MATRIX-MATRIX MULTIPLICATION IN MATLAB USING THE GPU

ANDRÉ RIGLAND BRODTKORB
DEPARTMENT OF INFORMATICS
UNIVERSITY OF OSLO
BABRODTK@IFI.UIO.NO

ABSTRACT. The use of GPU's as the main computing resource has yielded great speed-up factors in several fields including solving differential equations, linear algebra, signal processing and database queries.

There have been several attempts at implementing efficient algorithms for matrix-matrix products with varying results. In-depth analysis of the algorithms has been presented as well. In this paper I review the work done in the field, and present a crude implementation of matrix-matrix products using the GPU. The implementation is run in Matlab.

1. INTRODUCTION

Matrix-matrix multiplication is an operation that occurs in many mathematical problems, including matrix solvers, the Linear Complementarity Problem (LCP) and others. Computing a matrix-matrix multiplication is a heavy process for both memory bandwidth and processor capacity. New graphics cards have programmable GPUs (Graphics Processing Units) which can be utilized to speed up the computation, or offload the CPU. The GPU out-conquers the CPU when it comes to both memory bandwidth and FLOPS (Floating Point Operations Per Second), making it a candidate for a powerful coprocessor.

1.1. The GPU.

The GPU is the processor on the graphics card. Newer GPUs can be programmed to render not only geometry, but also the solution to other, more general problems. Such programming of the GPU is often called GPGPU (General Purpose computation on Graphics Processing Units), and has become a field of great interest. The reason is its vast processing capabilities.

While the arithmetic capacity of the CPU has followed Moore's law, thus doubling every 18 months, the GPU has doubled its processing capacity every 6 months [OLG+05]. The GPU has already passed the speed of the CPU, and is far ahead. Current high-end CPUs have a theoretical memory bandwidth of about 10 GB/s, and can theoretically achieve about 13 GFLOPS (Intel Core Duo T2600). High-end GPUs, on the other hand, have a theoretical memory bandwidth of about 50 GB/s, and theoretically achieve over 370 GFLOPS (ATI Radeon X1900 XTX). That is five times the memory bandwidth, and almost 30 times the processing capability. The prices are comparable as well. The Intel Core Duo T2600 is priced at about 600 USD, while the ATI Radeon X1900 XTX is sold for approximately 800 USD.

But the two architectures are very different. The CPU is constructed for sequential execution of operations, and operates on single data-values. The input to the processor is *one* instruction, as well *one* a data-value to apply the instruction to. The output is a single value as well. This architecture is called SISD (Single Instruction Single Data).

The GPU on the other hand is a collection of several processing units. It is divided into two subclasses, the vertex and the fragment processors. The ATI Radeon X1900 XTX has eight vertex, and 48 fragment processors. These processors are divided into 16 pipelines which operate in parallel.

The vertex processors processes the placement of vertices from world coordinates onto normalized view coordinates, and clips away what we cannot see. A vertex is a node in a geometry (e.g., one corner in a triangle).

After the vertex processor has transformed all vertices into normalized view coordinates, the scene is flattened into 2D, and sampled at regular intervals. This output is fragments, and the process is called rasterization. Fragments are pixels which have not yet been rendered to screen. They contain values interpolated from the vertices, such as color, normals and texture coordinates.

The fragment processor processes these fragments, and determines their final color. In the fixed function pipeline (i.e., the non-programmable pipeline), the final color is determined from texture lookups, lighting as well as the original interpolated color value. But when programming this stage, we can ourselves determine how we want to calculate the final color. And it is in this stage, in the fragment processor, that most of the processing capability lie.

The fragment processor is a SIMD (Single Instruction Multiple Data) processor in todays consumer level graphics cards, in contrast to the SISD architecture

of the CPU. This architecture is both the Achilles heel and the reason for its vast processing capabilities. The SIMD architecture of the fragment processor is a very powerful parallel architecture, but it is also very slow when it comes to certain areas such as branching [OLG+05]. Some problems are not even possible to solve on the GPU due to the restraints of the architecture. But if a problem fits the GPU programming model, it can yield large rewards. In simulation, which often include matrix multiplication, speedup factors of $15\times$ to $30\times$ have been presented [HHL+05].

