LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware *

Nico Galoppo Naga K. Govindaraju Michael Henson Dinesh Manocha
University of North Carolina at Chapel Hill
{nico,naga,henson,dm}@cs.unc.edu

Abstract

We present a novel algorithm to solve dense linear systems using graphics processors (GPUs). We reduce matrix decomposition and row operations to a series of rasterization problems on the GPU. These include new techniques for streaming index pairs, swapping rows and columns and parallelizing the computation to utilize multiple vertex and fragment processors. We also use appropriate data representations to match the rasterization order and cache technology of graphics processors. We have implemented our algorithm on different GPUs and compared the performance with optimized CPU implementations. In particular, our implementation on a NVIDIA GeForce 7800 GPU outperforms a CPU-based ATLAS implementation. Moreover, our results show that our algorithm is cache and bandwidth efficient and scales well with the number of fragment processors within the GPU and the core GPU clock rate. We use our algorithm for fluid flow simulation and demonstrate that the commodity GPU is a useful co-processor for many scientific applications.

1 Introduction

Commodity graphics processors (GPUs) have recently been used for many applications beyond graphics, introducing the term GPGP: *general-purpose* computation on *graphics processors*. GPUs have been shown useful for scientific computations, including fluid flow simulation using the lattice Boltzman model [Fan et al. 2004], cloud dynamics simulation [Harris et al. 2003], finite-element simulation [Rumpf and Strzodka 2001], ice crystal growth [Kim and Lin 2003], and many other applications [Lastra et al. 2004]. GPU-based algorithms have also been proposed for iterative solvers for sparse linear systems [Bolz et al. 2003; Krüger and Westermann 2003; Göddeke 2005]. However, there is little research on using GPUs for general problems such as dense linear systems, eigendecomposition and singular value decomposition. In this paper, we focus on developing GPU-based general linear equation solvers and use them for scientific applications. Such systems are part of LINPACK benchmark, introduced by Dongarra et al. [2003] for the TOP500[1].

GPUs are optimized for fast rendering of geometric primitives for computer games and image generation. They are now available in almost every desktop, laptop, game console and are becoming part of handheld devices. GPUs have high memory bandwidth and more floating point units as compared to the CPU. For example, a current top of the line GPU such as the NVIDIA GeForce 6800 Ultra, available at a price of \$420, has a peak performance of 48 GFLOPS and a memory bandwidth of 35.2 GB/s, as compared to 12.8 GFLOPS and 6 GB/s, respectively, for a 3.2 GHz Pentium IV CPU. Current GPUs also support floating point arithmetic. Moreover, the GPUs performance has been growing at a faster rate than Moore's law, about 2-3 times a year.

GPUs consist of a fixed rendering pipeline and memory architecture. Furthermore, current GPUs do not support scatter operations and tend to have smaller cache sizes. It is feasible to implement conditionals, but they can be inefficient in practice. These limitations make it hard to directly map many of the well known numerical and scientific algorithms to GPUs. One of the major challenges in developing GPGP algorithms is to design appropriate data representations and develop techniques that fully utilize the graphics pipeline, multiple fragment processors and high memory bandwidth.

Main Results: We present a new algorithm to solve dense linear systems by reducing the problem to a series of rasterization problems on the GPU. We map the problem to the GPU architecture based on the observation that the fundamental operations in matrix decomposition are elementary row operations. Our specific contributions include:

1. Two-dimensional data representation that matches the two dimensional data layout on the GPU.
2. High utilization of the graphics pipeline and parallelization by rendering large uniform quadrilaterals.
3. Index pair streaming with texture mapping hardware.
4. Efficient row and column swapping by parallel data transfer.

These techniques and underlying representations make our overall algorithm cache and bandwidth efficient. We avoid potentially expensive context switch overhead by using appropriate data representations. We also propose a new technique to swap rows and columns on the GPU for efficient implementation of partial and full pivoting. We apply our algorithm to two direct linear system solvers on the GPU: LU decomposition and Gauss-Jordan elimination. We have compared the performance of our algorithm with the LAPACK blocked LU factorization algorithm, implemented in the optimized ATLAS library that makes full use of vectorized SSE instructions on the latest CPUs. We benchmarked our implementation on two generations of GPUs: the NVIDIA GeForce 6800 and the NVIDIA 7800, and compared them with the CPU-based implementation on a 3.4GHz Pentium IV with Hyper-Threading. The implementation of our GPU-based algorithm on the GeForce 6800 is on par with the CPU-based implementation for matrices of size 500×500 and higher. Our implementation on the NVIDIA GeForce 7800 GPU outperforms the ATLAS implementation significantly. Furthermore, we observe that GPU-based algorithms for LU decomposition are more efficient than Gauss-Jordan elimination, due to reduced number of memory operations.

We analyze the performance of our algorithm and measure the impact of clock rates and number of fragment processors. Our experiments indicate that the performance of our algorithm scales almost linearly with the number of fragment processors within the GPU. We also demonstrate that the performance scales well with the GPU

*Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. SC—05 November 12-18, 2005, Seattle, Washington, USA (c) 2005 ACM 1-59593-061-2/05/0011 \$5.00

[1]http://www.top500.org/lists/linpack.php

core clock rate, and follows the theoretical $O(n^3)$ complexity of matrix decomposition. Finally, we use our GPU-based algorithm for fluid flow simulation. We use the GPU as a co-processor for solving dense linear systems at the the microscopic level which frees up CPU cycles for macroscopic level computations.

Organization: The remainder of the paper is organized as follows. In Section 2, we give a brief overview of previous work on solving linear systems with stream architectures and scientific computation on GPUs. Section 3 gives an overview of the architectural features of GPUs and introduces our approach. We present our algorithms in Section 4 and highlight many of their features. In Section 5, we analyze the efficiency of the algorithms and Section 6 highlights their performance.

2 Related work

In this section, we give a brief overview of prior work on solving linear systems on stream architectures and scientific computation on GPUs.

2.1 Numerical Computation on Stream Architectures

Despite significant increases in the peak capability of high performance computing systems, application performance has been stagnating, mainly due to the imbalance between application requirements, memory performance and instruction operation rate [Oliker et al. 2004]. Many researchers have investigated use of stream architectures for linear algebra computations. Stream architectures exploit locality to increase the *arithmetic intensity* (i.e. the ratio of arithmetic to bandwidth) by expressing the application as a stream program. Examples include the the Merrimac supercomputer [Erez et al. 2004], which organizes the computation into streams and exploits the locality using a register hierarchy. This enables the stream architecture to reduce the memory bandwidth. Many linear algebra routines and scientific applications have been ported to Merrimac [Erez et al. 2004; Dally et al. 2003]. The Imagine [Ahn et al. 2004] is another SIMD stream processor, which was able to achieve 4.81 GFLOPS on QR decomposition. An extensive comparison of superscalar architectures like the SGI Altix and IBM Power3/4 systems with vector-parallel systems such as Cray X1 is given in [Oliker et al. 2004]. It shows that the latter general-purpose parallel systems exhibit good performance for scientific codes, as long as the algorithms can exploit data-parallellism. The RAW chip [Taylor et al. 2002] is based on a wire-efficient tiled architecture. It is a highly parallel and programmable general-purpose VLSI architecture and has been used for numerical signal-processing applications [Suh et al. 2003].

Our overall approach to solve dense linear systems on GPUs is in many ways similar to earlier approaches on porting linear algebra algorithms to stream architectures. Our goal is to exploit the parallelism and high memory bandwidth within the GPUs. However, the GPU architecture is different from current stream processors, tiled architectures or superscalar architectures.

2.2 Scientific Computations on GPUs

Recently, several researchers have used GPUs to perform scientific computations, including fluid flow simulation using the lattice Boltzman model [Fan et al. 2004], cloud dynamics simulation [Harris et al. 2003], finite-element simulations [Rumpf and Strzodka 2001], ice crystal growth [Kim and Lin 2003], etc. For an overview of recent work, we refer to [Lastra et al. 2004].

Our work is more related in spirit to previously proposed frameworks for linear algebra computations on the GPU. Several GPU-based algorithms for sparse matrix computations have been proposed [Bolz et al. 2003; Krüger and Westermann 2003]. These are all *iterative* methods, such as Jacobi iteration [Göddeke 2005] and conjugate gradient methods [Bolz et al. 2003]. The core operations of these algorithms are either local stencil operations or matrix-vector multiplication and vector dot products. These operations are relatively hard to implement efficiently on GPUs because they are reduction operations, i.e. a set of values is reduced to a smaller set of values. Reduction operations are typically implemented on GPUs by gathering values in multiple rasterization passes. Another drawback of these algorithms is checking for convergence, i.e. at each iteration, a slow data read-back from GPU to CPU is needed to check if the algorithm has converged.

