INVITED PAPER

Autotuning Numerical Dense Linear Algebra for Batched Computation With GPU Hardware Accelerators

This paper discusses automatic performance tuning for small linear algebra kernels, which are important building blocks in many engineering and science applications.

By Jack Dongarra, *Fellow IEEE*, Mark Gates, Jakub Kurzak, Piotr Luszczek, and Yaohung M. Tsai

ABSTRACT | Computational problems in engineering and scientific disciplines often rely on the solution of many instances of small systems of linear equations, which are called batched solves. In this paper, we focus on the important variants of both batch Cholesky factorization and subsequent substitution. The former requires the linear system matrices to be symmetric positive definite (SPD). We describe the implementation and automated performance engineering of these kernels that implement the factorization and the two substitutions. Our target platforms are graphics processing units (GPUs), which over the past decade have become an attractive high-performance computing (HPC) target for solvers of linear systems of equations. Due to their throughput-oriented design, GPUs exhibit the highest processing rates among the available processors. However, without careful design and coding, this speed is mostly restricted to large matrix sizes. We show an automated exploration of the implementation space as well as a new data layout for the batched class of SPD solvers. Our tests involve the solution of many thousands of linear SPD systems of exactly the same size. The primary focus of our techniques is on the individual matrices in the batch that have dimensions ranging from 5-by-5 up to 100-by-100. We compare our autotuned solvers against the state-of-the-art solvers such as those provided through NVIDIA channels and publicly available in the optimized MAGMA library. The observed performance is competitive and many times superior for many practical cases. The advantage of the presented methodology lies in achieving these results in a portable manner across matrix storage formats and GPU hardware architecture platforms.

KEYWORDS | Dense numerical linear algebra; performance autotuning

I. INTRODUCTION

For large dense matrices, software libraries for numerical linear algebra methods are known to achieve high efficiency (in terms of the fraction of peak performance) for solving dense linear systems on graphics processing unit (GPU)-accelerated hardware [1]. However, good performance for many simultaneous small linear systems has been hard to achieve and continues to be a challenge. The available parallelism is inherently limited when dealing with small matrices, e.g., when sizes are of order 100 (matrix dimension is 100×100) or smaller. As a result, most methods fail to fully utilize the highly parallel computing hardware units that are available in today's high-performance computing (HPC) platforms. This is where batched mode of operation is helpful: in such a mode, the implementation is exposed to a significant amount of parallelism due to availability of a large set of small linear systems that can be accessed by the software library at the same time. This enables the linear algebra routine to use

Manuscript received July 3, 2017; revised May 31, 2018; accepted August 27, 2018. Date of publication September 28, 2018; date of current version October 25, 2018. This work was supported by the National Science Foundation under Grant #1642441: SI2-SSE: BONSAI: An Open Software Infrastructure for Parallel Autotuning of Computational Kernels, by the Department of Energy under Grant #DE-SC0010042, and by NVIDIA Corporation. *(Corresponding author: Piotr Luszczek.)*

J. Dongarra is with the Department of Electrical Engineering and Computer Science, The University of Tennessee, Knoxville, TN 37996 USA, with the Oak Ridge National Laboratory, Oak Ridge, TN 37831 USA, and with the University of Manchester, Manchester M13 9PL, U.K.

M. Gates, J. Kurzak, P. Luszczek, and **Y. M. Tsai** are with the Department of Electrical Engineering and Computer Science, The University of Tennessee, Knoxville, TN 37996 USA (e-mail: luszczek@icl.utk.edu).

Digital Object Identifier 10.1109/JPROC.2018.2868961

the parallel hardware in a significantly more efficient manner. On throughput-oriented streaming processors—the main execution units inside modern GPU accelerators—there are additional gains to be made because of the reduced overhead of launching individual kernels for each matrix; instead, it is possible to dispatch a single kernel as a batched function invocation. Making multiple kernel calls (one for each linear system) cannot be efficiently overlapped even with the communication of matrix data because, in the most common case, the matrices are already present in the accelerator's main memory.

Every one of a GPU's streaming multiprocessors (SMs) has a fast local memory with a large register file, shared memory, and cache-like scratch buffer that is, for optimal performance, managed manually by the programmer. Reusing data stored in these fast local memory caches is essential for achieving high performance by exploiting data locality [2], [3]. We combine these techniques with a data layout based on intermatrix interleaving of entries, which further helps to match the speed of the hardware processing units with loading of matrix elements from the main GPU memory.

The engineering and scientific applications that often need large sets of small linear solves are constituent in a surprising number of fields in computational science. Thus, they are a perfect match for batched operations on GPUs. Perhaps the most representative example from data analytics is the alternating least squares (ALS) method [4]. There are additional examples in digital volume correlation in experimental mechanics [5], [6], and rigorous coupled-wave analysis (RCWA) in computational lithography [7], [8]. Batched processing may also be applied as the basic building blocks to construct solvers for large sparse linear systems, where the coefficient matrices have many small dense blocks [9], [10].

II. RELATED WORK

Currently, there is significant interest in batched matrix operations. Of the major computing processor vendors, both NVIDIA with cuBLAS [11] and Intel with the Math Kernel Library (MKL) [12] provide extensive sets of batched routines for basic linear algebra tasks, and so does the Matrix Algebra on GPU and Multicore Architectures (MAGMA) library [13] from the University of Tennessee. Numerous papers have been published about the development and optimization of batched routines [14]–[18]. This paper is a followup on our previous work on the batched Cholesky factorization for smalled matrices [19]. The direct motivation for this work came from the ALS algorithm for recommender systems [20]. In our previous paper, we looked into the development of batched routines for the canonical columnwise data layout. Here we are investigating alternative data layouts for batches of extremely small matrices.

Our autotuning methodology is based on an automated performance engineering approach that we pioneered with the Automatically Tuned Linear Algebra Software (ATLAS) [21], which has since then grown into a vibrant field of experimental performance optimization guided by execution profiles. To name a few early efforts, we could start with Portable High Performance ANSI C (PHiPAC) [22] that generated code for superscalar processors implementing dense linear algebra operations. Sparse matrix computations were targeted by the optimized sparse kernel interface (OSKI) [23] and the fastest Fourier transform (FFT), and similar transforms were optimized by FFT in the West (FFTW) [24] and Spiral [25]. In fact, Spiral's automated code synthesis has recently addressed matrix–matrix multiply [26]. To the best of our knowledge, these projects do not target their autotuning efforts specifically for accelerators. Also, they concentrate on using the human expert knowledge of the tuned kernel and embed it as the core of the project and an integral part of the implementation code. Our approach is to expose this knowledge as a generic component in the form of user-defined templates and/or stencils. Finally, there are domain-specific languages (DSLs) designed specifically for the purpose of autotuning parallel scientific codes [27], [28]. A more exhaustive survey of recent advances in the area of autotuning is available elsewhere [29]. The recent work that targeted GPU-specific autotuning [19], [30], [31] is also relevant and some of the results are updated here.

1) Register file
2) Shared memory and/or Level 1 cache
3) Read-Only data cache
4) Level 2 cache
5) Device main memory

Fig. 1. *Architectural components of NVIDIA GPUs.*

III. HARDWARE PRELIMINARIES

For the sake of completeness, in this section, we provide an overview of the GPU-based hardware accelerators and their design features that are the most important from the standpoint of implementing the batched kernels that are the subject of this paper—Cholesky factorizations/solves.

The two most prominent aspects of modern compute-oriented GPUs are the single-instruction–multiple-threads (SIMT) processing model and the memory hierarchy. The code that mismanages either of them would quickly lower the performance rate below the optimal level.

Fig. 1 lists the basic elements of an architecture of modern NVIDIA and AMD GPUs. The most basic execution unit of an NVIDIA GPU is referred to as a CUDA core. One such core can possibly execute basic floating-point instructions at a throughput of one instruction per cycle. However, a single CUDA core may not follow an independent instruction stream that is completely unrelated to the nearby cores' instruction streams. This is in stark contrast to a more traditional core of a central processing unit (CPU). Instead, a set of 32 GPU cores has to follow the same execution path. Therefore, the basic unit of scheduling is

a warp, which is a set of 32 threads mapped by the GPU thread scheduler onto the CUDA cores.

CUDA cores are organized into multiprocessors, the names of which are shortened as SM or SMX. The Kepler architecture features multiprocessors that contain 192 cores, and a single GPU chip contains up to 15 streaming multiprocessors; this makes a total of up to 2880 CUDA cores per chip. NVIDIA Maxwell, Pascal, and Volta designs lower the per-multiprocessor core count to 64 but increase the per-chip multiprocessor count to 24, 60, or 80, respectively. In addition to the cores and their instruction-scheduling logic, the Kepler multiprocessor features a large register file of 65 536 registers of 32-b size and 48 KiB of read-only data cache, and 64 KiB of fast memory, which may be configured as a combination of a flexible/programmable level 1 (L1) cache and/or shared memory.

The fastest memory in the multiprocessor is the register file. The registers in the file are partitioned among threads, and each thread has a private set of registers at the time of execution. The second fastest memory in the multiprocessor is the L1 cache and/or shared memory. Much like in the CPU, the L1 cache is a standard hardware-controlled cache. On the other hand, the shared memory is controlled entirely through software. Two threads may exchange data by storing data pieces from registers into shared memory locations, then executing a synchronization operation, and finally reading the data from a shared memory location to a register. Thus, the shared memory is a scratchpad storage that is specific to GPUs (in contrast to CPUs). Initially, GPUs were exclusively designed for data-parallel workloads, and the individual cores could not communicate directly—collaboration in terms of data exchanges was not possible. The subsequently introduced GPU designs added the shared memory feature and thus allowed handling of more complex tasks, such as exchanging data among cores. In a way, the shared memory can be considered an extension of the register file rather than simply a traditional cache—and the CUDA constructs and compiler optimizations make that connection seamless from the programmer's standpoint.

