Using Modern Graphics Architectures for General-Purpose Computing: A Framework and Analysis

Chris J. Thompson Sahngyun Hahn Mark Oskin
Department of Computer Science and Engineering
University of Washington
{*cthomp, syhahn, oskin*}*@cs.washington.edu*

Abstract

Recently, graphics hardware architectures have begun to emphasize versatility, offering rich new ways to programmatically reconfigure the graphics pipeline. In this paper, we explore whether current graphics architectures can be applied to problems where general-purpose vector processors might traditionally be used. We develop a programming framework and apply it to a variety of problems, including matrix multiplication and 3-SAT. Comparing the speed of our graphics card implementations to standard CPU implementations, we demonstrate startling performance improvements in many cases, as well as room for improvement in others. We analyze the bottlenecks and propose minor extensions to current graphics architectures which would improve their effectiveness for solving general-purpose problems. Based on our results and current trends in microarchitecture, we believe that efficient use of graphics hardware will become increasingly important to high-performance computing on commodity hardware.

1. Introduction

Modern graphics hardware has reached a turning point where raw performance enhancements—as measured by traditional metrics such as polygon fill rates—are becoming less and less essential. As McCool observes [10], performance levels of "cheap" video game hardware are sufficient to overwrite every single pixel of a 640x480 display with its own transformed, lit, and textured polygon more than 50 times every 30th of a second. As such, recent graphics hardware architectures, including both research and commercial designs, have begun to move away from traditional fixed-function pipelines and emphasize versatility, providing rich new ways of programmatically reconfiguring the graphics pipeline [8, 10, 16]. As a result, powerful and potentially general-purpose constructs not unlike vector and stream processors are appearing in commodity PC machines, thanks to their graphics chips. The power of these "graphics" processors should not be underestimated; for example, the NVIDIA GeForce3 chip contains more transistors than the Intel Pentium IV, and its successor the GeForce4 is advertised as being able to perform more than 1.2 *trillion* internal operations per second. Most of this time, however, this power is going unused because it is only being exploited by graphics applications.

We believe that current graphics architectures, with minor evolutionary changes, could be used to accelerate other domains where vector processors might traditionally be used. This approach is important because, due to economic and other factors, it is unlikely that dedicated vector processors will ever become commonplace on the desktop. In contrast, powerful graphics chips are already widely available.

In this research, we investigate the programming, performance, and limitations of a recent graphics architecture on non-graphics problems. We begin by describing prior work in this area. In Section 3, we describe modern programmable graphics hardware architectures. The next section presents a programming framework we have devised that allows us to conveniently solve computational problems using graphics hardware. In Section 5, we implement a number of toy algorithms with our framework, and we also apply the framework to two real problems: matrix multiplication and 3-satisfiability. We then compare the speed of our graphics card implementations to CPU implementations. In various cases, we demonstrate significant performance gains. At the same time, there is potential for considerable improvement. We analyze the bottlenecks and in Section 6 we propose minor architectural extensions to current graphics architectures that would improve their effectiveness and efficiency for solving general-purpose problems. We conclude with a discussion of avenues for future work, including some which draw inspiration from earlier research in VLIW and vector processors.

2. Prior work

While the last decade was dominated by fixed function non-programmable graphics architectures, some programmable graphics architectures were also explored. Early systems, such as Pixar's CHAP [2, 7] and the commercially available Ikonas platform [3], had user microcodable SIMD processors that could process vertex and pixel data in parallel. Programmable MIMD machines that could process triangles in parallel, such as the Pixel Planes [4] and the SGI InfiniteReality, became popular for a short time, but their low-level custom microcodes were complex and rarely used by commercial developers.

As transistor costs decreased, CPU vendors began to introduce graphics-oriented SIMD processor extensions into general purpose CPU designs. Examples of these extensions include Intel's MMX/SSE instructions, AMD's 3DNow architecture, and Motorola's AltiVec technology. While such extensions can accelerate a variety of graphics operations, they fall far short of the functionality of even a basic graphics chipset; for instance, none offer high level support for a rendering pipeline. For this reason, it is likely that all modern computer architectures for the foreseeable future will include sophisticated graphics coprocessors, motivating the work in this paper. Following industry conventions, we refer to graphics coprocessors as GPUs [Graphics Processing Units].

More recently, Sony developed a custom dual-processor SIMD architecture for graphics called the Emotion Engine [6]. This design is fully programmable. The first of the two processors is interfaced to the main CPU as a coprocessor, running instructions from the application's instruction stream, much like MMX or AltiVec. The second processor executes custom assembly subroutines for graphics or sound [8]. While the Emotion Engine is powerful, its extremely high level of programmability has also earned it a reputation for being difficult to program, since application writers must pay careful attention to very low-level details such as pipeline latency, hazards, and stalls throughout the rendering process.

Most new graphics architectures strike a balance between programmability and manageability by exposing only part of the rendering process to programmers. NVIDIA's GeForce3/4 and ATI's Radeon grant full programmatic control over the process through which individual vertices are transformed from modeling space to world space. All attributes (position, color, normal vector, *etc.*) of each vertex may be programmatically altered via *vertex programs* written in a custom assembly language. Later in the rendering process, *shader programs* let the programmer control how textures are mapped as well as how the final colors of geometric fragments are computed. Each programming model is designed to limit implementation complexity. For instance, in vertex programs every assembly instruction has the same latency, memory accesses are only allowed to registers and the maximum number of different registers accessed by each instruction is capped so that the register files only need a small number of ports, and precisely one instruction is issued per clock. None of these programming models was designed with general purpose non-graphics programming in mind, but as we demonstrate in this paper, they have the potential to evolve in this direction.

