Benchmarking GPUs to Tune Dense Linear Algebra

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 recent NVIDIA GPUs. Our matrix-matrix multiply routine (GEMM) runs up to 60% faster than the vendor's implementation 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 ~540 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 over 300 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 [Anderson et al. 1990] for the GPUs.

Our results also include performance on the 8-series of NVIDIA GPUs that was not previously attained in the 1.5 years since these GPUs were available. We provide new insights into programming these and newer GPUs that help us achieve performance in such basic kernels as matrix-matrix multiply that is 60% faster than those in the optimized vendor's library CUBLAS 1.1. Some of our codes have been licensed by NVIDIA and included in CUBLAS 2.0. In our approach we think of the GPU as a multithreaded vector unit and our best algorithms were found to closely resemble earlier solutions found for vector processors.

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 overhead 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. For the first time we implement and measure the performance of a global barrier that runs entirely on the GPU. We believe this is an important step towards operating GPUs with lower CPU intervention.

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 quad core CPU are over 4× in all three factorizations.

The rest of this paper is organized as follows. Section 2 describes the architecture of the GPUs we used, highlighting the features common to vector 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 GTX280	GeForce 9800GTX	GeForce 8800GTX	GeForce 8600GTS
# of vector cores	30	16	16	4
core clock, GHz	1.30	1.67	1.35	1.45
registers/core	64KB	32KB	32KB	32KB
smem/core	16KB	16KB	16KB	16KB
memory bus, GHz	1.1	1.1	0.9	1.0
memory bus, pins	512	256	384	128
bandwidth, GB/s	141	70	86	32
memory amount	1GB	512MB	768MB	256MB
SP, peak Gflop/s	624	429	346	93
SP, peak per core	21	27	22	23
SP, flops:word	18	25	16	12
DP, peak Gflop/s	78	—	—	—
DP, flops:word	4.4	—	—	—

Table 1: The list of the GPUs used in this study. SP is single precision and DP is double precision. Smem is shared memory. Peak flop rates are shown for multiply and add operations. Flops:word is the ratio of peak Gflop/s rate to pin-memory bandwidth in words.

2 GPU Architecture

In this work we are concerned with programming 8 series, 9 series, and 200 series of NVIDIA GPUs, as listed in Table 1. For the description of their architecture see the CUDA programming guide [NVIDIA 2008a], technical briefs [NVIDIA 2006; NVIDIA 2008b] and lecture slides in the course on programming GPUs at the University of Illinois, Urbana-Champaign [Hwu and Kirk 2007]. Additional insights can be found in *decuda*[1], which is a third-party disassembler of GPU binaries based on reverse-engineering of the native instruction set. The instruction set called PTX that was released by vendor is an abstraction that requires further compilation and so provides fewer insights.

2.1 Notation

The GPU programming model used in the CUDA programming environment [NVIDIA 2008a] borrows much from abstractions used in graphics, e.g. such as used in the DirectX and OpenGL standards. GPU programs are run as collections of scalar threads that run faster if they remain convergent in an SIMD fashion. Similarly, individual arithmetic pipelines that execute scalar instructions are exposed as individual processing cores. For example, the technical brief on the latest GPU [NVIDIA 2008b]

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

SC2008 November 2008, Austin, Texas, USA 978-1-4244-2835-9/08 $25.00 ©2008 IEEE

details 240 “scalar processing cores” and 30 ”double-precision 64-bit processing cores”.

We seek a more traditional exposition of the GPU architecture and attempt one below.

The CUDA manual introduces groups of 32 parallel scalar threads called “warps”. “*A warp executes one common instruction at a time*” [NVIDIA 2008a, Ch. 3.1] is another way of saying that warp is a stream of vector instructions. Scalar threads are then vector elements. Similarly to other vector architectures and unlike SIMD extensions such as Intel’s SSE, a particular value of the vector length (*VL*) is not specified at the ISA level. Instead, it can be queried in the runtime [NVIDIA 2008a, Ch. 4.2.4.5]. So, a GPU program compiled once will run on GPUs with different vector lengths.

CUDA defines “multiprocessor” as a cluster comprised of one instruction issue unit, 8 single precision MAD pipelines (called SP), 2 transcendental function units (called SFU), 1 double precision MAD pipeline (we call it DP) and a 16KB local store also called shared memory. In this collection of functional units we recognize what is usually called “a core”. E.g. one core in Intel Core2 architecture also has one instruction unit, many arithmetic pipelines and can execute multiple scalar operations each cycle. We use term “vector core” to emphasize that GPU cores have no scalar capabilities.

Those arrays of scalar arithmetic units on each core can be understood as multi-lane arithmetic units that are common in vector architectures. The purpose of replicating lanes is to increase throughput. So fully pipelined SP, SFU and DP units have throughput of 4, 16 and 32 clocks per instruction respectively.

A “thread block” is defined as a collection of warps that run on the same core and share a partition of local store. The number of warps in the thread block is configurable. As a thread block is usually operated in an SPMD fashion, for programming purposes it can be considered as a single thread of vector instructions. Thread block size in this case is the programmable vector length.

2.2 Strip Mining on the GPU

Partitioning of long vectors into warps by the GPU environment corresponds to strip mining into independent instruction streams. This is an alternative to the more traditional strip mining into independent instructions in the same instruction stream. For example, an operation on a 512-element vector on a machine with $VL = 32$ is traditionally performed as 16 independent vector instructions. The GPU allows (but not requires) distributing these 16 independent instructions across 16 instruction streams. This is done to improve performance in branching — associating an individual program counter with a short subset of a long vector allows skipping branches not taken by this subset rather than masking them off.

However, strip mining into independent instruction streams is expensive as it requires replicating register data across all instruction streams in the thread. For example, a program operating on 512-element vectors consumes 2KB of register file per every pointer, temporary value or scalar value defined in the scalar thread as a 32-bit register variable.

Another associated overhead is the partitioning of the register data into private register spaces associated with different instruction streams in the thread. Accessing the data residing in the register space of another warp requires staging it via the local store, which incurs costs.

Note that the number of independent instructions supplied by a program does not depend on the kind of strip mining used. Whether independent instructions come in the same or different streams, they hide memory and pipeline latencies.

