Cache and Bandwidth Aware Matrix Multiplication on the GPU

Jesse D. Hall Nathan A. Carr John C. Hart
University of Illinois

Abstract

Recent advances in the speed and programmability of consumer level graphics hardware has sparked a flurry of research that goes beyond the realm of image synthesis and computer graphics. We examine the use of the GPU (graphics processing unit) as a tool for scientific computing, by analyzing techniques for performing large matrix multiplies in GPU hardware. An earlier method for multiplying matrices on the GPU suffered from problems of memory bandwidth. This paper examines more efficient algorithms that make the implementation of large matrix multiplication on upcoming GPU architectures more competitive, using only 25% of the memory bandwidth and instructions of previous GPU algorithms.

1 Introduction

The multiplication of matrices is one of the most central operations applied in scientific computing. Recent history has shown continued research for better tuned algorithms that improve the efficiency of matrix multiplication. The ATLAS system for automatic tuning of matrix multiplication for target CPU's has shown much success [Whaley et al. 2001]. New advances in PC chip design (such as streaming SIMD extensions) has led to research into how to best leverage modern mico-architectures for this task [Aberdeen and Baxter 2000].

Recently, consumer based graphics processors (GPU's) have become increasingly more powerful and are starting to support programmable features. The parallelism in graphics hardware pipelines makes the GPU a strong candidate for performing many computational tasks including matrix multiplication [Larson and McAllister 2001]. We detail a new approach for multiplying matrices on GPU hardware. This approach takes advantage of multiple levels of parallelism found in modern GPU hardware and reduces the bandwidth requirements necessary to make this technique effective.

The architecture of modern GPUs relevant to this paper is described in Section 2. In summary, GPUs receive 3D graphics primitives (typically triangles or quadrilaterals) specified as a set of vertices from the application. These vertices are transformed into screen coordinates, and a *fragment* is generated for each pixel covered by the primitive (generating these fragments is called *rasterization*). Fragments contain information such as colors and texture coordinates which are interpolated across the primitive from values associated with the vertices. These auxiliary attributes are used to shade each fragment, which results in a final color which is written to a pixel in the framebuffer. One of the most common ways of shading fragments is to use auxiliary attributes known as texture coordinates to index into a previously supplied image (texture). Multiple sets of texture coordinates can be used to retrieve colors from multiple textures; the results are combined to form the final color for the fragment. This is multitexturing. Finally, a shaded fragment can either replace the current value of the pixel in the frame buffer or it can be added to the current value. This version of the graphics pipeline is described in [Woo et al. 1997].

The performance of GPUs comes from the fact that large amounts of parallelism are available in this pipeline. In particular, each fragment is independent of all other fragments, so they can be processed in parallel. Processing of fragments can also be overlapped to hide pipeline stalls and memory latencies, resulting in very efficient use of the hardware.

Multitexturing can be used to multiply matrices [Larson and McAllister 2001]. An $m \times n$ matrix can be represented by a greyscale texture, with the each pixel containing an element of the matrix [1]. These matrices can be displayed on the screen by drawing an $m \times n$-pixel rectange with the texture coordinates $(0,0), (0,n-1), (m-1,n-1), (m-1,0)$ assigned to the vertices (clockwise starting with the upper-left vertex). One benefit of this is that we can access the transpose of a matrix by drawing an $n \times m$-pixel rectangle with texture coordinates $(0,0), (m-1,0), (m-1,n-1), (0,n-1)$. The exact same texture is used for drawing the matrix transposed and untransposed. We have only changed the mapping of the texture image onto the rectangle.

Matrix multiplication performs $C = AB$ where A is an $m \times l$-element matrix and B is an $l \times n$-element matrix. By storing matrices A and B as textures, we can compute C in l multitexturing passes as shown in Figure 1.

```
Clear the screen.
Set the drawing mode to overlay.
Load texture texA with matrix A.
Load texture texB with matrix B.
Set the multitexturing mode to modulate.
Set framebuffer write mode to accumulate.
for i = 0 ... l - 1
    draw an m × n-pixel rectangle with
        texA coords (0,i), (0,i), (m-1,i), (m-1,i), and
        texB coords (i,0), (i,n-1), (i,n-1), (i,0).
end
Screen contains result of A × B.
```

Figure 1: The Larson-McAllister multipass algorithm for multiplying two matrices.

