Implementation and Tuning of Batched Cholesky Factorization and Solve for NVIDIA GPUs

Jakub Kurzak, Hartwig Anzt, Mark Gates, and Jack Dongarra

Abstract—Many problems in engineering and scientific computing require the solution of a large number of small systems of linear equations. Due to their high processing power, Graphics Processing Units became an attractive target for this class of problems, and routines based on the LU and the QR factorization have been provided by NVIDIA in the cuBLAS library. This work addresses the situation where the systems of equations are symmetric positive definite. The paper describes the implementation and tuning of the kernels for the Cholesky factorization and the forward and backward substitution. Targeted workloads involve the solution of thousands of linear systems of the same size, where the focus is on matrix dimensions from 5 by 5 to 100 by 100. Due to the lack of a cuBLAS Cholesky factorization, execution rates of cuBLAS LU and cuBLAS QR are used for comparison against the proposed Cholesky factorization in this work. Execution rates of forward and backward substitution routines are compared to equivalent cuBLAS routines. Comparisons against optimized multicore implementations are also presented. Superior performance is reached in all cases.

Index Terms—Cholesky factorization, batched, kernel, GPU, CUDA, SIMT

1 Introduction

While linear algebra software has achieved high efficiency for solving large linear systems on GPU-based computers [1], achieving good performance for small linear systems has been more challenging. The inherently limited parallelism available in small linear systems, e.g., matrices of sizes less than 100×100 elements, fails to fully utilize today's highly parallel computing hardware. When a large set of small linear systems is presented simultaneously, using a batched implementation exposes significant parallelism, allowing for more significant use of parallel hardware. On streaming processors, like GPUs, an additional advantage comes from the reduced kernel launch overhead from a single batched function call, versus making multiple kernel calls, one for each linear system. Each GPU multiprocessor has fast local memory consisting of registers and shared memory. Exploiting data locality by reusing data kept in this fast local memory is crucial for attaining high performance [2], [3].

Numerous applications deal with large sets of small linear solves that call for batched processing on GPUs. Examples include: the *Alternating Least Squares* (ALS) method in data analytics [4], digital volume correlation in experimental mechanics [5], [6], and *Rigorous Coupled-Wave Analysis* (RCWA) in computational lithography [7], [8]. Batched operations can also be used as building blocks to solve large linear systems, where the coefficient matrices are sparse matrices with small dense blocks [9], [10].

- *J. Kurzak is with the Electrical Engineering and Computer Science, University of Tennessee, 1122 Volunteer Blvd, Ste 413 Claxton, Knoxville, Tennessee. E-mail: kurzak@eecs.utk.edu.*
- *H. Anzt, M. Gates, and J. Dongarra are with the Electrical Engineering and Computer Science, University of Tennessee, Knoxville, Tennessee. E-mail: {hanzt, mgates3, dongarra}@eecs.utk.edu.*

Manuscript received 14 July 2015; revised 16 Sept. 2015; accepted 19 Sept. 2015. Date of publication 23 Sept. 2015; date of current version 15 June 2016. Recommended for acceptance by A. Dubey.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2015.2481890

This work is mostly motivated by the need for a fast batched linear solve for the ALS algorithm. Therefore, routines are developed for applying the Cholesky factorization and forward and backward substitution to a large set of small linear systems with a single right hand side. First, algorithmic choices are considered carefully to exploit the small advantage in the number of floating point operations over memory operations in the factorization, while simultaneously applying aggressive vectorization. Then, the implementation is presented, which maps the factorization and the solve to the highly parallel vector architecture of the GPU. Finally, an automatic software tuning methodology is applied to find parameters that maximize performance for the each problem size in the range of interest (from 5×5 to 100×100).

1.1 Related Work

The use of batched operations, when targeting problem sizes that do not fully utilize the hardware resources, has been realized for numerous algorithms and hardware architectures. Villa et al. developed a batched LU solver for matrix sizes up to 128, where the complete factorization is handled by a single GPU thread block [11]. The implementation provided significant GPU acceleration to a subsurface transport simulation [12]. Dong et al. [13], [14] and Haidar et al. [15], [16] developed batched routines for LU, Cholesky and QR factorizations, targeting larger matrix sizes (up to 512×512), and relying mostly on recursion. Batched linear algebra routines have been part of the cuBLAS library for a few years now, and the current set includes routines for LU and QR factorizations, matrix multiplication, triangular solve, least squares minimization and matrix inversion [17].

1.2 Novelty

A substantial body of work on batched kernels has been done in the past. The main aspects of this work, that differentiate it from previous developments are as follows:

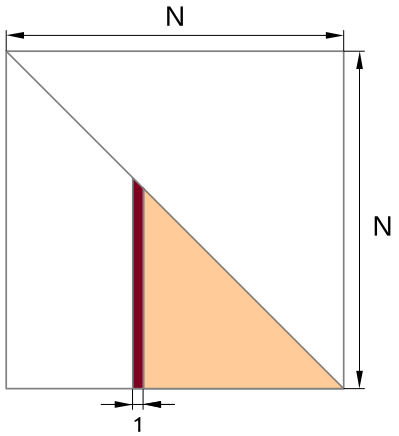

Fig. 1. Unblocked, right-looking, lower-triangular Cholesky factorization. Dark area is current column; light area is lower part of trailing matrix to update.

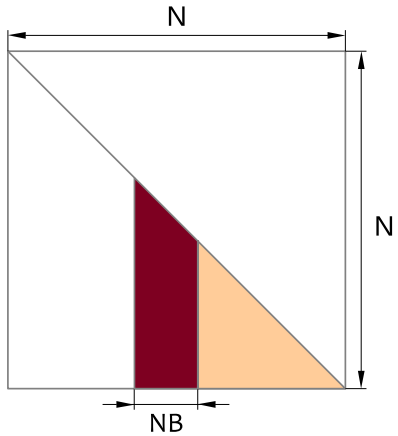

Fig. 2. Blocked, right-looking, lower-triangular Cholesky factorization. Dark area is current panel; light area is lower part of trailing matrix to update.

- While the objective of batched GPU kernels is to address the performance shortcomings of GPUs for very small matrix operations, no attention has been paid in the past to really small problems. While others reported performance improvements over cuBLAS for matrices of size up to 500, this work reports large performance improvements over past efforts for matrices of size up to 100.
- While the discussion of different algorithmic variants was included in previous work, the right-looking and left-looking formulation of the Cholesky factorization have not been used to their full advantage, i.e., the right-looking to provide SIMT parallelism, and the left-looking to minimize memory traffic.
- No previous work on batched kernels leveraged autotuning to the extend that this work does. In the past, only minimum amount of tuning was done or tuning was circumvented altogether with recursion, which has serious drawbacks for GPU kernel development.
- This article reports the development of a monolythic, tunable Cholesky factorization kernel, while other articles reported the development of composite routines, relying on both hand-coded kernels and cuBLAS calls.

2 Algorithm

A symmetric positive definite system of linear equations $Ax = b$ can be solved by computing the Cholesky factorization $A = LL^T$, and then applying forward substitution and backward substitution to the vector b using the factors L and L^T. Section 2.1 describes the factorization, while Section 2.2 describes the forward and backward substitutions.

2.1 Factorization

In linear algebra, the Cholesky factorization is a decomposition of a symmetric positive definite matrix into the product of a lower triangular matrix and its transpose. When applicable, the Cholesky factorization can be used for solving systems of linear equations roughly twice as efficiently as the LU factorization, in terms of the number of floating-point operations. Because of its simplicity, the Cholesky factorization is a common target for implementations on new and emerging architectures, as well as scalability studies on large scale distributed memory systems. In the LAPACK software package [18], the Cholesky factorization in single precision is implemented by the `SPOTRF` routine.

Algorithm 1 and Fig. 1 show the basic algorithm for computing the Cholesky factorization. The algorithm descends down the diagonal of the matrix and, in each step: replaces the diagonal element by its square root, divides each element in that column (the *panel*) by the resulting value, and applies a rank-1 update to the remaining part of the matrix to the right (the *trailing submatrix*). Normally, the algorithm operates on only half of the symmetric matrix, either lower or upper, leaving the other part untouched. When translated literally to code, this algorithm produces the *unblocked* implementation, i.e., where one column is factored at a time, and rank-1 updates are applied to the trailing submatrix. When built using the set of *Basic Linear Algebra Subroutines* (BLAS), only Level 1 and Level 2 BLAS calls are used, i.e., vector-vector and matrix-vector operations, which produce a memory bound implementation of an otherwise compute bound algorithm.

Algorithm 1. Unblocked, right-looking, lower-triangular Cholesky factorization using C-style (zero-based) indexing

```
1: for k = 0 to N − 1 do
2:     A_kk ← √A_kk
3:     for m = k + 1 to N − 1 do
4:         A_mk ← A_mk / A_kk
5:         for n = k + 1 to N − 1 do
6:             for m = n to N − 1 do
7:                 A_mn ← A_mn − A_nk × A_mk
8:             end for
9:         end for
10:    end for
11: end for
```

The main optimization of the Cholesky algorithm is *blocking* (Fig. 2). Blocking is the main optimization of dense linear algebra routines for cache-based systems, and is the main idea behind the LAPACK software library. Blocking replaces most of the Level 1 and Level 2 BLAS calls in the unblocked algorithm with Level 3 BLAS calls (matrix-matrix

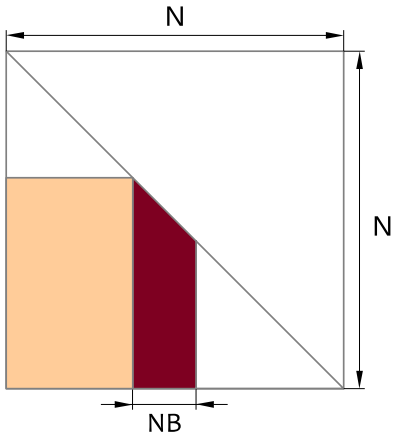

Fig. 3. Blocked, left-looking, lower-triangular Cholesky factorization. Dark area is current panel; light area is pending updates from previous panels to apply to current panel.

operations). Blocking leverages the *surface-to-volume* property of dense linear algebra routines like the Cholesky factorization, i.e., the fact that they perform $O(N^3)$ floating-point operations on $O(N^2)$ data. In the blocked Cholesky factorization, one panel of width NB is factored at a time, where $1 \ll NB \ll N$, followed by an update of rank NB. Described as a loop transformation, blocking means tiling of the outermost loop in line 1 of Algorithm 1.

