The GPU as a Scientific Computing Engine: Performance and Analysis

Navendu Jain[*]
Department of Computer Sciences
Univ. of Texas at Austin
Austin, USA 78751–0250

Jason Chaw[†]
Department of Computer Sciences
Univ. of Texas at Austin
Austin, USA 78751–0250

Abstract

The evolution of the modern day graphics hardware has exploited the data parallelism and high computational nature inherent in graphics applications. With the advent of programmability for commodity graphics processing units (GPUs), a whole new paradigm of harnessing the vast GPU resources towards performing general-purpose computations has appeared. In this paper, we aim at providing the complete functionality available in the BLAS (Basic Linear Algebra Subprograms) library, thus allowing general linear algebra packages to be built upon GPU implementations. We propose a programming framework used in porting the BLAS library and demonstrate the mapping of fundamental linear algebra operators on the graphics hardware. We demonstrate the effectiveness of our approach and analyze the performance bottlenecks. Our experiments reveal that the existing graphics architectures are bandwidth limited. Finally, we propose extensions to current graphics hardware which would improve it's effectiveness for general purpose computations.

Keywords: Graphics Hardware, Scientific Computing, BLAS, Cg, OpenGL, NV30, Vertex Processor, Fragment Processor

1 Introduction

In recent years, there has been a dramatic increase in the computational power and memory bandwidth of graphics processing units (GPUs). For example the NVIDIA GeForce4 was advertised to perform nearly one trillion operations per second and support a memory bandwidth of 10.4 GB/sec. They are now capable co-processors, and their highly parallel nature makes them a high speed computing engine for a variety of applications, traditionally implemented on general purpose CPUs. With the advent of programmability, the scientific community is aiming to utilize the resources of these high performance processing units for general-purpose numerical computations. For example, recent approaches have demonstrated using the GPU's vertex processor and the fragment processor in a vector [Thompson et al. 2002] and a stream [Buck and Hanrahan 2003; Bolz et al. 2003; Kruger and Westermann 2003] programming model respectively. Other emerging applications include numerical simulations [Bolz et al. 2003], classic computer science problems like Fast Fourier Transformation, sorting [Buck and Hanrahan 2003], and simulation of physical phenomena [Harris et al. 2002].

Many applications in the scientific computing domain can achieve a significant performance gain by off-loading some or all of their numerical computations onto the GPU, thereby freeing the CPU to perform other tasks.

In general, the core of a scientific application can be decomposed into a set of linear algebra computations. Between applications, there exists a high component similarity and reuse. Scientific applications also require tremendous computing power and are often highly optimized for a target machine. From these, we notice a trend where scientific application developers favour using some highly optimized linear algebra library such as BLAS.

We foresee the advent of more powerful and flexible GPUs in the future allowing developers to implement novel rendering pipelines. Such development would remove many of the current obstacles in implementing general purpose software on today's GPUs.

Our work makes the following contributions:

- A prototype BLAS library implementation on the NVIDIA NV30 architecture GPU which forms the core building block for many important software packages in the realm of scientific computing.
- Proposes a cascading loop-back model as a mechanism for chaining multiple GPU-based function calls efficiently on the GPU. As discussed later in this paper, for a single run through the graphics pipeline, the setup times are dominant compared to the actual execution time. Keeping intermediate results in the GPU memory, the complex scientific applications could be efficiently modeled as multiple computation phases through the pipeline.
- Identifying the architectural bottlenecks of existing GPUs and proposing enhancements so that running scientific applications on the GPU can benefit the most in terms of overall execution times and numerical accuracy.

The rest of this paper is organized as follows. Section 2 gives an overview of modern programmable graphics hardware and the related work in this area. In Section 3, we explain the framework for BLAS programming on the GPU. The execution models for basic vector-vector operations are presented. These are subsequently used to built more complex matrix-vector and matrix-matrix operations. We also introduce the cascading loop-back model of computation resulting in increased utilization of the graphics pipeline. The experimental setup and the results are evaluated in Section 4. Based on our evaluation, we discuss the potential bottlenecks and propose enhancements for future graphics architectures in Section 5. Finally, Section 6 concludes the paper.

2 Background and Prior Work

2.1 Graphics Hardware

Figure 1 shows the graphics pipeline and the programmable processors in the modern graphics architecture. Both the vertex pro-

[*]e-mail: nav@cs.utexas.edu
[†]e-mail: jchaw@cs.utexas.edu

cessor (VP) and the fragment processor (FP) can do 4-wide SIMD floating point arithmetic. This is well supported by the register set which can hold quad-valued floating point values e.g. *xyzw* for position and *rgba* for color. The latest graphics card from ATI (9800) supports 24-bit arithmetic whereas the NVIDIA GeForce NV30 architecture supports fixed-point (8 bit), half-float (16 bit) and single-precision IEEE floating point (32 bit) data types.

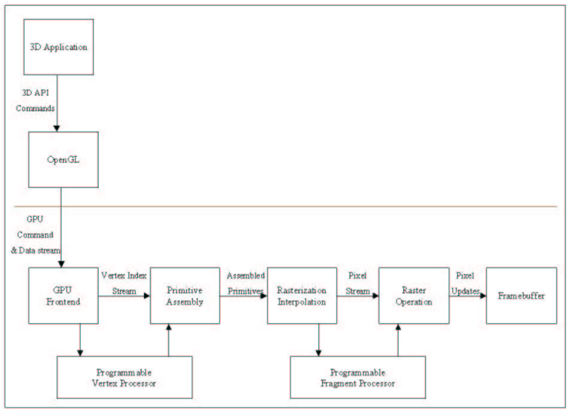

Figure 1: The programmable graphics pipeline.

2.1.1 Vertex and Fragment Processors

The vertex and the fragment processors execute a user-specified vertex and fragment program respectively. All the vertex attributes, e.g. position, color, normal, can be altered by the vertex program. Fragment programs define how fragments are to be shaded. At this stage, textures are mapped to each fragment for computing the final color of each pixel. Vertex programs get executed for each vertex of the input geometric primitives and fragment programs are executed for each fragment generated by the rasterizer.

In this context, there is a fundamental difference in the GPU programming model which closely resembles SIMD from that of the MIMD model for the CPUs.