It has been reported that matrix-matrix multiplication can be inefficient on current GPUs [Fatahalian et al. 2004]. This is mainly due to limited spatial and temporal locality in memory references and bandwidth limitations. Without cache-aware algorithms, matrix-matrix multiplication quickly puts the stress on GPU memory bandwidth. Many efficient matrix-matrix multiplication algorithms for CPU architectures have been investigated [Chatterjee et al. 1999; Thottethodi 1998]. The first GPU-based algorithm was proposed by Larsen and McAllister [2001]. It is a multiple pass algorithm, with a fairly decent cache access pattern. Although it predates programmable fragment processors, the authors reported performance equaling that of the CPU on 8-bit fixed-point data. Given the recent support for 16-bit floating point blending [NVIDIA Corporation 2004], this algorithm could be extended to 16-bit floating point data. Hall et al. [Hall et al. 2003] propose a cache-aware blocking algorithm for matrix multiplication on the GPUs. Their approach only requires a single rendering pass by using the vector capabilities of the hardware. The cache coherence of their algorithm is only slightly less than that of Larsen and McAllister's algorithm, but it requires 1.5 times less texture fetches overall. Most recently, both algorithms were compared and analyzed on NV40 and ATI X800XT hardware [Fatahalian et al. 2004]. It was demonstrated that the optimized ATLAS [Whaley et al. 2001] CPU implementation outperforms most GPU-based implementations. Only the ATI X800XT was able to outperform the CPU for matrix-matrix multiplication by 30%. Their algorithm is cache efficient, showing 80% peak cache hit rates, by explicitly applying classic blocking techniques. However, it was bounded by the inability to keep the computation units busy because of the limited bandwidth to the closest cache, which is several times smaller than L1 caches on current CPUs.

Recently, many high-level programming interfaces have been proposed for GPGP. These include BrookGPU [Buck et al. 2004] and Sh [McCool et al. 2004]. They use the programmable features of GPUs and hide the underlying aspects of graphics hardware. They are useful for prototyping of GPU based algorithms and have been used for a few scientific applications.

3 GPU Architectures

In this section, we briefly describe the different architectural features of GPUs and use them to design efficient dense linear solvers. Figure 1 shows a schematic overview of the architecture, pointing out the key components of our algorithm. The GPU is a pipelined architecture. The vertex and fragment processors are the programmable stages of the pipeline. Figure 2 shows a schematic of the internals of a fragment processor, also known as a pixel shader.

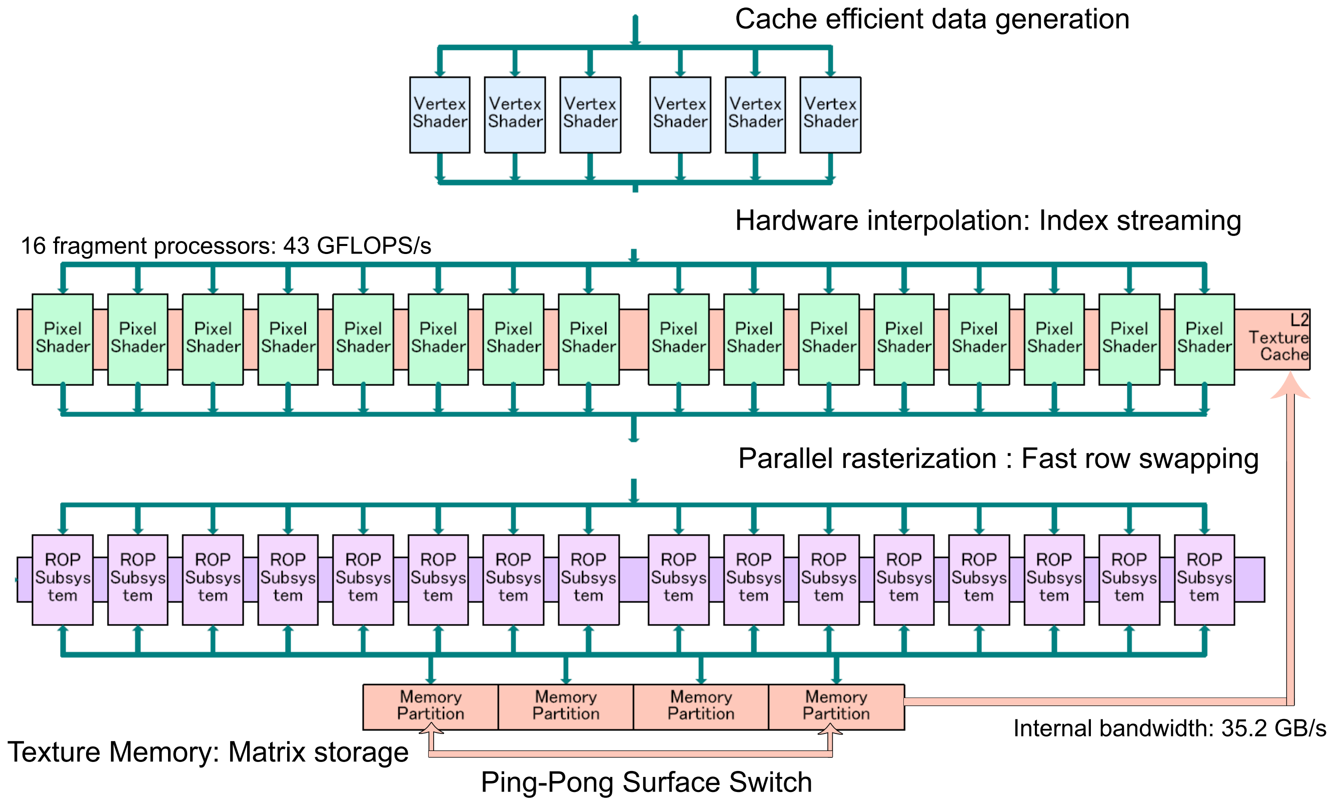
Cache efficient data generation
Vertex Shader
Hardware interpolation: Index streaming
16 fragment processors: 43 GFLOPS/s
Pixel Shader
L2 Texture Cache
Parallel rasterization : Fast row swapping
ROP Subsystem
Memory Partition
Internal bandwidth: 35.2 GB/s
Texture Memory: Matrix storage
Ping-Pong Surface Switch

Figure 1: Architecture of a commodity GPU: NVIDIA GeForce 6800 Ultra. This GPU has 6 programmable vertex processors and 16 programmable fragment processors, organized in a parallel structure, with a high bandwidth interface to video memory. We have shown how different components of our algorithm map to the vertex and fragment processors, including cache-efficient data representation, fast parallel row swapping, efficient index streaming with texture mapping hardware and fast surface switches. *(Illustration (c) 2004 Hiroshige Goto)*

3.1 Exploiting Parallelism on the GPUs

The number of fragment processing units governs the computational performance. For example, the commodity NVIDIA 6800 Ultra GPU has 16 parallel vector fragment processors; each processor can compute 4 components simultaneously (traditionally red, green, blue, alpha). It can thus perform 64 operations in one computational clock cycle and has an internal computational clock rate of 425 MHz. The fragment processor actually has 2 shader units, which allows it to execute two instructions in one cycle. In all non-texture situations, this dual-issue architecture results in a throughput of up to 8 scalar operations per cycle for each of the fragment processors, reaching a peak theoretical performance of approximately 48 GFLOPS per second. Note that many linear algebra routines and numerical computations are highly parallellizeable and can often be expressed in terms of vector operations [Erez et al. 2004; Oliker et al. 2004].

Graphics processors are highly optimized for rapidly transforming geometry into shaded pixels or fragments on the screen. The fragments generated during rendering of a geometric primitive are processed in parallel using fragment processors. Internally, the graphics primitives are processed in a *tiled* order (*internal tiled rasterization*), i.e. tiles of $m \times m$ adjacent fragments (say a constant m) are processed together. This way, the spatial coherence in geometric primitives can be utilized, while making the data access pattern independent of the orientation of the geometric primitives. On a NVIDIA 6800 Ultra GPU, each tile is processed by a quad pipeline of 4 fragment processors. Our algorithm fully utilizes each of those pipelines by rendering large uniform quadrilaterals.

3.2 Linear Algebra on CPUs and GPUs

The performance of optimized linear algebra algorithms such as LINPACK on CPUs is not only dependent upon the processor speeds but also on the size of the L1 and L2 caches [Dongarra et al. 2003]. A recent study on the performance of LINPACK based on the architectural evolution of two widely used processors, Pentium III and Pentium IV, indicates a decreasing trend of achievable computing power [Dongarra et al. 2003]. Moreover, the study indicates a substantial decrease in the performance due to the insufficient data buffering in L1 caches. As a result, the increasing CPU clock and the system bus speeds are not effectively utilized due to the substantial overhead of memory stalls. For example, a Pentium IV PC with a relatively high CPU clock of 2.5 GHz and a small L1 data cache size of 8KB only achieves a performance of 1.2 GFLOPS on the LINPACK benchmark. In comparison to its theoretical peak performance, the CPU is able to achieve only 23.5% system efficiency.

In contrast to the CPUs, GPUs have a relatively higher effective memory clock in comparison to its computational core clock. For example, a NVIDIA 6800 Ultra GPU has a 1.1 GHz memory clock and is nearly 3 times faster than its 425 MHz core clock to avoid memory stalls. In Section 4 we show that the data access pattern for elementary row operations used in Gauss-Jordan elimination or LU-decomposition algorithms exhibits high spatial coherence and maps well to the primitive rasterization operations on GPUs. Since GPUs are highly optimized for rendering geometric primitives, our algorithms usually achieve the high memory bandwidth available in GPUs. For example, a NVIDIA GeForce 6800 Ultra can achieve a peak memory bandwidth of 35.2 GBps, and is comparable to the peak memory bandwidth of a Cray X1 node [Erez et al. 2004]. Furthermore, the operations performed on each data element are independent of each other, and can be efficiently computed in parallel using the fragment processors on the GPUs.