1.2. **Motivation.** Matrix-matrix multiplication is a slow process. The standard way of multiplying two $N \times N$ matrices requires $O(N^3)$ multiplications, as well as memory access to all elements. The fastest known algorithm, first published by Coppersmith et. al. [CW87] in 1987, is of order $O(N^{2.38})$, but most researchers believe that $O(N^2)$ is optimal [Rob05].

If we assume that $O(N^2)$ is optimal, doubling the matrix size increases the complexity, and thus the computational time, by *at least* the square of N. For small matrix sizes, this is not a big issue. But for larger matrices, the computational time can become to long for practical use. Most real world problems end up in large matrices which are slow to solve. Especially when dealing with time dependent data where several matrix multiplications have to be computed per time-step, the computational time can become too long.

Using the GPU as a coprocessor will not only offload the CPU, in order for it to compute other tasks, but hopefully also speed up the computation itself. The rewards can be huge, and the commercial interests equally large.

Most scientists do not want to spend a lot of time programming. They want to develop algorithms, and quickly implement them. A tool often used for many problems is Matlab. Matlab is an easy programing language which can interactively be programed. It is a mathematical language with a lot of mathematical functions available, including matrix multiplication.

Implementing a matrix multiplication algorithm in Matlab that is faster than the provided, will not only speed up the computational time, but also let the scientist use a familiar and fast development tool. For problems involving a lot of matrix multiplication, this can save a lot of computational time, and possibly prevent the need to port the program to a faster language such as C or Fortran.

1.3. **Matrix multiplication on the GPU.** The GPU programming model enforces certain restraints which have to be taken into account when implementing algorithms. Branching is expensive, and all operations have to be independent of each other (i.e., one cannot implement an algorithm that is dependent on the order of computation.). Fortunately, matrix multiplication fits the GPU programming model, because of its highly parallel nature.

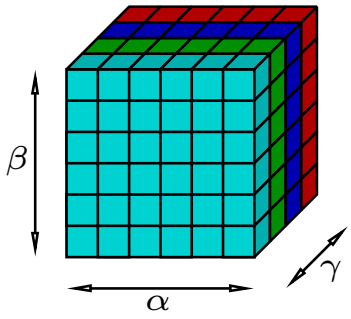

Figure 1. Visualization of matrix-matrix multiplication using a virtual cube of processors

Matrix multiplication can be seen as a parallel problem because all computation is independent of each other, and the same operation is applied to all elements. But in order to compute a matrix multiplication on the GPU, the problem has to be rephrased into graphical primitives rendered on screen.

In the following I review the work done in the field, present a crude implementation for Matlab use, and analyze the results.

2. Background

There have been several different approaches to implementing matrix-matrix multiplication on the GPU. The leap is quite large from forward looking research when only byte sized buffers were available [LM01], to Moravánszky's [AM03] approach to a GPU competitor to BLAS [LHKK79]. Memory and GPU use in this setting has been deeply analyzed [FSH04], and some weaknesses have been discovered.

2.1. **Byte-sized precision.** Larsen and McAllister were the first to implement matrix-matrix multiplication on the GPU [LM01]. They applied techniques from parallel processing to compute arbitrary matrix-matrix multiplications. Without using any shader language, and thus left with standard OpenGL function calls, they computed matrix-matrix multiplications achieving the same number of operations per second as a highly optimized CPU code, the ATLAS [WD98] library. The ATLAS library is a self tuning implementation of BLAS [LHKK79], a very efficient linear algebra library.

Their implementation is based on a virtual cube of processors, where each processor executes a single multiplication. The sub-results are then summed to compute the final result. The algorithm is as follows:

(1) The programmer visualizes a virtual array of processors with dimensions α, β, and γ (see Figure 1). These dimensions correspond to the matrix sizes where A is an $\alpha \times \gamma$ matrix, and B is a $\gamma \times \beta$ matrix.

