Graphics Hardware (2004)
T. Akenine-Möller, M. McCool (Editors)

# Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication

K. Fatahalian, J. Sugerman, and P. Hanrahan †

Stanford University

**Abstract**

*Utilizing graphics hardware for general purpose numerical computations has become a topic of considerable interest. The implementation of streaming algorithms, typified by highly parallel computations with little reuse of input data, has been widely explored on GPUs. We relax the streaming model's constraint on input reuse and perform an in-depth analysis of dense matrix-matrix multiplication, which reuses each element of input matrices $O(n)$ times. Its regular data access pattern and highly parallel computational requirements suggest matrix-matrix multiplication as an obvious candidate for efficient evaluation on GPUs but, surprisingly we find even near-optimal GPU implementations are pronouncedly less efficient than current cache-aware CPU approaches. We find the key cause of this inefficiency is that the GPU can fetch less data and yet execute more arithmetic operations per clock than the CPU when both are operating out of their closest caches. The lack of high bandwidth access to cached data will impair the performance of GPU implementations of any computation featuring significant input reuse.*

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Graphics processors

## 1. Introduction

The emergence of programmable graphics hardware has led to increasing interest in offloading numerically intensive computations to GPUs. The combination of high-bandwidth memories and hardware that performs floating-point arithmetic at significantly higher rates than conventional CPUs makes graphics processors attractive targets for highly parallel numerical workloads. Many applications beyond traditional graphics specific ones have been demonstrated to run on GPUs [HCSL02, BFGS03, KW03, MA03, BFH*04]. Particularly, algorithms that can be structured as *streaming* computations often realize notable performance gains. Streaming computations walk over a set of independent inputs and independently evaluate a kernel on each value. The combination of regular, predictable data access with the independence of each kernel invocation maps very nicely to graphics architectures.

Streaming computations can be characterized as being highly parallel and numerically intensive with little reuse of input data. However, many computations are both highly parallel and numerically intensive, but exhibit significant per element reuse. We study the efficiency of current graphics architectures in computations where each input contributes to many outputs. Optimized approaches to these problems on traditional CPUs are able to exploit cache hierarchies to effectively amplify memory bandwidth and keep processor functional units busy. We question whether calculations that benefit from memory "blocking" strategies will perform efficiently on GPUs and if analogous blocking strategies are necessary, or even possible, given the programming model and capabilities of current graphics architectures.

Specifically, we investigate dense matrix-matrix multiplication. It offers regular memory access and abundant parallel computation but features $O(n)$ data reuse and seems a natural candidate for a fast GPU implementation. Moreover, dense matrix-matrix multiplication is a building block of numerical libraries such as LAPACK [ABB*99]. These libraries are widely used in the development of scientific ap-

† {kayvonf, yoel, hanrahan}@graphics.stanford.edu

plications and must run efficiently if GPUs are to become a useful platform for numerical computing.

Cache-aware implementations for CPU architectures have been well studied [TCL98, WPD01] and several GPU algorithms have been developed. Larsen and McAllister [LM01] initially proposed an approach for computing matrix products on graphics hardware. They observed performance equaling that of CPUs but their computations were limited to 8-bit fixed-point data. Both Hall et al. [HCH03] and Moravánszky [Mor03] describe improved algorithms for modern hardware. Moravánszky reports his implementation is outperformed by optimized CPU code. Despite these results, little work has been done to analyze the performance limiting factors of these GPU implementations. We perform such an analysis and determine that the algorithms are bandwidth limited due to the inability of GPUs to provide high bandwidth access to frequently accessed data. Surprisingly, we find that existing algorithms are near-optimal for current hardware. That is, better performance will require architectural changes.

## 2. Implementations of Matrix-Matrix Multiplication

We consider the problem of computing the product, $C = AB$, of two large, dense, N×N matrices. We quickly describe naive and optimized CPU algorithms and then delve more deeply into solutions for a GPU.

### 2.1. Matrix-Matrix Multiplication on CPUs

The following CPU algorithm for multiplying matrices exactly mimics computing the product by hand:

```
for (i=0;i<N;i++)
   for (j=0;j<N;j++)
      C[i,j] = 0;
      for (k=0;k<N;k++)
         C[i,j] += A[i,k]*B[k,j];
```

Unfortunately, while simple, this algorithm suffers from poor locality. Elements of $B$ are accessed columnwise, and therefore not in sequential order in memory (if matrices are stored in row-major order). While each iteration in $j$ reuses row $i$ of $A$, that row may have been evicted from the cache by the time the inner-most loop completes. As a result, the algorithm is bandwidth limited and displays poor performance and low efficiency (i.e. time spent doing computation versus loading data).