The work of Proudfoot *et. al.* [18] is closest in spirit to our own. In that work, the authors describe a language for developing arbitrarily complicated graphical shaders and a compiler for that language that generates code targeted to modern graphics architectures. The authors propose an innovative abstraction called *computation frequencies* that allows them to combine vertex programs and pixel programs under one high-level umbrella, and their compiler is intelligent enough to virtualize hardware resources that may not exist on a given hardware target. However, because their work is oriented towards graphical shaders, it does not offer any support for branches, labels, or main memory access, making it unsuitable for the kind of general purpose programming explored in this paper. Nevertheless, Proudfoot's work was influential; the *Cg* compiler [14] released by Microsoft and NVIDIA very recently is essentially a commercial version of that work with some practical extensions such as labels. *Cg* and other similar emerging languages [1] remain too focused on graphical shader development to be used for the kinds of general-purpose programming explored in this paper, but their evolution towards generality remains an exciting area for future work.

3. Modern graphics hardware

3.1. The graphics pipeline

Traditionally, graphics hardware follows a fixed series of steps called the *graphics pipeline* to render an image. These steps are illustrated below:

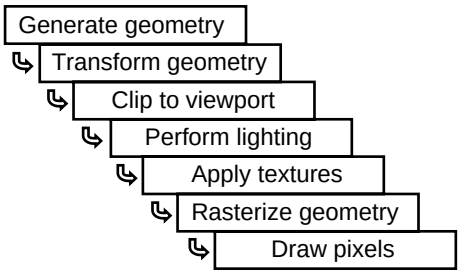

Figure 1. The traditional graphics pipeline.

Initially, a user application supplies the graphics hardware with raw geometry data (typically in the form of four-component homogenous vectors[1]) specified in some local coordinate system. The hardware transforms this geometry into world space, then clips away parts of the transformed geometry not contained within the user's viewport. Next, the hardware performs lighting and color calculations. Textures are applied, then the hardware converts the vector-based geometry to a pixel-based raster representation. Finally, the resultant pixels are composited into the screen buffer.

Most contemporary programmable architectures revise the standard pipeline as shown in Figure 2:

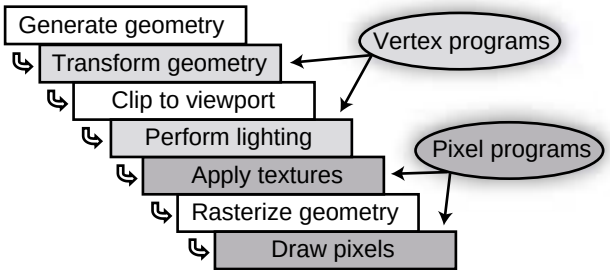

Figure 2. A programmable graphics pipeline.

The transform and lighting steps collapse together into one step in which the position, color, and lighting of geometry are determined by vertex programs written in a custom assembly language. Similarly, the texturing and color compositing stages are collapsed into a single abstract stage where the outputs are determined by shader programs, written in either an assembly language or configured through a simpler series of functional statements (depending on the particular programming interface).

For the remainder of this paper, we concentrate on a specific target, the NVIDIA GeForce4 chipset [8]. This chipset is representative of recent graphics architectures; it is functionally similar to ATI's Radeon chipset as well as the hardware in Microsoft's Xbox game console. It conforms fully to the abstract DirectX 8 interface [11] and can also be accessed using a more basic application programming interface called OpenGL [13].

3.2. Vertex programs

In this section, we describe the GeForce4's instruction set architecture for vertex programming. All registers in this architecture hold quad-valued floating point values. If the user tries to load a register with a scalar or integer value, the value is automatically converted to floating point and mirrored into each of the four vector slots.

Essentially, vertex programs compute functions. Each program takes its input from a series of read-only source attribute registers, and places its results into a series of write-only output attribute registers. Temporary values may be stored in temporary registers which are both readable and writeable. Vertex programs cannot access main memory directly, but they may read values from a block of 96 constant registers which can be used to pass information into vertex programs. The constant register file may also be accessed indirectly, through a special register called the address register (used as an index into the register file).

Each time the hardware receives a new geometry vertex, the hardware loads the 16 quad-float source attribute registers with attributes, such as position, normal vector, and color describing that vertex. The hardware also initializes the 16 temporary registers to $(0,0,0,0)$. It then invokes whatever vertex program is currently active.

Table 1. The source attribute registers.

Register name	Meaning
v[OPOS]	Object position
v[WGHT]	Vertex weight
v[NRML]	Normal vector
v[COL0]	Primary color
v[COL1]	Secondary color
v[FOGC]	Fog coordinate
v[6]–v[7]	Unused
v[TEX0]–v[TEX7]	Texture coordinates

To limit the number of ports required of the register file and ensure that one instruction can complete per clock, each individual assembly instruction may only read from a single source attribute register (*e.g.* ADD R0, v[OPOS], v[WGHT] would be invalid). Similarly, each instruction can only access a single constant register.

After computing its results, the program updates the relevant 15 output attribute registers. This information is tagged

Table 2. The output attribute registers.

Register name	Meaning
o[HPOS]	Clip space object position
o[COL0]–o[COL1]	Front colors
o[BFC0]–o[BFC1]	Back colors
o[FOGC]	Fog coordinate
o[PSIZ]	Point size
o[TEX0]–o[TEX7]	Texture coordinates

[1] Homogenous quad-vectors are popular in graphics because they allow perspective transformations to be represented as matrix multiplication. In this notation, (x,y,z,w) corresponds to the location $(x/w,\ y/w,\ z/w)$ in Cartesian space.

to the current vertex, and then the vertex is passed on to the next stage of the graphics pipeline.

There are 21 instructions that can be used in vertex programs. Each generates a single result and places it in a destination register. The instructions are summarized in Table 3. Most instructions operate on all four components of the input register, but a few instructions (*e.g.* EXP, LOG) operate only on a single scalar component. The hardware limits every vertex program to a maximum of 128 instructions.

Table 3. The vertex program instruction set.