The main memory, either graphics double data rate (GDDR) or high bandwidth memory (HBM) random access memory (RAM), consists of the slowest memory types in a GPU accelerator. Reads from the main memory must pass through the L2 and L1 caches or the read-only data cache. A GPU's main memory RAM's bandwidth is often a limited quantity and the most scarce resource in the context of batched matrix operations because they fall on the memory-bound side of the computational intensity spectrum. Critically, accessing the main memory must exhibit access coalescing, i.e., the reads execute most efficiently if all threads in a single warp read the same 128-B cache line. If the data being read are not cached, the cache line will be fetched from RAM in a single memory transaction and thus the data traffic is maximized and the control traffic is limited. If distinct threads in a warp access different cache lines, then multiple memory transactions will be issued and thus extra memory bandwidth will be consumed by the control packets with the addresses of the requested cache lines, thus using the bus bandwidth less efficiently.

Algorithm 1 Unblocked, POTF2, right-looking, lower-triangular Cholesky factorization using C-style (zero-based) indexing.

```
1 for k = 0 to n − 1 do
2     A_{k,k} ← √A_{k,k}
3     for m = k + 1 to n − 1 do
4         A_{m,k} ← A_{m,k} / A_{k,k}
5         for n = k + 1 to n − 1 do
6             for m = n to n − 1 do
7                 A_{m,n} ← A_{m,n} − A_{n,k} × A_{m,k}
```

IV. ALGORITHMIC OVERVIEW

A system of linear equations of the form $Ax = b$ with a symmetric positive definite (SPD) dense matrix $A \in \mathbb{R}^{n \times n}$ and real vectors $x, b \in \mathbb{R}^n$ can be solved efficiently with minimal space requirements through a decompositional approach by first computing the lower Cholesky factorization $A = LL^{\mathsf{T}}$, and then using forward and backward substitutions to apply on the right-hand-side vector b through the triangular factors L and L^{T}. In this paper, we focus on all three steps: the factorization and two solve steps—but only for the lower triangular formulation because the upper triangular one, $A = U^{\mathsf{T}}U$, is equivalent and may use very similar techniques without any loss of generality in the exposition.

In Figs. 2–6, the basic linear algebra subprograms (BLASs) and the Linear Algebra PACKage (LAPACK) routine names are used to refer to the building blocks of the different implementations of Cholesky factorization, including:

- POTF2—unblocked Cholesky factorization (of a small square submatrix on the diagonal);
- TRSM—solve for a set of vectors with triangular factor;
- SYRK—rank-k update to a symmetric matrix;
- GEMM—general matrix–matrix multiply.

Algorithm 1 and Fig. 2 show the canonical formulation for computing the factors of the Cholesky factorization without cache-friendly blocking of submatrix operations—this is called POTF2 in LAPACK. The algorithm proceeds along the diagonal from the upper left to the lower right corner of the matrix. As it proceeds, the diagonal elements are subsequently replaced by their square root after the prior updates have completed. These updates comprise divisions of each element in the current column (called a panel column) by the computed diagonal value. Additionally, a rank-1 update is applied to the remaining part of the matrix, i.e., to the right of the current diagonal element—called a trailing submatrix. It is custom for the implementations to operate only on one half of the matrix by assuming

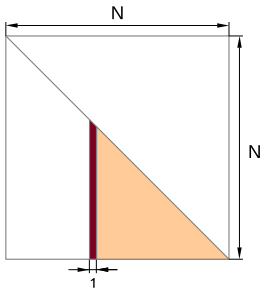

Fig. 2. *Canonical Cholesky factorization (no blocking, right looking, lower triangular).*

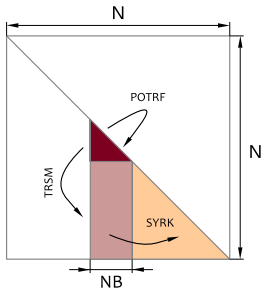

Fig. 3. *Right-looking blocked Cholesky factorization.*

symmetry. Either the lower or upper triangular portion is updated, leaving the other part unchanged. This particular implementation is called unblocked and there exist old codes that use this formulation with one column factored at a time and rank-1 outer-product updates applied to the trailing submatrix. In terms of existing software libraries, this algorithm uses the set of L1 and L2 BLAS: vector–vector and matrix–vector operations are needed. In practice, such implementations produce inferior performance results. This is because such ordering of computations is inherently memory bound, but it can be easily shown that the algorithm is a compute-bound algorithm if recast differently. We proceed to describe such a version of the factorization.

The use of blocking is one of the main optimization methods for the Cholesky factorization, and is shown in Algorithm 2 and Fig. 3. This optimization, which may be viewed as a form of blocking of the inner computational loops, is the primary choice for dense linear algebra routines suitable for cache-based multicore systems. It was heavily used throughout the LAPACK software library. The benefits of blocking stems from replacing most of the L1 and L2 BLAS calls in the canonical unblocked algorithm with L3 BLAS calls, which implement matrix–matrix operations. The performance gains come from leveraging the surface-to-volume effect of dense linear algebra routines. This is the inherent compute-bound nature of these algorithms: they perform $O(n^3)$ floating-point operations on $O(n^2)$ data. The blocked factorization takes advantage of this feature by applying the Cholesky algorithm on one panel of width n_b columns. If we let $1 \ll n_b \ll n$, and we follow by a rank-n_b update, then we take advantage of the inherent data locality property. When described in terms of loop transformations, blocking is similar to loop tiling of the outermost loop in line 1 of Algorithm 1.

Algorithm 2 Right looking variant of blocked Cholesky factorization POTRF of n-by-n matrix A with block size n_b.

```
1 for k = 0 to n/n_b − 1 do
2     POTF2(A_{k,k})
3     for m = k + 1 to n/n_b − 1 do
4         TRSM(A_{k,k}, A_{m,k})
5     for m = k + 1 to n/n_b − 1 do
6         SYRK(A_{m,k}, A_{m,m})
7         for n = k + 1 to m − 1 do
8             GEMM(A_{m,k}, A_{n,k}, A_{m,n})
```

A comparatively important choice to be made in the implementation is whether to use aggressive or delayed evaluation. This goes beyond what is called in loop-transformation parlance loop reordering, and advanced variants of such transformations may be applied in this context [32]–[34]. Another perspective is to look at how much parallelism is exposed to the hardware or whether the memory writes are minimized at the expense of repeated reads. The three main variants of the Cholesky factorization are the right-looking, the left-looking, and the top-looking factorizations. Fig. 3 shows the right-looking variant—it corresponds to aggressive evaluation because immediately after the panel is factored, the updates write to the entire trailing submatrix. In terms of parallelism, this variant sacrifices data locality but exposes a large amount of work quickly, and thus it favors highly parallel hardware. Furthermore, this implementation modifies the data by both reading and writing the entire trailing submatrix. This, therefore, does not map well onto small-sized fast memory structures, the very situation in which the left-looking variant of the factorization performs much better.

Note that the above transformations are followed by automated macro expansion and code generation, which then is finally fed to the NVIDIA compiler that performs additional work, including, for example, loop tiling across all levels of loop nests—provided they meet the requirements of the code mapping internals. These are based on the specific hardware models and are applied to a loop-based code with much concrete (ideally, constant at compile-time) bounds and iteration steps.

Fig. 4 shows the left-looking variant of the Cholesky factorization, which is characterized by its delayed evaluation style. This variant postpones the updates to, and thus the access of, the trailing submatrix. Specifically, the update operations are applied exclusively to the area of the current panel, which is then immediately followed by the panel factorization. The consequence of this is that at every step

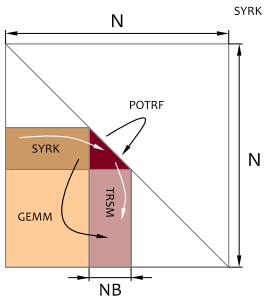

Fig. 4. ***Left-looking blocked Cholesky factorization (the implementation in LAPACK).***

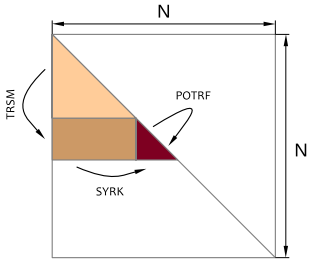

Fig. 5. ***Top-looking block Cholesky factorization.***

of the algorithm, all pending updates are applied from the left-hand side of the matrix, and then, and only then, the panel is factored. Thus, only the panel area is modified (both read from and written to) at any given algorithmic step, and the growing part of the matrix to the left of the current panel is accessed in read-only fashion. For these reasons, the left-looking formulation of the factorization is used as the starting point for implementations that have to take into account a small buffer memory (either a scratch pad or the first level cache) for storing data (commonly the panel data).

Historically speaking, on tiered memory systems with hierarchical storage structures, customized algorithms that efficiently deal with the limited size of the intermediate memory levels have been referred to as out-of-core (OOC) algorithms. The core would refer to the magnetic core storage used in the 1950s, 1960s, and 1970s. Presently, it is considered confusing to use the word "core" in such a context because of the emergence of multicore and many-core processor designs where the word now designates a central processing element. Consequently, since 2006, the word "core" established its new meaning as an independent processing unit that displaced the term CPU. As a consequence, the term out-of-core is often deprecated and the term out-of-memory (OOM) is preferred. Unfortunately, with respect to the content of this paper, the new term may also confuse the reader because we deal extensively with situations where the data exceeds the register file but the algorithmic design forces them to be fully contained inside the main memory. Therefore, we choose nonresident algorithm as the term of choice throughout this text.

