A Framework and Analysis of Modern Graphics Architectures for General Purpose Programming

Chris Thompson Sahngyun Hahn
University of Washington

Abstract

Modern graphics hardware has become so powerful that raw performance enhancements are increasingly unnecessary. As such, recent graphics hardware architectures have begun to de-emphasize performance enhancements in favor of versatility, offering rich ways of programmatically reconfiguring the graphics pipeline. A side effect of this versatility is that new, powerful general purpose constructs similar to vector processors are appearing in commodity hardware. In this paper, we explore whether current graphics architectures could be used to accelerate domains where vector processors would traditionally be used. We develop a programming framework that allows us to solve computational problems using graphics hardware, and we apply it to various problems. We compare the speed of our graphics card implementations to standard CPU implementations and demonstrate startling performance improvements in some cases, as well as room for improvement in others. We analyze the bottlenecks and propose minor architectural extensions to current graphics architectures which would improve their effectiveness for solving general purpose problems.

1 Introduction

Modern graphics hardware has reached a turning point where performance enhancements are becoming less and less necessary. As McCool observes, 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 [9]. As such, recent graphics hardware—in both research and commercial designs—has begun to de-emphasize performance enhancements in favor of versatility, offering rich ways of programmatically reconfiguring the graphics pipeline [[8], [9], [13]]. A side effect of this versatility is that new, powerful general purpose constructs similar to vector processors are appearing in commodity PC hardware, 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 can perform more than 1.2 *trillion* operations per second. Most of the 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. We believe that 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, whereas powerful graphics chips are already widely available. In this research, we investigate the programming, performance, and limitations of a recent graphics architecture on a pair of non-graphics problems, and we propose ways in which future iterations of the architecture could be improved to make it more suitable for such applications.

We begin by describing a programming framework we have devised that allows us to conveniently solve computational problems using graphics hardware. Our system includes a simulator for the graphics interface that can run on computers without special hardware. We implement a number of toy algorithms with our framework, and we also apply the framework to a real problem: 3-satisfiability, solved using a genetic algorithm approach. We then compare the speed of our graphics card implementations to CPU implementations. In some cases, the results are startlingly impressive, yet there is also significant room for improvement. We analyze the bottlenecks and propose minor architectural extensions to current graphics architectures which 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 processors.

2 Prior work

While the last decade was dominated by fixed function non-programmable graphics architectures, some programmable graphics architectures were also explored. Some early systems, such as Pixar's CHAP [7] and the commercially available Ikonas platform [2], 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 [3] 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 [5]. 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 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, as new pixels are composited with old pixels on the screen, the compositing process can be fully controlled through assembly language *pixel programs*. Each programming model is designed to limit implementation complexity. For instance, 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.* [14] 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.

3 Vertex and Pixel Programs

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:

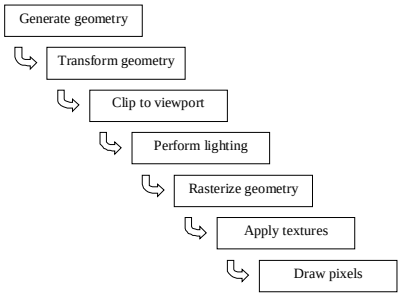

Figure 1 The traditional graphics pipeline.

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. The hardware then performs lighting and color calculations. Next, the geometry is converted from pure vectors to a pixel-based raster representation. Textures are applied, and then the raster version of the geometry is composited onto the screen.

Most contemporary programmable architectures, including architectures designed to conform to the modern DirectX 8 standard interface [1], revise the standard pipeline as shown in Figure 2.

[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.

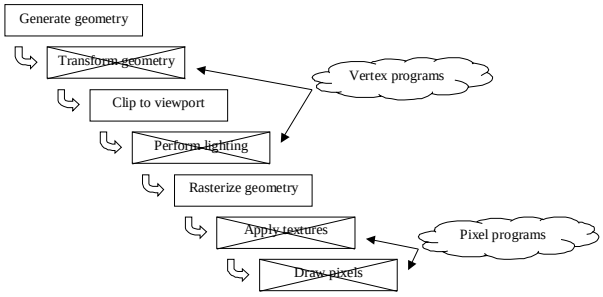

Figure 2 A programmable graphics pipeline.

