A Guide For Achieving High Performance With Very Small Matrices on GPU: A case Study of Batched LU and Cholesky Factorizations

Azzam Haidar*, Ahmad Abdelfattah*, Mawussi Zounon ‡, Stanimire Tomov*, Jack Dongarra*†‡
{haidar,ahmad,tomov,dongarra}@icl.utk.edu,mawussi.zounon@manchester.ac.uk
*Innovative Computing Laboratory, University of Tennessee, Knoxville, TN, USA
†Oak Ridge National Laboratory, Oak Ridge, TN, USA
‡University of Manchester, Manchester, UK

Abstract—We present a high-performance GPU kernel with a substantial speedup over vendor libraries for very small matrix computations. In addition, we discuss most of the challenges that hinder the design of efficient GPU kernels for small matrix algorithms. We propose relevant algorithm analysis to harness the full power of a GPU, and strategies for predicting the performance, before introducing a proper implementation. We develop a theoretical analysis and a methodology for high-performance linear solvers for very small matrices. As test cases, we take the Cholesky and LU factorizations and show how the proposed methodology enables us to achieve a performance close to the theoretical upper bound of the hardware. This work investigates and proposes novel algorithms for designing highly optimized GPU kernels for solving batches of hundreds of thousands of small-size Cholesky and LU factorizations. Our focus on efficient batched Cholesky and batched LU kernels is motivated by the increasing need for these kernels in scientific simulations (e.g., astrophysics applications). Techniques for optimal memory traffic, register blocking, and tunable concurrency are incorporated in our proposed design. The proposed GPU kernels achieve performance speedups vs. CUBLAS of up to $6\times$ for the factorizations, using double precision arithmetic on an NVIDIA Pascal P100 GPU.

Index Terms—batched computation, GPUs, variable small sizes

1 Introduction

Although it might seem like an attractive idea to focus the efforts of the high-performance computing (HPC) community on addressing large-scale problems, the experience of the research community over the last few years has clearly shown that applications that use many small matrices or tensors cannot make efficient use of modern HPC systems and the associated vendor-optimized linear algebra libraries. The existing libraries have been designed for large matrices, and—historically—the scope of vendor libraries has been too narrow to address matrix computation problems. Consequently, the performance that these libraries deliver tends to be inadequate. Moreover, there are good reasons to believe that neither improved compiler technology nor autotuning will make any significant headway on this problem. This lack of coverage by current library infrastructure is especially alarming because of the number of applications from important fields that fit this profile, including deep learning [9], data mining [32], astrophysics [24], image and signal processing [5], [25], hydrodynamics [11], quantum chemistry [6], and computational fluid dynamics (CFD) and the resulting partial differential equations (PDEs) through direct and multifrontal solvers [42], to name a few. Dramatically better performance on these applications can be achieved by using software that can repetitively execute small matrix/tensor operations grouped together in "batches." However, the near total lack of such capabilities in existing software has forced users to rely on different inefficient solutions.

For example, in the NWChem package used in chemistry problems, the computation of the two-electron integrals and the Fock matrix becomes a bottleneck when many integrals of small size have to be computed independently, which shows the necessity of optimized batched libraries. Moreover, in [35], the authors discussed the optimization of NWChem for Intel's MIC architecture and highlighted the need for tensor computations of about 200–2,000 matrices from 10×10 to 40×40 in size. In his dissertation, David Ozog discussed NWChem's Tensor Contraction Engine (TCE) and revealed how strongly it relies on the performance of general matrix-matrix multiplication (GEMM) in the computation of the tensor contraction. In the summary of the work done by [20], the authors noted that small matrix computations have a severe bottleneck, and specialized libraries are required. The need for efficient libraries for thousands of small matrix computations was also discussed at NVIDIA's GPU Technology Conference in 2016 by Mniszewski et al. [34] in the context of their research on quantum molecular dynamics, where the dense matrix computation can be performed as an independent set of computations. Another important motivation is that the matrix polynomial can be computed in a batch fashion as a set of small, independent computations [27].

Deep Learning communities have also showed a significant interest in computations involving many small matrices. NVIDIA [10] already highlighted the need for a batched GEMM routine and has also started providing some of the batched routines (e.g., GEMM, triangular solver matrix [TRSM], LU, QR, inversion) for fixed size in their cuBLAS library. Nervana [26], which is one of the pioneers of deep learning, demonstrated the critical need for batched matrix computation kernels for high-performance deep learning software. Intel has also provided batched GEMM and batched TRSM routines for fixed matrix sizes.

For the block-mass, weighted Hessian in the molecular dynamics simulation [36], computing the eigendecomposition involves computing the eigendecomposition of many small, independent matrices, which can be viewed as a batched eigendecomposition. In addition to the package cited above, we note that in the GROMACS package [2], because the particle-particle and mesh interactions are independent, batched computation can be used to speed up and overlap the expensive global communication in the Particle-mesh Ewald (PME). Also, in [31], the authors noted the need for optimized and hardware-aware basic linear algebra subprograms (BLAS) to perform many independent computations inside the GROMACS MD simulation.

The approach behind Flash Principle Component Analysis (FlashPCA) performs a large number of eigendecompositions across many samples. Also, in combustion and astrophysics supernova applications [7], [8], [18], [24], [33], the study of a thermonuclear reaction networks (XNet package) requires the solution of many sparse linear systems of around 150×150. Furthermore, the need for batched routines can be illustrated in radar signal processing [5], where a batch of 200×200 QR decompositions is needed, as well as in hydrodynamic simulations [11], where thousands of matrix-matrix and matrix-vector (GEMV) products of matrices of around 100×100 are needed.

Although the brief introduction above shows some viable approaches, it mostly highlights the keen awareness of the need for batched libraries that can enable small matrix applications to finally start exploiting the full power of current and future hybrid platforms. Some vendors have started to provide some batched functionalities in their numerical libraries (e.g., NVIDIA's CUBLAS and Intel's Math Kernel Library [MKL]). Additionally, some open-source libraries from the HPC community (e.g., the Matrix Algebra on GPU and Multicore Architectures [MAGMA] library [37]) have also started to deliver batched routines [12], [13], [19]. While performance has been improving with these contributions, there is still a lack of understanding of how to design, implement, analyze, and optimize batched routines to exploit modern architectures at full efficiency. The goal of this paper is to develop a theoretical analysis and a methodology for high-performance linear solvers. As test cases, we take the Cholesky and LU factorizations and show how the proposed methodology enables us to achieve performance close to the theoretical upper bound.

2 Contributions

The primary goal of this paper is to propose a framework design for batched algorithms and to study their efficient implementations. We believe this study will help HPC application developers more effectively harness GPUs and achieve performance close the theoretical peak of the hardware. Our primary contributions to this end are listed below.

- In addition to efficient implementations that exhibit a good speedup, we provide a detailed analysis of optimization techniques. We also present a collection of best practices to help users understand and develop batched computation kernels in a simple and efficient fashion.
- We propose a methodology for designing a performance model that provides insight into the performance spectrum of batched kernels. The main advantage of this model is that it helps predict the achievable performance of the kernels with increased accuracy.
- We investigated a model that simplifies the autotuning process by considering hardware information and representative experiments that enable us to considerably reduce the configuration/parametrization search space. We expect this contribution to drastically reduce the significant autotuning time required by complex GPU kernels.
- We propose a modular design that relies on standard data formats and interfaces to enable a straightforward connection with mainstream application code.

3 Related Work

At the time of writing, there are no specifications for the computation of batched small linear algebra problems. Consequently, the Linear Algebra PACKage (LAPACK), which is the de-facto standard library for linear algebra problems, doesn't provide such a functionality. However at the request of users, NVIDIA added a few batch kernels to CUBLAS 6.5 [28]. Those additions include a batched version of both BLAS and LAPACK. For BLAS kernels, batched GEMM and batched TRSM have been released. More effort has been put into the direction of LAPACK kernels, resulting in a batched version of LU and QR factorizations, matrix inversion, and a least squares solver. Similarly, in response to customer demand, Intel's MKL team released a batched GEMM, while—at the time of writing—AMD does not provide any batched operations. Vendor efforts on batched BLAS may have a tremendous impact and enhance the portability of HPC applications, much like optimized BLAS did [14] when released. While real-world applications may require solving a batch of small matrices of different dimensions, the batched kernels developed by NVIDIA and Intel are limited to the case where the matrix problems in the batch are of the same dimension. NVIDIA's release of four major batched LAPACK–based routines is significant; however, they did not address the problem of portability and device-specific redesigns of the batched LAPACK algorithms.