There is both an assembly-level as well as a high-level (ex. Cg [Fernando and Kilgard 2003; NVIDIA 2002]) programming interface for implementing these programs.

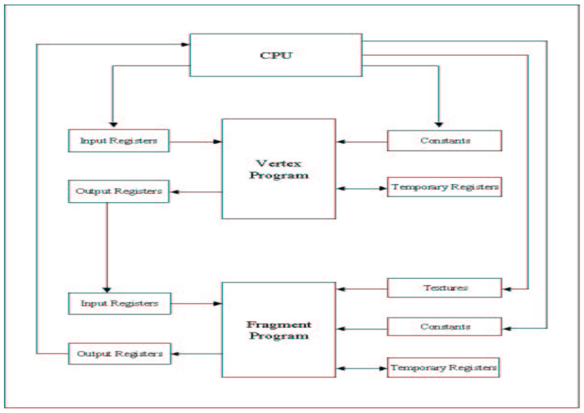

Figure 2: The execution model.

At the beginning of execution stage (Figure 2), the graphics hardware places the element's data fields in the read-only input registers and executes the user program. During the execution stage, the program has access to a set of temporary registers as well as constants defined by the user application. At the end of execution, the results are placed into write-only output registers.

An important point to note here is that there are several key differences in the functionality supported by the vertex and the fragment processors. Vertex programs can branch and use powerful instruction sets but do not have access to the texture memory. On the other hand, fragment programs can't do branching, logical operations (AND, XOR) and have limited instruction set but are able to do texture look-ups. Large datasets can be stored and accessed from the texture memory by fragment programs. Consequently, most of the recent efforts on harnessing GPUs for general purpose computations have been focussed on using fragment hardware.

In terms of execution flow (Figure 2), the data values are passed from the CPU to the GPU pipeline, go through the vertex and fragment processing stages which write the end result into the frame buffer. Thus, CPU can read output values only from the frame buffer even though results maybe available much earlier, for instance at the VP stage. This restriction of no direct data path from CPU to FP and VP to CPU proves to be a critical factor in defining the mapping of numerical computations onto the GPU.

2.1.2 Pixel Buffers

Pixel buffers(Pbuffers) are off-screen rendering contexts for an OpenGL renderer. Pbuffers are useful for computing and storing the results of intermediate rendering steps. One use is for implementing dynamic texturing, which we discuss shortly.

Modern graphic controllers contain immense amounts of video memory and extends PBuffers to support high precision. Instead of limiting rendering contexts to use fixed-point 8bit components for their RGBA or Texcoord components, these controllers provide PBuffers for 16 bit or 32 bit floating-point precision for their RGBA or Texcoord components.

Another feature provided by modern graphic controllers is for PBuffers to have dimensions which are not a power of two. Hence a developer has freedom in creating the required PBuffer for his requirements. Lastly, modern day PBuffers can also be tagged as a texture object, thus allowing even greater flexibility.

2.1.3 Dynamic Texturing

Dynamic texturing is a technique frequently used by graphic developers to attain greater realism. In its simplest form, dynamic texturing involves rendering a scene in a graphics context, downloading the rendered scene as an image onto the host computer, and subsequently uploading the image as a texture object for use in rendering a latter scene. The drawbacks for dynamic texturing are largely centered on the inefficiencies on moving data back and forth between the video memory and the host CPU. Modern graphics controllers provide features to enhance the performance for dynamic texturing. These additions largely center on increasing bandwidth and throughput in the communication channels between the GPU and the CPU. More recently, newer controllers provide support for tagging a off-screen rendering space, the PBuffer, as texture objects for use in subsequent rendering. This latter support is also known as render to texture. The use of render to texture eliminates the need to download a rendered image to the CPU and subsequently uploading it as a texture object, thereby eliminating the communication costs required in earlier implementations.

2.2 Related work

Using graphics hardware for general purpose computing has been an active area of research in the last decade [Lengyel et al. 1990; Hoff III et al. 1999]. The early applications used numerical computing in the context of radiosity [Cohen et al. 1988; Keller 1997].

With the introduction of programmability of the vertex processor [Lindholm et al. 2001] and recently, the fragment processor, the end-user has gained enormous power in terms of programming and executing his general-purpose tasks on the GPU. The recent efforts of mapping numerical algorithms on the GPU include non-linear diffusion [Strzodka and Rumpf 2001], fluids and steam [W. Lei and Kaufman 2003] and simulation of boiling phenomena [Harris et al. 2002].

The approach of Thompson *et. al.* [Thompson et al. 2002] focuses on using the vertex processor as a general-purpose vector processor. The authors implemented a software layer on top of graphics assembly routines. They demonstrated their framework by solving matrix multiplication and 3-SAT problems. Larsen *et. al.* [Larsen and McAllister 2001] also presented a technique for doing matrix multiplication on the graphics hardware. Ian *et. al.* [Buck and Hanrahan 2003] proposed a stream programming model on the fragment processor. They also identified that general-purpose programs which exhibit high arithmetic intensity and data parallel nature would show maximum performance gain from their GPU implementation.

The work of Kruger *et. al.* [Kruger and Westermann 2003] is closest in spirit to our own. In their work, the authors focused on solving sets of algebraic equations on the graphics hardware using linear algebra operators. However, their framework is restricted to doing basic matrix-vector operations. In contrast, our approach provides a framework for doing matrix-matrix computations, thereby closely providing the functionality available in the BLAS library. Bolz *et al.* [Bolz et al. 2003] showed the implementation of two basic computational kernels on the GPU, a sparse matrix conjugate gradient solver and a regular-grid multi-grid solver.

In contrast to virtually all of the previous approaches which execute a single execution sequence through the graphics pipeline, our approach introduces a cascading loop-back model of computation where intermediate results are stored in the video memory. We also show that many concerns related to the severe technical limitations identified by previous approaches, such as low precision [Larsen and McAllister 2001; Thompson et al. 2002] are no longer valid. Though numerical precision can be traded with faster execution timings, the modern GPU provides a full 32-bit floating point pipeline for storing each of RGBA or TexCOORD components. Therefore many complex and precision-sensitive numerical calculations can now be implemented on the GPU.