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 pixel compositing stages are collapsed into a single stage where the output pixels are determined by pixel programs written in assembly language.

For the remainder of this paper, we concentrate on a specific target, the NVIDIA GeForce4 chipset [[8], [12]]. This chipset is representative of modern graphics architectures. Its programming interface is identical to the interface for NVIDIA's earlier GeForce3 chipset, as well as for the chipset in Microsoft's Xbox game console. It conforms fully to the DirectX 8 interface and can also be accessed using a more basic application programming interface called OpenGL.

3.2 Vertex Programs

In this section, we describe the GeForce3'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.

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

Table 1 The source attribute registers.

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

Register name	**Meaning**
o[HPOS]	Clip space object position
o[COL0]	Primary color (front)
o[COL1]	Secondary color (front)
o[BFC0]	Primary color (back)
o[BFC1]	Secondary color (back)
o[FOGC]	Fog coordinate
o[PSIZ]	Point size
o[TEX0]-o[TEX7]	Texture coordinates

Table 2 The output attribute registers.

There are 17 instructions that can be used in vertex programs. Each generates a single result and places it in a destination register. The instructions are summarized below:

Opcode	**Inputs**	**Output**	**Description**
ARL	S	Index	Address register load
MOV	V	V	Move
MUL	VV	V	Multiply
ADD	VV	V	Add
MAD	VVV	V	Multiply and add
RCP	S	SSSS	Reciprocal
RSQ	S	SSSS	Reciprocal square root
DP3	VV	SSSS	3-component dot product
DP4	VV	SSSS	4-component dot product
DST	VV	V	Distance vector
MIN	VV	V	Minimum
MAX	VV	V	Maximum
SLT	VV	V	Set on less than
SGE	VV	V	Set greater/equal than
EXP	S	V	Exponential base 2
LOG	S	V	Logarithm base 2
LIT	V	V	Light coefficient formula

Table 3 The vertex program instruction set. “S” indicates a scalar and “V” indicates a vector.

3.3 Pixel Programs

Since vertex programs replace fixed-function hardware that performs computations (transformation, lighting) in geometric space, vertex programs operate on floating point numbers. In contrast, pixel programs are intended to replace pixel-level operations such as transparency and compositing. Pixel programs thus operate on the quad-byte vectors that hold red, green, blue, and alpha (transparency) color information.

Pixel programs share the same programming interface and register model as vertex programs, but the specific set of source and output attribute registers, as well as the specific opcodes, reflect quantities and functions relevant to pixel composition. For instance, the add and dot product opcodes are still present, but there are is also a new opcode for linear interpolation as well as a custom opcode that performs bump mapping computations. Somewhat surprisingly, no opcodes exist for logical operations (AND, XOR, *etc.*). In many cases, these operations can be simulated using other opcodes. Since logical operations are important in image processing, future architectures will almost certainly include logical opcodes.

In developing our programming framework, we decided not to explore the use of pixel programs. Though such an investigation would be interesting, we felt that our efforts would be more fruitfully applied to other areas, for two reasons. First, the bytes that pixel programs manipulate are inherently lower precision than the single precision floating point numbers that vertex programs manipulate. Second, the lack of logical operations means that pixel programs would not provide significantly more power or 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 designed to hide the hardware. Nevertheless, the framework does important things: it presents vectors as a C++ data type, 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 scalars and 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 Components

4.2.1 DProgram

The DProgram class encapsulates the instructions of an assembly language program meant for the graphics hardware. The framework provides a function for converting an array of strings specifying an assembly language program into a DProgram. When this conversion occurs, the framework adds three things transparently to the user: a prologue and epilogue mandated by the programming interface, as well as a MOV instruction to transfer v[OPOS] into

o[HPOS] so that the output stream of vertices will be ordered identically to the input vertices. When a DProgram is created, the framework also parses the program text and converts it to an internal representation that our simulator can understand. Finally, the DProgram class maintains two other important pieces of information: a unique program ID (useful for managing multiple programs loaded into the graphics hardware) and a Boolean flag indicating whether the program is currently loaded into the hardware.