Batched linear algebra concepts could be applied to multicore CPUs as well. Indeed, small problems can be solved efficiently on a single core (e.g., using vendor-supplied libraries like MKL [22] or the AMD Core Math Library [ACML] [4]), because the CPU's memory hierarchy would support a "natural" data reuse (small enough problems can fit into small, fast memory). Besides memory reuse, to further speed up the computation, vectorization can be added to use the supplementary single instruction, multiple data (SIMD) processor instructions—either explicitly as in the Intel Small Matrix Library (SML) [21], or implicitly through the vectorization in BLAS. Batched factorizations can then be efficiently computed for multicore CPUs by having a single core factorize a single problem at a time.

For higher-level routines, prior work has concentrated on achieving high performance for large problems using hybrid algorithms [38]. The motivation came from the fact that the GPU's compute power cannot be used on a panel factorization as efficiently as it can on trailing matrix updates [39]. As a result, various hybrid algorithms were developed—where the panels are factorized on the CPU, while the GPU is used for trailing matrix updates (mostly GEMMs) [3], [15]. For large-enough problems, the panel factorizations and associated CPU-GPU data transfers can be overlapped with GPU work. For small problems, however, this is not possible, and our experience has shown that hybrid algorithms would not be as efficient as they are for large problems.

To compensate for the lack of support for batched operations, application developers implemented customized batched kernels using various approaches. For example, targeting very small problems (no larger than 128×128), Villa et al. [29], [30] designed a GPU kernel for a batched LU factorization, where a single CUDA thread, or a single thread block, was used to solve one system at a time. Similar techniques, including the use of single CUDA thread warp for a single factorization, were investigated by Wainwright [40] for LU factorization with full pivoting on small matrices of up to 32×32. These problems are small enough to fit in the GPU's shared memory (48 KB on a K40 GPU) and thus can benefit from data reuse. Unfortunately, the results showed these strategies do not exceed the performance of memory-bound kernels like GEMV.

4 Methodology, Analysis, and Discussion

In this section, we present our methodology for analyzing high-performance batched linear algebra computations and discuss the insight and theory required to design, implement, and optimize algorithms to run efficiently on modern GPU architectures. We also provide algorithm design guidance to ensure an efficient and effortless portability of batched linear algebra kernels across a large range of GPU architectures.

It is a common misconception that algorithm analysis is only useful for squeezing the last possible 5% of performance from very small matrix applications. In other words, when forgoing this analysis, one sacrifices only 5% in performance while avoiding a rigorous algorithm analysis. While it is true that some specific multicore CPU applications do not gain much from algorithm analysis, accelerator-based applications have far more potential, since their underlying principles are fundamentally different from conventional CPUs.

CPUs have accumulated design complexity that enables them to optimize instruction streams by looking ahead hundreds of instructions at a time. Consequently, CPUs can resolve data dependence hazards, predict branch decisions, and buffer cache/memory requests efficiently. The majority of these features are missing from GPU "cores," which—for the sake of accuracy—should be called "processing units." Intel Xeon Phi coprocessors are only marginally better with their basic cache coherency, but they still require programmer/compiler-directed scheduling to use their in-order execution at full efficiency. Thus, many factors and constraints should be studied carefully to achieve relevant insight and provide a design framework that could benefit the research and development community. One of the main issues associated with working on small matrices is that the overall execution time is dominated by data transfer, because the time required to process a small matrix on modern GPUs is negligible. Put differently, the computation of small matrices follows the trend of memory-bound algorithms, and the performance strongly correlates with data transfer rather than floating-point operations (FLOPs).

On CPUs, very small matrices are more likely to remain in *at least* the L2 cache. With such a configuration, using each core to solve one problem at a time is enough to achieve reasonably high performance. This makes the development of batched algorithms easy to handle and optimize with a relatively small effort from the CPUs. However, for GPUs the cache size is very small (48–64 KB for on newer GPUs), which makes batched linear algebra kernel implementation more challenging. Building and implementing an efficient GPU batched algorithm kernel requires an understanding and analysis of many factors, which we address in this work.

In addition, we demonstrate that, because GPU application design puts the focus on maximizing data throughput rather than minimizing processing latency, the common practice of optimizing a sequential implementation first and then optimizing the parallel version no longer applies. For well designed kernels focused on large matrix-matrix type operations, autotuning helps a lot in achieving a good performance. However, autotuning is becoming overrated in the GPU programming community, where smaller matrices reign supreme. A common misinterpretation of recent autotuning studies is to believe that an efficient autotuning framework is enough to harness a GPU's performance. However, when it comes to operations like solving a batch of very small matrix problems, even the most powerful autotuning framework, without the correct algorithm design, will fail to provide a good performance. That said, if one designs the kernel using both an algorithmic analysis and a good understanding of the underlying hardware, the autotuning framework can be simplified and tested/implemented. Unlike most autotuning approaches, though, our strategy does not require hundreds of thousands of runs followed by result interpretations to provide an efficient kernel.

Finally, we analyze the effect of reshaping the data storage into a non-conventional storage to design highly optimized kernels for batches of small linear algebra operations.

4.1 Theoretical Analysis and Performance Roofline Model

In this section, we discuss the theoretical analysis of algorithms designed for batches of very small matrix problems.

A detailed study based on Cholesky and LU factorization algorithms are used for the sake of illustration. The roofline model for processing very large matrices can be easily presented without much complexity. On the other hand, working out an accurate roofline model for matrices of small to medium size involves a lot of complexities related to both the algorithm and the hardware itself.

The roofline model for large size matrix computations has been widely studied, and a remarkable discussion of matrix computation roofline models by James Demmel can be found in [41]. In general, large matrix computation algorithms can be classified into three categories, listed below.

(1) Compute-Intensive Algorithms: The first category includes compute-intensive algorithms, which are characterized by a high arithmetic intensity. Arithmetic intensity is the ratio of total FLOPs to total data movement (bytes). In this case, the computation time is dominant compared to the data transfer time. Consequently, a good implementation could easily overlap the communication time with computation, which leads to a roofline model mostly defined by the arithmetic intensity. The upper bound is then limited by the computation's peak performance. As an example, an optimized GEMM kernel for large matrices can achieve performance close to that of the machine's peak.

(2) Memory/Bandwidth-Bound Algorithms: The second category includes low arithmetic intensity algorithms—also known as memory-bound algorithms or bandwidth-bound algorithms. Matrix-vectors and vector-vector operations are typical examples of these algorithms.

(3) Compute-Intensive, Bandwidth-Bound Algorithms: The third category deals with algorithms that lie somewhere between the previous two. These algorithms require a detailed analysis in order to evaluate their performance upper bound. This is the case, for example, when applying matrix-matrix operations on small matrices. While a matrix-matrix operation itself has a high arithmetic intensity, owing to the significant data transfer latency, the time to process the small matrices may be closer to the time required to move the matrices. Since processing very small matrices falls in this category, we provide more details on assessing its performance upper bound and give some recommendations for medium-size matrix computations.

The theoretical bound of floating-point performance is computed as follows: $P_{max} = FLOPs/T_{min}$, where $FLOPs$ is the number of floating-point operations, and T_{min} the minimum time to solution. The T_{min} can be defined as

$$T_{min} = min(T_{Read} + T_{Compute} + T_{Write}). \qquad (1)$$

Depending on the implementation and the type of algorithm, there might be some overlap between the data transfer step (read and write) and the computation steps. However, on modern high-performance architectures that are capable of achieving many FLOPs per cycle per core, the time required to read and write very small size matrices is about 1–2 orders of magnitude higher than the computation time. The time to read or write is predefined by the bandwidth, while the computation time is determined by both the speed of the hardware and the efficiency of the implementation. Thus, for very small matrix operations, any algorithm—even the historically compute-intensive matrix-matrix multiplication—is bounded by the data movement, and its upper-bound performance can be easily derived. For a generic description, and for matrices of medium size (larger than 32×32), we assume that the algorithm is elegantly designed. This means that the algorithm has the best computation/communication overlap, even though it is less likely to be the case for very small matrices. In such a situation, one can easily predict the minimal time T_{min} and thus the performance upper bound.

