2017 IEEE International Parallel and Distributed Processing Symposium Workshops

# Autotuning Batch Cholesky Factorization in CUDA with Interleaved Layout of Matrices

Mark Gates
Jakub Kurzak
Piotr Luszczek
Yu Pei
*Innovative Computing Laboratory, University of Tennessee*
Jack Dongarra
*University of Tennessee, Oak Ridge National Laboratory, University of Manchester*

***Abstract*—Batch matrix operations address the case of solving the same linear algebra problem for a very large number of very small matrices. In this paper, we focus on implementing the batch Cholesky factorization in CUDA, in single precision arithmetic, for NVIDIA GPUs. Specifically, we look into the benefits of using noncanonical data layouts, where consecutive memory locations store elements with the same row and column index in a set of consecutive matrices. We discuss a number of different implementation options and tuning parameters. We demonstrate superior performance to traditional implementations for the case of very small matrices.**

***Keywords*-batch computation; GPU computing; numerical linear algebra; Cholesky factorization; data layout**

## I. Introduction

### A. Motivation

While linear algebra software has achieved high efficiency for solving large linear systems on GPU-based computers, 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 $100 \times 100$ elements, fails to fully utilize today's highly parallel computing hardware. However, when a large set of small linear systems is presented simultaneously, using a batch implementation exposes significant parallelism, allowing for better use of parallel hardware. Numerous applications deal with large sets of small linear solves that call for batch processing on GPUs: finite element methods, computational lithography, and collaborative filtering, to name a few.

### B. Related Work

Currently, there is a lot of interest in batch matrix operations. Both NVIDIA cuBLAS [1] and Intel MKL [2] include extensive sets of batch routines for basic linear algebra tasks, and so does the MAGMA library from University of Tennesseec [3]. Numerous papers have been published about the development and optimization of batch routines [4–8]. This paper is a follow up to our previous work on the batch Cholesky factorization for small matrices [9]. The direct motivation for this work came from the *Alternating Least Squares* (ALS) algorithm for recommender systems [10]. In our previous article, we looked into the development of batch routines for the canonical column-wise data layout. Here we are investigating alternative layouts for batches of extremely small matrices.

Our autotuning methodology is based on the autotuning approach that we pioneered with ATLAS [11] and that now grew into a vibrant field of experimental performance optimization guided by execution profiles. To name a few efforts, we could start with Portable High Performance ANSI C (PHiPAC) [12] that generated code for superscalar processors implementing dense linear algebra operations. Sparse matrix computations were targetted by Optimized Sparse Kernel Interface (OSKI) [13] and FFT and similar transforms were optimized by the Fastest Fourier Transform in the West (FFTW) [14] and Spiral [15]. In fact, Spiral recently addressed matrix-matrix multiply [16]. To our best knowledge, these projects do not address autotuning for accelerators, and they mostly embed the expert knowledge of the tuning inside the code rather than expose it in the form of stencils as we do. Also, DSLs exist for the sole purpose of autotuning parallel scientific codes [17, 18]. A much more complete survey of recent advances in autotuning is available elsewhere [19].

### C. Original Contribution

The main contribution of this work is in dramatically improving the performance of a batch matrix operation for batches of extremely small matrices, by introducing unorthodox data layouts. We believe that there is also a lot of value in the autotuning section of this article, where we discuss a number of implementation options and tuning parameters, and show their clear impact on performance.

### D. GPU Architecture

In this section, we are offering a minimal overview of the GPU architectural features that are most important from the standpoint of implementing the kernel. The two most prominent features are the *Single Instruction Multiple Threads* (SIMT) processing model and the memory hierarchy.

978-1-5386-3408-0/17 $31.00 © 2017 IEEE
DOI 10.1109/IPDPSW.2017.18

Figure 1 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 independent instruction stream. Instead, a set of 32 cores, have to follow the same execution path. Therefore, the basic unit of scheduling is a *warp*, which is a set of 32 threads.

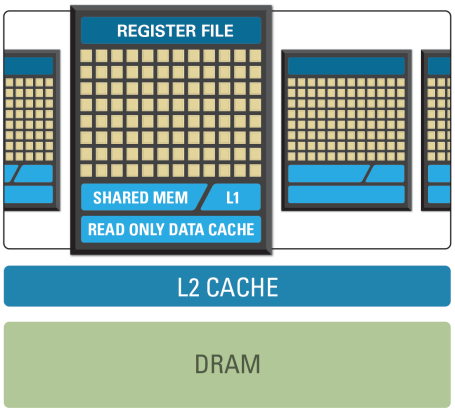


Figure 1. Architecture of NVIDIA GPUs. The tan colored squares represent CUDA cores.