4.2.2 DChunk

Each instance of the DChunk class represents a "chunk" of floating point data. Internally, DChunk is implemented using the C++ vector<float> class, and the interface of DChunk follows vector semantics, so DChunk might have reasonably been called DVector. (We decided against this name in order to avoid confusion about what might be stored in a DChunk. One could, for instance, store a matrix in a DChunk.) For convenience, we provide helper functions to generate large vectors of various sizes and types, including random vectors and integer ranges. Apart from the underlying vector<float>, no other data is stored in the DChunk class.

4.2.3 DBlock

The DBlock class, so named because of its similarity to a basic block in a regular programming language, encapsulates three things:

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

Our framework supports four kinds of functional semantics: unary, binary, and ternary functions from vector(s) to a vector, and a cross product function that takes two vectors as input and returns a matrix (a DChunk whose size is the square of the size of the input DChunk). The specific functional semantics of a DBlock determine the calling conventions that the DProgram must follow. Each type of DBlock has an execute() function which takes an appropriate number of DChunks as input and returns a single DChunk. For instance, a binary vector->vector function takes two DChunks as input. For this binary function, the framework breaks the input DChunks 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].

4.2.4 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 7 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 DBlocks, one for the code before the branch, and one for the subsequent code. To decide whether to branch, the programmer would examine the DChunk output from the first DBlock 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.

```
DChunk inputChunk;
DChunk outputChunk;

// Prepare a test vector.
inputChunk.setSize(5);
inputChunk.getDataRef()[0] = 6;
inputChunk.getDataRef()[0] = 2;
inputChunk.getDataRef()[0] = 3;
inputChunk.getDataRef()[0] = 1;
inputChunk.getDataRef()[0] = 4;

char* program[] = {
          // The result will be stored in R1.
          // We use R2 to count down by 1 each time.
          "MOV R1, v[11]",
          "MOV R2, v[11]",

          // 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 five more times.
          ...

          // Store the result in the output register.
          "MOV o[1], R1",
          0 };

DBlockUnary programBlock( makeProgram( program ) );
programBlock.setConstant( 11, 1 );
programBlock.setConstant( 12, 2 );


outputChunk = programBlock.execute( inputChunk );
```

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

4.3 Implementation Notes

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. We did not, however, have a chance to modify the simulator to experiment with architectural enhancements.

Another important implementation detail was our method of retrieving output data from the graphics card for storage in the output DChunk. Unfortunately in OpenGL there is no mechanism for redirecting vertex program output to a buffer in memory.

Individual vertex program results can be queried, one by one, after each vertex is sent through the graphics card, but we felt that retrieving results this way would be much too slow. It would also inhibit the card's ability to process vertex data asynchronously (calculating even as the CPU is working to send the card more data). Hence, we were forced to direct the outputs of the vertex program onto the screen, then grab the resulting chunk 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 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 [1] but the reasons for it are not given).

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. If we accept the 136 million vertices/second number, a more reasonable estimate for GPU's vertex program speed in terms of vertex program opcodes would be:

Max. opcodes per program $*$ Vertices per second =

128 $*$ 136 million = 17 billion opcodes per second.

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

For testing purposes, we compared CPU and GPU implementations of various toy routines, as well as a more realistic 3-SAT solver, on a 1.5GHz Pentium IV computer. To ensure that the implementations being compared were structurally similar, we did not write custom native CPU code for each of the problems. Rather, our CPU tests involved running our simulator on the same code used for the GPU tests. Some might argue that comparing the GPU implementations with carefully hand-coded CPU implementations would be a more realistic comparison. However, such results would differ only by a constant factor from the results presented in the paper, and the overall asymptotic trends would remain the same.

To give a more realistic picture of the relative performance of the CPU and GPU, we provide one set of results where the C++ compiler optimizations were turned on, and one set where they were turned off. Our framework does not incorporate a GPU-specific compiler or any kind of GPU optimization algorithms, so the argument could be made that comparing results with C++ optimizations turned off is a more fair comparison. One can certainly imagine developing algorithms that optimize the way the GPU is used.