The texture coordinates $(0,i), (0,i), (m-1,i), (m-1,i)$ replicate the ith column across the entire rectangle, whereas the texture coordinates $(i,0), (i,n-1), (i,n-1), (i,0)$ replicate the ith row. These textured rectangles are then combined as demonstrated in Figure 2.

[1] Textures are typically indexed using (s,t), with s indexing the horizontal axis and t indexing the vertical axis. This is the opposite of standard matrix notation, where the first index represents the row and the second index representing the column. In the rest of this paper, we'll use the matrix style rather than the texture style. Also, texture coordinates have traditionally been in the range $[0 \ldots 1]$, with various filters used to generate a color for indices that fall between pixels. We use an extension [Kilgard 2001] to allow integer indexing, and disable filtering.

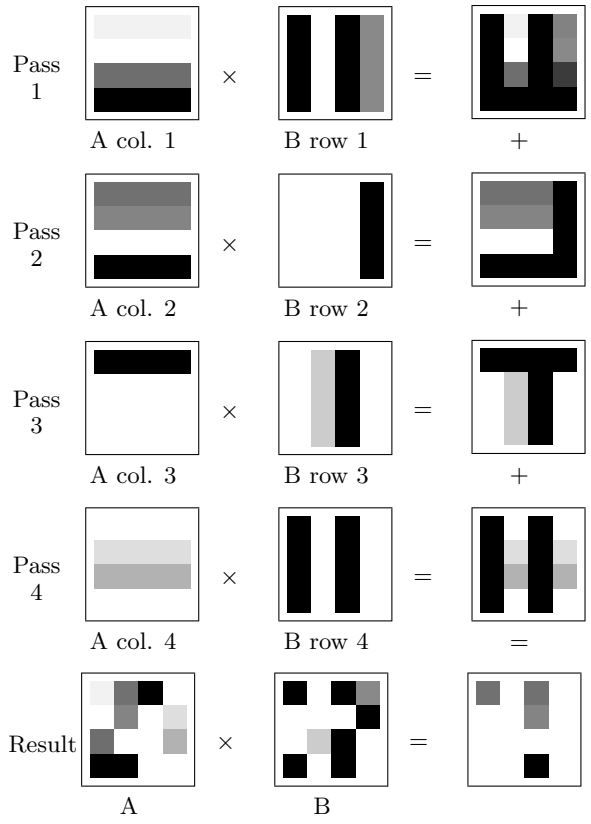

Figure 2: Demonstration of Larson-McAllister matrix multiplication on a pair of 4×4 matrices. (The output in this example is saturated, such that results greater than one appear uniformly white.)

2 Modern GPU Organization

The graphics pipeline implemented on graphics acceleration hardware was classically organized in a series of transformations, clipping and rasterization steps. Modern graphics hardware has generalized this pipeline into programmable elements. The modern graphics pipeline consists of vertex processing, rasterization and fragment processing. The vertex processor performs operations on the individual vertices of triangles sent to the graphics accelerator. Once transformed, these triangles are rasterized into a collection of pixels. Each pixel output by the rasterizer is called a *fragment*. The rasterization process linearly interpolates attributes, such as texture coordinates, stored at the vertices and stores the interpolated values at each fragment. A fragment processor uses the interpolated texture coordinates to lookup texture values from texture memory, and can perform special-purpose arithmetic operations on both the texture addresses and the fetched texture values.

The vertex processor is structured similarly to vector processors (pipeline on a stream of vertices), whereas the fragment processor is structured similarly to a SIMD array processor (one processor per pixel). Our experiments have shown that the vertex processor does not provide much advantage over existing CPU capabilities, whereas the fragment processor already outperforms the CPU on some operations like ray-triangle intersections [Carr et al. 2002].