Let us take LU factorization as an example. The LU factorization algorithm reads an $n \times n$ matrix, meaning n^2 elements, and processes $\frac{2}{3}n^3$ FLOPs and writes back n^2 elements. On the other hand, in double precision, an NVIDIA P100 GPU can perform 5,300 gigaFLOP/s, while its maximum achievable read/write bandwidth is about 600 GB/s. On the P100 hardware, the time to transfer one byte is approximately $9\times$ higher than the time required to complete one FLOP. Consequently, the time to complete a batch of small matrices operations will be dominated by the data transfer time. In other words, even with a full overlap, the minimal time will be bounded by the data movement time, making the computation time negligible. To approximate the roofline upper bound, we can define:

$$P_{\text{upper bound}} = FLOPs/T_{\text{data transfer}} = FLOPs \times \frac{\beta}{M}, \qquad (2)$$

where $FLOPs$ is the number of floating-point operations, β is the achievable bandwidth of the hardware, and M is the size of the data to be moved (load and store).

4.2 The Occupancy and Bandwidth Analysis

GPU occupancy has been treated as a performance indicator for many years. Unfortunately, although the occupancy is correlated to performance, it does not necessarily imply that a code that achieves a high occupancy will also deliver high performance.

The importance of the occupancy parameter and its correlation to high performance varies between compute-intensive kernels and bandwidth-bound kernels. For example, with about 12% occupancy, a compute-intensive, matrix-matrix multiplication achieves performance close to that of the machine's peak. The performance of a compute-intensive kernel is determined by the amount of data reuse (i.e, a high arithmetic intensity). More over, occupancy is not the most determinant factor in achieving good performance. In fact, the occupancy is defined as the ratio of active threads running over the total number of active threads supported by the hardware (2,048 threads for the NVIDIA P100 GPU). For example, if there are 336 thread blocks (TBs) executing simultaneously on each of the P100's 56 streaming multiprocessors (SMXs), and each uses 256 threads, then the occupancy is $(6 \times 256)/2048 = 75\%$. A bandwidth-bound, matrix-vector kernel attains about 3% of the machine peak, while it reaches about 95% of the achievable bandwidth with an occupancy between 60%–90%. Since it is a bandwidth-bound kernel, it is always preferable to use bandwidth as the metric rather than FLOPs per second (FLOP/s).

As shown by the representative experiment in Figure 1, the achievable bandwidth strongly correlates to the number of threads per TB and to the number of TBs running simultaneously per SMX. In other words, the achievable bandwidth is strongly correlated to the total number of threads running on an SMX. We use the metric "per SMX" since the GPU hardware specifications are mostly defined by the number of SMXs. However, for the sake of clarity, when we show 6 TBs/SMX, we mean that there are $56 \times 6 = 336$ TBs running on the whole P100 GPU. This will be one of the important data points to consider when implementing a memory-bound algorithm or when designing a batched algorithm for small-size matrices. Note that any type of algorithm that operates on small amounts of data is considered to be bandwidth-bound as shown in Section 4.1.

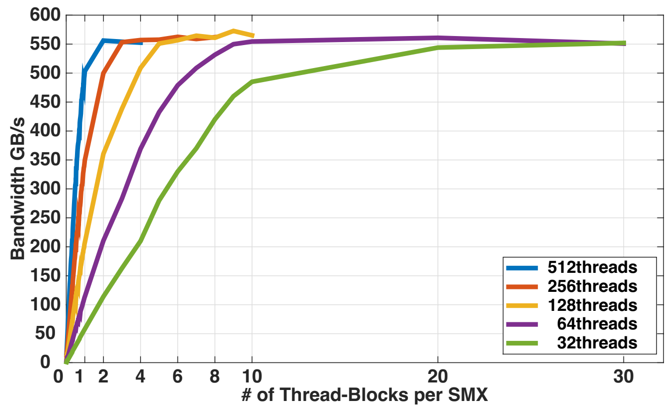


Fig. 1. The achievable bandwidth based on the number of TBs per SMX and the number of threads in each TB on the NVIDIA P100 GPU. Note: to obtain the total number of TBs running on the whole GPU, the x axis value should be multiplied by 56 since the P100 has 56 SMXs.

The number of threads per TB is mostly related to the algorithmic design of the target application. For example, a Cholesky factorization proceeds column by column, from left to right. One column is processed at a time followed by the update of all the right-side columns, while the columns on the left remain untouched. These algorithm details, along with the matrix size, are very important and should guide the design choices when considering thread configurations.

To design a kernel for a batched Cholesky factorization, one has the choice of using a 1-D or a 2-D grid of threads for each matrix factorization. For an $m \times m$ square matrices factorization, the 2-D configuration is the most intuitive choice since the implementation can be straightforward, with an $m \times m$ thread mapping, resulting in a $1:1$ thread-to-matrix entry. Unfortunately, this will require using heavyweight TBs, which are relatively expensive to manage. Put differently, using a 2-D configuration will result in limited TBs per SMX, consequently inducing a low-bandwidth situation, as illustrated by the experiment depicted in Figure 1. In addition, the Cholesky algorithm is sequential by column, and it is only during the update steps that the whole 2-D grid can be involved (i.e, during a column process, all of the threads that are not involved will be in an idle state, losing a lot of resources). Another penalty associated with the 2-D configuration is the synchronization. In fact, a 2-D configuration is more likely to exceed the warp size (32 threads), and barriers will be required when accessing shared data.

To avoid the drawbacks exhibited by the 2-D configuration, we propose a design based on a 1-D configuration. For example, for an $m \times m$ matrix, we use a TB with m threads. This means more work per thread and therefore more room for parallelism.

4.3 An Analytical study of the Algorithmic Design

This section is dedicated to the efficient implementation of the Cholesky factorization kernel based on a design that uses a 1-D grid of threads. Designing this kernel involves a relatively high level of complexity. The design is critical because a non-optimal decision could be penalizing in terms of autotuning time.

Simply put, a basic kernel design could consist of the following steps. (1) Load the whole matrix into the shared memory or into the register. (2) Perform all necessary computations. (3) Finally, write the result back to the main memory. This approach is the most common for compute-intensive kernels used to solve large matrix problems. However, this design decision is not an attractive option for solving thousands of small, independent matrix operations. On modern GPUs, the shared memory size is 64 KB per SMX. This means that, in double precision, each SMX cannot hold matrices larger than 80×80. Therefore, a shared-memory kernel that implements the algorithm, which consists of loading the whole matrix into the shared memory and performing the factorization, will be limited to solving matrices that are 80×80 or smaller in double precision and 160×160 or smaller in single precision. Similarly, if we use the register to hold the matrix, we will also be limited to matrices that are 128×128 or smaller in double precision (the register file size is about 256 KB, while about half of that will be used for internal variables and parameters). In addition, using the full shared memory or too many registers will severely limit the number of active TBs per SMX.

According to our representative experiments illustrated in Figure 1, having 1–2 TBs per SMX will result in low bandwidth. Since small-size matrix computations are bandwidth-bound, using a few TB per SMX is a bad design decision that can lead to poor performance. To achieve good performance from very small–size matrices (up to 32×32), one option is to use roughly 8^+KB of shared memory per TB, with 7 TBs per SMX. However, when the matrix size exceeds 32×32, another design option should be investigated.

For the sake of simplicity, we rely on the shared-memory implementation to describe the design for matrices larger than 32×32. The register-based version is very similar. Later, we will revisit both the shared memory and the register versions for LU factorization. To address matrices larger than 32×32, the key idea is to divide the whole matrix into block columns—also known as "panels." For example, an $n \times n$ matrix will be divided into blocks of $n \times ib$, where ib—called the "block size"—is the number of columns per block. This modification helps us avoid loading the whole matrix directly into shared memory and instead allows us to load it panel by panel. This is known as a "blocking technique." Since the panel factorization is mainly sequential, splitting the factorization into panels is a reasonable design decision.