One of the primary objectives of nonresident algorithms is, first, to allocate a small amount of read/write storage in the fastest memory and, subsequently, stream that data in read-only mode from the lower level storage, presumably a slower part of the memory hierarchy. Clearly, this is a very generic method that may refer to a number of different situations: hard disk may be the slow memory and the fast memory may be the main memory (RAM). Or slow is RAM and fast is cache. Or slow is cache and fast are registers. Or the main memory of the CPU host against the accelerator device memory. The central concept of the nonresident batched Cholesky factorization and solve is to use the left-looking variant by first placing the matrix panel entries in the fast memory; second, streaming in the pending updates from the slow memory; third, factoring the panel in the fast memory; and finally, saving the fully processed data in the slow memory.

Fig. 5 shows another factorization variant: the top-looking Cholesky factorization, which does delay the updates the most. Instead of factoring the whole panel, the top-looking factorization only factors a diagonal triangle of limited size, the blocking factor n_b, and defers all updates to the rows below the triangle, as well as all updates to the trailing submatrix to the right. In other words, each step of the algorithm is as follows: first, all pending updates (from the top of the matrix) are applied to a stripe of the matrix (to the left of the diagonal triangle); then, that stripe is used to update the diagonal triangle; finally, the diagonal triangle is factored.

Also, all operations involved in the Cholesky factorization can be tiled, i.e., expressed as a set of operations on blocks of size $n_b \times n_b$ as shown in Fig. 6. This is due to the fact that the Cholesky factorization is self-similar: it admits a fully recursive formulation without extra floating-point operations. Any efficient BLAS implementation applies hierarchical tiling to facilitate data reuse at multiple levels of the memory hierarchy (registers, caches, etc.). Tiled formulation is yet another loop transformation—a combination of tiling and reordering. The order of tile accesses determines whether the implementation is left-, right-, or top-looking.

The system of dense linear equations may be solved by applying the so-called forward and backward substitutions. These use the factors obtained through the factorization

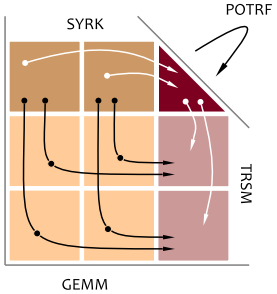

Fig. 6. ***Tile Cholesky factorization (left-looking).***

Algorithm 3 Forward substitution using C-style zero-based indexing.

```
1 for k = 0 to n − 1 do
2 |   b_k ← b_k / L_{k,k}
3 |   for n = k + 1 to n − 1 do
4 |   |   b_n ← b_n − b_k / L_{kn}
```

process, whereby the input matrix is factored into the product of a lower triangular matrix and its transpose. Algorithm 3 shows the forward-substitution procedure and Algorithm 4 shows the backward-substitution procedure. Triangular solves, as these algorithms are known, may easily be implemented by the TRSV routines—a part of the set of L2 BLAS. Crucially, unlike the cubic-complexity factorization, the solves require only $O(n^2)$ operations while using $O(n^2)$ data, which results in constant O(1) reuse ratio. As a result, both triangular solves are completely memory bound and are consequently unlikely to deliver distinct benefits, regardless of using either multiple algorithmic alternatives such as those that originate in loop reordering [34] or those that deal with imperfect loop nests [32], [33].

V. IMPLEMENTATION OF CHOLESKY FACTORIZATION

The two main problems that need to be addressed in order to achieve efficient batched factorization for dense matrices are, first, a near-optimal use of the memory bandwidth and, second, high utilization of the hardware's SIMT scheduler (the lack of thread divergence). As described in Section IV, the left-looking algorithm is ideal for solving the bandwidth issue while the right-looking algorithm is perfect for solving the other—a clear instance of divergent optimization goals. As a solution, maximizing memory bandwidth is addressed first by choosing the nonresident Cholesky algorithm with GPU dynamic random-access memory (DRAM) acting as the slow storage and the shared memory and register file represent the fast storage. Algorithm 5 describes such an implementation that can be considered the nonresident Cholesky factorization. This algorithm requires only $O(n \times n_b)$ worth of matrix elements stored in the fast memory to factor the entire $n \times n$ matrix while still keeping the schedule busy enough.

For the sake of convenience, the term "global memory" will be used here as a stand-in for the GPU RAM, while

Algorithm 4 Backward substitution using zero-based indexing.

```
1 for k = N − 1 to 0 do
2 |   b_k ← b_k / L_{k,k}
3 |   for n = k − 1 to 0 do
4 |   |   b_n ← b_n − b_k / L_{n,k}
```

Algorithm 5 Cholesky factorization as a non-resident implementation in the context of GPU's SIMT.

```
1  for ℓ = 0 to ⌈n/n_b⌉ − 1 do
2  |   Read panel ℓ: global memory → local memory
3  |   __syncthreads()
4  |   for 𝒦 = 0 to ℓ − 1 do
5  |   |   Read panel 𝒦: global memory → shared memory
6  |   |   __syncthreads()
7  |   |   Apply update from panel 𝒦 to panel ℓ
8  |   |   __syncthreads()
9  |   Factor panel ℓ using only local and shared memory
10 |   __syncthreads()
11 |   Save panel ℓ: shared memory → global memory
12 |   __syncthreads()
```

"local memory" will be, for the most part, synonymous with the register file. The outermost loop of the factorization encompasses the following actions in each step: first, a panel of width n_b columns is loaded to the local memory (line 2). Second, the loop iterates over all n_b-size stripes to the left of the panel and performs the following: it loads each stripe, one at a time, into the shared memory (line 5) and then applies the pending update to the current panel (line 7). That panel is then factored, using both the register file and the shared memory as storage (line 10). Writeback of the data to the global memory finishes this second step. One notable feature of this implementation pertains to the limited storage requirement: only $n \times n_b$ entries of the shared memory and $n \times n_b$ registers per thread block are needed. Memory coherence is provided in Algorithm 5 by repeated calls to CUDA's `__syncthreads()` that synchronizes the threads with respect to one another and thus guarantees a consistent memory view.

Our tool of choice for explicit transformation of the source code is `pyexpander`, a macroprocessor for text documents written in Python and with Python-like syntax. This offloads our preprocessing tasks to a generally available code base with ongoing maintenance and support. Also, our specification language for search space iterators and constraints is also based on Python-exclusive syntax and search execution, which lowers the user's burden of entry into our autotuning infrastructure.

The factorization is implemented in C/C++ and CUDA, but without any use of low-level constructs: no intrinsics or embedded PTX. Additionally, the optimized code includes `#pragma unroll` statements for all relevant loops—except for the outermost one, in line 1 of Algorithm 5, which is explicitly unrolled using the `pyexpander` preprocessor. The main reason for this loop to be explicitly unrolled is the fact that the value of the loop counter affects the boundaries of the inner loops, and the CUDA compiler does not unroll those loops that have nonconstant

boundaries. This explicit inlining does not yield a substantial code size increase because the number of the outermost loop's iterations is small.

From a performance engineering standpoint, the compiler is expected to completely unroll all loops and place the majority of local memory variables in the register file. In particular, the local array that holds the panel should remain in registers throughout the entire factorization. Whether this optimization was applied as a transformation and performed by the compiler can be determined with an inspection of the compiler-produced assembly. This file is not created by default while compiling the source code, but the `cubin` format file can be easily produced and then disassembled. Compilation to `cubin` format is done by the compiler with the `-cubin` flag supplied on the command line when using the `nvcc` compiler. The disassembly, on the other hand, is performed with the NVIDIA toolchain by passing the resulting `cubin` file to the `nvdisasm` tool.

Clearly, our approach transcends a single platform and a single implementation choice. Instead, it moves toward portable performance without resorting to the aforementioned low-level techniques, opting instead for autotuning—a systematic process of generating generating high performance implementations for a variety of problem sizes when using only CUDA source code. In what follows, we document this technique for Cholesky factorization across data storage formats and NVIDIA's hardware platforms in the form of the company's successive GPU generations.

VI. IMPLEMENTATION OF FORWARD- AND BACKWARD-SUBSTITUTION STEPS

Unfortunately, a triangular solve with only a single right-hand side offers very little compute-bound parallelism and is mostly memory bound. The vector b is updated by one column of matrix A at any given time. There is only work for one GPU warp for matrices of size 32 and smaller. Seemingly, there is work for more than one warp in the initial steps when dealing with larger matrices, but this quickly drops off as the loop finishes its initial iterations. Keeping this in mind, using just a single thread warp for each solve presents no serious downsides, and boasts two distinct advantages. First, if only a single thread warp is used, then an explicit synchronization through a call to `__syncthreads()` may be omitted because the threads are implicitly synchronized at the warp level. Second, direct register-to-register communication may be used to send the results of the division in line 2 of Algorithms 3 and 4 to all the threads in the warp, rather than relying on shared memory copies. GPU devices with compute capability of at least 3.0 perform this operation with the `__shfl()` instruction known as a warp shuffle.

Matrix data alignment in memory is a not an issue in the forward substitution because the warp may be aligned with the columns of the matrix. For backward substitution, this poses a new challenge that needs to be addressed. Commonly, in software libraries for dense linear algebra such as LAPACK, the Cholesky factorization modifies only one half of the input matrix, either lower or upper triangular portion, while leaving the other portion untouched. We chose to present the lower triangular implementation here. Accordingly, both the forward substitution and backward substitution use the same matrix elements. The latter accesses the data with the matrix viewed in a transposed form. Because of this, any GPU implementation faces a performance issue: the transposition causes the threads to access the memory in a detrimental pattern with a stride that is equal to the size of the matrix. A common solution to this problem is to use shared memory for reading the matrix in limited-size stripes, and then perform the transposed access in shared memory, where such an access pattern incurs much less severe penalty. This solution encounters the shared resource problem, however, because the total usage of shared memory limits the maximum occupancy that the code may reach. We propose a solution below that could be considered a novel contribution.