Opcode	Description
ARL	Address register load
MOV	Move
MUL	Multiply
ADD	Add
SUB	Subtract
MAD	Multiply and add
ABS	Absolute value
RCP	Reciprocal
RCC	Reciprocal (clamped)
RSQ	Reciprocal square root
DP3	3-component dot product
DP4	4-component dot product
DPH	Homogenous dot product
DST	Cartesian distance
MIN	Minimum
MAX	Maximum
SLT	Set on less than
SGE	Set on greater/equal than
EXP	Exponential base 2
LOG	Logarithm base 2
LIT	Light coefficient formula

3.3. Shader programs

Unlike the vertex programming hardware, the shading hardware is not naturally programmable in a general assembly language. Four texture units capable of performing various kinds of indexed lookups into 1, 2, or 3-dimensional banks of memory may be turned on or off by the programmer and assigned specific mapping operations. Every geometric fragment passes through each of the four texture units in sequence, then finally through a "combiner" stage which generates an output color based on the final texture unit's result and the geometric fragment's untextured color (computed earlier by a vertex program). This final combiner can compute a fixed set of functions.

The DirectX programming interface attempts to abstract the pixel shading hardware behind an assembly language similar to the language used for vertex programs. However, this language is much more restricted, since any shader program must ultimately be no more complex than the underlying hardware. A fixed number of "texture address" opcodes are allowed to control texture lookup and mapping, and similarly a fixed number of "texture register" opcodes control the computation performed by the final combiner stage. The OpenGL interface to the pixel shading hardware does not attempt a similar abstraction, instead exposing the underlying hardware directly through a configurable, but non-programmable interface.

Even though the shading hardware is relatively restricted in a general-purpose sense—always performing four (customizable) texture lookups in serial then a limited function computation—it can nevertheless be used for some nontrivial computation. For instance, clever programmers can run two-dimensional physical simulations in hardware using texture lookups and combiners to perform finite-difference calculations. Similarly, it is possible to arrange the texture lookups to perform Gaussian blur and generalized convolution at interactive rates [15]. As future hardware architectures for shading inevitably evolve towards full programmability, shader programs will be a powerful resource for general-purpose computation.

Somewhat surprisingly for hardware designed to process pixel samples, the shading hardware offers no built-in support for logical operations (AND, XOR, *etc.*). In some cases, however, these operations can be simulated using the available hardware.

In developing our programming framework, we decided not to explore the use of shader programs. Though such an investigation would be interesting, we felt that our current efforts would be more fruitfully applied to other areas, for several reasons. First, the vertex programming architecture is inherently much more general and programmable than the current shading architecture; vertex programs are thus more naturally suited for general-purpose computation. Second, the shading hardware currently uses 10-bit fixed point numbers for computation while the vertex processing hardware uses higher precision 16-bit floating point. Finally, the lack of logical operations means that pixel programs would not provide significantly more convenience to the programmer.

4. A new programming model

4.1. Overview

The cornerstone of this work is a simple C++ framework we have developed for writing general-purpose programs with vector operations. This framework stays fairly close to the underlying hardware; we do not develop an intricate abstraction or new language designed to hide the hardware. Nevertheless, the framework does important things:

it presents an abstraction of the hardware as a device for computing vector functions, exposes vectors as a C++ data type, manages memory for efficient computation on those vectors, mostly hides the fact that the hardware is designed to operate at quad-float granularity rather than on vectors of single floats, and takes care of interfacing with the underlying hardware, hiding the necessary bookkeeping data and operations from the programmer.

We would have liked to completely shield the programmer from the quad-float register concept, but because the hardware is fundamentally designed this way, and because opcodes do not all behave consistently (some operate on all four vector components simultaneously, while some operate on a single scalar component of a source register or distribute their results unevenly into the components of target registers), we would have had to create a new, higher-level assembly language to completely hide the quad-float registers from the user. In practice, this is only an issue with peculiar opcodes like LOG and EXP, and it does not affect most example programs discussed in this paper.

4.2. Framework components

4.2.1. DVector. The DVector class encapsulates a vector of data. When created, DVectors are assigned a fixed size and typed as containing either single-precision floating point numbers or unsigned bytes. Like the standard C++ vector class, the programmer can access individual elements of a DVector using the [] operator. When each DVector is created, our framework's memory manager works behind the scenes to allocate a buffer of video memory for it, as well as maintain state information.

For convenience, we provide helper functions to generate large vectors of various sizes and types, including random vectors and unsigned byte ranges.

4.2.2. DProgram. The DProgram class encapsulates the instructions of an assembly language program meant for the graphics hardware. The framework provides a function for creating a DProgram from an array of strings containing an assembly language program.

Programmers only need to write assembly instructions that perform the desired computation. Our framework transparently takes care of inserting additional assembly instructions for bookkeeping before a DProgram is executed on the hardware. While the precise calling convention for assembly language programs depends on the programmer's choice of functional semantics (specified through a DFunction object, described in the next section), assembly programmers can depend on the framework to load source attribute registers with program inputs before each assembly program invocation, and programmers are responsible for placing computed return values in output attribute registers before returning.

4.2.3. DFunction. The DFunction class, the heart of our framework, encapsulates three things:

- a DProgram,
- a choice of functional semantics for the DProgram, and
- bindings for the constant registers.

The framework supports six kinds of functional semantics, shown below, where v represents an n-element vector and s represents a scalar:

- unary: $v \to v$ and $v \to s$
- binary: $[v,v] \to v$ and $[v,v] \to s$
- ternary: $[v,v,v] \to v$ and $[v,v,v] \to s$.

For instance, a DFunction with $[v,v] \to v$ semantics takes a pair of n-element vectors as input and computes an n-element vector as output.

The specific functional semantics assigned to a DFunction determine the conventions that the framework will follow in calling the associated DProgram. Each type of DFunction has an execute() method which takes an appropriate number of DVectors as input and returns either a DVector or a scalar as appropriate. For instance, a binary $[v,v] \to v$ DFunction's execute() method takes two DVectors as input. To compute this binary function, the framework breaks the input DVectors into a series of quad-floats, pairs of which are placed in registers v[1] and v[2]. The binary DProgram is expected to read its input from these two registers and store its results in register o[1]. If the input DVectors have size n, this means that the assembly routine stored in the DProgram will be called $\lceil n/4 \rceil$ times in order to completely process the input vectors.