(2) The programmer then gives each little processor in the (α, γ) plane a value from A that corresponds to the position of the processor in the plane. All of the planes in the β direction are given the same value as the first plane, effectively replicating the values throughout the cube.
(3) The (γ, β) plane is given a value from B corresponding to the position of the processor, in the exact same manner as A is distributed. The only difference is that the data is now copied throughout the array in the α direction.
(4) Now every processor has a value from A and a value from B. Processor $P_{i,j,k}$, where (i, j, k) denote the position, has the values $A_{i,k}$ and $B_{k,j}$.
(5) Each processor computes it's little multiplication. In order to compute the final result, all of the planes in the γ-direction are summed. All of the processors in the γ-direction holds the result of $P_{i,j,k} = A_{i,k} \cdot B_{k,j}$, where k corresponds to the position along the γ dimension. The sum along the γ-direction can be written as $\sum_k P_{i,j,k} = \sum_k A_{i,k} \cdot B_{k,j}$ which is the definition of matrix-matrix multiplication.

The great advantage of this type of implementation is that it requires nothing more than standard OpenGL function calls. It is also a well known and tested technique applied in parallel computing. The implementation of Larsen and McAllister ran on a GeForce 3 graphics cards. Their research was not practical, but rather theoretical and forward-looking. This because of the limitations in hardware:

- The GeForce 3 only supported byte-sized operations.
- The internal precision of the GeForce 3 was limited to 16 bits.
- Graphic cards used saturation arithmetic[1].

The first two limitations have become less important with the emergence of floating-point precision. But floating-point precision is still to little for many numerical applications and double precision is needed.

The primary drive behind the development of graphics cards is the gaming industry. It was the gaming industry that brought the need for floating-point buffers when byte-sized buffers didn't give enough precision. Floating-point buffers, however, seem to be more than the gaming industry needs in order to calculate beautiful visual effects. This, sadly, seems to prevent the expensive development of double precision buffers. At best, the development of double precision buffers are on the horizon [OLG+05].

The last limitation, however, is a bit more troublesome. To remedy this, one can scale all values so that their sum must be below the largest representable value [LM01]. The downside of this is that you increase the number of erroneous digits. There exist classic algorithms that improve precision, but unfortunately these cannot be used with saturation arithmetic.

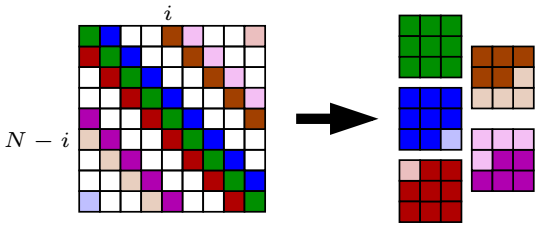

Figure 2. Representation of banded matrices as a set of diagonal-vectors.

2.2. **A GPU BLAS library.** Moravánszky [AM03] proposed a GPU BLAS library to compete with the BLAS implementations known from the CPU world. Only implementing some of the basic functionality, the results are mostly of interest for further research, and as a proof of concept. When compared to the highly optimized ATLAS implementation for linear algebra, his implementation used 40% of the time ATLAS used on the Conjugate Gradients problem, and 25% when solving the Linear Complementarity Problem. The tests were run on a P4 1.6GHz with a Radeon 9700 Pro graphics card on 1000×1000 matrices.

Another similar approach was presented by Krüger and Westermann [KW03]. They had a program structure consisting of functions which you can pass operators to. Their function clMatVec computes Ax op y where you pass op to the function, as well as the matrices A, x and y.

Fatahalian, et. al. [FSH04] discovered that using 2D textures to represent vectors was twice as fast as using one dimensional textures, as well as enabling representation of far larger vectors. Krüger et. al. [KW03] used this knowledge in their implementation, and represented all banded matrices as diagonal vectors packed into small 2D textures (see Figure 2).

Their algorithm for packing the data is as follows:

(1) The diagonal is stored as a 2D texture.
(2) Each diagonal with a non-zero entry is stored in a 2D texture. Diagonal i is joined with diagonal $N - i$, where N is the size of the matrix.
(3) If i or $N - i$ consist of only zeros, it is still padded, as long as one of them contain a non-zero entry.
(4) The remaining diagonals have only zeros, and are discarded from further computation.