Because these processors were originally developed for texturing, the programs the GPU executes are called *shaders*. A *fragment shader* is a program executed by the fragment processor that describes the color each fragment before it is possibly assigned to its corresponding pixel. The inputs to a fragment shader are a set of program constants, interpolated attribute data from triangle vertices, and texture maps (blocks of texture memory addressed by the texture coordinates). Before rendering, the fragment shader is compiled and loaded into the graphics hardware. Primitives (in our case quadrilaterals) are then sent down the graphics pipeline invoking the enabled fragment shader as they are rasterized. The output of a fragment shader is an output color plotted to the screen.

A fragment shader is alloted a fixed set of temporary registers $R0 \ldots Rn$. Each register holds a single 4-vector corresponding to the four color channels red, green, blue, alpha. The color channels of each register may be accessed individually as follows: $Ri.c$ where $c \in \{r, g, b, a\}, i \in \{1..n\}$. Standard arithmetic operations are defined over the set of registers, such as addition and multiplication. For example: $R2.b = R1.a + R0.g$, assigns the blue channel of register R2 to be the sum of the alpha channels of registers $R1$ with the green channel of $R0$.

Modern fragment shaders allow for up to four-instruction to be executed simultaneously, much like that of the SIMD instructions found in modern PC architectures. For example:

$$R2 = R1.abgr * R0.ggab \tag{1}$$

can be issued as a single GPU instruction which refers to four simultaneous multiplications numerically equivalent to:

$$\begin{aligned} R2.r &= R1.a * R0.g \\ R2.g &= R1.b * R0.g \\ R2.b &= R1.g * R0.a \\ R2.a &= R1.r * R0.b \end{aligned} \tag{2}$$

The SIMD nature of the operation defined in (1) allows for four additions to occur in parallel, taking one fourth the computation time of (2).

In equation (1), $R1$'s color channels are referenced in arbitrary order. This is referred to as *swizzling*. The green channel of $R0$ is referenced multiple times. This is known as *smearing*. Arbitrary swizzling and smearing (and also negation) of input operands can be done with no performance penalty. This is in contrast to the Intel's SSE instructions, where moving data between channels requires additional instructions.

The output color of a fragment program is placed in a designated register (usually $R0$) upon termination of the program. This value is written to the fragment's screen location in the frame buffer.

Fragment shaders have access to three kinds of data: constants, interpolated vertex attribute data (e.g. texture coordinates), and texture data (indexed by texture coordinates). Interpolated vertex attributes data are accessed as the registers $T0 \ldots Tm$ where $m + 1$ is the number of attributes stored with each vertex.

Data is fetched from texture memory by the lookup() operation. For example, $R0 = \text{lookup}(T0.r, T0.g, M)$ uses the first two coordinates of $T0$ to access the texture M. We can also perform arithmetic on the texture coordinate before the fetch, or we can use the result of one texture fetch as the coordinates for a second texture fetch (dependent texturing).

Although fragment shaders provide a very powerful SIMD model for programming, they are currently limited in a number of ways. Modern implementations restrict the number of available registers, the total instruction length, and the

number of lookup() operations that may occur in a given fragment shader. Control flow is also restricted in fragment shading. For example, branching is not supported and conditional execution is limited to predicating instructions on previously set condition codes.

Our model for a fragment shaders is based on the one described by the upcoming DirectX 9.0 specification [Marshall 2001]. This model provides capabilities currently found in vertex shaders at the fragment shader level. This model has also been used to describe the implementation of a ray tracer as a fragment shader [Purcell et al. 2002]. This paper assumes similar fragment processor capabilities, specifically fragment shaders of up to 256 instructions, an unrestricted number of texture access operations, a set of at least six registers, and standard single-precision floating point data formats.

