# LU, QR and Cholesky Factorizations using Vector Capabilities of GPUs

*LAPACK Working Note 202*

Vasily Volkov

Computer Science Division
University of California at Berkeley

James W. Demmel

Computer Science Division and Department of Mathematics
University of California at Berkeley

## Abstract

We present performance results for dense linear algebra using the 8-series NVIDIA GPUs. Our matrix-matrix multiply routine (GEMM) runs 60% faster than the vendor implementation in CUBLAS 1.1 and approaches the peak of hardware capabilities. Our LU, QR and Cholesky factorizations achieve up to 80–90% of the peak GEMM rate. Our parallel LU running on two GPUs achieves up to ~300 Gflop/s. These results are accomplished by challenging the accepted view of the GPU architecture and programming guidelines. We argue that modern GPUs should be viewed as multithreaded multicore vector units. We exploit blocking similarly to vector computers and heterogeneity of the system by computing both on GPU and CPU. This study includes detailed benchmarking of the GPU memory system that reveals sizes and latencies of caches and TLB. We present a couple of algorithmic optimizations aimed at increasing parallelism and regularity in the problem that provide us with slightly higher performance.

## 1 Introduction

We make the following contributions. For the first time, we show an LU, QR and Cholesky factorization that achieve computational rates that approach 200 Gflop/s on a GPU. These are three of the most widely used factorizations in dense linear algebra and pave the way for the implementation of the entire LAPACK library for the GPUs.

These rates are achieved on the 8-series of NVIDIA GPUs that have been available for about 1.5 years. However, we program these GPUs in a way that was not done before and achieve performance in such basic kernels as matrix-matrix multiply that is 60% higher than those in the optimized vendor's library CUBLAS 1.1. In the core of our approach we think of the GPU as a multithreaded vector unit, which provides many insights and inspirations from the mature field of vector computing.

We perform detailed benchmarks of the GPU and reveal some of the bottlenecks, such as access to the on-chip memory that bounds the performance of our best codes, and kernel launch overheads that prohibits efficient fine-grain computations. The benchmarks reveal the structure of the GPU memory system, including sizes and latencies of the L1 and L2 caches and TLB. We implement and measure the performance of global synchronization primitives such as barrier for the first time on the 8-series of GPUs. We believe this is an important component for the overall programmability of current GPUs.

To achieve the best performance in matrix factorizations we use state of art techniques such as look-ahead, overlapping CPU and GPU computation, autotuning, smarter variants of 2-level blocking, and choosing the right memory layout; we also use a novel algorithm with modified numerics. We analyze the performance of our implementations in detail to show that all components of the final system run at the nearly optimal rates.

Our best speedups vs. one dual core CPU are up to 7–8× in all 3 factorizations and 3–5.5× vs. one quad core CPU.

The rest of this paper is organized as follows. Section 2 describes the architecture of the GPUs we used, highlighting the features important for performance, and drawing parallels to conventional vector and SIMD architectures. Section 3 benchmarks operations including memory transfer, kernel start-up, and barriers, and uses these to analyze the performance of the panel factorization of LU. Section 4 discusses the design and performance evaluation of matrix multiplication. Section 5 discusses the design of LU, QR and Cholesky, and Section 6 evaluates their performance. Section 7 summarizes and describes future work.

| GPU name | GeForce 8800GTX | Quadro FX5600 | GeForce 8800GTS | GeForce 8600GTS |
|---|---|---|---|---|
| # of SIMD cores | 16 | 16 | 12 | 4 |
| core clock, GHz | 1.35 | 1.35 | 1.188 | 1.458 |
| peak Gflop/s | 346 | 346 | 228 | 93.3 |
| peak Gflop/s/core | 21.6 | 21.6 | 19.0 | 23.3 |
| memory bus, MHz | 900 | 800 | 800 | 1000 |
| memory bus, pins | 384 | 384 | 320 | 128 |
| bandwidth, GB/s | 86 | 77 | 64 | 32 |
| memory size, MB | 768 | 1535 | 640 | 256 |
| flops:word | 16 | 18 | 14 | 12 |

Table 1: The list of the GPUs used in this study. Flops:word is the ratio of peak Gflop/s rate in multiply-and-add operations to pin-memory bandwidth in words.

## 2 GPU Architecture

In this work we are concerned with programming 8-series NVIDIA GPUs, as listed in Table 1. They expose extensive programming flexibility such as being able to execute scalar threads with arbitrary memory access patterns and branch behaviors, which are best described in the CUDA programming guide [NVIDIA 2007]. However, exploiting this flexibility may cost 10–100× loss in performance; see Section 3.7 for an example that exposes the factor of 100×.

In this section we describe GPUs as multithreaded SIMD architectures. Novel facilities that are not usually present in SIMD architectures and designed to support non-SIMD programs at performance cost are briefly reviewed but not used in the rest of the paper. The purpose of this exposition is to encourage the user to expose parallelism as required by the hardware and reuse previous findings in programming vector and SIMD architectures.

### 2.1 SIMD Cores

Earlier GPUs had a 2-level SIMD architecture — an SIMD array of processors, each operating on 4-component vectors. Modern GPUs have a 1-level SIMD architecture — an SIMD array of scalar processors. Despite this change, it is the overall SIMD architecture that is important to understand.

The fastest GPU in our study, GeForce 8800GTX, has 128 scalar processors which are partitioned into 16 multiprocessors [NVIDIA 2006; NVIDIA 2007]. The multiprocessors have an SIMD architecture [NVIDIA 2007, Ch. 3.1]. Scalar threads are

grouped into SIMD groups called "warps" [NVIDIA 2007, Ch. 3.2], with 32 scalar threads per warp [NVIDIA 2007, Ch. A.1]. There is one instruction fetch/dispatch unit per multiprocessor that synchronously broadcasts instructions to the scalar processors. It takes 4 cycles to execute one instruction for the entire warp [NVIDIA 2007, Ch. 5.1.1.1], i.e. one scalar instruction per cycle per scalar processor.

Thus, a multiprocessor is an SIMD unit with scalar processors effectively being its SIMD lanes. We refer to it as a SIMD core to put it into the context of other modern multicore systems, such as Core2 and Cell. This definition is also convenient from a performance perspective. One GPU core has 19–23 Gflop/s of peak arithmetic performance in multiply-and-add operations (see Table 1), which is similar to the peak performance of the CPU cores (~21.3 Gflop/s/core for 2.66GHz Intel Core2) and SPE units of the Cell processor (25.6 Gflop/s per SPE).

Another important feature of GPUs is multithreading that is designed to hide memory and pipeline latencies. To facilitate a low-cost context switch, all simultaneously running threads keep their register states in the same register file. The number of registers consumed by a thread depends on the program. There is also a small local memory storage on each SIMD core called shared memory that is partitioned among groups of threads called thread blocks. The user is allowed to create more threads than can fit simultaneously in registers and local memory. In that case some of the threads are not initiated until others finish. However, this mechanism is provided for convenience and promises little performance and functionality gain.

### 2.2 Executing Non-SIMD Programs

SIMD architectures permit simulating fully MIMD execution by following all execution paths in the program and masking non-participating processors off. GPUs offer a novel optimization to this execution style. First, they manage the stack of predication masks in hardware to save and restore the previous predication state at branch instructions. Second, if all scalar threads within the SIMD width take same path, other paths are not executed at all. The mechanism is discussed in detail in other companies' manuals, such as [AMD 2006]. NVIDIA describes a similar mechanism, but in less detail [NVIDIA 2007, Ch. 5.1.1.2].

The purpose of this mechanism is functionality, not performance. Branch divergence within SIMD width should be avoided if possible to avoid idle cycles on the scalar processors that are masked off.

### 2.3 SIMD Memory Access

GPUs provide high-bandwidth non-cached SIMD memory loads and stores. These operate on correctly aligned contiguous locations in memory.

As with control structures, non-cached non-SIMD memory operations are also supported in hardware but run at an order of magnitude lower bandwidth. Thus, they must be avoided if possible. For example, a stride-2 vector fetch may be implemented instead as an aligned stride-1 fetch by discarding the redundantly fetched data.

GPUs also provide cached memory fetches. These do not require exposing an SIMD pattern for better performance. However, cached access requires high spatial locality within every vector gather due to the small cache size, see Section 3.3 for more details.

All these types of memory accesses are exposed in the ISA as indexed gathers and scatters. This implies supplying redundant indices in the case of SIMD accesses.

### 2.4 On-Chip Memory Hierarchy

Each SIMD core has 32KB register file partitioned across SIMD lanes. For the GeForce 8800 GTX this amounts to 512KB on a single chip, which is larger than any other particular level of the on-chip memory hierarchy. This motivates different design decisions than those used for superscalar processors that have relatively few registers (e.g. 128–256 bytes in SSE units) and massive caches.