3.3 Parallel Algorithms for Solving Dense Linear Systems

Many distributed and shared memory algorithms have been proposed to efficiently perform linear algebra computations. The parallel dense linear algebra algorithms have a total computational complexity of O(n^3) and require O(n^2) steps. In each step, O(n) parallel work is performed [Grama et al. 2003]. However, these parallel algorithms may not map well to current GPUs due to the following reasons:

- In order to minimize the communication costs, many of the algorithms employ blocking strategies. The performance of these algorithms on GPUs can depend on a number of factors such as the texture cache sizes, the block sizes used to fetch data from the texture memory, the texture access pattern of each fragment processor and the total number of fragment processors on the GPU. Furthermore, many of these parameters such as texture cache and block sizes are not disclosed by the GPU vendors, and it is difficult to optimize these algorithms for different GPUs. The graphics architects have designed these parameters to perform efficient rendering on the GPUs. Therefore, we use the rasterization engine to design a new cache-oblivious algorithm that maps well to the GPUs.
- Blocked LU decomposition algorithms such as the right-looking algorithm [Dongarra et al. 1998] perform LU decomposition efficiently on distributed-memory systems, but use matrix-matrix multiplications. However, implementation of matrix-matrix multiplication on the GPU can have performance issues [Fatahalian et al. 2004].
- Distributed memory and shared memory algorithms for Gauss elimination have also been proposed [McGinn and Shaw 2002], but these algorithms require *scatter* operations. Current GPUs do not support scattering data to arbitrary memory locations in order to avoid write-after-read hazards.

3.4 Texture Memory and Texture Cache Architecture

In graphics rendering, texture memory is used as lookup tables for quantities such as material properties, local surface orientation and precomputed light transfer functions. Texture coordinates define a mapping from image space or world space coordinates to texture memory locations, often referred to as *texels*. In linear algebra algorithms implemented on the GPU, texels are used to store array elements. Additionally, the two-dimensional nature of texture memory allows for a one-to-one mapping between texels and matrix entries.

The texture memory architecture is highly optimized for graphics applications; for a detailed discussion and analysis, we refer to [Hakura and Gupta 1997]. In order to increase computational efficiency, each fragment processor has a local L1 texture cache associated with it and multiple fragment processors share a L2 cache. These caches are static memories yielding fast data access and therefore reduce cache miss penalty. Furthermore, the latency of data accesses is reduced using texture prefetching and transferring 2-dimensional blocks of data from the texture memory to the video memory. Our algorithm uses data access patterns that match the spatial locality in block accesses and this leads to an improved performance. Figure 2 illustrates the memory interface of a fragment processor, in which we have highlighted the data paths that are primarily used by our algorithm.

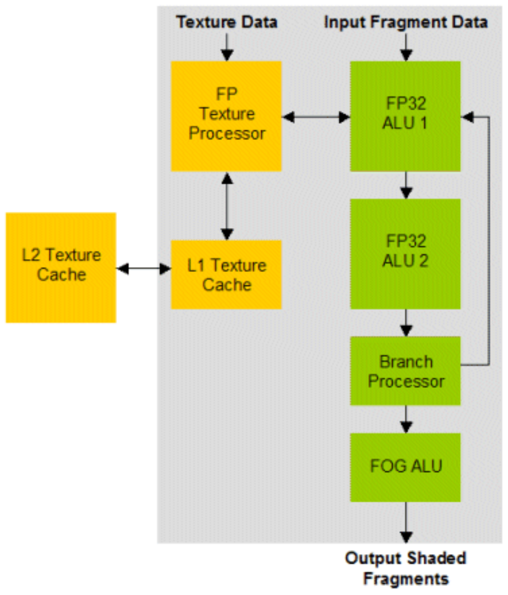

Figure 2: Schematic of a fragment processor with its texture caches on the commodity NVIDIA 6800 Ultra: each fragment processor has access to a local L1 texture cache and multiple fragment processors share a L2 cache. Our data representations make our algorithm cache efficient.

3.5 Mapping linear algebra algorithms to GPUs

In this section, we describe our data representation and different steps that are needed to run our algorithm on a GPU.

3.5.1 Data representation

We represent matrices using single-component, two-dimensional textures. As GPUs are optimized for two-dimensional textures, this setup can utilize the high memory bandwidth available in GPUs. Note that the image space coordinates are reversed in relation to matrix element indices: a texel $T(i, j)$ in texture space or a fragment $f(i, j)$ in image space corresponds to a matrix element a_{ji}.

Multiply-add (MAD) operations are the core arithmetic operation in matrix decomposition. Our algorithm performs MAD operations using single-component textures. As a preprocess to the algorithm, we copy the input matrix data to texture memory, which avoids further data transfer over the main bus. During the execution of the algorithm, only data that is stored locally to the GPU is being used. This way, we fully utilize the large bandwidth within the GPU.

3.5.2 Running a computational kernel

The fragment processors run basic computational kernels. The fragment programs are executed in parallel and produce output for all pixels that are being rendered. The output is written to the currently active memory *surface* of the framebuffer (double-buffered framebuffers have two surfaces: front and back).

The basic operations in matrix decomposition are repeated, element-wise *row operations*. Reduction operations are not required unless we perform pivoting. Unlike iterative matrix solvers, read-back operations are not required to check for convergence. Therefore, our algorithm uses the following techniques:

1. Bind the input matrix as two-dimensional textures, forming the input for the kernel.

2. Set the target surface for rendering. This surface forms the output of the kernel.

3. Activate a fragment program, i.e. set up the fragment pipeline to perform the kernel computation on every fragment (equivalent to a CPU **foreach** loop).

4. Render a single quadrilateral with multi-texturing enabled, sized to cover as many pixels as the resolution of the output matrix.

4 Direct dense linear solvers on the GPU

In this section, we present the details of two direct, i.e. non-iterative algorithms for solving linear systems of the type $AX = B$, where A is a large, dense, $N \times N$ matrix, and X and B are dense $N \times 1$ vectors: Gauss-Jordan elimination and LU decomposition with full pivoting, partial pivoting and without pivoting.

4.1 Gauss-Jordan elimination

Gauss-Jordan elimination is a multi-pass algorithm which computes the solution of the equations for one or more right-hand side vectors B, and also the matrix inverse A^{-1}. First, the matrix A is concatenated with B: $A' = [A|B]$; the first N columns are then successively reduced to columns of the identity matrix by performing row operations [Demmel 1997].

When the algorithm terminates, the solution vector X is located in the last column of the extended $N \times (N+1)$ matrix. The algorithm performs N elimination passes, updating $N(N-k+1)$ elements of the matrix at the k^{th} pass. On the CPU, this step is performed by using a nested loop, running over the rows and over the columns respectively.

On the GPU, the nested loop of the k^{th} elimination pass is performed in parallel by rendering a $N \times (N-k+1)$ quadrilateral that covers the elements that require updating. The sequential row-wise texture accesses in this algorithm fetch successive data values, and the data accesses performed by a tile of fragment processors during rasterization correspond to a 2D block in the texture memory. For example, a 2×2 tile requires a 2D block of 2×2 texture values. Due to large 2D block-based data fetches into the texture cache, the fragment processors in the tiles can readily fetch data values from the cache.

Because of the parallel nature of the GPU pipeline, one cannot read or write to the same surface in the same pass reliably. The hardware allows it, but the effect is defined to be unpredictable. Therefore, we use two surfaces, each representing the same matrix and we alternate the output to one of the two surfaces. In order to avoid context-switch overhead, we use only one double-buffered 32-bit off-screen buffer. We switch the role of front and back surface at each pass, being either source or target surface.

In the k^{th} iteration, the leftmost $N \times (k-1)$ columns of the matrix are reduced to the identity matrix and do not need further updates in the remainder of the algorithm. Using this observation, we reduce the number of fragments rasterized over the entire algorithm in half. To preserve data consistency during the ping-pong technique, we also rasterize the $(k-1)^{th}$ column during the k^{th} pass, such that column $(k-1)$ at the $(k-1)^{th}$ pass is propagated to the k^{th}pass. Algorithm 1 describes our GPU based algorithm for Gauss-Jordan elimination.

Algorithm 1 Gauss-Jordan elimination on the GPU

```
for k = 1 to N,
   // (1) normalize $k^{th}$ row
   glDrawBuffer('target surface');
   Bind 'source surface' as input texture;
   Load 'normalize' fragment program;
   Render quad covering $k^{th}$ row;
   // (2) copy back to source surface
   glDrawBuffer('source surface');
   Bind 'target surface' as input texture;
   Load 'copy' fragment program;
   Render quad covering $k^{th}$ row;

   // (3) eliminate $k^{th}$ column
   glDrawBuffer('target surface');
   Bind 'source surface' as texture;
   Load 'rowop' fragment program;
   Render quad from (k-1,k-1) to (N,N);

   Swap 'target' and 'source' surfaces;
endfor
```

4.2 LU decomposition without pivoting

If the inverse of the matrix A is not desired, the Gauss-Jordan method is actually less efficient than the alternate method for dense linear systems: LU decomposition. Algorithm 2 describes the algorithm without pivoting in vectorized notation.