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. The task becomes uniquely difficult in the case of batched operations, where the sizes of the input matrices are close to the size of the warp.

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 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 the DRAM. Reads from the DRAM pass through the L2 cache, and the L1 cache or the read-only data cache. DRAM bandwidth is a precious commodity for batched matrix operations, which are very close to being memory bound. A critical issue in accessing DRAM is *coalescing*, i.e., reads are most efficient if all threads in a warp read the same 128-byte cache line. If data is not cached, the cache line will be fetched from DRAM in a single memory transaction. If different threads in a warp access different cache lines, multiple memory transactions will occur, consuming extra memory bandwidth.

## II. Solution

### A. 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 $L^T$. In this article we focus solely on the factorization step.

Algorithm 1 and Figure 2 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 an inferior, memory bound, implementation of an otherwise compute bound algorithm.

**Algorithm 1** Canonical Cholesky factorization (unblocked, right-looking, lower-triangular) 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
```

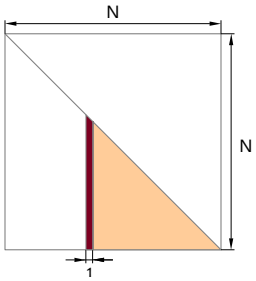


Figure 2. Canonical Cholesky factorization (no blocking, right-looking, lower-triangular).

The main optimization of the Cholesky algorithm is *blocking* (Figure 3). 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 operations). Blocking leverages the *surface-to-volume* property of dense linear algebra routines, 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 $n_b$ is factored at a time, where $1 \ll n_b \ll n$, followed by an update of rank $n_b$. Described as a loop transformation, blocking means tiling of the outermost loop in line 1 of Algorithm 1.

In the figures to follow, BLAS and LAPACK routine names are used to refer to the building blocks of the different implementations of the Cholesky factorization:

- **POTRF:** Cholesky factorization (of a small block),
- **TRSM:** triangular solve,
- **SYRK:** symmetric rank-k update,
- **GEMM:** general matrix matrix multiply.

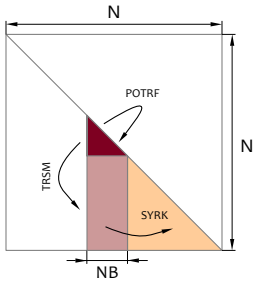


Figure 3. Right-looking blocked Cholesky factorization.

Another important aspect of the implementation is the choice between aggressive and lazy evaluation. The three common versions of the Cholesky factorization are the *right-looking* factorization, the *left-looking* factorization, and the *top-looking* factorization. The right-looking implementation corresponds to aggressive evaluation, where, as soon as the panel is factored, the entire trailing submatrix is updated. Figure 3 shows 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.

Figure 4 shows the left-looking 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.

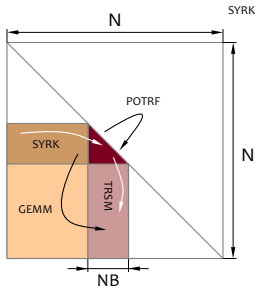


Figure 4. Left-looking blocked Cholesky factorization (the implementation in LAPACK).

Figure 5 shows the top-looking Cholesky factorization, which is the "laziest" one. Instead of factoring the whole panel, the top-looking factorization only factors a diagonal triangle of size $n_b$, and defers updates to the rows below the triangle, as well as updates to the trailing submatrix. In each step of the algorithm, 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, and, finally, the diagonal triangle is factored.

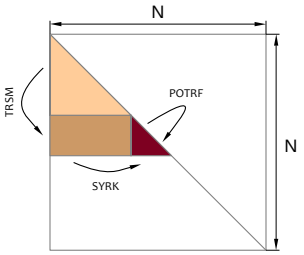


Figure 5. Top-looking block Cholesky factorization.

Also, all operations involved in the Cholesky factorization can be tiled, i.e., expressed as a set of operations on blocks of size $n_b \times n_b$ (Figure 6). Any efficient BLAS implementations applies hierarchical tiling, to facilitate data reuse at multiple levels of the memory hierarchy (registers, caches, etc.)