3 Cache Aware Matrix Multiply

Suppose we are muliplying two large matrices X and Y, *wlog* who's dimensions are a perfect power of two, with $n = 2^i$ rows and $n = 2^i$ columns. A general algorithm for computing $Z = XY$ can be expressed as follows:

$$\begin{array}{l}\text{for } i = 1 \ldots n \\ \quad \text{for } j = 1 \ldots n \\ \quad\quad Z_{ij} = 0 \\ \quad\quad \text{for } m = 1 \ldots n \\ \quad\quad\quad Z_{ij} = Z_{ij} + X_{im} * Y_{mj} \\ \quad\quad \text{end} \\ \quad \text{end} \\ \text{end}\end{array} \tag{3}$$

The outer two loops are implemented on the GPU by rendering a single screen filling quadrilateral. This implies that everthing within the outer two loops must be handled by the fragment shader. Below we have inserted pixel shader pseudo-code in the appropriate places. Matrices X and Y are now assumed to be stored in single channel texture maps and accessed through the fragment shader lookup() operation.

$$\begin{array}{l}\text{for } i = 1 \ldots n \\ \quad \text{for } j = 1 \ldots n \\ \quad\quad \text{fragment shader} \left\{ \begin{array}{l} R3.r = 0 \\ \text{for } m = 1 \ldots n \\ \quad R1.r = \text{lookup}(i, m, X) \\ \quad R2.r = \text{lookup}(m, j, Y) \\ \quad R0.r = R0.r + R1.r * R2.r \\ \text{end} \end{array} \right. \\ \quad \text{end} \\ \text{end}\end{array} \tag{4}$$

The above psuedo-code in (4) requires that either loops are available in fragment shaders or that the fragment shader instruction count is long enough to allow the innermost loop to be unrolled. We assume neither are realistic assumptions. To address this issue, we turn to a standard blocking strategy.

Blocking has been shown to improve cache performance, but for this application blocking also serves the purpose of allowing us to work within the constraints of our fragment programming model. The psuedo-code for our blocking strategy is shown in (5). A new matrix F is introduced that is initialized to be all zeroes and used as a temporary store by the routine. The value b is a scalar representing the block size.

$$\begin{array}{l}\text{for } k = 1..n \text{ step } b \\ \quad \text{for } i = 1 \ldots n \\ \quad\quad \text{for } j = 1 \ldots n \\ \quad\quad\quad \text{fragment shader} \left\{ \begin{array}{l} R3.r = 0 \\ \text{for } m = k \ldots k + b - 1 \\ \quad R1.r = \text{lookup}(i, m, X) \\ \quad R2.r = \text{lookup}(m, j, Y) \\ \quad R3.r = R3.r + R1.r * R2.r \\ \text{end} \\ R4.r = \text{lookup}(i, j, F) \\ R0.r = R3.r + R4.r \end{array} \right. \\ \quad\quad \text{end} \\ \quad \text{end} \\ \quad \text{Copy frame buffer into texture } F \\ \text{end}\end{array} \tag{5}$$

This new algorithm is a multipass method requiring multiple renderings to the frame buffer. The outer three loops are handled by rendering screen filling quadrilateral to the frame buffer n/b times. Between each of the n/b passes, the frame buffer is copied into texture map F to be accumulated with result of the next pass. This copy operation is required since modern GPU hardware does not support direct lookup operations on the framebuffer. Some graphics hardware does however, support blending modes allowing fragment values to accumulate directly with the contents of the frame buffer eliminating the need for the temporary texture F, and consequently more efficient rendering.

The fragment program (which covers the portion inside the j loop), can now be unrolled by choosing an appropriate value of b. For our tests we have chosen $b = 32$. We were able to reduce our total fragment program instruction count to be four instructions per iteration of the m loop, for a total of $6 + 4b = 134$ total instructions.

4 Multi-Channel GPU Matrix Multiplies

Texture map sizes on modern day GPU's are often restricted. Let n being the maximum size of any dimension. NVidia's GeForce4 Ti4600 has a maximum allowable renderable size of $n = 2048$, limiting multipass programs to two-dimensional textures containing at most 2048×2048 elements. Texture maps may consist of between one and four channels (luminance, luminance-alpha, RGB or RGBA). This implies that the GeForce4 can handle textures in size up to $2048 \times 2048 \times 4$.