Algorithm 2 Vectorized LU decomposition

```
for k = 1 to N-1
   A(k+1:N, k) = A(k+1:N, k) / A(k,k);
   A(k+1:N, k+1:N) = A(k+1:N, k+1:N)
      - A(k+1:N, k) * A(k, k+1:N);
endfor
```

Note that, during the k^{th} pass, only the lower right $(N-k) \times (N-k+1)$ part of the matrix is updated. As with Gauss-Jordan elimination, the memory references in this algorithm are highly coherent, independent of row-wise or column-wise rasterization. It is well known that the computational complexity of LU decomposition is 3 times less than that of Gauss-Jordan elimination. Later we show in Section 6 that 1.5 times fewer data elements are updated, resulting in a significant drop in texture fetches.

The pseudo-code for a GPU implementation is described in Algorithm 3. In this algorithm, we also use the ping-pong technique that was introduced in Section 4.1. Contrary to the Gauss-Jordan GPU algorithm, we need not update the $(k-1)^{th}$ row at step k. This is due to Lemma 4.1.

Lemma 4.1 *After the k^{th} elimination pass, on both surfaces of the off-screen buffer, column k and row k are up to date.*

Proof The proof is based on two observations in Algorithm 3:

1. Step (3) copies the updated column k back to the source surface.

2. Row k is not updated in pass k.

After step k of the algorithm, column k and row k remain unchanged for the rest of the algorithm. The two previous observations guarantee that source and target surface contain correct values, which means that they contain the values of the end result. □

Algorithm 3 LU decomposition on the GPU

```
for k = 1 to N-1
    // (1) Copy row k to destination surface
    glDrawBuffer('target surface');
    Bind 'source surface' as texture;
    Load 'copy' fragment program;
    Render quad covering $k^{th}$ row;
    // (2) Normalize column k
    Load 'normalize' fragment program;
    Render quad covering $k^{th}$ column;
    // (3) Copy updated column k back to
    //     source surface
    glDrawBuffer('source surface');
    Bind 'target surface' as texture;
    Load 'copy' fragment program;
    Render quad covering $k^{th}$ column;
    // (4) Update lower right
    //     (N-k+1) × (N-k+1) submatrix
    glDrawBuffer('target surface');
    Bind 'source surface' as texture;
    Load 'rowop' fragment program;
    Render quad from (k+1,k+1) to (N,N);

    Swap 'target' and 'source' surfaces;
endfor
```

4.3 LU decomposition with partial pivoting

The main issues that arise in handling pivoting are (1) the search for the maximum element in the k^{th} column, and (2) swapping the k^{th} row with the row that contains the maximum element.

1. **Find pivot in column** k

   In order to know which rows to swap, the CPU program that drives the GPU operations needs to compute the index i_{max} of the row that contains the pivot element. We use the GPU to compute i_{max} and then read it back to the CPU.

   We run a sequential fragment program "maxop" in *a single* fragment at texel location $(k+1,k)$. This fragment program loops over the column elements (i,k) where $k \leq i \leq N$ and writes the *index* $i_{\max}$ of the row with the maximum element to the output texel. Then, we perform a *read-back* operation of texel $(k+1,k)$ from the "target" surface to the CPU. This introduces a stall in the pipeline, as the readback operation needs to wait until the "maxop" fragment program has safely written its result to the "target" surface.

2. **Row swapping**

   Traditionally, the most efficient way to swap rows is to swap pointers to the start of the row instead of actually moving the data in memory. On the other hand, on GPUs, this pointer-based technique is very inefficient because it requires dependent texture fetches, i.e. texture fetches in which the fetch location depends on a prior texture fetch (the pointer fetch). Dependent texture fetches stall the GPU pipeline, as the actual data fetch instruction has to wait for the pointer fetch. Furthermore, the data fetches using dependent accesses are usually not 2D block coherent and can result in significant cache evictions. On the other hand, texture fetches and arithmetic instructions that do not depend on a previous fetch can run at full speed due to the fragment processor architecture (Section 3).

   Our procedure for row swapping does not require CPU read-back, and does not involve GPU pipeline stalls due to dependent reads. We swap rows by moving the corresponding data in the texture memory. During this step, our algorithm maintains the coherence in terms of texture memory accesses. The swap operation is performed efficiently by utilizing the high bandwidth and parallel capabilities of the GPU architecture: we generate the data stream by rendering the two rows to be swapped, running a simple "copy" fragment program that takes the opposite row data as input texture. This way, the two rows end up swapped in the "target" surface. Then, we simply copy back the two rows to the "source" surface by swapping "source" and "target" surface, rendering the two rows with the same "copy" program but with aligned row data as input. Finally, we swap "source" and "target" surfaces again, such that the normal LU decomposition algorithm can proceed as before. Note that the data never crosses over the bus to the CPU in this part of the process, all data is moved within high-bandwidth texture memory. Also, the amount of data swapped ($4N$ bytes) is relatively small and requires little time in comparison to the computation time in a pass ($O(N^2)$ operations and memory fetches).

4.4 LU decomposition with full pivoting

Our full pivoting algorithm is slightly more involved than our partial pivoting algorithm. Specifically, the pivot element has to be searched in the entire remaining lower right submatrix and one has to find both row index i_{max} and column index j_{max} of the pivot element. As our algorithm uses single component textures, we store the intermediate results such as indices and temporary maximum values at separate locations in the texture memory. The search algorithm for the maximum value in the lower right submatrix during iteration k proceeds as follows:

1. **Collect maximum value and maximum index across each column**

   Two separate fragment programs are run over rows $k+1$ and $k+2$, respectively. The first fragment program loops over the column elements below the fragment, storing the index i^l_{max} in each texel $(k+1,l)$ with $k \leq l \leq N$. The second fragment program loops over the same elements for each generated texel, storing the maximum value across each column v^l_{max} in texels $(k+2,l)$ with $k \leq l \leq N$.

2. **Find index pair of pivot element**

   Two more fragment programs are run on a single fragment by drawing two points. First, a fragment program loops over texture memory locations $(k+2,l)$ with $k \leq l \leq N$, writing out the index j_{max} of the pivot element. Then, just as for partial pivoting, a read-back operation is performed to transfer j_{max} to the CPU. Then, j_{max} is used to perform an extra read-back operation of the texel at $(k+1, j_{max})$ to transfer i_{max} to the CPU.

Once i_{max} and j_{max} have been computed, appropriate rows and columns are swapped as described in section 4.3.

4.5 Index pair streaming

In this section, we present an efficient technique to generate the index pairs (i,k) and (k,j) that are referenced in the row operations (Algorithm 2), for each element a_{ij}. This is a fundamental operation of our algorithm.

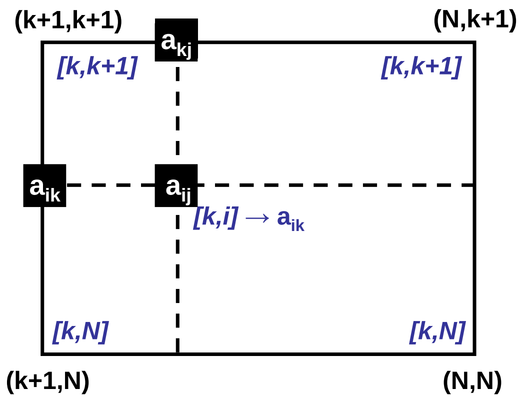

Figure 3: Texture coordinates are being interpolated by the texture mapping hardware. By assigning these texture coordinates (in square brackets) to the vertices (denoted by parentheses), the correct element a_{ik} is referenced.

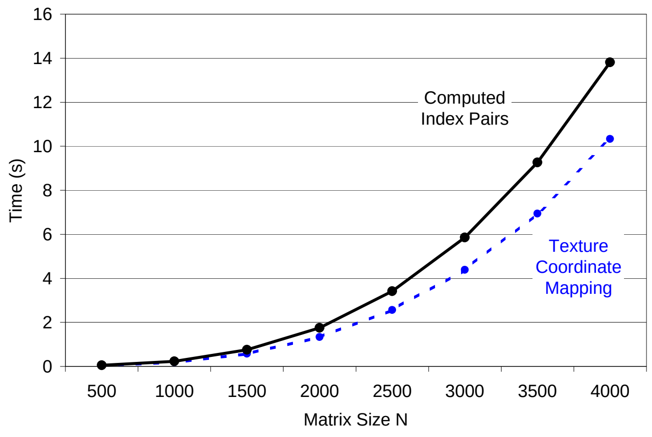

Figure 4: Utilizing the specialized texture mapping hardware on GPUs yields a 25% performance improvement to our algorithm (NVIDIA 6800 Ultra).

GPUs have specialized texture mapping hardware that is designed to map vertex coordinates in image space to the texel coordinates in texture memory, also known as texture coordinates. Each vertex of the rasterized primitive is assigned a texture coordinate; the texture coordinates of a fragment are automatically generated by the hardware by performing bilinear interpolation of the texture coordinates at the vertices. In graphics applications, this feature is commonly used to look up and interpolate material properties such as color and local surface orientation between vertices.

We utilize the texture mapping hardware to efficiently pass the index pairs (i,k) and (k,j) to the fragment processor. Figure 3 illustrates that, by assigning texture coordinates $[k,k+1],[k,N],[k,N],[k,k+1]$ to quad vertices $(k+1,k+1),(k+1,N),(N,N),(N,k+1)$ in iteration k, the texture mapping hardware automatically interpolates and passes the correct index pair (i,k) to the fragment program. Similarly, we assign a second set of texture coordinates $[k+1,k],[k+1,k],[N,k],[N,k]$ to the vertices to generate the index pair (k,j).