When sustaining cached accesses, the Pentium 4 can fetch a 128 bit SSE value (4 packed 32 bit floats) in a single cycle [HSU*01]. The basic flaw of the naive triple-for-loop implementation is that the size of its per-loop working set prevents bandwidth amplification from the closest (and fastest) levels of the memory hierarchy. A standard solution "blocks" the inputs into small submatrices that fit in the processor caches, and decomposes the computation of $C$ into subcomputations on the submatrices. Additionally, $B$ is often stored transposed to offer a more cache-friendly representation when walking its columns. The ATLAS [WPD01] software package, benchmarked in section 3, works even harder to achieve cache efficiency. ATLAS tests the host CPU to determine the sizes of system caches, then self-tunes its algorithms to use code snippets that are optimal for the detected cache sizes. Effective cache-aware implementations of matrix multiplication on CPUs achieve much higher effective bandwidth and hence numerical efficiency.

### 2.2. Matrix-Matrix Multiplication on the GPU

A simple approach to compute the product of two matrices on a GPU, although feasible only on architectures that support sufficiently long shaders, is to compute elements of the resulting matrix in a single rendering pass. We refer to this approach as **NV Single**. We store 2×2 blocks of matrix elements in 4 component texels as described by [HCH03]. This packing allows each input fetched from memory to be used twice in a fragment program. Thus the access of data used in the second operation is performed at register speed. Our shader, given below, reads 2 rows from matrix $A$ and 2 columns of $B$ to compute 4 elements of the output matrix. The values of variables $i$ and $j$ are provided to the shader via interpolated texture coordinates.

```
for (k=0;k<N/2;k++)
   C[i,j].xyzw += A[i,k].xxzz*B[k,j].xyxy +
                 A[i,k].yyww*B[k,j].zwzw;
```

Each shader program performs 4 inner loop iterations of the CPU algorithm given in section 2.1. GPU texture caches are organized to capture 2D locality, so sequential accesses along either axis are cache coherent. Additionally, shaders executing in parallel on adjacent fragments will fetch the same data from a row or column of the inputs simultaneously. In practice, for sufficiently large matrices, instruction counts exceed the limits of all architectures and a small number of passes are required to complete the computation.

In contrast, multipass algorithms are potentially more cache friendly with respect to inputs at the cost of performing extra framebuffer reads and writes. We implement a variation of the multipass algorithm presented by Larsen and McAllister [LM01]. Our implementation packs 4 consecutive elements from a matrix column into a texel, and thus performs a small number of 4×4 matrix by 4×1 vector products in a shader as described in detail by [Mor03]. This method makes 4x reuse of data in the texel obtained from matrix B, but only uses data fetched from A once. 6 fetches are needed to retrieve the operands needed for 4 `mad` operations, so in total this approach requires 1.5x more texture fetches than **NV Single**. However, since only a few rows and columns of the input textures are accessed in a given rendering pass, cache hit frequency is expected to be higher. A shader performing a single matrix-vector product is given given below.

It accepts $i$, $j$, and $k$ arguments via interpolated texture coordinates. $accum[i, j]$ holds the partial result of $C[i, j]$ from the previous rendering pass. In each pass, $k$ is incremented by 4.

```
C[i,j].xyzw = accum[i,j].xyzw +
              A[i,k].xyzw   * B[k/4,j].xxxx +
              A[i,k+1].xyzw * B[k/4,j].yyyy +
              A[i,k+2].xyzw * B[k/4,j].zzzz +
              A[i,k+3].xyzw * B[k/4,j].wwww;
```

Our NVIDIA implementation, **NV Multi**, computes a single matrix-vector product per shader. We unroll this kernel 3 to 6 times (the maximum possible amount allowed by the driver) to construct our shader for the ATI hardware, **ATI Multi**. We tested various amounts of unrolling on each platform. The chosen values maximized algorithm performance.

Performance tuning of algorithms on graphics hardware is difficult because vendors do not disclose specific architectural details, such as cache parameters or the physical layout of texture data in memory. We tested a variety of techniques intended to increase cache efficiency and leverage the parallel fragment processing architecture of the GPU. We varied the packing of matrix elements into textures, the unrolling of loops in shaders to reduce rendering passes, and utilized multiple render targets to perform multiplication of $4\times4$ submatrices of data within each shader. Additionally, we attempted to force blocked texture and framebuffer access across fragments by altering the size and order of geometric primitives sent to the card. We observed that the rasterization order of fragments produced by a single primitive covering the entire framebuffer is as efficient as any ordering created as a result of tiling the screen with geometry. Utilizing multiple shader outputs, despite allowing more computation to be performed per data access, yielded unsatisfactory results. Despite our optimization attempts, the two algorithms described above yielded the best performance.