To summarize, one should use as short vectors as possible to avoid the extra costs associated with spreading the data across many warps. See Section 3.6 for the evaluation of the minimum vector length that does not compromise throughput and Section 4.3 for the comparison of long-vector and short-vector implementations of the same routine.

2.3 Using Register File vs. Using Shared Memory

The largest and the fastest level of the on-chip memory hierarchy is the register file. It provides 32–64KB space per core and up to 1.9MB space for the entire chip. Register-to-register instructions achieve the peak instruction throughput if the vector length is large enough.

Shared memory is a smaller and slower level of the on-chip memory hierarchy. Only 16KB of shared memory is provided per core and only one shared memory operand is allowed per instruction (according to *decuda*). For example, at least two instructions are required to copy data from one location in shared memory to another. Furthermore, instructions that use operands in shared memory may run at lower throughput as found in Section 3.7.

If the algorithm permits, shared memory should be used less intensively in favor of using the register file.

3 Microbenchmarks

3.1 Kernel Launch Overhead

The minimum time to asynchronously invoke a GPU kernel using either the low-level or the high-level CUDA API was 3–7 μs across a variety of systems equipped with different GPUs, operating systems and CUDA versions. This was measured by asynchronously invoking the same kernel a very large number of times and synchronizing once at the end. The program used was the simplest possible, such as copying one word from one location in the GPU memory to another. This ensures that the program runtime does not contribute substantially to the overall time. The time increases to 10–14 μs when synchronizing at each kernel invocation. This shows the expense of synchronization.

To ensure that we do not sacrifice performance by choosing CUDA for programming the GPU we also measured the overheads in DirectX 9.0c, which is a mature graphics API widely used in computer games. The timings were 7–8 μs for invocation alone and 20–23 μs for invocation with synchronization (synchronization is required when computing with DirectX to ensure correctness, but not in CUDA). This indicates that CUDA is as efficient as or better than DirectX in terms of the launch overhead.

3.2 CPU-GPU Data Transfers

Our primary system is equipped with a PCIe 1.1 ×16 interface that bounds the bandwidth of the CPU-GPU link by 4 GiB/s. We found that transferring contiguous pieces of data with sizes from 1 byte to 100 MB long across this link using pinned memory takes about

$$Time = 11\mu s + \frac{bytes\ transferred}{3.3GB/s}. \quad (1)$$

This fits the measured data within a few percent. Similar fitting on other systems yielded similar accuracy with different numbers, such as 10–17 μs overheads and 2.2–3.4 GB/s bandwidths.

When using two GPUs in the system, transfer to the second GPU run only at up to 1.8GB/s, i.e. about what one may expect from PCIe 1.1 ×8. This result was obtained when using various ×16 slots in the nForce 680i SLI motherboard.

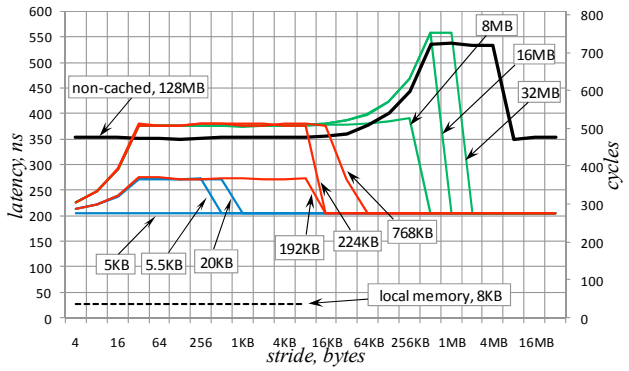

Figure 1: Memory latency as revealed by the pointer chasing benchmark on GeForce 8800 GTX for different kinds of memory accesses. Array size is shown in the boxes. Cached access assumed unless otherwise specified. Blue, red and green lines highlight 5KB cache, 192 KB cache, and 512KB memory pages respectively.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

Operating with two GPUs concurrently poses new difficulties. CUDA requires attaching each CPU thread to a fixed GPU context, so multiple CPU threads must be created. According to our experience, pinning of memory is effective only with the GPU context that performed the memory allocation. Other GPU contexts perform at non-pinned rates when operating with this memory space. So, if two GPU contexts run transfers across the same main memory locations, at least one of the contexts will run at the non-pinned transfer rate, which is about 2× lower.

Benchmarks on a few different machines with PCIe 2.0 ×16 have shown 3.9–6.1 GB/s transfer rates.

3.3 GPU Memory System

The vendor's manuals supply limited information on the GPU caches. The CUDA programming guide specifies an 6–8KB cache working set per vector core [NVIDIA 2008a, Ch. A.1], i.e. 96–128KB for the entire 8800GTX 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, some of the vendor's manuals detail one L1 cache per two cores and six L2 caches on 8800GTX [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 shared 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 8800GTX 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

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

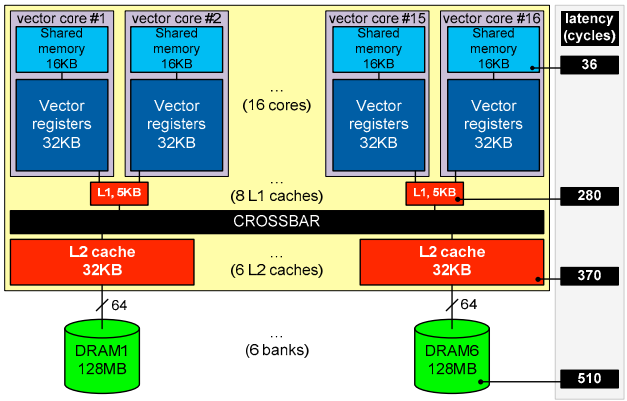

Figure 2: Summary of the memory system of 8800GTX according to our study. Sizes of the on-chip memory levels are shown in the same scale. Latencies shown are for the cached access. Note the small L1 caches and large register files.

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, the working set decreases until it again fits in the cache, this time producing conflict misses if the 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.

Fig. 1 also reveals a 470–720 cycle latency non-cached memory access that roughly matches the official 400–600 cycle figure [NVIDIA 2008a, 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 shared memory per thread), each traversing through a private 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 for some other GPUs in the 8-series 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 most GPUs, but twice as much on Quadro FX5600. 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 across few GPUs in the 8-series. Note that an L1 cache hit costs about 280 cycles 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 2008a, Ch. 5.1.2.4]. Interestingly, the same purpose is followed in the design of the vector cache in the Cray BlackWi-

dow vector computer [Abts et al. 2007].