Timing measurements were taken using the computer's real time clock, since we could not be sure that CPU and kernel times would include time spent waiting for the graphics card. The real time clock on our test computer had a (relatively coarse) resolution of 10 milliseconds, which explains why some of the shorter tests cases measured as taking 0 milliseconds.

5.2 Test Programs

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

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

exponents – Computes x^{10} for each element of a vector. The program includes 10 opcodes.

muls – Performs a variety of multiplies on each element of a vector; the total number of opcodes can be made to vary from 2 to 100.

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.

Below, we present results showing how computation times for both the CPU and GPU compare and grow with increasing input vector sizes.

		Vector size			
		500	5000	50000	500000
arithmetic	cpu noopt	0	20	211	2163
	gpu noopt	10	0	30	331
	cpu opt	0	10	80	751
	gpu opt	0	0	10	170
exponents	cpu noopt	0	30	241	2434
	gpu noopt	10	0	30	330
	cpu opt	0	10	80	831
	gpu opt	0	0	20	161
factorial	cpu noopt	20	130	1222	12288
	gpu noopt	0	0	40	340
	cpu opt	0	40	431	4296
	gpu opt	0	10	20	161

Table 4 The effects of increasing vector sizes. Numbers indicate run times in milliseconds and "opt" denotes an executable compiled with optimizations turned on.

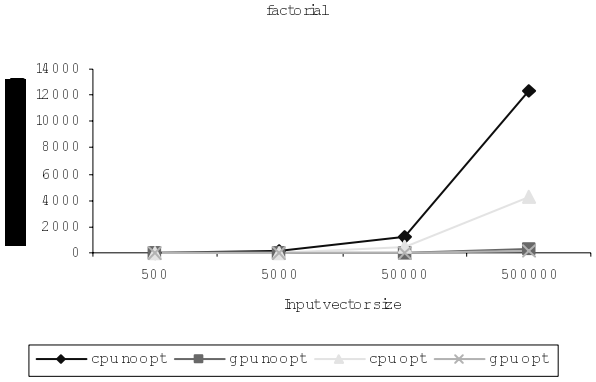

Figure 4 The effect of input vector size on the run time of the factorial program. Though both the CPU and GPU run times increase in proportion to the input size, the CPU run time grows more dramatically.

Because of the system timer's coarse 10 millisecond granularity, our results for a vector size of 500 are largely meaningless. It is difficult to accurately compare the performance the CPU and GPU for such small vectors. However, for all the larger sizes of input we tested, our graphics card implementations beat the CPU implementations. The superiority of the GPU is apparent even in cases where compiler optimizations were used. For one example program—factorial, having 59 opcodes—the optimized CPU implementation is more than 26 times slower than the GPU implementation on a vector size of 500,000. 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 be streamed through with devastating efficiency.

As Figure 4 illustrates, both the CPU and GPU run times increase in proportion to the input problem size, but the CPU implementation times grow much more quickly.

On the other hand, the GPU run times do *not* increase with program complexity. For a given input size, the GPU run times are essentially identical across all three example programs. This suggests that transmission of data between the main processor and the GPU is more of a bottleneck than computation time. In order to verify this, we ran the muls test program on a 500,000 element vector, varying the total number of opcodes between 2 and 100. The results are shown below.

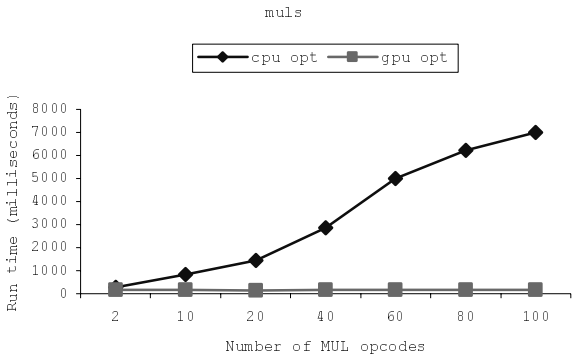

Figure 5 The effect of program size (number of opcodes) on the runtime of the muls test program with a 500,000 element input vector. The CPU run time increases roughly linearly with the input size, whereas the GPU run time remains constant.