Using this technique they reduced the execution time from 0.23 seconds to a mere 0.72 milliseconds. The matrix size was 4096×4096 with ten diagonals with

1. Saturation arithmetic is arithmetic which gets saturated. The effect is that if you add something to the largest representable value, the result is the largest representable value (i.e., 10 + 1 = 10 if 10 is the largest number you can represent).

non-zero entries, multiplied by a 4096 vector. This is particularly useful for many numerical simulations, such as the heat equation discretisized over a regular grid. In one dimension the discretization leads to three non-zero diagonals, and five in two dimensions.

2.3. **Analysis.** Fatahalian, Sugerman and Hanrahan [FSH04] investigated the efficiency of matrix-matrix multiplication on the GPU. They implemented two different shaders, the first a single pass algorithm, while the second a multi-pass. A single-pass shader is a shader that is invoked once, whereas a multi-pass shader is invoked multiple times. This can be done by drawing many small quads covering the data instead of one large quad. The reason for multi-pass algorithms is restrictions in the drivers of the graphics cards limiting the number of operations.

Their deep analysis of the algorithms points out that memory bandwidth is the limiting factor in matrix-matrix multiplication. When comparing FLOPS to the highly optimized ATLAS code, they observed that it achieved about 65% efficiency, while their best result with a graphic card was 19%. This means that the processors are effectively idling 80% of the time.

When comparing memory access, they could show that the graphics cards peaked the memory bandwidth while the ATLAS implementation achieved about 65% efficiency. It didn't even help trying to block the data on the GPU using multiple quads. To block the data is to partition it into smaller blocks which fit into the cache in order to maximize the use of the high cache bandwidth. A single large quad covering the whole matrix was equally efficient as many smaller.

The reason Fatahalian, Sugerman and Hanrahan point out for the low utilization of arithmetic units was the difference in bandwidth to the closest cache on the two platforms. The CPU has a much higher bandwidth than the GPU has to its closest cache. This despite the fact that the GPU has a much higher bandwidth to it's main memory than the CPU has to RAM [FSH04].

Their conclusion was that as long as the hardware does not dramatically alter the bandwidth to the closest cache, matrix-matrix multiplication will remain inefficient. Their prediction is not that this bandwidth will increase, but rather slowly decrease relative to the computational power.

2.4. **Efficient data-structures.** The GPU is constructed for calculating a color in the RGBA color-space. The RGBA color-space consists of Red, Green, Blue and Alpha. The Alpha channel represents opacity, adding the ability of transparency to the RGB color-space. This construction makes sense when handling colors, since one Multiply And Add (MAD) instruction can be processed per pixel (consisting of four colors) per clock cycle. This specialization, however, makes it harder to utilize all of the GPU's processing capability. When using just one color, three out of four arithmetic units are simply idling.

Krüger et. al. [KW03] implemented the technique as described by Moravánszkyi [AM03]. They packed four consecutive elements of a matrix into a single texel (see Figure 3), thus using all four color channels. Fatahalian et. al. [FSH04] used another approach proposed by Hall, Carr and Hart [HCH03], where they packed a small two-by-two matrix into each texel. They also implemented the four-by-one packing scheme, and analyzed the efficiency of both.

The two different algorithms have different abilities. Packing into a four-by-one texel requires $1.5\times$ more texture reads [FSH04], but packs data in a more cache-friendly manner. Fatahalian et. al. showed that with the increase in cache hits, this packing of data was the fastest of the two.

Listing 1. Matlab function call.

```
[a, b, ...] = func(α, β, ...);
```

3. A MATLAB IMPLEMENTATION

I have implemented a crude matrix-matrix multiplication algorithm on the GPU. The program is compiled into a MEX file, runnable from Matlab. A MEX file is a C/Fortran program that is compiled with the Matlab Mex script, producing a function callable from Matlab in a Dynamic Link Library [Mat06c].

Implementing a shader using Matlab is a bit more tricky than implementing the same functionality as a standalone executable. This is because of the way Matlab interacts with it's MEX files. Once Matlab runs a MEX file, it acquires a file lock for the MEX file. When debugging, this requires you to restart Matlab each time you encounter an error in order to test your new code. Even after you have debugged your C/C++ code, you are left with incomplete debugging tools for the Shader programming.