There are many versions of the Cholesky factorization. Here, we discuss the *right-looking*, the *left-looking*, and the *top-looking* variants. As an example, we show the analysis of the *left-looking* and the *right-looking* variants. In the *right-looking* variant illustrated in Algorithm 1, the matrix is factorized, per panel, from left to right. At each step, the current panel is processed followed by an immediate update of the panel at the right side of the current panel. In the *left-looking* variant described in Algorithm 2, at each step the update from previous panels (left side) are applied to the current panel before performing the computations on the current panel. In other words, the updates are applied as late as possible, while on the *right-looking* algorithm, the updates are applied as soon as possible. We refer the reader to [23] for further details on the different Cholesky implementations.

Algorithm 1: The right-looking fashion.

```
for $i \leftarrow 0$ to $m$ Step ib do
    // Panel factorize $\mathbf{A_{i:m,i:i+ib}}$
    POTF2 $A_{i:i+ib,i:i+ib}$;
    TRSM
     $A_{i+ib:m,i:i+ib} = A_{i+ib:m,i:i+ib} \times A^{-1}_{i:i+ib,i:i+ib}$;
    // Update trailing matrix $\mathbf{A_{i+ib:m,i+ib:m}}$
    SYRK $A_{i+ib:m,i+ib:m} =$
     $A_{i+ib:m,i+ib:m} - A_{i+ib:m,i:i+ib} \times A^{T}_{i+ib:m,i:i+ib}$;
end
```

Algorithm 2: The left-looking fashion.

```
for $i \leftarrow 0$ to $m$ Step ib do
    if $(i > 0)$ then
        // Update current panel $\mathbf{A_{i:m,i:i+ib}}$
        SYRK $A_{i:i+ib,i:i+ib} =$
         $A_{i:i+ib,i:i+ib} - A_{i:i+ib,0:i} \times A^{T}_{i:i+ib,0:i}$;
        GEMM $A_{i+ib:m,i:i+ib} =$
         $A_{i+ib:m,i:i+ib} - A_{i+ib:m,0:i} \times A^{T}_{i:i+ib,0:i}$;
    end
    // Panel factorize $\mathbf{A_{i:m,i:i+ib}}$
    POTF2 $A_{i:i+ib,i:i+ib}$;
    TRSM
     $A_{i+ib:m,i:i+ib} = A_{i+ib:m,i:i+ib} \times A^{-1}_{i:i+ib,i:i+ib}$;
end
```

To analyze the roofline bound of the *right-looking* variant, let's calculate the volume of the data transfer. At each iteration, applying the updates from the current panel requires loading and storing back all the panels at the right of the current one. For the sake of exposure, the panels at the right of the current panel will be referred to as the "trailing matrix." If the shared memory is used to hold the current panel, then a register will be dedicated to the storage of a portion of the trailing matrix to update and vice versa. The main drawback exhibited by this algorithm is the unnecessarily large amount of data movement. In fact, the first panel will be loaded once, only for its factorization. The second panel will be loaded once to apply the updates for the first panel, and the second panel will be loaded the second time for its own factorization; in general, the k^{th} panel will be loaded k times. In terms of communication volume, at iteration k, the current panel will be of size $(n-(k-1)ib) \times ib$, and the trailing matrix will be of size $(n-(k-1)ib) \times (n-k \times ib)$. The sum gives $(n-(k-1)ib)^2$. However, since the Cholesky factorization is applied to a symmetric positive definite matrix, only half of the matrix needs to be considered, which reduces the communication volume to $\frac{1}{2}(n-(k-1)ib)^2$. Since each panel is loaded and then stored back, we can infer that the volume of data transfer can be multiplied by two, which means that the total communication volume at the k^{th} iteration is $(n-(k-1)ib)^2$. Assuming that the matrix is divided into p same-size panels ($n = p \times ib$), the total volume of communications for the right looking variant is

$$\begin{aligned} V &= \text{data read} + \text{data written} \\ &= \textstyle\sum_{k=1}^{p} \left(n-(k-1)ib\right)^2, \\ &= \textstyle\sum_{k=1}^{p} (p \times ib - (k-1)ib)^2, \\ &= ib^2 \textstyle\sum_{k=1}^{p} (p+1-k)^2, \\ &= ib^2 (\tfrac{p^3}{3} + \tfrac{p^2}{2} + \tfrac{p}{6}), \\ &\approx ib^2 \tfrac{p^3}{3} = ib^2 \tfrac{n^3}{3ib^3}, \\ &\approx \tfrac{1}{3}\tfrac{n^3}{ib}. \end{aligned} \tag{3}$$

In contrast, to the *right-looking* algorithm, the *left-looking* variant loads a panel, applies the updates from previous panels (left side of the panel), factorizes, and stores back. With respect to the *right-looking* design, the current panel is stored into the shared memory, portions of the matrix on its left loaded into a register, in order to perform its update. At the k^{th} iteration, the volume of data involved in loading the current panel and storing it back is $(n-(k-1)ib) \times ib$, while the volume of data required for moving the updates from previous panels is $(n-(k-1)ib) \times (k-1)ib$. Thus, the volume of communication at iteration k is $k(n-(k-1)ib)ib$. Since $n = p \times ib$, we get $k(p-k+1)ib^2$. As consequence, the total amount of communications for the left looking variant is:

$$\begin{aligned} V &= \text{data read} + \text{data written} \\ &= ib^2 \textstyle\sum_{k=1}^{p} k(p-k+1), \\ &= ib^2 \textstyle\sum_{k=1}^{p} (k(p+1)-k^2), \\ &= ib^2 \left((p+1)\tfrac{(p+1)p}{2} - (\tfrac{p^3}{3} + \tfrac{p^2}{2} + \tfrac{p}{6})\right), \\ &= ib^2 \left(\tfrac{1}{6}p^3 + \tfrac{1}{2}p^2 + \tfrac{1}{3}p\right) \approx \tfrac{1}{6} ib^2 p^3, \\ &\approx \tfrac{1}{6}\tfrac{n^3}{ib}. \end{aligned} \tag{4}$$

Since very small size matrix processing is bandwidth-bound, and the performance depends on the volume of data transfer, we can now derive the roofline performance upper bound for both the *right-looking* and the *left-looking* variants. The Cholesky factorization of an $n \times n$ matrix consumes about $\frac{1}{3}n^3$ FLOPs. Thus, with respect to Equation (2), in double precision, we can expect the *right-looking* version to have an asymptotic performance upper bound of $\frac{1}{3}n^3 \times \frac{3ib\beta}{8n^3} = \frac{ib\beta}{8}$. Using the same method, the *left-looking* variant will be bounded by $\frac{ib\beta}{4}$, in double precision, which means that—in theory—the *left-looking* variant could achieve twice the performance of the *right-looking* implementation, in the context of very small size matrices. To decide whether to use the *right-looking* or the *left-looking* algorithm, a traditional approach consists of prototyping both approaches and going through long autotuning sweeps before assessing the performance of the two algorithms. However, based on our effective algorithm analysis, we proved that the *right-looking* variant is not suitable for very small matrices.

At this point, the primary question is the effectiveness of our algorithm analysis. In an ideal scenario, the bandwidth $\beta \approx 600GB/s$. With $ib = 8$, the performance *left-looking* kernel is bounded by $1,200$ gigaFLOP/s. On the other hand, the experimental results of our kernel based on the *left-looking* design is depicted in Figure 2a. As illustrated by the chart, the performance obtained is far from the 1,200 gigaFLOP/s performance upper bound. Achieving half of the upper bound performance does not necessary mean that the computation is expensive or sequential. It is important to note that the results displayed have been intensively autotuned, and only the best results have been reported.

A careful study of the design, along with the information reported in Figure 1, could—in fact—allow an accurate guess of most of the results displayed in Figure 2a. For the sake of illustration, let us take $n = 512$ and $ib = 8$ as examples. Based on the algorithm characteristics, the code requires about $n \times ib + ib^2$ elements to be stored in shared memory, which is about 32.5 KB using 512 threads. As result, only 1 TB can run per SMX, meaning our achievable bandwidth in this condition is about 420 GB/s. Consequently, a reasonable performance upper bound is 840 gigaFLOP/s. This demonstrates the effectiveness of our implementation. However, more investigations may provide additional information that will help us understand why we did not make it close to the upper bound. An advanced analysis of the Cholesky factorization revealed that, for the first panel, the algorithm requires all 512 threads to work. However, for the next panel, the number of threads required decreases by ib and so on until the last panel, where only ib threads have to work. This is an interesting clue for understanding why we failed to reach the performance upper bound. Since the real bandwidth is a function of the number of threads and the number of TBs, we achieve 420 GB/s with 1 TB per SMX, when 512 threads are working. But by using only 128 threads with 1 TB per SMX, the bandwidth decreases up to 200 GB/s. Thus, if we reformulate our performance analysis based on these details, we can display the upper bound corresponding to each ib. With this, we realized in advance that $ib = 8$ or $ib = 10$ will be among the optimal configurations. This shows that our model can also serve as a base to prune the autotuning search space. Consequently, the autotuning process is simplified considerably.