Because the current hardware architecture does not give us a way to preserve state information between assembly routine invocations (since the temporary and output registers are zeroed before each vertex program invocation), for DFunctions whose semantics require producing scalar output our framework uses the computer's CPU to sum the results produced by each individual vertex program invocation in order to generate the final scalar result.[2] Naturally, this limits the variety of scalar-valued functions that can be computed using our framework, but architectural limitations make greater generality difficult. In order to perform as much computation on the hardware as possible and limit the amount of work done by the CPU, our assembly calling convention for scalar-valued functions saturates the source attribute registers as completely as possible, making the most data available to each vertex program invocation. For instance, while computing a unary $v \to s$ function,

[2] This sum can also be performed on the graphics hardware using a feature called "pixel blending." However, our experiments suggest that each pixel blending update causes the whole graphics hardware pipeline to stall, significantly reducing overall vertex program throughput.

we break the input vector into blocks of 15 quad-floats and place them in registers v[1] through v[15]. The assembly program is still expected to store a single result in register o[1]. If the input DVectors have size n, this means that the assembly routine stored in the DProgram will be called only $\lceil n/15 \rceil$ times in order to completely process the input vectors, and the CPU only needs to sum $\lceil n/15 \rceil$ values to produce a final output scalar.

4.2.4. DSemaphore. A major advantage of using graphics hardware as a computational co-processor is that the CPU is completely idle while the graphics hardware is working (*i.e.* using DMA [Direct Memory Access] to retrieve its input, performing computation, and writing its results to a memory buffer). Because the CPU is free to perform unrelated computation as the GPU is churning away; the aggregate computational throughput of the computer should be viewed as the sum of the maximum computational throughput of the CPU and of the maximum computational throughput of the GPU.

By default, each DFunction's execute() method causes the CPU to idle until the graphics hardware finishes processing. However, our framework also provides an executeAsync() method which causes the GPU to begin computing asynchronously, freeing the CPU to compute simultaneously. The executeAsync() method returns a DSemaphore object with one method, waitForCompletion(). Should the CPU reach a point where it needs to use the results computed by the GPU, calling waitForCompletion() causes the CPU to wait for the GPU to finish.

To avoid exaggerating the quantitative performance of the graphics hardware, ***none of the empirical results reported in this paper make use of executeAsync() to simultaneously perform computation on both the CPU and GPU***. A goal of this paper is to show that modern graphics hardware can—working alone—outperform modern CPUs for large vector computation.

4.2.5. Example. The code fragment shown in Figure 3 uses our framework to compute the factorial function for a vector of single-digit integers. The assembly fragment uses two techniques to avoid branching: conditional set opcodes are used to select vector elements that need to be affected by subsequent operations, and the main loop has been manually unrolled 8 dimes (enough to compute the factorial of any single-digit integer). While both these techniques potentially perform a lot of unnecessary computation for some input elements, the vector processing engine is fast enough to compensate for this extra work, as we demonstrate later in this paper. Naturally, such techniques are not sufficient to always avoid branching. To implement a branch, the programmer would create two DFunctions, one for the code before the branch, and one for the subsequent code. To decide whether to branch, the programmer would examine the DVector output from the first DFunction with the main CPU with standard C++ code. There is no way with our current framework to branch in different directions for each individual element of a vector.

4.3. Framework implementation

We implemented our framework using the OpenGL programming interface because it allows lower-level control of the underlying hardware than the DirectX interface. The OpenGL extensions for managing banks of video memory proved essential to this research—we discovered that having the hardware perform DMA from the computer's main memory is extremely slow. We use the CPU to transfer blocks of data to video memory in large, single passes, and then let the GPU retrieve its input from video memory directly. Output from vertex programs uses the OpenGL "pbuffer" feature to direct the results of the rendering pipeline to invisible video memory buffers, rather than the screen.

To facilitate experimentation and also to make it possible to run and debug programs on a computer without a programmable GPU, our framework includes a simulated implementation of the vertex engine. This simulator is only functionally identical to the graphics card; we did not have enough information about the low-level hardware architecture of the GeForce4 to attempt a hardware simulation or to get details such as timing correct. Our work on the simulator proved to be helpful for debugging our assembly routines.

Another important implementation detail was our method of retrieving output data from the graphics card for storage in the output DVector. Unfortunately in OpenGL there is no mechanism for redirecting vertex program output directly to a buffer in memory without having it also pass through the rest of the graphics pipeline and get converted to pixel data. Hence, we were forced to direct the outputs of the vertex programs into pixel buffers, then grab the resulting chunks of pixels. While OpenGL supports retrieving a chunk of pixels as floating point numbers, the results are only as accurate as the underlying 8-bit pixel representation! This means that the results returned by our framework, even though they were computed internally at single precision, suffer a significant loss of precision when retrieved from the graphics card. Such a situation is clearly suboptimal, but it was the best we could do. Clearly there is a need to enhance OpenGL to allow rendering of vertex program output to a memory buffer. DirectX 8.1 has such a feature, but the particular function that implements it always uses a software-based emulation of a GPU even if a graphics card is available. This unusual behavior is documented in the DirectX SDK [11] and reasons for it are not given, but it is likely that older generations of graphics hardware were not

```
DVector* inputVec  = allocDVector(5, DV_UCHAR);
DVector* outputVec = allocDVector(5, DV_UCHAR);

// Fill the test input vector.
(*inputVec)[0] = 6;
(*inputVec)[1] = 2;
(*inputVec)[2] = 3;
(*inputVec)[3] = 1;
(*inputVec)[4] = 4;

char* program[] = {
          // The result will be stored in R1.
          // We'll use R2 to store the next number to
          // multiply into R1.
          // R3 is a bit vector to decide which
          // elements still need to be multiplied.
          // R4 will hold the result of multiplying
          // R3 by the next constant.
          "MOV R1, v[1];",
          "MOV R2, v[1];",

          // If R2>=2 decrement R2.
          // If R2< 2 leave unchanged.
          "SGE R3, R2,  c[12];",
          "SLT R4, R2,  c[12];",
          "MUL R5, R3, -c[11];",
          "ADD R2, R2,  R5;",

          // Multiply the updated values into R1.
          "MUL R6, R3, R2;",
          "MUL R7, R1, R6;",
          "MUL R8, R1, R4;",
          "ADD R1, R7, R8;",

          // Unroll once.
          "SGE R3, R2,  c[12];",
          "SLT R4, R2,  c[12];",
          "MUL R5, R3, -c[11];",
          "ADD R2, R2,  R5;",
          "MUL R6, R3, R2;",
          "MUL R7, R1, R6;",
          "MUL R8, R1, R4;",
          "ADD R1, R7, R8;",
          ...
          // Unroll six more times.
          ...

          // Store the result in the output register.
          "MOV o[COL0], R1;",
          0 };

DFunctionUnary programFunc( makeProgram( program ) );
programFunc.setConstant( 11, 1 );
programFunc.setConstant( 12, 2 );

programFunc.execute( inputVec, outputVec );
```

