The Graphics Processor as a Mathematical Coprocessor in MATLAB *

André Rigland Brodtkorb

2008

Abstract

We present an interface to the graphics processing unit (GPU) from MATLAB, and four algorithms from numerical linear algebra available through this interface; matrix-matrix multiplication, Gauss-Jordan elimination, PLU factorization, and tridiagonal Gaussian elimination. In addition to being a high-level abstraction to the GPU, the interface offers background processing, enabling computations to be executed on the CPU simultaneously. The algorithms are shown to be up-to 31 times faster than highly optimized CPU code. The algorithms have only been tested on single precision hardware, but will easily run on new double precision hardware.

1 Introduction

The graphics processing unit (GPU) is the processor on graphics cards, dedicated to rendering images on screen. The rendered images typically consist of millions of pixels that can be computed in parallel. The GPU exploits this fact, and exhibits high levels of parallelism with up-to several hundred processors. Recent generations of off-the-shelf GPUs have become programmable, enabling the use of GPUs for general purpose computations. This has opened a new field of research called GPGPU.

The reason for interest in GPUs is their massive floating point performance. They offer far higher peak performance than CPUs, and the performance gap is increasing. While the processing power of CPUs has followed Moore's law closely, doubling every 18-24 months, the processing power of the GPU has doubled every 9 months [OLG+07]. An argument against using GPUs, however, has been the lack of double precision. This is now outdated with new double precision hardware.

Numerical linear algebra includes many computationally heavy operations that are central in many fields, ranging from search engines to games, cryptology and solving partial and ordinary differential equations numerically. Using the GPU to speed up such computations is important for all these applications. Harvesting the raw power of the GPU, however, is nontrivial and not even possible for some problems. In order to solve a problem using the GPU, it has to fit the GPU programming model and have a highly parallel nature. Traditionally, the GPU had to be accessed via a graphics API such as OpenGL [SWND05] and DirectX [Mic07], requiring that the problem is rewritten in terms of operations on graphical primitives. New vendor-specific APIs such as "Close To the Metal" (CTM) [Adv06] from AMD and "Computer Unified Device Architecture" (CUDA) [NVI07b] from NVIDIA, however, offer access to the hardware without going through the graphics API. There also exists two free APIs for GPGPU: Brook [BFH+04] and Sh [MDP+04]. These two, however, do not seem to be actively developed, as Sh has been commercialized as RapidMind [MD06], and Brook has been commercialized as Brook+ from AMD [Adv07].

This article presents four selected operators from numerical linear algebra implemented on the GPU. We have used OpenGL to access the GPU, as this was our best alternative before CUDA was released. The algorithms are chosen because of their importance and how they fit the

*This is a draft of the following article: A. R. Brodtkorb, The Graphics Processor as a Mathematical Coprocessor in MATLAB, The Second International Conference on Complex, Intelligent and Software Intensive Systems, pp. 822–827, March 2008, DOI: 10.1109/CISIS.2008.68.

GPU programming model: Full matrix-matrix multiplication is one of the building blocks in numerical linear algebra; Gauss-Jordan elimination is a direct solver that fits the GPU programming model well; PLU factorization is another direct solver, efficient for solving a system for multiple right hand sides; tridiagonal Gaussian elimination solves a tridiagonal system of equations efficiently. The four operators are accessed via MATLAB using familiar MATLAB syntax, and the MATLAB interface uses a processing queue and background processing on the GPU. This enables the use of the GPU *and* the CPU simultaneously for maximum performance.

2 Related work

Coupling the GPU and MATLAB has been shown by NVIDIA, who presented a MATLAB plug-in for 2D FFT using CUDA [NVI07a]. This speeded up simulation of 2D isotropic turbulence by almost a factor 16 on a 1024×1024 grid.