## 3. Performance Results

We benchmarked our GPU algorithms and the CPU based matrix-matrix multiplication routine (sgemm) provided by ATLAS. Experiments were performed on a 3 GHz Pentium 4 CPU (512 KB L2 cache) featuring peak performance of 12 GFLOPS and an L1 cache bandwidth of 44.7GB/sec. We tested our GPU algorithms on the ATI Radeon 9800XT, a prerelease Radeon X800XT (500mhz core clock/500mhz mem), the NVIDIA GeForce FX 5900 Ultra, and a prerelease GeForce 6800 Ultra (350Mhz core/500Mhz mem), capable of achieving 26.1, 63.7, 40.0, and 43.9 GFLOPS respectively. We only measured the **NV Single** algorithm on the NVIDIA hardware because for large matrices its kernel requires more instructions than the ATI card supports.

We seek to specifically evaluate the computational performance of our GPU implementations and so explicitly exclude the overhead of repacking input matrices in system memory, transferring them to the graphics card, and initially

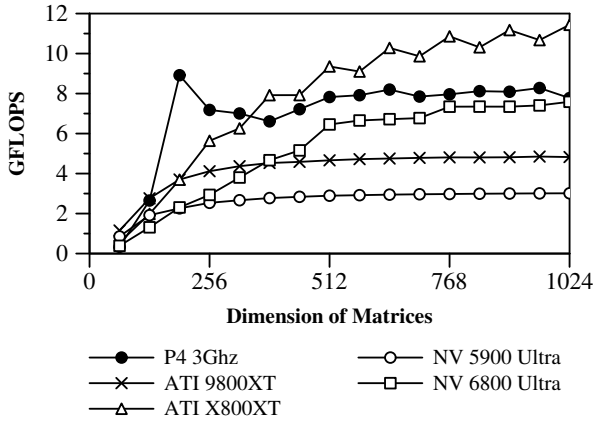


**Figure 1:** *Performance of multiplying square matrices on a 3 GHz Pentium 4, NVIDIA GeForceFX 5900 Ultra, prerelease GeForce 6800XT, ATI Radeon 9800XT, and prerelease Radeon X800XT.*

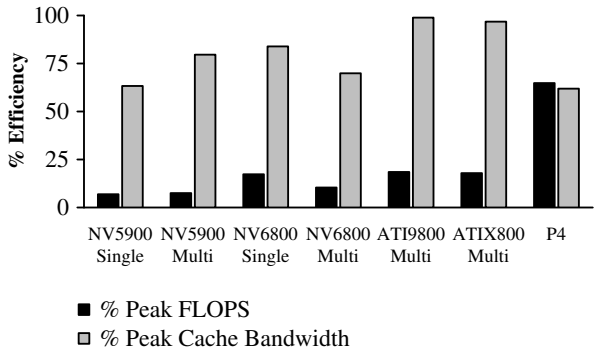


**Figure 2:** *Percentage computational and bandwidth efficiency when multiplying 1024x1024 matrices.*

loading and binding shader programs. In contrast, rather than attempt to instrument ATLAS code, we measure the entire time the ATLAS routine runs. ATLAS' initial self-tuning metacompilation, however, was performed during library installation and is not included in the timings. A summary of algorithm performance on the different platforms is given in Figure 1. GFLOPS values count each multiplication and each addition as distinct floating point operations (a `mad` instruction using `float4`s is thus eight operations).

ATLAS outperforms all GPU implementations except those running on the ATI X800XT despite being billed for any repacking or setup overheads and despite the significantly higher peak computational capability of all the graphics adapters. As shown in Figure 2, the ATLAS implementation achieves 65% efficiency while we observe no more than 17% efficiency on the NVIDIA chips and 19% on ATI platforms. Table 1 breaks down the multiplication of $1024\times1024$ matrices in more detail. In addition to the GFLOPS rate of each implementation, we computed effective input bandwidth. For the GPU implementations, we

| | Time (s) | GFLOPS | BW (GB/s) |
|---|---|---|---|
| NV 5900 Single | 0.781 | 2.75 | 7.22 |
| NV 5900 Multi | 0.713 | 3.01 | 9.07 |
| NV 6800 Single | 0.283 | 7.59 | 15.36 |
| NV 6800 Multi | 0.469 | 4.58 | 12.79 |
| ATI 9800 Multi | 0.445 | 4.83 | 12.06 |
| ATI X800 Multi | 0.188 | 11.4 | 27.5 |
| P4 ATLAS | 0.289 | 7.78 | 27.68 |