As predicted, the run time for the GPU implementation remains nearly constant and is not significantly affected by program size, while the CPU implementation run time increases in rough linear proportion to the input size. The GPU run time is constrained by the time it takes to transfer the vertex data to the graphics card and then retrieve it, not by computation time. This suggests that future research should concentrate on finding ways to make the DBlocks sent to the graphics processor as large as possible. Investigating ways to reduce the transfer time would also be worthwhile.

In order to better understand the bottlenecks, we instrumented a 20 opcode version of the muls example with code to measure the time to transfer data to the graphics card as well as the time to retrieve it. The results, gathered from a run with a 500,000 element input vector, are shown in Table 5.

Total run time	160 milliseconds
Time to send data to GPU	70 milliseconds
Time to retrieve output	90 milliseconds

Table 5 Run time bottlenecks for the muls example.

Observe that 44% of the time is spent sending data, and 66% is spent retrieving it. The time for all other types of processing in the program is negligible. There is no separate category titled "time for the GPU to compute the results" because the graphics card operates asynchronously to the main CPU and processes the most recently available data while the CPU is busy transferring the next vertex to the GPU. One unfortunate consequence of these results is that unless more efficient ways are found to "feed the beast," it may be difficult in future work to develop a system that performs computation on the CPU and GPU simultaneously (in parallel). The CPU may always need to be busy feeding the GPU. Had we known that transfer times would have been such a bottleneck at the beginning of the project, we would have explored using more sophisticated techniques, such as compiled vertex arrays, to pass data to the GPU. These could be dramatically more efficient, but their syntax and semantics (particularly the requirement that vertex arrays be a declared as a fixed size) make them more difficult to work with in a context like ours.

5.3 A More Realistic Test: 3-SAT

5.3.1 Motivation

The test programs demonstrated earlier were enlightening, but because they are implemented with single DBlocks, they do not reflect typical uses of our framework to solve real problems. In particular, realistic programs involve a nontrivial amount of branching. To adequately demonstrate our framework, we implemented a solver for the Boolean satisfiability problem with three variables per clause (3-SAT).

We use a genetic algorithm solution technique for 3-SAT because it represents a common idiom used in a variety of problems. Genetic algorithms (GA) are a random local search technique premised on the evolutionary ideas of natural selection and genetic, which has been successfully applied to many NP-complete or NP-hard optimization problems. In this algorithm, a fixed number of potential solutions (a *population* of *chromosomes*) are maintained for a generation, and a new generation is reproduced by applying genetic operators such as *cross-over* and *mutation* and selecting the chromosomes fittest for the solution.[15] Its common application problems are NP-complete or NP-hard and in its nature it requires extensive use of CPU time, high-performance processors are crucial for its efficiency, and several implementation techniques on parallel machines has been studied. This is why we decide to implement and simulate this algorithm for the processor of a programmable graphics card with performance better than most today's general-purpose microprocessor. Though this card provides only 17 instructions and some useful instructions like branch and logical operations are not included in the instruction set, its fine-grained multithreaded vector processor executes more than trillion instructions per second, and enables parallel processing of properly scheduled multiple data at a time.

5.4 The 3-SAT Algorithm

The satisfiability problem is a problem of deciding whether a consistent value can be assigned to each variable in a CNF formula, and the 3-SAT problem is a special case of satisfiability problem where the number of variables in a clause is limited up to 3. This problem is well-known NP-hard problem, and various techniques have been applied to solve this problem. In GA, two schemes are commonly used to represent possible solutions, bit string [11] and clausal representation.[4] In bit string encoding scheme, the value for each variable is represented as a bit (gene), and each chromosome is a string of bits representing the values for all the variables in the formula. Typical cross-over and mutation operators