Latency to the shared memory is an order of magnitude less than to the cache — 36 cycles. We'll see shortly that it is close to the pipeline latency.

3.4 Pipeline Latency

To measure pipeline latency we execute dependent operations such as $a = a * b + c$ or $a = \log_2 |a|$ many times in an aggressively unrolled loop, one scalar thread per entire GPU. (We assume that similarly to as done on AMD GPUs [AMD 2006], taking absolute value of an argument does not require a separate instruction.) We used *decuda* to ensure that this operation maps to a single native instruction for all but double precision tests which are not supported by this tool. We made sure that arithmetic does not overflow, but assume that execution units are not optimized for special values of operands, such as 0 or 1. The following table lists the average time per instruction in cycles for GPUs in Table 1 (decimal fractions are not shown but are ≈ 0.1):

Operation	Unit	GTX280	other GPUs
$a = a + b$, $a = a * b$	SP	24	20
same w/ b is in smem		26	24
$a = a * b + c$		24	24
same w/ b is in smem		28	26
$a = log2(\|a\|)$, $a = rsqrt(a)$	SFU	28	26
$a = a + b$, $a = a * b$	DP	48	—
$a = a * b + c$		52	—

For example, the register-to-register multiply-and-add instruction runs at 24 cycles throughput per instruction. This number is 6× larger than at the peak throughput and is an estimate of the pipeline latency. 24 cycle latency may be hidden by running simultaneously 6 warps or 192 scalar threads per vector core, which explains the number cited in the CUDA guide [NVIDIA 2008a, Ch. 5.1.2.6]. Note, that 6 instruction streams is the largest number that may be required to hide this latency. Smaller number may also be sufficient if instruction level parallelism is present within the streams. This is an example where strip mining into same or independent warps makes no difference.

Latency of SP and SFU pipelines is similar. Latency of the double precision pipeline is substantially larger than in single precision. However, less parallelism is needed if overlapping it with other double precision instructions as they run at low throughput.

3.5 GPU Memory Bandwidth

The table below lists pin bandwidths and their fractions attained when copying very large blocks of data in the GPU memory. The "aligned copy" numbers in the table are the maximum over the rates achieved copying data in 32-, 64- and 128-bit words (copying in 64-bit words was often slightly faster than the rest). All other numbers in the table correspond to 32-bit words. "Misaligned" implies pointers shifted one word off the aligned address. Stride given is also in words, e.g. stride-10 means copying every 10-th word in the array.

Stream copy on all these GPUs shows a high fraction of the pin-bandwidth if data is aligned. GPUs earlier than GTX280 show strong deterioration when data is not aligned or non-unit-stride. When stride is 10 or above, all GPUs run at ~10× lower bandwidth. When stride is of order of 1000, all GPUs run at ~100× lower bandwidth.

GPU	8800GTX	8600GTS	9800GTX	GTX280
at pins, GB/s	86	32	70	141
aligned copy	89%	83%	85%	89%
misaligned	9%	10%	9%	51%
stride-2	9%	10%	9%	45%
stride-10	10%	10%	9%	10%
stride-1000	0.9%	2.1%	1.1%	1.1%

3.6 Attaining Peak Instruction Throughput

We were able to achieve 98% of the arithmetic peak in register-to-register multiply-and-add instructions. This was achieved running a single vector thread per core. In the test, each thread performs a group of 6 independent multiply-and-adds a million times in an aggressively unrolled loop. This is designed to hide the pipeline latency even at a small number of threads per core.

The smallest vector length that yielded so high a fraction of peak was 64 elements, i.e. two warps. We couldn't achieve comparable rate with shorter vectors even when running many vector threads per core.

We got similar results in double precision with 16-element vectors. 16-element vectors were also capable of filling the SFU pipeline but only if running many vector threads per core. Note that both 64 and 16 differ from the native vector length of 32.

Since most of the instructions go through the SP pipeline, 64 appears to be the optimal vector length. Although the CUDA programming guide mentions that multiples of 64 are best due to the conflicts in accessing the register file [NVIDIA 2008a, Ch 5.1.2.6], it also discourages using such short vectors saying that "192 or 256 threads per block is better" [NVIDIA 2008a, Ch 5.2].

3.7 Throughput when using Shared Memory

According to *decuda*, locations in shared memory can be used as an instruction operand. However, our benchmarks show that instructions that use such operands may run slower. In the following table a and b are in registers, $s[i]$ is a shared memory operand (it doesn't matter if it is indexed or a fixed address), and numbers are the fraction of the peak throughput:

Operation	8800GTX	other GPUs in Table 1
$a+b*s[i]$	66%	66%
$a+a*s[i]$	66%	75%
$a+s[i]$	74%	99%

Note that newer GPUs handle shared memory operands faster. However, the most generic multiply-and-add instruction runs at 66% of the peak on all GPUs, i.e. as it takes 6 cycles per warp instead of the usual 4. We use these numbers for performance modeling in Section 4.2

3.8 Global Barrier on the GPU

The CUDA programming manual assumes that a global barrier in the GPU programs should be implemented by launching a new kernel [NVIDIA 2008a, Ch. 5.5], i.e. synchronizing with the CPU. This costs at least the kernel launch overhead. An ideal global barrier would be much cheaper by communicating entirely within the GPU chip. Although there is a memory crossbar on the GPU chips, it cannot be used for communication among

cores. As an early work-around, we implemented a global barrier that runs entirely on the GPU, but not entirely on-chip. In our implementation threads running on different cores communicate via the off-chip GPU memory.

Implementation of a barrier requires atomic operations on synchronization variables. We enable this by replicating synchronization variables across the entire thread array. In that case each vector thread can update only private variables. This eliminates the race conditions. In more detail, one *arrival* and one *wakeup* variable is allocated for each vector thread. The first vector thread is assigned to be the master and others are slaves. The *i*-th slave updates the *i*-th *arrival* variable and spins on the *i*-th *wakeup* variable until it is updated. The master thread spins on the *arrival* variables until they all are updated, then updates every *wakeup* variable. This implements the barrier operation.