The second largest level of the on-chip memory hierarchy is the local memory — it is 16KB per SIMD core and 256 KB in total. It can effectively be used as scalar registers and also permits indexed access.

Other important on-chip memories are L2 and L1 read-only caches that amount to 192KB and 40KB on 8800GTX respectively according to our research as presented in Section 3.3.

### 2.5 Vector Program Model

Performance programming for GPUs is most similar to programming for other multicore SIMD architectures, such as Core2 SSE and Cell SPE units. Such programs are often expressed as operations on vectors. This can be applied to GPUs by exposing a single warp as one *SIMD* or *vector thread*. Scalar threads such as exposed in CUDA correspond to *vector elements*. We also use term *vector thread* when referring to a thread block. This conveniently expresses independent threads of data-parallel work, encapsulates data-parallel communication using local memory and simulates configurable vector length.

The vector length should be set to a small multiple of the native SIMD width. The CUDA programming guide recommends a multiple of 64 scalars to avoid register bank conflicts [NVIDIA 2007, Ch. 5.1.2.5]. In practice we found that 64 may both give the nearly best performance as demonstrated in Section 3.4 and keep resource consumption low. Indeed, temporary values in data-parallel program consume at least the vector length. Scalar values and operations may also need to be replicated across the entire vector length. Memory scatters and gathers require as many pointers as the vector length. This all motivates using short vectors. Longer vectors that may be convenient to deal with in applications should then be strip-mined into shorter vectors. As longer vectors are implemented using more warps, i.e. native SIMD threads, strip-mining converts thread-level parallelism into instruction-level parallelism that also contributes towards hiding pipeline and memory latencies.

## 3 Microbenchmarks

The results reported in this section were measured on a few different systems running Windows XP and 64-bit Linux, each equipped with one of the four GPUs listed in Table 1. The name of a particular GPU is mentioned if it matters. All programming of GPUs was done using CUDA 1.1 if not specified otherwise.

To ensure the quality of the GPU code produced by the CUDA compiler we used *decuda*[1], which is a third-party disassembler of the GPU binaries. The vendor's tools, such as the virtual ISA called PTX, was less helpful, as it exposes different consumption of instruction slots and registers. The native instruction set for NVIDIA GPUs is not officially released.

### 3.1 Kernel Launch Overhead

It takes 5 μs to invoke a GPU kernel using the low-level CUDA

[1] http://www.cs.rug.nl/~wladimir/decuda/

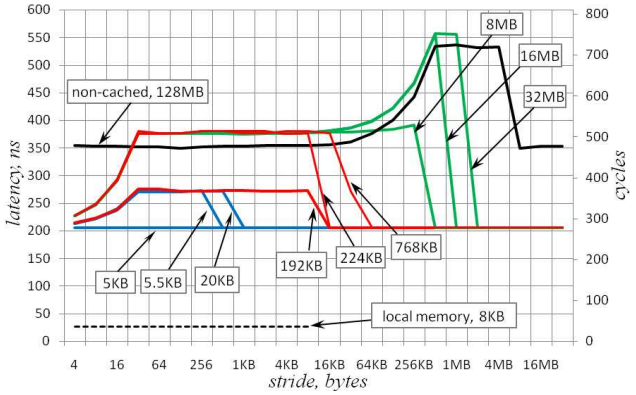


Figure 1: Memory latency as revealed by the pointer chasing benchmark on GeForce 8800 GTX for different kinds of memory accesses. Array size is shown in the boxes. Cached access assumed unless otherwise specified. Blue, red and green lines highlight 5KB cache, 192 KB cache, and 512KB memory pages respectively. Solid black is non-cached access, dashed black is local memory.

API (one `cuLaunchGrid` call or two `cuParamSetv` and `cuLaunchGrid` calls if new arguments must be supplied — both cases give similar timings). It takes 5–7 µs if done with the higher level API (`<<< >>>` expression). This was measured by asynchronously invoking the same kernel a very large number of times and synchronizing once at the end (e.g. using `cudaThreadSynchronize` API call). The program used was the simplest possible, such as copying one word from one location in the GPU memory to another. This is to ensure that the program runtime does not contribute substantially to the overall time. The time increases to 11–12 µs when synchronizing at each kernel invocation. This gives an idea of how expensive the synchronization is.

To ensure that we do not sacrifice performance by choosing CUDA for programming the GPU we also measured overheads in DirectX 9.0c, which is a mature graphics API widely used in computer games. The timings were 7 µs for invocation alone and 21 µs for invocation with synchronization (synchronization is required when computing with DirectX to ensure correctness). This indicates that CUDA is as efficient as or better than DirectX.

### 3.2 CPU-GPU Data Transfers

The GPUs used in this study were designed for the PCIe 1.1 ×16 interface that bounds the bandwidth of the CPU-GPU link by 4 GB/s (newer GPUs support PCIe 2.0 which is twice as fast). We found that transferring contiguous pieces of data with sizes from 1 byte to 100 MB long across this link using CUDA with pinned memory takes about

$$Time = 15\mu s + \frac{bytes\ \ transferred}{3.3GB/s}. \qquad (1)$$

This fits the measured data within a few percent.

### 3.3 GPU Memory System

The vendor's manuals supply limited information on GPU caches. The CUDA programming guide specifies an 8KB cache working set per SIMD core [NVIDIA 2007, Ch. A.1], i.e. 128KB for the entire 8800 GTX chip (there is also a cache for small constant memory that we leave out of scope in this paper). He et al. [2007] estimate the size of the 8800GTX cache to be 392KB. None of them differentiate levels of cache. However, it

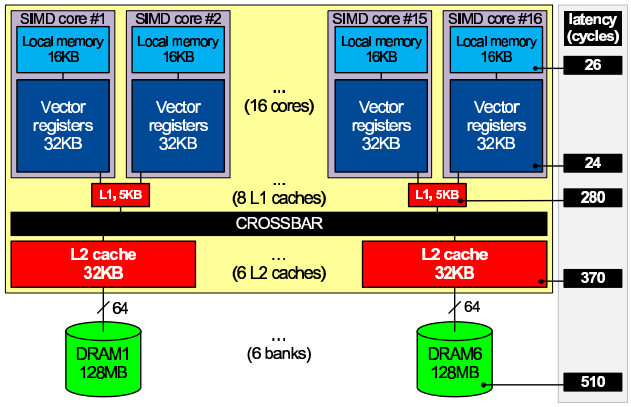


Figure 2: Summary of the memory system of 8800GTX according to our study. Sizes of the on-chip memory levels are shown in the same scale. Latencies shown are for the cached access. Note the small size of the L1 caches.

is known that the 8800GTX has one L1 cache per two cores and six L2 caches [NVIDIA 2006]. L1 caches are connected with L2 caches via a crossbar.

We use a traditional pointer chasing benchmark similar to that used, for example, in LMBench[2] to reveal the latency and structure of the memory system. It traverses an integer array $A$ by running $k = A[k]$ in a long unrolled loop, yielding the time per one iteration. This time is dominated by the latency of the memory access. The traversal is done in one scalar thread, and so utilizes only one GPU core and may not see caches associated with other cores. The array is initialized with a stride, i.e. $A[k] = k + stride$ mod $array\ size$. We test cached and non-cached memory access to the off-chip memory and also access to the local memory (in which case data is first copied from the off-chip memory and this time is later subtracted). Results for different array sizes and strides on the 8800 GTX are shown in Fig. 1.

A larger latency indicates more cache misses. The array size defines the working set and reveals the cache size, such as 5KB and 192KB in the Figure. The higher latency of the long-stride non-cached access indicates the presence of a TLB, which is not officially documented to the best of our knowledge. The stride reveals cache lines and memory pages, such as 32 bytes and 512KB in the Figure. When the stride is very large, working set decreases until it again fits in the cache, this time producing conflict misses if cache is not fully associative. The data in Fig. 1 suggests a fully associative 16-entry TLB (no TLB overhead for 128MB array, 8MB stride), a 20-way set associative L1 cache (20KB array at 1KB stride fits in L1), and a 24-way set-associative L2 cache (back to L2 hit latency for 768KB array, 32KB stride). These are the effective numbers and the real implementation might be different. Six 4-way set-associative L2 caches match this data as well.

According to this data, L1 cache has 160 cache lines only (in 8 fully associative sets). This promises a 100% miss rate in every cached access unless scalar threads are sufficiently coordinated to share cache lines.

Figure 1 reveals a 470–720 cycle latency non-cached memory access that roughly matches the official 400–600 cycle figure [NVIDIA 2007, Ch. 5.1.1.3].

To find the total amount of the partitioned cache memory, we run a multithreaded test that utilizes all cores. We run one thread per core (this is enforced by holding a large amount of local memory per thread), each traversing through a private

[2] http://www.bitmover.com/lmbench

array so that their working sets do not overlap. The results match the official data, with the effective size of L1 cache scaling with the number of cores. Effective L2 cache size did not scale. Fig. 2 summarizes the parameters of memory system of 8800GTX including the findings cited above. Preliminary study shows that TLB also scales with number of cores.