Larsen and McAllister [LM01] were the first to present matrix-matrix multiplication on the GPU, prior to the arrival of programmable hardware. Hall, Carr and Hart [HCH03] presented a way of blocking the computation by using several passes, and reported 25% less data transfer and executed instructions compared to the former. Jiang and Snir [JS05] showed a way of tuning matrix multiplication automatically to the underlying hardware, and reported 13 GFLOPS for a hand-tuned version, compared to 9 GFLOPS for the automatically tuned version on an NVIDIA GeForce 6800 U. For other hardware setups, the automatically tuned version outperformed hand-tuned equivalents. The efficiency of using blocking techniques for the matrix-matrix product was analyzed by Govindaraju et al. [GLGM06], where they reported less than 6% cache misses when using efficient block-sizes, and 17.6 GFLOPS on an NVIDIA GeForce 7900 GTX. Peercy, Segal and Gerstmann [PSG06] presented an implementation of matrix-matrix multiplication using CTM on an ATI X1900 XTX graphics card, where they reported their implementation to perform 110 GFLOPS.

Galoppo et al. [GGHM05] presented PLU factorization using the Doolittle algorithm on the GPU, and single-component textures to store the data. Using consecutive passes, they first located the pivot element. Then the rows (and columns for full pivoting) were interchanged in yet another pass, and finally the matrix was reduced. This process was repeated until the whole matrix was decomposed. They claimed their algorithm to be 35% faster than ATLAS [WD98] for PLU factorization with partial pivoting, and an order of magnitude faster than the Intel Math Kernel Library (MKL) [Int07] for full pivoting. However, the benchmarks were highly synthetic, and assumed no cache misses.

Bolz et al. [BFGS03] presented a conjugate gradient algorithm using the GPU, where they stored their sparse matrices using two textures. The first texture simply contained all nonzero elements in the matrix packed row by row. The second texture contained pointers to the first element in each row. Their conjugate gradient algorithm was implemented using the GPU to compute sparse matrix-vector multiplication and sparse vector-vector inner product. They reported overhead connected with pixel buffer switching as the limiting factor. This overhead is now far less when using framebuffer objects instead of pixel buffers. Krüger and Westermann [KW03] used another storing strategy for banded sparse matrices, where each diagonal-vector was stored in a separate texture. They reported precision issues, and claimed a speedup over their reference CPU implementation. Vectorized SSE implementations, however, are supposed to be $2 - 3\times$ faster than their CPU implementation.

Part of the material presented here is a result of the master's thesis "A MATLAB Interface to the GPU" [Bro07]. The reader is encouraged to consult the master's thesis for implementation and other details.

3 A GPU toolbox for MATLAB

MATLAB is a standard tool for scientists and engineers all over the world. Utilizing the GPU as a mathematical coprocessor will not only offload the CPU, but possibly also increase performance.

MATLAB supports user-defined classes and operator overloading for these classes. This enables us to implement a *gpuMatrix* class, which can be programmed to execute custom code for all standard MATLAB operators and functions. These functions can be programmed in C/C++ as a MATLAB executable (MEX) file. By programming the MEX file to use the GPU as the computa-

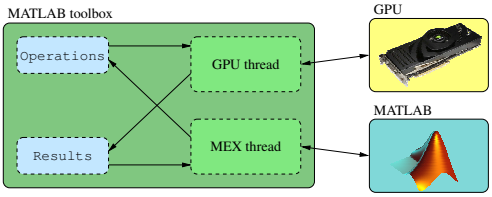

Figure 1: Splitting of program execution into one part dealing with MATLAB, and one part dealing with the GPU

tional engine, we can utilize the GPU in MATLAB. However, using the GPU most often excludes use of the CPU simultaneously because many calls to OpenGL are *blocking*. Synchronous data transfer to or from the CPU are examples of blocking calls where both the CPU and the GPU stop executing while data is transferred. Using the GPU to compute results in the background, however, will enable the use of the CPU simultaneously.

To use the GPU as a mathematical coprocessor, working in the background, we utilize threads that execute code independently from each other. Because neither MATLAB [KÖ7] nor most OpenGL driver implementations are thread-safe, we cannot arbitrarily call MATLAB and OpenGL functions from different threads. To circumvent this, the program execution is split into two separate threads, as shown in Figure 1. The *MEX thread* holds a MATLAB context and communicates with MATLAB, while the *GPU thread* holds an OpenGL context and communicates with the GPU. The two threads communicate with each other via a queue of operations, and a map of results.