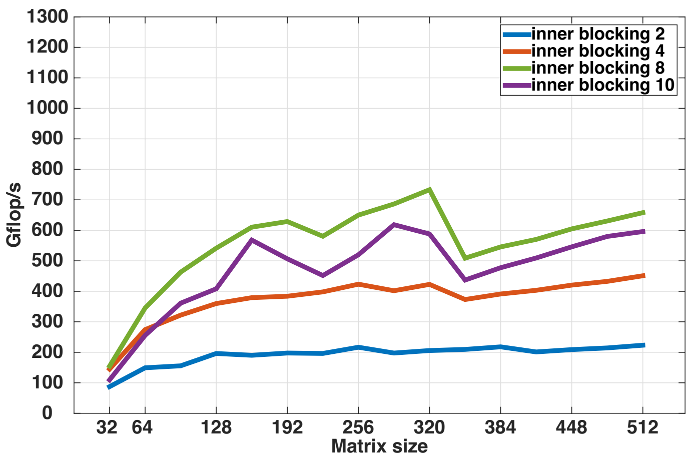
As shown in Figure 1, when a small number of threads are used in a TB (e.g., 32 threads) it is beneficial to run more than 8 TBs per SMX in order to extract a high bandwidth. Unfortunately, run more TBs we have to decrease the value of ib, which is more likely to decrease our roofline bound. Therefore, $ib = 8$ is the best scenario for the current design. For example, for $n = 512$, we need to allow more than 1 TB per SMX when, say, only 64 threads are working. By studying the algorithm, we found that the shared memory requirement to reserve $n \times ib + ib^2$ elements, where n is the size of the matrix, is the constraint that does not allow more than 1TB per SMX. However, for the factorization of a remaining portion of size 64×64, only 64 threads are required, and in terms of memory, only $64 \times ib + ib^2$ is required, not the entire $512 \times ib + ib^2$. To optimize the shared memory usage, instead of allocating $n \times ib + ib^2$ for entire factorization, we revisited the algorithm to enable launching a kernel for every panel with the appropriate memory requirement. For example, when the remaining portion is 64×64, we need $64 \times ib + ib^2$ as a shared memory space, which will allow for an $ib = 8$ achieving up to 14 TBs per SMX using 64 threads each. Such a configuration is bandwidth friendly, and we can expect to extract more than 550 GB/s—instead of 100 GB/s with the previous design. This improvement is beyond the insight one can gain from autotuning.

When implementing this third design, we used the same kernel proposed above (design of Algorithm 2 *left-looking*), but we now use an optimal shared memory allocated at each step. Thus, we designed a fused GPU kernel that performs the four routines of one iteration of Algorithm 2. Such a design will minimize the memory traffic, increase the data reuse from shared memory, and reduce the overhead of launching multiple kernels. Our optimized and customized *fused kernel* performs the update (SYRK and GEMM operations) and keeps the updated panel in shared memory to be used by the factorization step. The cost of the left-looking algorithm is dominated by the update step, (SYRK and GEMM). The panel C, illustrated in Figure 3, is updated as $C = C - A \times B^T$. In order to decrease its cost, we implemented a double buffering scheme as described in Algorithm 3. We prefix the data array with "r" to specify the register and "s" to specify the shared memory. We prefetch data from A into the register array, rAk, while a multiplication is being performed between the register array rAkk and the array sB stored in shared memory. Since the matrix B is the shaded portion of A, our kernel avoids reading it from the global memory and transposes it in situ to the shared memory, sB. Once the update is finished, the factorization (POTF2 and TRSM) is performed as one operation on the panel C, held in shared memory.

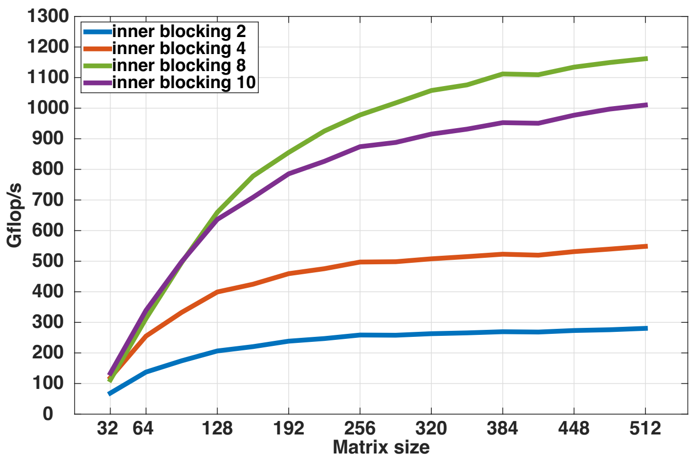
Algorithm 3: The fused kernel correspond to one iteration of Algorithm 2.

```
rAk ← $A_{(i:\mathrm{m},0:\mathrm{lb})}$; rC ← 0;
**for** $k$ ← 0 **to** *m-i* **Step** *lb* **do**
    rAkk ← rAk;
    sB ← $\mathrm{rAk}_{(i:\mathrm{lb},k:k+\mathrm{lb})}$ inplace transpose;
    barrier();
    rA1 ← $A_{(i:\mathrm{m},k+\mathrm{lb}:k+2\mathrm{lb})}$ prefetching;
    rC ← rC + rAkk×sB multiplying;
    barrier();
**end**
sC ← rA1 - rC;
factorize sC;
```

This implementation achieves close to the peak bandwidth, and—according to our model—in double precision $P_{\text{upper bound}} = \frac{ib\beta}{4} = 1,200$ gigaFLOP/s. The performance



(a) using more shared



(b) using min shared.

Fig. 2. Kernel design and autotuning of the Cholesky factorization.

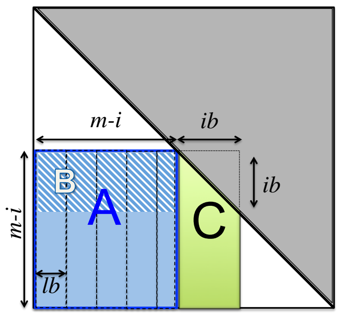


Fig. 3. left-looking Cholesky factorization

of this implementation is depicted in Figure 2b. Not only did the performance improve considerably, but we achieved performance very close to the predicted theoretical peak, which highlights both the efficiency of the implementation and the accuracy of our model.

The objective of this example is to learn and understand the expectation of a design without necessarily delving into the implementation and autotuning efforts. There is also a practical lesson from this study in that we now know autotuning strategies can only help one get close to the theoretical peak of the algorithm being designed. To achieve reasonable performance, one should investigate the design that has the highest theoretical bound before investing in autotuning. This should reduce the effort of researchers/developers in exploiting modern GPUs at full efficiency and provide an accurate performance spectrum.

4.4 Analysis and Design of Batched LU Factorization of Very Small–Size Matrices

In this section, we analyze another example of kernel design for batched computations of very small matrices ranging in size from 2×2 to 32×32. Using the LU factorization as an example in this study, the analysis performed in this section is similar to the experiments we defined for Cholesky. Since the target sizes are very small (less than 32×32), it is beneficial to keep the whole matrix in shared memory or in the register.

To accurately estimate the number of TBs that we should run simultaneously, we need a reliable estimate of the amount of shared memory required. This is also true for the number of registers. To ensure good performance, only a limited number of registers can be allocated per thread. In fact, for the number of registers per threads, beyond a certain bound, the number of TBs to run simultaneously will be reduced to only a few, which is detrimental to the bandwidth. To start our performance analysis, we first evaluated the amount of shared memory required in cases where the shared memory implementation is beneficial. We did the same study for the register version (i.e, where the whole matrix is stored in the registers). The left y-axis of Figure 4 shows the amount of shared memory (in KBs) required for the shared memory design (SH-MEM). The right y-axis of Figure 4 shows the number of registers per thread required by the register design (REG-v1).