Similar tests and plots for other GPUs in this study suggested the same sizes of L1 caches (5KB per 2 cores) and TLB (16 entries per TLB), and showed that L2 caches scale as memory pins: 32KB for each 64 pins (to match 6 caches in the 8800 GTX [NVIDIA 2006]). Also, it matches 128MB memory per 64 memory pins on all GPUs but the FX5600, which has twice as much memory. Our guess is that L2 GPU caches are similar in function to the hot-spot caches on the earlier highly multithreaded processors such as Tera MTA [Alverson et al. 1990] that were designed to alleviate contention at memory banks.

Latencies expressed in cycles were about same on all four GPUs. Note that an L1 cache hit costs about 280 cycles (260 on 8600 GTS) which is about half of the memory access latency. According to the vendor's manual, the purpose of the GPU cache is to reduce "DRAM bandwidth demand, but not fetch latency" [NVIDIA 2007, Ch. 5.1.2.3]. Interestingly, the same purpose is followed in the design of the vector cache in the Cray BlackWidow vector computer [Abts et al. 2007].

Latency to the local memory is an order of magnitude less than to the cache — 36 cycles. To measure it more accurately and compare to the pipeline latency we performed a separate experiments on 8800GTX. This time we execute $a = a * b + c$ operation many times in an aggressively unrolled loop. We used *decuda* to ensure that this operation maps to a single native instruction. When all three variables are in registers the measurements show 24 cycle throughput per instruction that is 6× larger than at the peak throughput and is an estimate of the pipeline latency. Same test showed 26 cycles when $b$ was in local memory. 2 extra cycles for operations with local memory appear again in Section 3.5.

24 cycle latency may be hidden by running simultaneously 6 warps or 192 scalar threads per SIMD core, which explains the number cited in the CUDA guide [NVIDIA 2007, Ch. 5.1.2.5].

### 3.4 Attaining Peak Instruction Throughput

We were able to achieve 98% of the arithmetic peak on 8800GTX in register-to-register multiply-and-add instructions. This was achieved running a single vector thread per SIMD core. Vector length was two warps or 64 elements (1024 scalar threads in total). Each thread performs a group of 6 independent multiply-and-adds a million times in an aggressively unrolled loop. The number 6 was chosen to hide the pipeline latency no matter how many threads are run per SIMD core. However, it didn't work when there was only one warp per SIMD core (we got 50% of peak only) or two warps in different thread blocks (66% of peak) when using the same code.

### 3.5 Throughput when using Local Memory

According to *decuda*, locations in local memory can be used as an instruction operand. However, the best of our experiments yielded only 66% of the arithmetic peak on all four GPUs in multiply-and-add instructions with one operand in local memory. This corresponds to 6-cycle throughput per warp versus the usual 4 and 230 Gflop/s on 8800GTX.

To isolate the effect, we tried different local memory access patterns keeping the structure of the inner loop unperturbed. We found that it takes 12 cycles per instruction if each local memory access involves a 2-way local memory bank conflict (as defined in [NVIDIA 2007, Ch. 5.1.2.5]), 24 cycles if conflicts are 4-way, etc. This fits the description in the CUDA guide that says that conflicting accesses are serialized. This indicates that the bottleneck is in the local memory access, not in the hidden pointer arithmetic.

### 3.6 Faster Global Barrier

A global barrier is a basic synchronization primitive that is widely used in many parallel algorithms. Currently, it is common to assume that global barrier on the GPU should be implemented by invoking a new kernel, which involves expensive overhead as we have found in Section 3.1. For the first time we show that it is possible to implement global a barrier within a single kernel run. Of course, it synchronizes only among threads that run simultaneously and thus introduces explicit dependence on the number of GPU cores. In Section 3.8 we argue that this mechanism may be important in fine-grain computations on the GPUs.

Our implementation does not use atomic primitives available in the newer GPUs (however, it assumes that word-wide non-cached memory reads and writes are atomic). Instead, we replicate the variables used in the synchronization to ensure that different threads never write to the same address. This keeps changes to these variables atomic. In our implementation we allocate *arrival* and *wakeup* variables for each vector thread. There is one master vector thread and others are slaves. The *i*-th slave updates the *i*-th *arrival* variable and spins on the *i*-th *wakeup* variable until that is updated. The master thread spins on the *arrival* variables until every one is updated, then updates every *wakeup* variable. For better efficiency we lay out the counters in memory to ensure that the head thread fetches and stores the variable values using a single SIMD memory access. In our prototype implementation we pick the vector length equal to the total number of vector threads created.

1.3–1.6 μs or 1920–2000 cycles per barrier was observed on all four GPUs used in the study when running one vector thread per core. This is about 4 memory latencies of non-cached access, which is the minimum number of data trips between the processing chip and the memory as assumed by the algorithm. This time was up to 2800 cycles when running multiple (up to 8) vector threads per core.

Although substantial, this time is 3–4× less than the kernel launch overhead, so it can be used to speedup fine-grain computations that otherwise require multiple kernel runs. However, we note that this barrier does not guarantee that previous accesses to all levels of the memory hierarchy have completed unless a memory consistency model is assumed. Also, its practical application is complicated by the inability to change the thread block size and the register partitioning during the kernel execution.

### 3.7 GPU Memory Bandwidth

The best bandwidth in copying within GPU memory on the 8800GTX that we could attain was 76 GB/s, which is 88% of the pin-bandwidth. This proves that a large fraction of the peak bandwidth may be attained in SIMD accesses, which should be the goal in all bandwidth-bound codes. However, the same code gets 11× lower bandwidth if the supplied pointers are not aligned. Thus, performance codes must always include basic optimizations to avoid misaligned accesses. For example, a misaligned stride-1 copy should be converted into a large aligned stride-1 copy and one less efficient but much smaller copy at the head of the array. This can be performed within a single kernel call.

Even stronger deterioration is possible in strided accesses.

Copying $2^{18}$ 32-bit numbers in GPU memory with strides 1, 8 and 512 takes 34 µs, 260 µs and 2600 µs correspondingly, which is effectively 62 GB/s, 8.1 GB/s and 0.81 GB/s. This is done by a simple code, where each vector thread reads and writes one vector with the given stride. Performing the same operation using the possibly better optimized vendor's library CUBLAS 1.1 gave similar results.

### 3.8 Implications for Panel Factorization

Results shown by Barrachina et al. [2008] and Baboulin et al. [2008] indicate that copying a matrix panel to the CPU memory, factorizing it using the CPU and transferring it back to the GPU may take less time than performing panel factorization on the GPU using the vendor-supplied BLAS library. We analyze this behavior below using a performance model.

Consider running the LU factorization of a panel as done in LAPACK's `sgetf2` on the GeForce 8800GTX. This routine is built of calls to BLAS1 and BLAS2 that have flop-to-word ratio as low as 1 or 2, which is much below the flop-to-word ratio of the GPUs (see Table 1) and thus are bandwidth bound. Let us assume they are efficiently implemented, i.e. run in

$$Time = 5\mu s + \frac{bandwidth\ required}{75GB/s}, \qquad (2)$$

where we used the kernel launch overhead and the peak sustained bandwidth found above. (Although one might imagine an environment where runtime of the GPU program and launch overhead overlap, we observed that these times sum up in practice if the most efficient invocation routine in CUDA 1.1 is used.) For example, consider the BLAS1 routine `sscal` that scales a vector by a constant factor. It does $n$ flops and requires $2n$ words of bandwidth for an $n$-element vector. The highest rate it can attain on this GPU is therefore $r_\infty$ = 9.4 Gflop/s. Half of this rate is achieved at $n_{1/2} \approx 47{,}000$, i.e. it runs at 5–10µs for any $n < 47{,}000$. The best of our GPUs has 1.5GB memory that fits up to 20,000×20,000 matrices. Thus, for practical square matrix sizes `sscal` effectively runs in O(1) time instead of the asymptotic O($n$).

Using this model we estimate the runtime of the entire factorization of an $m \times n$ panel, $m > n$. It involves $n$ calls to each of the `isamax` (finds largest element in a vector), `sscal`, `sswap` (swap rows) and $n$–1 call to `sger` (rank-1 update). Assume an ideal implementation, i.e. each routine reads and writes inputs and outputs only once (e.g. the input row and column in the rank-1 update stay in cache, registers or local memory). Fig. 3 shows the result for $n$ = 64 (labeled "estimate") compared with few other Gflop/s rates. 64 is the panel width we use in practice as described in Section 5.

According to the plot, the GPU may yield up to 2.4× speedup vs. the CPU (labeled "CPU+transfer") but may achieve this efficiency only at large problem sizes. The CPU timings in the plot are handicapped by including the time to transfer panel to the CPU and back computed as in (1).