When MATLAB operates on a gpuMatrix object, the MEX thread is called. It then creates the wanted operation, adds it to the operations queue, notifies the GPU thread of a change, and returns a unique ID to the operation. The GPU thread receives the notification, and executes all elements in the operations queue. Every completed operation is moved from the operations queue to the results map, where the ID of the operation is the key. When the result is requested by the user via a new call from MATLAB, the MEX file simply waits until the correct ID appears in the results map. When it is found, the result returned to MATLAB. The conversion between an operation ID and the corresponding matrix is transparent for the user, and an operation ID can be enqueued in further operations even before the result is computed.

4 Operators on the GPU

Data transfer between the GPU and the CPU has to pass through the 4GB/s full duplex PCI Express 16× bus [Mic04]. To prevent the need to repack data (and transfer it over the PCI Express bus again) for reuse in other GPU computations, we have implemented all algorithms using a consistent data-structure. We have used the 2×2 packing scheme proposed by Hall, Carr and Hart [HCH03], where 2×2 sub-matrices are packed into the four color vectors in each pixel (red, green, blue and alpha). This packing utilizes the vectorized arithmetic found in many GPUs, and offers good performance for most applications, even though other packing schemes might fit specific algorithms better.

4.1 Full matrix-matrix multiplication

We have implemented two versions of matrix multiplication. One single-pass, and one multi-pass. Hall, Carr and Hart [HCH03] presented a multi-pass algorithm that views the matrix-matrix product as a sum of individual multiplications. But because the matrix is packed using the 2×2 schema, the algorithm computes the "inner product" of two 2×2 matrices as

$$C_{i,j}^{k+1} = C_{i,j}^{k} + \begin{bmatrix} a_{i,2k+1} & a_{i,2k+2} \\ a_{i+1,2k+1} & a_{i+1,2k+2} \end{bmatrix} \begin{bmatrix} b_{2k+1,j} & b_{2k+1,j+1} \\ b_{2k+2,j} & b_{2k+2,j+1} \end{bmatrix}, \quad (1)$$

where $C_{i,j}^{k+1}$ is the result buffer, and $C_{i,j}^{k}$ is an intermediate accumulation buffer. The result is computed in $n/2$ passes, so that the product $AB = C^{n/2}$. Here the role of the accumulation and destination buffers are swapped each pass.

This algorithm forms the basis for our implementation of the multi-pass algorithm. Instead of using an extra accumulation buffer, we accumulate using a single buffer. Writing to a texture which is also input to the computation is undefined [GGHM05], because the order of computation is unknown, i.e., you do not know which pixels are

computed first. Nevertheless, our empirical tests on the NVIDIA GeForce 7800 GT show that writing to the same buffer works as long as the input and output texels are at the exact same position. Utilizing this eliminates the need for a separate accumulation buffer in our algorithm, thus significantly lessening memory requirements.

Fatahalian, Sugerman and Hanrahan [FSH04] presented a single-pass matrix multiplication algorithm that corresponds to viewing the matrix multiplication as a series of vector-vector inner products. Each output element is then computed as

$$(AB)_{i,j} = \sum_{k=1}^{n/2-1} \begin{bmatrix} a_{i,2k+1} & a_{i,2k+2} \\ a_{i+1,2k+1} & a_{i+1,2k+2} \end{bmatrix} \begin{bmatrix} b_{2k+1,j} & b_{2k+1,j+1} \\ b_{2k+2,j} & b_{2k+2,j+1} \end{bmatrix}. \quad (2)$$

The main difference from the multi-pass algorithm is that the for-loop is moved from the CPU to the GPU, eliminating the need for several passes. Our algorithm is implemented as a GPU program that runs once, where one 2×2 sub-matrix of the result matrix is computed for each pixel.

Jiang and Snir [JS05] reported the single-pass algorithm as faster than the multi-pass algorithm for all the hardware setups they benchmarked on. Nevertheless, they did not benchmark on the hardware used here, the NVIDIA GeForce 7800 GT and 8800 GTX. Since we are operating on new hardware setups, we have implemented both the single- and multi-pass algorithms.