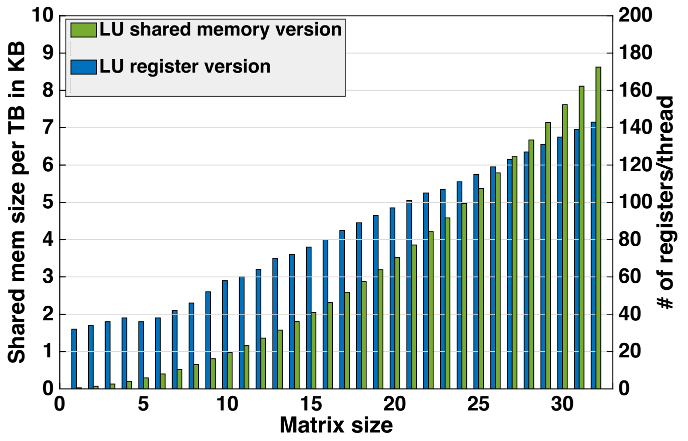


Fig. 4. The amount of shared memory (in KB) required by the SH-MEM design (left y-axis) and the amount of registers per thread required by the REG-v1 design (right y-axis) for the LU factorization in double precision for matrices ranging from 2×2 to 32×32 on an NVIDIA P100 GPU.

Using the data from Figure 4, we try to predict the performance bound of each design (SH-MEM and REG-v1) to select the most suitable candidate.

Based on the design options and hardware constraints, the optimal number of TBs per SMX can be approximated. For the SH-MEM case, the estimated optimal ratio of TBs per SMX is illustrated in Figure 5 (orange curve). With respect to hardware constraints, the maximum number of TBs per SMX is 32 (depicted in grey). Consequently, any efficient implementation of the SH-MEM design would need to set the TBs per SMX ratio based on the minimum of the hardware constraint (i.e., 32) and the estimate obtained from the algorithm analysis. The SH-MEM design is implemented, with the results illustrated by the blue curve of Figure 4, where the executed number of TBs per SMX were measured using the NVIDIA profiler tools. The effectiveness of our design is illustrated by the fact that the measured data matches the model estimations. However, our objective is not limited to determining the optimal TBs per SMX but rather to deduce the possible performance peak based on the number of TBs per SMX and to figure out the best implementation to optimize (i.e., SH-MEM vs. REG-v1). To this end, we did the same study for the REG-v1 design and presented (Figure 6) the number of optimal TBs per SMX (orange curve) computed from data on Figure 4. Similarly, we also measured the executed TBs per SMX during a real run of the code (blue curve) to assess our prediction. A comparison between Figure 5 and Figure 6 reveals that both versions allow and use the same number of TBs per SMX for matrices up to 20×20, above which the REG-v1 design allows a higher number of TBs per SMX. Consequently, for matrices ranging from 2×2 to 20×20, the same performance can be expected from both designs. However, for matrices beyond 20×20, the REG-v1 design should be preferred.

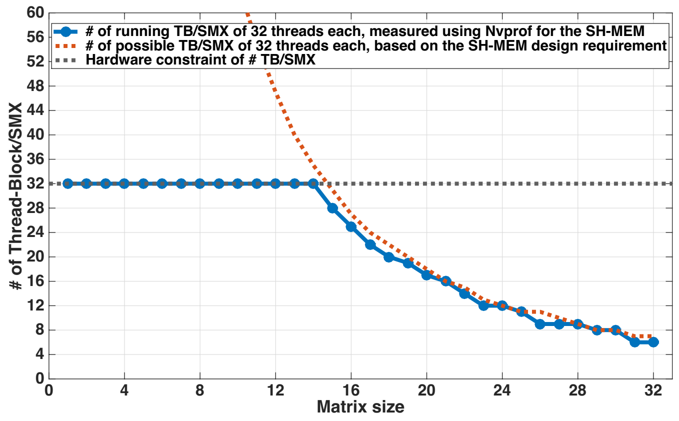


Fig. 5. Analysis of the number of TBs per SMX estimated. Real run for the SH-MEM design of the LU factorization on an NVIDIA P100.

Figure 7 shows the performance obtained (in gigaFLOP/s) using the two designs. This experimental result is consistent with our analysis (i.e, for matrices ranging from 2×2 to 20×20, the SH-MEM design and the REG-v1 design exhibit similar performance). Also, as expected, for matrices larger than 20×20, the REG-v1 design outperformed the SH-MEM design. Such consistency should be of a great interest to the community, and the proposed model is an excellent tool for understanding and analyzing the design of algorithms prior to implementation. The model also allows us to understand our code and let us find the correct

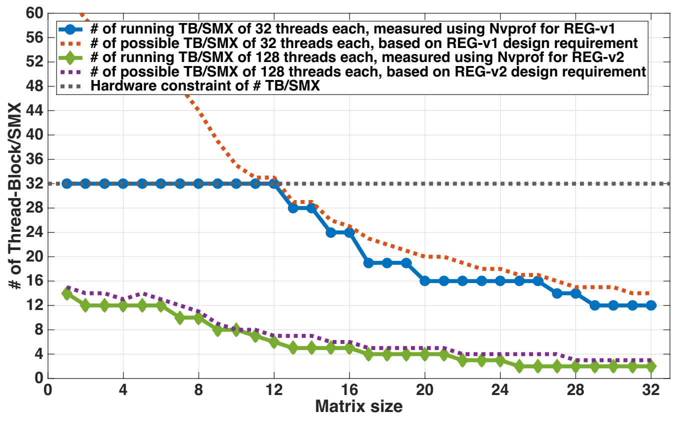


Fig. 6. Analysis of the number of TBs per SMX estimated. Real run for the REG-v1 design of the LU factorization on an NVIDIA P100.

optimization path in order to achieve performance close to the theoretical upper bound.

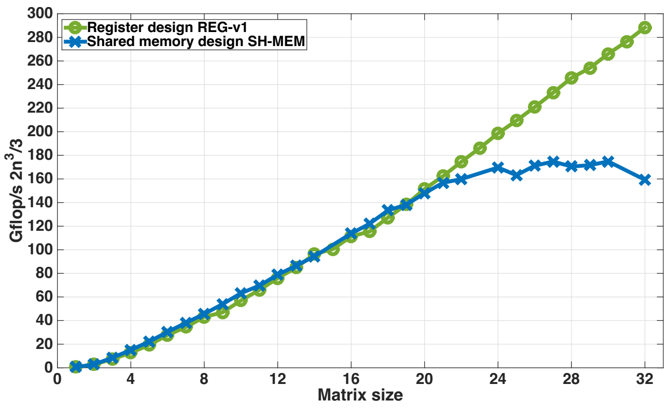


Fig. 7. Performance comparison of SH-MEM vs. REG-v1 for the LU factorization on an NVIDIA P100.

A further comparison of data from Figure 5 and Figure 6, reveals that the hardware constraint of the maximal number of TBs per SMX was reached for matrices smaller than 12×12. Following up, we focused on REG-v1, because it provided better performance. The hard constraint we found for the optimization was that we could not go beyond 32 TBs per SMX. However, for matrices smaller than 12×12, the TBs used less than 32 threads; consequently, they were using a sub-optimal amount of bandwidth. A higher bandwidth could be achieved by increasing the number of working threads. This was possible by revisiting our design and allocating a 2-D grid of threads, where each 1-D grid operated on an independent matrix. This workaround allowed us to reach the maximal ratio of TBs per SMX. Now, one TB would operate on $Dim.y$ matrices. For example, we could assign 128 threads (32×4) to a TB and make it operate on four independent matrices. This configuration increased the total number of registers per SMX ($4\times$ in this example).

Using data from Figure 4, we computed the number of TBs per SMX, as illustrated in Figure 6 (purple curve). The information depicted in Figures 1 and 6 has been decisive in evaluating the optimal configuration of 2-D threads for matrices smaller than 16×16. With this in mind, we can say confidently that the 32×4 threads configuration would help achieve even better performance. However, for matrices larger than 16×16, the new version of the register design (REG-v2) will be more likely to run less than 6 TBs per SMX, leading to an upper bandwidth of less than 10 TBs per SMX of 32 threads each; consequently, the performance will be lower when compared to REG-v1. This is not surprising since REG-v2 is designed specifically for matrices smaller than 16×16.

