Redesigning Triangular Dense Matrix Computations on GPUs

Ali Charara, Hatem Ltaief, and David Keyes

Extreme Computing Research Center,
King Abdullah University of Science and Technology,
Thuwal, Jeddah 23955, Saudi Arabia
[Ali.Charara,Hatem.Ltaief,David.Keyes]@kaust.edu.sa

Abstract. A new implementation of the triangular matrix-matrix multiplication (TRMM) and the triangular solve (TRSM) kernels are described on GPU hardware accelerators. Although part of the Level 3 BLAS family, these highly computationally intensive kernels fail to achieve the percentage of the theoretical peak performance on GPUs that one would expect when running kernels with similar surface-to-volume ratio on hardware accelerators, i.e., the standard matrix-matrix multiplication (GEMM). The authors propose adopting a recursive formulation, which enriches the TRMM and TRSM inner structures with GEMM calls and, therefore, reduces memory traffic while increasing the level of concurrency. The new implementation enables efficient use of the GPU memory hierarchy and mitigates the latency overhead, to run at the speed of the higher cache levels. Performance comparisons show up to eightfold and twofold speedups for large dense matrix sizes, against the existing state-of-the-art TRMM and TRSM implementations from NVIDIA cuBLAS, respectively, across various GPU generations. Once integrated into high-level Cholesky-based dense linear algebra algorithms, the performance impact on the overall applications demonstrates up to fourfold and twofold speedups, against the equivalent native implementations, linked with cuBLAS TRMM and TRSM kernels, respectively. The new TRMM/TRSM kernel implementations are part of the open-source KBLAS software library[1] and are lined up for integration into the NVIDIA cuBLAS library in the upcoming v8.0 release.

Keywords: Triangular dense matrix computations; high performance computing; recursive formulation; KBLAS; GPU optimization.

1 Introduction

Most large-scale numerical simulations rely for high performance on the BLAS [13], which are often available through various vendor distributions tuned for their own architecture, e.g., MKL on Intel x86 [3], ACML [5] on AMD x86, ESSL [2] on IBM Power architecture. Indeed, BLAS kernels are considered as building blocks, and the library represents one of the last layers of the usual software stack, which is usually where application performance is extracted from the underlying hardware. This drives

[1] http://ecrc.kaust.edu.sa/Pages/Res-kblas.aspx

continuous efforts to optimize BLAS kernels [8]. While performance of Level 1 and 2 BLAS kernels is mainly limited by the bus bandwidth (memory-bound), Level 3 BLAS kernels display a higher flop/byte ratio (compute-bound), thanks to high data reuse occurring at the upper levels of the cache hierarchy. However, BLAS operations on triangular matrix structure, i.e., the triangular matrix-matrix multiplication (TRMM) and the system of linear equations triangular solvers (TRSM), have demonstrated limited performance on GPUs using the highly optimized NVIDIA cuBLAS library [5]. Although in the category of Level 3 BLAS operations, the triangular structure of the input matrix and the in-place nature of the operation may generate many Write After Read (WAR) data hazards. WAR situations usually reduce the level of concurrency due to inherent dependencies, and incur excessive data fetching from memory.

We describe a recursive formulation that enriches the TRMM/TRSM inner structures with GEMM calls and, therefore, enhances data reuse and concurrency on the GPU by minimizing the impact of WAR data hazards and by mitigating the memory transactions load overhead. The idea of casting level 3 BLAS operations into GEMM operations has been promoted by several previous works, including Kågström et. al [16], Goto et. al [12], Andersen et. al [9], and Elmorth et. al [11]. In this paper, we describe a recursive formulation of the TRMM and TRSM kernels, which suits well the aggressively parallel many-core GPUs architecture. Performance comparisons show up to sixfold and twofold speedups for large dense matrix sizes with our implementation, against the existing state-of-the-art TRMM/TRSM implementations from NVIDIA cuBLAS, respectively, across various GPU generations. After integrating it into high-level dense linear algebra algorithms, such as the Cholesky-based symmetric matrix inversion and the Cholesky-based triangular solvers, the performance impact on the overall application demonstrates up to fourfold and twofold speedups against the equivalent native implementations, linked with NVIDIA cuBLAS TRMM/TRSM kernels.