Our approach eliminates the need to compute the index pairs in the fragment program, and reduces the number of instructions in the *rowop* fragment program from 8 to 4. Due to the use of texture coordinates, the graphics architectures can prefetch data values from the texture memory and can result in an improved performance. Figure 4 shows that this optimization yields a 25% performance improvement in our algorithm.

	# Frags	B/frag	Total bytes
Gauss-Jordan			
Normalization	$\frac{n(n+1)}{2}$	12	$6n(n+1)$
Copyback	$\frac{n(n+1)}{2}$	8	$4n(n+1)$
Elimination	$n\frac{n(n+1)}{2}$ $\approx \frac{n^3}{2}$	16	$8n^2(n+1)$
Total bytes Gauss-Jordan			$\approx 8n^3$
LU decomposition			
Copy	$\frac{n(n+1)}{2}$	8	$4n(n+1)$
Normalization	$\frac{n(n+1)}{2}$	12	$6n(n+1)$
Copyback	$\frac{n(n+1)}{2}$	8	$4n(n+1)$
Elimination	$\frac{(n-1)n(2n-1)}{6}$ $\approx \frac{n^3}{3}$	16	$\frac{8}{3}(n-1)n(2n-1)$ $\approx 16\frac{n^3}{3}$
Total bytes LU Decomposition			$\approx \frac{16}{3}n^3$

Table 1: Memory operation counts for our GPU based algorithms, algorithms 1 and 3. The second column shows the bytes per fragment.

5 Analysis

In this section, we analyze the Gauss-Jordan elimination and LU decomposition algorithms in terms of memory bandwidth and instruction count.

5.1 Memory bandwidth

The external bandwidth is the rate at which data may be transferred between the GPU and the main system memory. New bus technologies, such as PCI-express make this overhead relatively small for reasonably sized matrices (1000×1000 and up). The use of static vertex buffer objects [Hammerstone et al. 2003] also alleviates external bandwidth requirements for transferring geometry.

The internal bandwidth is the rate at which the GPU may read and write from its own internal memory, and depends on the data access pattern and the texture caches. For Algorithm 1, step (1) requires two texture fetches and writes one value per fragment. Step (2) requires one fetch and one write, step (3) requires three texture fetches and one write per fragment. The number of memory references per processed fragment are the same for Algorithm 3, but the number of processed fragments differs. In our algorithms, every memory operation transfers 4 bytes of data.

Table 1 summarizes the number of memory operations required for each algorithm. It clearly justifies why LU decomposition should be preferred over Gauss-Jordan elimination for a GPU-based implementation. Independent of computational complexity, based on number of fragments processed, LU decomposition is preferred because it requires 1.5 times less fragments to be processed, giving it a significant advantage over Gauss-Jordan elimination in terms of internal bandwidth requirements.

5.2 Computational complexity

The first shader unit in the fragment processor (Section 3.1) allows the NV40 GPU to perform a texture fetch in parallel with an arithmetic operation. Compared to the number of memory instructions in our fragment programs, the number of arithmetic instructions is low, i.e. 3 texture fetches vs. 1 multiply-add operation. Therefore, we only include arithmetic instructions in this analysis. The *rowop* fragment program in Algorithms 1 and 3 requires exactly

	# Frags	Instr/frag	Total ops
Gauss-Jordan			
Normalization	$\frac{n(n+1)}{2}$	1	$\frac{n(n+1)}{2}$
Elimination	$n\frac{n(n+1)}{2}$	1	$n\frac{n(n+1)}{2}$
	$\approx \frac{n^3}{2}$		$\approx \frac{n^3}{2}$
Total instructions Gauss-Jordan			$\approx \frac{n^3}{2}$
LU decomposition			
Normalization	$\frac{n(n+1)}{2}$	1	$\frac{n(n+1)}{2}$
Elimination	$\frac{(n-1)n(2n-1)}{6}$	1	$\frac{(n-1)n(2n-1)}{6}$
	$\approx \frac{n^3}{3}$		$\approx \frac{n^3}{3}$
Total instructions LU decomposition			$\approx \frac{n^3}{3}$

Table 2: Arithmetic instruction counts for our GPU based algorithms, Algorithms 1 and 3.

one `mad` instruction, whereas the *divide* fragment program requires exactly one `div` instruction. The *copy* fragment program doesn't perform arithmetic operations. Table 2 summarizes the computational complexity. Using GPUBench [2004], we have benchmarked the GeForce 6800 to be able to perform 5.2 GInstructions/s for both `mad` and `div`. A single `mad` instruction performs 2 floating point operations, therefore a GPU implementation in the best case can perform 32 floating point operations in one clock cycle. However, this ignores the time taken for reading data from textures. Therefore, the peak computation rate is at most 12 GFLOPS/s (assuming the data values for MAD instruction are in the registers). In practice, due to the latency in data fetches, the computation rate is lower than 12 GFLOPS.

5.3 Cache efficiency

In our algorithm, the updates of neighboring elements in both dimensions are cache coherent because we make use of two features of the GPU architecture:

- Internal tiled rasterization (Section 3.1).
- Two-dimensional block-coherent data accesses (Section 3.4).

The application is not aware of the internal tile or cache size, i.e. our algorithm is cache oblivious. This is the main reason GPU based dense matrix solvers are more efficient than GPU based matrix-matrix multiplication algorithms, where cache efficiency has to be handled explicitly by using a blocked algorithm.

5.4 Measuring peak performance

We have measured the peak performance achieved by our algorithms by restricting data access to a single location in texture memory. By accessing a single data value, we ensure peak texture cache hit rate. We have also measured the peak computational performance by removing the texture fetches and replacing the mathematical instruction operands by constants. Note that, to simulate 3 cache hits, three *different* constant texels are sampled, otherwise the fragment processor compiler optimizes the code to a single texture fetch.

6 Applications and Results

In this section, we analyze the performance of our linear algebra algorithms on different matrices. We present experimental studies to identify the different factors that affect the performance and compare the computation- and memory-efficiency against peak theoretical efficiency. We also use our algorithm for fluid flow simulation and discuss the implications for future GPU architectures.

6.1 Performance

We have benchmarked the algorithms on a high-end 3.4 GHz Pentium IV CPU with Hyper-Threading, 16 KB L1 data cache and 1024 KB L2 cache.The GPU algorithms were benchmarked on two commodity GPUs:

- **NVIDIA GeForce 6800 GT:**
  - 12 parallel fragment processors (3 quad pipelines)
  - 350 MHz core clock rate
  - 256 Mb texture memory, 900 MHz memory clock rate
- **NVIDIA GeForce 6800 Ultra:**
  - 16 parallel fragment processors (4 quad pipelines)
  - 425 MHz core clock rate
  - 256 Mb texture memory, 1.1 GHz memory clock rate
- **NVIDIA GeForce 6800 Ultra:**
  - 24 parallel fragment processors (6 quad pipelines)
  - 430 MHz core clock rate
  - 256 Mb texture memory, 1.2 GHz memory clock rate

We have compared the performance of our algorithm (as described in Algorithm 3) against the LAPACK algorithms in the ATLAS library. The ATLAS implementations are highly optimized, and use the SSE instructions to improve the performance. We have used the ATLAS library in MATLAB. These routines in MATLAB have been reported to be highly optimized, and are comparable with the fastest benchmarks on our platform [McLeod and Yu 2002]. The linear algebra matrix A is generated with random values and conforms the LINPACK Benchmark [Dongarra et al. 2003], and ensure that MATLAB does not use other optimized solvers that exploit symmetry or sparsity. The timings exclude external bandwidth cost of transferring the matrix to the GPU. In Section 6.2, it is shown that this cost is negligible in our experiments.

In our benchmarks, we have used different matrix sizes. However, the maximum matrix size is 4096×4096, as this is the maximum texture size current GPUs support.

LU decomposition without pivoting. We have compared the performance of our algorithm without pivoting and the platform-optimized LAPACK `getrf()` implementation in ATLAS using different matrix sizes. `getrf()` uses a blocked algorithm and is considered to be one of the fastest existing implementations of LU decomposition with partial pivoting on the Intel CPU platform. Therefore, we compare the performance of our algorithm without pivoting against this optimized non-pivoting CPU implementation. Fig. 5 indicates that the performance of our algorithm without pivoting is comparable to the ATLAS implementation with pivoting. The data also confirms that our algorithm conforms to the asymptotic $O(n^3)$ complexity of Gauss-Jordan elimination and LU decomposition, and suggests that the GPU algorithm is compute- and memory-efficient.