3 BLAS Programming Framework

The BLAS library comprises of a set of high quality "building block" routines for performing basic vector and matrix operations. The library can be separated into three categories of routines, namely the vector-vector, matrix-vector and matrix-matrix operations. Because the BLAS subroutines are well abstracted, efficient, portable, and widely available, they are commonly used in the development of high quality linear algebra software. Notable examples include the LINPACK and the LAPACK.

3.1 Execution Model

To model the different types of BLAS operations, we first describe the mapping for basic vector operations. For each of the following cases, we illustrate the mapping using the the addition operation.

Vector-Scalar operation: In this example, our purpose is to increment a vector by some scalar value. In this operation (Figure 3), a vector of length *n* is segmented into *m* other vectors of length 4 in the CPU function vsAdd. The vertex program vsAdd.cg is loaded onto the vertex processor and the scalar value is passed as a *uniform* parameter. Subsequently, the CPU function vsAdd would stream the set of *m* vectors onto the CPU as OpenGL primitive points. The

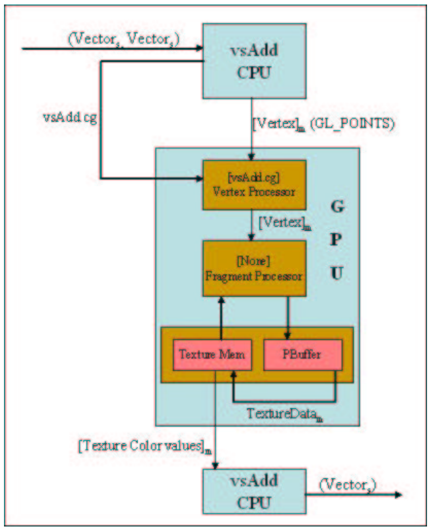

Figure 3: Vector-Scalar operation.

vertex program, vsAdd.cg, would add the scalar value to all the fields in the m vertices. Consequently, the vertices proceed to the fragment processor and are then written into the off-screen PBuffer context. The CPU function vsADD continues to read the color values of each pixel representation of the vertices. These color values contain the result of a vector-scalar operation. The output in the PBuffer is then converted into a texture object. Finally, the CPU function downloads the texture map from the video RAM, concatenates the sequence of color values into a vector of length *n* and return it as the result.

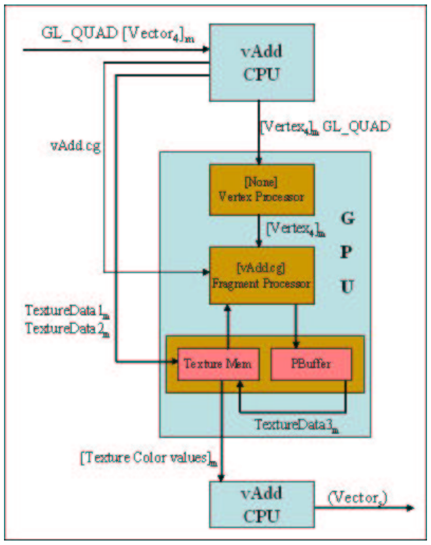

Figure 4: Vector-Vector operation.

Vector-Vector operation: In this operation (Figure 4), two vectors of length *n* are transformed into texture data in the CPU function vAdd. The fragment program vAdd.cg and the texture data are loaded into the fragment processor and the GPU memory respectively. Subsequently, the CPU function vAdd would draw a quadrilateral primitive having *n* pixels. The vertex processor does nothing and passes on the vertices to the rasterizer which generates their pixel representation (fragments). For every pixel, the fragment processor would do two texture look-ups and determine the color value of each pixel based on the operation (addition in this example). These pixels are then written into the off-screen PBuffer context, bound as a texture, and consequently downloaded onto the CPU and returned as the result.

3.2 Folding Technique

The video memory addressing modes are restrictive compared to a CPU. As previously described, the VP is not able to access the video memory and FP can only access it via texture lookups. Hence, it is a challenge to implement *reduce* routines such as summation, dot product, absolute and maximum, which iterate over the contents of some contiguous memory segment (for a vector).

Our implementation for such algorithms uses a technique similar to the one described in [Kruger and Westermann 2003]. It is a recursive algorithm, where for some given input size n, the computation would be performed using a total of $\log(n)$ recursive calls. To illustrate, we describe the execution of the summation algorithm for a matrix containing 128 elements.

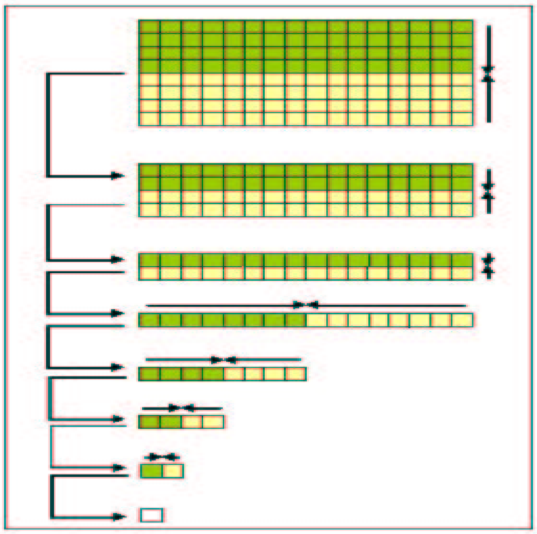

Figure 5: The Folding Technique.

As seen in Figure 5, the matrix is of dimensions, width 16 and height 8, containing 128 elements. At each recursion step, we divide the matrix into 2 equal-sized segments such that each element in the first segment maps to an element in the second segment. The summation process for 128 elements involves $\log(128) = 7$ steps. At each step, the input for our summation routine is a texture of size p pixels, and the output is a texture image of size $p/2$ pixels. Specifically, each step renders an image half the size of the input texture image size. Thus, in our OpenGL rendering commands, each pixel in the quadrilateral is mapped to 2 texture coordinates of the input texture. Consequently, in the last step, the resulting image is 1 pixel in size, which also holds the final result.

From an implementation viewpoint, this technique requires the availability of multi-texturing extensions available in advanced graphics controllers.