Another important aspect of the implementation is the choice between aggressive and lazy evaluation. The two common versions of the Cholesky factorization are *right-looking* and *left-looking*. The right-looking implementation corresponds to aggressive evaluation, where, as soon as the panel is factored, the entire trailing submatrix is updated. Algorithm 1 and Figs. 1 and 2 show the right-looking Cholesky factorization. The right-looking factorization favors parallelism over data locality by quickly exposing a large volume of work. At the same time, it modifies (reads and writes) the entire trailing submatrix, and therefore does not readily target a constrained memory situation, which is where the left-looking factorization has an advantage.

Fig. 3 shows the left-looking (blocked) Cholesky factorization, which corresponds to lazy evaluation. The left-looking factorization relies on deferred updates to the trailing submatrix, where updates are applied only to the panel area immediately before the panel factorization. This means that in each step of the algorithm, all pending updates (from the left side of the matrix) are applied, and then the panel is factored. In this case, in each step, only the panel area is modified (read and written), while the large part of the matrix to the left of the panel is only read. Because of that, the left-looking variant is the basis for implementations in constrained memory situations.

Traditionally, algorithms dealing with limited memory capacity have been referred to as *Out-of-Core* (OOC) algorithms, by reference to the *magnetic core memories* used in the 50, 60, and 70 s. Today many consider this term confusing, due to the emergence of *multicore* processors, where the word *core* refers to a processing unit. Therefore, the term Out-of-Core is sometimes replaced with the term *Out-of-Memory* (OOM). However, in the context of this work, this term is also misleading, because we are dealing with a situation where the data does not fit into registers, rather than memory. Here, the term *non-resident algorithm* is adopted instead.

The objective of a non-resident algorithm is to place a small amount of read-and-write data in fast memory and stream in all read-only data from slow memory. This may refer to many different situations. Slow memory versus fast memory may mean disk versus RAM, RAM versus cache, cache versus registers, or main (host) memory versus accelerator (device) memory. The main idea of the non-resident batched Cholesky factorization is to use the left-looking variant, place the panel in the fast memory, stream in the pending updates from the slow memory, factor the panel in the fast memory, and finally save it back to the slow memory.

2.2 Forward and Backward Substitution

When the input matrix is factored into the product of a lower-triangular matrix and its transpose, the system of linear equations can be solved by applying forward and backward substitutions. Algorithm 2 shows the forward substitution procedure; Algorithm 3 shows the backward substitution procedure. These algorithms, know as *triangular solves*, are implemented by the `STRSV` routine in the set of Level 2 BLAS. Unlike the factorization, the triangular solves apply $O(N^2)$ operations to $O(N^2)$ data, and therefore are completely memory bound and do not offer multiple algorithmic alternatives.

Algorithm 2. Forward Substitution Using Zero-Based Indexing

1: **for** $k = 0$ to $N - 1$ **do**
2: $\quad b_k \leftarrow b_k / L_{kk}$
3: $\quad$ **for** $n = k + 1$ to $N - 1$ **do**
4: $\quad\quad b_n \leftarrow b_n - b_k / L_{kn}$
5: $\quad$ **end for**
6: **end for**

Algorithm 3. Backward Substitution Using Zero-Based Indexing

1: **for** $k = N - 1$ to 0 **do**
2: $\quad b_k \leftarrow b_k / L_{kk}$
3: $\quad$ **for** $n = k - 1$ to 0 **do**
4: $\quad\quad b_n \leftarrow b_n - b_k / L_{nk}$
5: $\quad$ **end for**
6: **end for**

3 Architecture

The two most prominent features of GPUs are their *Single Instruction Multiple Threads* (SIMT) architecture and their memory model. A brief architecture overview is in place, because these features dictate the algorithmic choices while implementing batched matrix operations. While SIMT favors aggressive evaluation, memory efficiency favors lazy evaluation.

Fig. 4 shows the basic architecture of NVIDIA GPUs. The basic execution unit of an NVIDIA GPU is referred to as a *CUDA core*. A single core is capable of executing floating point instructions at a throughput of one instruction per cycle. However, a single core is not capable of following an

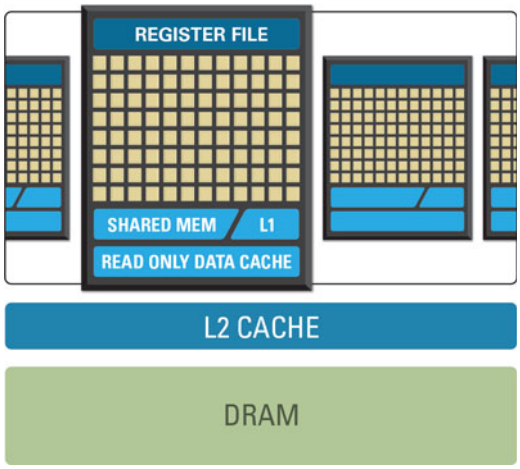

Fig. 4. Architecture of NVIDIA GPUs.

independent instruction stream. Instead, a set of 32 cores, referred to as a *warp*, have to follow the same execution path.

Cores are organized into *multiprocessors*. In the Kepler architecture, a multiprocessor contains 192 cores, and the GPU contains up to 15 multiprocessors, for a total of 2,880 cores. Aside from the cores and their instruction scheduling logic, the Kepler multiprocessor also contains a large register file of 65,536 32-bit registers, 48 KB of read-only data cache, and 64 KB of fast memory, which serves as L1 cache and *shared memory*.

The shared memory is a type of memory that is specific to GPUs. Originally, GPUs were designed with data-parallel workloads in mind, and cores had no way of collaborating by exchanging data. To handle more complex tasks, the shared memory was introduced to allow for exchanging data among cores, and, conceptually, is more of an extension of the register file than a cache.

3.1 SIMT

The programming model for NVIDIA GPUs is referred to as *Single Instruction Multiple Threads*. The thread is the basic software abstraction for a core. Most of the time, each core handles execution of multiple threads. Threads are organized into one-, two-, or three-dimensional *thread blocks*. In the Kepler architecture, a block can contain as many as 1,024 threads. Each block is assigned to a multiprocessor, and each multiprocessor executes multiple blocks.

The main challenge in SIMT programming is in writing code where a large number of threads execute the same instruction in each cycle, most of the time. In dense linear algebra, a two-dimensional thread block is a natural match for a two-dimensional matrix. Therefore, the problem is basically one of code vectorization using fairly large two-dimensional vectors. The task becomes uniquely difficult in the case of batched operations, where the sizes of the input matrices are close to the sizes of the thread blocks.

From the algorithmic standpoint, the appropriate response is to use aggressive evaluation, where the largest piece of work is exposed at each step. In the case of the Cholesky factorization, this means using the right-looking variant. From the implementation standpoint, the appropriate response is to allow all threads to execute, regardless of their location within the bounds of the data or outside of the bounds of the data. Corruption of valid results by out of bounds operations is avoided by checking the bounds only when writing the results from registers to main memory.

3.2 Memory

The fastest memory in the multiprocessor is the register file. Registers are partitioned among threads, and, at the time of execution, each thread has a private set of registers. The Kepler architecture introduced the shuffle instruction, which allows exchanging data directly among registers of threads within the same warp. However, the shared memory remains the main mechanism of exchanging data among threads.

The second fastest memory in the multiprocessor is the shared memory and L1 cache. The L1 cache is a standard hardware-controlled cache. Shared memory, on the other hand, is completely software-controlled. A pair of threads can exchange data by storing values from registers to shared memory, synchronizing, and reading the data from shared memory to registers.

The slowest memory in the system is DRAM. Reads from DRAM pass through L2 cache, and L1 cache or read-only data cache. DRAM bandwidth is a precious commodity for batched matrix operations, which are very close to being memory bound. From the algorithmic standpoint, the appropriate response is to use lazy evaluation, which minimizes the number of writes to memory. In the case of the Cholesky factorization, this means using the left-looking variant.

4 Implementation

This section describes the implementation of the factorization and the forward and backward substitutions. The factorization is the more challenging part. The objective is to exploit the $O(N^3)$ floating-point intensity for small values of N. Autotuning is a critical factor in accomplishing this objective. On the other hand, the triangular solves are completely memory bound, and the objective of the implementation is to saturate the memory bandwidth. Although there is not much room for autotuning, satisfying this goal still calls for a creative implementation.