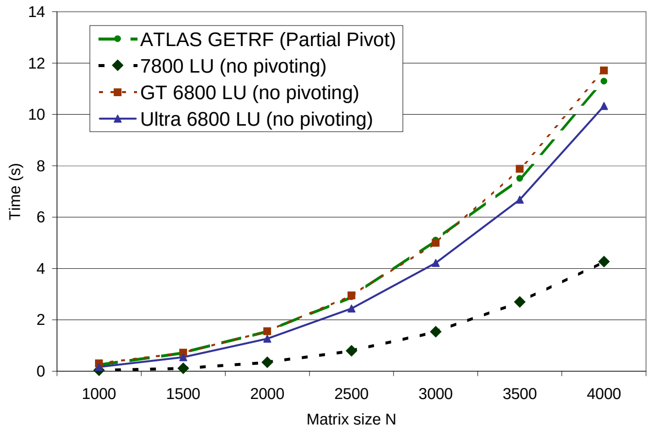

Figure 5: Average running time of LU-matrix decomposition without pivoting as a function of matrix size: This graph highlights the performance obtained by our algorithm on three commodity GPUs and ATLAS `getrf()` on a high-end 3.4GHz Pentium IV CPU. Note that the performance of our algorithm *without* pivoting is comparable to the optimized ATLAS `getrf()` implementation of LU decomposition with pivoting. Furthermore, the CPU and GPU algorithms indicate an asymptotic $O(n^3)$ algorithm complexity.

Partial Pivoting. In Figure 6, we have compared the performance of our algorithm with partial pivoting with the highly optimized LAPACK `getrf()` implementation with partial pivoting. We observe that our implementation is 35% faster for matrix size 3500, on the NVIDIA 7800 GPU. Moreover, our algorithm follows ATLAS' execution time growth rate closely.

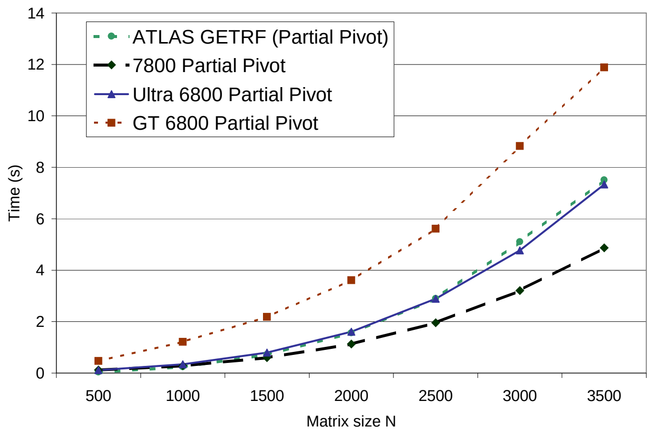

Figure 6: Average running time of a LU-matrix decomposition with partial pivoting as a function of the matrix size. Our algorithm is compared to ATLAS `getrf()` for three commodity GPUs. Note that our implementation is 35% faster for matrix size 3500 on the fastest GPU, and that our algorithm follows ATLAS' execution time growth rate closely.

Full Pivoting. For full pivoting, we use the LAPACK `getc2()` auxiliary algorithm. This algorithm is not implemented in ATLAS; instead, we have used the implementation that is available in the optimized SSE-enabled Intel Math Kernel Library. Figure 7 shows that our implementation is an order of magnitude faster. In this case, the CPU implementation may not be fully optimized. We conjecture that the CPU's performance degrades due to frequent row and column swaps and cache misses. On the CPU, these operations are implemented by pointer swaps, causing increased cache thrashing because the data coherence is not preserved. Our GPU-based algorithm does not have this problem, as the actual row and column data is moved within the memory.

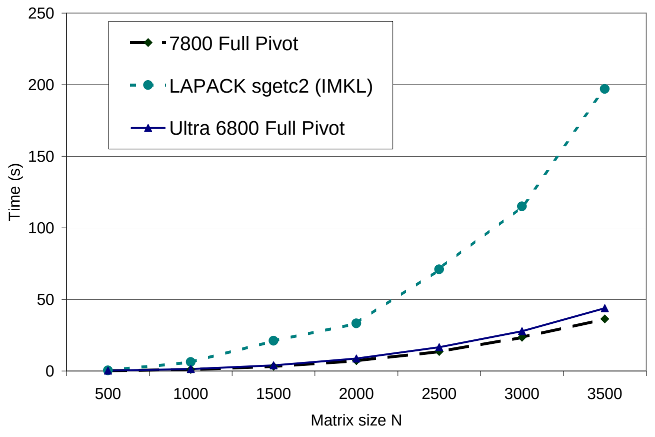

Figure 7: Average running time of one matrix decomposition with full pivoting in function of matrix size. Our algorithm is an order of magnitude faster than the implementation of LAPACK `getc2()` in the Intel Math Kernel Library (IMKL).

Overhead of pivoting. In Figure 8, we show the overhead of pivoting in our GPU-based algorithm. As the data show, the overhead of partial pivoting and full pivoting is bounded by approximately a factor of 2, respectively 3, which is to be expected from the conditionals that were introduced to compute the pivot element. We believe that this overhead is reasonable given the well-known difficulties that streaming architectures such as GPUs have with conditional operations. Our efficient row and column swapping techniques compensate these drawbacks partially: in case of frequent swaps, moving the data does not fragment the location of adjacent matrix elements as much as pointer swapping, which in turn avoids cache thrashing.

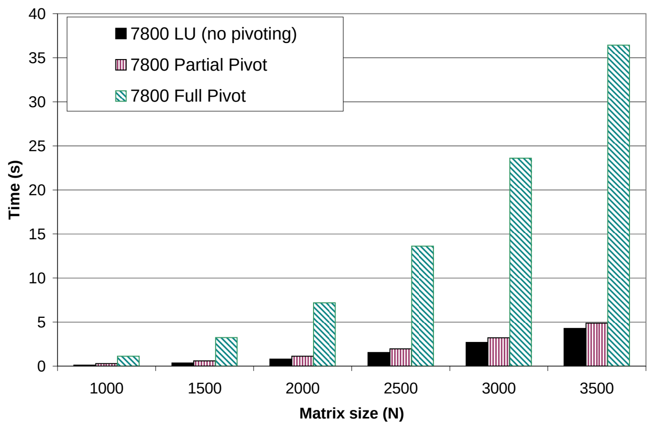

Figure 8: Comparison of three instances of our GPU-based algorithm (without pivoting, partial pivoting and full pivoting), which shows the overhead incurred by pivoting.

Influence of fragment processor parameters. Next, we measure the influence of the number of parallel fragment processors and the fragment processor core clock rate. Using publicly available tools, we were able to disable one quad pipeline at a time (4 fragment processors). We measure the performance of our algorithm without pivoting for 4, 8, 12 and 16 pipelines enabled on the 6800 Ultra GPU. The trend in Figure 9 shows that the performance of our algorithm improves almost linearly as the number of parallel pipelines is increased. We also measure the performance with all fragment pipelines enabled, with varying core clock rates of 200 – 425 MHz in steps of 50 MHz. The graph in Figure 10 indicates that the performance also increases almost linearly with the rate.

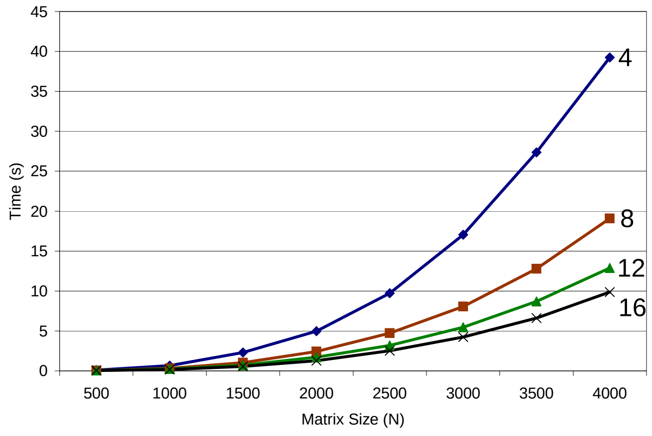

Figure 9: Average running time of LU matrix decomposition without pivoting, for different number of parallel fragment processors disabled in a NVIDIA 6800 Ultra GPU, in function of matrix size. The number of fragment processors associated with each data curve is shown on the right. The data shows almost linear improvement in the performance of our algorithms as the number of fragment processors is increased.

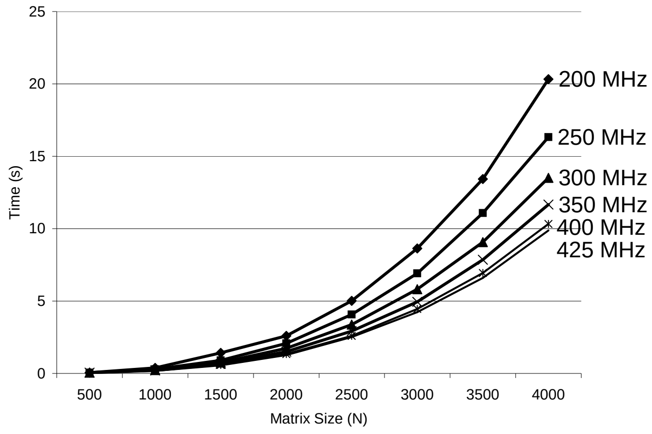

Figure 10: Average running time of LU matrix decomposition without pivoting, for different core clock rate settings of a NVIDIA 6800 Ultra GPU, in function of matrix size. The rate associated with each data curve is shown on the right. The performance of our algorithm improves well with increasing clock rate.