The remainder of the paper is organized as follows. Section 2 presents related work. Section 3 recalls the TRMM/TRSM kernel operations and identifies the performance bottlenecks seen on NVIDIA GPUs. The implementation details of the high-performance recursive TRMM/TRSM kernels are given in Section 4. Section 5 shows TRMM/TRSM performance results on various GPU generations, compares against the state-of-the-art high-performance NVIDIA cuBLAS implementations, and shows the impact of integrating our TRMM/TRSM implementations into the Cholesky-based symmetric matrix inversion and the Cholesky-based triangular solver from MAGMA [7], a high performance dense linear algebra library on GPUs. We conclude in Section 6.

2 Related Work

The literature is rich when it comes to BLAS kernel optimizations targeting different x86 and vector architectures [19, 17, 18, 8]. We focus only on a few, which are directly related to the topic of the paper.

Kågström et al. [16] proposed a method to express Level 3 BLAS operations in terms of general matrix-matrix multiplication kernel (GEMM). The core idea is to cast the bulk of computations involving off-diagonal blocks as GEMMs, and to operate on the diagonal blocks with Level 1 and 2 BLAS kernels, which increases the byte/flops

ratio of the original Level 1 and 2 BLAS operations. Furthermore, recursive formulations for several LA kernels have been promoted by Andersen et. al [9] with packed and general data formats, and by Elmorth et. al [11] with recursive blocking. However, their formulations are designed for the hierarchical memory architecture of x86 multicore processors. Goto et al. [12] proposed another approach, which casts the computations of Level 3 BLAS operations in terms of a General Panel-Panel multiplication (GEPP). For operations that involve triangular matrices like TRMM and TRSM. The authors customized the GEPP kernel by adjusting the involved loop bounds or by zeroing out the unused elements above the diagonal (for lower triangular TRMM). They show that, with such minimal changes to the main GEPP kernel, this approach enhances the performance of Level 3 BLAS operations against open-source and commercial libraries (MKL, ESSL and ATLAS) on various CPU architectures. Later, Igual et al. [15] extended the same approach to GPUs and accelerated the corresponding Level-3 BLAS operations.

Though these methods are effective for multicore CPU processors, especially when the employed block sizes fits well in the L1 and L2 cache sizes, GPU architecture imposes fundamental changes to the programming model to extract better performance. In general, because the GPU is a throughput-oriented device, Level 3 BLAS kernels perform better when processing larger matrix blocks, which also help mitigate the overhead of extra kernel launches. In particular, GEMM kernel reaches the sustained peak performance with matrix blocks higher than 1024 as shown in Figure 1(a), an important fact that we will use to explain our method in Section 4 to speed up the execution of both TRMM and TRSM operations.

Both Kågström's and Goto's methods require additional temporary buffers to store temporary data, whose size needs to be tuned for better cache alignment. The later method adds the overhead of packing data of each panel before each GEPP operation and unpacking it after GEPP is over. This incurs extra memory allocations and data transfer costs that would prevent an efficient GPU port. Moreover, Kågström's method requires a set of sequentially invoked GEMV kernels to achieve a TRMM for small diagonal blocks, which may be prohibitive on GPUs, due to the overhead of kernel launches. We study the recursive formulations of these kernels in the context of massively parallel GPU devices.

3 The Triangular Matrix Operations: TRMM and TRSM

This Section recalls the general TRMM and TRSM operations and highlights the reasons behind the performance bottleneck observed on GPUs.