3.1. **Matlab entry point.** In order for Matlab to use a function defined in an external library, it needs to have a specific entry point. This entry point is called mexFunction which is the heart of any MEX file. The function is defined as shown in Listing 2. Matlab also has bindings to Fortran, but these are not as useful as the C bindings when programming with OpenGL.

From the Matlab function call in Listing 1 the arguments $(\alpha,\ \beta,\ \ldots)$ are passed to the appropriate MEX function via `nrhs` and `prhs[ ]`. `nrhs` is the number of arguments passed, while `prhs[ ]` contains pointers to the data $(\alpha, \beta, \ldots)$ in a C struct, `mxArray`. There can be multiple outputs from the function, and this is done via the variables `nlhs` and `plhs[ ]`, where `nlhs` is the number of outputs, and `plhs[ ]` contains pointers to the data `(a, b, ...)`.

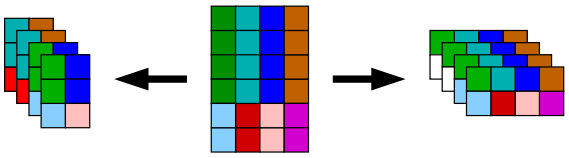

Figure 3. Packing of elements from a matrix into a single pixel. The center is the original matrix we want to pack. The left-hand figure is packed using the two-by-two algorithm, and the right is packed with the four-by-one algorithm. Notice that the four-by-one algorithm needs padding since the height of the texture is not divisible by four, requiring two more pixels than the two-by-two algorithm. The two-by-two algorithm will also need packing if the dimensions of the original matrix are not divisible by two.

Listing 2. MEX file entry point.

```
1 void mexFunction(int nlhs, mxArray* plhs[], int nrhs, const mxArray* prhs[]) {
2   //Body of function
3 }
```

Apart from the entry point, Matlab also requires you to use Matlab's own functions for dealing with memory. This is to prevent memory leaks, and all memory allocated with these functions are automatically added to Matlab's memory management system.

3.2. **Program structure.** The program structure bears sign of the restraints enforced by Matlab. The heart of the program is the mexFunction, which has the same role as main in a standalone C application. The reason the program has to be called this way is because Matlab only can call the mexFunction, just like an ordinary executable only calls main.

I use the first parameter passed from Matlab as a function selector, which corresponds to different internal functions. By abstracting away the most basic of GPU programming, matrix-matrix multiplication can easily be done with just a few lines of Matlab code (see Listing 3).

The trade-off between generality and efficiency is the Achilles heel of any abstraction, and becomes even more vital in Matlab. Calling C-functions from Matlab is considerably slower than calling the C-function directly. Hence, abstracting away too little gives a lot of overhead in function calls, while abstracting away too much lessens the usability of the functions.

My program design is influenced by the work done by Gary Holt [Hol01]. His Matwrap program vectorizes C/C++ functions, and wraps them for Matlab use. The input is the header-file which you want to call from Matlab, and the outputs are Matlab functions ready to be compiled with the MEX script.

In order to run you C/C++ programs in Matlab, all you have to do is Matwrap and MEX them. The wrapping of functions in C does not work as well as writing programs specifically for Matlab, because the functions themselves are not written with Matlab use in mind. Writing a function purely for Matlab use enables the programmer to vectorize the functions internally, as well as using the Matlab MEX API properly.

3.3. **Garbage collection.** When Matlab runs a MEX file, it waits until the function has cleared. When the function has cleared, all memory allocated by the MEX API is automatically freed. All of the free calls are in C libraries, but run from Matlab. This is a slow process, as discussed in Section 3.2. It is important that the programs free their own memory when clearing in order to execute as efficiently as possible. But there are some cases where the internal Matlab garbage collection is needed to prevent memory leaks [Mat06b]:

(1) A call to `mexErrMsgText`.
(2) An error occurs in a `mexCallMATLAB` call.
(3) The user aborts the MEX execution (with `[Ctrl]+C` for example).
(4) The MEX file runs out of memory.

The first two cases can be circumvented in the C code if programmed correctly. The two last cases, however, cannot be predicted or cleaned up properly by the C function alone. This is why it is extremely important to use the MEX API's functions when handling dynamic memory. Without proper use of the functions in the API, the MEX file will leak memory.