Bandwidth Usage. We have estimated bandwidth usage, based on the total amount of data transferred in LU decomposition (Table 1). Our measurements are estimates because there is some overhead introduced for binding textures as input (Section 6.3). This overhead may be significant for small size matrices. Figure 11 clearly shows that our algorithm's bandwidth usage is most efficient for large matrices. On the other hand, our experimental data suggests that our algorithm makes efficient use of the internal memory bandwidth, as we observe a sustained bandwidth usage of ~ 33 GB/s, which is close to the NVIDIA Ultra 6800's peak internal bandwidth limit of 35.2 GB/s.

Cache efficiency. Using the approach described in Section 5.4, we measured the cache efficiency rate of the LU algorithm, as a ratio of performance versus peak cache performance with varying matrix size. The results in Figure 12 suggest that our algorithm is cache-efficient. Moreover, our algorithm needs no knowledge of the cache parameters of the GPU, i.e. it is cache-oblivious.

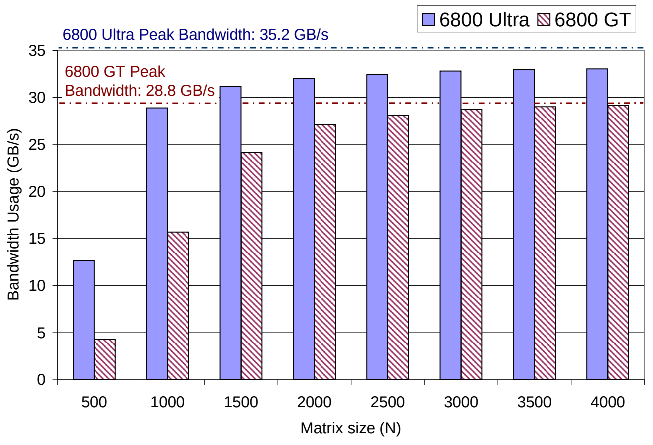

Figure 11: Bandwidth usage of our LU decomposition algorithm (without pivoting), in function of matrix size. The experiments were performed with two commodity GPUs: NVIDIA GeForce 6800 GT and NVIDIA GeForce 6800 Ultra. The dashed lines indicate peak internal bandwidth limits of both cards. The data indicate that our algorithm is bandwidth-efficient.

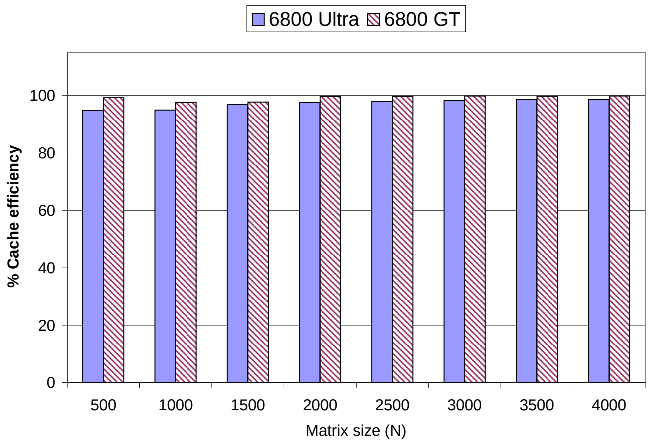

Figure 12: Cache efficiency rate of our LU decomposition algorithm (without pivoting), in function of matrix size, for two commodity GPUs.

6.2 Application to bubbly fluid simulation

We have used our GPU-based solver for bubbly fluid simulation. A common approach to the simulation of *bubbly fluid* flows consists of a divide-and-conquer algorithm [Russo and Smereka 1996a; Russo and Smereka 1996b; Mitran 2000]. The algorithm proceeds at macroscopic and microscopic level separately, with the results of both levels being communicated between each other. We give an overview of the equations of a two-dimensional simulation, but they naturally extend to three dimensions.

At the *macroscopic level*, the densities on a coarse $K \times L$ grid are advected by upward differencing of the (simplified) advection equation $n_t(i,j) + \nabla(n(i,j)u_{ij}(n)) = 0$, where $n(i,j)$ is the density at cell (i,j) and u is a vector field defined on the grid, which is a function of n. At the *microscopic level*, within each cell (i,j), a discretization of the integral equation that governs the dynamics of the surface tension of the *inclusions* or *bubbles* can be expressed as a dense linear system of equations of the form

$$A^t_{ij} q = b, \quad (1)$$

where q^t is a $N \times 1$ sized vector, representing discretized potentials on the surface of the bubbles. Solving for q^t allows to compute the average fluid velocity of cell (i, j) at time instant t: $u^t_{ij} = Bq$, where tensor B is a $2 \times N$ matrix. Matrix A^t_{ij} is of size $N \times N$ and holds information about the statistical distribution for the bubble configuration, given by the first moments of the fluid densities at time t.

Interestingly, the matrix A is guaranteed to be *diagonally dominant*, meaning that, for solving Equation (1), our GPU algorithm without pivoting is sufficient and numerically stable within the limits of single precision. Also note that typically $N = mk \approx 2048$, where m is the number of inclusions in a cell and k is the number of sample points on the surface of an inclusions. This means that A fits well in the GPU texture memory and lies within the size range that our GPU algorithm is at its peak performance, as was shown in Figure 11. Additionally, it can be shown that, because of the nature of the kernel of the integral equation in the case of dilute bubbly fluids, i.e. when the bubbles are well separated, it is sufficient to decompose A by using single precision arithmetic.

The algorithm is trivially *parallelizeable* by recognizing that the per-cell computations are independent from each other and that the communication cost C between macroscopic and microscopic simulation is relatively low. More specifically $C \approx 2fPKL$, where f is the fraction of all cells being computed ($\sim 15\%$) and P is the number of first moments characterizing the bubble configuration: bubble number, average size, etc. Hence, a cluster of GPUs could be used to solve several per-cell integral equation systems in parallel. Moreover, it has already been shown that one can improve the performance of scientific simulation of a CPU cluster by replacing it with a cluster of GPUs, particularly in the field parallel flow simulation using the lattice Boltzmann model [Fan et al. 2004].

The results of our experiments are shown in Figure 13. We have implemented the complete fluid flow simulation as described above in MATLAB. We compare the time it takes to solve the microscopic-level equations with our algorithm (without pivoting) on a NVIDIA 6800 Ultra versus the time it takes using the optimized ATLAS' `getrf()` implementation, which includes pivoting. The results of our simulations are shown in Figure 13. Our algorithm is 15% faster for $N = 2048$ and it obtains sufficient accuracy for this application.

The data in Figure 14 show that there is virtually no overhead of transferring the data between CPU and GPU, as it is a one time transfer, and the amount of data is relatively small compared to the total number of memory operations performed by the algorithm.

6.3 Implications of graphics hardware architecture

In this section, we highlight some features in GPUs that can lead to improved performance. These include:

- **Improved floating point bandwidth:** The performance of linear algebra algorithms could be significantly improved based on the sequential texture access bandwidth to 4-component floating point textures. The linear algebra matrices can be represented using the four color components of each texel and our algorithms can be directly used to perform vector computations, thereby improving the overall performance.

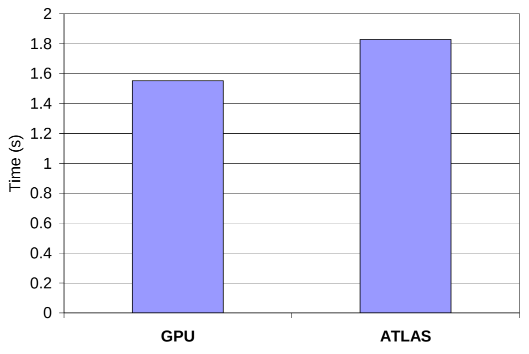

Figure 13: Comparison of computation time of our GPU based algorithm on a NVIDIA 6800 Ultra versus a ATLAS implementation of the microscopic level computations in a flow simulation of bubbly fluid. The graph indicates computation time per cell and per time step.

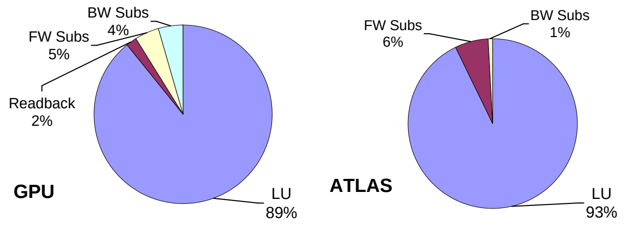

Figure 14: Breakdown of our GPU based algorithm on a NVIDIA 6800 Ultra versus a ATLAS implementation of the microscopic level computations in a flow simulation of bubbly fluid. Note that there is no matrix transfer required for the CPU, but this overhead is negligible for the GPU.

- **Blending:** The performance of our algorithm can also be improved by using floating point blending. In our algorithm, we would perform two passes, a first pass to multiply the index pairs, and a second pass to blend using vector addition. The performance can further be improved if blending hardware can directly support multiple sources and perform a MAD operation. Current GPUs support 16-bit floating point blending. Based on the technological trend, we expect that future GPUs might support IEEE 32-bit floating point blending.

6.4 Limitations

Our current GPU-based algorithm has a few limitations. First of all, current GPUs only support 32-bit floating point arithmetic. All the timings and results presented in the paper were using 32-bit floating point arithmetic. Secondly, the overhead of data transfer from CPU to the GPU can be significant for small matrices, e.g. of order less than 500. For large matrices, the data transfer time is insignificant. Finally, the maximum matrix size is 4096×4096 due to the limitations on the maximum texture size on current GPUs.