Methods have already been presented for handling matrices whose dimensions are at most $n = 2048$ using single channel texture maps. It is naturally desirable to be able to handle matrices of larger sizes. The GeForce4 for example should be able to multiply matrices 4096×4096 in size by utilizing all four of the color channels. This section describes a matrix multiplication algorithm that takes advantage of this four-component storage capability.

These four-channel textures store matrices of 4096×4096 32-bit floating point values, and occupy 64MB. Our GPU matrix multiplication implementation requires four times this space for storing the two operands, a temporary store, and the result. Current consumer level GPU's such as the GeForce4 currently ship with 128MB of on-board memory, suggesting a maximum capable matrix side of 2828×2828. Exceeding this memory threshold under a non-unified memory architecture of present-day PC GPU's results in paging to main system memory, and increased traffic over the graphics card bus.

4.1 Basic Formulation

Suppose we are muliplying two large matrices X and Y, wlog whose dimensions are a perfect power of two, with 2^i rows and 2^i columns.

$$XY = Z \tag{6}$$

The matrix multiply in (6) can be expressed as the following series of matrix multiplies of smaller matrices

$$\overbrace{\begin{bmatrix} A & B \\ C & D \end{bmatrix}}^{X}\overbrace{\begin{bmatrix} E & F \\ G & H \end{bmatrix}}^{Y} = \overbrace{\begin{bmatrix} AE+BG & AF+BH \\ CE+DG & CF+DH \end{bmatrix}}^{Z} \tag{7}$$

Elements A, B, C, D, E, F, G, and H are sub-matrices decomposing X and Y. Let the dimensions of $A \ldots H$ be 2^{i-1} rows by 2^{i-1} columns.

4.2 Blocked Matrix Texture Maps

We can store matrices X and Y as texture maps in a 2^{i-1} by 2^{i-1} sized texture maps on the GPU, by placing the four sub-matrices in the different color channels RGBA as

$$X = \begin{pmatrix} A_r & B_g \\ C_b & D_a \end{pmatrix}, Y = \begin{pmatrix} E_r & F_g \\ G_b & H_a \end{pmatrix}. \tag{8}$$

We now introduce subscript notation on matrices M, such that M_i for $i \in r, g, b, a$ refers to the submatrices composing M. For example $X_r Y_g = AF$.

We have presented our technique for multiplying matrices contained in a single color channel $X_r Y_r = Z_r$ in Section 3. We can extend this same approach to a SIMD notation by using multiple subscripts r, g, b, a. For example:

$$X_{rrbb} Y_{rgrg} = \begin{pmatrix} AE_r & AF_g \\ CE_b & CF_a \end{pmatrix}. \tag{9}$$

The above notation assumes an architecture where four sub-matrices may be operated on in parallel by a single instruction. This notation is useful since graphics hardware is designed in a SIMD manner to work simulateously on four color channels at a time. Using this notation, we can now concisely express matrix multiplication X and Y as follows:

$$\begin{aligned} X_{rgba} Y_{rgba} &= X_{rrbb} Y_{rgrg} + X_{ggaa} Y_{baba} \\ &= \begin{pmatrix} AE_r & AF_g \\ CE_b & CF_a \end{pmatrix} + \begin{pmatrix} BG_r & BH_g \\ DG_b & DH_a \end{pmatrix} \\ &= Z_{rgba} \end{aligned} \tag{10}$$

4.3 The Multi-Channel Algorithm

To apply the formulation derived in (10) into an algorithm usable by graphics hardware, we must first re-examine fragment shader programming. As discussed in Section 1, a single lookup() operation can retrieve a 4-vector corresponding the four color channels. If we store X as a texture map in blocked form (8) then a single lookup $R0 = \text{lookup}(i, j, X)$ can be used to retrieve four values coming from the submatrices of $R0 = A_{ij}, B_{ij}, C_{ij}, D_{ij}$. The four matrix multiplies from equation (10) may now be parallelized within a single fragment shader. The matrix swizzling and smearing suggested by (10) is handled at the per-element level utilizing the capabilities of graphics hardware, as shown in (11).