When working with OpenGL and texture memory, this becomes a great problem. The MEX API does not have support for automatic clearing of texture memory, as this is a rare application of the API. Cases `(1)` and `(2)` can still be solved, but cases `(3)` and `(4)` lead to texture memory leaks. When working with large matrices, the texture memory very quickly gets eaten up.

To prevent this problem, the MEX API has a function `mexAtExit()`, which tells Matlab what function it needs to call when the function clears. By properly disposing of texture memory in the function `cleanup()`, and calling `mexAtExit(cleanup);` in your C code, you can prevent texture memory leaks.

Listing 3. GPU matrix-matrix function (Matlab part)

```
function C = gpuMult(A, B);
  //Check input matrices
  if (size(A,2) ~= size(B,1))
    error('gpuMult:invalidInput', '%s', 'Matrix sizes do not agree');
  end

  //Render the result
  C = gpuLAS(0, A, B);
```

4. Comparison

The algorithm for matrix-matrix products I have implemented has the ambition of becoming a competitor to the internal ATLAS implementation of Matlab. It is therefore the most natural implementation to compare against. The ATLAS implementation within Matlab is a highly tuned cache-aware library built upon BLAS. I am therefore comparing a crude implementation on the GPU to a highly tuned and optimized CPU implementation.

4.1. **gpuLAS and Matlab.** As mentioned in Section 2.4, packing of data is important to utilize all arithmetic units. But due to time constraints, I have not implemented any packing of data in gpuLAS yet, thus only using one fourth of the processor capacity.

Matlab uses the highly efficient ATLAS implementation when calculating with matrices and vectors [Mat06a]. Since the data already is on the CPU platform I have compared the time of the calculation, and discarded the time it takes to set up OpenGL, and transfer the matrices to the GPU. This has become a standard way of benchmarking matrix-matrix multiplication on the GPU [OLG+05, LM01, KW03, FSH04].

Figures 4 and 5 show a graphical representation of the results. Matlab was run on a Pentium IV 3.0 GHz with 2 GB RAM, and gpuMult was run on the same system, the graphics card being an Nvidia 7800GT with 256MB texture memory. The CPU times can be approximated by a linear function, $f(N) = c \cdot N$ where N is the matrix size (for an $N \times N$ matrix) and c is a constant. The GPU times, however do not fit this linear function, but they fit a logarithmic function $f(N) = c \cdot \log(N)$ better. For sufficiently large matrices, the two functions will cross, and the GPU version will be faster.

The execution time of gpuMult is about 0.5 seconds slower than Matlab for 2048×2048 matrices, as shown in Figure 4. It is 0.5 seconds slower for 1024×1024 matrices as well, which indicate that the GPU implementation will be faster for a matrix size larger than 2048×2048 when seen in context with Figure 5.

The timings are taken without the constant time it takes to set up the drivers and transfer the data to the GPU, but include the time it takes to transfer the result back. The algorithm implemented is not efficient at all. It does not reuse fetched data, nor does it utilize all the arithmetic units it can.

When the implementation of gpuMult includes these features, the algorithm will be able to perform four times as many MAD instructions, thus reducing the execution time considerably. The effect is that the time it now takes to compute two $N \times N$ matrices will approximately be the same as computing two matrices of size $\frac{N}{2} \times \frac{N}{2}$.

5. Summary

Linear algebra seemed like a perfect candidate for implementation on the GPU. The problem has a highly parallel nature, and fits the GPU programming model nicely.

But even though relatively high speedup factors have been presented, there are hardware limitations hindering full use of the GPU's arithmetic units. The main limitation is the lacking bandwidth to cache. Even though the GPU has a high bandwidth to its texture memory, it lacks the efficient bandwidth to its cache which is present on the CPU. The cache bandwidth on the CPU is several times faster than that on the GPU, which leads to the GPU idling 80% of the time while waiting for data.

When implementing an algorithm, the efficiency depends on the use of the GPU's arithmetic units. To get an efficient implementation, the data has to be packed into the four color channels in a way that reuses fetched data. The more reuse of data, the more efficient the algorithm will perform.