**Table 1:** *Measurements from the multiplication of 1024x1024 matrices.*

| | GFLOPS | Cache BW | Seq BW |
|---|---|---|---|
| NV 5900 | 40.0 | 11.4 | 4.1 |
| NV 6800 | 43.9 | 18.3 | 3.8 |
| ATI 9800 | 26.1 | 12.2 | 7.3 |
| ATI X800 | 63.7 | 28.4 | 15.6 |

**Table 2:** *Peak computation and bandwidth rates (cache hit and sequential fetch, in GB/sec) from our microbenchmarks.*

measure bandwidth as the total number of bytes fetched via `tex` instructions divided by total time to run our algorithm using a modified shader with nearly all math operations removed (drivers eliminate texture instructions if data is never used by shader, thus we cannot eliminate all math operations). Since we do not know the extent to which ATLAS is able to avoid memory accesses by reusing operands in registers, we compute its bandwidth as if all $2n^3$ accesses were from memory. In practice, we expect this significantly overestimates ATLAS' effective memory bandwidth. On the ATI platforms, bandwidth is computed assuming the transfer of 4 words per floating point value, although computation is performed at 24 bit precision.

Table 2 shows the results from a trio of microbenchmarks on our graphics cards. The first measures peak arithmetic performance by running a shader consisting entirely of `mad` instructions. The second measures peak bandwidth by repeatedly accessing texel (0, 0) from a number of input textures. We measure throughput approaching that of the theoretical peak bandwidth of the cards. The final microbenchmark runs a similar test, but walks the input textures sequentially by mapping each texel of the input textures to each rendered fragment. This is essentially the bandwidth realized by kernels that perform pure streaming over their inputs.

The key to the poor efficiency of the GPU algorithms can be seen by comparing Tables 1 and 2. The implementations are only able to utilize a small fraction of the GPUs' computational power despite data access patterns that yield high bandwidth. This inefficiency does not result from poor algorithm design or sub-optimal implementation. **ATI Multi** achieves over 95% of the bandwidth of perfect cache accesses yet cannot keep the computation units busy. The NVIDIA algorithms also leave chip arithmetic units idle despite accessing data at a large percentage of peak bandwidth. From the microbenchmarks and the cards' clock rates we compute that the 6800 Ultra and X800XT can perform 16 `mad`'s per clock and fetch 16 floats per clock. Since our four component `mad` takes 8 floats as input (and one register for the third operand), a shader with optimal memory access would require in excess of 8 math operations *per float fetched* in order to fully utilize the cards' computational capacity. Our matrix-matrix multiplication kernels cannot approach this ratio given current hardware constraints on instruction counts, register space, and the number of shader outputs.

## 4. Discussion and Conclusions

We expected that matrix-matrix multiplication would be a natural fit for the GPU and instead observe that optimized implementations achieve bandwidth approaching that of a theoretically optimal access pattern and yet achieve no better than 19% utilization of the arithmetic resources on our ATI hardware and no better than 17% utilization on our NVIDIA hardware. Our intuition proved completely misleading. Given current hardware designs, no algorithm will perform significantly better than existing approaches.

Indeed, we discover that available floating point bandwidth from the closest cache on a GPU is up to several times lower than that of current CPUs to their L1 caches. Without increasing this bandwidth (we expect the ratio of available computation to bandwidth to slowly worsen unless architectural design efforts specifically strive to reduce it), the efficiency of computing matrix-matrix products on graphics hardware will remain low.

One obvious way to increase performance is to perform more computation on data each time it is fetched by performing a level of blocking in the shader register file. The amount of potential reuse increases with the size of the submatrix multiplication performed in a shader. Current architectures provide sufficient registers to multiply two $6\times6$ matrices in a single shader, but do not permit shaders to generate more than a small number of outputs. As a result, we cannot use larger blocking within shaders to increase the compute to fetch ratio of our algorithms.