### B. Data Layout

Typically, for batch operations, matrices are laid out in memory one after another, where each one occupies a contiguous piece of memory, usually in column major layout. This makes it progressively more challenging to have coalesced memory reads, as the dimensions of the matrices become smaller, eventually making it impossible to have any coalesced reads for matrices smaller than 32 in single precision.

The easiest way to solve this problem is to reorder the dimensions, and use the matrix index as the fastest growing index (Figure 7). In this case, one warp reads 32 elements,

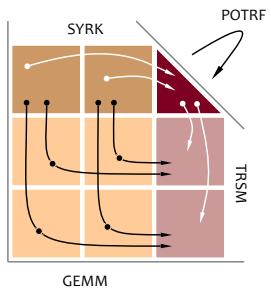


Figure 6. Tile Cholesky factorization (left-looking).

with the same row and column index in 32 consecutive matrices. As long as the whole dataset is 128-byte aligned, and the number of matrices is divisible by 32, data will always be read with perfect coalescing, regardless of the dimensions of the matrices. Although this kind of layout is fairly unorthodox, from the dense linear algebra standpoint, there is a precedence for it in the cuDNN library for deep learning, where the convolution filters can be stored in the NCHW and the NHWC layouts.

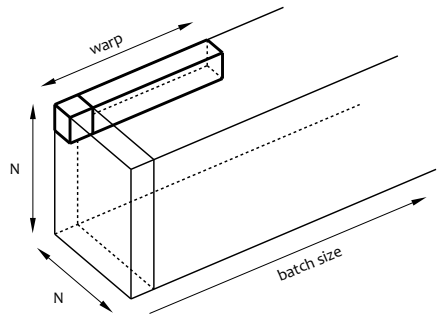


Figure 7. Simple interleaved batch layout.

There are two problems with this layout, one more obvious and one more subtle. The obvious problem is when the number of matrices is not divisible by 32. It can easily be solved by padding the dataset to the divisible size. This is trivial and we are not going to look into it any further. The other, more subtle problem, is the fact that, in this arrangement, elements of a single matrix are spread far apart in the memory. The effects of this are less obvious, and we are going to take a closer look into it.

A simple solution to this problem is grouping of matrices in chunks of 32, or larger multiples of 32 (Figure 8). Chunks are stored one after another and each one occupies a contiguous region of memory. In this case, all reads are coalesced and elements of each matrix reside close by in memory. We are going to look at the performance impact of using one layout versus the other, and the impact of using different chunk sizes for the chunked layout.

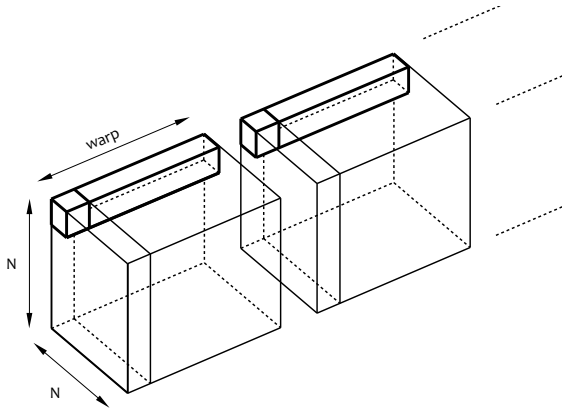


Figure 8. Interleaved chunked batch layout.

### C. Implementation

The code is implemented in C using CUDA, compiled for a specific size of the matrices in the batch, and heavily, or even completely, unrolled using the *pyexpander* preprocessor. The factorization is built from a set of four fundamental operations on $n_b \times n_b$ tiles (Figure 9). `spotrf_tile` applies a Cholesky factorization to a tile, while `strsm_tile`, `ssyrk_tile`, and `sgemm_tile`, perform their corresponding BLAS operations (triangular solve, symmetric rank-k update, and matrix multiplication). Each one is fully unrolled using pyexpander.

These operations work on tiles stored in local variables, numbered according to the elements' locations in the tile. The assumption is that, once loaded to registers, the tile can be fully utilized, and then disposed of. This requires a set of operations for loading and storing the tiles. Figure 10 shows the 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. Here, we only support lower triangular matrices. Upper triangular matrices can be supported in the same manner. The load and store operations are also completely unrolled.

Finally, Figure 11 shows how the tile operations, mentioned so far, can be combined into a complete Cholesky factorization. In Figure 11 the outer loops are not unrolled, but they can also be completely unrolled, which is shown in Figure 12. Complete unrolling of the entire factorization, to a single block of straight line code, may seem like an extreme measure, but may still make sense for very small matrices, so we are going to investigate this further.

If the matrix dimension is divisible by $n_b$, then the codes from Figure 11 and Figure 12 are used. However, we are also handling the cases where the dimension is not divisible,

```
#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
```

Figure 9. Microkernels for implementing tile operations.