4.1 Factorization

The two critical issues to address when coding an efficient batched dense factorization are efficient use of the memory and efficient vectorization, in the context of SIMT programming. While the left-looking algorithm is perfect for solving the first problem, the right-looking algorithm is suitable for solving the second one. To address memory efficiency first, the non-resident Cholesky algorithm is used, where DRAM is considered the slow memory, and shared memory and registers are used as the fast memory. Algorithm 4 describes the non-resident Cholesky factorization. This algorithm requires only $O(N \times NB)$ of fast memory to factor an $N \times N$ matrix.

Here, *global memory* is synonymous with GPU RAM, while local memory is, under certain conditions, synonymous with registers. The following actions take place in one step of the outermost loop. First, a panel of width NB is loaded to local memory (line 2). Then a loop goes over all NB-size stripes to the left of the panel, loads one stripe at a time into shared memory (line 5) and applies the pending update to the current panel (line 7). The current panel is then

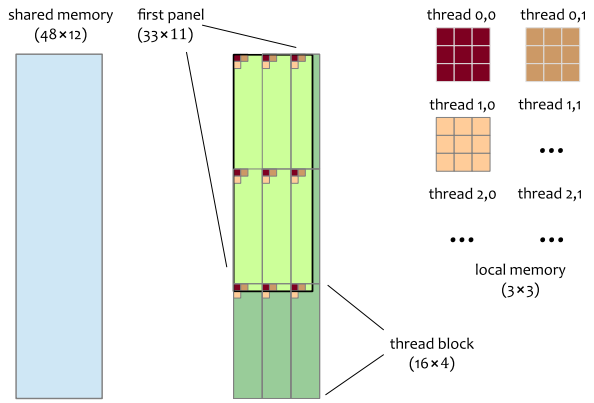

Fig. 5. Shared memory and register requirements of the batched Cholesky factorization for the specific case of factoring a 33×33 matrix using a panel of width $NB = 11$ and a thread block of size 16×4.

factored, using both registers and shared memory (line 10), and then written back to the global memory. One important aspect of this implementation is that it requires only $N \times NB$ entries of shared memory and $N \times NB$ registers per thread block. Algorithm 4 also shows the `__syncthreads()` synchronizations, providing memory coherency.

Algorithm 4. Non-resident Cholesky Factorization in the context of its GPU Implementation

```
1: for K = 0 to ⌈N/NB⌉ − 1 do
2:    Read K'th panel from global memory to local memory.
3:    __syncthreads();
4:    for L = 0 to K − 1 do
5:       Read L'th panel from global memory to shared memory.
6:       __syncthreads();
7:       Apply update from L'th panel to K'th panel.
8:       __syncthreads();
9:    end for
10:   Factor K'th panel using local and shared memory.
11:   __syncthreads();
12:   Save K'th panel from shared memory to global memory.
13:   __syncthreads();
14: end for
```

Fig. 5 shows the shared memory and register requirements for a specific case of factoring a 33×33 matrix using a panel of width $NB = 11$ and a thread block of size 16×4. In the first step, the thread block needs to cover the first panel of the matrix, of size 33×11, which means it needs to loop over a 3×3 pattern. Effectively, the matrix elements are assigned to the threads in a 2D block-cyclic fashion, and each thread ends up working on a 3×3 array in local memory. To avoid `if` conditions in the code, all work is actually done on the 48×12 area in local memory, and the same goes for shared memory, which is also allocated in a 48×12 chunk. This is wasteful in terms of operations and register usage, but it spares the code a number of conditional (`if`) statements, which degrade performance by preventing loop unrolling and causing thread divergence.

Fig. 6 shows what happens in the three steps of the factorization. The first panel is factored, then one pending update is applied and the second panel is factored, then two pending updates are applied and the third panel is factored. In this particular case, each thread's working set

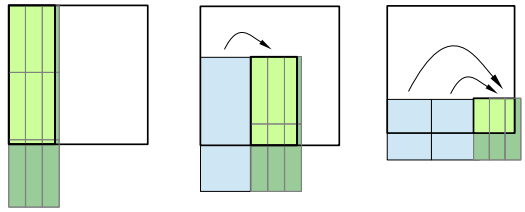

Fig. 6. The steps of the batched Cholesky factorization of a 33×33 matrix using a panel of width $NB = 11$ and a thread block of size 16×4.

shrinks by one, in the vertical dimension, after each step of the factorization.

Fig. 7 shows what happens during the panel factorization. This is where effective SIMT vectorization is the main challenge, with conditionals posing the main threat to performance. The top part shows the operations that have to be applied to produce the factorization within the bounds of the panel, following Algorithm 1. This is the area of the matrix which would be accessed by a serial implementation. The bottom part shows the operations executed in the presented SIMT implementation. In the first step, the diagonal element is taken from local memory, its square root computed, and the result stored in shared memory. In the second step, the column elements are taken from local memory, their values divided by the diagonal, and the results stored in shared memory. In the third step, a rank-1 update is applied to the entire panel in local memory, using the results of the previous operation in shared memory. As soon as the third step completes, the first column of the trailing submatrix is copied from registers to shared memory, so that it can be used in step one of the next iteration.

The first step involves 1 thread and executes 1 operation, the second step involves $blockDim.x$ threads and executes

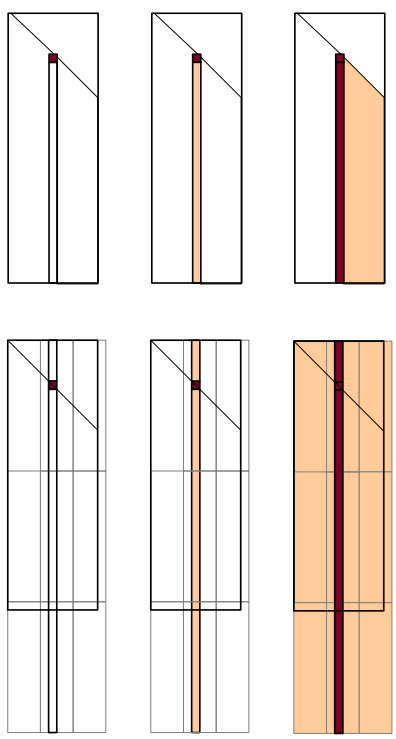

Fig. 7. The steps of factoring a panel of size 33×11 using a thread block of size 16×4. Top row is generic Cholesky algorithm; bottom row is corresponding GPU implementation.

```
LDS R40, [R7+0xc0];
LDS R28, [R6+0x100];
FFMA R29, -R29, R34, R19;
LDS R27, [R7+0xd0];
FFMA R43, -R40, R28, R43;
LDS R24, [R7+0xe0];
LDS R38, [R6+0x180];
FFMA R44, -R25, R31, R10;
LDS R37, [R7+0x180];
FFMA R25, -R25, R34, R4;
LDS R30, [R6+0x1c0];
LDS R26, [R7+0x190];
FFMA R48, -R24, R28, R25;
LDS R23, [R7+0x1a0];
FFMA R41, -R33, R31, R22;
LDS R35, [R6+0x240];
LDS R36, [R7+0x240];
LDS R32, [R6+0x280];
FFMA R41, -R40, R39, R41;
LDS R22, [R7+0x250];
LDS R21, [R7+0x260];
FFMA R20, -R27, R39, R20;
LDS R42, [R6+0x300];
FFMA R29, -R27, R28, R29;
LDS R33, [R7+0x300];
FFMA R44, -R24, R39, R44;
LDS R31, [R6+0x340];
LDS R4, [R7+0x310];
FFMA R34, -R26, R38, R20;
LDS R19, [R7+0x320];
FFMA R44, -R23, R38, R44;
FFMA R24, -R37, R38, R41;
LDS R39, [R6+0x3c0];
FFMA R27, -R37, R30, R43;
FFMA R41, -R26, R30, R29;
FFMA R30, -R23, R30, R48;
LDS R40, [R7+0x3c0];
LDS R28, [R6+0x400];
LDS R25, [R7+0x3d0];
FFMA R23, -R36, R35, R24;
LDS R10, [R7+0x3e0];
FFMA R26, -R36, R32, R27;
FFMA R34, -R22, R35, R34;
FFMA R41, -R22, R32, R41;
FFMA R48, -R21, R32, R30;
FFMA R44, -R21, R35, R44;
```

Fig. 8. A portion of the assembly code of the panel update operation (Kepler instruction set).

```
        MOV R4, R11;
        CAL `(__cuda_sqrt_rn_f32);
        STS.S [RZ], R4
.L_1:
        ISETP.EQ.AND P4, PT, R0, RZ, PT;
        BAR.SYNC 0x0;
        SHF.L.W R6, RZ, 0x2, R5;
        SSY `(.L_2);
  @!P4 NOP.S
        LDS R4, [RZ];
        CAL `(__cuda_rcp_rn_f32);
        FMUL R7, R11, R4;
        FMUL R10, R13, R4;
        FMUL R4, R8, R4;
        STS [R6], R7;
        STS [R6+0x40], R10;
        STS.S [R6+0x80], R4
.L_2:
        IADD R10, R5, 0x10;
        BAR.SYNC 0x0;
        SHF.L.W R7, RZ, 0x2, R0;
        LDS R24, [R6];
        LDS R23, [R6+0x40];
        ISETP.NE.AND P0, PT, R0, 0x1, PT;
        LDS R20, [R7];
        LDS R21, [R6+0x80];
        FMUL R13, -R20, R23, R13;
        LDS R19, [R7+0x10];
        FMUL R11, -R20, R24, R11;
        LDS R4, [R7+0x20];
        FMUL R12, -R19, R24, R12;
        FMUL R15, -R20, R21, R8;
        IADD R8, R0, 0x4;
        FMUL R14, -R19, R23, R14;
        FMUL R16, -R19, R21, R16;
  @!P0 STS [R6+0xc0], R11;
        FMUL R17, -R4, R24, R17;
  @!P0 STS [R6+0x100], R13;
        FMUL R18, -R4, R23, R18;
  @!P0 STS [R6+0x140], R15;
        FMUL R25, -R4, R21, R25;
        BAR.SYNC 0x0;
        ISETP.EQ.AND P0, PT, R0, 0x1, PT;
        SSY `(.L_3);
        ISETP.EQ.AND P0, PT, R5, 0x1, P0;
  @!P0 NOP.
```

Fig. 9. Assembly code for factoring a 33×11 panel using a 16×4 thread block (Kepler instruction set).