3.3 Cascading Loop-Back

For maximum throughput of the graphics pipeline, we propose a cascading loop-back computation model. In this model, we extend the single (binary) operations described in the previous subsection to a sequence of N operations where the intermediate results are stored and accessed as texture objects in the video memory. The cascading flow of the intermediate results from the i^{th} to the $i+1^{th}$ stage ($i = 1,\ldots,N$) alleviates the need of passing data values to and fro between the video and main memory. This novel concept of performing loop-back computations also reduces the expensive hardware set-up times required for each of the independent N passes through the pipeline stages. Moreover, this finds a direct mapping in the domain of linear algebra operations, ex. adding N vectors $\sum_{i=1}^{N} V_i$, multiplying N matrices $\prod_{i=1}^{N} M_i$ etc. We illustrate the model of N Vector-Vector operations using an example of addition of three vectors each of length n, performed in two stages.

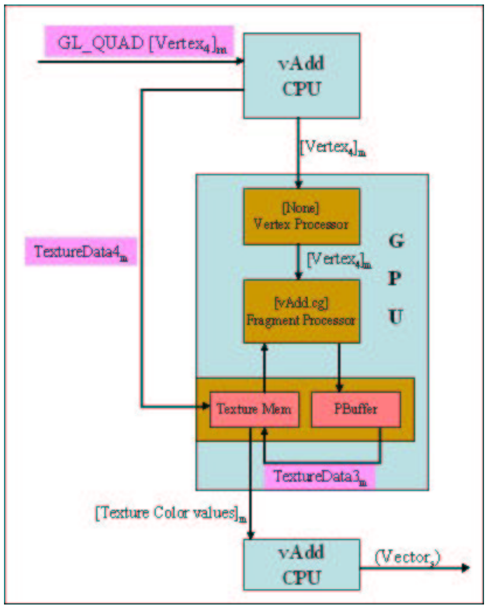

Figure 6: Cascading Loop-back Model: 2 Vector-Vector operations.

N Vector-Vector operations: In this example (Figure 6), we perform two separate vector vector add operations. The first operation proceeds as described earlier for the vector-vector add operation. The output of this operation is used as an input for the second operation. Since the same operation needs to be performed (with different inputs), we do not load a new vertex or fragment program. However, a new texture map needs to be loaded corresponding to the third vector. After that, the second operation proceeds as normal. Lastly, the CPU function concatenates the sequence of color values into a vector of length of length n as the final result.

Matrix operations: In our framework, the implementation of matrix-vector and matrix-matrix operations builds in a bottom-up fashion on top of vector-vector operation modules. The input matrices are stored as texture maps in the video memory. Given an operation, the rows (and columns) of a matrix are treated as vectors and an execution call is made to a vector-vector subroutine corresponding to the desired operation. These calls are made in conjunction with the *reduce* operations to process (e.g. accumulate dot-products in matrix multiplication) the output of vector-operation sub-routines.

3.4 Framework Implementation

3.4.1 BLAS Type handling (blas_data)

BLAS input types have to be represented as OpenGL elements for processing by the GPU. We describe the interface for the C++ class blas_data which acts as a wrapper for BLAS subroutine inputs.

```
class blas_data
{
  public:
     blas_data(float* data,int w,int h,int datatype);
     blas_data(PBuffer* pad, GLuint id,
               int w, int h, int datatype);
     ~blas_data();
     PBuffer* getPBuffer();
     float* readBack();
     bool isPBufferValid();
     int getWidth();
     int getHeight();
     int getPixelCount();
     GLuint getTextureID();
};
```

The constructors create a blas_data instance containing a *GL_TEXTURE_RECTANGLE_NV* texture object representing the *data* object (or referencing a PBuffer). This constructor is used to represent data corresponding to the *datatype* for latter computation by a GPUOp object.

The read-back function downloads the contents of the *GL_TEXTURE_RECTANGLE_NV* referenced by the blas_data instance and separates the RGBA components into individual floats. Consequently the values are returned in a float array. The remaining functions return the dimensions, pixel count and texture reference of the blas object respectively.

3.4.2 GPU Operation (GPUOp)

```
class GPUOp
{
  public:
    GPUOp(CGprofile VP_program,
          CGprofile FP_program,
          int datatype);
    ~GPUOp();
    blas_data* Compute();
    void setArg(blas_data* in, int arg_idx);
    int getArgumentCount();
};
```

The GPUOp constructor sets up the proper contexts which are used subsequently for setting up the vertex and fragment programs and the PBuffer. The destructor de-allocates the PBuffer memory. The compute function loads the appropriate vertex and fragment programs. It also instantiates the necessary PBuffer space and performs the desired computation by rendering a geometric primitive.

The *setArg* function pushes a blas_data instance onto a stack for latter use as an input for GPU computations. Figure 7 illustrates this framework for the Vector-Vector operation of adding 2 vectors, vAdd.

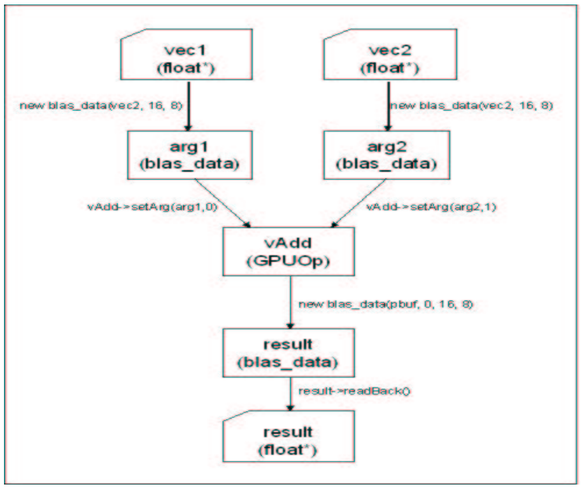

Figure 7: Implementation of BLAS vector-vector operation.

4 Experiments and Results

In this section, we describe the performance benchmarks and analyze the experimental results from their execution on the testbed machine.

Our BLAS library prototype has been developed on a representative hardware system. The experimental testbed was a 1.8 GHz Intel Pentium IV workstation with 512 MB of RAM running Windows XP. It's graphics system was a NVIDIA Quadro TI 5800 Ultra with 128 MB DDR SDRAM on an AGP4X bus. The card device driver was NVIDIA 6.14.01.4345. The timing measurements were taken using the reference hardware's high-resolution performance counter. The frequency of the counter is 3579545 ticks per second. As such our timings are of high precision.