```
#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
```

Figure 10. Microkernels for loading and storing tiles.

by using another set of kernels for handling the corner cases. For brevity, we are not showing the sources here. The kernels follow the same principle of fully unrolling each operation (load, store, compute).

### D. Autotuning

The code has five tunable parameters. They are all compile time parameters except chunk size, which is a run time parameter. The following parameters are taken into account when compiling one instance of the kernel:

*1) Tile Size:* This parameter, referred to as $n_b$, defines the size of tiles used in the factorization. The factorization executes in alternating steps of loading tiles, computing on tiles, and storing tiles. Tiling defines the size of code for each operation in Figures 9 and 10.

*2) Looking:* This parameter decides the order of evaluation of the tile operations in the Cholesky factorization and provides the choice of the right-looking (aggressive) factorization, the left-looking (lazy) factorization, and the top-looking (laziest) evaluation.

*3) Chunking:* This parameter defines the data layout and allows to switch from the simple layout without chunking (Figure 7) to the, more complex, layout with chunking (Figure 8).

*4) Chunk Size:* This parameters controls the chunk size. It is not clear if using the warp size is always optimal, and, therefore, we investigate larger multiples of 32 here (64, 128, 256, 512).

*5) Unrolling:* This parameter decides if the outer loops are also unrolled, in addition to the inner loops of tile operations. If the outer loops are not unrolled, the code looks like the code in Figure 11. If the outer loops are unrolled, the code looks like the code in Figure 12. The inner loops of tile operations (Figures 9 and 10) are always unrolled.

## III. Results and Discussion

Figure 13 shows the overall performance for a batch of size 16,384 using the NVIDIA P100 (Pascal) card, using CUDA 8.0. When computing the Gflop/s value, the standard formula, $\frac{1}{3}N^3$, 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

```
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);
}
```

Figure 11. Top-looking Cholesky factorization implemented using microkernels.