can be applied naturally, and its fitness can be expressed as the number of clauses that are evaluated as true. In clausal representation, a chromosome is a string of numbers (genes,) which indicates a possible assignment for a clause. For example, In 3-SAT problem, 8 possible assignments for a clause is encoded as an integer number between 0 and 7, and a chromosome is just a string of these numbers. The fitness of a chromosome can be expressed as the number of variables with a consistent value throughout the formula. The following is the algorithm we use:

```
Procedure GA_3SAT(Formula F, int M, int T, float r)
Parameter : F - formula to be solved,
N - population size,
     T - # of generation for search.
     R -  mutation rate.
Local : N - # of variables in F,
     L - # of genes in F,
     K - integer indicating the current generation
     P[i, j] - the value of j-th gene in i-th chromosome.
Randomly generate the population of size M.
while a solution is not found and k < T do
    for I < M do
       Generate two random number a and b between 1 and L.
       Swap P(I, a)~P(I,b) with P(i+1,a)~P(i+1,b).
       Randomly flip a gene according to the mutation rate R.
              Add the new chromosomes to the population of
          children.
       I = i+2
    End for
    Select M chromosomes of the next generation from
    the population of the parents and children,
    M/2 chromosomes with best fitness value and M/2 at
    random from the rest.
End while
End procedure
```

Figure 6 Our algorithm for 3-SAT.

To implement this algorithm in this hardware, we face the following three challenges: how to implement the genetic operator, how to calculate fitness values, and how to select the chromosomes for the next generation. The difficulties lie in the limitation that we cannot use logical operation or branch instructions to should implement these. To tackle this problem, we implement logical operator using currently available instructions, and static scheduling to avoid branching. Some tasks, like sorting, should be done in the main processor so how works can be distributed between the main processor and the graphics card, and how the high-performance of the card can be fully utilized is another problem. By maintaining some number of subpopulations and feeding them to the processor, best performance can be achieved.

5.5 Implementation of 3-SAT

Our implementation details are as follows.

Encoding: Although it is more common to represent a gene using a bit, the limitation of operations we can use prevented us from following the numbers and instead we had to devise an alternative way of representing a bit. Our way of doing this is to keep a number 0 or 1 in a component of a register. Each of the 4 register components can contain a 16-bit number, and we use this just for one bit, so we are wasting some resources. However, the ease of encoding and implementation of problem and operations made us think that is a reasonable choice at least. Thus each gene is represented by a number 0 or 1, each chromosome by a string of a genes. Since your registers can hold up to 4 element at a time, 4 genes are fed together to the processor when an operation is performed on it.

Basic logical operators: Before we discuss our implementation of GA operations, we should say something about logical operations. Since GA usually manipulates bits, it is essential to use bit-wise logic operator to get a good performance result. But in our instruction set, logical operators are not listed so we had to come up with a way of emulating those operations. Based on our encoding scheme that permits only 0 or 1 in a component of a register, we could devise arithmetic operations that will emulate the logical operations. First, logical AND can be emulated easily by multiplication. Only when both operands are 1, the result of the operation will be 1.

1x1 = 1&1 = 1

1x0 = 1&0 = 0

0x1 = 0&1 = 0

0x0 = 0&0 = 0

Likewise, other logical operations can be emulated. The following table shows logical operations and their equivalent arithmetic operations.

Logical operations	Corresponding arithmetic operations
A AND B	A x B
A OR B	(A + B) > 0.5
NOT A	$(A - 1)^2$
A XOR B	$(A - B)^2$
A NOR B	$(A + B - 1)^2$

Table 6 Logical operations that can be emulated.

Cross-over: Cross-over, one of the major operations in GA, is to swap some bits between a pair of chromosomes. Thus this operation takes two operands and outputs two bit strings with some bits exchanged. However, since in our hardware model, one operation can produce only one result string, not two, and the instruction set doesn't include conditional branch and the hardware does not support some kind random number generator, we had devise another clever trick.

First, we produced one bit string per crossover operation with the same length as the other chromosomes in addition to the original two operands. The additional string has some consecutive 1's in some part. When these three strings of bits are produced to the processor, the bits from the original chromosomes in the same position as the 1's in the third string are swapped by some arithmetic operation. Since the processor can produce one output string, we used this operation twice to get both of the swapped chromosomes. Swap operation can be implemented in our setting as follows:

Swap(A, B) = ((C -> A) && (!c -> b)) || ((C -> B) && (!C -> A))

So in our implementation, first half was done to get one result, and next, the other was performed to get the other half.

