphics hardware*, Computer Graphics **33** (1999), no. Annual Conference Series, 277–286.
11. M. Hopf and T. Ertl, *Hardware accelerated wavelet transformations*, 2000.
12. Matthias Hopf and Thomas Ertl, *Accelerating 3D convolution using graphics hardware*, IEEE Visualization '99 (San Francisco) (David Ebert, Markus Gross, and Bernd Hamann, eds.), 1999, pp. 471–474.
13. H. Igehy, M. Eldridge, and K. Proudfoot, *Prefetching in a texture cache architecture*, Proceedings of the 1998 SIGGRAPH / Eurographics Workshop on Graphics Hardware, 1998, pp. 133–142.
14. Intel, *Intel math kernel library reference manual*, 2001.
15. Doug L. James and Dinesh K. Pai, *Dyrt: Dynamic response textures for real time deformation simulation with graphics hardware.*
16. Jens Kruger and Rdiger Westermann, *Linear algebra operators for gpu implementation of numerical algorithms*, SIGGRAPH 2003 Conference Proceedings, Annual Conference Series, ACM Press/ACM SIGGRAPH, 2003.
17. E. Scott Larsen and David McAllister, *Fast matrix multiplies using graphics hardware*, Supercomputing, 2001.
18. Daniel Lee and H. Sebastian Seung, *Algorithms for non-negative matrix factorization*, NIPS, 2000, pp. 556–562.
19. Daniel D. Lee and H. Sebastian Seung, *Learning the parts of objects by non-negative matrix factorization*, Nature **401** (1999), 788–791.
20. Aaron Lefohn and Ross Whitaker, *A gpu-based, three-dimensional level set solver with curvature flow*, Technical report, University of Utah, December 2002.
21. Wei Li, Xiaoming Wei, and Arie Kaufman, *Implementing lattice boltzmann computation on graphics hardware*, The Visual Computer (2001).
22. Erik Lindholm, Mark J. Kligard, and Henry Moreton, *A user-programmable vertex engine*, Proceedings of the 28th annual conference on Computer graphics and interactive techniques, ACM Press, 2001, pp. 149–158.
23. Dinesh Manocha, *Interactive geometric computations using graphics hardware*, ACM SIGGRAPH, 2002.
24. William R. Mark and Kekoa Proudfoot, *The f-buffer: A rasterization-order fifo buffer for multi-pass rendering.*

25. M. McCool, *Smash: A next-generation api for programmable graphics accelerators*, 2000.
26. Michael D. McCool, Zheng Qin, and Tiberiu S. Popa, *Shader metaprogramming*, Proceedings of the conference on Graphics hardware 2002, Eurographics Association, 2002, pp. 57–68.
27. Microsoft, *Directx 9 documentation*, November 2002.
28. nVidia, *Cg toolkit user's manual*, 1.1 ed., February 2003.
29. Mark Olano, *Real-time shading languages*, ACM SIGGRAPH, 2002.
30. John D. Owens, William J. Dally, Ujval J. Kapasi, Scott Rixner, Peter Mattson, and Ben Mowery, *Polygon rendering on a stream architecture*, 2000 SIGGRAPH / Eurographics Workshop on Graphics Hardware (2000), 23–32.
31. Mark S. Peercy, Marc Olano, John Airey, and P. Jeffrey Ungar, *Interactive multi-pass programmable shading*, Proceedings of the 27th annual conference on Computer graphics and interactive techniques, ACM Press/Addison-Wesley Publishing Co., 2000, pp. 425–432.
32. Kekoa Proudfoot, William R. Mark, and Pat Hanrahan, *A real-time procedural shading system for programmable graphics hardware*, Proceedings of the 28th annual conference on Computer graphics and interactive techniques, ACM Press, 2001, pp. 159–170.
33. Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan, *Ray tracing on programmable graphics hardware*, Proceedings of the 29th annual conference on Computer graphics and interactive techniques, ACM Press, 2002, pp. 703–712.
34. Guennadi Riguer, *Performance optimization techniques for ati graphics hardware with directx 9.0*, Tech. report, ATI, March 2003.
35. Randi Rost, *Opengl shading language*, Presentation, 3DLabs, March 2003.
36. M. Rumpf and R. Strzodka, *LEVEL SET SEGMENTATION IN GRAPHICS HARDWARE*, pp. 1103–1106.
37. Martin Rumpf and Robert Strzodka, *Nonlinear diffusion in graphics hardware*, pp. 75–84.
38. R. Strzodka, *Virtual 16 bit precise operations on rgba8 textures*.
39. Chris J. Thompson, Sahngyun Hahn, and Mark Oskin, *Using modern graphics architectures for general-purpose computing: A framework and analysis*, International Symposium on Microarchitecture (MICRO), 2002.
40. C. Trendall and A. Stewart, *General calculations using graphics hardware with applications to interactive caustics*, 2000.
41. Daniel Weiskopf, Matthias Hopf, and Thomas Ertl, *Hardware-accelerated visualization of time-varying 2d and 3d vector fields by texture advection via programmable per-pixel operations*.

1322 S. Negley Ave., Pittsburgh, PA 15217

E-mail address: mmt@cmu.edu