```
$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\
```

Figure 12. Top-looking Cholesky factorization implemented using microkernels (completely unrolled).

flushes denormalized number to zero. For smaller matrices, the code achieves 600 GFLOPS for the IEEE compliant case, and approaches 800 GFLOPS for the `--use_fast_math` case, and substantially outperforms the traditional implementation in MAGMA 2.2.0. Figure 14 shows the speedup over MAGMA.

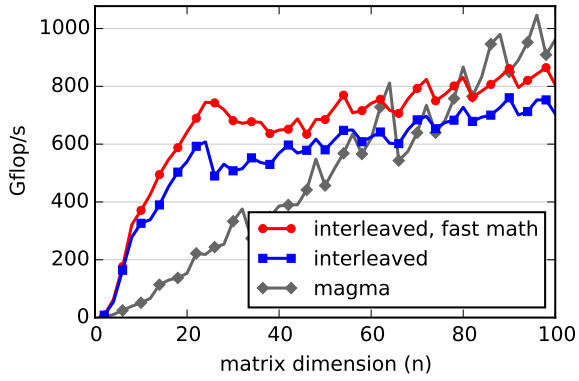


Figure 13. Top performance of the interleaved implementation, with IEEE compliant arithmetic and with the `--use_fast_math` option.

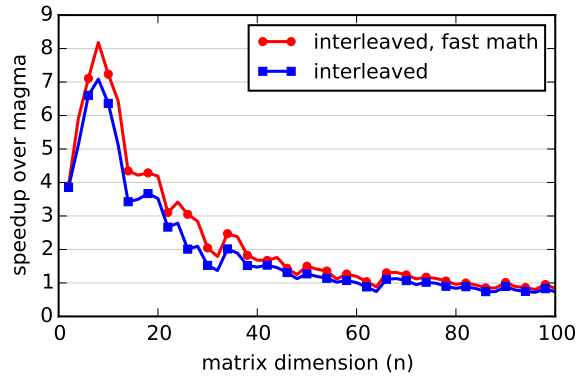


Figure 14. Speedup of the interleaved implementation over the traditional implementation in MAGMA.

Figure 15 shows the best performance of the interleaved implementation for different tiling factors. For sizes smaller than 20, tiling makes no difference, as the system is able to preserve data in registers throughout the factorization. This behavior deteriorates between 20 and 40. Past 40, no blocking ($n_b = 1$) has no data reuse and the code becomes memory bound. Introducing blocking ($n_b > 1$) gradually increases performance, until it levels off around 8.

Figure 16 shows the best performance of the interleaved implementation for different orders of evaluation of the outer loops. Up to the size of 20, there is no difference in performance. This is because the fastest codes, in that range, end up fully unrolled and the order of evaluation in the source code is optimized by the compiler in the stage of instruction scheduling. Past the size of 20, full unrolling stops being beneficial and tile operations are executed according to the order in the source code. At this point, the implementation with the least memory traffic wins. While there is no difference in the number of memory reads, the lazier the order of evaluation, the less writes there are. Therefore, the right looking implementation is the slowest, the left looking is faster, and the top looking is the fastest.

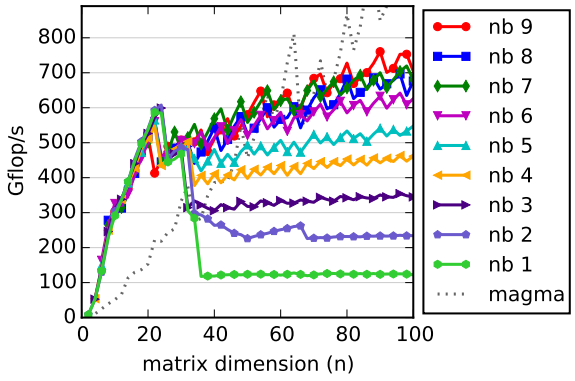


Figure 15. The best performance of the interleaved implementation for different tiling factors.

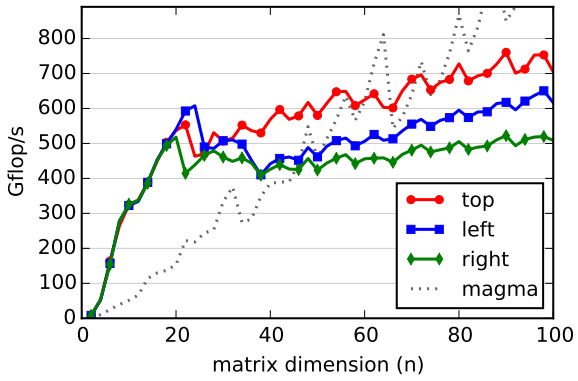


Figure 16. The best performance of the interleaved implementation for different orders of evaluation of the outer loops.

Figure 17 shows the best performance of the interleaved implementation with and without chunking. Clearly, chunking is very beneficial to performance. While we cannot say exactly why this is the case, intuitively, this is the expected outcome. The spatial locality principle takes effect at some level of the memory hierarchy.

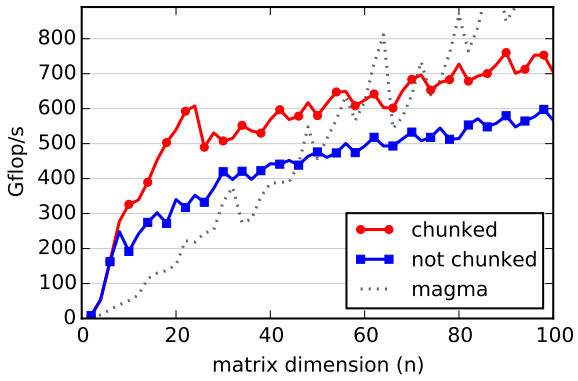


Figure 17. The best performance of the interleaved implementation with and without chunking.

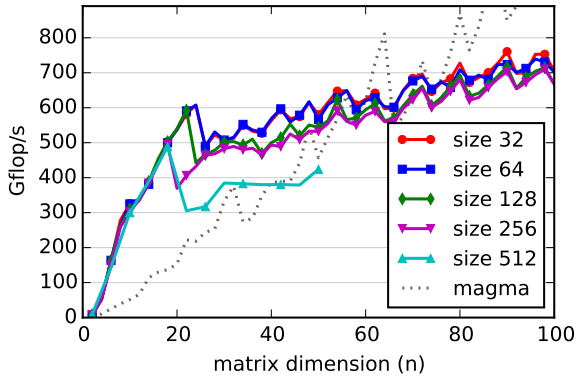


Figure 18. The best performance of the interleaved implementation with chunking, for different chunk sizes.

Figure 18 shows the best performance of the interleaved implementation with chunking, for different chunk sizes. It is important to observe that this parameter also defines the number of threads in a thread block. 32 seems to be the best choice. Clearly, for this kind of workload and this architecture, it is perfectly fine to have thread blocks with a single warp. 64 performs almost equally well, but then the performance drops slightly for 128 and 256, and significantly for 512.

Finally, Figure 19 shows the best performance of the interleaved implementation with partial unrolling (tile operations only) and full unrolling (the whole factorization). Full unrolling pays off up to the size of 20, and then the benefits diminish, and the partial unrolling takes over. Either the number of instructions overwhelm the compiler, or instruction fetching and caching becomes a problem, or both.