Prior to being saved to the device's main memory, each panel is factored using the combined storage of both the register file and the shared memory. In order for the transposed access to be efficient as well, the triangular matrix is written to the device memory twice: in nontransposed and transposed form. The latter one is placed in the unused, upper part of the input matrix. Now, the factorization is present in shared memory, and by reorganizing the GPU threads, aligned writes are made possible in both the nontransposed and transposed cases. Clearly, a slight penalty on the factorization performance is imposed by the additional transposed matrix writeback. However, the cost is amortized because it enables a fast implementation of the transposed triangular solve. Ultimately, this cost is basically a trivial result of the nontransposed triangular solve.

Last, the remaining problem that needs addressing is the potential low occupancy value caused by the implementation with only a single warp. It is known that high occupancy is important for bandwidth-bound kernels. Creating only one warp per thread block yields low occupancy because the maximum number of thread blocks per multiprocessor is much lower than the maximum number of warps per multiprocessor and the thread scheduler has little possibility for parallel execution. However, the solution to this problem turns out to be simple: each thread block is given multiple warps, because repeated triangular solves are available in batch mode and each warp ends up being responsible for a different triangular solve. And as an added bonus, each warp is independent of all others. Hardware-dependent optimization involves choosing the optimal number of warps per block.

Thus, the only parameter to be tuned for the triangular-solve kernels is the number of warps per block. The objective is simple: maximize the achieved memory bandwidth. It turns out that the easiest way to accomplish this is to maximize value of hardware occupancy. For example,

with respect to the Kepler architecture, the maximum number of blocks per multiprocessor is limited to 16, but the maximum number of warps per multiprocessor is 64. Consequently, launching a single warp per block reaches only 16/64 = 25% occupancy. Increasing the number of warps per block enables the implementation to increase the resulting performance, with the limit being four warps per block, at which point the maximum occupancy (4× 16/64 = 100%) is reached; any further increases of the warp counts may only keep the performance at the same level and would not improve it. Alternatively, the same result can be achieved by creating two blocks with 32 warps per thread block, which would also enable the hardware scheduler to run the maximum number of warps per block.

VII. AUTOTUNING OF THE CHOLESKY FACTORIZATION

We maximize the achieved performance by identifying the fastest among a large number of generated implementations—a classic heuristic for automatic software tuning commonly referred to as autotuning methodology. For every size of the matrix to be factored (n), a number of implementation choices need to be made, as different sizes are possible for the width of the panel—the blocking factor n_b in addition to the two dimensions of the CUDA thread block. To facilitate the automation, the actual kernel source code is generalized so that any value for n_b can be used for a given n, and any shape and size of the GPU thread block can be used for a given n_b; i.e., there are no constraints on the shapes and sizes other than the correctness, which is checked against the reference code. In our case, we use the original LAPACK code running on the host CPU. It would be possible to check against cuBLAS in case of LU and QR factorizations, but since cuBLAS currently does not provide batched Cholesky factorization, we default to the canonical CPU code at the moment—arguably a more time-consuming option. For performance comparisons, we use the MAGMA numerical library [13]. The batch size of 10 000 is used for all runs reported in this paper to serve as a reference point that has a good standing in practical cases. At such large batch sizes, the kernel's performance is very close to its asymptotic value, and increasing the batch size any further would not noticeably increase the gigaFLOP/s rate.

Recall that the primary objective of batched factorization routines is to achieve good performance for a significantly large number of small-sized factorizations/solves. The tuning we perform within this context is done for all matrices in the range of 5×5 and 100×100, inclusively, i.e., $5 \leq n \leq 100$. Then, for each matrix size n, the panel widths n_b (also called blocking factors) are taken to be the following: $n, \lceil n/2 \rceil, \lceil n/3 \rceil, \lceil n/4 \rceil, \ldots, 1$. In particular, for a matrix of size $n = 33$, the set of values for n_b is: 33, 17, 11, 9, 7, 6, 5, 4, 3, 2, and 1. Furthermore, for each shape of the panel ($n \times n_b$), the height of the GPU thread block varies across all powers of two smaller than n ($2^k,\ k < n$) and the first value larger than n, and the width of the GPU thread block ranges through the powers of two smaller than n_b with the first value larger than n_b. For example, for a panel of size 33×11, the values for the height of the GPU thread block (`blockDim.x`) are: 1, 2, 4, 8, 16, 32, and 64; and the values for the width of the GPU thread block (`blockDim.y`) are: 1, 2, 4, 8, and 16. To reduce the search space with respect to practical considerations, two additional filters are applied. The first one ensures that the GPU thread blocks with the total number of threads not divisible by the warp size (32) are not considered. The second one makes sure that the thread blocks with the total number of GPU threads that exceed the hardware-defined maximum number are skipped. For example, for the Kepler architecture this maximum is 1024. As a result, our runs do not represent a fully exhaustive sweep. Ideally, more panels could be tested for almost all matrix sizes. Also, even more thread block shapes and sizes could have been tried for each panel. However, the currently implemented tool chain for code generation, code compilation, and binary timing is still a sequential process. In practice, it occupies a single GPU device for long periods of time. In the future, we plan to include in the tool chain more parallelism in shared and possibly distributed-memory environments for massively parallel evaluation, which would result in a more exhaustive sweep with less constraints. Clearly, some of the constraints we impose at the generation time may still be imposed during code compilation or kernel launch, or both.

VIII. AUTOTUNING OF THE FORWARD- AND BACKWARD-SUBSTITUTION SOLVES

As discussed above, there is one main parameter of the triangular solve in their respective implementation kernels: the number of warps per GPU thread block. The objective of choosing the right value for that parameter is to maximize the memory bandwidth achieved during the execution. The most straightforward way to accomplish this goal is to maximize the hardware occupancy value. For instance, on the NVIDIA Kepler architecture, the maximum number of thread blocks per multiprocessor is 16, while the maximum number of warps per multiprocessor is 64 (see Table 1 for more details). Thus, an implementation that launches a single thread warp per thread block reaches only 16/64 = 25% occupancy. With an increased number of thread warps per thread block, greater performance may be attained, but only up to four warps per block—the saturation point with the maximum hardware occupancy (100%) is reached. Any further increases in the number of warps would only keep the execution performance at the same level. As an alternative, one could create two thread blocks with as many as 32 thread warps in each, which would again maximize the number of warps per block as far as the NVIDIA Kepler hardware is concerned.

Table 1 Side-By-Side Comparison of Three Recent Generations of NVIDIA GPU Cards

Tesla Products:	Tesla K40	Tesla M40	Tesla P100
Name	Kepler	Maxwell	Pascal
GPU	GK110	GM200	GP100
SMs	15	24	56
FP32 cores / SM	192	128	64
FP32 cores / GPU	2880	3072	3584
FP64 cores / SM	64	4	32
FP64 cores / GPU	960	96	1792
Base clock MHz	745	948	1328
Boost clock MHz	810/875	1114	1480
FP64 Gflop/s*	1680	213	5304
Texture units	240	192	224
Memory type	GDDR5	GDDR5	HBM2
Memory i-face bits	384	384	4096
Memory size	12	24	16
Level 2 cache KiB	1536	3072	4096
Register file / SM KiB	256	256	256
Register file / GPU KiB	3840	6144	14336
TDP Watts	235	250	300
Transistors billion	7.1	8	15.3
Die Size mm^2	551	601	610
Manufacturing process nm	28	28	16

* based on boost clock

IX. ALTERNATIVE DATA LAYOUTS FOR BATCHED OPERATION

Commonly, matrices for batch operations are laid out in the main memory one after another, and each one occupies a contiguous piece of memory, usually in column-major data layout. This layout makes it more challenging to have coalesced memory reads, especially as the dimensions of the matrices become smaller. Eventually, it is nearly impossible to have any remaining coalesced reads for matrices smaller than 32 in single-precision arithmetic.

The most straightforward way to solve this issue is to reorder the dimensions. In addition to matrix width and height, there is an additional dimension for batch processing mode: the matrix index. This extra dimension can be used as the fastest growing of the three, as illustrated in Fig. 7. In this case, one thread warp reads 32 elements, with the same row and column index in 32 consecutive matrices. Observe that as long as the whole data set is 128-B aligned, and the number of matrices is divisible by 32, the data will always be read with optimal coalescing occurring regardless of the other two dimensions. An admittedly uncommon layout, we observe that it has its precedence not in dense linear algebra [35], [36], but

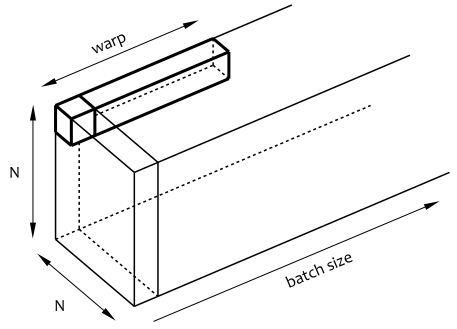

Fig. 7. ***Simple interleaved batch layout.***

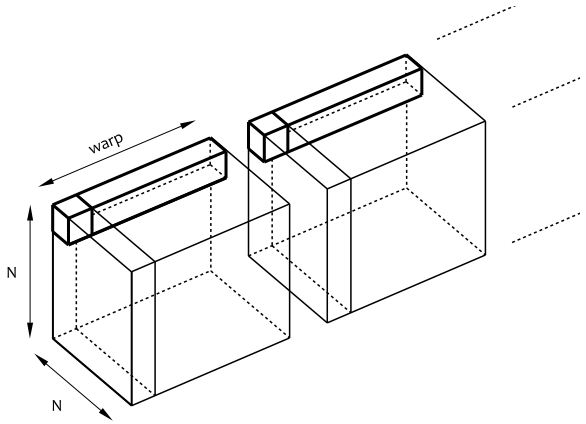

Fig. 8. ***Batch data layout with interleaved chunks.***

in the NVIDIA's CUDA Deep Neural Network (cuDNN) library for deep learning. The convolution filters in cuDNN matrices and tensors can be stored in either the so-called NCHW and the NHWC layouts [37] (W is width, H is height, N is the image number in the batch, and C is the filter index).