$$\begin{array}{l} \text{for } k = 1 \ldots n/2 \text{step } b \\ \quad \text{for } i = 1 \ldots n/2 \\ \quad\quad \text{for } j = 1 \ldots n/2 \\ \quad\quad\quad \text{fragment shader} \left\{ \begin{array}{l} R3 = 0 \\ \text{for } m = k \ldots k+b-1 \\ \quad R1 = \text{lookup}(i, m, X) \\ \quad R2 = \text{lookup}(m, j, Y) \\ \quad R3 = R1.rrbb * R2.rgrg + R3 \\ \quad R3 = R1.ggaa * R2.baba + R3 \\ \text{end} \\ R4 = \text{lookup}(i, j, F) \\ R0 = R3 + R4 \end{array} \right. \\ \quad\quad \text{end} \\ \quad \text{end} \\ \quad \text{copy frame buffer into texture } F \\ \text{end} \end{array} \tag{11}$$

The above algorithm represents an efficient use of the SIMD computation power of the GPU by working on all four color channels in parallel. This implementation only increases the fragment shader instruction count by one per iteration of m, thus resulting in a total fragment program length of $6 + 5b = 166$ with $b = 32$.

5 Analysis

We have analyzed the new blocked and multichannel GPU matrix multiplication algorithms with respect to memory bandwidth, instruction count and predicted performance.

5.1 Bandwidth Considerations

To analyze the potential bandwidth limitations for our approach, we first distinguish between the two bandwidth limited areas of modern GPUs. The *external bandwidth* is the rate at which data may be transferred between the GPU and the main system memory. On modern PC's this is limited by the speed of the AGP graphics bus which can transer data at the rate of 1GB/sec. The *internal bandwith* is the rate at which the GPU may read and write from its own internal memory. The GeForce4 Ti 4600 is currently capable of transferring 10.4 GB/sec.

For our application the external bandwidth of the GPU affects our application in two areas. First, the matrices must be copied into the GPU's memory as texture maps, and the result of the computation must be read back from the card into main memory. These transfers use the AGP bus, which currently has a theoretical bandwidth of about 1 GB/s (for AGP 4x). However, in practice sending data to the GPU is much faster than reading data back from the GPU (since the hardware and drivers are optimized for this case), and the best speed we've measured for reading data back into host memory is 175 MB/s. Even assuming an average of 200 MB/s transfer for both reads and writes, transfering two 1024×1024 single-precision matrices to the GPU and reading the 1024×1024 result back requires 60 ms. At 4 GFLOPS the actual computation takes about 510 ms. For this problem, transfer time is about 11% of the total time. Thus, external bandwidth is a significant but not overwhelming fraction of the total time.

The second part of the algorithm affected by external bandwidth is the time to send the geometry, the screen filling quads on which the matrices are texture mapped. In fact this is a very small amount of data (about 48 bytes per pass), and graphics hardware is very good at transferring

geometry information in parallel with other tasks, including running fragment shaders. Thus, this cost is negligible.

One of the primary bottlenecks in performing matrix multiplies on the GPU is the internal bandwidth [Larson and McAllister 2001]. This is also true for CPU implementations. For our analysis we consider mutiplying two $n \times n$ matrices. For both the multi-channel and single-channel block-matrix approaches, the processing of each fragment requires two texture lookup() operations per iteration of the inner m loop, plus an additional lookup() to combine it with the results from the previous pass. A single write to output occurs per fragment per pass as its result is written to the frame buffer. Thus, there are $2b + 2$ memory operations per fragment per pass. In our single channel method, every memory operation transfers 4 bytes of data. Our multi-channel method transfers 16 bytes of data (4 channels, 4 bytes per channel) per memory operation.

method	*passes*	*frags/pass*	*bytes/frag*	*total*
L-M	n	n^2	16	$16n^3$
single	$\frac{n}{b}$	n^2	$(2b+2)4$	$\frac{8n^3(b+1)}{b}$
multi	$\frac{n}{2b}$	$(\frac{n}{2})^2$	$(2b+2)16$	$\frac{4n^3(b+1)}{b}$

Table 1: Bytes transferred internally by each GPU matrix multiplication method.