This barrier does not guarantee that previous accesses to all levels of the memory hierarchy have completed unless a memory consistency model is assumed.

We observed 1.3–2.0 µs per barrier in a microbenchmark on all four GPUs. This is 1.5–5.4× less than the cost of the new kernel launch that requires synchronizing with the CPU.

3.9 Implications for Codes Based on BLAS1/2

BLAS1 and BLAS2 operations are bandwidth bound on the GPU as their flop:word ratio is below the flop:word ratio of GPUs. Assuming that each operation involves an average launch overhead and all memory operations run at the sustained peak, we get the following estimate for the GTX280:

$$Time = 4\mu s + \frac{bandwidth\ required}{127GB/s} . \qquad (2)$$

For example, BLAS1's `saxpy` operation adds a multiple of one vector to another vector and so requires 3×4×*n* bytes of bandwidth per 2×*n* flops if computing in single precision. So, the flop rate is bounded by r_∞ = (2*n* flops / 12*n* bytes) × 127 GB/s = 21 Gflop/s. This bound scales linearly with memory bandwidth. However, half of this rate is achieved only at $n_{1/2}$ = 4 µs × 127 GB/s / 12 bytes ≈ 42,000. Thus, at *n* < 42,000, the operation takes 4–8 µs. The largest square matrix that fits into 1GB of the GTX280 memory is 16,384×16,384. Thus, for practical square matrix sizes `saxpy` and other BLAS1 routines take a large nearly constant time to run.

Now consider running the entire LU factorization an of *n*×64 matrix using BLAS1 and BLAS2 operations as done in LAPACK's `sgetf2` code[3]. Such factorizations are called panel factorizations and are used in the blocked factorization algorithms. Approximating the runtime of each BLAS call in `sgetf2` using (2) we get the upper bound on the performance that is plotted in Fig. 3. This figure also shows the rates sustained in the optimized GPU implementations and in the CPU-based solver that reads input data over PCIe 2.0 ×16 from the GPU, computes on the 3.0GHz Core2 Quad using 64-bit Intel MKL 10.0 and copies the result back to the GPU. In practice, this CPU-based version outperforms the fastest GPU for all *n* < 10,000. Furthermore, the newer GPU does not show substantially better performance at smaller matrices and the bounds indicate that this GPU has no potential to outperform the CPU version at *n* < 1600.

Fig. 3 compares further a hypothetical 2× improvement in the memory bandwidth and a 10× improvement in the overhead. We assume that both improvements are possible to accomplish by industry within a few years. For example, the overhead can be improved by introducing the on-chip communication. The

[3] http://www.netlib.org/lapack/single/sgetf2.f

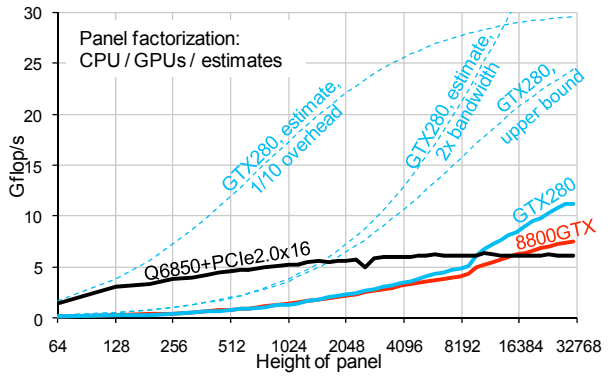

Figure 3: Gflop/s rates achieved in the LU factorization of *n*×64 matrices. Solid lines are sustained in factorizations using GPU or CPU. Dashed lines are the theoretical estimates.

plot clearly shows that the smaller overhead would make a dramatic change in the balance between the CPU and GPU performance for small problems.

4 Design of Block Matrix-Matrix Multiply Routine

Consider evaluating the 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 on-chip memory can hold one block in A, B and C 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 $bm = bn = bk = 1$. Note, that this does not depend on bk. For example, blocks in A and B don't have to be square for this technique to work.

The amount of bandwidth reduction should be at least as large as the flop:word ratio of the machine. The largest ratio among the GPUs in this study is 25 on 9800GTX (Table 1). However, it is only 19 if our goal is approaching 66% of the peak arithmetic throughput as when using operands in shared memory under the 60 GB/s cap of the peak sustained bandwidth. This is achieved, for example, using 19×19 or larger blocks in C.

4.1 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 α and β are scalars. Also, we implemented $C := \alpha AA^T + \beta C$ for lower-triangular C. These operations are part of BLAS3's GEMM and SYRK routines [Dongarra et al. 1990]. We restrict our scope to matrix sizes that are multiples of the block sizes.

We pick a vector length of 64, which is the smallest that yields arithmetic peak in single precision according to Section 3.6. All data parallelism above this length is explicitly strip-mined into independent operations in the same thread program. We orient vectors along the columns of C to enable stride-1 memory access in fetching and storing back C's block. Similarly, B's block is chosen to be 16×16, as this enables aligned loads for both blocks in B and B^T. This leaves us with three choices for C's block: 16×16, 32×16 and 64×16. Larger blocks are not necessary. The two smallest blocks will require sharing A's ele-

GPU	SP peak, Gflop/s	SGEMM("N", "N", …)			SSYRK("L", "N", …)		DP peak, Gflop/s	DGEMM	DSYRK	
		CUBLAS1.1	ours	estimate	CUBLAS2.0	ours		ours	CUBLAS2.0	ours
8600GTS	93	37%	60%	58%	36%	60%	—	—	—	—
8800GTX	346	37%	60%	58%	37%	60%	—	—	—	—
9800GTX	429	36%	58%	58%	36%	58%	—	—	—	—
GTX280	624	44%	60%	58%	45%	60%	78	97%	35%	95%

Table 2: The estimated and the best observed rates in matrix-matrix multiply routines shown as a fraction of the peak.

ments among vector elements via shared memory. But this is not needed if we choose the 64×16 block.

The above decisions require sharing *B*'s block, so it is stored in shared memory. Row-major layout is preferred as it yields stride-1 accesses to shared memory in the inner loop of the matrix-multiply which is a rank-1 update of *C*'s block. This layout requires transposing *B*'s block in the *AB* case and padding the shared memory array to avoid the bank conflicts.