This new layout has the benefit of coalesced reads, which is complicated by two new problems: one obvious and one more subtle in nature. Clearly, divisibility by 32 is an obvious problem and there is a need for a solution for any number of matrices in the batch. Fortunately, it can easily be solved by padding the data set to the 32-divisible size (round up to 32). This is trivially accomplished in practice, and we are not going to look into it any further by assuming that the padding is done for all the presented results. The more subtle issue is the fact that the elements of a single matrix would become positioned far apart in the memory. The consequences of this are far less obvious, and we show issues in greater detail below.

One simple solution to this problem is grouping the matrices in a batch into chunks of 32—maybe even larger multiples of 32 as illustrated in Fig. 8. Each chunk is stored immediately after the previous chunk, and each chunk gets to occupy a contiguous region of the main memory. In this solution, all reads are coalesced, as they were before, and the elements of each matrix reside much closer in memory. We look at the performance consequences of using this layout versus the original one. We will also study the impact of using varying chunk sizes for the chunked layout.

A. Implementation of Code for Interleaved Layout

All of the code is implemented in C and uses CUDA for GPU code. It is compiled for a specific size of the matrices in the batch. The code is completely or almost completely unrolled using the `pyexpander` preprocessor. The factorization is assembled from a set of four basic operations that deal with $n_b \times n_b$ tiles as shown in Fig. 9. `spotrf_tile` applies the Cholesky factorization to a single tile, while `strsm_tile`, `ssyrk_tile`, and `sgemm_tile` perform their corresponding L3 BLAS operations (triangular solve,

```
#define spotrf_tile(rA)\\
$for(k in range(0, NB))\
 $("rA##_%d%d = sqrtf(rA##_%d%d);\\\n" % (k,k,k,k))\
 $("inv = 1.0f/rA##_%d%d;\\\n" % (k,k))\
 $for(m in range(k+1, NB))\
  $("rA##_%d%d *= inv;\\\n" % (m,k))\
 $endfor\
 $for(n in range(k+1, NB))\
  $for(m in range(n, NB))\
   $("rA##_%d%d -= rA##_%d%d*rA##_%d%d;\\\n" % (m,n,n,k,m,k))\
  $endfor\
 $endfor\
$endfor

#define strsm_tile(rA1, rA2)\\
$for(m in range(0, NB))\
 $for(k in range(0, NB))\
  $("rA2##_%d%d /= rA1##_%d%d;\\\n" % (m,k,k,k))\
  $for(n in range(k+1, NB))\
          $("rA2##_%d%d -= (rA2##_%d%d*rA1##_%d%d);\\\n"  %
(m,n,m,k,n,k))\
  $endfor\
 $endfor\
$endfor

#define ssyrk_tile(rA1, rA2)\\
$for(m in range(0, NB))\
 $for(n in range(0, m+1))\
  $for(k in range(0, NB))\
           $("rA2##_%d%d -= rA1##_%d%d*rA1##_%d%d;\\\n"  %
(m,n,m,k,n,k))\
  $endfor\
 $endfor\
$endfor

#define sgemm_tile(rA1, rA2, rA3)\\
$for(m in range(0, NB))\
 $for(n in range(0, NB))\
  $for(k in range(0, NB))\
           $("rA3##_%d%d -= rA1##_%d%d*rA2##_%d%d;\\\n"  %
(m,n,m,k,n,k))\
  $endfor\
 $endfor\
$endfor
```

Fig. 9. *Generalized microkernels for implementing the operations on individual tiles.*

```
#define load_full(_m, _n, rA)\\
dAp = dA + _m*NB*32 + _n*NB*N*32;\\
$for(n in range(0, NB))\
 $for(m in range(0, NB))\
  $("rA##_%d%d = *dAp;\\\n" % (m,n))\
  dAp += 32;\\
 $endfor\
 dAp += (N-NB)*32;\\
$endfor

#define store_full(_m, _n, rA)\\
dAp = dA + _m*NB*32 + _n*NB*N*32;\\
$for(n in range(0, NB))\
 $for(m in range(0, NB))\
  $("*dAp = rA##_%d%d;\\\n" % (m,n))\
  dAp += 32;\\
 $endfor\
 dAp += (N-NB)*32;\\
$endfor

#define load_lower(_m, _n, rA)\\
dAp = dA + _m*NB*32 + _n*NB*N*32;\\
$for(n in range(0, NB))\
 $for(m in range(n, NB))\
  $("rA##_%d%d = *dAp;\\\n" % (m,n))\
  dAp += 32;\\
 $endfor\
 $("dAp += (N-NB+%d)*32;\\\n" % (n+1))\
$endfor

#define store_lower(_m, _n, rA)\\
dAp = dA + _m*NB*32 + _n*NB*N*32;\\
$for(n in range(0, NB))\
 $for(m in range(n, NB))\
  $("*dAp = rA##_%d%d;\\\n" % (m,n))\
  dAp += 32;\\
 $endfor\
 $("dAp += (N-NB+%d)*32;\\\n" % (n+1))\
$endfor
```

Fig. 10. *Generalized microkernels that implement loading and storing of individual tiles.*

symmetric rank-k update, and matrix–matrix multiplication). Each one is fully unrolled using `pyexpander`. Note that this is the right-looking variant and the others may be developed in a similar fashion with some reuse of the auxiliary components, described below in more detail.

The above operations work on tiles stored in local variables that are numbered according to the elements' locations in the tile. The assumption is that, once loaded into the register file, the tile can be fully utilized, i.e., modified with updates, after which it may be disposed of. This calls for a set of specialized operations for loading and storing the tiles into the slow memory. Fig. 10 shows these memory operations `load_full` and `store_full` for reading and writing full (square) tiles, and `load_lower` and `store_lower` for reading and writing diagonal (lower triangular) tiles. In this paper, we only support lower triangular matrices, but the upper triangular matrices can trivially be supported in a similar fashion. Both load and store operations are completely unrolled as well.

Finally, Fig. 11 shows how the aforementioned tile operations can be combined into a complete implementation of the Cholesky factorization. Note in Fig. 11 that the outer loops are not unrolled, but they also can still be completely unrolled, and this is shown in Fig. 12. The entire factorization can be completely unrolled into a single block of basic code, which may seem at first like an extreme performance measure, but it still makes sense for the smallest matrices, and we look into that case further in the investigation below.

```
for (int kk = 0; kk < N/NB; kk++) {

   for (int nn = 0; nn < kk; nn++) {

      load_full(kk, nn, rA3);

      for (int mm = 0; mm < nn; mm++) {

         load_full(kk, mm, rA1);
         load_full(nn, mm, rA2);
         sgemm_tile(rA1, rA2, rA3);
      }
      load_lower(nn, nn, rA1);
      strsm_tile(rA1, rA3);
      store_full(kk, nn, rA3);
   }
   load_lower(kk, kk, rA1);

   for (int nn = 0; nn < kk; nn++) {

      load_full(kk, nn, rA2);
      ssyrk_tile(rA2, rA1);
   }
   spotrf_tile(rA1);
   store_lower(kk, kk, rA1);
}
```

Fig. 11. *Implementation using generalized microkernels for top-looking Cholesky factorization.*

```
$for(kk in range(0, N/NB))\

   $for(nn in range(0, kk))\

      load_full($(kk), $(nn), rA3);

      $for(mm in range(0, nn))\
         load_full($(kk), $(mm), rA1);
         load_full($(nn), $(mm), rA2);
         sgemm_tile(rA1, rA2, rA3);
      $endfor\

      load_lower($(nn), $(nn), rA1);
      strsm_tile(rA1, rA3);
      store_full($(kk), $(nn), rA3);
   $endfor\

   load_lower($(kk), $(kk), rA1);

   $for(nn in range(0, kk))\
      load_full($(kk), $(nn), rA2);
      ssyrk_tile(rA2, rA1);
   $endfor\

   spotrf_tile(rA1);
   store_lower($(kk), $(kk), rA1);
$endfor\
```

Fig. 12. ***Completely unrolled microkernels that implement the top-looking variant of the Cholesky factorization.***

Once we ensure that the matrix dimension is divisible by n_b, then the codes from Figs. 11 and 12 may readily be used. However, we also handle the cases where the dimension is not divisible by tile size n_b. For that, we use another set of optimized kernels for handling these corner cases. For brevity, however, we do not show the sources here as it is a straightforward extension of the code that we did show. The kernels follow the same principle of fully unrolling each operation (load, store, and compute).

X. AUTOTUNING OF THE ALTERNATIVE DATA LAYOUT CODE

The resulting code has five tunable parameters. They are all made into compile time parameters, except for the chunk size, which is a runtime parameter. The following parameters are taken into account when compiling a single instance of the GPU kernel.

A. Tile Size Parameter

This parameter, referred to as n_b, defines the size of tiles used in the factorization. The Cholesky factorization executes in three alternating steps of loading tiles, computing on tiles, and storing tiles. Matrix tiling defines the size of code for each operation in Figs. 9 and 10.

B. Looking Parameter

This tuning parameter decides the order in which the tile operations are evaluated in the Cholesky factorization and provides the choice of the right-looking (aggressive evaluation) factorization, the left-looking (lazy) factorization, and the top-looking (the most delayed) evaluation.

C. Chunking Parameter

This tuning parameter defines the data layout feature and enables a switch from the simple layout without chunking (Fig. 7) to the more complex layout with chunking (Fig. 8).

D. Chunk Size Parameter

This tuning parameter controls the size of chunks. It is not clear to us at this point if using the warp size is always optimal, and, therefore, we investigated larger multiples of 32 here (64, 128, 256, 512).