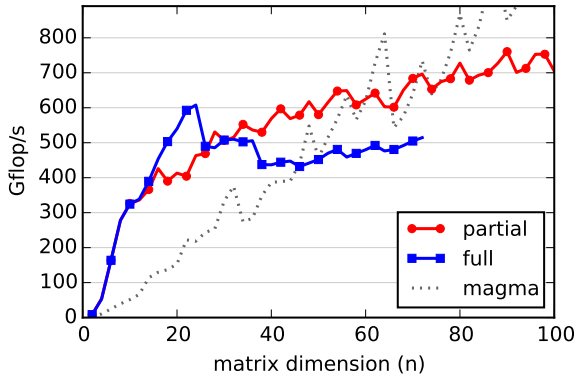


Figure 19. The best performance of the interleaved implementation with partial unrolling (tile operations only) and full unrolling (the whole factorization).

Overall, we can draw some very clear conclusions, none of them really unexpected:

- For very small sizes of matrices in the batch, the interleaved code outperforms traditional implementations,

because it allows for perfect coalescing, regardless of the matrices' dimension.

- Tiling is a critical optimization for dense matrix operations, unless the dimension is so small that the matrix can fit entirely in registers.
- The laziest evaluation is the most beneficial from the standpoint of memory traffic, as it minimizes the number of write operations.
- Chunking is beneficial to performance as the hardware is able to exploit the spatial locality principle.
- The newest GPUs from NVIDIA perform very well with a large number of small thread blocks.
- Aggressive unrolling of the source code works to a point. At extreme sizes, the benefits diminish.

It is important to point out that the performance of the interleaved implementation levels off, and is surpassed by the performance of the traditional implementation in MAGMA, for larger sizes. This is because the interleaved code relies on data reuse in registers only. Data reuse only happens within a single thread. There is no data to be shared by different threads, and, therefore, there is no reuse in shared memory of caches. Caches only serve the purpose of streaming buffers.

Examining all the kernels for particular sizes reveals several interesting features. Figure 20 shows kernels for $n = 24$ and $n = 48$. For clarity, we show kernels only for chunk size 64, which we have already observed is optimal. The kernels are sorted into 9 bins across the $x$-axis by their $n_b$; within each $n_b$ there are up to 12 kernels. For $n = 24$, the chunked, fully unrolled versions (solid red triangles) were best, and in particular the left-looking one (◀) with $n_b = 2$. However, for $n = 48$, the chunked, fully unrolled versions are no longer the best performers, instead being overtaken by the top-looking, partially unrolled versions (empty red △ triangles), in particular with $n_b = 7$. For all sizes, the non-chunked, fully unrolled codes (solid blue triangles) were consistently the worst performing. In general, the chunked version was better than its non-chunked counterpart. For other parameters, however, we cannot make universal statements that large $n_b$ is always best; left, right, or top-looking is always best; or fully unrolling is always best – only certain combinations perform well. Predicting these winning combinations beforehand is difficult.

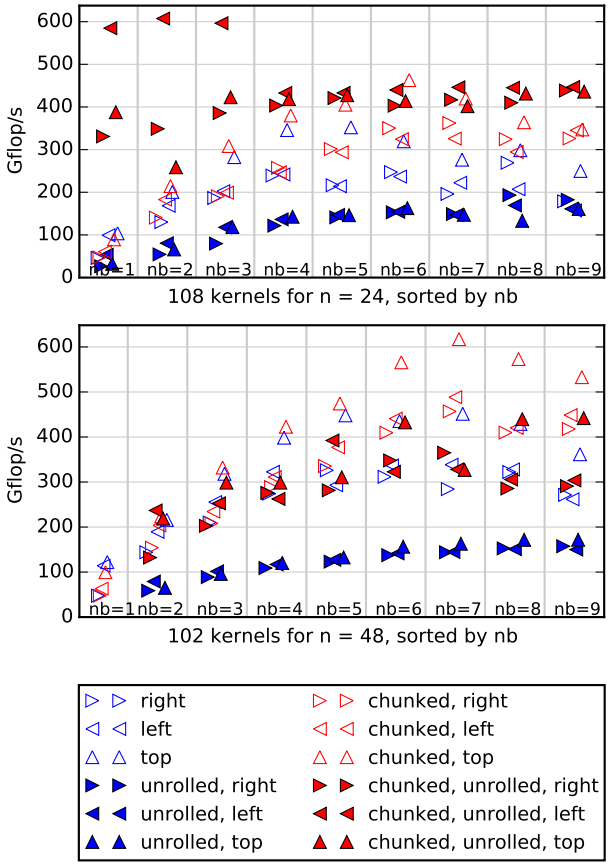


Figure 20. All kernels for $n = 24$ and $n = 48$ with chunk size 64.

## IV. Analysis of the Autotuning Data Set

We performed an exhaustive search of the autotuning space of code parameters. At first, it might seem counter productive due to a large number of runs and existence of workable heuristics to guide the search more efficiently towards a nearly-optimal solution while skipping large portions of suboptimal combinations. But our goal is not the minimal search time but rather meaningful exploration of the parameter configurations and automated extraction of useful metrics and analysis that leads to informative models of the complete performance data. Clearly, using a guided search which skips the some areas of the search space represents a form of *selection bias* committed in the name of minimization of execution time. This trade-off we make allows us to present the following analysis.