Earlier versions of the code used compiler options to enforce a tighter register budget and software prefetching. This was not found necessary in the later code.

The resulting thread program is outlined in Fig. 4.

We launch as many threads as there are non-zero blocks in *C*. Threads are created with 2D thread IDs as permitted in CUDA. When *C* is triangular, a naïve implementation would be to create as many threads as there are blocks in the full matrix and require threads corresponding to the zero blocks to exit immediately. This involves the overhead of creating and scheduling ~2× more threads than necessary. A better implementation is to cut the block index space of the triangular matrix into left and right parts and "glue" them together into a single rectangular piece. This requires slightly more sophisticated decoding of the 2D thread ID into the C's block index. In a similar fashion we programmed a routine that works with submatrices of triangular matrices.

The code is written in CUDA's C to offload register allocation and instruction scheduling to the compiler. *Decuda* was used to control compiler's efficiency.

We also compiled our codes into double precision by changing all "floats" into "doubles" and using proper compiler options. A minor adjustment that we did was keeping the padding of the shared memory arrays equal to one 32-bit word. Otherwise the padding increases producing bank conflicts.

Although we developed our algorithm independently as an optimization over the algorithm given in the CUDA programming manual, we later found that it closely resembles earlier algorithms designed for vector processors such as one by Agarwal and Gustavson [1989] designed for IBM 3090 Vector Facility and Anderson et al. [2004] for the Cray X1. As in our algorithm, these implementations keep blocks in *A* and *C* in the vector registers and keep the block in *B* in other fast memory that is shared across different vector elements — scalar registers and cache respectively. This similarity in the algorithms highlights similarities in the architectures.