The asymptotic performance of the factorization computed using the model is 18.8 Gflop/s. Only a small fraction of it is achieved when factorizing a small panel mostly due to the large startup times. To highlight the importance of the faster global barrier introduced in Section 3.6, we computed similar estimates using its overhead, which is 1.4µs on this GPU, as the startup time in (2). The new estimate is plotted in the same Figure and labeled "fast barrier, estimate". The new barrier promises to yield up to 3× speedup and outperforms CPU starting at smaller $m \approx 750$.

In the same graph we show two implementations of the panel factorization on the GPU that are labeled "optimized" and "naive". One of them is a naïve implementation that executes

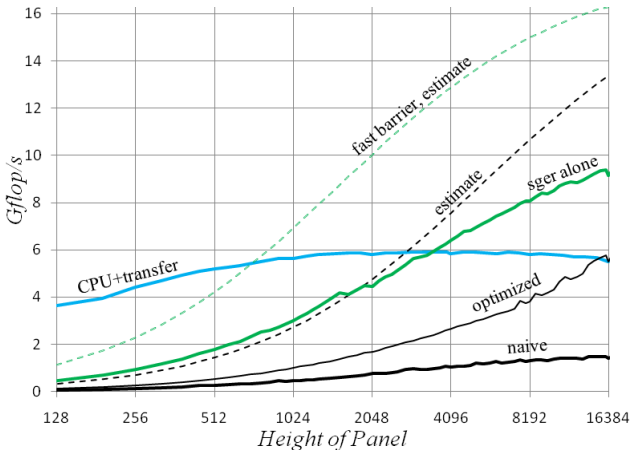


Figure 3: Gflop/s rates achieved in LU factorization of $m$×64 panel. GPU is 8800GTX programmed using CUDA 2.0 beta, CPU is 2.66GHz Core2 Duo CPU using Intel MKL 10.0. Curve corresponds to the GPU performance if not specified otherwise. Dashed lines are theoretical estimates, others are empirical data.

LAPACK's `sgetf2` code using CUBLAS 2.0 beta. This is a newer version of CUBLAS that contains a faster implementation of `sger` than in CUBLAS 1.1. The only substantial difference with the LAPACK code that we introduced was not checking if $1/a_{ii}$ underflows, thus always using `cublasSscal`. The factors $a_{ii}$ were fetched to the CPU using `cublasGetVector`. This version achieves up to 1.5 Gflop/s which is only 11% of the estimated value. This is also 10% of the peak sustained rate of `cublasSger` which is 15.0 Gflop/s for aligned data. The low fraction is due to working with unaligned data.

The second implementation included a couple of basic optimizations. The most important optimization was to implement `sger` as two calls to `cublasSger` in the manner described in Section 3.7 to reduce the work done with unaligned data. Another optimization was implementing `isamax` on the CPU if $m$ is small enough. In that case the data is first fetched to the CPU memory, then reduction is performed. This implementation runs up to 5× faster than `cublasIsamax` in CUBLAS 2.0 beta, which never takes less than 74µs (~15 kernel launch overheads!). Finally, scaling was implemented by a custom kernel that does not require copying $a_{ii}$ to the CPU. The optimized version runs at up to 5.8 Gflop/s which is nearly 4× speedup compared to the non-optimized code. However, this is only 41% of the estimated value.

The rate achieved in `cublasSger` in the optimized code is shown in the same plot (labeled "sger alone"). Most of the work is done in this routine. Note the gap in performance with the entire panel factorization. This is the practical slowdown due to the low work-to-cost ratio of `isamax`, `sscal` and `sswap`.

Further optimization might bring performance closer to the estimates. However, there could be other fundamental bottlenecks that prohibit achieving this simplified estimate. Implementing panel factorization on the GPU may not worth the effort and it may be preferable to offload this work to the CPU. We always perform panel factorization on the CPU in practice.

## 4 Design of Matrix-Matrix Multiply Routine

In this Section we describe the designs of $AB$ and $AB^T$ matrix-matrix multiplication routines that run at up to 90% of the arithmetic peak for operations using local memory as found in Section 3.5. This is 60% faster than in the CUBLAS 1.1 library released by NVIDIA. At the time of writing, our codes have been adopted NVIDIA and are included in CUBLAS 2.0 beta. In

```
// version: C := αAB^T + βC
Compute pointers in A, B and C using thread IDs
s[1:4] = next 64×16 block in A
t = next 4×16 block in B
c[1:16] = 0
do
   copy s[1:4] into a[1:4]
   copy t into b[1:4][1:16]
   (local barrier)
   s[1:4] = next 64×16 block in A
   t = next 4×16 block in B
   c[1:16] += a[1:4]*b[1:4][1:16]
   (local barrier)
   update pointers in A and B
repeat until pointer in B is out of range
copy t into b[1:4][1:16]
(local barrier)
c[1:16] += s[1:4]*b[1:4][1:16]
Merge c[1:16] with 64×16 block of C in memory
```

```
// version: C := αAB + βC
Compute pointers in A, B and C using thread IDs
c[1:16] = 0
do
   a[1:4] = next 64×16 block in A
   b[1:16][1:16] = next 16×16 block in B
   (local barrier)
   c[1:16] += a[1:4]*b[1:4][1:16]
   a[1:4] = next 64×16 block of A
   c[1:16] += a[1:4]*b[5:7][1:16]
   a[1:4] = next 64×16 block of A
   c[1:16] += a[1:4]*b[8:11][1:16]
   a[1:4] = next 64×16 block of A
   c[1:16] += a[1:4]*b[12:15][1:16]
   (local barrier)
   update pointers in A and B
repeat until pointer in B is out of range
Merge c[1:16] with 64×16 block of C in memory
```

Figure 4: Vector thread programs for matrix-matrix multiply. Bold characters **a**, **c**, **s** and **t** represent vector registers, *b* is in local memory.

Section 6.3 we describe a few independent works that have been done using our routines.

### 4.1 Block Matrix Multiply

Consider evaluating product $C := C + AB$, where $A$, $B$ and $C$ are $m \times k$, $k \times n$ and $m \times n$ matrices resp. Partition these matrices into $M \times K$, $K \times N$ and $M \times N$ grids of $bm \times bk$, $bk \times bn$ and $bm \times bn$ blocks. Suppose that fast memory can hold one block in $C$, $A$ and $B$ at the same time. Consider the *ijk*/*jik*-variant of the algorithm that holds the block of $C$ until all updates to it are accumulated (other variants may involve multiple updates of $C$ from different threads resulting in a race condition). Then computing one block in $C$ requires fetching $K$ blocks of $A$ and $B$. There are $M \cdot N$ blocks in $C$, so in total these fetches consume $M \cdot N \cdot K \cdot bm \cdot bk + M \cdot N \cdot K \cdot bk \cdot bn = m \cdot n \cdot k \cdot (1/bn + 1/bm)$ words of bandwidth. This is $2/(1/bn+1/bm)$ times less than if no blocking is used, i.e. if $bm = bn = bk = 1$. Since this factor does not depend on $bk$, small $bk$ can be recommended when the fast memory is small. For example, $bm = bn = b$, $bk = 1$ requires nearly 3× less local storage than $bm = bn = bk = b$ but requires the same bandwidth. We use $bk = 4$, which corresponds to a moderate unrolling of the inner loop. Below we refer to $bm \times bn$ as the block size.

The amount of bandwidth reduction should be at least as large as the flop-to-word ratio of the machine. This ratio is 18 for GeForce 8800 GTX if our goal is approaching 346 Gflop/s under 75 GB/s cap on the bandwidth and 12 if the goal is 230 Gflop/s. Thus, the minimum block is 18×18 and 12×12 resp. We were satisfied with the results we got with 32×32 (not discussed in this paper) and 64×16 blocking and did not try using smaller blocks.

Our algorithm is similar to one by Agarwal and Gustavson [1989] designed for IBM 3090 Vector Facility and Anderson et al. [2004] for Cray X1. In these implementations blocks in $A$ and $C$ are stored in registers and blocks in $B$ are in other fast memory that is shared across different vector elements — scalar registers and cache respectively. We keep $B$ in local memory. Therefore, each multiply-and-add instruction uses data in local memory that bound the performance of algorithm by 66% of the arithmetic peak as found in Section 3.5. In practice we achieve 60% of the arithmetic peak as discussed below in detail. Substantially faster solution should rely on using less expensive sharing of data in matrix blocks than local memory.