Figure 3. A vector implementation of the single-digit factorial function.

fast enough to justify a round trip of data from memory to the GPU, and perhaps the DirectX designers assumed that this would always be true. As our results in the next section demonstrate, modern graphics architectures have invalidated that assumption.

5. Results

5.1. Overview

To judge the effectiveness of our framework for solving general-purpose programming problems, we obtained a GeForce4 Ti4600, the fastest consumer GPU manufactured by NVIDIA at the time of writing. Advertised specifications for this card include:

- 136 million vertices per second,
- 1.23 trillion operations per second, and
- 10.4GB/sec memory bandwidth.

The 1.23 trillion "operations per second" number seems somewhat high, and presumably it reflects micro-ops used by the chip internally across all functional units, not just the vertex processing engine. If we accept the 136 million vertices/second number, a more reasonable estimate for the GPU's vertex program speed in terms of vertex program opcodes might be:

Opcodes per program estimate * Vertices per second
= 15*136 million
= 2.04 billion opcodes per second.

While this number still seems high, the empirical results presented in this section do demonstrate remarkable performance.

For testing purposes, we compared CPU and GPU implementations of various toy routines, as well as two more realistic problems—dense matrix multiplication and 3-SAT—on a 1.5GHz Pentium IV computer with 1GB of RAM. The GeForce4 had 128MB of onboard video RAM (none of our example programs used all of this space). The CPU implementations of each algorithm were written in straight C++ and used raw arrays rather than the standard vector class to avoid unneccessary overhead. All tested programs were compiled with Microsoft Visual C++ 6.0 using the "maximize speed" optimization setting.[3]

Timing measurements were taken using the test computer's real time clock. This clock had a (relatively coarse)

[3]Our framework does not incorporate a GPU-specific compiler or any kind of automated optimizer for the GPU assembly language, so an argument could be made that comparing results with the C++ optimizer turned off might be a more fair comparison. Naturally, one can imagine developing algorithms that optimize the way the GPU is used. However, even without such a GPU code optimizer, our results compare very favorably with heavily optimized native code.

resolution of 10 milliseconds, which explains why some of the shorter test cases measured as taking 0 milliseconds.

5.2. Test programs

We used the following toy programs as test cases; each consists of a single DFunction:

arithmetic Evaluates $\log(\pi x^3)$ for each element of a vector. The program contains 13 opcodes.

exponents Computes x^{50} for each element of a vector. The program includes 49 opcodes.

factorial Computes the factorial function for a vector of random integers, each between 0 and 9. This program was illustrated in Figure 3. It simulates branching using the SLT and SGE opcodes and manual loop unrolling. The program contains 59 opcodes.

muls Performs a variety of multiplies on each element of a vector; the total number of opcodes can be made to vary from 10 to 120. We use this to study the effects of increasing program sizes.

In Table 4, we present results showing how computation times for both the CPU and GPU compare with respect to increasing input vector sizes for three of the test programs. Figure 4 illustrates the data for the arithmetic program graphically.

Table 4. The effects of increasing vector sizes.

Test case		Vector size 10000	100000	1000000	10000000
arithmetic	CPU	0	10	120	1220
	GPU	10	10	20	190
exponents	CPU	10	50	420	4180
	GPU	0	0	30	250
factorial	CPU	0	20	170	1650
	GPU	10	10	40	330

Run times in milliseconds

It is difficult to accurately compare the performance the CPU and GPU for the smallest vector size tested, since the clock resolution is too coarse. However, two of the test cases suggest that for such small vectors the overhead inherent in using our framework causes the GPU implementations to be slightly slower than the CPU implementations. However, on all the larger vector sizes, our graphics card implementations outperformed the CPU implementations. For the largest vector size, our GPU implementations were between 5.0 to 16.7 times faster than their CPU brethren.

These results may seem surprising or even shocking at first glance, but they make sense. Essentially, the graphics hardware allows us to establish a high-speed custom data processing pipeline. Once the pipeline is set up, data can

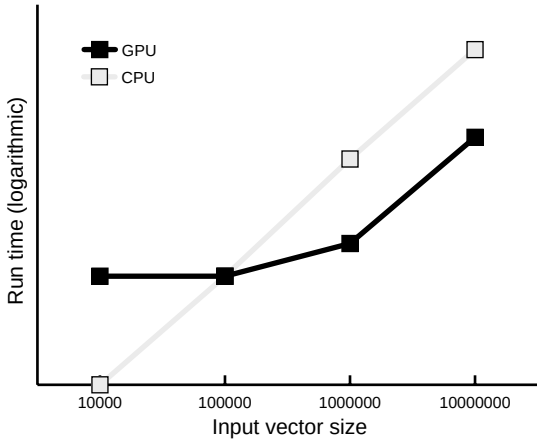

Figure 4. The effects of increasing vector sizes on the run time of the arithmetic program, plotted using a logarithmic scale on the y-axis. Though both the CPU and GPU run times increase in proportion to the input size, the GPU is significantly faster (up to 6.4 times quicker) for large vectors.

be streamed through with devastating efficiency. The two test cases where the GPU and CPU must perform roughly the same number of calculations, arithmetic and exponents, show that there is a correlation between how much work is being done on the graphics card and how strongly the GPU implementations outperform equivalent CPU implementations. The arithmetic test case, which executes 13 opcodes per vector element, beats the CPU implementation by a factor of 6.4 on the largest vector size. In comparison, the exponents test case, which executes 49 opcodes per vector element, is 16.7 times faster on the GPU. (The factorial test case is an oddity, because the GPU performs far more operations per element of the input vector than the equivalent CPU implementation. This is because the CPU implementation of the factorial function only needs to perform four multiplications to compute the factorial of the number 5, and can branch and begin processing the next vector element immediately after finishing the fourth multiplication. In contrast, the GPU must execute the entire unrolled loop shown in Figure 3 on every element of the input vector.) These results demonstrate how important it is to carefully plan how the graphics hardware will be used so that a maximal amount of work can be done within the GPU on each chunk of data submitted, and they suggest future opportunities for compiler research.