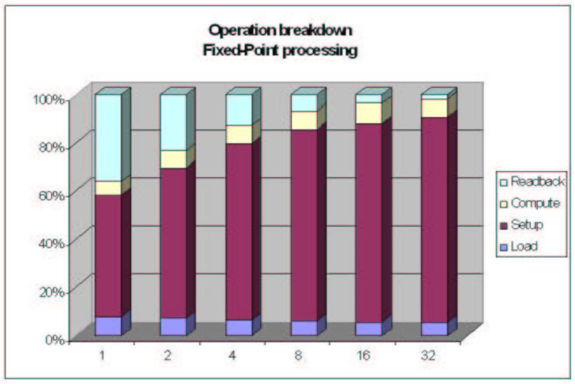

Figure 8: Execution time for matrix multiplication (8 bit Fixed Point).

4.1 Computation Costs

Offloading a computation onto the GPU is associated with certain costs. These costs can be broadly classified as *upload time*, *computation setup time*, *computation time*, and *download time*. For many computations, the input has to be morphed and uploaded onto the video memory as texture objects. We define the cost associated with converting matrices and vectors into textures and uploading them onto the GPU as the upload time. For any computation, the software also has to instantiate and initialize an off-screen rendering context (a PBuffer) for computation. This is refered as the computation setup time. Consequently, the actual time taken to render a scene for a BLAS subroutine computation is known as the computation time. Lastly, the computation results are bound as a texture object. The contents have to be downloaded, and represented in some form for further processing. The associated cost to this is refered as the download time. Hence the total computation time in this step is the sum of all these four components.

To determine the individual contribution of each cost to the total execution time, we performed a matrix multiplication benchmark. The size of each matrix is four million elements. The multiplication program $\prod_{i=1}^{N} M_i$ runs for an entire sequence of 32 cascading loop-back operations ($N = 32$). The break-down percentages of the execution timings for the fixed point and floating point are shown in Figures 8 and 9 respectively.

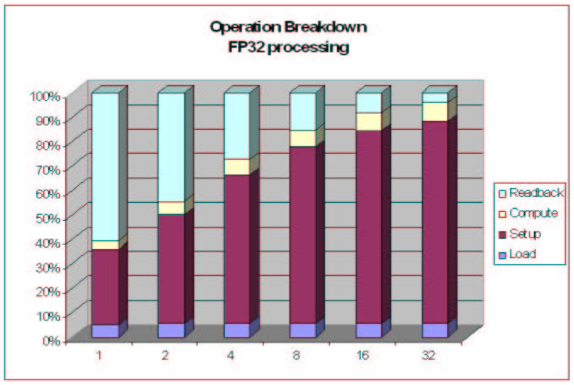

Figure 9: Execution time for matrix multiplication (32 bit Fixed Point).

Compared to the fixed point implementation, the read-back timings from floating point PBuffers are significantly expensive for a single operation. However, amortizing the timings for the entire sequence of operations, the PBuffer set-up times play the dominant role in both implementations. Since only a single read-back of the results is required at the end of execution sequence, the ratio of read-back to the total time for floating-point computation is only marginally higher than that of fixed-point. Consistent with this observation, [Bolz et al. 2003] also identified a high performance penalty associated with PBuffer switches.

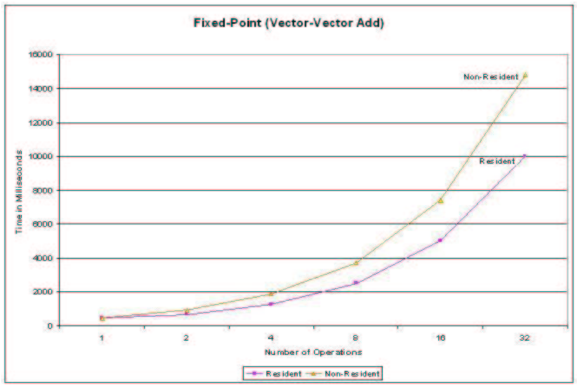

Figure 10: Execution time for matrix multiplication (8 bit Floating Point).

4.2 Fixed point v/s Floating point

This benchmark demonstrates the effectiveness and performance gain of cascaded loop-back computation model for both the fixed point and the floating point pipelines. Figures 10 and 11 show the total execution time versus number of operations for fixed-point and floating-point computations respectively. The plots in each figure correspond to two scenarios, the cascade loop-back model where intermediate results are stored on GPU's video memory and the other depicting single independent executions where intermediate results are downloaded from video to main memory and subsequently, uploaded again for latter computations. From our bench-

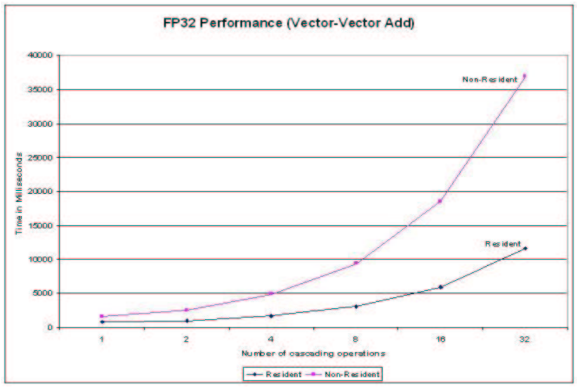

Figure 11: Execution time for matrix multiplication (32 bit Floating Point).

mark timings, we observe that the execution time decreases by a factor of 3 for floating point computations. The corresponding factor for the fixed point case is about 1.5. This can be largely attributed to the significant difference in the download timings between fixed-point and floating-point data from the GPU. This result strongly advocates our claim that for achieving high performance, high-precision numerical computations should be mapped onto the GPU in a cascading loop-back model. Another important conclusion is that the performance is limited by bandwidth in the current GPUs.

4.3 Computation Efficiency

The computation efficiency is defined as the ratio of the actual GPU computation time to the total execution time comprising of load, setup, compute and read-back. As observed in Figure 12, the efficiency of fixed point calculations is about 60 percent compared to about 33 percent for the floating point. This can be explained on the basis of dominating set-up and read-back timings for the floating point case compared to fixed point.