3.1 Recalling TRMM and TRSM Operations

As described in the legacy BLAS library [1], TRMM performs one of the triangular matrix-matrix operations as follows: $B = \alpha\ op(A)\ B$, if side is left, or $B = \alpha\ B\ op(A)$, if side is right. On the other hand, TRSM solves one of the matrix equations as follows: $op(A)\ X = \alpha\ B$, if side is left, or $X\ op(A) = \alpha\ B$, if side is right. For both formulas, α is a scalar, B and X are $M \times N$ matrices, A is a unit, or non-unit, upper or lower

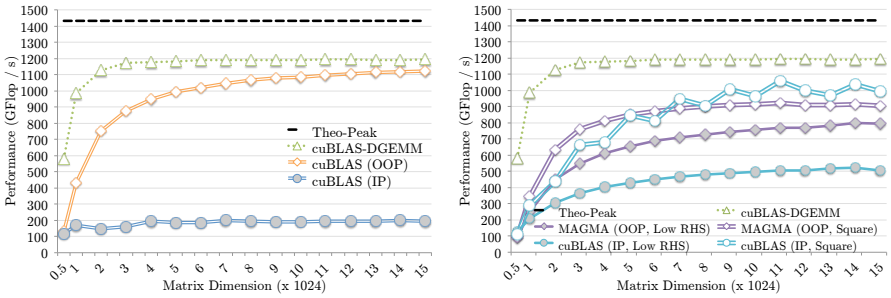
triangular matrix and $op(A)$ is one of $op(A) = A$ or $op(A) = A^T$. The resulting matrix X is overwritten on B. It is noteworthy to mention that these operations happen *in-place* (IP), i.e., B gets overwritten by the final output of the TRMM or TRSM operations. This in-place overwriting may engender lots of anti-dependencies or WAR data hazards, from successive memory accesses, due to the triangular structure of the input matrix, and may generate lots of memory traffic.

3.2 Current State of Art Performance of TRMM and TRSM

The NVIDIA cuBLAS library currently provides two APIs for TRMM, in-place (IP) and out-of-place (OOP) and a single IP API for TRSM. Figure 1(a) shows the performance of NVIDIA cuBLAS (v7.5) DTRMM in-place (IP) against out-of-place (OOP) using a Kepler K40 GPU, in double precision arithmetic. The theoretical peak performance of the card and the DGEMM sustained peak are given as upper-bound references. The OOP DTRMM runs close to DGEMM performance for asymptotic sizes. However, the IP DTRMM achieves only a small percentage of DGEMM peak and one can notice a factor of six between IP and OOP DTRMM performances. The OOP TRMM removes the WAR data dependencies on B and can be implemented in an embarrassingly parallel fashion, similar to GEMM, at the expense of increasing by a factor of two the memory footprint, besides violating the legacy BLAS TRMM API. On the other hand, MAGMA provides an OOP TRSM implementation, in which the diagonal blocks of matrix A are inverted, followed by a set of GEMM calls. Figure 1(b) shows the performance of NVIDIA cuBLAS (v7.5) IP DTRSM for a low number of right-hand sides (RHS), i.e. 512, as well as a number of RHS equal to the matrix size (square problem), in comparison to the OOP implementation from MAGMA. cuBLAS DTRSM for square problems runs close to DGEMM peak, although it highlights a jaggy performance behavior. However, for low RHS, cuBLAS DTRSM looses computational intensity and seems to further suffer from lack of parallelism. MAGMA DTRSM presents more stable performance for square matrices and much better performance for low RHS cases, at the cost of doubling the memory footprint and thus excessive data transfer.