4.2 Performance Results and Analysis

Table 2 shows the fractions of peak and Figures 5 and 6 show the absolute rates achieved in our implementations of matrix-matrix multiplies. Note, that the best performance over different GPUs is the same 58–60% of peak, i.e. it scales linearly with the clock rate and the number of cores. This fraction approaches 66% of the peak that bounds multiply-and-add instructions with shared memory operands. All our flops are done in such instructions.

```
Vector length: 64 //stripmined into two warps by GPU
Registers: a, c[1:16] //each is 64-element vector
Shared memory: b[16][16] //may include padding

Compute pointers in A, B and C using thread ID
c[1:16] = 0
do
  b[1:16][1:16] = next 16×16 block in B or B^T
  local barrier //wait until b[][] is written by all warps
  unroll for i = 1 to 16 do
    a = next 64×1 column of A
    c[1] += a*b[i][1]     // rank-1 update of C's block
    c[2] += a*b[i][2]     // data parallelism = 1024
    c[3] += a*b[i][3]     // stripmined in software
    ...                   // into 16 operations
    c[16] += a*b[i][16]   // access to b[][] is stride-1
  endfor
  local barrier //wait until done using b[][]
  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: The structure of our matrix-matrix multiply routines.

The table does not include rates achieved in GEMM in CUBLAS 2.0 as it is based on our code and runs at similar rates. However, the table implies that SYRK in CUBLAS 2.0 is based on the earlier GEMM codes.

To understand the performance of the algorithm, we perform cycle-counting on the disassembler output (*decuda*). The inner loop of the SGEMM program (both *AB* and AB^T cases) has 312 instructions, 20 of which are memory load instructions and 256 are multiply-and-adds (MAD) with one operand in shared memory. Only MADs contribute to the flop count. Assuming that multiply-and-adds consume 6 and all other instructions consume 4 cycles per warp, we get an estimate of 1760 cycles per loop body per warp. 256 MADs perform 256*32*2 = 16384 flops per warp, which is 58% of the peak value of 1760*8*2 = 28160 flops that can be done on a single core in this number of cycles. 58% of peak closely matches the observed rates as listed in Table 2. This result implies that performance is bound by the instruction throughput and not memory bandwidth or latency. We get 61% of peak in a similar estimate if assume that memory loads are processed in parallel in a different pipeline. This may explain why sustained values are higher than the expected number.

If slowdown in accessing shared memory were eliminated and MAD ran at full throughput of 4 cycles, a similar estimate would give 256/312 = 82% of the compute peak or 88% if memory loads were done in parallel. In comparison, we get 89–92% of peak on Core2 Duo and Quad processors when using SGEMM in 64-bit Intel MKL 10.0.

To understand how multithreading factors into the performance we varied the number of threads running simultaneously by allocating more shared memory than necessary and checking the resulting occupancy using a profiler. The code performed at 32%, 49%, 58% and 59% of the peak when running 1, 2, 3 and 4 threads per core resp. on all four GPUs (with non-substantial variations). GTX280 allows running up to 8 threads but performance didn't improve after 4 threads. These 4 threads per GTX280 core correspond to 25% occupancy. This indicates that one should not over-optimize for the occupancy, although extremely low occupancy may also hurt.

To highlight the importance of locality in accessing shared memory, we also run a version that keeps *B*'s block in column-major layout instead of row-major layout. Then the shared memory array is scanned at large stride in the inner loop that introduces extra 60–70 pointer arithmetic instructions in the loop body. Resulting performance was slightly lower, at 53–56% of peak. Thus, optimizing the locality in shared memory accesses is a valuable technique at least for finer tuning.

According to Table 2, peak rates in SYRK are nearly the same as in GEMM. This is natural, as the same main loop code is used in both routines. However, the naïve solution that creates 2× more threads half of which exit immediately runs at 1.1× lower peak rate and at up to 1.7× lower rate for some matrix sizes. Thus, although GPUs are known for efficient multithreading, unnecessary thread parallelism may result in a slowdown.

The double precision versions of these routines run at up to 95–97% of peak. This is much higher than we could expect in single precision and is possible only because double precision operations are much slower than other instructions and thus consume most of the execution time.

64×16 blocking yields 25.6× reduction of bandwidth consumption. Thus, 375 Gflop/s achieved on GTX280 corresponds to 59 GB/s in reading matrices *A* and *B*, which is 46% of the peak sustained bandwidth. In contrast, some of the earlier implementations, such as Govindaraju et al. [2006] and Fatahalian et al. [2004] are bandwidth-bound. The compute-bound implementation is due to using large blocks, which was made possible by introduction of shared memory into the GPUs architecture.

Results for modern GPUs include 91 Gflop/s by Ryoo et al. [2008] and 97 Gflop/s by Baskaran et al. [2008]. Both are on a 8800GTX and correspond to 26–28% of the compute peak. This is over 2× slower than achieved in our code. They follow traditional guidelines as those outlined in the CUDA programming guide, e.g. use longer vectors, optimize for fewer registers per scalar thread, get higher occupancy and use shared memory as the primary local storage space.

4.3 Comparison to CUBLAS 1.1

Table 3 compares the details of our implementation and the implementation in CUBLAS 1.1 as was released in public domain by NVIDIA (sgemm_main_gld_hw_na_nb_fulltile routine). Both implementations perform the same amount of work per vector thread computing 1024 elements in matrix *C*. Bandwidth reduction by blocking in CUBLAS's code is 32 which is better than 25.6 in our code. Also, CUBLAS's code uses 2× less registers per scalar thread and runs at higher occupancy of 2× more warps per core. However, it is 1.6× slower.

Much of this slowdown is due to the poorer instruction mix. Both codes use MAD instructions with an operand in shared memory. But CUBLAS holds both *A*'s and *B*'s blocks in shared memory and so must fetch second operand before use. This requires one MOV instruction per two MAD instructions and results in the small fraction of MADs instructions in the loop body — it is 56% versus 82% in our code.

The poor instruction mix in CUBLAS can be improved by

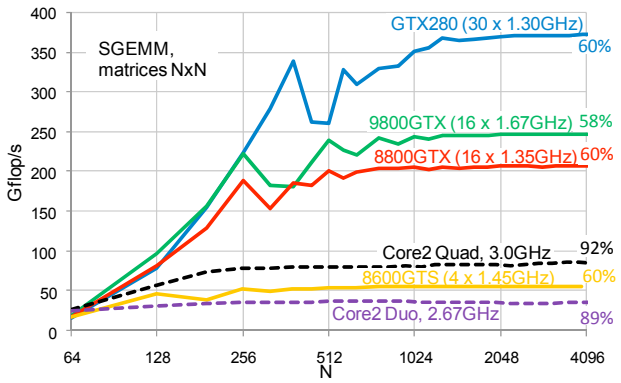

Figure 5: Rates in single precision matrix-matrix multiply on GPUs and CPUs. Percents indicate the fractions of peak.

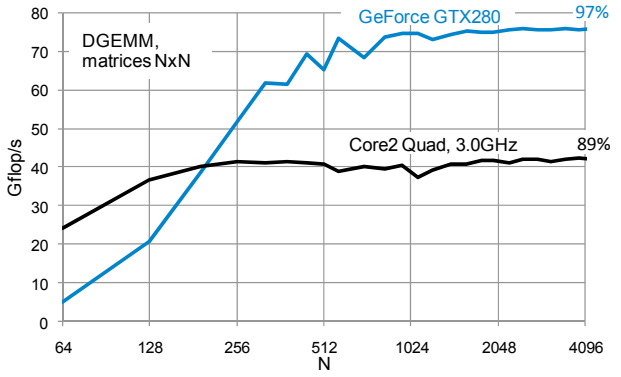