An interesting observation here is that there is a peak in efficiency for a vector of size approximately 16 million ($8*2$ million). We explain this on the basis of size of the video memory (128MB in our testbed). Since we are storing each 32 bit (4 byte) vector as textures in the video memory, the space occupied would be $8*2*1e6*4$ (nearly 64MB). Therefore, the total space used for two vectors would be about $2*64$MB which is nearly equal to the video memory. After the cross-over point, AGP memory would also be used for storing the data elements. Since the transfer speeds of AGP4x are slower compared to the video RAM, there is a decrease in efficiency as the total time increases. The efficiency for the fixed-point computations doesn't exhibit the same behaviour since only the video memory is used for storing the texture maps of the two vectors.

4.4 Implementation

We mention here some observations we made during our implementation that may be of interest to the developer community porting scientific applications onto the graphics hardware.

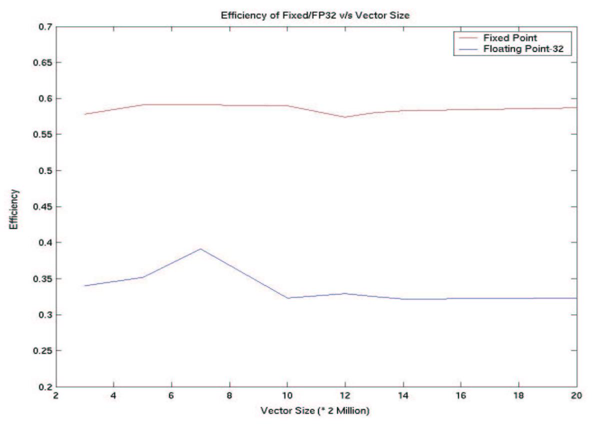

Figure 12: Efficiency of 8-bit fixed-point v/s 32-bit floating point (matrix multiplication - each 2 million elements).

- Numerical Accuracy
  We use 32 bit floating point pixel buffers. In many cases, numerical methods have to be performed in double precision to allow for accurate and stable computations. So, it becomes important for the software library as well as the target architecture to support sufficient accuracy.

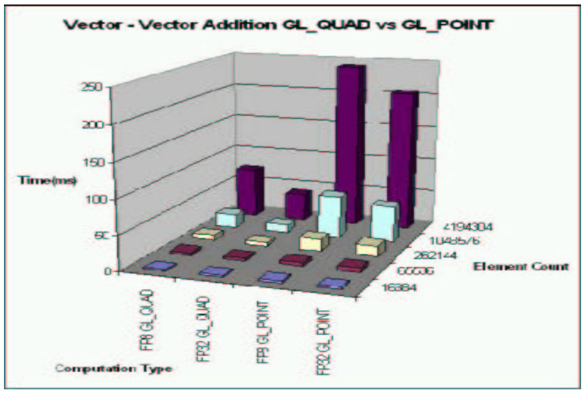

Figure 13: Comparison of GL_QUAD v/s GL_POINTS.

- GL_QUAD v/s GL_POINT
  To map numerical calculations to graphics, a geometric primitive is used for on-screen as well as off-screen rendering. In this experiment, we did a performance comparison between GL_POINTS and GL_QUAD. For two input vectors of size four million each, the timings in Figure 13 show a *five* times speed-up using GL_QUAD for floating point and a *three* times speed-up for fixed point over rendering GL_POINTS. Though this observation might appear intuitive, it has strong implications for achieving high performance in scientific applications.

- PBuffer Read-back Timings Degradation
  In the matrix multiplication operation, say $M_1 * M_2$, a PBuffer is created for each column (vector) of M_2 to be multiplied with a row (vector) of M_1. For such successive PBuffer invocations, we observed (Figure 14) a significant degradation in read-back timings for floating point PBuffers as the number of invocations increase. This degradation is also observed (Figure 15) for the fixed point case. A possible explanation could be the overheads in transferring the data to and fro between the AGP and the video memory. The result also shows that the existing GPUs are bandwidth limited and not computation limited.

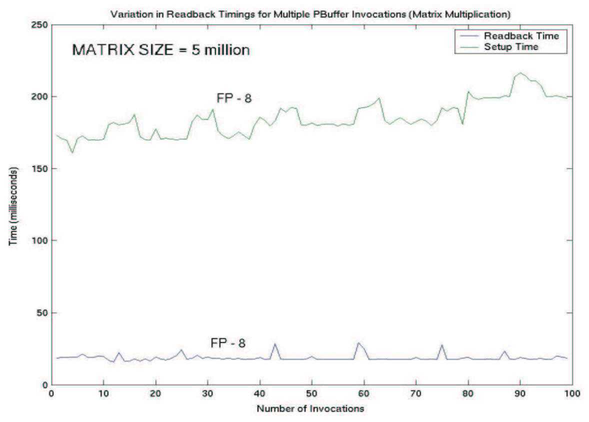

Figure 14: Degradation in read-back timings (fixed-point PBuffer).

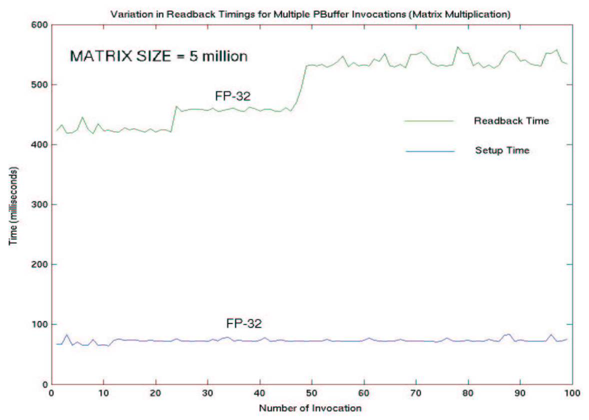

Figure 15: Degradation in read-back timings (floating-point PBuffer).

- RGBA pixel format
  We found that 4 numbers can be packed into a single pixel by setting R, G, B and A values to the corresponding vector components. This also results in a improved performance compared to using RGB format.

5 Discussion

In this section, we analyze the important implications for the graphics architectures arising from the experimental results in the previous section. We propose a set of changes to the hardware and programming interfaces which has potential for enhancing the efficiency of general-purpose computing on GPU.