4.2 Gauss-Jordan elimination

We have implemented Gauss-Jordan elimination with partial pivoting. Gauss-Jordan elimination fits the GPU programming model better than standard Gaussian elimination because only *half* the number of passes are needed. Because we have packed 2×2 sub-matrices into each pixel, we exchange rows of 2×2 elements. This optimization increases performance since we only need *half* the number of passes compared to exchanging single rows, but will possibly create larger numerical errors than standard partial pivoting.

Finding the largest element of our pivot candidates requires a measure for each candidate. We use the value of the diagonal-elements after forward substitution,

$$\begin{bmatrix} r & g \\ b & a \end{bmatrix} \xrightarrow{\text{Subst.}} \begin{bmatrix} r & g \\ 0 & a - \frac{b}{r}g \end{bmatrix}. \quad (3)$$

This gives us the diagonal-elements $q_{1,1} = r$ and $q_{2,2} = a - \frac{b}{r}g$, where we compute

$$k = \frac{q_{1,1} \cdot q_{2,2}}{q_{1,1} + q_{2,2}} \quad (4)$$

similarly to the harmonic mean. We have experimentally found k to be a good measure for our application.

Finding the pivot element is done using a multi-pass reduction shader (GPU program) that first computes k for each element, and then reduces the vector down to one element, the maximum. In addition to finding the largest element, we also need to find the corresponding coordinate. It is not trivial to compute both the maximum *and* its norm effectively in one shader on the GPU. The naïve approach of using if-tests is a possibly expensive task, as all processors in the same single instruction multiple data (SIMD) group have to execute the same instructions; if one of the processors branches differently from the others, all processors have to evaluate both sides of the branch. We can, however, rewrite the branches into implicit if-tests, e.g., `float(a == b)*result`. This gives us the maximum, as well as the correct coordinate. If two elements have identical norms, the largest coordinate is selected.

When the reduction is complete, we are left with the greatest coordinate of the largest norm. If the largest norm is sufficiently close to zero, the matrix is assumed to be singular or near-singular.

After we have located the pivot element we need to swap the top row with the pivot row, convert the leading element to a leading one, and reduce all elements above and below to zero. Since we are using 2×2 packing, we normalize two rows, and eliminate two columns. This is done in a ping-pong fashion reading from the previously computed values, writing to the destination buffer. The pivot row is normalized when it is written to the position of the top row. The top row is simultaneously written at the position of the pivot row, and eliminated. The rest of the matrix is then eliminated in the next pass. This process is repeated until the matrix is reduced to the identity in the left part of the matrix, with the solution to the

right. It should be noted that the algorithm easily can be extended to full pivoting as well.

4.3 PLU factorization

The Doolittle algorithm for computing the PLU factorization is a small alteration of Gaussian elimination. The pivoting order is used to construct P, the multipliers used in the elimination are stored to create the unit lower triangular matrix L, and the matrix resulting from pivoting and forward substitution is the upper triangular U. The algorithm is executed as follows:

1. Find the pivot element.
2. Calculate two rows of U from the pivot row. Render the top row at the position of the pivot row simultaneously, thus swapping the two rows.
3. Eliminate below the top row, using the normalized pivot row.

The pivoting strategy used is the same as described for Gauss-Jordan elimination, and the pivoting order is stored on the CPU to construct P. When using the 2×2 packing scheme, we have to calculate two rows of U simultaneously. We do not need to alter the top row in the 2×2 row, but we have to reduce the bottom row in the same fashion as shown in Eq. (3). The last step of the algorithm calculates the multipliers needed to eliminate the rest of the column, and reduces the lower right part of the matrix accordingly. This process is repeated until the whole matrix is factorized.

When the matrix is factorized, we have L and U stored on the GPU as one texture. After transferring back to the CPU, L is constructed by adding the lower part of the texture with the identity matrix, and U is simply the upper triangular part of the texture.