$blockDim.x \times 3$ operations, and the third step involves $blockDim.x \times blockDim.y$ threads, and executes $(blockDim.x \times 3) \times (blockDim.y \times 3)$ operations. Here, $blockDim.x$ and $blockDim.y$ are the CUDA variables describing the dimensions of the thread block. In this scenario, unnecessary floating-point operations are executed in the area of the panel above the diagonal, to the left of the column factored in each step, and below the boundary of the panel, for the sake of avoiding `if` statements. Interestingly, since $x/\sqrt{x} = \sqrt{x}$, the diagonal element does not have to be excluded from step 2, which saves yet another `if` statement.

The factorization is coded in C/CUDA, without the use of lower level constructs such as intrinsics or embedded PTX. Also, the code relies on `#pragma unroll` statements for all loops except for the outermost loop, in line 1 of Algorithm 4, which is explicitly unrolled using the *pyexpander* preprocessor. Fig. 6 shows the three iterations of this outer loop for the matrix of size 33 with the panel of size 11. The reason this loop is explicitly unrolled is because the value of the loop counter affects the boundaries of the inner loops, and loops with non-constant boundaries do not get unrolled. This does not cause substantial code bloat because the number of iterations of the outermost loop is small.

The expectation is that the compiler completely unrolls all loops and places most local memory variables in registers. Specifically, it is important that the local array holding the panel remains in registers throughout the entire factorization. Whether this happens or not can be determined by a quick inspection of the produced assembly, which can be created by compiling the source to the *cubin* format and then disassembling it. Compilation to cubin is done by passing the `-cubin` flag to the *nvcc* compiler. Disassembly is accomplished by passing the resulting file to the *nvdisasm* tool.

Figs. 8 and 9 show portions of the Kepler assembly code of the 33×33 matrix factorization using a panel of width 11 and a thread block of size 16×4. Fig. 8 shows part of the update operation. As intended, the code contains loads from shared memory (`LDS`) and *Fused Multiply-Add* (FMA) operations (`FFMA` in single precision). Fig. 9 shows the entire assembly code of the first step in the first panel factorization, i.e., factoring one column and applying one rank-1 update. The first block computes the square root of the diagonal element (`__cuda_sqrt_rn_f32`) and stores it in shared memory. The second block starts with a barrier (`BAR.SYNC`), computes the reciprocal of the diagonal (`__cuda_rcp_rn_f32`), applies it to the three values in the column ($3 \times$ `FMUL`), and stores the results in shared memory ($3 \times$ `STS`). The third block starts with a barrier (`BAR.SYNC`), applies the rank-1 update ($9 \times$ `FMUL`), and saves the first column of the trailing submatrix to shared memory ($3 \times$ `STS`).

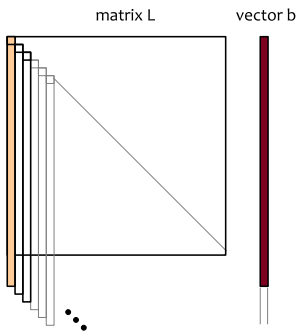

Fig. 10. Forward substitution using the lower triangular matrix L.

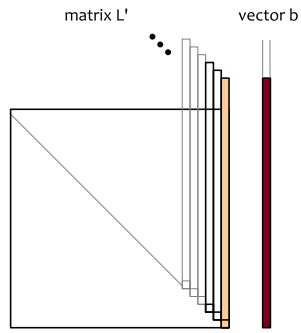

Fig. 11. Backward substitution using the upper triangular matrix L^T.

4.2 Forward and Backward Substitution

Triangular solves, with a single right hand side, offer very little parallelism. The vector b is updated by one column of A at a time. For matrices of size 32 and smaller, there is only work for one warp. For larger matrices, there is work for more warps in the initial steps, but it drops off as the loop progresses (Fig. 10). Under such circumstances, there is no real downside to just using a single warp for each solve. This solution provides two distinct advantages. If only one warp is used, implicit warp-level synchronization obviates the need for explicit `__syncthreads()` calls to synchronize. Also, the results of the division in line 2 of Algorithms 2 and 3 can be sent to all the threads in the warp by using direct register-to-register communication, instead of shared memory. This can be accomplished by the `__shfl()` instruction, available in devices of compute capability 3.0 and above.

While the forward substitution can be implemented by aligning the warp with the columns of the matrix, as shown in Fig. 10, the backward substitution presents a new challenge. Traditionally, in linear algebra software (e.g., LAPACK), the Cholesky factorization modifies only half of the input matrix, either lower or upper, leaving the other part unaffected. The lower part is updated throughout this paper. Then both the forward substitution and the backward substitution use the same matrix, the latter accessing the matrix in a transposed fashion. This creates a real problem for the GPU implementation, by imposing upon the threads a memory access pattern with stride equal to the size of the matrix. The standard solution to this problem is to use shared memory for reading the matrix in stripes, and performing the transposed access in shared memory, where it has less penalty. However, being a shared resource, shared memory limits the maximum occupancy reachable by the code. An alternative and somewhat unorthodox solution is proposed here.

Each panel is factored using both registers and shared memory, and ultimately saved to device memory from shared memory. For the transposed access to also be efficient, the triangular matrix is written to device memory in both non-transposed form and transposed form, the latter placed in the unused, upper part of the input matrix. With the factorization present in shared memory, reorganizing the threads allows for aligned writes in both the non-transposed and transposed cases. The additional transposed write-back imposes a slight penalty on the factorization, but allows for a fast implementation of the transposed triangular solve, which is basically a reflection of the non-transposed triangular solve (Fig. 11).

Finally, the last issue to address is the potential low occupancy problem created by the single-warp implementation. High occupancy is important for bandwidth bound kernels. At the same time, creating one warp per thread block does not yield high occupancy, because the maximum number of thread blocks per multiprocessor is much lower than the maximum number of warps per multiprocessor. The solution to this problem is simple: each thread block is given multiple warps, with each warp responsible for a different triangular solve (Fig. 12). Each warp is completely independent. The optimal number of warps per block depends on the hardware (see Section 6.2).

5 Hardware and Software Setup

The GPU system is an NVIDIA Keper K40c card, with 15 multiprocessors, each containing 192 CUDA cores. The theoretical peak floating point performance in single precision is 4,290 Gflop/s. The GPU contains 11.25 GB of ECC-protected DRAM with a theoretical bandwidth of 288 GB/s. Each multiprocessor contains 64 KB of shared memory/L1 cache with a theoretical peak bandwidth of 216 GB/s. Other important hardware characteristics include: maximum number of active threads per multiprocessor (2,048), maximum number of thread blocks per multiprocessor (16), maximum number of threads per block (1024), maximum number of registers per thread (256), size of the shared memory (configurable to 16, 32, or 48 KB at the expense of L1 cache). In terms of the software stack, CUDA version 7.0 [19] was used for compiling all codes and producing cuBLAS performance numbers.

The CPU system is an Intel Sandy Bridge Xeon E5-2670 running at 2.6 GHz, in a two-socket configuration featuring 8 cores in each socket, with a theoretical peak of 666 Gflop/s (single precision). Each core features a 32 KB L1 data cache, 32 KB L1 instruction cache, and 256 KB L2 cache. Each socket

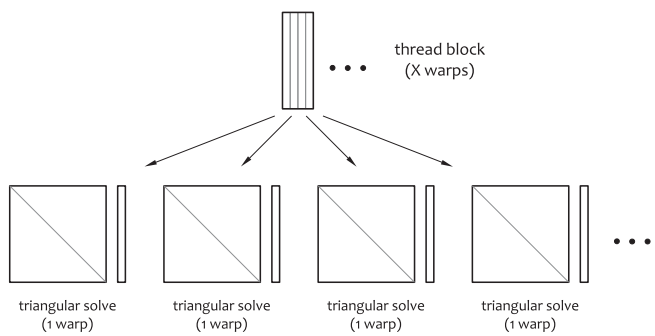

Fig. 12. Intra-thread-block parallelization of multiple triangular solves.

has 20,480 KB of shared L3 cache. There is also 64 GB of accessible main memory with a theoretical bandwidth of 51 GB/s. The CPU implementation is based on the routines from Intel *Math Kernel Library* (MKL) version 11.1.2 [20].

6 Autotuning

The optimal parameters for the factorization kernel are not obvious and not easy to derive by an analytical formula. Therefore the factorization calls for a real autotuning sweep (Section 6.1). On the other hand, parameterization of the triangular solves is rather straightforward and the optimal settings are a simple function of the hardware specification (Section 6.2).