Mutation: This operator takes one chromosome and it may flip one or two bits or just pass it to the output, and the probability for abit to be flipped is determined by the mutation rate. In our implementation, the main processor produced random numbers and fed it with the original operand to the GPU. Then GPU flips a bit with the probability of mutation rate. To do this, we did the following operations:

Mutation(A) = (slt(B,R) -> !A),

where R is mutation rate, B is the randomly generated string of bits, and slt() is "set on less than" instruction. -> (imply) operator can be translated to our basic operators like:

A -> B = !A || B

we could implement this operation.

Fitness evaluation: As mentioned in section 5.2, fitness value in SAT problem can be the number of clauses satisfied. To calculate thisThe assignment was done by the main processor, and then whether the assignment is satisfactory is determined by GPU using the following formula:

Fitness_eval(A) = slt(0.5, dp3(A,A)),

where dp3 is three component dot product instruction in the GeForce4.

Other parameters: As a selection mechanism, we used tournament and random selection with the best keeping strategy. For each generation, N (population size) new chromosomes are created using crossover and mutation. First, a parent from older generation and a child from the new generation have a tournament and the one with higher fitness will survive. After the tournament throughout the population, we have N winners , half of which will replaced by randomly chosen chromosomes from the losers. This prevents the entire population from being mired in a local minima. For the performance reason, the chromosome with the highest fitness value are preserved in the next generation. We used 1% mutation rate, population size of 100. 3SAT problems were generated using a random automatic generator.

5.6 Results

The following table illustrates the performance of our implementation of 3-SAT:

	Problem size			
	32/64	32/128	48/96	48/196
cpu noopt	11527	16694	14932	23074
gpu noopt	15252	18236	17405	21371
cpu opt	4326	5699	5318	7651
gpu opt	10625	11937	11887	13249

Table 7 Performance of our 3-SAT solver. Numbers represent run times in milliseconds. The notation "32/128" indicates a problem instance with 32 variables and 128 clauses.

These results are not as impressive as the toy programs earlier, but they are not unreasonable either. In one instance (without compiler optimization) the GPU manages to outperform the CPU, and in all other cases the performance of the two implementations does not differ by more than a factor of 2.4. Our relatively poor performance is primarily due to making frequent round trips of data between the CPU and the GPU. After doing this implementation, we were able to determine a few areas in which the architecture of the card could be improved in order to make it more convenient and efficient to program, minimizing round trips of large chunks of data. Our suggestions are discussed in the next section.

6 Analysis

Though the framework we have developed in this paper is effective for simple 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.

Better interfaces to memory buffers: Rather than having to render vertex data to the screen, then read it back, the graphics programming interface should provide a way to render directly to a memory buffer in main memory. Perhaps the GPU could render to a section of graphics memory, then transfer the results using DMA [Direct Memory Access] to main memory. Likewise, there should be an interface where the programmer passes a pointer to a block of input vertices directly to the GPU, and the GPU takes care of pulling all the vertices into the graphics core. The end result would be reduced latency for moving data to and from the GPU. The DMA scheme would also reduce the load on the main CPU, potentially allowing it to be used for computations in parallel with the GPU.

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 [12]), 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). It will be interesting to see whether or not the NVIDIA engineers eventually develop a more complicated out-of-order core that can handle opcodes with variable latency.

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 as discussed in this paper 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. Currently, 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. This change would also be an extremely easy to make to the architecture, since it just involves not doing something that is currently being done. 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. Regardless of how vertex state preservation is accomplished, we feel that it would dramatically improve the speed of our 3-SAT implementation, since more could be done directly on the GPU and less in the CPU.

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. Trace scheduling, which is inherently designed for processors in which branching may be inconvenient and for which large basic blocks are desirable, could potentially deliver dramatic performance improvements.