The data set comprising the complete autotuning sweep of the parameter space has over 14,000 performance measurements of successful runs. Unlike heuristic-driven autotuning systems, our methodology delivers a data-rich view of the performance landscape and allows a postmortem analysis of the data set. In the example studied in this experiment, we have a mix of parameters that are represented by discrete (e.g., blocking factor) and categorical (e.g., unrolling) variables. Each class of these variables can be addressed independently by various machine learning classifiers, but mixing them together poses some challenges. For starters, encoding of the categories may adversely influence the classification outcome. Also, the variables easily identifiable in the auto-generated code may influence the performance very little due to the latent structure of our tested system, i.e., the compiler's optimizations and details of the accelerator hardware. Both of these are only partially known to the programmer and may only be discovered in our case through statistical inference

because we do not use low-level techniques such as reverse engineering of the GPU's binary code [20]. To identify the aforementioned influence of the parameters, we show in Table I their predictive power of performance. We can see that the tile size $n_b$ and chunking have the strongest effect, while cache has the weakest.

Table I
PREDICTIVE POWER OF VARIOUS TUNING PARAMETERS ON PERFORMANCE IN TERMS OF MEAN SQUARE ERROR.

| Parameter | Inclusive MSE | Type | Explanation |
|---|---|---|---|
| $n$ | 43.1 | integer | size of single matrix |
| Tile size, $n_b$ | 103.9 | integer | internal blocking |
| Looking | 99.9 | ternary | Left, Right, or Top |
| Chunking | 157.4 | binary | yes or no |
| Chunk size | 25.9 | integer | matrix count in chunk |
| Unrolling | 85.7 | binary | use unrolling? |
| L1 or shared cache | −18.6 | binary | more L1 or shared mem. |

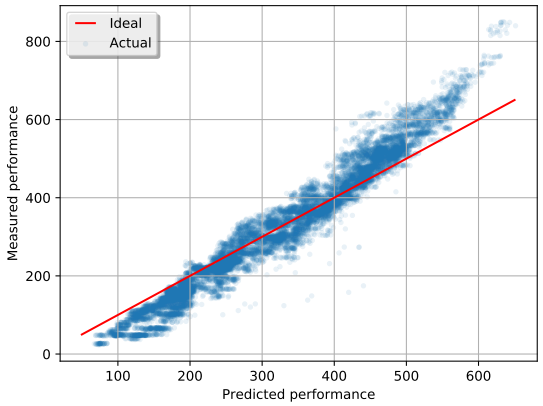


Figure 21. Accuracy of the Random Forest model shown as a correlation between predicted and observed performance.

We then proceed to model the data with a random forest [21, pages 592-593], and its implementation in R [22], to arrive at the basic characterization of the autotuning data set. The original random forest algorithm [23] may be used for either classification or regression. Additionally, the algorithm is suitable as the first step of classification because it can compute proximities between the data points. We used it in the regression mode.

The generated model has 500 trees of average depth 11. The constructed model allows us to plot a density point cloud that indicates the quality of the predictive power with respect to the measured performance. That plot is shown in Figure 21, with the red line indicating the ideal correlation between measurement and prediction from the model.

## V. ACKNOWLEDGMENTS

This work is supported by grant #1642441: "SI2-SSE: BONSAI: An Open Software Infrastructure for Parallel Autotuning of Computational Kernels" from the National Science Foundation.

## REFERENCES

[1] *cuBLAS Library User Guide*, DU-06702-001_v8.0 ed., NVIDIA Corporation, September 2016.

[2] *Intel Math Kernel Library Developer Reference*, Revision: 011 ed., Intel Corporation, 2017.

[3] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, "Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects," in *Journal of Physics: Conference Series*, vol. 180, no. 1. IOP Publishing, 2009, p. 012037.

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