There is also a constant price to pay when utilizing the GPU as a computational resource. The driver has to be initialized, and the data has to be transferred to the GPU. If computing a single matrix-matrix product, the GPU will not be the best alternative because of this constant price. But if the reuse of the matrices and number of products computed are high, the the GPU can give a large speed-up factor.

References

[AM03] Ádám Moravánszky. Dense matrix algebra on the gpu. Online; `http://www.shaderx2.com/shaderx.pdf`, 2003. [accessed 2006-05-11].

[CW87] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *STOC '87: Proceedings of the nineteenth annual ACM conference on Theory of computing*, pages 1–6, New York, NY, USA, 1987. ACM Press.

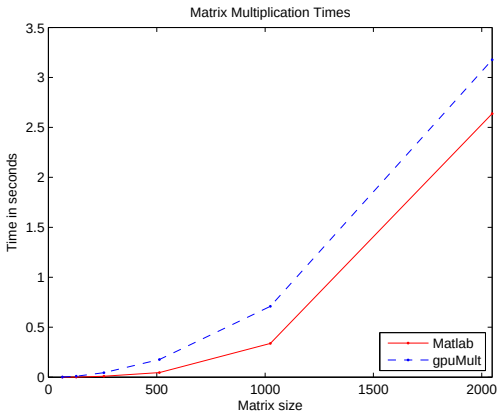

Figure 4. Plot of matrix-matrix multiplication times.

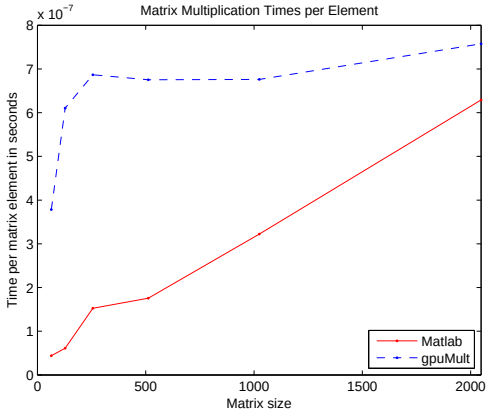

Figure 5. Plot of matrix-matrix multiplication times per element.

[FSH04] K. Fatahalian, J. Sugerman, and P. Hanrahan. Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 133–137, New York, NY, USA, 2004. ACM Press.

[HCH03] J. Hall, N. Carr, and J. Hart. Cache and bandwidth aware matrix multiplication on the gpu, 2003.

[HHL+05] Trond Runar Hagen, Jon Mikkelsen Hjelmervik, Knut-Andreas Lie, Jostein R. Natvig, and Martin Ofstad Henriksen. Visual simulation of shallow-water waves. *Simulation Modelling Practice and Theory Journal*, 13(8):716–726, November 2005.

[Hol01] Gary Holt. Matwrap: a wrapper generator for matrix languages. Online; `http://lnc.usc.edu/~holt/matwrap/`, 2001. [accessed 2006-05-15].

[KW03] Jens Krüger and Rüdiger Westermann. Linear algebra operators for gpu implementation of numerical algorithms. *ACM Trans. Graph.*, 22(3):908–916, 2003.

[LHKK79] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, 1979.

[LM01] E. Scott Larsen and David McAllister. Fast matrix multiplies using graphics hardware. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 55–55, New York, NY, USA, 2001. ACM Press.

[Mat06a] The MathWorks. Matlab software acknowledgements. Online; `http://www.mathworks.com/access/helpdesk/help/base/relnotes/software.html`, 2006. [accessed 2006-05-14].

[Mat06b] The MathWorks. Memory management. Online; `http://www.mathworks.fr/access/helpdesk/help/techdoc/matlab_external/f25255.html`, 2006. [accessed 2006-05-26].

[Mat06c] The MathWorks. Mex-files guide. Online; `http://www.mathworks.com/support/tech-notes/1600/1605.html`, 2006. [accessed 2006-05-13].

[OLG+05] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, August 2005.

[Rob05] Sara Robinson. Toward an optimal algorithm for matrix multiplication. *SIAM News*, 38(9), November 2005.

[WD98] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society.