Table 1 summarizes these results and shows the total internal bandwidth in bytes transferred by each method, which is just the product of the number of passes, the fragments per pass and the bytes per fragment. The L-M (Larson-McAllister) figures assume an implementation based on a single floating-point channel. (Even though multiple channels were mentioned, [Larson and McAllister 2001] did not describe a multi-channel implementation.) The four-byte floats are accessed four times (two matrices, a temporary store and a result) for a total of 16 bytes transferred per fragment.

Our single-channel block matrix algorithm performs identically to L-M when b is set to one (no blocking). As the blocking size b grows, the bandwidth drops by nearly a factor of two when compared to L-M. The multi-channel method further reduces the memory bandwidth by exactly one half over the single channel method, reducing the bandwidth to nearly 25% of L-M.

5.2 Instructions

The GPU uses the fact that the same instructions are being executed for a large number of fragments to overlap the processing of different fragments. As no communication between executions of the fragment shaders is needed, a large amount of parallelism is available. This parallelism is used to hide the latency of memory operations and other causes of stalls. As a result, when enough fragments are available (as in our case), the running time of a fragment shader is approximately linear in the number of instructions executed (assuming computation is the limiting factor). Therefore, it makes sense to analyze the number of instructions used by our algorithm.

Each execution of the fragment shader needs four instructions for setup and adding in the result of the previous pass. The multi-channel algorithm also needs three instructions per iteration of the inner loop (two multiply-add instructions and one addition to update the texture indices). The single-channel algorithm removes one of the multiply-add instructions. Therefore, the instruction counts are $3b + 4$ for the multi-channel case and $2b + 4$ for the single-channel case. Note that in the multi-channel case, each many of the instructions involve a 4-wide data issue, operating on the four color channels in parallel. Currently there is no performance penalty for this since modern GPU's designed to natively work on multiple channels. Table 2 summarizes the analysis and the total GPU floating point instructions required by each method.

method	*passes*	*frags/pass*	*inst/frag*	*total inst*
L-M	n	n^2	2	$2n^3$
single	$\frac{n}{b}$	n^2	$2b+4$	$\frac{n^3(2b+4)}{b}$
multi	$\frac{n}{2b}$	$(\frac{n}{2})^2$	$3b+4$	$\frac{n^3(3b+4)}{8b}$

Table 2: Floating point operations required by each GPU matrix multiplication method.

The additional fragment program overhead makes the single channel block-structured matrix multiplication longer than the L-M algorithm. As b increases, the instruction count asymptotically approaches that of the L-M algorithm. The multi-channel method executes 3/16ths as many instructions of either of the single channel methods as the block size grows.

5.3 Performance

ATLAS has demonstrated 4.0 GFLOPS/s for matrix multiplication on a 1.5GHz Pentium4 using Intel's SSE2 SIMD instructions [Dongarra 2001]. Is the GPU comparable to this? For a matrix size of 1024×1024 and a block size of 32 ($n = 1024, b = 32$), our multi-channel algorithm transfers 4.125 GB of data. In order to match the ATLAS P4 SSE numbers, we need to perform this multiplication in 0.5 seconds. This means we will need 8.25 GB/s of bandwidth. Current hardware has a theoretical bandwidth of 10.4 GB/s to the main memory; as with CPUs, it also has a cache between the GPU and memory which supports much higher bandwidth. Thus, existing hardware should be able to support our bandwidth needs. Future hardware is likely to improve both cache size and performance and memory bandwidth.

Performance of CPU implementations of matrix multiplication are typically limited by memory bandwidth, not CPU speed. Larson and McAllister [Larson and McAllister 2001] reported similar results with their GPU implementation. However, due to much lower clock speeds on CPUs relative to CPUs, the move from fixed-point byte operations to floating point may increase the processing requirements enough to make the GPU speed the bottleneck.