In Figure 8, we show the previous register design (REG-v1) and the latest register design (REG-v3), where we used the 2-D grid for matrices smaller than 16×16 and kept the REG-v1 design for matrices larger than 16×16. We also compared our implementation to the cuBLAS 8.0 batched LU solver. The first observation is that the results match the expectations of our analysis. The second point is related to the efficiency of our implementation since it outperforms the cuBLAS batched LU solver by about 3× on 32×32 matrices.

This paper does not aim to provide details on the advancement of the LU factorization algorithm. However, we would like to mention that it is possible to improve the performance by delaying the algorithm's swapping process, as reported in [1]. This optimization does not affect the number of registers or shared memory required and can improve performance by 10% (purple curve of Figure 8).

Finally, after all possible algorithm analysis and optimization, we found it reasonable to apply autotuning strategies. The autotuning experiments showed that an improvement of only about 5% could be obtained on top of our algorithm analysis.

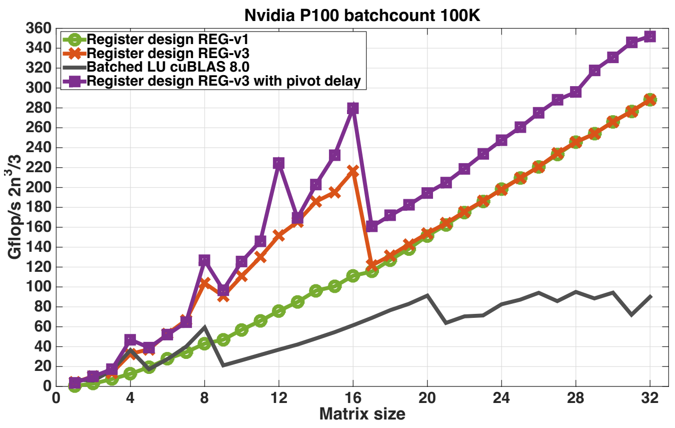


Fig. 8. Performance comparison of the two-register design of the LU factorization on an NVIDIA P100.

4.5 The Interleaved Data Layout

We also investigated alternatives to conventional data storage to assess possible improvements. Based on the analysis outlined in this work, we believe that there may be a little performance gain for matrices smaller than 20×20.

When the matrices are very small, it becomes more challenging to have a coalesced memory read, and as the dimensions of the matrices become smaller, it eventually becomes impossible to have any coalesced reads at all for matrices smaller than 16×16. The easiest and most basic way to solve this problem is to reorder the dimensions in an interleaved fashion (e.g., the first element of Matrix 1 will be followed by the first element of Matrix 2 and so on, eventually moving through all elements of every matrix). In this case, one warp reads 32 elements, with the same row and column index in 32 consecutive matrices. It is true that data is now 128-byte aligned, but this kind of storage will not allow for an efficient implementation on a GPU for matrices larger than 16×16 in double precision and for matrices larger than 32×32 in single precision. The reason being: 32 threads will be working on 32 different matrices, thereby making it impossible to hold data in shared memory or in the registers. On the P100 for example, this will limit the amount of shared memory available for the panel of the Cholesky factorization to 2 KB. This results in bad performance except for the sizes mentioned above, where the performance obtained from such a design was close to that obtained with the standard design. This kind of design might be interesting for a GEMV or TRSV type of operation, where the matrix is read only once. Recent studies on optimized batched BLAS kernels designed for multicore architectures have shown promising results over the classical approach of solving one problem per core at a time [16], [17].

5 Conclusion and Future Remarks

This paper presented a model and an analysis on how to design GPU kernels for very small matrix computations. We provided a detailed study of the optimization process, and we also demonstrated how a detailed performance analysis could help considerably reduce developer efforts and man hours required to design an efficient GPU kernel. The proposed work will also simplify the autotuning process, where—instead of generating, running, and analyzing tens of thousands of configurations—one can dramatically decrease this number to a small subset. We showed a Cholesky factorization and an LU factorization as case studies and showed how we were able to reach the theoretical peak performance of these kernels. The model is designed specifically for very small matrices, where the computation is memory-bound, and high performance can be achieved through a set of optimizations different from those used in the more standard large matrix computations. Future directions include studying other algorithms of interest to the scientific community and discovering more detailed optimization techniques in the area of deep learning, where little research has been conducted from this perspective.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grants No. CSR 1514286 and No. OAC 1740250, NVIDIA, and the Department of Energy.

References

[1] A. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra. Factorization and inversion of a million matrices using gpus: Challenges and countermeasures. *Procedia Computer Science*, 108:606 – 615, 2017. International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland.

[2] M. J. Abraham, T. Murtola, R. Schulz, S. Pll, J. C. Smith, B. Hess, and E. Lindahl. Gromacs: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. *SoftwareX*, 1-2(Supplement C):19 – 25, 2015.

[3] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, and S. Tomov. Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs. In W. mei W. Hwu, editor, *GPU Computing Gems*, volume 2. Morgan Kaufmann, Sept. 2010.

[4] ACML - AMD Core Math Library, 2014. Available at http://developer.amd.com/tools-and-sdks/cpu-development/amd-core-math-library-acml.

[5] M. Anderson, D. Sheffield, and K. Keutzer. A predictive model for solving small linear algebra problems in gpu registers. In *IEEE 26th International Parallel Distributed Processing Symposium (IPDPS)*, 2012.

[6] A. A. Auer, G. Baumgartner, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, X. Gao, R. Harrison, S. Krishnamoorthy, S. Krishnan, C.-C. Lam, Q. Luc, M. Nooijene, R. Pitzerf, J. Ramanujamg, P. Sadayappanc, and A. Sibiryakovc. Automatic code generation for many-body electronic structure methods: the tensor contraction engine. *Molecular Physics*, 104(2):211–228, 2006.

[7] B. Brock, A. Belt, J. J. Billings, and M. Guidry. Explicit Integration with GPU Acceleration for Large Kinetic Networks, Jan. 2015.

[8] J. H. Chen, A. Choudhary, B. de Supinski, M. DeVries, E. R. Hawkes, S. Klasky, W. K. Liao, K. L. Ma, J. Mellor-Crummey, N. Podhorszki, R. Sankaran, S. Shende, and C. S. Yoo. Terascale direct numerical simulations of turbulent combustion using S3D. *Comput. Sci. Disc.*, 2, 2009.

[9] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. cudnn: Efficient primitives for deep learning. *CoRR*, abs/1410.0759, 2014.

[10] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. cudnn: Efficient primitives for deep learning. *CoRR*, abs/1410.0759, 2014.

[11] T. Dong, V. Dobrev, T. Kolev, R. Rieben, S. Tomov, and J. Dongarra. A step towards energy efficient computing: Redesigning a hydrodynamic application on CPU-GPU. In *IEEE 28th International Parallel Distributed Processing Symposium (IPDPS)*, 2014.

[12] T. Dong, A. Haidar, P. Luszczek, A. Harris, S. Tomov, and J. Dongarra. LU Factorization of Small Matrices: Accelerating Batched DGETRF on the GPU. In *Proceedings of 16th IEEE International Conference on High Performance and Communications (HPCC 2014)*, August 2014.

[13] T. Dong, A. Haidar, S. Tomov, and J. Dongarra. A fast batched Cholesky factorization on a GPU. In *Proceedings of 2014 International Conference on Parallel Processing (ICPP-2014)*, Septembe 2014.

[14] J. Dongarra, I. Duff, M. Gates, A. Haidar, S. Hammarling, N. J. Higham, J. Hogg, P. Valero-Lara, S. D. Relton, S. Tomov, and M. Zounon. A proposed API for Batched Basic Linear Algebra Subprograms. MIMS EPrint 2016.25, Manchester Institute for Mathematical Sciences, The University of Manchester, UK, Apr. 2016.

[15] J. Dongarra, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, and A. YarKhan. Model-driven one-sided factorizations on multicore accelerated systems. *International Journal on Supercomputing Frontiers and Innovations*, 1(1), June 2014.

[16] J. Dongarra, S. Hammarling, N. J. Higham, S. D. Relton, P. Valero-Lara, and M. Zounon. The design and performance of batched blas on modern high-performance computing systems. *Procedia Computer Science*, 108:495–504, 2017.