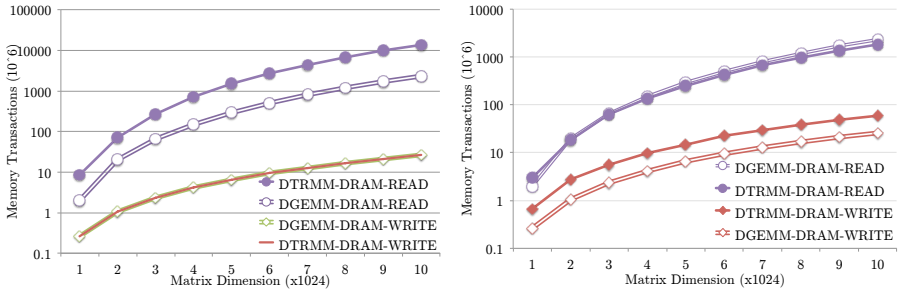
3.3 Identifying the Performance Bottlenecks

We can observe limited concurrency for IP TRSM, since updating right columns of matrix B (for a right sided TRSM) needs to read updated left columns of the same matrix, causing a multitude of RAW dependencies. Other variants of TRSM exhibit similar RAW dependencies. The IP TRMM has also limited concurrency due to the fact that updating the left columns of matrix B (for a right sided TRMM) need to read initial values from subsequent right columns of matrix B before the right columns get updated, causing a multitude of WAR dependencies. Other variants of TRMM exhibit similar WAR dependencies. On the other hand, the OOP TRMM/TRSM incurs additional overheads, due to data transfers through the slow PCIe link, as well as extra memory allocation, which would be prohibitive for large matrix sizes, especially since memory is a scarce resource on GPUs.



(a) **DTRMM for square matrices.** (b) **DTRSM for square and low RHS (512) matrices.**

Fig. 1: Performance comparisons of cuBLAS (v7.5) and MAGMA (v2.0.1) in-place (IP) against out-of-place (OOP) of TRMM and TRSM using an NVIDIA Kepler K40 GPU, in double precision arithmetic.



(a) **IP DTRMM.** (b) **OOP DTRMM.**

Fig. 2: Profiling of memory transactions for cuBLAS IP and OOP DTRMM against that of DGEMM.

3.4 Profiling of NVIDIA cuBLAS TRMM

Assuming square matrices, GEMM performs $2N^3$ floating-point operations (flops) on $3N^2$ data, and TRMM performs N^3 flops on $3/2N^2$ data. In fact, TRMM can be ideally thought of as an IP GEMM; thus, the memory transactions involved are expected to be proportional to the processed data size. However, Figure 2(a) highlights that cuBLAS IP TRMM implementation performs almost an order of magnitude of DRAM read memory accesses higher than a GEMM for the same input size, and an equivalent number of DRAM memory writes as GEMM for the same input size. On the other hand, OOP TRMM implementation exhibits much better memory traffic load, with almost equivalent DRAM read memory accesses to GEMM for the same input size, but astonishingly more DRAM write accesses as shown in Figure 2(b). This hints that the cuBLAS implementation of TRMM is not optimal and produces excessive memory accesses, due to its inability to efficiently overcome the WAR dependencies. We note that profiling of cuBLAS and KBLAS memory transactions has been done using the *nvprof* tool available from NVIDIA CUDA Toolkit [4].

In this paper, we propose to further improve the performance of IP TRSM and IP TRMM on NVIDIA GPUs, by running closer to GEMM peak, in addition to staying compliant with the legacy BLAS regarding operating in-place. We refer to IP TRMM and IP TRSM as TRMM and TRSM onward in this paper.

4 Recursive Definition and Implementation Details

In this section, we describe the recursive formulation of the TRMM and TRSM operations, as illustrated in Figure 3, and their implementation over NVIDIA GPUs. In the following definition, we illustrate the Left Lower Transpose TRMM, and the Left Lower NonTranspose TRSM cases, where $TRMM \rightarrow B = \alpha\ A^T\ B$ and $TRSM \rightarrow A\ X = \alpha\ B$, B is of size $M \times N$ and A of size $N \times N$. All other variants are supported and operate in a similar manner. As illustrated in Figure 3, we first choose a suitable partition of the number of B rows $M = M_1 + M_2$ as discussed in the next paragraph. We then partition the triangular matrix A into three sub-matrices: two triangular matrices A_1 and A_3 (i.e. two diagonal blocks) of sizes $M_1 \times M_1$ and $M_2 \times M_2$, respectively, and a third rectangular non-diagonal block A_2 of size $M_2 \times M_1$. We correspondingly partition the rectangular matrix B into two rectangular matrices B_1 and B_2 of sizes $M_1 \times N$ and $M_2 \times N$, respectively (such partitioning would be instead along the columns $N = N_1 + N_2$ for right sided TRMM/TRSM, where A is of size $N \times N$). We then re-write these operations as follows:

$$TRMM: \begin{cases} B_1 = \alpha\ A_1^T\ B_1 & \text{recursive TRMM} \\ B_1 = \alpha\ A_2^T\ B_2 + B_1 & \text{GEMM} \\ B_2 = \alpha\ A_3^T\ B_2 & \text{recursive TRMM} \end{cases}$$

$$TRSM: \begin{cases} A_1\ X_1 = \alpha\ B_1 & \text{recursive TRSM: Solve for } X_1 \text{ over } B_1 \\ B_2 = \alpha\ B_2 - A_2\ B_1 & \text{GEMM} \\ A_3\ X_2 = B_2 & \text{recursive TRSM: Solve for } X_2 \text{ over } B_2 \end{cases}$$

Recursion stops when reaching the size where cuBLAS TRMM/TRSM exhibits good performance in comparison to the performance of further splitting, in order to minimize the extra overhead of continuing the recursion on small sub-matrix sizes. At this level we call again the cuBLAS TRMM/TRSM routine, respectively, which launches one kernel with enough thread blocks to process the small sub-matrix in parallel. The stopping size is left as a tuning parameter. In our tests, we used 128 as the stopping size, a value that was effective across various GPU devices.

To maximize the performance of the generated GEMM calls, we need to be careful about the choice of partitioning the rows. Our target is to generate GEMM calls with maximum possible sub-matrix size, which in turn makes the best boost in performance. We choose the partitioning $M_1 = M/2$ if M is a power of 2, otherwise we choose M_1 as the closest power of 2 strictly less than M; in both cases $M_2 = M - M_1$. The same logic applies when partitioning along columns for right sided TRMM or TRSM. Such a strategy for partitioning the matrices is also needed to ensure that the generated GEMM calls operate on suitable matrix sizes and to minimize the number of trailing sub-matrices with odd matrix sizes during the subsequent recursion.

By casting the TRMM and TRSM operations into a set of large GEMMs and a set of small TRMMs and TRSMs, respectively, we benefit not only from the GEMM speed but also from its optimized memory access pattern that fits the GPU memory hierarchy, and thus removing most of the redundant memory accesses observed in cuBLAS TRMM implementation, for instance. The next section details such performance gains.