6 Conclusion

We have presented a multichannel block-based GPU matrix multiplication algorithm. The block structured approach should yield greater cache coherence than previous methods. We also demonstrated that our implementation uses only about 25% of the memory bandwidth and instructions when compared to the previous method.

Our results are currently theoretical as we anticipate the implementation of graphics hardware that supports the DirectX 9.0 standard. We expect such hardware will be available before the final version of this paper is required. (For example, as of this writing, 3Dlabs has just announced a processor, the P10, that partially satisfies upcoming standards.)

We will then be able to provide actual implementation times comparing our cache and bandwidth aware algorithms to the previous work [Larson and McAllister 2001].

We have made numerous assumptions about the performance of the upcoming hardware. These assumptions are based on the speed of existing hardware, but with the generality to handle the upcoming standards. These simulations and analyses suggest our method to be competitive with modern CPU implementations. The existence of hardware implemenations will allow us to perform further tuning and validate our claims with empirical data.

Available hardware would allow us to automatically tune our algorithm for a given GPU in much the same manner as performed by Altas. Emperical tests may be run to provide searches over algorithm's parameter space to select the best aglorithm for a target GPU. Our algorithm is currently parameterized by its block size b, but we could also introduce additional parameters to control order of rasterization and blocking of memory layouts.

The increased memory bandwidth and SIMD organization of the GPU should make it a good choice for scientific applications. We have nonetheless found that the GPU remains about as powerful as the CPU on actual tasks like matrix multiplication and ray tracing [Carr et al. 2002]. This has been disappointing given the increased bandwidth and processing power of the GPU. The constraints of GPU programming coupled with the low bandwidth connection from the GPU back into the CPU have been major obstacles in the capitalization of the GPU for scientific applications.

We are nonetheless encouraged by the potential of the GPU. Whereas future enhancements to the CPU explore parallelism through speculative execution and other probabilistic methods, the GPU can exploit parallelism across the frame buffer and across the geometric data. This has been partially responsible for the dominance of the GPU performance growth rate over that of the CPU. As GPU growth continues to outpace CPU growth, we expect the GPU will become the preferred platform for personal high-performance scientific computing.

Acknowledgments

This research was supported in part by the NSF under the ITR grant ACI-0113968, and by NVidia Corp. Conversations with Jack Dongarra and Jim Demmel (and his students) were also quite helpful.

References

Aberdeen, D., and Baxter, J. 2000. General matrix-matrix multiplication using SIMD features of the PIII (research note). In *European Conference on Parallel Processing*, 980–983.

Carr, N. A., Hall, J. D., and Hart, J. C. 2002. The ray engine. Tech. Rep. UIUCDCS-R-2002-2269, University of Illinois at Urbana-Champaign, Mar.

Dongarra, J. 2001. An update of a couple of tools: ATLAS and PAPI. DOE Salishan Meeting (Available from http://www.netlib.org/utk/people/JackDongarra/SLIDES/salishan.ps), Apr.

Kilgard, M. J. 2001. GL_NV_texture_rectangle. http://www.nvidia.com/dev_content/nvopenglspecs/GL_NV_texture_rectangle.txt.

Larson, S. E., and McAllister, D. 2001. Fast matrix multiplies using graphics hardware. *Super Computing* (Nov.).

Marshall, B. 2001. DirectX graphics future. *Microsoft DirectX Meltdown 2001* (Jul.).

Purcell, T. J., Buck, I., Mark, W. R., and Hanrahan, P. 2002. Ray tracing on programmable graphics hardware. In *Proceedings of SIGGRAPH 2002*, ACM Press / ACM SIGGRAPH, J. F. Hughes, Ed., Computer Graphics Proceedings, Annual Conference Series, ACM.

Whaley, R. C., Petitet, A., and Dongarra, J. 2001. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing 27*, 1-2, 3–35.

Woo, M., Neider, J., Davis, T., and Shreiner, D. 1997. *OpenGL Programming Guide*. Addison-Wesley, Reading, MA, USA.