6.1 Factorization

To achieve high performance, a classic heuristic automatic software tuning methodology is applied, where a large number of kernels are generated and run, and the fastest ones are identified. For every size of the factorization (N), different values are possible for the width of the panel (NB) and the two dimensions of the thread block. The kernel is generalized so that any value for NB can be used for a given N, and any shape of the thread block can be used for a given NB, i.e., there are no artificial constraints on the shapes and sizes. Correctness is checked against LAPACK code running on a CPU. Unlike for LU and QR factorizations, there is no option to check against cuBLAS, since cuBLAS does not provide batched Cholesky factorization at this time. All runs reported in this work are done for the batch size of 10,000. At this batch size, the kernel is close to its asymptotic performance, such that increasing the batch size does not noticeably increase the Gflop/s rate.

The objective of batched factorizations is to deliver good performance for a large number of small factorizations. Here, tuning is done for all matrices in the range of 5×5 to 100×100, inclusively, i.e., $5 \leq N \leq 100$. For each matrix size (N), the following panel widths (NB) are taken: $N, \lceil N/2 \rceil, \lceil N/3 \rceil, \lceil N/4 \rceil, ..., 1$. For example, for a matrix of size $N = 33$, the set of values for NB is: 33, 17, 11, 9, 7, 6, 5, 4, 3, 2, and 1. For each shape of the panel ($N \times NB$), the height of the thread block goes through all powers of two smaller than N and the first one larger than N, and the width of the thread block goes through all powers of two smaller than NB and the first one larger than NB. For example, for a panel of size 33×11, the values for the height of the thread block (`blockDim.x`) are: 1, 2, 4, 8, 16, 32, and 64, and the values for the width of the thread block (`blockDim.y`) are: 1, 2, 4, 8, and 16. Additionally, two simple filters are applied. Thread blocks with the total number of threads not divisible by the warp size (32) are rejected, as well as thread blocks with the total number of threads exceeding the hardware maximum for the Kepler architecture (1,024). This is not a fully exhaustive sweep. In principle more panels could be tried for each matrix size, and more thread block shapes could be tried for each panel. However, the current process of generation, compilation, and timing is still sequential and occupies a single GPU for a long time. Future plans include implementing a parallel process, possibly massively parallel, to do a more exhaustive sweep.

Fig. 13 shows the number of kernels generated for each matrix size. It also shows the number of kernels that successfully compiled after generation, and the number of kernels that successfully launched after compilation. Starting at size 65, a small number of kernels fail to compile. What happens is that the nvcc compiler produces a segmentation fault, with the most likely reason being the overwhelming size of the unrolled code. A small number of kernels that successfully compile, fail to launch, also starting at size 65. The likely reason for this is exceeding some resource limit, like the total number of registers required to launch the kernel.

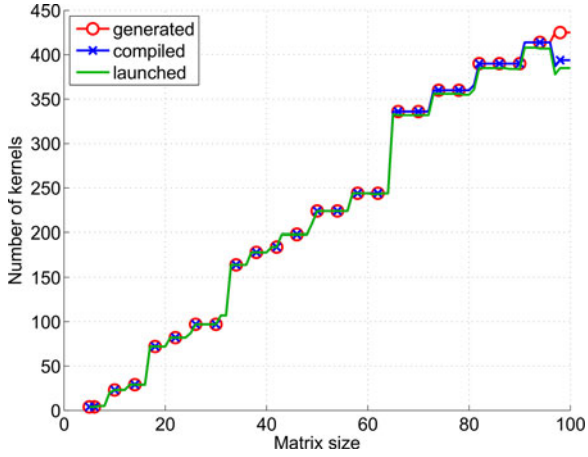

Fig. 13. Number of kernels generated for each matrix size.

6.2 Forward and Backward Substitution

The only parameter of the triangular solve kernels is the number of warps per block. The objective is to maximize the memory bandwidth, and the easiest way to accomplish this is to maximize occupancy. In the Kepler architecture, the maximum number of blocks per multiprocessor is 16, while the maximum number of warps per multiprocessor is 64. Therefore, an implementation that launches a single warp per block reaches only 25 percent occupancy. Increasing the number of warps per block increases the performance, up to four warps per block, at which point the maximum occupancy (100 percent) is reached, and further increasing the number of warps keeps the performance at the same level. Alternatively, one could create two blocks with 32 warps each, which is the maximum number of warps per block.

7 Results

Experimental results are presented in the following sections. Section 7.1 provides details about the experiences when autotuning the parameterized Cholesky factorization kernel. Section 7.2 investigates performance benefits when relaxing the IEEE floating-point compliance of this kernel. Sections 7.3 and 7.4 present a performance comparison of the developed kernels to routines taken from NVIDIA's cuBLAS library. Finally, Section 7.5 combines the kernels into a complete solution process for SPD linear systems, and relates the respective contributions to the overall runtime for different problem sizes.

These results focus on single precision for several reasons. Obviously, single precision handles larger problems, as the memory size poses a hard constraint on all levels. Single precision also has the largest parameter space for possible configurations, leading to the largest number of kernels

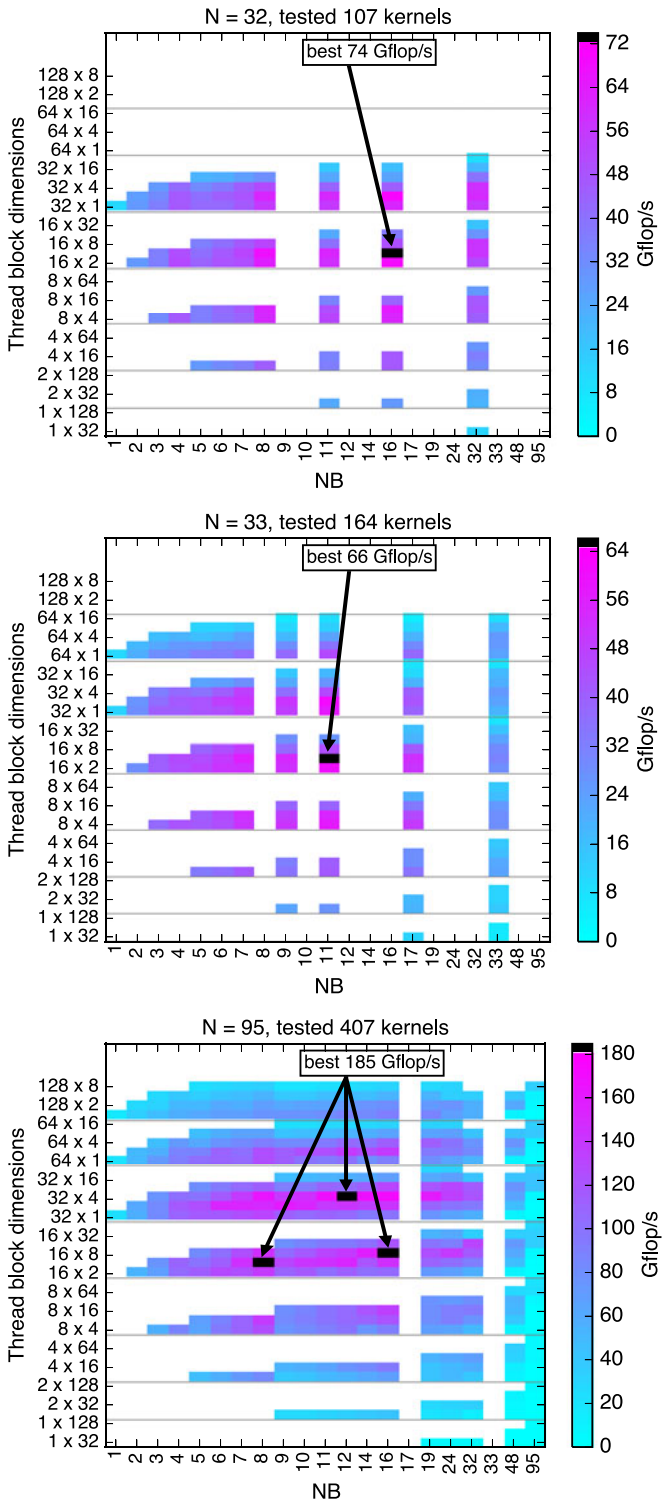

Fig. 14. Performance of all tested kernels for three different matrix sizes, $N = 32$, 33, and 95.

to test. Other precisions impose tighter memory constraints, pruning the parameter search space, and leading to fewer kernels. Kernels for other precisions can be generated from the same Cholesky and triangular solve templates using the appropriate parameter space.

7.1 Autotuning Results

For each matrix size (N), the autotuning search space has three parameters: the panel width (NB) and the two

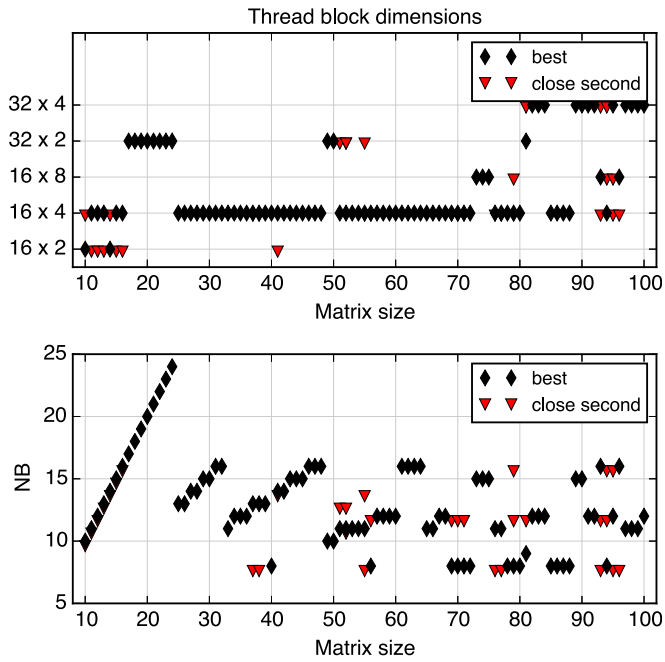

Fig. 15. Optimal tuning parameters for each matrix size.

dimensions of the thread block (*blockDim.x* and *blockDim.y*). Fig. 14 shows the performance achieved for all the kernels tested for three example matrix sizes, $N = 32$, 33, and 95. Along the vertical axis are the different thread block shapes, grouped by *blockDim.x*, then by *blockDim.y*. Along the horizontal axis are different panel widths. The best performing kernel is highlighted in black. For $N = 95$, three different kernels achieved substantially the same top performance at 185 Gflop/s. Blank areas are configurations that were not tested, either because the kernel was invalid or because the autotuning sweep was not exhaustive. As the matrix size increases, there are more valid kernels, going from 107 kernels for $N = 32$ to 407 kernels for $N = 95$. A small change in the matrix size can substantially affect the optimal parameters. Going from $N = 32$ to 33, the optimal NB changed from 16 to 11, which evenly divide 32 and 33, respectively. In this case the thread block shape was the same for both, 16×4.