7 Conclusion and Future Work

We have presented a novel GPU-based algorithm for solving dense linear systems. We reduce the problem to a series of rasterization problems and use appropriate data representations to match the blocked rasterization order and cache pre-fetch technology of a GPU. We exploit high spatial coherence between elementary row operations and use fast parallel data transfer techniques to move data on GPUs. Our experimental results indicate that the performance of our algorithm scales well with the number of fragment processors as well as with the core clock rate of the GPU. Furthermore, our algorithm is cache and bandwidth efficient and has been applied to solve large linear systems. The performance of our current GPU-based implementation is comparable to that of optimized ATLAS library routines, running on state of the art workstations with vectorized SSE instructions. By applying our algorithm to a fluid flow simulation, we have used the GPU as a co-processor, freeing up CPU cycles for intermediate computations.

The performance of our GPU-based algorithms is comparable to the performance of optimized CPU-based implementations. The high growth rate and the increasing programmability in GPU architecture indicates that our linear algebra algorithms can provide significant performance improvement, faster than Moore's law, within the next few years.

There are many avenues for future work. As GPU architectures evolve, we would like to exploit the programmability and architectural features to further improve the performance of our algorithm. We would like to use our algorithm for other scientific applications. We would also like to investigate existing division-free algorithms [Peng et al. 1996] to increase robustness, and would also like to extend our approach for other matrix computations including eigendecomposition and singular value decompositions. Finally, we would like to develop GPU-clusters for solving large scientific problems that currently do not fit into the memory of a single GPU.

Acknowledgements. This work is supported in part by ARO Contracts DAAD19-02-1-0390 and W911NF-04-1-0088, NSF awards 0400134 and 0118743, ONR Contract N00014-01-1-0496, DARPA/RDECOM Contract N61339-04-C-0043 and Intel. We would like to thank Jan Prins for many useful discussions.

References

AHN, J. H., DALLY, W. J., KHAILANY, B., KAPASI, U. J., AND DAS, A. 2004. Evaluating the imagine stream architecture. In *Proceedings of the 31st Annual International Symposium on Computer Architexture, Munich, Germany*.

BOLZ, J., FARMER, I., GRINSPUN, E., AND SCHRÖDER, P. 2003. Sparse matrix solvers on the gpu: conjugate gradients and multigrid. *ACM Trans. Graph. 22*, 3, 917–924.

BUCK, I., FOLEY, T., HORN, D., SUGERMAN, J., FATAHALIAN, K., HOUSTON, M., AND HANRAHAN, P. 2004. Brook for gpus: stream computing on graphics hardware. *ACM Trans. Graph. 23*, 3, 777–786.

CHATTERJEE, S., LEBECK, A. R., PATNALA, P. K., AND THOTTETHODI, M. 1999. Recursive array layouts and fast parallel matrix multiplication. In *ACM Symposium on Parallel Algorithms and Architectures*, 222–231.

DALLY, W. J., HANRAHAN, P., EREZ, M., KNIGHT, T. J., LABONTE, F., A., J.-H., JAYASENA, N., KAPASI, U. J., DAS, A., GUMMARAJU, J., AND BUCK, I. 2003. Merrimac: Supercomputing with streams. In *SC'03*.

DEMMEL, J. W. 1997. *Applied Numerical Linear Algebra*. SIAM Books.

DONGARRA, J. J., DUFF, I. S., SORENSEN, D. C., AND VAN DER VORST, H. A. 1998. *Numerical Linear Algebra for High-Performance Computers*. SIAM.

DONGARRA, J. J., LUSZCZEK, P., AND PETITET, A. 2003. The LINPACK benchmark: Past, present, and future. *Concurrency and Computation: Practice and Experience 15*, 1–18.

EREZ, M., AHN, J., GARG, A., DALLY, W. J., AND DARVE, E. 2004. Analysis and Performance Results of a Molecular Modeling Application on Merrimac. In *SC'04*.

FAN, Z., QIU, F., KAUFMAN, A., AND YOAKUM-STOVER, S. 2004. Gpu cluster for high performance computing. In *ACM / IEEE Supercomputing Conference 2004*.

FATAHALIAN, K., SUGERMAN, J., AND HANRAHAN, P. 2004. Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, Eurographics Association.

GÖDDEKE, D. 2005. Gpgpu performance tuning. Tech. rep., University of Dortmund, Germany. `http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/`.

GPUBENCH, 2004. A GPU benchmarking suite. *GP*^{2} Workshop, Los Angeles. Available online: `http://graphics.stanford.edu/projects/gpubench/`.

GRAMA, A., GUPTA, A., KARYPIS, G., AND KUMAR, V. 2003. *Introduction to Parallel Computing (2nd ed.)*. Addison Wesley.

HAKURA, Z., AND GUPTA, A. 1997. The design and analysis of a cache architecture for texture mapping. In *Proc. of the 24th International Symposium on Computer Architecture*, 108–120.

HALL, J. D., CARR, N., AND HART, J. 2003. Cache and bandwidth aware matrix multiplication on the gpu. Technical Report UIUCDCS-R-2003-2328, University of Illinois at Urbana-Champaign.

HAMMERSTONE, CRAIGHEAD, AND AKELEY, 2003. ARB_vertex_buffer_object OpenGL specification. `http://oss.sgi.com/projects/ogl-sample/registry/ARB/vertex_buffer_object.txt`.

HARRIS, M. J., BAXTER, W. V., SCHEUERMANN, T., AND LASTRA, A. 2003. Simulation of cloud dynamics on graphics hardware. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 92–101.

KIM, T., AND LIN, M. 2003. Visual simulation of ice crystal growth. In *Proc. of ACM SIGGRAPH / Eurographics Symposium on Computer Animcation*.

KRÜGER, J., AND WESTERMANN, R. 2003. Linear algebra operators for gpu implementation of numerical algorithms. *ACM Trans. Graph. 22*, 3, 908–916.

LARSEN, E. S., AND MCALLISTER, D. 2001. Fast matrix multiplies using graphics hardware. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, ACM Press, 55–55.

LASTRA, A., LIN, M., AND MANOCHA, D. 2004. ACM workshop on general purpose computation on graphics processors.

MCCOOL, M., TOIT, S. D., POPA, T., CHAN, B., AND MOULE, K. 2004. Shader algebra. *ACM Trans. Graph. 23*, 3, 787–795.

MCGINN, S., AND SHAW, R. 2002. Parallel gaussian elimination using openmp and mpi. In *Proceedings. 16th Annual International Symposium on High Performance Computing Systems and Applications*, 169–173.

MCLEOD, I., AND YU, H. 2002. Timing comparisons of mathematica, matlab, r, s-plus, c & fortran. Technical report. Available online: `http://fisher.stats.uwo.ca/faculty/aim/epubs/MatrixInverseTiming/default.htm`.

MITRAN, S. 2000. Closure models for the computation of dilute bubbly flows. Wissenschaftliche Berichte FZKA 6357, Forschungszentrum Karlsruhe, April.

NVIDIA CORPORATION. 2004. The geforce 6 series of gpus high performance and quality for complex image effects. Tech. rep. Available on: `http://www.nvidia.com/object/IO_12394.html`.

OLIKER, L., CANNING, A., CARTER, J., AND SHALF, J. 2004. Scientific computations on modern parallel vector systems. In *Supercomputing 2004*.

PENG, S., SEDUKHIN, S., AND SEDUKHIN, I. 1996. Parallel algorithm and architecture for two-step division-free gaussian elimination. In *1996 International Conference on Application-Specific Systems, Architectures and Processors (ASAP'96)*.

RUMPF, M., AND STRZODKA, R. 2001. Using graphics cards for quantized FEM computations. In *Proc. of IASTED Visualization, Imaging and Image Processing Conference (VIIP'01)*, 193–202.

RUSSO, G., AND SMEREKA, P. 1996. Kinetic theory for bubbly flow i: Collisionless case. *SIAM J. Appl. Math. 56*, 2, 327–357.

RUSSO, G., AND SMEREKA, P. 1996. Kinetic theory for bubbly flow II: Fluid dynamic limit. *SIAM J. Appl. Math. 56*, 2, 358–371.

SUH, J., KIM, E.-G., CRAGO, S. P., SRINIVASAN, L., AND FRENCH, M. C. 2003. A performance analysis of pim, stream processing, and tiled processing on memory-intensive signal processing kernels. In *Proceedings of the International Symposium on Computer Architecture*.

TAYLOR, M. B., KIM, J., MILLER, J., WENTZLAFF, D., GHODRAT, F., GREENWALD, B., HOFFMANN, H., JOHNSON, P., LEE, J.-W., LEE, W., MA, A., SARAF, A., SENESKI, M., SHNIDMAN, N., FRANK, V. S. M., AMARASINGHE, S., AND AGARWAL, A. 2002. The raw microprocessor: A computational fabric for software circuits and general purpose programs. *IEEE Micro*.

THOTTETHODI, M. S. 1998. Tuning Strassen's matrix multiplication for memory efficiency. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, IEEE Computer Society, 1–14.

WHALEY, R. C., PETITET, A., AND DONGARRA, J. J. 2001. Automated empirical optimization of software and the ATLAS project. *Parallel Computing 27*, 1–2, 3–35. Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000 (`www.netlib.org/lapack/lawns/lawn147.ps`).