### 4.2 Implementation Details

We implemented the $C := \alpha AB + \beta C$ and $C := \alpha AB^T + \beta C$ cases of matrix multiplication for matrices in column-major layout, where $\alpha$ and $\beta$ are scalars. We got our best results with vector length 64. We create $M \cdot N$ vector threads, one thread per 64×16 block in $C$. Only non-cached memory access is used. Matrix $A$ is fetched in 64×4 blocks. Matrix $B$ is fetched in 16×16 blocks and matrix $B^T$ is fetched in 4×16 blocks. This ensures that all memory fetches follow the SIMD restrictions and can run at maximum bandwidth. In multiplying $AB$, a block of $B$ is laid out in the row-major format in local memory (same for the block of $B^T$ in $AB^T$). This reduces the amount of pointer arithmetic required to iterate through it. 16×16 arrays are padded as advised in [NVIDIA 2007] to avoid bank conflicts in column access when storing 16×4 blocks in the $AB$ case. To improve latency hiding, we prefetch one block ahead in the $AB^T$ code[3]. The other code has a sufficiently long body so that the compiler puts gather instructions sufficiently early. Figure 4 illustrates these algorithms.

Similar code is used to deal with block triangular matrices such as that appear in Cholesky factorization. When $C$ is block triangular, we create as many threads as it has nonzero blocks. Since threads are always created as a 1D or 2D array, this involves extra arithmetic in converting the thread ID into a block index (a block triangular matrix is cut in two pieces that are flipped and fit together to make a rectangle). A simpler solution would be creating threads as for a full matrix, and making threads corresponding to the non-participating blocks (those below or above the diagonal) exit immediately. If either $A$ or $B$ is block triangular, we modify the thread's startup code to adjust pointers in matrices $A$ and $B$ and/or number of iterations of the inner loop according to the thread ID. Either way we don't change the structure of the inner loop, thus the code runs at similar peak rates as the usual matrix-matrix multiply.

The code is written in CUDA's C to offload part of the work

[3] using prefetching and "`-maxrregcount 32`" compiler option is due to Paul Leventis

| GPU | peak | $C := C + AB$ | | | | | $C := C + AB^T$ | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | CUBLAS | est. | **actual** | no fetch | no pref. | CUBLAS | est. | **actual** | no fetch | no pref. |
| FX5600 | 230 | 127 | 202 | **205** | 210 | 183 | 122 | 197 | **205** | 205 | 191 |
| 8800GTX | 230 | 128 | 202 | **206** | 210 | 186 | 122 | 197 | **205** | 205 | 192 |
| 8800GTS | 152 | 84 | 133 | **136** | 138 | 123 | 81 | 130 | **136** | 136 | 127 |
| 8600GTS | 62 | 35 | 55 | **56** | 57 | 49 | 33 | 53 | **56** | 56 | 50 |

Table 2: Estimated and best observed rates in Gflop/s when multiplying square matrices up to 4096×4096. *Peak* — the peak for multiply-and-add operation with one argument in local memory according to our benchmarks. *CUBLAS* — `cublasSgemm` in CUBLAS 1.1, *est.* — the estimated rate, *actual* – the observed rate, *no fetch* — fetches in $A$ and $B$ substituted with assignments, *no pref.* — no prefetching used. For comparison, `sgemm` in Intel MKL 10.0 runs at up to 70 Gflop/s on 2.4GHz Core2 Quad.

to the compiler. As runtime of our implementation is bound by instruction slots and register usage, it was important to ensure efficient code generation. This was accomplished using *decuda*. We needed to enforce a tight register budget by using the "`-maxrregcount 32`" compiler option. This ensures that each vector thread uses not more than 32 × *vector length* = 2048 registers, so 4 vector threads can fit on one core at a time. This is important in hiding memory latency.

Our algorithm uses little of local memory — up to 7.5% and 28% of the resource at full load in $AB^T$ and $AB$ respectively. This is an example of efficient usage of registers as the primary scratch space.

The performance of the code is highly compiler-dependent. The performance cited was obtained using the compiler in CUDA SDK 1.1. Code produced by the previous compiler version performs substantially slower. Code produced when compiling on 64-bit Linux runs ~10% slower unless forced to compile into 32-bit with compiler option "`-m 32`".

### 4.3 Optimization for Small Matrices

Thread-level parallelism may be not sufficient to hide memory latencies if matrix $C$ is small. For example, at least 64 vector threads are required to achieve full-occupancy on 16-core GPUs. This corresponds to 256×256 or 1024×64 dimensions of $C$. Such matrices may be important in some applications, such as in Crout version of the LU decomposition. Importance of this issue is likely to grow with time, as future processors may have many more loosely connected cores.

In this case we might wish to extract another level of parallelism available in matrix multiply — in the dot products that define entries of the output matrix: $C_{ij} = \alpha\Sigma A_{ik}B_{kj} + \beta C_{ij}$. We split the dot product into partial sums that are computed with different threads. The partial sums are summed up and merged with matrix $C$ using a different kernel. In our prototype implementation, the optimal number of partial sums is found using brute force search for every particular matrix dimension. This algorithm consumes extra bandwidth by keeping intermediate data in the off-chip memory and costs another kernel launch overhead. But it may worth it as we shortly see.

### 4.4 Performance Analysis

To estimate the performance of the algorithm, we analyze the disassembler output (*decuda*). The inner loop of the $C := \alpha AB + \beta C$ ($C := \alpha AB^T + \beta C$) program has 312 (83) instructions, 256 (64) of which are multiply-and-adds with operands in local memory and 4 (6) instructions are half-width (i.e instruction code is 32-bit wide; other instruction codes are 64-bit wide). Assuming that multiply-and-adds have a throughput of 6 cycles per warp, half-width instructions take 2 cycles and all other operations take 4 cycles, we estimate the asymptotic Gflop/s rate. The estimated and observed rates are listed in Table 2. There is a consistent underestimate within 2–4%.

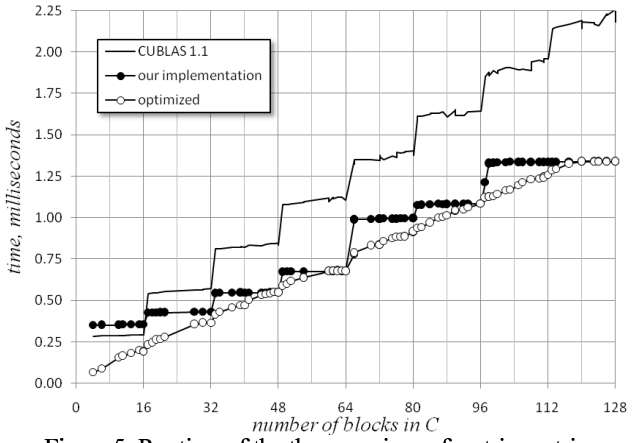


Figure 5: Runtime of the three versions of matrix-matrix multiply run on 8800GTX: one in CUBLAS 1.1, our algorithm and the prototype optimized for small $C$. In all cases the shared dimension of $A$ and $B$ was 1024. In our implementation, blocks in $C$ are 64×16, in CUBLAS it is 32×32.

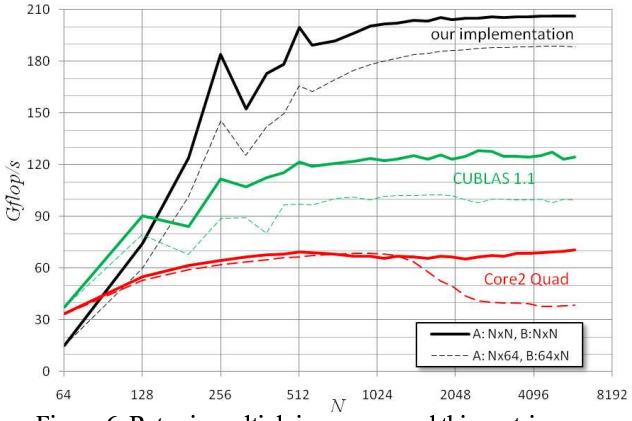


Figure 6: Rates in multiplying square and thin matrices on 8800GTX and 2.4GHz Core2 Quad.

To check that our code is not bound by the memory bandwidth and latency in reading $A$ and $B$, we substituted assigning a value at the pointer with assigning the pointer value itself in all these reads. This preserves the dependence pattern and produces similar machine code as we checked with the disassembler. To make feasible assigning pointers to floating point variables in CUDA we also substitute pointers to $A$ and $B$ with floating point indices. Table 2 shows that performance is within 3% of the original, i.e. memory bandwidth or latency do not factor in the performance in a substantial degree.

Another test version of the code does not use prefetching. In the case of the $AB^T$ version this amounts to substituting copies

before the first local barrier as in Fig. 4 with appropriate memory accesses. In the case of the *AB* version, this means that all fetches of *A* are placed after the first local barrier to avoid extra overlap in latencies. According to the table, these codes run 7–11% slower indicating the effectiveness of using prefetching in hiding the latency.