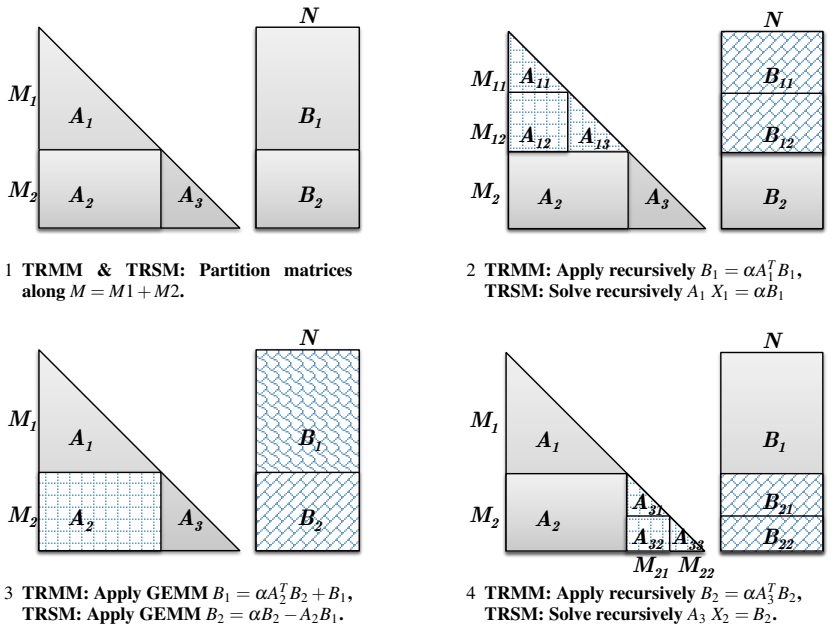


Fig. 3: Illustrating a Left-Lower-Transpose recursive TRMM, and Left-Lower-NonTranspose recursive TRSM, partitioning along the rows. Steps are to be applied at each recursive call in the order depicted in the picture.

5 Experimental Results

This section features the performance results of our recursive KBLAS TRMM and TRSM on GPUs and compares it against state-of-the-art implementation from NVIDIA cuBLAS and MAGMA. The experiments have been conducted on various NVIDIA GPUs generations: Fermi, K20, K40, GTX Titan Black, and the latest Maxwell GPU card. Since the Titan and Maxwell are gaming cards, ECC memory correction has been turned off on the other GPUs for consistency. The latest CUDA Toolkit v7.5 has been used. The four precisions for TRMM and TRSM as well as their eight variants are supported in our KBLAS software distribution. It is important to recall the profiling graphs from Section 3.4, in which the performance bottleneck of the IP cuBLAS TRMM has been identified, i.e., the excessive amount of memory accesses.

Figure 4 shows how KBLAS TRMM is capable of fetching less data from global memory in favor of reusing already fetched data in the higher cache levels, as opposed to IP cuBLAS TRMM. KBLAS TRMM also performs less data traffic compared to cuBLAS DGEMM, however, this is commensurate to the processed data size. In addition, it performs less data fetching from global memory than the more regular cuBLAS OOP TRMM. Note that the increase in global memory data writes in KBLAS TRMM is due to the fact that some horizontal panels are updated several times by the recursive nature of the algorithm. The number of updates of a horizontal panel is equal to the number of GEMMs that updates in the left path to the recursion root, plus one for the TRMM leaf. However this increase in the number of global memory writes is much less expensive than the significantly greater number of global memory reads needed by cuBLAS IP TRMM.

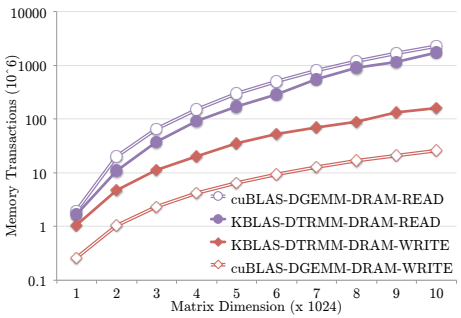
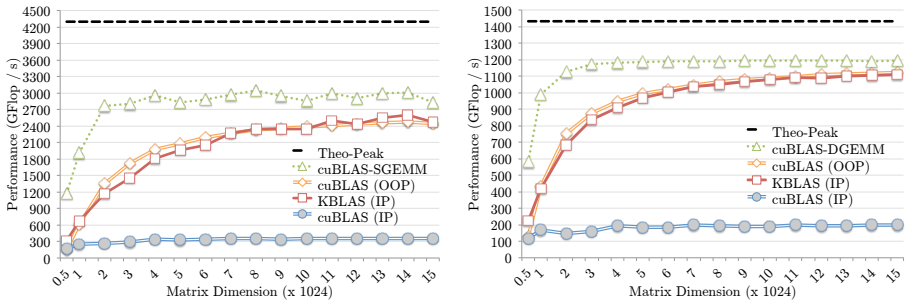