E. Unrolling Parameter

This tuning parameter decides if the outer compute loops are also unrolled in addition to the inner loops of tile operations that are unrolled by default. If the outer loops are not unrolled, then the code looks simply like the code in Fig. 11. Conversely, if the outer loops are unrolled, the code looks like the code in Fig. 12. The inner loops of tile operations (Figs. 9 and 10) are always unrolled.

XI. HARDWARE AND SOFTWARE SETUP

We used two GPU systems in our tests: the NVIDIA Kepler K40c card (15 SM multiprocessors, each containing 192 CUDA cores) and the NVIDIA Pascal P100 (56 SM multiprocessors, each containing 64 CUDA cores).

The theoretical peak floating-point performance of the Kepler card in single precision is 4.7 teraFLOP/s and 10.6 teraFLOP/s for the Pascal card. The Kepler GPU features 11.25 GB of error-correcting code (ECC) protected DRAM with a theoretical bandwidth in excess of 288 GB/s. Each multiprocessor has 64 KB of shared memory/L1 fast cache with a theoretical peak bandwidth of 216 GB/s. The Pascal card, on the other hand, features 16 GiB of HBM2 with theoretical bandwidth of about 0.5 TB/s. Other important hardware characteristics for the Kepler card include the following: maximum number of active threads per each multiprocessor (2048), maximum number of thread blocks per each multiprocessor (16), maximum number of threads per thread block (1024), maximum number of registers per thread (256), total size of the shared memory is fixed at the software level and is configurable to 16, 32, or 48 KB at the expense of L1 cache. CUDA toolkit version 7.0 [38] comprised the software stack and was used for compiling all Kepler card codes and producing MAGMA performance numbers. The Pascal results use version 8 of the CUDA toolkit. Based on our prior experience, such a mixed-software setup guarantees no performance regressions.

The Kepler system used an Intel Sandy Bridge Xeon E5-2670 running at 2.6 GHz as the host CPU, in a two-socket configuration featuring 8 cores (octa-core) in each socket, and the theoretical peak of all cores was 666 gigaFLOP/s in single precision. Each core has a 32 KiB Ll 1 data cache, 32 KiB L1 instruction cache, and shared

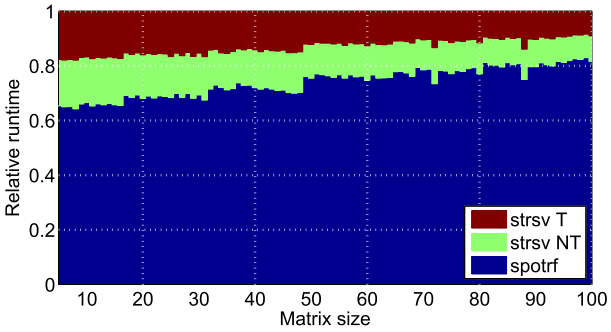

Fig. 13. *Batched Cholesky factorization and the relative contribution of forward substitution and backward substitution to the overall runtime while solving a set of 10 000 symmetric-positive-definite matrices on the Kepler card.*

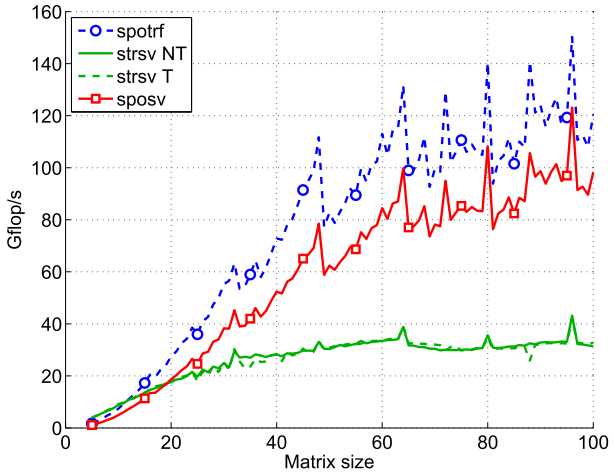

Fig. 14. *Performance of the IEEE compliant batched Cholesky factorization, the batched triangular solves, and the solution process combining the factorization with the forward and backward substitutions on the Kepler card. The target problem is a batch of 10 000 SPD matrices.*

256 KiB L2 cache. Per socket the L3 cache was 20480 KiB and was shareable socket-wide. The main memory was 64 GiB (globally accessible) with a theoretical bandwidth of 51 GiB/s. The CPU software implementation of tested codes was based on the vendor routines from Intel MKL version 11.1.2 [39].

A more comprehensive comparison of the tested hardware and NVIDIA Maxwell is given in Table 1.

XII. PERFORMANCE OF SOLVING A SEQUENCE OF LINEAR SPD PROBLEMS FOR DENSE MATRICES

A sequence of SPD linear systems and their complete solution process is done by calling the following as three separate kernels: the batched Cholesky factorization, the batched forward substitution, and the batched backward substitution. In Fig. 13, we show the size-dependent contributions to overall execution time coming from the distinct kernels when solving a set of 10 000 randomly generated SPD matrices. One may observe that the execution time of the batched Cholesky factorization increases faster with the problem size than the execution time for the batched solves: the solves taking from about 35% for smaller sizes down to about 15% for larger sizes. To an extent, this is expected as the factorization is cubic in complexity $O(n^3)$, while the solves are square in complexity $O(n^2)$.

Fig. 14 presents the performance results for the IEEE-compliant batched Cholesky factorization that writes the triangular factors to the main memory twice (as opposed to classical implementation that writes it once), the batched triangular matrix solves, and the resulting performance of the overall time of the solution process, combining the factorization execution with the execution of forward and backward substitutions. The performance results for the batched solves show much less variance when compared against the batched factorization. In fact, in an asymptotic sense they approach 30 Gflop/s, with some isolated performance peaks exceeding 40 Gflop/s. For the overall performance when the algorithm combines the three kernels, with each handling the most of floating-point operations in the local multiprocessor memory, up to 100 Gflop/s is achieved for larger linear system sizes, with one isolated peak exceeding 120 Gflop/s for the matrix size 96×96.

Next, we proceed with the results of our experiments on two generations of GPU compute cards: Kepler and Pascal. The following figures will focus on studying the effects of changing one specific parameter—e.g., tiling factor—while varying all other parameters to maximize the resulting performance rate. Consequently, every point on a single line in the following charts corresponds to one fixed value of a given parameter with all the other parameters being potentially different. Also, between two different lines on a single chart, at least one parameter is different. Due to a wide range of variability of all the parameters between different points and lines, we do not include exact settings in the figures.

Fig. 15 shows the best performance of the interleaved implementation for varying tiling factors. For sizes smaller

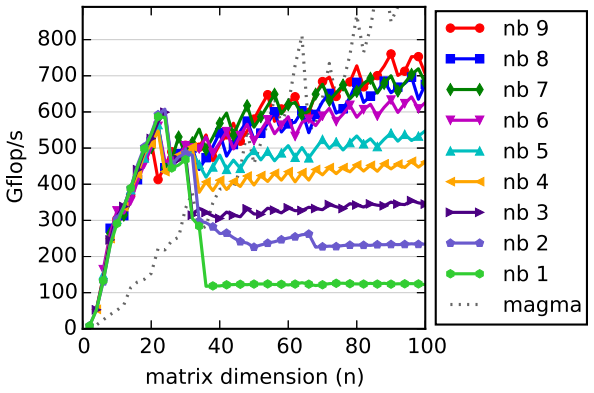

Fig. 15. *The best performance results of the code implementing the interleaved variant for different tiling factors on the Kepler card.*

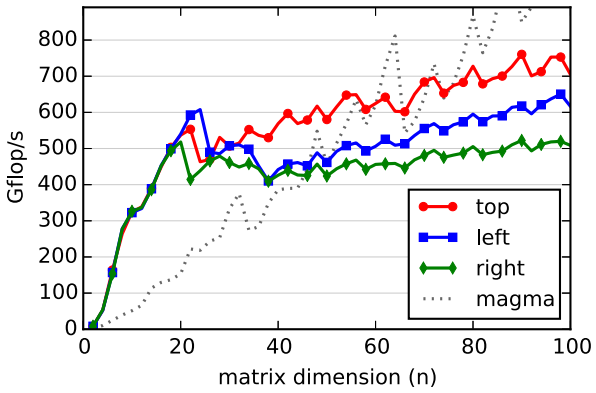

Fig. 16. ***The best performance of the interleaved implementation for different orders of evaluation of the outer loops on the Kepler card.***

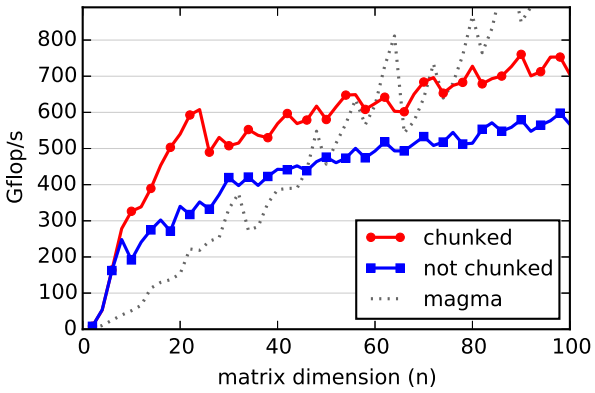

Fig. 17. ***The best performance results of the code implementing interleaved data layout either with and without chunking on the the Kepler card.***

than 20, the tiling implementation makes no difference for performance because the system is able, for the most part, to preserve data in registers throughout the factorization. This behavior shows deteriorated results between sizes 20 and 40. After that, past 40, lack of blocking ($n_b = 1$) produces no data reuse and then the code becomes mostly memory bound. By introducing blocking optimization ($n_b > 1$), performance gradually increases, until it eventually levels off around size 8.