The optimal tuning parameters for all matrix sizes are given in Fig. 15. While 39 different thread block shapes were tested, only five shapes produce optimal performance. The most common optimal thread block shape was 16×4. Even when it was not the best, it still achieved 85 percent or better of the maximum performance, as shown by the blue circles in Fig. 16. In all cases, a tall thread block (e.g., 16×4) rather than a wide thread block (e.g., 4×16) achieved good performance. This corresponds with a column of the thread block reading contiguous blocks of data to achieve coalesced memory reads on the GPU. The fact that only five thread block shapes were needed for optimal performance suggests that a more exhaustive tuning sweep should focus on testing more panel widths (NB), rather than more thread block shapes.

For 25 matrix sizes, there were multiple kernels that achieved 98 percent or better of the maximum performance, which are shown by the red triangle "close seconds" in Fig. 15. For instance, from $N = 10$ to 16, both 16×4 and 16×2 thread block shapes achieved top performance. For $N = 95$, three combinations achieved close

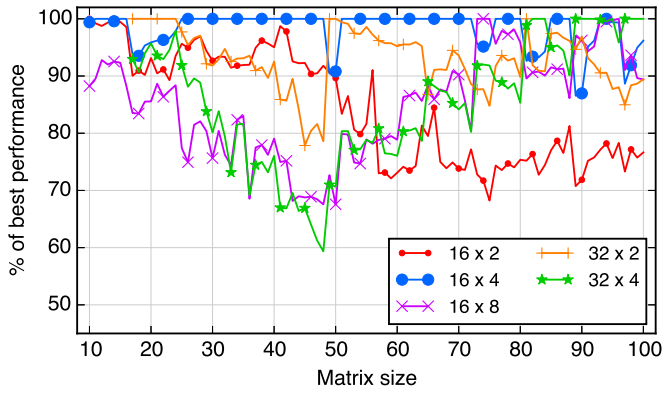

Fig. 16. Performance of selected thread block dimensions for all matrix sizes. For each dimension, the best performance over all NB is shown.

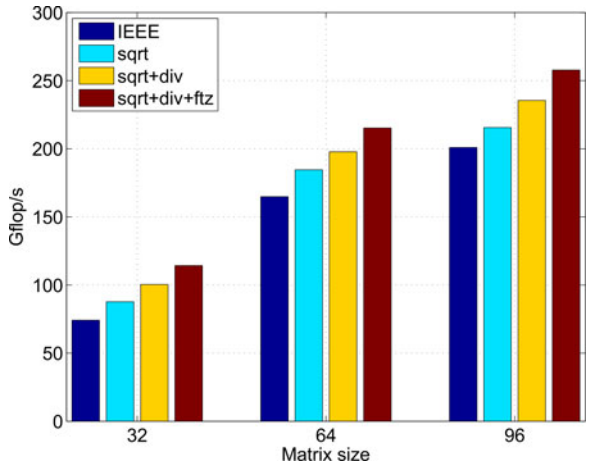

Fig. 17. Performance improvements from relaxing compliance with the IEEE floating-point standard for the computation of square root and reciprocal, and the treatment of denormalized numbers.

to 185 Gflop/s: 32×4 with $NB = 12$, 16×8 with $NB = 16$, and 16×4 with $NB = 8$. This allows some flexibility in choosing which kernel to use.

The optimal panel width (NB) had much more variability than the thread block shape. Fig. 15 shows that up to $N = 24$, the best panel width was $NB = N$. After this, the best was $NB = \lceil N/2 \rceil$ up to $N = 32$, then $NB = \lceil N/3 \rceil$ up to $N = 48$, except at $N = 40$. Beyond that point, there was no clear pattern relating the optimal NB to N.

To demonstrate how the optimal configuration for one matrix size performed on other matrix sizes, Fig. 16 shows the performance of the five optimal thread block shapes for all matrix sizes. The performance here is the maximum over all NB for each matrix size. Using the wrong thread block shape loses up to 40 percent of the potential performance. However, the 16×4 shape achieved 85 percent of the optimal performance in all cases, and is the clear all-around winner.

7.2 Relaxing IEEE Compliance

The above performance results are generated using IEEE floating-point compliant operations. Performance can be further improved by relaxing the requirement of IEEE compliance. Transcendental functions, including reciprocal and reciprocal square root, are computed by the GPU's *Special Function Units* (SFUs) [21], and require a few extra instructions to achieve IEEE compliance. Because of that, they show up as function calls in Fig. 9. If the application can tolerate some loss of precision, the rules can be relaxed by using compiler flags. The `--prec-sqrt` flag allows for approximate square root, the `--prec-div` flag allows for approximate reciprocal, and the `--ftz=true` flag allows for flushing of denormals to zero. Fig. 17 shows the effects of adding the flags, one by one, to the list of compiler options, using the best kernel configuration from the autotuning sweep in Section 7.1. For the case of a 96×96 factorization, the IEEE compliant run delivers 200 Gflop/s of performance. Adding `--prec-sqrt` increases the performance to 215 Gflop/s, adding `--prec-div` increases the performance to 235 Gflop/s, and adding `--ftz` increases the performance to 257 Gflop/s. All these flags are contained in the `--use_fast_math` flag, which also includes the `--fmad` flag. The `--fmad` flag enables the contraction of a floating-point multiply and add into a floating-point multiply-add operation (FMAD, FFMA, or DFMA), and is enabled by default. The optimizations remove the function calls from the assembly code, and replace them with the MUFU instruction (FP Multi-Function Operator). A comparison of the factorization results did not detect any differences coming from the use of fast-math. Section 7.3 reports performance for both IEEE compliant and fast-math versions.

7.3 Performance of the Batched Cholesky Factorization

The performance of the batched Cholesky factorization is shown in Fig. 18, where it is compared to NVIDIA's cuBLAS library on GPU and Intel's MKL on CPU. Unfortunately, cuBLAS does not include a batched Cholesky implementation, however, batched LU and batched QR are provided and serve as reference points. Neither LU nor QR leverage the properties of an SPD matrix. In contrast to Cholesky, LU contains pivoting. Applying LU to a symmetric positive definite matrix reduces the overhead, as no row-swapping is needed.

The CPU implementation uses OpenMP to parallelize the loop over the batch of matrices, and calls single-threaded MKL on each core for each Cholesky factorization and

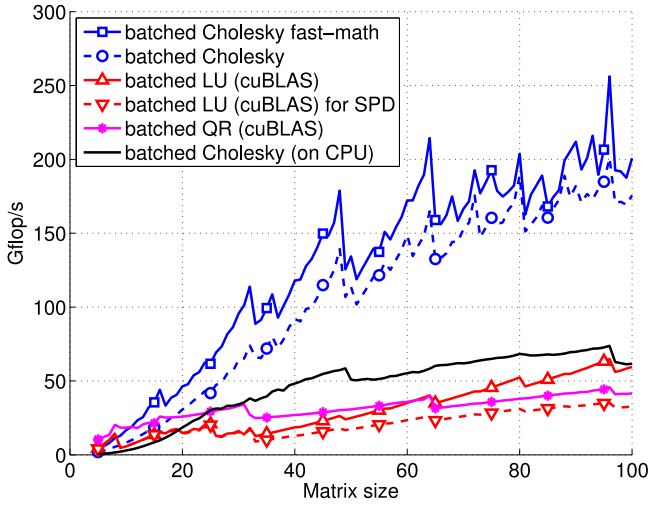

Fig. 18. Performance comparison between the developed batched Cholesky, batched QR, and batched LU applied to a random matrix and to a symmetric positive definite matrix.