In order to explore the relationship between vertex program complexity and run times, we ran the muls test program on a 1,000,000 element vector, varying the total number of opcodes between 10 and 120. The results are shown in Figure 5.

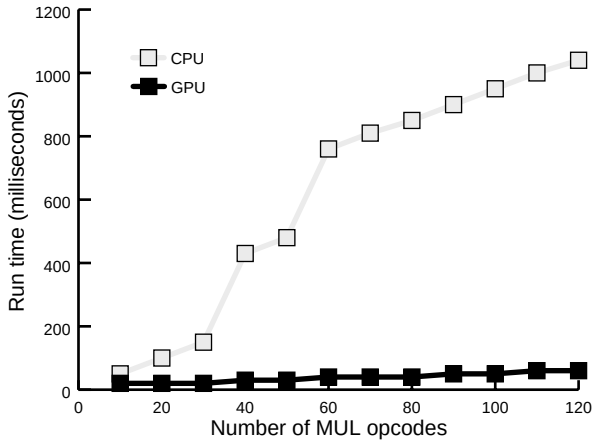

Figure 5. The effects of program size (number of multiplications) on the run time of the muls test program with a 1,000,000 element input vector. The CPU run time roughly doubles as the program complexity doubles, whereas the GPU run time grows much more slowly, merely tripling as the program complexity increases by 12 times.

While the run time of the CPU implementation approximately doubles as the number of multiplications doubles, the GPU run time grows far more slowly. In fact, even though we increase the number of opcodes by a factor of 12, the run time of the GPU implementation only increases by a factor of 3. Programmers should thus feel free to write very complicated assembly routines, because additional complexity is relatively cheap on the GPU as compared to a conventional CPU. These results indicate that the current hardware architecture's limitation on vertex programs being a maximum of 128 opcodes is suboptimal, because programmers could fruitfully make use of longer vertex programs. Moreover, these results suggest that either the hardware's transfer of data between video memory and the GPU or the CPU's fetch of the computational results from video memory is more of a bottleneck than the hardware's capacity for computation. (If we were pushing the vertex processing hardware to its limits, we would expect the GPU's run time to double as the number of opcodes doubles.)

In order to better understand the bottlenecks, we instrumented a 50 opcode version of the muls example with code to measure the time to retrieve data from the video card. The results, gathered from a run with a 5,000,000 element input vector, are shown in Table 5.

Observe that only 21% of the runtime is spent retrieving data, indicating that the major bottleneck is in sending data to the GPU. The time for all other types of processing in the program is negligible. One consequence of these results is that future hardware architectures should concentrate on finding more efficient ways to "feed the beast."

Table 5. Output retrieval time for the muls test.

Total run time	140 milliseconds
Time to retrieve output	30 milliseconds

5.3. A more realistic test: Matrix multiplication

The test programs demonstrated earlier were enlightening, but because each is implemented with only a single DFunction, they do not reflect typical uses of our framework to solve real problems. To adequately demonstrate our framework, we implemented a dense matrix multiplication algorithm.

Our matrix multiplier begins by allocating two large DVectors; one to hold the first source matrix and another to hold the transpose of the second source matrix. The algorithm also allocates an array in main memory to hold the result of the matrix multiplication. Our algorithm performs multiplication in the naïve way: for each (i, j)-th element of the output matrix, we use a $[v, v] \rightarrow s$ DFunction to multiply one row of the first matrix with one column of the second matrix. We store the scalar result in the (i, j)-th location in the output matrix and then move on to the next output matrix element. Our results are shown in Figure 6.

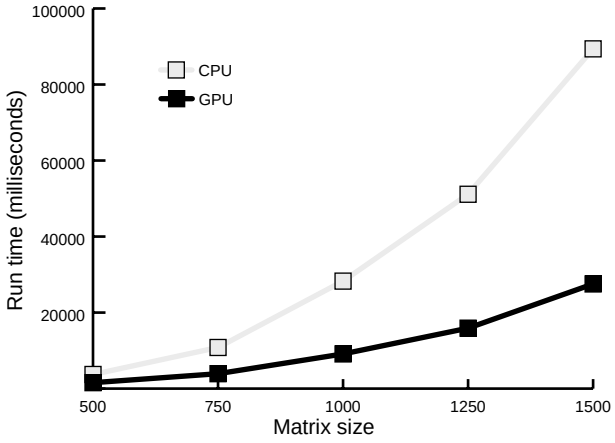

Figure 6. Matrix multiplication run time as a function of matrix size. For a 1500x1500 matrix, the GPU outperforms the CPU by a factor of 3.2.

While the runtimes of both implementations have $O(n^3)$ growth rates, the GPU implementation of matrix multiply significantly outperforms the CPU implementation, and this performance differential increases as the matrix size increases. For a 1500x1500 matrix, the GPU is 3.2 times

faster than the CPU. Much of this advantage undoubtedly stems from the graphics hardware's ability to perform more than one floating-point multiplication per clock.

5.4. Another realistic test: 3-SAT

As another illustration of our programming framework, we implemented a solver for the 3-satisfiability problem [3-SAT]. Our solver uses a simple, common genetic algorithm [19] to search for solutions to 3-SAT problems that are too large to exhaustively enumerate. Similar, though more sophisticated, algorithms for solving 3-SAT have in the past been applied to traditional vector processors [9, 12].