Fig. 16 presents the best performance of the interleaved data layout, implemented for changing orders of code evaluation—reordering of the outer loops. Up to the size of about 20, there is no observable difference in the performance results. In that range, this is mostly due to the fastest codes likely being fully unrolled, and the resulting order of evaluation in the source code well optimized by the NVIDIA compiler at the stage of instruction scheduling. Past the size of about 20, the full unrolling feature stops exerting beneficial influence, and the tile operations are executed according to the order given in the source code. At that point, the implementation with the least memory traffic would win. While we observe no difference in the total number of memory reads, the more lazy the order of evaluation is, the less data writes there are. Hence, the right looking implementation turns out to be the slowest, the left-looking becomes faster, and the top-looking one ends up being the fastest.

Fig. 17 illustrates the best performance of the interleaved data layout implementation with and without chunking. It may be observed that chunking is very beneficial to the resulting performance. While we cannot explain fully why this is in fact the case, intuitively, it is the outcome one would expect. The principle of maximizing spatial locality takes a leading role at some levels of the memory hierarchy.

Fig. 18 features the chart with the best performance of the interleaved data layout implementation with chunking, with varying chunk size. It is interesting to observe how this parameter has the property of defining the number of threads in a thread block. It seems that 32 is the best choice. Obviously, for this kind of workload and on this particular hardware platform, it is perfectly acceptable to have thread blocks with only a single warp. The configuration with 64 performs almost equally well, but then the execution performance drops slightly for 128 and 256, and much more significantly diminishes for 512.

Finally, Fig. 19 shows the best performance results of the interleaved data layout implementation with partial unrolling (for tile operations only) and full loop unrolling (the whole factorization). Full unrolling pays off, but only up to the size of 20, and then the benefits progressively diminish, and the partial unrolling takes over in terms of performance. As an explanation, we may offer a comment that either the number of instructions overwhelms the optimizing compiler from NVIDIA, or the instruction-fetching process combined with data caching become a primary problem, or perhaps both.

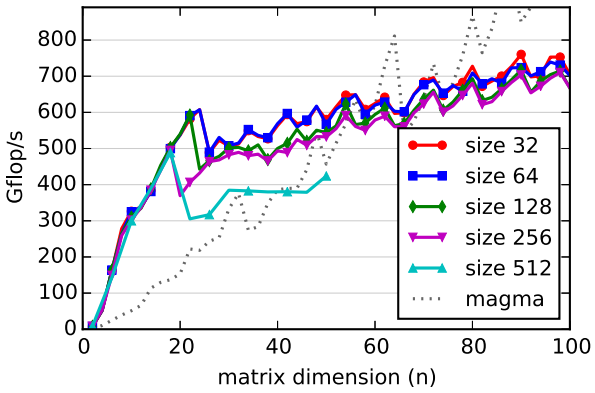

Fig. 18. ***The best performance results of the code implementing interleaved data layout with chunking, for varying chunk sizes on the Kepler card.***

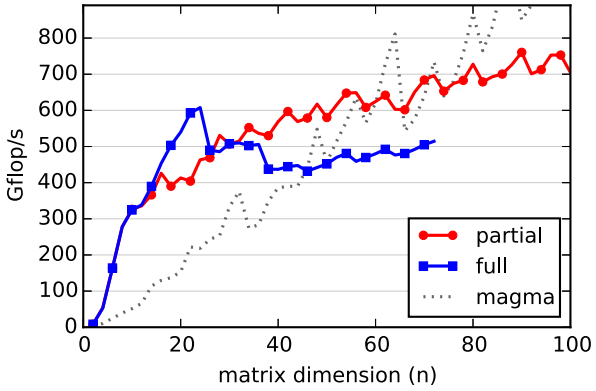

Fig. 19. *The best performance of the implementation for interleaved data lout with partial unrolling (tile operations only) and full unrolling (the whole factorization) on the Kepler card.*

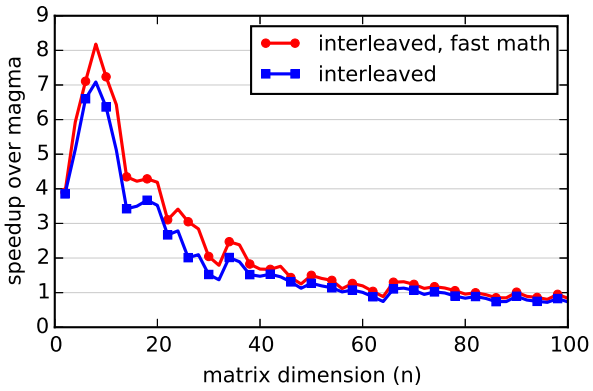

Fig. 21. *Speedup of the interleaved implementation over the traditional implementation in MAGMA on the Pascal card.*

Fig. 20 shows the overall performance for a batch of size 16 384 using the NVIDIA P100 (Pascal) card with CUDA 8.0. When computing the flop/s value, the standard formula $(1/3)n^3 + O(n^2)$ is always used for the number of floating-point operations. The figure shows performance when using IEEE-compliant arithmetic, and when using the `-use_fast_math` option, which relaxes the IEEE compliance for the square root and division operations, and flushes denormalized numbers to zero. For smaller matrices, the code achieves 600 Gflop/s for the IEEE-compliant case, and approaches 800 Gflop/s for the `-use_fast_math` case, and substantially outperforms the traditional implementation in MAGMA 2.2.0. Fig. 21 shows the speedup over MAGMA.

XIII. GENERAL COMMENTS AND DISCUSSION OF RESULTS

Overall, we can draw some very clear conclusions substantiated by our results, and none of these seem to us all that unexpected.

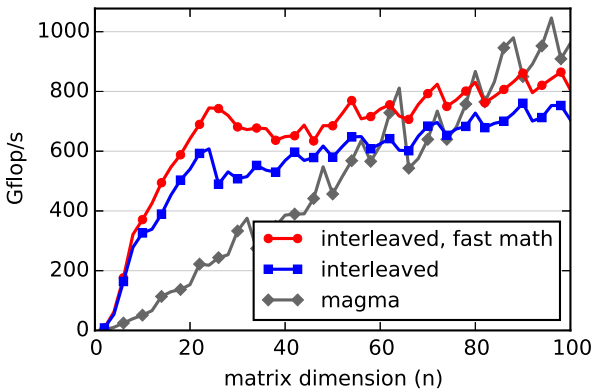

Fig. 20. *Top performance of the interleaved implementation, with IEEE compliant arithmetic and with the `-use_fast_math` option on the Pascal card.*

- For very small sizes of matrices in the batch mode, the interleaved data code outperforms all traditional implementations because it allows the GPU load/store units to transfer data with perfect coalescing, regardless of the matrix dimension.
- Tiling data and code is a critical optimization for dense matrix operations, unless of course the dimension is so small that the matrix can fit entirely inside the register file.
- The most lazy evaluation order is the most beneficial from the standpoint of optimal memory traffic, as it minimizes the number of write operations.
- Chunking of matrix data is beneficial to overall performance because the hardware is able to fully exploit the spatial data locality principle.
- The newest compute GPU generations from NVIDIA perform very well with a large number of small thread blocks.
- Complete (or almost complete) unrolling of the source code works well—but only up to a certain point. At larger matrix sizes, the benefits visibly diminish.

Also, it is important to point out here that the execution performance of the interleaved data implementation does tend to level off, and it is eventually surpassed by the performance of the traditional implementation in the MAGMA software library, especially for larger sizes. This is due to the interleaved data code's reliance on the data reuse in the register file only. But data reuse only happens within the context of a single thread. While there are no data to be shared by distinct threads, there is no reuse in the shared memory space. In this context, the cache memories only serve as streaming data buffers.

XIV. CONCLUSION AND FUTURE RESEARCH

This paper presented a batched Cholesky factorization for GPUs that handle most of the factorization and solve tasks

in fast multiprocessor memory. By using our autotuning framework, called BONSAI [30], we arrived at optimal (within the confines of the initial template and parallelization strategy) kernel configurations as they were identified automatically. The achieved performance was significantly higher than its CPU counterpart and similar routines taken from the MAGMA software library. In this context, a set of specialized routines were produced for efficient handling of the forward and backward substitutions (the triangular solves). By comparing our proposed implementations to the equivalent routines from the MAGMA library, we show significant performance advantages. The resulting performance of the complete solution execution—where the factorization and the forward and backward substitutions are available in three distinct kernels—exceeds 120 gigaFLOP/s on state-of-the-art GPU cards, when matrix sizes do not exceed 100×100. Our future research will correlate the observed performance of all the kernel configurations to the available GPU metrics, such as hardware occupancy, achieved memory bandwidth, and the rate of instruction execution. Also, the QR and LU factorization will be treated in the same framework and methodology. Finally, a related research effort will dive into the overall hardware resource efficiency by taking the energy balance into account in addition to performance alone. ■

Acknowledgments

The authors would like to thank the Oak Ridge National Laboratory for access to the Titan supercomputer, where the infrastructure for the BONSAI 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 (IPDPSW)*, Apr. 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.*, 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 (ICDM)*, Dec. 2008, pp. 263–272.

[5] H. A. Bruck, S. R. McNeill, M. A. Sutton, and W. H. Sutton, III, "Digital image correlation using Newton-Raphson method of partial differential correction," *Exp. Mech.*, vol. 29, no. 3, pp. 261–267, 1989.

[6] M. Gates, M. T. Heath, Jr., and J. Lambros, "High-performance hybrid CPU and GPU parallel algorithm for digital volume correlation," *Int. J. High Perform. Comput. Appl.*, vol. 29, no. 1, pp. 92–106, 2015.

[7] X. Ma and G. R. Arce, *Computational Lithography*, vol. 77. Hoboken, NJ, USA: Wiley, 2011.