4.4 Tridiagonal Gaussian elimination

Tridiagonal systems of equations arise e.g., when solving PDEs and ODEs numerically. Solving these systems using a full matrix solver is highly inefficient, as most elements are known to be zero. We can exploit the structure of the matrix to provide an efficient tridiagonal solver. We store the non-zero diagonals and the right hand side of the system in the four color channels red, green, blue and alpha:

$$\left[\begin{array}{ccccc|c} g_1 & b_1 & 0 & 0 & 0 & a_1 \\ r_2 & g_2 & b_2 & 0 & 0 & a_2 \\ 0 & r_3 & g_3 & b_3 & 0 & a_3 \\ 0 & 0 & r_4 & g_4 & b_4 & a_4 \\ 0 & 0 & 0 & r_5 & g_5 & a_5 \end{array}\right] \rightarrow \left[\begin{array}{cccc|c} 0 & g_1 & b_1 & a_1 \\ r_2 & g_2 & b_2 & a_2 \\ r_3 & g_3 & b_3 & a_3 \\ r_4 & g_4 & b_4 & a_4 \\ r_5 & g_5 & 0 & a_5 \end{array}\right].$$

We perform $n-1$ passes where we forward substitute, thus eliminating r_{i+1} at pass i, followed by $n-1$ passes where we backward substitute, eliminating b_{n-i} in pass i.

This is, however, a highly serial computation, as only one row of the matrix is updated in each pass. In order to benefit from the parallel execution mode of the GPU, we solve many such systems in parallel. Our tridiagonal solver is created specifically to solve many tridiagonal systems of equations, such as those that arise in the semi-implicit alternating direction discretization of the shallow water equations [Cas90]. It is also possible to solve a single tridiagonal system in parallel (see e.g., [Sto75]), but this is not the focus of our approach. To solve many systems in parallel, we stack our systems beside each other so that system i is in column i of the texture. This allows us to solve up-to 8192 tridiagonal systems of up-to 8192 equations in parallel on the NVIDIA GeForce 8800 GTX, which is the maximum texture size.

5 Results

The algorithms presented in the previous section all have native MATLAB implementations as well. MATLAB uses ATLAS [WD98], LAPACK [DW99], and BLAS [LHKK79] routines for its numerical linear algebra algorithms [The06, Mol00]. The routines offered by these libraries are regarded as highly optimized, and the MATLAB interface to them is considered efficient as well [MY02]. It should be noted, however, that the GPU does not offer fully IEEE-754 compliant floating point (there are some anomalies with e.g. denormals), which might be discerning for some uses.

The following compares the time used by the native MATLAB algorithm and the GPU algorithm. The benchmark times are measured using MATLABs internal timing mechanism. Time spent packing and transferring data between the GPU and the CPU is not included, as these

are looked upon as constant startup costs. When we reuse data in multiple computations as described in Section 4 this cost will become insignificant. For single computations, however, the startup-cost will have a larger influence on the runtime, and should be included.

To measure how fast the GPU is, we have approximated a polynomial, $a + bn + cn^2 + dn^3$, to the measured runtimes using the method of weighted least squares. In our experiments, we found this to give good approximations to the data-points, even though we do not know the specific complexity of the native MATLAB implementation. By comparing the dominant factor, d, we can compute a more realistic speedup-factor than by only comparing peak measured GFLOPS. We have used this approach for all algorithms except the tridiagonal Gaussian elimination.

The algorithms have been benchmarked on two GPUs, an NVIDIA GeForce 7800 GT (G70), and an NVIDIA GeForce 8800 GTX (G80). The CPU used for the native MATLAB implementation is a 3 GHz Pentium IV system (P IV) with 2GB of RAM. The matrices being used for the benchmarks are random matrices, as neither condition number, sparsity or structure influences the runtime of the algorithm. Since the tested hardware only supports single precision, there are precision issues for poorly conditioned matrices.

5.1 Full matrix-matrix multiplication

Figure 2 shows the execution times of MATLAB and the presented toolbox. The GPU implementation is slower than the highly optimized CPU code for small matrices. This is because there is a larger overhead connected with starting computation on the GPU than on the CPU. For large matrices, however, we experience a speedup. We computed the coefficient, d_{CPU}, of our polynomial approximating the CPU runtime to be $8.36\text{e-}10$, while the single-pass coefficients for the G70 and G80 GPUs are $4.26\text{e-}10$ and $2.67\text{e-}11$. This gives us speedup factors of $1.96\times$ and $31.29\times$ for the G70 and G80 GPUs respectively. Using the same approach for the multi-pass algorithm, we get speedup factors of $3.04\times$ and $14.25\times$. These speedup factors are good approximations for matrices larger than $\sim 500 \times 500$.