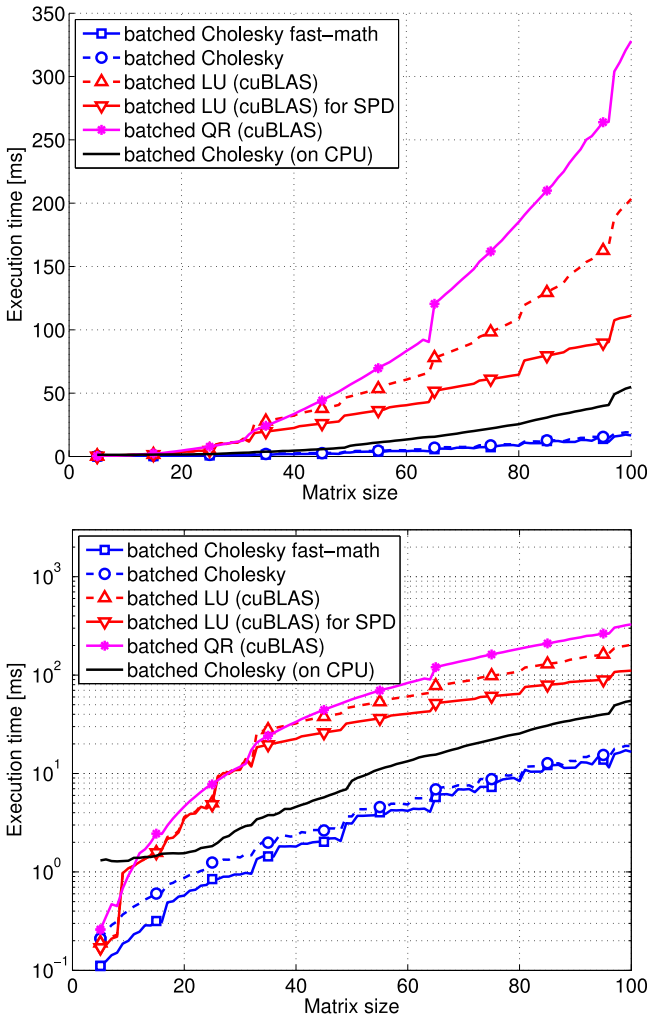

Fig. 19. Runtime comparison, in linear and log scale, between the developed batched Cholesky, batched QR, and batched LU applied to a random matrix and to a symmetric positive definite matrix. The reported runtime is to factorize a batch of 10,000 SPD matrices.

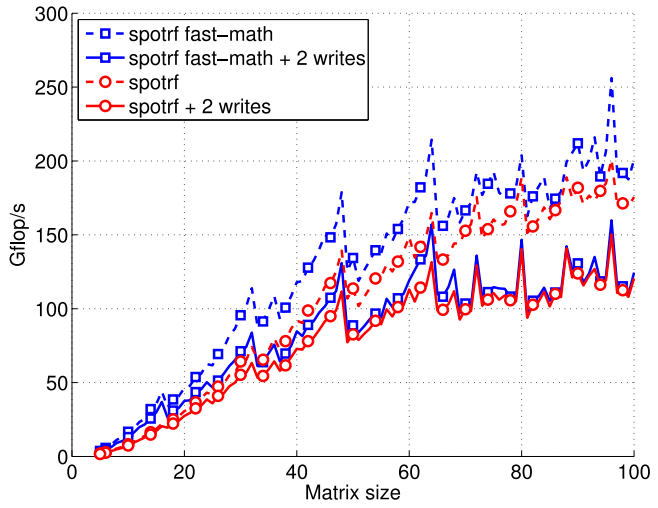

Fig. 20. Performance penalty of writing the Cholesky factors to main memory twice, in transposed and non-transposed layout. The performance reported is for the IEEE compliant batched Cholesky factorization, and the version based on relaxed IEEE standard, both were applied to a batch of 10,000 SPD matrices.

triangular solve. This implementation has linear scaling with the number of CPU cores, and is much faster than calling multi-threaded MKL for these small sizes. In results presented here, all 16 available cores were used.

Note that Fig. 18 compares performance, which in some sense is an unfair comparison as LU and QR execute two and four times, respectively, the operation count of Cholesky; for a comparison of runtimes, see Fig. 19. This gives some disadvantage to the Cholesky performance, measured in Gflop/s, as LU and QR have higher compute intensity, and explains the high performance of the batched QR for small sizes. The performance scaling of QR is, however, smaller than for batched LU, which indicates that much effort was spent on tuning for small sizes. For batched LU with sizes smaller than 32, the performance is independent of whether a general or an SPD matrix is factored, indicating that the factorization is handled in fast local memory where row swapping does not impact runtime. An interesting result is that the performance of Intel's batched Cholesky is higher than the performance of the batched algorithms provided by NVIDIA for linear systems with dimensions larger than 24, however below the performance of the batched Cholesky developed in this paper. With increasing dimension, the batched Cholesky achieves a speedup factor of 3 over NVIDIA's batched LU applied to a symmetric positive definite matrix. Allowing for fast-math, the performance of the batched Cholesky exceeds 250 Gflop/s for a set of matrices with dimension 96×96.

For completeness, Fig. 19 reports the runtime needed for the factorization of a batch of 10,000 matrices. For system dimensions up to 40, the batched LU and the batched QR from NVIDIA's cuBLAS library have similar factorization runtimes, while the developed batched Cholesky is about five times faster. For larger matrix sizes, NVDIA's batched LU is faster than NVIDIA's batched QR, which executes twice as many flops. Intel's batched Cholesky executes faster than NVIDIA's batched LU for systems of size larger than 16. The runtime of the developed batched Cholesky is an order of magnitude faster than NVIDIA's QR factorization time, and a factor of 5 below NVIDIA's Batched LU applied to an SPD matrix.

7.4 Performance of the Batched Triangular Solves

As elaborated in Section 4.2, the batched triangular solve can be realized in a more efficient way if the triangular factor is written twice to main memory, in transposed and non-transposed fashion. The penalty to the performance of the batched Cholesky is visualized in Fig. 20 for both the IEEE complaint and fast-math implementations. An interesting observation is that the performance differences between the IEEE compliant batched Cholesky and the fast-math version become negligible when writing the results twice, implying that the kernel becomes memory bound. This motivates the rest of the paper to focus on the IEEE compliant implementation.

Fig. 21 compares the performance of the developed triangular solve working in local memory against the reference implementations available in NVIDIA's cuBLAS library and the CPU version based on Intel's MKL. The graph shows performance for both the non-transposed forward substitution and the transposed backward substitution, labeled *strsv NT* and *strsv T*, respectively. The developed batched

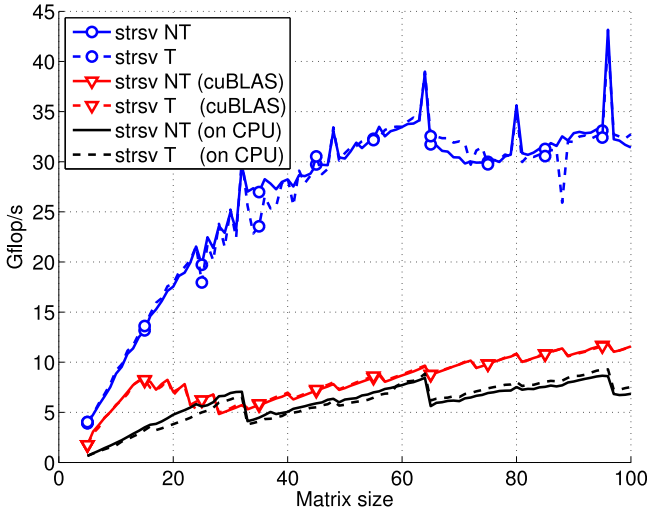

Fig. 21. Performance comparison between the cuBLAS batched solve, a CPU implementation based on Intel's solve in MKL, and the developed batched triangular solves for the non-transposed forward substitution and the transposed backward substitution. The target problem is a batch of 10,000 factorized SPD matrices where the triangular factors are stored in non-transposed and transposed fashion.

triangular solve working in local memory shows small performance differences between the forward and the backward substitution, while for the cuBLAS and the MKL-based CPU implementation, the differences are negligible. Intel's batched triangular solve shows a characteristic pattern indicating that the implementation works with submatrices of size 32. Neglecting the sizes 28-32, its performance is below the performance of NVIDIA's batched triangular solve. The new batched triangular solve proposed in this paper outperforms NVIDIA's and Intel's batched triangular solve by a factor of 2 for small sizes and a factor of 4 for large sizes.

7.5 Performance of Solving a Sequence of SPD Problems

The complete solution process of a sequence of SPD linear systems can be realized by calling the batched Cholesky factorization, the batched forward substitution, and the batched backward substitution as three distinct kernels. In Fig. 22, we present the size-dependent contributions of the distinct kernels to the overall execution time when solving a set of 10,000 SPD matrices. One observation is that the runtime of the batched Cholesky factorization grows faster with the

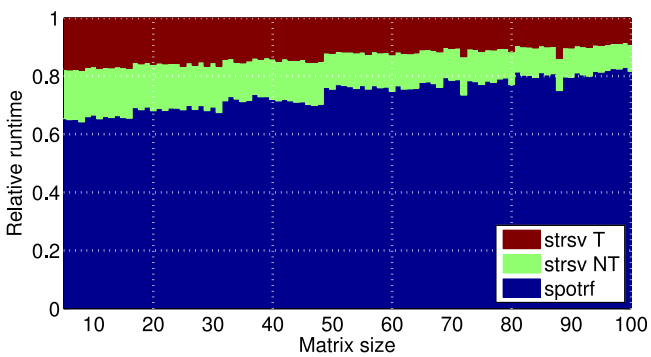

Fig. 22. Relative contribution of batched Cholesky factorization, batched forward substitution and batched backward substitution to the overall runtime when solving a set of 10,000 SPD matrices.