On all four GPUs the code performed at 89–90% of the peak performance with participation of local memory, or 59–60% of the peak arithmetic performance. This means it scales well with the number of cores and clock rate, corresponds to 11–14 Gflop/s per core. Substantially faster code could be built only if using local memory less intensively.

Future GPUs might have faster access to local memory. If accessing local memory in multiply-and-add did not involve extra cost and latency was still well-hidden, a similar performance estimate promises ~80% of arithmetic peak bound by ~20% of instructions spent in pointer arithmetic, fetch instructions, synchronization and flow control. For comparison, `sgemm` in Intel MKL 10.0 run on a 2.4GHz Core2 Quad runs at 70 Gflop/s, which is 91% of the arithmetic peak.

64×16 blocking yields 25.6× reduction of bandwidth consumption. Thus, 206 Gflop/s achieved on 8800 GTX corresponds to 32 GB/s in reading matrices *A* and *B*, which is 43% of the peak sustained bandwidth. In contrast, some of the earlier methods, such as Govindaraju et al. [2006] and Fatahalian et al. [2004] are bandwidth-bound.

Fig. 5 plots runtime of the algorithm versus the number of blocks in *C*, i.e. the number of vector threads. Step-pattern is due to the round-robin distribution of vector threads across the GPU cores. This produces poor load balance when number of threads is small. The runtime of the two-stage version optimized for small matrices grows nearly linearly with the amount of the work done as it creates many more threads than the number of blocks in *C*. Fig. 6 summarizes the performance of our algorithm for square and skinny matrices compared to CUBLAS 1.1 and matrix multiply as implemented in Intel MKL 10.0 and run on 2.4GHz Core2 Quad Q6600 running 64-bit Linux.

The version of the code that deals with a square block triangular matrix *C* and full matrices *A* and *B* runs at the same peak rates as for a full matrix *C*. However, a simpler solution based on more threads some of which exit early runs at 10% lower peak and was ~30% slower for some matrix sizes. Thus, creating large number of empty threads may have a substantial impact on the overall performance.

## 5 Implementation of One-Sided Matrix Factorizations

We consider the factorization of matrices that reside in the CPU memory in column-major layout, and whose factorizations overwrite the original data. The intention is to match the semantics of LAPACK routines [Anderson et al. 1990]. However, for the purpose of this study we restrict our attention to square matrices whose dimension is a multiple of the block size used.

There are three classical bulk-synchronous variants of LU factorization — left-looking, right-looking and Crout [Dongarra et al. 1998]. We dismiss the left-looking scheme as it does about half its flops in triangular solves with small number of right-hand sides and so has limited inherent parallelism. We prefer the right-looking algorithm to the Crout algorithm because it exposes more thread-level parallelism in the calls to matrix-matrix multiply. Cholesky and QR factorizations work in the same manner — the entire matrix is updated as soon as next block column is available.

Panel factorization is done on the CPU as done independently by Barrachina et al. [2008] and Baboulin et al. [2008]. However, in our implementation triangular solve in Cholesky is also done on the CPU (we are most interested in better performance at large problem sizes). The panel factorization is overlapped with computation on the GPU using a look-ahead technique (see e.g. Dongarra and Ostrouchov [1990] who call it pipelined updating). This requires transferring matrix panels from the GPU to CPU memory and back. The transfers are currently not overlapped with the computation on the GPU, as our best GPUs (8800GTX and FX560) do not permit it (unlike the newer GPUs).

To avoid extra overhead in the transfers, the panels are placed into their final output location when transferred to the CPU memory. Thus panel factorization produces the final results for those locations, except for LU factorization, which requires pivoting of the entire matrix at each panel factorization, which is done on the GPU. The transfer of the triangular matrix in the Cholesky factorization is done by transferring a set of rectangular blocks that includes the triangular part. The width of the blocks is optimized using the performance model presented in Section 3.2.

To avoid the severely penalized strided memory access in pivoting on the GPU, the matrix is laid out in the GPU memory in row-major order. This involves extra overhead for the transposition and applies to LU factorization only. The transposition of the square matrix is done in-place to avoid extra space requirements (a slightly more complicated solution may be used with non-square matrices). When the panel is transferred to the CPU memory and back, it is transposed on the GPU using an additional, smaller, buffer. Pivoting kernel does 64 row interchanges per call to amortize the kernel launch overhead. The pivot indices are passed in as function parameters that are accessible via local memory in CUDA. This avoids any memory access overhead in reading them.

Only the lower triangular part of the output of the panel factorization in the QR algorithm is needed for the update done on the GPU. It is filled in with zeros and a unit diagonal to create a rectangular matrix, so that it can be multiplied using a single matrix-matrix multiply. A similar technique is used in ScaLAPACK [Choi et al. 1996]. The same technique is used with the small triangular matrix that arises in the panel factorization in QR. These fill-ins are done on the CPU to overlap with the work on the GPU.

Instead of running triangular solve in the LU decomposition we run matrix-matrix multiply with the inverse of the triangular matrix. The inverse is computed on the CPU. Unlike other optimizations, this may affect the numerical stability of the algorithm. However, our numerical tests so far show no difficulty and in fact the stability of either algorithm depends on the essentially the same assumption, namely that $L^{-1}$ is not too large in norm, since this bounds both pivot growth and the accuracy of the triangular solve. In the future we might revert to triangular solve when $\|L^{-1}\|$ is too large.

The block size used is the same as in the matrix multiply (64). A larger block size could reduce bandwidth consumption and improve performance with large matrices. We address the bandwidth consumption using two techniques.

The first technique is a variant of 2-level blocking (this was independently done in [Barrachina et al. 2008]). Both levels are done in the right-looking manner to use a large number of threads in the matrix multiply. A novel tweak is that we switch to the coarse blocking level when only half the finer level is complete. This avoids updating matrices that have too few block columns and so offer little thread parallelism in the matrix multiplies. Note, that this approach is not a subset of the traditional recursive blocking.

A different technique is used in QR factorization, which has

a different structure of updates. We used autotuning to choose best block size (multiple of 64) at every stage of the algorithm. Each stage is parameterized with a 3-tuple: the size of the trailing matrix, the block size used in panel factorization and the block size used in the update (same as used in the last panel factorization). In the current prototype we measure the runtime for every instance of this 3-tuple within the range of interest. Dynamic programming is then used to choose the best sequence of the block sizes, similarly to [Bischof and Lacroute 1990]. Block triangular matrix multiplies are used wherever possible.

### 5.1 LU factorization on two GPUs

We consider using two GPUs attached to the same workstation. We use a column-cyclic layout to distribute the matrix over two GPUs. It is convenient, as it does not require communication in pivoting, distributes the workload evenly and keeps CPU-GPU data transfers simple. Each GPU sees only its own fraction of the matrix (even or odd columns). The exception is the updates, which require the transfer to each GPU of an entire panel. The columns that do not belong to the layout are discarded after the update is complete. The structure of the algorithm is same as in the single-GPU case but no 2-level blocking is currently implemented as it requires extra space.

## 6 Results

All single-GPU results in this section are given for a desktop system that has one 2.67GHz Core2 Duo E6700 and one GeForce 8800 GTX running Windows XP. Two-GPU results are given for the same system with two GeForce 8800 GTXs. We also compare results with one 2.4GHz Core2 Quad Q6600 running 64-bit Linux. In all cases the Intel MKL 10.0 library is used for factorizations on the CPU and CUDA SDK 1.1 for programming the GPU.

Input and output data are in the pinned CPU memory, which provides a compromise between usefulness in applications (that are likely to run on the CPU) and performance (slower transfers to/from GPU if the data is in pageable memory). The cost of the memory allocation is not included in the timings.

The correctness of the algorithms is tested in the following way. Input matrix $A$ is synthesized with random entries uniformly distributed in [−1,1] (to guarantee symmetric positive definiteness, $A = 0.001 \cdot I + X^T X$ is used instead in testing the Cholesky factorization, where $X$ is the random matrix as described above and $I$ is the identity matrix). Output factors are multiplied and max-norm of its difference with the input matrix is found. This measures the backward error in the factorization. We found that this error is about the same whether using our GPU-based algorithm or the purely CPU-based algorithm in the Intel MKL (always within a factor of 2, and within 20% in most cases). The variant of the LU factorization that multiplies by the inverses of the diagonal blocks of the triangular matrix has shown about same accuracy as when running triangular solves on the GPU. As an example, the errors as measured above in LU, QR and Cholesky at $n = 8192$ are about $2000 \cdot \varepsilon \|A\|_{max}$, $200 \cdot \varepsilon \|A\|_{max}$ and $17 \cdot \varepsilon \|A\|_{max}$ resp., where $\varepsilon = 2^{-23}$ is machine epsilon in IEEE single precision and $\|A\|_{max}$ is the max-norm of $A$.