Figure 6: Rates in double precision matrix-matrix multiply on a GPU and a CPU. Percents indicate the fractions of peak.

	CUBLAS 1.1	Our code
C's block, where stored	32×32, regs	64×16, regs
A's block, where stored	32×32, smem	64×1, regs
B's block, where stored	32×32, smem	16×16, smem
Vector length	512	64
Scalar registers per scalar thread	15	30
Registers per vector thread	30 KB	7.5 KB
smem per vector thread	8.3 KB	1.1 KB
Threads/core, 8800GTX	1	4
Warps/core, 8800GTX	16	8
Instructions in inner loop	115	312
MAD instructions	64	256
smem-to-register MOVs	32	0
Expected, % of peak	44%	58%
Sustained, % of peak	36–44%	58–60%

Table 3: Details of our code and the code in CUBLAS 1.1. Instructions counts are for the inner loop only and were obtained using *decuda*. *A*'s 64×1 blocks are given as defined in the C-level program. This block size is increased when compiling by unrolling the loop and assigning the blocks fetched in different iterations to different registers.

using shorter vectors. In CUBLAS 1024-element blocks in *C* are strip mined at the program level into two 512-element vectors. This allows reusing the result of each MOV two times. Strip mining down to 64 element vectors instead would allow reusing data 1024/64 = 16 times. Finally, if *C*'s block is 64×16, then the data can be reused across an entire row of the block and there is no need to stage it via shared memory at all. This is what is done in our code.

However, if we use modeling as in the previous Section, we get an estimate of 44% of the peak for the CUBLAS code that is substantially higher than 36–37% that it sustains on all GPUs earlier than GTX280. This might be due to the insufficient parallelism to hide the memory latency. Although CUBLAS runs 2× more warps than we do, these warps come in one vector thread synchronized with local barriers. These barriers separate memory accesses and computation, so these two do not overlap. That's not an issue in our code where memory accesses in one thread may overlap with computations in several other concurrent threads.

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 the 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 a 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 the 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. 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 the CPU memory and back. Overlapping these transfers with computation, enabled on the newer GPUs, is left for the future work.

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 by buffering tiles in the on-chip memory to avoid extra consumption of DRAM. When the panel is transferred to the CPU memory and back, it is transposed on the GPU using an additional, smaller, buffer in DRAM. Pivoting is done in batches of 64 row interchanges to amortize the kernel launch overhead. The pivot indices are passed in as function parameters that are accessible in CUDA via shared memory. This is to avoid the substantial bandwidth requirements in reading the parameters from DRAM.

The upper triangular part of the output of the panel factorization in the QR algorithm is not needed for the later updates and is not transferred back to GPU. The lower triangular part is filled in with zeros and a unit diagonal before the transfer 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 and avoid paying extra GPU overheads.

Instead of running triangular solve (TRSM) in the LU decomposition we run matrix-matrix multiply with the inverse of the triangular matrix. The inverse is computed on the CPU. As a result, the GPU does no other arithmetic besides matrix-matrix multiplies. 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 by 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 that was used by Barrachina et al.

A different technique is used in QR factorization, which has a different structure of updates. We used autotuning to choose the 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. For example, transferring all even or odd columns of matrix can be done with one call to a CUDA routine, but transferring even or odd block columns requires many calls and so many transfer overheads. Each GPU holds only its own fraction of the matrix (even or odd columns). The exception is the updates, which require the transfer of an entire panel to both GPUs. 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 without 2-level blocking as in this case it requires substantial extra space for

storing the coarser blocks.

6 Results for Matrix Factorizations

For the results in this section we used a desktop system based on 2.67GHz Core2 Duo E6700 equipped with multiple PCIe 1.1 ×16 slots. For the results with one or two GeForce 8800GTX we used 32-bit Windows XP and CUDA 1.1. For the results with GeForce GTX280 we used 64-bit Windows XP and CUDA 2.0. CPU-only results were obtained on 3.0GHz Core2 Quad Q6850 running 64-bit Linux. In all cases the Intel MKL 10.0 library is used for factorizations on the CPU. We noted that it runs substantially slower in 32-bit. All results are in single precision.

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.

Matrices are padded to an odd multiple of 64 words. This helps avoiding anomalous performance drops at some matrix sizes.

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 $2000 \cdot \varepsilon \cdot \|A\|_{max}$, $200 \cdot \varepsilon \cdot \|A\|_{max}$ and $17 \cdot \varepsilon \cdot \|A\|_{max}$ 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

Fig. 7 shows the Gflop/s rates sustained in the GPU-based matrix factorization routines and using Core2 Quad alone, and Fig. 8 details the speedups vs. Core2 Quad. According to the Figure, the crossover between the GPU-based and CPU-alone implementations is around $n = 1000$ for all but Cholesky run on GTX280, which is around $n = 600$. The best performances are summarized in Table 4. It shows that the speedup is nearly the same as the speedup in matrix-matrix multiply (SGEMM). However, difference in theoretical arithmetic peak rates is substantially higher highlighting that there are more computational resources available than we could harvest.

Fig. 9 shows the performance of the LU decomposition that achieves 538 Gflop/s at $n \approx 21{,}000$ by running two GPUs in parallel. Note that a single GTX 280 yields higher rates than two 8800 GTX. See the notes below on scaling.

6.2 Performance Analysis

Fig. 10 shows the breakdown of runtime in the LU factorization on 8800GTX. 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**.** We expect the GPU part to be smaller when computing with faster GPUs producing better overlap at large matrix sizes. Time spent in the CPU-GPU transfers is substantial at small and medium sized

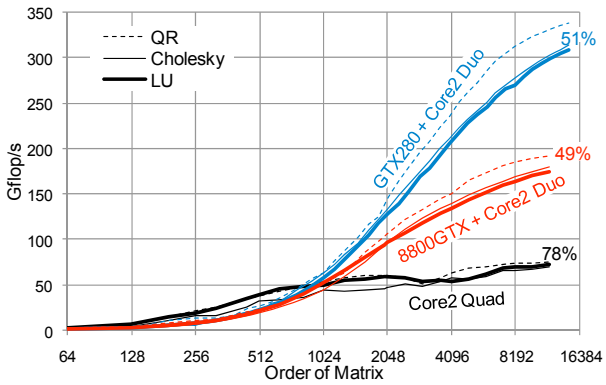

Figure 7: Rates achieved in the factorizations, percents indicate the highest fraction of the system's peak (GPU+CPU or CPU only) achieved.