Our algorithm works by encoding the set of clauses into the elements of a DVector, placing variable assignments in the constant registers, and using a unary $v \rightarrow s$ function to determine how many clauses are currently satisfied (if there are more variables than can be accommodated in the constant registers, we perform multiple DFunction evaluations with a different set of constant registers each time). We then randomly select a variable, flip its value, and again compute how many clauses are satisfied. If the new variable value is an improvement, we keep it. Otherwise, we revert to our old set of variable values. If we cease to make progress for a number of iterations, we discard the entire set of current variable assignments, generate a new set, and continue from there. After a fixed number of iterations without finding a solution, we give up and report "no solution found." Our results are shown in Figure 7.

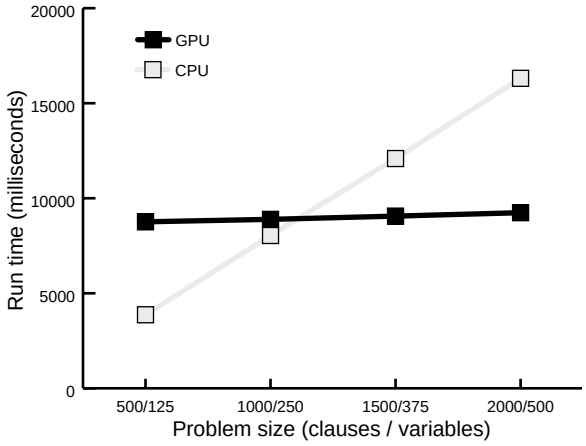

Figure 7. Run time as a function of problem size for 100,000 iterations of our 3-SAT solver. All problem instances selected are unsatisfiable.

Due to the way we structured our GPU implementation of the 3-SAT solver, our run time for each iteration essentially only depends on the number of clauses, and the increase in vector size between 500 and 2000 clauses is not sufficient to greatly affect the run time of our GPU implementation. In contrast, the run time of the CPU implementation increases very noticeably with the increasing number of clauses.

6. Analysis

Though the framework we have developed in this paper is effective for real problems, our framework could be much more useful and powerful if a number of limitations of the graphics hardware were addressed. In this section, we discuss some of the architectural weaknesses we discovered. Many of these limitations could be easily resolved with minor changes to the hardware and programming interfaces. By adopting such changes, graphics hardware manufacturers could potentially increase the size of the market for their products. Hence, we feel that these observations could have a real impact on future graphics architectures.

A faster memory interface: Transferring data to the GPU is a bottleneck that makes it difficult to fully exploit the potential computational throughput of the vertex processing engine. Also, this transfer introduces overhead that makes applying the GPU to small vector problems inefficient.

Hardware to perform DMA from the graphics card to main memory: While the graphics card can fetch data from memory without CPU intervention, freeing the CPU for other types of calculation, DMA hardware does not exist for transferring data in the opposite direction. Since any significant use of graphics hardware for general-purpose computation requires retrieving data from the graphics card, such hardware is potentially important. Moreover, this hardware would also help accelerate many popular multi-pass graphics rendering algorithms.

Operating system management of video memory: If graphics hardware becomes a popular way to accelerate computation, multiple processes may contend for video memory. Currently, there is no mechanism for dealing with this; each process is responsible for periodically checking whether memory it has allocated on the video card has been trampled by another process. If the memory has been overwritten by another process, the process must reallocate its memory buffers. Without OS-level management of video memory, frequent context switches may lead to thrashing.

Better control over arithmetic precision: Suggestions from the previous paragraph would avoid the ugly step of us having to access single-precision floating point numbers by reading a bank of 8-bit pixels. However, a larger issue is that the graphics card is designed for speed rather than accuracy. The LOG opcode, for instance, is fast partially because it is not as accurate as a traditional $log()$ function. The NVIDIA engineers were insightful enough to have the LOG function place a few scalars in output registers that can be used to

increase the precision of logarithmic results (the mechanism for doing this is somewhat esoteric and is described in the NVIDIA OpenGL SDK [13]), but it should be possible to have a true, high-precision LOGHI opcode. Unfortunately, it may be difficult to implement such an opcode while still preserving the assembly language's uniform semantics (all opcodes take the same amount of time).

Logical Boolean operations: It is somewhat surprising that such simple operations are not available, especially since they are very common in image processing. Simulating logical operations using other opcodes is tedious and many times more inefficient than a good hardware implementation.

Ability to preserve state across vertex program invocations: This is perhaps our most important recommendation. Often, the programmer wants to inspect a vector for some global property. For instance, a programmer might want to inspect a vector to determine if it contains the number 42. Ideally, the programmer should be able to get more than a yes/no answer—the vertex program should be able to return the vector index where number 42 is located. Currently, such global vector operations must be done by the CPU rather than the GPU because there is no way of sharing data between multiple vertex program invocations. At the beginning of each vertex program, the temporary and output registers are cleared and set to zero. If this zeroing was an optional behavior that could be turned on or off, the programmer would be able to share state across vertex program invocations. OpenGL is filled with a myriad of Boolean flags for controlling the hardware, so such an optional behavior would not be unusual. Though this change may initially seem like a very straightforward modification of the current architecture—since it involves *not* doing something that is currently being done— it introduces complications which inevitably compromise either the semantics of the current instruction set architecture or constrain the implementation of the underlying hardware. By zeroing the output registers between vertex program invocations, the current architecture ensures that programmers have the illusion that the GPU has a single functional unit that processes vertices in sequence, even though the actual hardware implementation has multiple functional units processing vertices in parallel, and each functional unit has its own set of output and temporary registers. Not zeroing the registers would require either a single shared set of output registers, or synchronization between functional units. Alternatively, the architecture could be extended to include a new bank of global registers that would be accessible both from OpenGL and from any vertex program. These registers would be shared between functional units and incur the inevitable overhead of synchronization, but standard graphics applications not needing to share data between vertex program invocations would simply avoid accessing the global registers and suffer no performance penalty. Regardless of how vertex state preservation is accomplished, we feel that it would dramatically improve the speed of many general-purpose algorithms.