### 6.1 Summary of Performance

Figure 7 shows the Gflop/s rates sustained in the GPU-based matrix factorization routines and their BLAS3 calls. Redundant flops, such as when multiplying by the zero fill-ins in the triangular matrices, are not counted. Operations with block triangular matrices are also counted as BLAS3, although they do not strictly match the semantics. One can see that BLAS3 rates

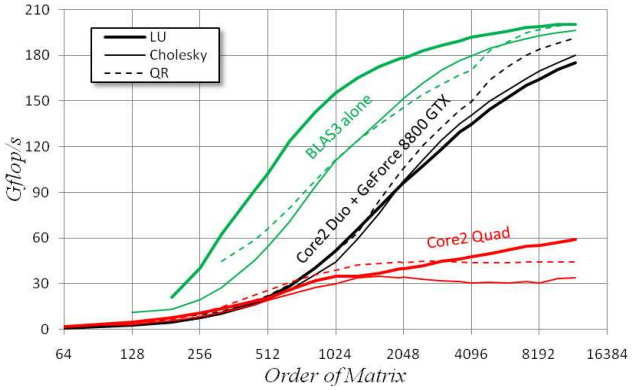


Figure 7: Rates achieved in the factorizations using Core2 Duo with GeForce 8800GTX (black), using Core2 Quad alone (red) and in the BLAS3 operations on the GPU as used in the factorizations (green).

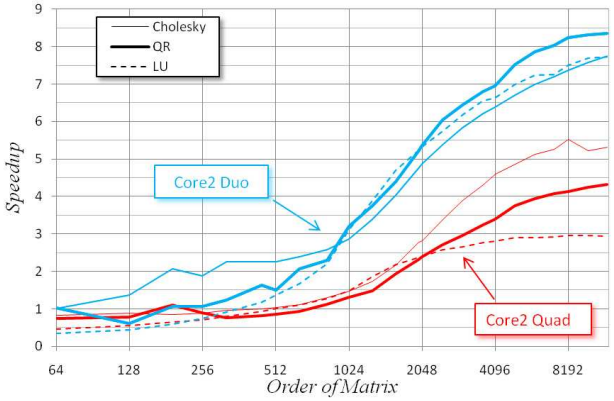


Figure 8: Speedup versus CPU-alone versions.

| | 8800GTX Gflop/s | Core2 Duo | | Core2 Quad | |
|---|---|---|---|---|---|
| | | Gflop/s | speedup | Gflop/s | speedup |
| Cholesky | 183 | 23.0 | 7.4× | 34.9 | 5.5× |
| LU | 179 | 22.7 | 7.7× | 59.2 | 3.0× |
| QR | 192 | 22.5 | 8.3× | 44.8 | 4.3× |
| sgemm | 206 | 25.9 | 8.0× | 69.8 | 3.0× |

Table 3: Comparison of best Gflop/s rates in the GPU version and the two CPU-alone versions. The speedups shown are the best speedups vs. CPU-alone versions that were observed for some $n$.

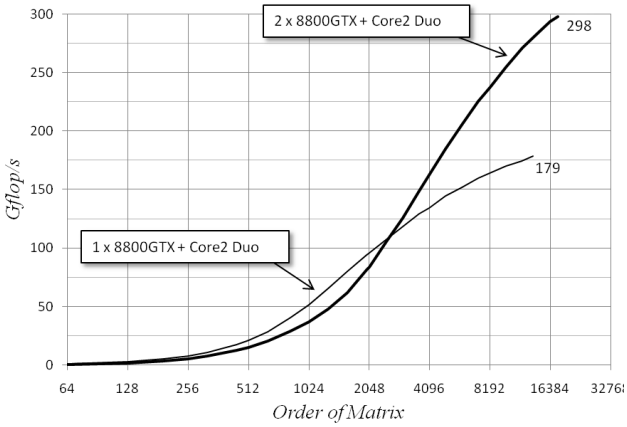


Figure 9: Performance of one-GPU and two-GPU versions of the LU decomposition.

approach peak rates of the matrix-matrix multiply presented in Section 4.4. Performance of BLAS3 bounds performance of entire factorization as it's the fastest component of the algorithm. The gap between BLAS3 and factorization rates illustrates the impact of the other work done. The gap is the largest in the case of LU decomposition, which requires pivoting, transposition and transferring the final result back to the CPU unlike the two other factorizations.

The same plot also includes the rates achieved in the factorizations done using Core2 Quad alone, and Figure 8 details the speedups vs. Core2 Duo and Core2 Quad. According to the Figure, the crossover between the GPU-based and CPU-alone implementations is at $n = 200–400$ for LU and QR algorithms. Cholesky always runs faster than the CPU-alone implementation on the Core2 Duo (which might indicate inefficiency of the CPU implementation). Crossover with the performance of the CPU-alone algorithm running on Core2 Quad is at $n = 500–700$. The best performances are summarized in Table 3. It shows that the speedup is nearly the same as the speedup in the matrix-matrix multiply (`sgemm`) and even better when comparing to the Core2 Quad.

Finally, Fig. 9 shows the performance of the LU decomposition that achieves ≈ 300 Gflop/s at $n \approx 18{,}000$ by running two GPUs in parallel.

## 6.2 Performance Analysis

The different rates in the BLAS3 routines in Fig. 8 are due to different amounts of the thread level parallelism (TLP) exposed in the bulk matrix multiplies. Right-looking LU decomposition exposes the most TLP, and right-looking Cholesky exposes less, as it runs similar updates but for triangular matrices that have about half as many blocks (this may explain why the Cholesky curve looks similar to the LU curve shifted right by factor of two). Around half of the BLA3 operations in QR factorization are involved in producing a skinny matrix, thus running slower than in BLAS3 in LU factorization. BLAS3 in QR speeds up for larger $n$ as adaptive blocking switches to less skinny matrices. However, QR factorization achieves the overall highest speed among the three factorization as it does more flops in matrix multiplies than any other algorithm.

Fig. 10 shows the breakdown of runtime in the LU factorization. The breakdown for Cholesky looks similar, but does not have the transpose time, the CPU-GPU transfer takes less time (no transferring the entire matrix back) and overlap between CPU and GPU is not as good due to less work done in matrix-matrix multiply (the panel factorization has about same cost). We did not do this analysis for QR but expect the work done on the CPU to be hidden better than in LU. The breakdown shows that up to 90% of the runtime is consumed by computing on the GPU and about of 10% of this time overlaps with computing on the CPU. It reveals a potential use of asynchronous transfers that could save another 10% of time, if available in hardware. Another speedup is possible with offloading more work to the CPU as currently up to 80% of time GPU works alone. Time spent in transposing the matrices is not substantial. Individual measurements have shown that transpose runs at 25–45 GB/s for $n > 1000$. This variation in bandwidth is due to the moderate granularity of this operation. For example, it takes ~7.5μs to copy or transpose a 1024×64 matrix at 70 GB/s, which is about the same as the kernel launch overhead. CPU-GPU transfers run at 3.0–3.3GB/s for $n > 1000$, which approaches the peak sustained bandwidth as in Section 3.2.

Fig. 11 evaluates the impacts of different optimizations used. The most important optimization was using row-major layout on the GPU. If not used, we lose about half of the Gflop/s rate. We measured pivoting individually and found that it takes

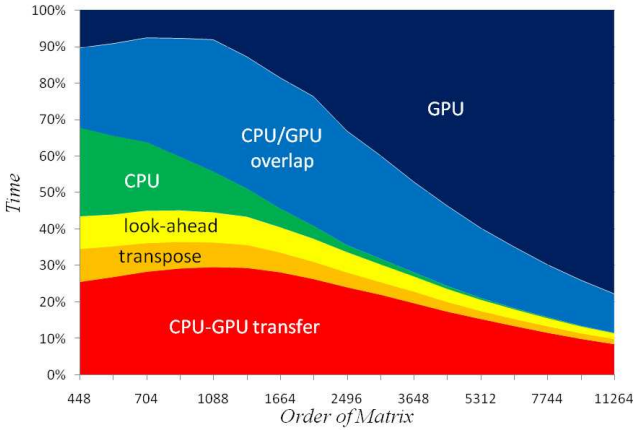


Figure 10: The breakdown of time in the LU decomposition.

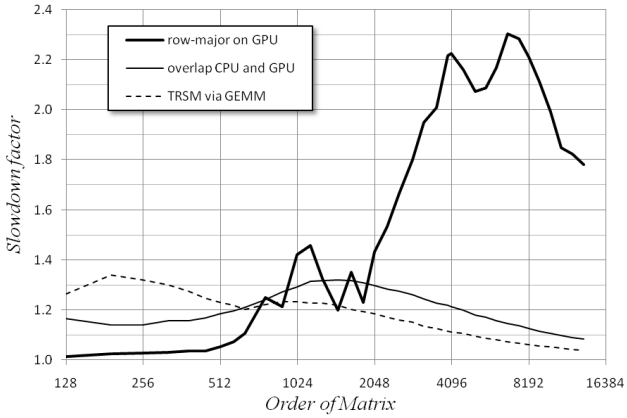


Figure 11: Slowdown when omitting one of the optimizations used.