[17] J. J. Dongarra, S. Hammarling, N. J. Higham, S. D. Relton, and M. Zounon. Optimized batched linear algebra for modern architectures. In *Euro-Par 2017: Parallel Processing - 23rd International Conference on Parallel and Distributed Computing, Santiago de Compostela, Spain, August 28 - September 1, 2017, Proceedings*, pages 511–522, 2017.

[18] M. W. Guidry, J. J. Billings, and W. R. Hix. Explicit integration of extremely stiff reaction networks: partial equilibrium methods. *Computational Science & Discovery*, 6(1):015003, 2013.

[19] A. Haidar, P. Luszczek, S. Tomov, and J. Dongarra. Towards batched linear solvers on accelerated hardware platforms. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, San Francisco, CA, 02/2015 2015. ACM, ACM.

[20] J. R. Hammond, S. Krishnamoorthy, S. Shende, N. A. Romero, and A. D. Malony. Performance characterization of global address space applications: a case study with nwchem. *Concurrency and Computation: Practice and Experience*, 24(2):135–154, 2012.

[21] Intel Pentium III Processor - Small Matrix Library, 1999. Available at http://www.intel.com/design/pentiumiii/sml/.

[22] Intel Math Kernel Library, 2014. Available at http://software.intel.com/intel-mkl/.

[23] H. Ltaief, S. Tomov, R. Nath, P. Du, and J. Dongarra. *A Scalable High Performant Cholesky Factorization for Multicore with GPU Accelerators*, pages 93–101. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[24] O. Messer, J. Harris, S. Parete-Koon, and M. Chertkow. Multicore and accelerator development for a leadership-class stellar astrophysics code. In *Proceedings of "PARA 2012: State-of-the-Art in Scientific and Parallel Computing."*, 2012.

[25] J. M. Molero, E. M. Garzn, I. Garca, E. S. Quintana-Ort, and A. Plaza. Efficient implementation of hyperspectral anomaly detection techniques on gpus and multicore processors. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 7(6):2256–2266, June 2014.

[26] Going beyond full utilization: The inside scoop on nervanas winograd kernels, 2016. Available at https://www.nervanasys.com/winograd-2/.

[27] A. M. N. Niklasson, S. M. Mniszewski, C. F. A. Negre, M. J. Cawkwell, P. J. Swart, J. Mohd-Yusof, T. C. Germann, M. E. Wall, N. Bock, E. H. Rubensson, and H. Djidjev. Graph-based linear scaling electronic structure theory. *The Journal of Chemical Physics*, 144(23):234101, 2016.

[28] CUBLAS 6.5, Jan. 2015. Available at http://docs.nvidia.com/cuda/cublas/.

[29] V. Oreste, M. Fatica, N. A. Gawande, and A. Tumeo. Power/performance trade-offs of small batched LU based solvers on GPUs. In *19th International Conference on Parallel Processing, Euro-Par 2013*, volume 8097 of *Lecture Notes in Computer Science*, pages 813–825, Aachen, Germany, August 26-30 2013.

[30] V. Oreste, N. A. Gawande, and A. Tumeo. Accelerating subsurface transport simulation on heterogeneous clusters. In *IEEE International Conference on Cluster Computing (CLUSTER 2013)*, Indianapolis, Indiana, September, 23-27 2013.

[31] S. Páll, M. J. Abraham, C. Kutzner, B. Hess, and E. Lindahl. Tackling exascale software challenges in molecular dynamics simulations with GROMACS. *CoRR*, abs/1506.00716, 2015.

[32] E. E. Papalexakis, C. Faloutsos, and N. D. Sidiropoulos. Tensors for data mining and data fusion: Models, applications, and scalable algorithms. *ACM Trans. Intell. Syst. Technol.*, 8(2):16:1–16:44, Oct. 2016.

[33] W. Raphael and Friedrich-Karl. Silicon burning II: Quasi-equilibrium and explosive burning. *ApJ*, 511:862–875, February 1999.

[34] M. J. C. S. M. Mniszewski, C. F. A. Negre and A. M. N. Niklasson. Distributed graph-based density. matrix calculation for quantum. molecular dynamics using gpus, 2016.

[35] H. Shan, S. Williams, W. de Jong, and L. Oliker. Thread-level parallelization and optimization of nwchem for the intel mic architecture. In *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '15, pages 58–67, New York, NY, USA, 2015. ACM.

[36] J. C. Sweet, R. J. Nowling, T. Cickovski, C. R. Sweet, V. S. Pande, and J. A. Izaguirre. Long timestep molecular dynamics on the graphical processing unit. *Journal of Chemical Theory and Computation*, 9(8):3267–3281, 2013.

[37] S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid gpu accelerated manycore systems. *Parellel Comput. Syst. Appl.*, 36(5-6):232–240, 2010.

[38] S. Tomov, R. Nath, and J. Dongarra. Dense linear algebra solvers for multicore with GPU accelerators. In *Proc. of the IEEE IPDPS'10*, Atlanta, GA, April 19-23 2014.

[39] V. Volkov and J. W. Demmel. LU, QR and Cholesky factorizations using vector capabilities of GPUs. Technical Report UCB/EECS-2008-49, University of California, Berkeley, May 13 2008. Also available as LAPACK Working Note 202.

[40] I. Wainwright. Optimized LU-decomposition with full pivot for small batched matrices, April, 2013. GTC'13 – ID S3069.

[41] S. Williams, A. Waterman, and D. Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, Apr. 2009.

[42] S. N. Yeralan, T. A. Davis, and S. Ranka. Sparse mulitfrontal QR on the GPU. Technical report, University of Florida Technical Report, 2013.

Azzam Haidar holds a research scientist position at the Innovative Computing Laboratory at the University of Tennessee. His research revolves around Parallel Linear Algebra for Scalable Distributed Heterogeneous Architectures such as multicore CPUs and accelerators (Intel Xeon-Phi, NVIDIA and AMD GPUs). His goal is to create software that simplifies development of applications that achieve high-performance and portability. Such programming models rely on asynchronous and out-of-order scheduling of operations. These concepts are the basis for scalable and efficient software for Computational Linear Algebra and applications. Another research interest is the development/implementation of numerical algorithms and software for large scale parallel sparse problems in order to develop hybrid approaches combining direct and iterative algorithms to solve systems of linear algebraic equations with large sparse matrices. Contact him at haidar@icl.utk.edu.

Ahmad Abdelfattah received his PhD in computer science from King Abdullah University of Science and Technology (KAUST) in 2015, where he was a member of the Extreme Computing Research Center (ECRC). He is currently a research scientist in the Innovative Computing Laboratory at the University of Tennessee. He works on optimization techniques for many dense linear algebra algorithms at different scales. Ahmad has B.Sc. and M.Sc. degrees in computer engineering from Ain Shams University, Egypt. Contact him at ahmad@icl.utk.edu.

Mawussi Zounon is a Research Associate in the Numerical Linear Algebra group at the University of Manchester. He received a PhD in computer science and applied mathematics from the University of Bordeaux for his contribution to numerical fault tolerant strategies for large sparse linear algebra solvers with a special focus on Krylov subspace methods. His research interests are in parallel algorithms, numerical algorithms in linear algebra, computer architures, and fault tolerance. Contact him at mawussi.zounon@manchester.ac.uk.

Stanimire Tomov received a M.S. degree in Computer Science from Sofia University, Bulgaria, and Ph.D. in Mathematics from Texas A&M University. He is a Research Director in ICL and Research Assistant Professor in the EECS at UTK. Tomov's research interests are in parallel algorithms, numerical analysis, and high-performance scientific computing (HPC). Currently, his work is concentrated on the development of numerical linear algebra software, and in particular MAGMA, for emerging architectures for HPC. Contact him at tomov@icl.utk.edu.

Jack Dongarra holds an appointment at the University of Tennessee, Oak Ridge National Laboratory, and the University of Manchester. He specializes in numerical algorithms in linear algebra, parallel computing, use of advanced-computer architectures, programming methodology, and tools for parallel computers. He was awarded the IEEE Sid Fernbach Award in 2004; 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 Charles Babbage Award; and in 2013 he received the ACM/IEEE Ken Kennedy Award. He is a Fellow of the AAAS, ACM, IEEE, and SIAM and a foreign member of the Russian Academy of Science and a member of the US National Academy of Engineering.