A compiler: We have demonstrated in this paper that the most efficient way to use the graphics hardware is to supply it with basic blocks that contain as many instructions as possible. Coding such blocks in assembly is tedious and error prone. Moreover, techniques such as loop unrolling for getting around the hardware's lack of branching are more practical for a compiler to implement. Also, a compiler could make more efficient use of the vertex attribute registers. This would reduce the number of vertices that would need to be sent to the GPU, reducing the costs associated with transferring data from main memory to the graphics hardware. While some progress has recently been made in this direction [14], researchers need to review the catalogue of modern, sophisticated compiler techniques and determine which are most fruitfully applied to graphics hardware. For instance, trace scheduling, which is inherently designed for processors in which branching may be inconvenient and for which large basic blocks are desirable (*e.g.* VLIW processors), could potentially deliver dramatic performance improvements.

7. Discussion and future work

In this paper we have described a programming framework we devised for solving general-purpose problems using graphics hardware. We demonstrated the framework's surprising effectiveness at accelerating highly regular operations on large vectors. We then described implementations of matrix multiplication and 3-SAT that used our programming framework. We investigated the sources of bottlenecks and proposed minor architectural enhancements which would help to reduce the bottlenecks and make general-purpose programming more effective on modern programmable graphics hardware.

The results described in this paper are much better than we expected. Our feeling prior to beginning this research was that bottlenecks, architectural limitations, or simply slow hardware would have put the GPU much farther behind the CPU, even for the simplest examples. That our dense matrix multiplication algorithm was so fast surprised us.

Graphics hardware similar to the hardware discussed in this paper is available in the majority of consumer oriented desktop PCs sold today, as well as in the Microsoft Xbox game console. This means that commodity hardware is being shipped with vector units capable of previously inconceivable—and largely untapped—computational power. As evidenced by the increasing popularity of distributed computing clusters and the decreas-

ing popularity of supercomputers, we believe that there is sufficient demand for low cost alternative computing technologies using commodity hardware to make our approach a valuable contribution. If current performance trends in graphics hardware continue, in a few years it may not be unusual to go into the server room of a biotech research company and find clusters of cheap PCs containing multiple low cost, high performance video cards selected specifically for their vector capabilities. One can even imagine the emergence of a class of thin "blade" computational servers designed specifically to accomodate multiple graphics chips, rather than multiple CPUs.

Of course, to reach those kinds of goals, much research remains to be done and this paper has only scratched the surface. In the future, we intend to evaluate our framework in the context of larger, real-world problems in areas such as computational biochemistry and mechanical engineering. We are also very interested in exploring the performance characteristics of networked clusters of GPUs, and potentially developing a cluster-aware version of our framework. We also hope to investigate applications of modern compiler techniques to graphics hardware. Part of the reason we believe that the emergence of powerful desktop GPUs is so exciting is because it lets us reexamine much of the rich body of research that has been done about vector processors and potentially apply it on a much larger scale than has previously been possible.

Acknowledgements

An excellent talk by Pat Hanrahan encouraged us to start investigating the computational capabilities of modern graphics hardware. We are also grateful to Stephen Spencer for helping us set up the hardware used in this research.

References

[1] ATI Corporation. *RenderMonkey Version 0.5 Beta Documentation*, August 2002.

[2] R. Barzel and D. Salesin. Patchwork: A fast interpreter for a restricted dataflow language. *The Journal of Systems and Software*, 6(3):251–259, August 1986.

[3] N. England. A graphics system architecture for interactive application-specific display functions. *IEEE Computer Graphics and Applications*, 6(1):60–70, January 1986.

[4] H. Fuchs et al. Pixel-planes 5: A heterogeneous multiprocessor graphics system using processor-enhanced memories. In *Computer Graphics (Proceedings of SIGGRAPH 89)*, volume 23, pages 79–88, Boston, Massachusetts, July 1989.

[5] J. K. Hao. A clausal genetic representation and its related evolutionary procedures for satisfiability problems. In *Proceedings of the International Conference on Artificial Neural Nets and Genetic Algorithms*, pages 289–292, Ales, France, April 1995.

[6] A. Kunimatsu et al. Vector unit architecture for emotion synthesis. *IEEE Micro*, 20(2):40–47, March/April 2000.

[7] A. Levinthal and T. Porter. CHAP: A SIMD graphics processor. In *Computer Graphics (Proceedings of SIGGRAPH 84)*, volume 18, pages 77–82, Minneapolis, Minnesota, July 1984.

[8] E. Lindholm, M. J. Kilgard, and H. Moreton. A user-programmable vertex engine. In *Proceedings of ACM SIGGRAPH 2001*, pages 149–158, August 2001.

[9] A. M. Logar, E. M. Corwin, and T. M. English. Implementation of massively parallel genetic algorithms on the MasPar MP-1. In *Proceedings of the 1992 ACM/SIGAPP Symposium on Applied Computing*, pages 1015–1020, 1992.

[10] M. D. McCool. SMASH: A next-generation API for programmable graphics accelerators. Technical report CS-2000-14, Computer Graphics Lab, University of Waterloo, 2000.

[11] Microsoft Corporation. *DirectX Software Development Kit*, 2001. Version 8.1.

[12] N. Nemer-Preece and R. Wilkerson. Parallel genetic algorithm to solve the satisfiability problem. In *Proceedings of the 1998 ACM Symposium on Applied Computing*, pages 23–28, 1998.

[13] NVIDIA Corporation. *OpenGL Extension Specifications*, May 2001.

[14] NVIDIA Corporation. *Cg Language Specification*, August 2002.

[15] NVIDIA Corporation. *NVSDK Software Development Kit*, 2002. Version 5.2.

[16] M. Olano. *A Programmable Pipeline for Graphics Hardware*. Ph.D. thesis, University of North Carolina at Chapel Hill, 1998.

[17] B. Paul et al. *The Mesa 3D Graphics Library*, 2002. Version 4.0.3.

[18] K. Proudfoot et al. A real-time procedural shading system for programmable graphics hardware. In *Proceedings of ACM SIGGRAPH 2001*, pages 159–170, August 2001.

[19] A. Whitley. A genetic algorithm tutorial. Technical report CS-93-103, Department of Computer Science, Colorado State University, 1993.