5.1 Hardware

These recommendations are important from the perspective of GPU architecture designers. Furthermore, we feel that incorporating the changes into the target architecture itself would be a vital factor in boosting performance.

VP, FP as 1^{st} class processors: For computer graphics applications, the vertex and the fragment processors perform the specialized task of geometry and fragment level processing respectively. In contrast, for general purpose computing, each processing unit should provide an access mechanism to all the hardware functionalities for maximum throughput and easy portability. The immediate implications for GPUs are that both the VP and FP should be first class processors having similar cores and instruction sets. Specifically, FP should have access to the powerful instruction set of VP and have branching instructions. Likewise, VP should have memory access operations to fetch the operands from the texture memory. To fully exploit the parallelism, the execution speeds should be similar so that the output from one could be consumed by the other at the same rate without any need for expensive intermediate register storage.

Increase GPU-CPU Memory Bandwidth: As observed from the dominant set-up and read back timings, both transferring and reading back the results from the GPU is a bottleneck in existing architectures. This overhead is considerably large compared to the actual execution time. Our proposed cascading loop-back model attempts to alleviate this but a stronger impact can be achieved by providing a high bandwidth memory interface between the two processing units.

For some architectures in particular, the read back from the GPU is implemented in software by passing the values through the device driver to the CPU. This might not be an issue for graphics applications but for scientific computing, a large volume of numerical results need to be read from the video memory. Hence, providing a fast hardware read back mechanism becomes all the more important to contend as an efficient platform for these applications.

Reuse of Pixel Buffers: In our implementation framework, each subroutine is computed on the GPU in a separate PBuffer and it's results, which is the rendered scene are bound as texture objects in the video memory. Further invocations of other subroutines would require creating new instantiations of PBuffers. As evinced by the performance benchmarks, the PBuffer instantiation costs make up a significant proportion of the total subroutine run-time. According to NVIDIA's white-papers, most developers seldom require more than one PBuffer invocation. Hence, for graphics applications, reducing the PBuffer set-up overheads might not be a priority but it is critical for general-purpose applications which require multiple phases. Thus, rather than creating a new PBuffer each time, an interface to re-use an existing PBuffer would reduce execution times considerably.

Multiple Texture Fetches: Currently, a texture fetch instruction accesses the RGBA vales of a single texel in the video memory. Enhancing the instruction set and the memory interface for reading multiple texel values in one instruction (e.g. a constant, say c, number of consecutive locations) could potentially increase the efficiency of vector computations by a factor of c.

Furthermore, instructions having additional offsets to the texture coordinates can be used for doing data-dependent computations.

Global Registers: Providing a set of globally read-write registers would help passing important values between subsequent execution calls. For example, an accumulation register would be useful for doing sum-reduction operations.

Preserving state across multiple calls: Preserving state information can lead to an effective information flow across execution calls. Globally accessible locations is one mechanism to achieve this - other features which can be possibly included are a stack and a register set for passing intermediate values and state parameters.

Allow CPU to read-write GPU registers: Both the VP and FP contain a set of input, temporary and output registers for program execution. Currently, access to these registers is limited and there's a restriction on data placement ex. the developer can't directly read or write the individual registers from the host application. The values in the VP registers can be set using OpenGL calls such as *glTexCoord**, *glSetColor** but the host application is not allowed to access registers in the FP. The restriction on precision is also observed when accessing VP registers. For instance, in the VP, the values for COLOR0 register can only be fixed-point whereas the values for *Texcoord** registers can be 32 bit floating point.

Introduce a Stack: Numerical calculations (e.g. time-dependent differential equations) often compute the results in a recursive fashion. Allowing a mechanism to support stack operations, these applications can be easily mapped onto the GPU.

Pointers: The concept of a memory pointer is meaningful for scientific applications, but not necessarily for graphics. Again, this has important implications from the perspective of both the portability and efficiency of numerical programs.

Introduce bit-wise operations: The logical operations (ex. AND, XOR) are important for many image-processing applications and in our opinion, the existing architecture can include hardware features to support them with minimal extensions.

5.2 Graphics Developers

These recommendations are based on the insights we gained during the implementation of our framework. From the viewpoint of graphics hackers, these critical points must be kept in mind while developing high performance applications for the graphics hardware.

Load important data into video memory: Since the difference in memory latency between video and system memory is significant, the video RAM should be utilized for storing textures and frequently accessed parameters.

Texture Interpolation: A primary requirement for supporting scientific applications is to provide floating point arithmetic. Modern day GPUs support 8, 16, 24 and 32 bit floating point PBuffers and texture objects.

It is noteworthy that fixed-point PBuffers and texture objects need to be dimensions of size, power of 2. In addition, fixed-point PBuffers do not inherently provide support for negative values. A potential workaround is to introduce a bias which involves additional computation after downloading the GPU computation results. However, these restrictions are not present in floating-point PBuffers or texture objects.

During the course of our implementation, we also found floating point PBuffers and texture objects easier to work with. Using fixed-point PBuffers, the texture coordinates are clamped to [0, 1] and often during computation, we realized that our texture coordinates are off by some precision. However, when using floating-point PBuffers, the texture coordinates are not clamped and are indexed relative to the width and height dimensions. This has contributed to easier programming and removed the computation costs required to clamp some texture coordinate.

From our survey, we found that current hardware implementing the CineFX architecture just has hardware support for the *GL_FLOAT_R32_NV*, *GL_FLOAT_RGBA16_NV* and *GL_FLOAT_RGBA32_NV* internal formats. The other formats are supported but currently expanded into the driver using one of the three formats in the list. In practical terms, that means that a 2 component texture takes up as much space as a 4 component texture. As such it is prudent to use 4 components even if we are going to use 2 or 3 components.

Maximum use of Fixed-Point Pipeline: As evinced by the experiments, there is a big performance gap between the fixed point 8-bit pipeline vis-a-vis it's 16-bit and 32-bit counterparts. This can be used effectively for computing applications which can trade-off numerical accuracy for higher speed. Another important point is that the fixed point pipeline in graphics architectures has been well-tested and comparatively more stable than the floating point ones.