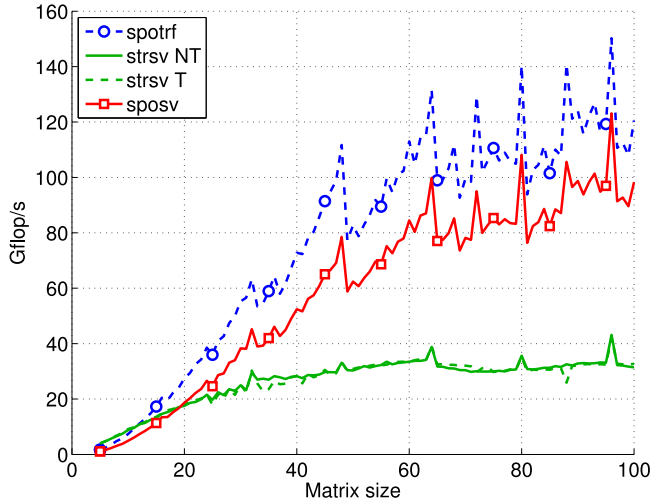

Fig. 23. Performance of the IEEE compliant batched Cholesky factorization, the batched triangular solves, and the solution process combining the factorization with the forward and backward substitutions. The target problem is a batch of 10,000 SPD matrices.

problem size than the runtime for the batched solves, with the solves taking from 36 percent for small sizes, down to 17 percent for large sizes. This is expected as the factorization is $O(N^3)$, while the solves are $O(N^2)$ complexity.

Finally, Fig. 23 reports the performance for the IEEE compliant batched Cholesky factorization writing the Cholesky factors to main memory twice, the batched triangular solves, and the performance of the overall solution process, combining the factorization with forward and backward substitutions. The performance for the batched solves shows less variations than the batched factorization, asymptotically approaching 30 Gflop/s, with isolated peaks exceeding 40 Gflop/s. The performance of the algorithm combining the three kernels, each handling most operations in local multiprocessor memory, achieves up to 100 Gflop/s for large system sizes, with an isolated peak exceeding 120 Gflop/s for the matrix size 96×96.

8 Conclusions and Future Research

In the course of this work, a batched Cholesky factorization for GPUs was developed, that handles most of the factorization and solve in the fast multiprocessor memory. Using the BEAST autotuning framework, optimal kernel configurations were identified that provide performance significantly higher than a CPU counterpart or similar routines taken from NVIDIA's cuBLAS library. To this end, routines were developed for handling the forward and backward substitutions (triangular solves). A comparison to equivalent routines from the cuBLAS library revealed significant performance advantages of the proposed implementations. The performance of the complete solution process, where the factorization and the forward and backward substitutions are realized in three distinct kernels, exceeds 120 Gflop/s on a state-of-the-art GPU, for matrix sizes up to 100×100. Future research will correlate the performance of the different kernel configurations to the GPU metrics, like occupancy, memory bandwidth, and executed instructions. Also, the QR and LU factorization will be addressed in a similar fashion. Finally, a different research effort will investigate the overall resource efficiency by also taking the energy balance into account.

ACKNOWLEDGMENTS

This work is supported by grant #SHF-1320603: "Benchtesting Environment for Automated Software Tuning (BEAST)" from the National Science Foundation, by the Department of Energy grant #DE-SC0010042, and by the NVIDIA Corporation. Furthermore, the authors would like to thank the Oak Ridge National Laboratory for access to the Titan supercomputer, where the infrastructure for the BEAST project is being developed.

REFERENCES

[1] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra, "Dense linear algebra solvers for multicore with GPU accelerators," in *Proc. IEEE Int. Symp. Parallel Distrib. Process., Workshops Phd Forum*, 2010, pp. 1–8.
[2] H. Anzt, W. Sawyer, S. Tomov, P. Luszczek, and J. Dongarra, "Acceleration of GPU-based Krylov solvers via data transfer reduction," *Int. J. High Perform. Comput.*, vol. 29, pp. 366–383, 2015.
[3] J. Kurzak, S. Tomov, and J. Dongarra, "Autotuning GEMM kernels for the Fermi GPU," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 11, pp. 2045–2057, Nov. 2012.
[4] Y. Hu, Y. Koren, and C. Volinsky, "Collaborative filtering for implicit feedback datasets," in *Proc. 8th IEEE Int. Conf. Data Mining*, 2008, pp. 263–272.
[5] H. Bruck, S. McNeill, M. Sutton, and W. Peters, III, "Digital image correlation using Newton-Raphson method of partial differential correction," *Exp. Mech.*, vol. 29, pp. 261–267, 1989.
[6] M. Gates, M. T. Heath, and J. Lambros, "High-performance hybrid CPU and GPU parallel algorithm for digital volume correlation," *Int. J. High Perform. Comput. Appl.*, vol. 29, pp. 92–106, 2015.
[7] X. Ma and G. R. Arce, *Computational Lithography* Hoboken, NJ, USA: Wiley, 2011, vol. 77.
[8] M. Moharam and T. Gaylord, "Rigorous coupled-wave analysis of metallic surface-relief gratings," *J. Optical Soc. Am. a*, vol. 3, no. 11, pp. 1780–1787, 1986.
[9] T. George, V. Saxena, A. Gupta, A. Singh, and A. R. Choudhury, "Multifrontal factorization of sparse SPD matrices on GPUs," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2011, pp. 372–383.
[10] X. Lacoste, P. Ramet, M. Faverge, I. Yamazaki, and J. Dongarra, "Sparse direct solvers with accelerators over DAG runtimes," INRIA Bordeaux, Tech. Rep. RR-7972, 2012.
[11] O. Villa, M. Fatica, N. Gawande, and A. Tumeo. (2013). Power/performance trade-offs of small batched LU based solvers on GPUs. in *Euro-Par*, ser. Lecture Notes in Computer Science, F. Wolf, B. Mohr, and D. an Mey, Eds., vol. 8097, pp. 813–825. [Online]. Available: http://dblp.uni-trier.de/db/conf/europar/europar2013.html#VillaFGT13
[12] O. Villa, N. Gawande, and A. Tumeo, "Accelerating subsurface transport simulation on heterogeneous clusters," in *Proc. IEEE Int. Conf. Cluster Comput.*, Indianapolis, IN, USA, Sep. 23-27, 2013, pp. 1–8.
[13] T. Dong, A. Haidar, P. Luszczek, J. A. Harris, S. Tomov, and J. Dongarra, "LU factorization of small matrices: Accelerating batched DGETRF on the GPU," in *Proc. IEEE 6th Int. Symp. Cyberspace Safety Security High Perform. Comput. Commun., IEEE 11th Int. Conf. Embedded Softw. Syst.*, 2014, pp. 157–160.
[14] T. Dong, A. Haidar, S. Tomov, and J. Dongarra, "A fast batched cholesky factorization on a GPU," in *Proc. 43rd Int. Conf. Parallel Process.*, 2014, pp. 432–440.
[15] A. Haidar, T. Dong, P. Luszczek, S. Tomov, and J. Dongarra, "Towards batched linear solvers on accelerated hardware platforms," in *Proc. 20th ACM SIGPLAN Symp. Principles Practice Parallel Programm.*, 2015, pp. 261–262.
[16] A. Haidar, T. Dong, P. Luszczek, S. Tomov, and J. Dongarra, "Batched matrix computations on hardware accelerators based on GPUs," *Int. J. High Perform. Comput. Appl.*, p. 1–16, 2015.
[17] *NVIDIA CUDA CUBLAS Library Programming Guide*, 1st ed. NVIDIA Corp., Jun. 2007.
[18] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammerling, A. McKenney et al., *LAPACK Users' Guide.* Philadelphia, PA, USA: SIAM, 1999, vol. 9.
[19] *NVIDIA CUDA TOOLKIT 7.0*, NVIDIA Corp., Mac. 2015.
[20] "Intel Math Kernel Library for Linux OS," Document Number: 314774-005US, Oct. 2007, Intel Corporation.
[21] S. F. Oberman and M. Y. Siu, "A high-performance area-efficient multifunction interpolator," in *Proc. 17th IEEE Symp. Comput. Arithmetic*, 2005, pp. 272–279.

Jakub Kurzak received the MSc degree in electrical and computer engineering from the Wroclaw University of Technology, Poland and the PhD degree in computer science from the University of Houston. He is a research director in the Innovative Computing Laboratory, Department of Electrical Engineering and Computer Science, University of Tennessee. His research interests are in high-performance computing with multicore and accelerators.

Hartwig Anzt received the PhD degree in mathematics from the Karlsruhe Institute of Technology (KIT) in 2012. He is a postdoctoral researcher in Jack Dongarra's Innovative Computing Lab (ICL) at the University of Tennessee. His research interests include simulation algorithms, sparse linear algebra, hardware-optimized numerics for GPU-accelerated platforms, communication-avoiding and asynchronous methods, and power-aware computing.

Mark Gates received the BS, MS, and PhD degrees in computer science from the University of Illinois at Urbana-Champaign, in 1998, 2007, and 2011, respectively. He is a research scientist in the Innovative Computing Laboratory at the University of Tennessee, where he researches algorithms for linear algebra on multi-core and GPU-based computers. He also worked at NCSA from 1998 to 2000, on optimizing applications for high-performance wide-area networks. His research interests are in high-performance computing, particularly linear algebra, and GPU computing.

Jack Dongarra holds appointments at the University of Tennessee, Oak Ridge National Laboratory, and the University of Manchester. He specializes in numerical algorithms in linear algebra, parallel computing, use of advanced computer architectures, programming methodology, and tools for parallel computers. He was awarded the IEEE Sid Fernbach Award in 2004, received the first IEEE Medal of Excellence in Scalable Computing in 2008, the first SIAM Special Interest Group on Supercomputing's award for Career Achievement in 2010, and the IEEE IPDPS 2011 Charles Babbage Award. He is a fellow of the AAAS, ACM, IEEE, and SIAM and a member of the National Academy of Engineering.