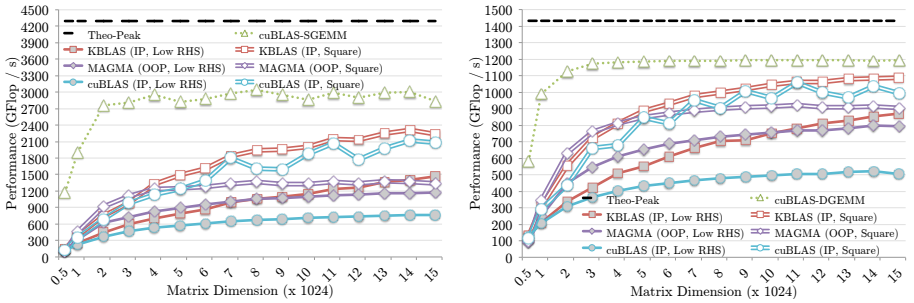
Fig. 4: Profiling of memory transactions for KBLAS DTRMM against that of DGEMM.

Figures 5(a) and 5(b) show the performance comparisons of KBLAS TRMM against that of IP and OOP cuBLAS TRMM for single and double precision arithmetics, respectively. KBLAS TRMM achieves similar performance as OOP cuBLAS TRMM, while maintaining the same memory footprint of IP cuBLAS TRMM. For large matrix sizes in double precision, KBLAS DTRMM achieves almost one Tflop/s, six times more than the IP cuBLAS DTRMM from NVIDIA. KBLAS DTRMM attains almost 90% performance of DGEMM, thanks to the recursive formulation, which inherently enriches TRMM with GEMM calls. Furthermore, Figures 6(a) and 6(b) show performance speedup of KBLAS IP TRSM against cuBLAS IP TRSM and MAGMA OOP TRSM for single and double precision arithmetics, respectively, on an NVIDIA K40 GPU. It



(a) **Single precision.** (b) **Double precision.**

Fig. 5: Performance comparisons of KBLAS TRMM against that of IP and OOP cuBLAS TRMM running on NVIDIA K40 GPU.



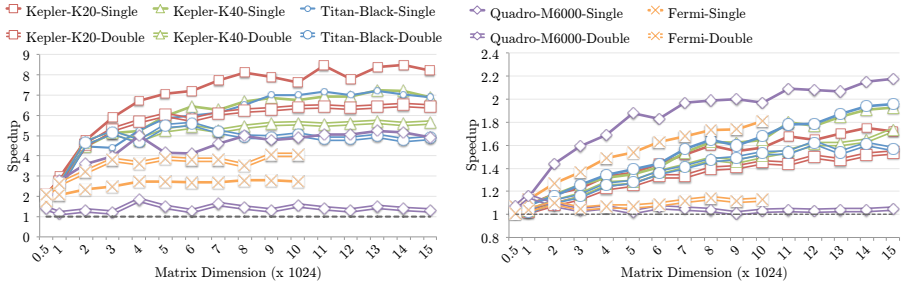
(a) **Single precision.** (b) **Double precision.**

Fig. 6: Performance comparisons of KBLAS IP TRSM against that of cuBLAS IP TRSM and MAGMA OOP TRSM running on NVIDIA K40 GPU, with square and low RHS matrices.

is noteworthy to mention that although the performance gain with square matrices is not significant, and in fact smooths out the irregular performance from cuBLAS, the performance gain with triangular solves with small number of RHS is important. In practice, it is more common to encounter a triangular solve with only a few RHS. Note also that KBLAS IP TRSM achieves similar (sometimes better) performance as MAGMA OOP TRSM without the overhead of excessive data transfer and extra memory allocations.

Figures 7(a) and 7(b) show performance speedup of KBLAS TRMM and TRSM against cuBLAS TRMM and TRSM, respectively, on various generations of NVIDIA GPUs. The speedup ranges shown proves the performance portability of the implementation across different NVIDIA GPUs. We note that the double precision capability ratio in reference to single precision capability of the Quadro-M6000 GPU (a Maxwell architecture) has been intentionally fixed by NVIDIA to 1:32, hence its limited double precision computation performance. Although Titan Black GPU is also a graphics card, the user can still control its GPU's double precision performance by switching between 1:3 and 1:24 ratios, which explains its ability to ramp up its double precision performance, as opposed to the Quadro-M6000 GPU.