Geometric Primitives: As discussed in the previous section, the performance gain using GL_QUAD is significant compared to using GL_POINTS as the geometric primitive.

Code Optimization (Instruction, Memory): In conjunction with the compiler optimizations, we feel that the developer also needs to optimize the vertex and the shader program in terms of instructions and memory accesses. This is based on the following observations in the Cg language:

- Using In-built functions
  We observed that using the library functions from the language was more efficient than doing the same computation from the basic set. For example,

  ```
  float vec = dot(point, float3(0.1, 0.2, 0.3));
  ```

  is faster compared to :

  ```
  float vec = p1.x * 0.1 + p1.y * 0.2 + p1.z * 0.3;
  ```

- Register accesses vis-a-vis number of computations
  We observed that for complex calculations, it is often faster and easier to break into small sub-expressions and re-using the common terms by storing in temporary variables (registers). For example, (mi are matrices for $i = 1, \ldots, 4$)

  ```
  result = m1*m2*m3*m4 + m1*0.5*m2 + 0.8*m2*m3*m4;
  ```

  is slower than executing the following calls :

  ```
  mat12  = m1 * m2;
  mat34  = m3 * m4;
  result = m12 * m34 + 0.5 * m12 + 0.8 * m2 * m34;
  ```

Video Card Drivers: Debugging graphics hardware is a difficult task considering the non-stability of the video device drivers. According to our literature survey, ATI's drivers are more stable but do not achieve optimum performance. On the other hand, NVIDIA's drivers are smarter but generate more errors. Thus to harness the capabilities of the hardware, it is pertinent to frequently update the drivers.

In our experience, the techniques to extract the maximum performance from the graphics hardware are certainly non-trivial and moreover, not well documented. A fairly detailed set of experiments and many trials are required to figure out how to make things run fast and sometimes, even in a correct or expected manner.

5.3 Bug Reports

During the course of our implementation, we found the following bugs in NV30 device driver and Cg compiler respectively. These bugs are reproducible and have been confirmed by NVIDIA.

- After rendering to a PBuffer and making a PBuffer a texture, it is not possible to do a read-back from the texture object. This bug exists on the actual NV30 machine but works fine on the NV30 emulator. One workaround is to do a read-back on the PBuffer instead of the texture object.

- *cgSetColor* function call does not update the alpha component of the color register. One workaround is to use the OpenGL *glSetColor4f* function instead.

6 Conclusion

In this research, we have investigated the GPU's potential as an efficient computing engine for scientific, specifically linear algebra computations. We describe a framework for the implementation of linear algebra operators on GPUs, providing the building blocks for the design of more complex numerical algorithms. We identify the potential architectural bottlenecks and analyze their pros and cons based on our experiments. Furthermore, in order to achieve maximum performance, we propose important enhancements for future graphics architectures.

References

BOLZ, J., FARMER, I., GRINSPUN, E., AND SCHRODER, P. 2003. Sparse matrix solvers on the gpu: Conjugate gradients and multigrid. In *Proceedings of the SIGGRAPH 2003 Conference*.

BUCK, I., AND HANRAHAN, P. 2003. Data parallel computation on graphics hardware.

CGSHADERS. 2003. *Cg Discussion Forum, http://cgshaders.org*.

COHEN, M. F., CHEN, S. E., WALLACE, J. R., AND GREENBERG, D. P. 1988. A progressive refinement approach to fast radiosity image generation. In *Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, ACM Press, 75–84.

FERNANDO, R., AND KILGARD, M. J. 2003. *The Cg Tutorial*.

HARRIS, M. J., COOMBE, G., SCHEUERMANN, T., AND LASTRA, A. 2002. Physically-based visual simulation on graphics hardware. In *Proceedings of the conference on Graphics hardware 2002*, Eurographics Association, 109–118.

HARRIS, M. 2003. General purpose computation using graphics hardware.

HOFF III, K. E., KEYSER, J., LIN, M., MANOCHA, D., AND CULVER, T. 1999. Fast computation of generalized Voronoi diagrams using graphics hardware. *Computer Graphics 33*, Annual Conference Series, 277–286.

KELLER, A. 1997. Instant radiosity. *Computer Graphics 31*, Annual Conference Series, 49–56.

KRUGER, J., AND WESTERMANN, R. 2003. Linear algebra operators for gpu implementation of numerical algorithms. In *Proceedings of the SIGGRAPH 2003 Conference*.

LARSEN, E. S., AND MCALLISTER, D. 2001. Fast matrix multiplies using graphics hardware. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, ACM Press, 55–55.

LENGYEL, J., REICHERT, M., DONALD, B. R., AND GREENBERG, D. P. 1990. Real-time robot motion planning using rasterizing computer graphics hardware. *Computer Graphics 24*, 4, 327–335.

LINDHOLM, E., KILGARD, M. J., AND MORETON, H. 2001. A user-programmable vertex engine. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, ACM Press, 149–158.

NVIDIA-OPENGL. 2003. *NVIDIA OpenGL Extension Specifications, http://developer.nvidia.com/docs/IO/1174/ATT/nvOpenGLspecs.pdf*.

NVIDIA-OPENGL. 2003. *NVIDIA OpenGL Extension Specifications, http://developer.nvidia.com/view.asp?IO=nvidia_opengl_specs*.

NVIDIA, I. 2002. *Cg Toolkit user's manual*.

NVSHADER. 2003. *NV_texture_shader extension, http://oss.sgi.com/projects/ogl-sample/registry/NV/texture_rectangle.txt*.

OWENS, J. D., DALLY, W. J., KAPASI, U. J., RIXNER, S., MATTSON, P., AND MOWERY, B. 2000. Polygon rendering on a stream architecture. In *2000 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, 23–32.

STRZODKA, R., AND RUMPF, M. 2001. Nonlinear diffusion in graphics hardware. In *Proceedings of the Joint Eurographics-IEEE TCVG Symposium on Visualization (VisSym01)*.

THOMPSON, C. J., HAHN, S., AND OSKIN, M. 2002. Using modern graphics architectures for general-purpose computing: A framework and analysis. In *Proceedings of the 35th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35)*.

W. LEI, X. W., AND KAUFMAN, A. 2003. Implementing lattice boltzmann computation on graphics hardware.