7 Discussion and Future Work

In this paper we describe a programming framework we have devised for solving general-purpose problems using graphics hardware. We demonstrate the framework's surprising effectiveness at accelerating highly regular operations on large vectors. We then describe an implementation of 3-SAT that uses our programming framework. For the most part, our SAT solver does not achieve greater speeds using the GPU, though it is not slower by a large margin. We investigate the sources of bottlenecks and propose 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 ever 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 the CPU implementation of our factorial example is 26 times slower than the GPU implementation was, and still is, nearly mind-boggling. Likewise, the fact that we were unable to ever force the graphics hardware to fall prey to computational rather than data transfer rate bottlenecks was sobering.

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 definitely untapped—computational power. Perhaps vector processing is not dead after all. As evidenced by the increasing popularity of distributed computing clusters and the decreasing popularity of supercomputers, there is sufficient demand for low-cost alternative computing technologies using commodity hardware to make our approach a valuable contribution. In a few years it may not be unusual to go into the server room of a biotech research company and find Beowulf clusters of cheap Xboxes, selected specifically for their vector capabilities.

We believe that the next logical step in this research is to define a higher-level vector programming language and construct a compiler, perhaps using sophisticated techniques such as trace scheduling, that outputs code that takes more careful advantage of modern graphics architectures. There are many interesting research issues here. For instance, a regular VLIW compiler only has a single, one-dimensional line of operation "slots" to consider per instruction when deciding how to schedule code. In contrast, optimal scheduling of pixel operations on a GPU may involve making effective use of a two-dimensional matrix of "slots." In addition, because round trips are so costly and support for branching is poor or non-existent on modern graphics architectures, it may be most effective to do unusual things like always speculate down both sides of a branch rather than attempt branch prediction at compile-time.

Acknowledgements

We are grateful to Stephen Spencer for helping us set up the hardware we needed for this research.

References

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

[2] Nick England. "A Graphics System Architecture for Interactive Application-Specific Display Functions." *IEEE Computer Graphics and Applications*, 6(1): 60-70, January 1986.

[3] Henry Fuchs *et. al.* "Pixel-Planes 5: A Heterogeneous Multiprocessor Graphics System Using Processor-Enhanced Memories." *Proceedings of SIGGRAPH 1989*, July 1989.

[4] J. Hao. "A Clausal Genetic Representation and its Evolutionary Procedures for Satisfiability Problems." *Artificial Neural Nets and Genetic Algorithms: Proceedings of the 1995 International Conference*, Ales, France, 1995.

[5] A. Kunimatsu *et. al.* "5.5 GFLOPS Vector Units for Emotion Synthesis." *Conference Record, Hot Chips 11*, August 1999.

[6] A. M. Logar and E. M. Corwin. "Implementation of Massively Parallel Genetic Algorithms on the MasPar MP-1." *Proceedings of the 1992 ACM/SIGAPP Symposium on Applied Computing*, 1015-20, 1992.

[7] Adam Levinthal and Thomas Porter. "Chap – A SIMD Graphics Processor." *Proceedings of SIGGRAPH 1984*, July 1984.

[8] Erik Lindholm, Mark J. Kilgard, and Henry Moreton. "A User-Programmable Vertex Engine." *Proceedings of SIGGRAPH 2001*, August 2001.

[9] Michael D. McCool. "SMASH: A Next-Generation API for Programmable Graphics Accelerators." Technical Report CS-2000-14, Computer Graphics Lab, University of Waterloo.

[10] The Mesa 3D Graphics Library. http://www.mesa3d.org

[11] N. Nemer-Preece and R. Wilkerson. "Parallel Genetic Algorithms to Solve the Satisfiability Problem." *Proceedings of the 1998 ACM Symposium on Applied Computing.*

[12] *NVIDIA OpenGL Extension Specifications*. Mark Kilgard, editor. NVIDIA Corporation, May 2001.

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

[14] Kekoa Proudfoot et. al. "A Real-Time Procedural Shading System for Programmable Graphics Hardware." *Proceedings of SIGGRAPH 2001*, August 2001.

[15] A. Whitley. "A Genetic Algorithm Tutorial." *Technical Report CS-93-103*, Department of Computer Science, Colorado State University, 1993.