[5] 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 *High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC, CSS, ICESS), 2014 IEEE Intl Conf on*. IEEE, 2014, pp. 157–160.

[6] T. Dong, A. Haidar, S. Tomov, and J. Dongarra, "A fast batched Cholesky factorization on a GPU," in *Parallel Processing (ICPP), 2014 43rd International Conference on*. IEEE, 2014, pp. 432–440.

[7] A. Haidar, T. Dong, P. Luszczek, S. Tomov, and J. Dongarra, "Towards batched linear solvers on accelerated hardware platforms," in *ACM SIGPLAN Notices*, vol. 50, no. 8. ACM, 2015, pp. 261–262.

[8] ——, "Batched matrix computations on hardware accelerators based on GPUs," *The International Journal of High Performance Computing Applications*, vol. 29, no. 2, pp. 193–208, 2015.

[9] J. Kurzak, H. Anzt, M. Gates, and J. Dongarra, "Implementation and tuning of batched Cholesky factorization and solve for NVIDIA GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 7, pp. 2036–2048, 2016.

[10] M. Gates, H. Anzt, J. Kurzak, and J. Dongarra, "Accelerating collaborative filtering using concepts from high performance computing," in *Big Data (Big Data), 2015 IEEE International Conference on*. IEEE, 2015, pp. 667–676.

[11] R. C. Whaley, A. Petitet, and J. Dongarra, "Automated empirical optimizations of software and the ATLAS project," *Parellel Comput. Syst. Appl.*, vol. 27, no. 1-2, pp. 3–35, 2001, http://dx.doi.org/10.1016/S0167-8191(00)00087-9.

[12] 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," LAPACK Working Note, Tech. Rep. 111, August 8 1996, university of Tennessee Computer Science Technical Report UT-CS-96-326. [Online]. Available: http://www.netlib.org/lapack/lawnspdf/lawn111.pdf

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

[14] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, 2005, special issue on "Program Generation, Optimization, and Adaptation".

[15] M. Puschel, J. M. F. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: Code generation for DSP transforms," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 232–275, 2005, http://dx.doi.org/10.1109/JPROC.2004.840306.

[16] R. Veras and F. Franchetti, "Capturing the expert: Generating fast matrix-multiply kernels with spiral," in *High Performance Computing for Computational Science – VECPAR 2014, The Ninth International Workshop on Automatic Performance Tuning (iWAPT)*, M. Daydé, O. Marques, and K. Nakajima, Eds., 2014, pp. 236–244.

[17] C. A. Schaefer, V. Pankratius, and W. F. Tichy, "Atune-IL: An instrumentation language for auto-tuning parallel applications," in *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, S. B. . Heidelberg, Ed., vol. LNCS, no. 5704/2009, Aug. 2009, publikation, pp. 9–20. [Online]. Available: /Tichy/uploads/publikationen/216/original.pdf

[18] S. Kamil, "Productive high performance parallel programming with auto-tuned domain-specific embedded languages," Ph.D. dissertation, University of California, Berkeley, 2012, tech Report EECS-2012-255.

[19] S. Benkner, F. Franchetti, M. Gerndt, and J. K. Hollingsworth, "Automatic application tuning for hpc architectures," *Dagstuhl Reports*, vol. 3, no. 9, pp. 214–244, January 2014, http://dx.doi.org/10.4230/DagRep.3.9.214; http://drops.dagstuhl.de/opus/volltexte/2014/4423.

[20] "Assembler for NVIDIA Maxwell architecture," Online https://github.com/NervanaSystems/maxas, 2015. [Online]. Available: https://github.com/NervanaSystems/maxas

[21] T. Hastie, R. Tibshirani, and J. Friedman, *Elements of Statistical Learning*, 2nd ed. Springer, 2009.

[22] A. Liaw and M. Wiener, "Classification and regression by randomForest," *R News*, vol. 2, no. 3, pp. 18–22, 2002. [Online]. Available: http://CRAN.R-project.org/doc/Rnews/

[23] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.