5.2 Gauss-Jordan elimination

MATLAB implements Gauss-Jordan elimination as the function `rref()`. Benchmarking the GPU version against the native MATLAB implementation, however, gave a speedup of $170\times$ and $680\times$ for the G70 and G80 GPUs respectively. This indicates that the MATLAB version is sub-optimal. To give a more appropriate speedup factor, we have chosen to compare against the PLU factorization in MATLAB. However, Gauss-Jordan elimination is a far more computationally heavy operation than PLU factorization. PLU factorization is also a far less memory heavy operation. In effect, our speedup factors are highly modest.

Figure 2a shows the execution time of the *PLU factorization in MATLAB*, and *Gauss-Jordan elimination on the GPU*. By comparing the dominant coefficient of our approximating polynomial, we can estimate the speedup factor to be $1.07\times$ and $4.27\times$ for the G70 and G80 GPUs. This speedup seems to fit the sample points well for matrices larger than $\sim 1000 \times 1000$.

5.3 PLU factorization

Figure 2b shows both the measured times for the CPU and the GPU, as well as the least squares approximation of the sample points. The approximated speedup over MATLAB is $1.48\times$ for the G70 GPU, and $7.55\times$ for the G80. These speedup factors seem to be valid for matrices larger than $\sim 1000 \times 1000$.

5.4 Tridiagonal Gaussian elimination

Benchmarking the tridiagonal solver is done using sparse storage on both the CPU and GPU. On the CPU, the systems have to be solved sequentially in a for-loop, while the systems are solved with one call to the GPU. On the GPU, the time includes time spent transferring data between MATLAB and the GPU, as this algorithm does not use the 2×2 packing.

Figure 3a shows a plot of the execution time for MATLAB, and Figure 3b shows the execution time for the GPU. Figure 3c shows the computed speedup-factors. Notice that there is almost no speed-up gain by increasing the size of the systems, whilst increasing the number of systems solved in parallel yields massive speedups. The

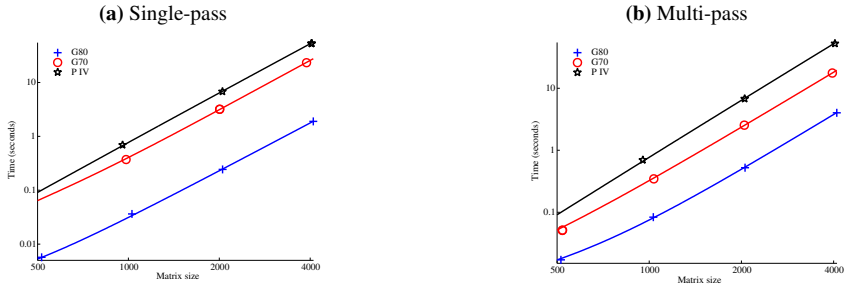

Figure 2: The weighted least squares approximation to the execution time for full matrix multiplication, and a subset of the measured execution times. Notice that the G70 GPU performs best using single-pass algorithm, and the G80 using the multi-pass algorithm.

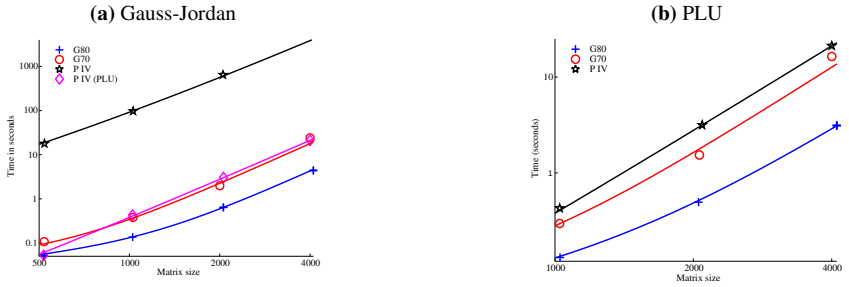

Figure 3: The weighted least squares approximation to the execution time for Gauss-Jordan elimination and PLU factorization with a subset of the measured execution times. Notice that Gauss-Jordan on the GPU is compared to PLU factorization on the CPU.

maximum observed speedup is $125\times$ for solving 4096 systems of 2048 equations in parallel.