[8] M. G. Moharam and T. K. Gaylord, "Rigorous coupled-wave analysis of metallic surface-relief gratings," *J. Opt. Soc. Amer. A, Opt. Image Sci.*, 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. (IPDPS)*, May 2011, pp. 372–383.

[10] X. Lacoste, P. Ramet, M. Faverge, Y. Ichitaro, and J. Dongarra, "Sparse direct solvers with accelerators over DAG runtimes," INRIA, Paris, France, Tech. Rep. RR-7972, 2012.

[11] *cuBLAS Library User Guide*, document DU-06702-001_v8.0, NVIDIA Corporation, Sep. 2016.

[12] *Intel Math Kernel Library Developer Reference, Revision: 011*, Intel Corp., Santa Clara, CA, USA, 2017.

[13] E. Agullo *et al.*, "Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects," *J. Phys., Conf. Ser.*, vol. 180, no. 1, p. 012037, 2009.

[14] O. Villa, M. Fatica, N. Gawande, and A. Tumeo, "Power/performance trade-offs of small batched LU based solvers on GPUs," in *Proc. Eur. Conf. Parallel Process.* Springer, 2013, pp. 813–825.

[15] 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. Cybersp. Saf. Secur., IEEE 11th Int. Conf Embedded Softw. Syst (HPCC, CSS, ICESS), IEEE Int. Conf. High Perform. Comput. Commun.*, Aug. 2014, pp. 157–160.

[16] T. Dong, A. Haidar, S. Tomov, and J. Dongarra, "A fast batched Cholesky factorization on a GPU," in *Proc. 43rd Int. Conf. Parallel Process. (ICPP)*, Sep. 2014, pp. 432–440.

[17] 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. Princ. Pract. Parallel Program.*, 2015, pp. 261–262.

[18] 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. 1094342014567546, 2015.

[19] J. Kurzak, H. Anzt, M. Gates, and J. Dongarra, "Implementation and tuning of batched Cholesky factorization and solve for NVIDIA GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 7, pp. 2036–2048, Jul. 2016.

[20] M. Gates, H. Anzt, J. Kurzak, and J. Dongarra, "Accelerating collaborative filtering using concepts from high performance computing," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Nov. 2015, pp. 667–676.

[21] R. C. Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimizations of software and the ATLAS project," *Parellel Comput. Syst. Appl.*, vol. 27, nos. 1–2, pp. 3–35, 2001, doi: 10.1016/S0167-8191(00)00087-9.

[22] J. Bilmes, K. Asanović, J. W. Demmel, D. Lam, and C.-W. Chin, "Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology," Dept. Comput. Sci., Univ. Tennessee, Knoxville, TN, USA, Tech. Rep. UT-CS-96-326, Aug. 1996. [Online]. Available: http://www.netlib.org/lapack/lawnspdf/lawn111.pdf

[23] R. Vuduc, J. W. Demmel, and K. A. Yelick, "OSKI: A library of automatically tuned sparse matrix kernels," *J. Phys., Conf. Ser.*, vol. 16, pp. 521–530, 2005, doi: 10.1088/1742-6596/16/1/071.

[24] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proc. IEEE*, vol. 93, no. 2, pp. 216–231, Feb. 2005.

[25] M. Püschel *et al.*, "SPIRAL: Code generation for DSP transforms," *Proc. IEEE*, vol. 93, no. 2, pp. 232–275, Feb. 2005.

[26] R. Veras and F. Franchetti, "Capturing the expert: Generating fast matrix-multiply kernels with spiral," in *High Performance Computing for Computational Science—VECPAR*, M. Daydé, O. Marques, and K. Nakajima, Eds. 2014, pp. 236–244.

[27] C. A. Schaefer, V. Pankratius, and W. F. Tichy, "Atune-IL: An instrumentation language for auto-tuning parallel applications," in *Proc. 15th Int. Euro-Par Conf. Parallel Process.*, vol. 5704. Heidelberg, Germany: Springer, Aug. 2009, pp. 9–20. [Online]. Available: http://Tichy/uploads/publikationen/216/original.pdf

[28] S. Kamil, "Productive high performance parallel programming with auto-tuned domain-specific embedded languages," Ph.D. dissertation, Dept. Elect. Eng. Comput. Sci., Univ. California, Berkeley, Berkeley, CA, USA, 2012.

[29] S. Benkner, F. Franchetti, H. M. Gerndt, and J. K. Hollingsworth, "Automatic application tuning for HPC architectures," *Dagstuhl Rep.*, vol. 3, no. 9, pp. 214–244, Jan. 2014. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2014/4423, doi: 10.4230/DagRep.3.9.214.

[30] P. Luszczek, M. Gates, J. Kurzak, A. Danalis, and J. Dongarra, "Search space generation and pruning system for autotuners," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, Chicago, IL, USA, May 2016, pp. 1545–1554.

[31] M. Gates, J. Kurzak, P. Luszczek, Y. Pei, and J. Dongarra, "Autotuning batch Cholesky factorization in CUDA with interleaved layout of matrices," in *Proc. IPDPSW*, Orlando, FL, USA, May/Jun. 2017, pp. 1408–1417.

[32] V. Menon and K. Pingali, "Look left, look right, look left again: An application of fractal symbolic analysis to linear algebra code restructuring," *Int. J. Parallel Program.*, vol. 32, no. 6, pp. 501–523, 2004.

[33] N. Mateev, V. Menon, and K. Pingali, "Fractal symbolic analysis," in *Proc. 15th Int. Conf. Supercomput.*. Sorrento, Italy: ACM Press, 2001, pp. 38–49.

[34] Q. Yi, K. Kennedy, H. You, K. Seymour, and J. Dongarra, "Automatic blocking of QR and LU factorizations for locality," in *Proc. 2nd ACM SIGPLAN Workshop Memory Syst. Perform. (MSP)*, 2004, pp. 12–22.

[35] B. Akin, F. Franchetti, and J. C. Hoe, "FFTs with near-optimal memory access through block data layouts: Algorithm, architecture and design automation," *J. Signal Process. Syst.*, vol. 85, no. 1, pp. 67–82, Oct. 2016, doi: 10.1007/s11265-015-1018-0.

[36] N. Park, B. Hong, and V. K. Prasanna, "Tiling, block data layout, and memory hierarchy performance," *IEEE Trans. Parallel Distrib. Syst.*, vol. 14, no. 7, pp. 640–654, Jul. 2003, doi: 10.1109/TPDS.2003.1214317.

[37] Y. M. Tsai, P. Luszczek, J. Kurzak, and J. Dongarra, "Performance-portable autotuning of OpenCL kernels for convolutional layers of deep neural networks," in *Proc. 2nd Workshop Mach. Learn. HPC Environ. (MLHPC)*. Salt Lake City, UT, USA: IEEE Press, Nov. 2016, pp. 9–18, doi: 10.1109/MLHPC.2016.5.

[38] *NVIDIA CUDA TOOLKIT 7.0*, NVIDIA Corp., Santa Clara, CA, USA, Mar. 2015.

[39] *Intel Math Kernel Library for Linux OS*, document 314774-005US, Intel Corporation, Oct. 2007.

ABOUT THE AUTHORS

Jack Dongarra (Fellow, IEEE) holds an appointment at the University of Tennessee, Knoxville, TN, USA; Oak Ridge National Laboratory, Oak Ridge, TN, USA; and the University of Manchester, Manchester, U.K. He specializes in numerical algorithms in linear algebra, parallel computing, use of advanced-computer architectures, programming methodology, and tools for parallel computers.

Dr. Dongarra was awarded the IEEE Sid Fernbach Award in 2004. In 2008 he was the recipient of the first IEEE Medal of Excellence in Scalable Computing. In 2010 he was the first recipient of the SIAM Special Interest Group on Supercomputing's award for Career Achievement. In 2011 he was the recipient of the IEEE IPDPS Charles Babbage Award. In 2013 he received the ACM/ IEEE Ken Kennedy Award. He is a Fellow of the American Association for the Advancement of Science (AAAS), the Association for Computing Machinery (ACM), and the Society for Industrial and Applied Mathematics (SIAM) and a member of the National Academy of Engineering.

Mark Gates received the B.S., M.S., and Ph.D. degrees in computer science from the University of Illinois at Urbana-Champaign, Urbana, IL, USA, in 1998, 2007, and 2011, respectively.

Currently, he is a Research Scientist in the Innovative Computing Laboratory at the University of Tennessee, Knoxville, TN, USA, where he researches algorithms for linear algebra on multicore and GPU-based computers. His research interests are in high-performance computing, particularly linear algebra and GPU computing.

Jakub Kurzak received the M.Sc. degree in electrical and computer engineering from Wroclaw University of Technology, Wroclaw, Poland, and the Ph.D. degree in computer science from the University of Houston, Houston, TX, USA.

He is a Research Director in the Innovative Computing Laboratory, Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville, TN, USA. His research interests are in high-performance computing with multicore and accelerators.

Piotr Luszczek started his research in sparse direct and iterative methods for linear systems of equations with optimized computational kernels. Subsequently, he worked on out-of-core linear solvers, self-adaptation, and autotuning software. He codesigned parallel programming languages in industrial and governmental lab positions. He published conferences and journal articles as well as book chapters and patents. The major themes of his research work is performance modeling and evaluation in the context of tuning of parallelizing compilers as well as energy-conscious aspects of heterogeneous and embedded computing. Throughout his professional career, he has been the principal developer of established industry benchmarks: HPL, HPCC, and HPCG.

Yaohung (Mike) Tsai is a graduate student at the Electrical Engineering and Computer Science Department, University of Tennessee, Knoxville, TN, USA. His dissertation work encompasses automated performance engineering of computationally intensive codes for dense numerical linear algebra and deep learning.