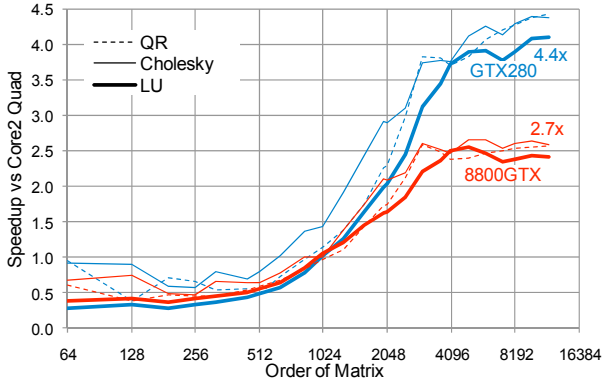

Figure 8: Speedup versus 3.0GHz Core2 Quad. Numbers on the right are the best speedups.

	Q6850	8800GTX+E6700		GTX280+E6700	
	Gflop/s	Gflop/s	speedup	Gflop/s	speedup
LU	73	179	2.5×	309	4.1×
Cholesky	70	183	2.7×	315	4.4×
QR	75	192	2.6×	340	4.4×
SGEMM	88	208	2.4×	375	4.3×
peak	96	388	4.0×	667	6.9×

Table 4: Comparison of best Gflop/s rates in the CPU and GPU versions and best speedup vs. the CPU-alone versions. SGEMM rates for the GPU+CPU systems include GPU rates only.

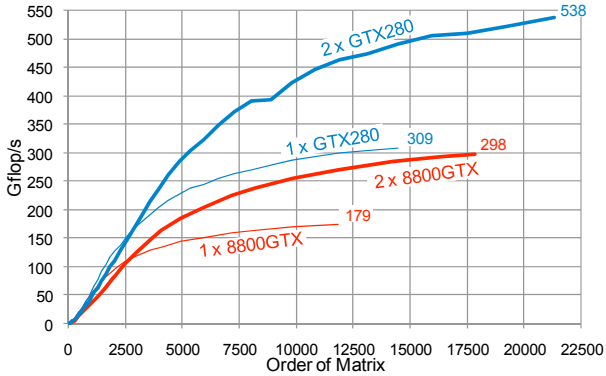

Figure 9: Performance of one-GPU and two-GPU versions of the LU decomposition with best rates in Gflop/s shown on right.

matrices and should be improved with the newer PCIe interconnect supported by the newer GPUs and motherboards. 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 ~7μs to copy or transpose a 1024×64 matrix at the peak sustained bandwidth of 76 GB/s, which is close to the kernel launch overhead. CPU-GPU transfers run at 3.0–3.3 GB/s for $n > 1000$, which approaches the peak sustained rate.

Fig. 11 evaluates the impacts of different optimizations used when computing on GTX280. The most important optimization was using row-major layout on the GPU that nearly doubled the performance at large problem sizes. Individual measurements have shown that pivoting takes 1–10% of time in the entire computation for $n > 1500$ if done in the row-major layout. In that case it achieves 7–25 GB/s of effective bandwidth. When using column-major layout, it takes 27–50% of the total time and run at 0.3–1.8GB/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 8192 right hand sides runs at 18 Gflop/s on GTX280 when using CUBLAS 2.0. It is an order of magnitude slower than the 268 Gflop/s rate achieved in multiplying a 64×64 matrix by a 64×8192 matrix that does the same work (this is 134 Gflop/s if not counting the redundant work).

Amortization of kernel launch overhead due to batch pivoting yields 30–100% speedup at $n < 1024$. Effect of all optimizations decreases at larger problem sizes, where time is dominated by matrix-matrix multiplies. Rates in these multiplies are affected by using 2-level schemes in LU and Cholesky and using autotuning to choose block size in QR. These techniques gave up to 4–7% speedup and factored in only for $n > 4096$.

According to Fig. 9, using two 8800GTX yields only 67% improvement in the peak Gflop/s rate. This result corresponds to pre-allocating pinned memory in the master CPU thread before GPU contexts are created in the child CPU threads. As a result, all transfers run at a small fraction of the peak PCIe bandwidth as if the memory was not pinned. Higher improvement of 74% when using two GTX280 corresponds to allocating pinned memory in one of the child CPU threads after the GPU contexts are attached. This memory is used to store the CPU's copy of the matrix, i.e. both the input and output data of the routine. This allows running transfers at full bandwidth to one of the GPUs. There are other reasons for less than ideal scaling, such as extra CPU-GPU bandwidth consumption, lack of 2-level blocking and not scaling the CPU side of the system.

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. They use a non-blocked algorithm that is bandwidth and/or overhead bound. Scaling these numbers with bandwidth gives up to 26 Gflop/s on GTX280, an order of magnitude less than in our implementation. Our solution works faster due to large blocking enabled by shared memory. Our high performance when pivoting is enabled by the high-bandwidth access to linear address space available on modern GPUs.

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. Our implementation

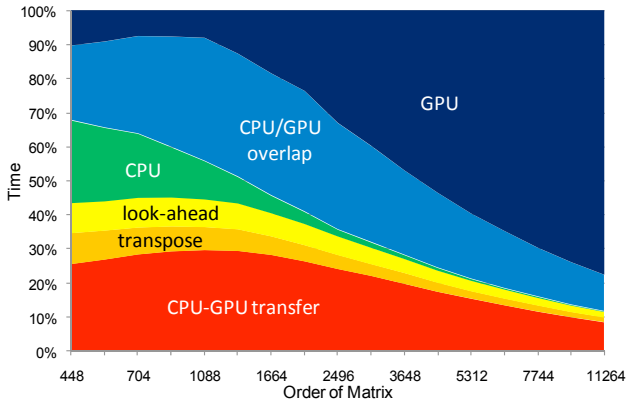

Figure 10: The breakdown of time in the LU decomposition run on GeForce 8800 GTX.

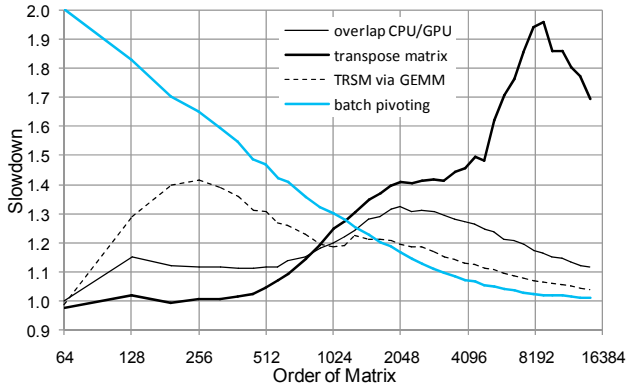

Figure 11: Slowdown when omitting one of the optimizations used when running on GeForce GTX 280.

achieves 2.9× and 3.7× higher speed for LU and Cholesky resp. on the slightly slower 8800GTX. 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 has slightly slower memory than 8800GTX and otherwise similar. 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 due to a deeper performance analysis and tuning.

Castillo et al. [2008] report results for Cholesky factorization run on 4-GPU NVIDIA Tesla S870. Each of these GPUs is similar to Quadro FX5600 described above. Authors report 180 Gflop/s on a system at $n \approx 10{,}000$. We achieve this performance using a single 8800GTX. Their result was later improved to 424 Gflop/s at $n \approx 20{,}000$ by using the 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 hard-

ware, 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 arithmetic throughput, which are important to understanding the limits of performance of many algorithms including our own. We highlighted some of the optimization guidelines, such as using shorter vectors at the program level and using register file as the primary on-chip storage space.

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.

Acknowledgements

We want to thank NVIDIA for the donated hardware, John Nickolls for the detailed discussions on the GPU architecture, Sam Williams for the generic discussions on computer architecture, Paul Leventis for discussions on optimizing matrix-matrix multiply, Professor Krste Asanović for the vector insights into the GPU architecture, John Shalf for hints on barrier synchronization, and Massimilano Fatica and Jon Kuroda for assistance in the production of this paper.

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.

ALVERSON, 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.

BASKARAN, M., BONDHUGULA, U., KRISHNAMOORTHY, S., RAMANUJAM, J., ROUNTEV, A., AND SADAYAPPAN, P. 2008. A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs, *ISC'08*.

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. J., DU CROZ, J., HAMMARLING, S., AND DUFF, I. 1990. A Set of Level 3 Basic Linear Algebra Subprograms, *ACM Transactions on Mathematical Software 16*, 1, 1–17.

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*.

HWU, W. W., AND KIRK, D. 2007. ECE 498 AL1: Programming Massively Parallel Processors, Lecture Slides, University of Illinois, Urbana-Champaign.

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

NVIDIA. 2008a. NVIDIA CUDA Compute Unified Device Architecture, Programming Guide, Version 2.0.

NVIDIA. 2008b. NVIDIA GeForce GTX 200 GPU Architectural Overview, Technical Brief, May 2008.

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.

RYOO, S., RODRIGUES, C. I., BAGHSORKHI, S. S., STONE, S. S., KIRK, D. B., AND HWU, W. W. 2008. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU using CUDA, *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ACM Press, 2008, 73–82.