5.5 Background computation

In addition to using the GPU alone, the presented interface also offers background computation. This background processing enables us to utilize both the CPU and the GPU simultaneously for maximum performance. Because they operate asynchronously, we can compute a crude approximation to a good load balance by timing the CPU and the GPU.

Using the load ratio to distribute work between the CPU and the GPU, we achieve background computation on the GPU that is virtually free. The load ratio is set to solve two systems on the CPU while one system is solved simultaneously on the GPU. Figure 5 shows the average time per system for the GPU, the CPU, and the total average time when using background processing. The largest cost connected with using the GPU is copying the data into memory allocated by MATLAB. Unfortunately this step is required, as MATLAB has internal memory management, disabling thread-safe use of its memory. Because we now include the overhead imposed by packing, transferring and unpacking each matrix in our timings, we experience less speedups than previously reported. This is the worst case scenario. Typical use should try to reuse data that resides on the GPU in multiple computations to hide the overhead of data-transfer. Nevertheless, the background processing decreases the overall execution time, and we can compute the speedup-factor to be $1.68\times$ over using just the CPU.

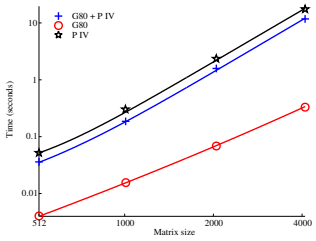

Figure 5: Average time spent computing the PLU factorization of 30 systems using both the CPU and the GPU. The graphs show time spent waiting for the result to be computed on the CPU, the GPU and total average time. Notice that utilizing the GPU is very inexpensive.

6 Conclusions and further research

We have presented an interface to the GPU from MATLAB, enabling the use of both the CPU and the GPU for maximum performance. In addition, four algorithms from numerical linear algebra have been presented for this interface, and shown to be up to $31\times$ faster than highly optimized CPU equivalents. Furthermore, a new pivoting strategy, and the use of 2×2 packing for Gauss-Jordan and PLU factorization has been presented.

The presented interface can be used to utilize both the CPU and the GPU simultaneously, and an automatically tuned load distributor will be of great interest in such a setting. The presented algorithms, matrix multiplication, Gauss-Jordan elimination and PLU factorization, are all implemented using 2×2 storage. The G80 GPU from NVIDIA implements scalar arithmetic, eliminating the need for this packing. Jiang and Snir [JS05] have presented an approach to automatic tuning of matrix-matrix multiplication on the GPU to the underlying hardware. Such an automatic tuning of the presented algorithms will further increase the usefulness the GPU toolbox for MATLAB. Utilizing APIs such as CUDA and CTM instead of OpenGL will probably also increase performance, as we do not need to rephrase the problem in terms of graphics. An optimized high-level mathematical interface to the GPU, such as described in this article, is not only interesting for MATLAB, but also other high-level languages, e.g., Python.

Acknowledgments

I would like to thank K.-A. Lie, T. R. Hagen, T. Dokken, J. Hjelmervik and J. Seland for their guidance and help with this document.

References

[Adv06] Advanced Micro Devices Inc. ATI CTM guide, 2006.

(a) CPU execution time (P IV) **(b)** GPU execution time (G80) **(c)** Speedup factor

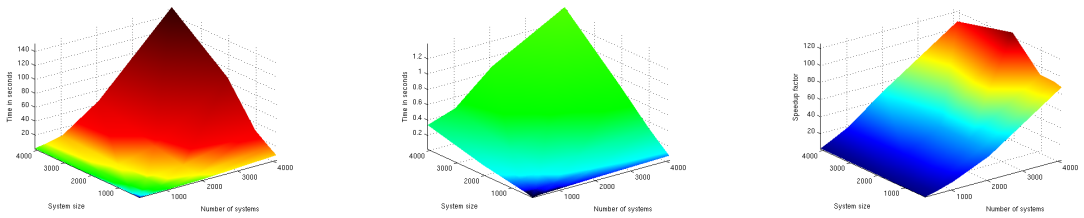

Figure 4: The measured execution time for tridiagonal Gaussian elimination using sparse storage. Notice that it is highly efficient to solve many systems in parallel on the GPU.