1–10% of time in the entire computation for $n > 1000$ if done in the row-major layout. In that case it achieves 7–17 GB/s of effective bandwidth. When using column-major layout, it can take up to 56% of total time and run at 0.2–2GB/s, with slower rates for larger matrices.

A surprisingly large speedup (up to 30%) was obtained by performing triangular solve via multiplying by the inverse matrix. Triangular solve with a 64×64 triangular matrix and 4096 right hand sides runs at 13 Gflop/s when using CUBLAS 1.1 or CUBLAS 2.0 beta. It is an order of magnitude slower than the 160 Gflop/s rate achieved in multiplying a 64×64 matrix by a 64×4096 matrix that does the same work (this is 80 Gflop/s if not counting the redundant work).

The best speedups by using autotuning to choose block sizes in QR and in 2-level schemes in LU and Cholesky were 7%, 6% and 4% correspondingly and factored in only for $n > 4096$.

According to Fig. 9, using two GPUs yields only ~67% improvement in the Gflop/s rate. Benchmarks of CPU-GPU transfers showed only 1.6 GB/s peak in the transfers with the second GPU. Apparently it is working at PCI-e ×8 rate, we do not know if this can be remedied. As we perform factorization in a bulk-synchronous manner, the runtime is bound by the slowest transfer. There are other sources of slowdown, such as extra CPU-GPU bandwidth consumption, lack of 2-level blocking and keeping the number of CPUs constant.

## 6.3 Comparison with Other Work

The first implementation of the LU factorization using GPUs that we know was published by Galoppo et al. [2005] and ran at

up to ~10 Gflop/s for $n = 4000$ without pivoting and at ~6 Gflop/s for $n = 3500$ with partial pivoting on the older GeForce 7800. This is 13× and 22× slower than our implementation done with partial pivoting at these matrix dimensions. As discussed above, in our implementation pivoting does not introduce as much slowdown.

Barrachina et al. [2008] report 50 Gflop/s in LU factorization and 41 Gflop/s in Cholesky factorization for $n = 5000$ using CUBLAS 1.0 on GeForce 8800 Ultra. This GPU is faster than the 8800 GTX that we use (6.4% higher processor clock rate and 11% higher pin-bandwidth). Our implementation achieves 2.9× and 3.7× higher speed for LU and Cholesky resp. This is due to our improved matrix-matrix multiply routine and the optimizations evaluated above.

Baboulin et al. [2008] describes implementation of LU and QR algorithms that run at up to ≈55 Gflop/s on Quadro FX5600 for $n \approx 19{,}000$ using CUBLAS 1.0. This GPU is similar to what we use (see Tables 1 and 2). Their implementation of Cholesky runs at up to 90 Gflop/s if using CUBLAS and approaches 160 Gflop/s if using an early version of the matrix multiply described in this paper and offloading BLAS1/BLAS2 operations to the CPU. Our implementation achieves higher rates at smaller orders of matrix, thus is more efficient.

Castillo et al. [2008] report results for Cholesky factorization run on 4-GPU NVIDIA Tesla S870. Each of these GPUs is roughly equivalent to Quadro FX5600. Authors report 180 Gflop/s on a system at $n \approx 10{,}000$. We achieve this performance using one GPU only. Their result was later improved to 424 Gflop/s at $n \approx 20{,}000$ by using matrix multiply routine presented in this paper [Quintana-Orti et al. 2008].

## 7 Conclusions

We have presented the fastest (so far) implementations of dense LU, QR and Cholesky factorizations running on a single or double NVIDIA GPUs. Based on our performance benchmarking and modeling, they attain 80%–90% of the peak speeds possible for large matrices. This speed was achieved by carefully choosing optimizations to match the capabilities of the hardware, including using the CPU in parallel with the GPU to perform panel factorizations, which are dominated by BLAS1 and BLAS2 operations done faster on the CPU. We also changed the LU algorithm to use explicit inverses of diagonal subblocks of the $L$ factor, and showed this was both faster than and as numerically stable as the conventional algorithm.

We also presented detailed benchmarks of the GPU memory system, kernel start-up costs, and barrier costs, which are important to understanding the limits of performance of many algorithms including our own. We also identified a new way to do global barriers faster, but which may or may not provide memory consistency.

Future work includes designing two-sided factorizations, such as in dense eigenvalue problems, one-sided factorizations on a GPU cluster and exploring the new performance opportunities offered by newer generations of GPUs.

## References


ABTS, D., BATAINEH, A., SCOTT, S., FAANES, G., SCHWARZMEIER, J., LUNDBERG, E., JOHNSON, T., BYE, M., AND SCHWOERER, G. 2007. The Cray BlackWidow: A Highly Scalable Vector Multiprocessor, *SC'07*.

AGARWAL R. C., AND GUSTAVSON, F.G. 1989. Vector and parallel algorithms for Cholesky factorization on IBM 3090, *Supercomputing'89*, 225–233.

ELVERSON, R., CALLAHAN, D., CUMMINGS, D., KOBLENZ, B., PORTERFIELD, A., AND SMITH, B. 1990. The Tera Computer System, *ICS'90*, 1–6.

AMD. 2006. *ATI CTM Guide, version 1.01*.

ANDERSON, E., BAI, Z., DONGARRA, J., GREENBAUM, A., MCKENNEY, A., DU CROZ, J., HAMMERLING, S., DEMMEL, J., BISCHOF, C., AND SORENSEN, D. 1990. LAPACK: a portable linear algebra library for high-performance computers, *Supercomputing'90*, 2–11.

ANDERSON, E., BRANDT, M., AND YANG, C. 2004. LINPACK Benchmark Optimizations on a Virtual Processor Grid, In *Cray User Group 2004 Proceedings*.

BABOULIN, M., DONGARRA J., AND TOMOV, S. 2008. Some Issues in Dense Linear Algebra for Multicore and Special Purpose Architectures, Technical Report UT-CS-08-200, University of Tennessee, May 6, 2008 (also LAPACK Working Note 200).

BARRACHINA, S., CASTILLO, M., IGUAL, F. D., MAYO, R, AND QUINTANA-ORTI, E. S. 2008. Solving Dense Linear Systems on Graphics Processors, Technical Report ICC 02-02-2008, Universidad Jaime I, February 2008.

BISCHOF, C. H., AND LACROUTE, P. G. 1990. An adaptive blocking strategy for matrix factorization, in *Proceedings of the joint international conference on Vector and parallel processing*, 210–221.

CASTILLO, M., CHAN, E., IGUAL, F. D., MAYO, R., QUINTANA-ORTI, E. S., QUINTANA-ORTI, G., VAN DE GEIJN, R., AND VAN ZEE, F. G. 2008. Making Programming Synonymous with Programming for Linear Algebra Libraries, FLAME Working Note #31. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-08-20. April 17, 2008.

CHOI, J., DONGARRA, J. J., OSTROUCHOV, L. S., PETITET, A. P., WALKER, D. W., AND WHALEY, R. C. 1996. The Design and Implementation of the ScaLAPACK LU, QR, and Cholesky Factorization Routines, *Scientific Programming 5*, 3, 173–184 (also LAPACK Working Note 80).

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

DONGARRA, J., AND OSTROUCHOV, S. 1990. LAPACK Block Factorization Algorithms on the Intel iPSC/860, Technical Report CS-90-115, University of Tennessee (also LAPACK Working Note 24).

GALOPPO, N., GOVINDARAJU, N. K., HENSON, M., AND MANOCHA, D. 2005. LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware, *SC'05*.

GOVINDARAJU, N. K., LARSEN, S., GRAY, J., AND MANOCHA, D. 2006. A Memory Model for Scientific Algorithms on Graphcs Processors, *SC'06*.

FATAHALIAN, K., SUGERMAN, J., AND HANRAHAN, P. 2004. Understanding the efficiency of GPU algorithms for matrix-matrix multiplication, In *Graphics Hardware 2004*, 133–137.

HE, B., GOVINDARAJU, N. K., LUO, Q., AND SMITH, B. 2007. Efficient Gather and Scatter Operations on Graphics Processors, *SC'07*.

NVIDIA. 2006. NVIDIA GeForce 8800 GPU Architecture Overview, Technical Brief, November 2006.

NVIDIA. 2007. NVIDIA CUDA Compute Unified Device Architecture, Programming Guide, v. 1.1.

QUINTANA-ORTI, G., IGUAL, F. D., QUINTANA-ORTI, E. S., AND VAN DE GEIJN, R. 2008. Solving Dense Linear Systems on Platforms with Multiple Hardware Accelerators, FLAME Working Note #32. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-08-22. May 9, 2008.