Multipass algorithms rely upon the ability to accumulate partial results into a final output. GPUs currently cannot accumulate into a full precision floating point framebuffer so our kernels must explicitly fetch the intermediate results produced by prior passes. However, this only accounts for 1/6

of the total input bandwidth of **NV Multi** and even less for **ATI Multi**. The benefit of adding hardware support for accumulation into floating point buffers pales in comparison to the advantages of significantly widening the path from the texture fetch units to the arithmetic units or the ability to do register level blocking.

We realize that when evaluating non-rendering algorithms on GPUs, it is important to remember that the design of graphics architectures is optimized for graphics workloads. Graphics applications primarily use 8-bit fixed point formats. Since four times as many values can be fetched per clock, compute to fetch ratios are much closer to those of conventional CPUs. Additionally, shaders often utilize interpolated values and constant parameters instead of texture memory for the majority of their inputs. Thus, graphics applications are more likely to perform many instructions per texture fetch. In contrast, practical numerical applications use full precision floating point values and depend upon iterating through input textures for access to data. Rebalancing GPUs to include wider or closer caches or an increased ability to operate on and output larger amounts of data per shader would make GPU architectures better suited for numerical workloads. Even streaming applications, already well suited for GPU architectures, would benefit from bandwidth improvements for regularly patterned texture accesses.

Although we investigate only matrix-matrix multiplication in detail in this paper, our arguments apply to many algorithms that feature data reuse. A wide variety of linear algebra operations rely upon blocking to amplify bandwidth and similar concerns apply to many general algorithms that repeatedly compute on large 2D (or n-D) pieces of memory. So long as graphics architectures continue to provide a low bandwidth connection between arithmetic units and nearby local memories, such algorithms will continue to suffer in efficiency when adapted to GPUs, despite increases in GPU clock rates and raw computational power, and despite the high levels of parallelism and numerical complexity inherent in the operations. Addressing these issues in future hardware designs will make GPUs a significantly more powerful platform for this broad class of applications.

## 5. Acknowledgments

We would like to thank ATI and NVIDIA for providing access to their latest hardware, and Sean Treichler of NVIDIA and Steve Morein for invaluable discussions about the design of graphics architectures. Support for this research was provided by the Rambus Stanford Graduate Fellowship and Stanford School of Engineering fellowship programs.

## References

[ABB*99] ANDERSON E., BAI Z., BISCHOF C., BLACKFORD S., DEMMEL J., DONGARRA J., DU CROZ J., GREENBAUM A., HAMMARLING S., MCKENNEY A., SORENSEN D.: *LAPACK Users' Guide*, third ed. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1999.

[BFGS03] BOLZ J., FARMER I., GRINSPUN E., SCHRODER P.: Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Trans. Graph. 22*, 3 (2003), 917–924.

[BFH*04] BUCK I., FOLEY T., HORN D., SUGERMAN J., FATAHALIAN K., HOUSTON M., HANRAHAN P.: Brook for GPUs: Stream computing on graphics hardware. In *Proceedings of ACM SIGGRAPH 2004 (to appear)* (2004).

[HCH03] HALL J. D., CARR N. A., HART J. C.: Cache and bandwidth aware matrix multiplication on the GPU. *UIUC Technical Report UIUCDCS-R-2003-2328* (2003).

[HCSL02] HARRIS M. J., COOMBE G., SCHEUERMANN T., LASTRA A.: Physically-based visual simulation on graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (2002), Eurographics Association, pp. 109–118.

[HSU*01] HINTON G., SAGER D., UPTON M., BOGGS D., CARMEAN D., KYKER A., ROUSSEL P.: The microarchitecture of the pentium 4 processor. *Intel Technology Journal 22*, 1 (2001).

[KW03] KRUGER J., WESTERMANN R.: Linear algebra operators for GPU implementation of numerical algorithms. *ACM Trans. Graph. 22*, 3 (2003), 908–916.

[LM01] LARSEN E. S., MCALLISTER D.: Fast matrix multiplies using graphics hardware. In *Proc. Supercomputing 2001* (2001).

[MA03] MORELAND K., ANGEL E.: The FFT on a GPU. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (2003), Eurographics Association, pp. 112–119.

[Mor03] MORAVANSZKY A.: Dense matrix algebra on the GPU, 2003. http://www.shaderx2.com/shaderx.PDF.

[TCL98] THOTTETHODI M., CHATTERJEE S., LEBECK A. R.: Tuning strassen's matrix multiplication for memory efficiency. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)* (1998), IEEE Computer Society, pp. 1–14.

[WPD01] WHALEY R. C., PETITET A., DONGARRA J. J.: Automated empirical optimizations of software and the ATLAS project. *Parallel Computing 27*, 1–2 (2001), 3–35.