[Adv07] Advanced Micro Devices Inc. AMD delivers first stream processor with double precision floating point technology, 2007.

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

[BFH+04] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *SIGGRAPH '04*, pages 777–786. ACM Press, 2004.

[Bro07] André Rigland Brodtkorb. A MATLAB interface to the GPU. Master's thesis, University of Oslo, May 2007.

[Cas90] V. Casulli. Semi-implicit finite difference methods for the two-dimensional shallow water equations. *Journ. of Comp. Phys.*, 86:56–74, 1990.

[DW99] J. Dongarra and J. Wasniewski. High performance linear algebra package - lapack90. volume 5 of *Combinatorial Optimization, Advances in Randomized Parallel Comp.* Kluwer Academic Publishers, 1999.

[FSH04] K. Fatahalian, J. Sugerman, and P. Hanrahan. Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. In *Graph. Hardw.*, pages 133–137. ACM Press, 2004.

[GGHM05] N. Galoppo, N. K. Govindaraju, M. Henson, and D. Manocha. LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware. In *Supercomputing '05*, page 3. IEEE CS, 2005.

[GLGM06] N. K. Govindaraju, S. Larsen, J. Gray, and D. Manocha. A memory model for scientific algorithms on graphics processors. In *Supercomputing '06*, page 89. ACM Press, 2006.

[HCH03] J. D. Hall, N. A. Carr, and J. C. Hart. Cache and bandwidth aware matrix multiplication on the GPU, 2003.

[Int07] Intel Corporation. Intel math kernel library 9.1 – product brief, 2007.

[JS05] C. Jiang and M. Snir. Automatic tuning matrix multiplication performance on graphics hardware. In *Parallel Arch. and Compilation Techniques*, pages 185–196. IEEE CS, 2005.

[KÖ7] A. Källander. Multithreading in MEX files. Personal email communication, 2007.

[KW03] J. Krüger and R. 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. S. Larsen and D. McAllister. Fast matrix multiplies using graphics hardware. In *Supercomputing '01*, pages 55–55. ACM Press, 2001.

[MD06] M. D. McCool and B. D'Amora. Programming using RapidMind on the Cell BE. In *Supercomputing '06*, page 222. ACM Press, 2006.

[MDP+04] M. D. McCool, S. Du Toit, T. Popa, B. Chan, and K. Moule. Shader algebra. *ACM Trans. Graph.*, 23(3):787–795, 2004.

[Mic04] Microsoft Corporation. PCI Express FAQ for graphics, 2004.

[Mic07] Microsoft Corporation. Microsoft DirectX, 2007.

[Mol00] C. Moler. MATLAB news & notes - winter 2000, 2000.

[MY02] I. Mcleod and H. Yu. Timing comparisons of Mathematica, MATLAB, R, S-Plus, C & Fortran, 2002.

[NVI07a] NVIDIA Corporation. Accelerating MATLAB with CUDA using MEX files, 2007.

[NVI07b] NVIDIA Corporation. CUDA programming guide, 2007.

[OLG+07] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Comp. Graph. Forum*, 26(1):80–113, 2007.

[PSG06] M. Peercy, M. Segal, and D. Gerstmann. A performance-oriented data parallel virtual machine for gpus. In *SIGGRAPH '06*, page 184. ACM Press, 2006.

[Sto75] Harold S. Stone. Parallel tridiagonal equation solvers. *ACM Trans. Math. Softw.*, 1(4):289–307, 1975.

[SWND05] D. Shreiner, M. Woo, J. Neider, and T. Davis. *OpenGL Programming Guide*. Addison-Wesley, fifth edition, 2005.

[The06] The MathWorks. MATLAB software acknowledgements, 2006.

[WD98] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Supercomputing '98*, pages 1–27. IEEE CS, 1998.