In scientific computations, often data resides on the host memory. Data is shipped to device memory on demand, and as a best practice, one repeatedly operates on the data while it is in the device memory until operations are done or the result is needed back on the CPU memory [6]. It is becoming a common practice within scientific libraries to provide an API that handles the data transfer between the host and the device implicitly and to operate on it in one function call. This practice simplifies the API and puts less burden on programmers who want to replace a CPU call by an equivalent GPU function call, without explicitly handling data transfer. The only way to perform such API with the existing cuBLAS routines is to wrap it within a function call that will initiate the data transfer, wait for it to arrive on device memory, operate on it, then initiate its transfer back to host memory. This results in a severe synchronous communication and computation scheme. In this context, our implementation brings an added value, in that due to the recursive nature of the routine calls, we can overlap communication with computation, i.e., by operating on parts of the data while waiting for the other parts to arrive

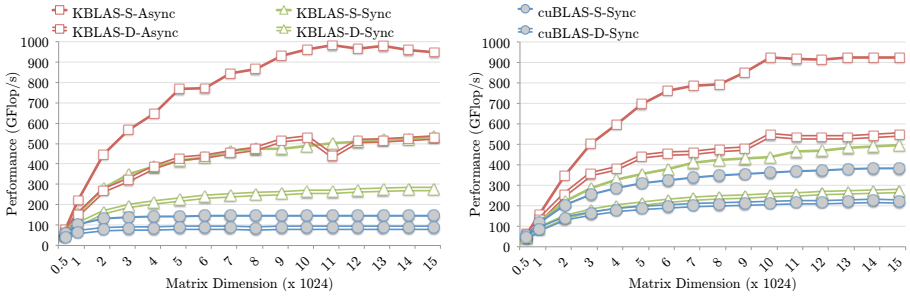


(a) **TRMM.** (b) **TRSM with small RHS.**

Fig. 7: Performance speedup of KBLAS TRMM and TRSM against cuBLAS TRMM and TRSM, respectively, on various generations of NVIDIA GPUs.

into device memory and vice versa. We provide an asynchronous API that achieves this goal. Figures 8(b) and 8(a) show the performance gain of such asynchronous TRMM and TRSM API's in comparison to the synchronous cuBLAS and KBLAS API's.

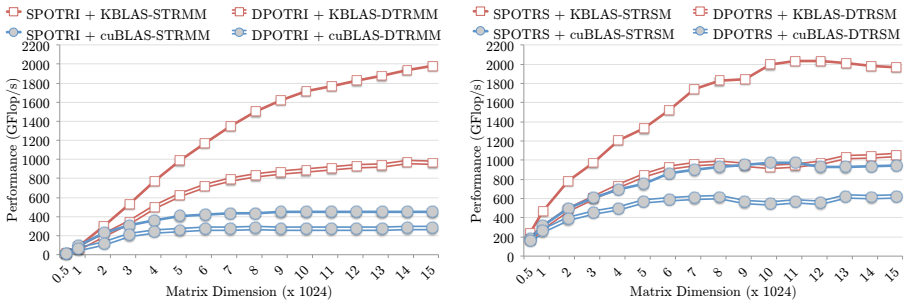
To conclude, we show the impact of KBLAS TRMM and TRSM when integrated into



(a) **TRMM.** (b) **TRSM.**

Fig. 8: Performance impact of asynchronous APIs for TRMM and TRSM.

high-level dense linear algebra algorithms. In particular, the Cholesky-based symmetric matrix inversion is one of the main operations for the computation of the variance-covariance matrix in statistics [14]. The inversion algorithm first factorizes the dense symmetric matrix using the Cholesky factorization (POTRF) and then inverts and multiply *in-place* the triangular Cholesky factor by its transpose (POTRI) to get the final inverted triangular matrix. Figure 9 shows the performance impact of linking the MAGMA [7] library with the KBLAS library. With KBLAS TRMM, MAGMA POTRI gets up to twofold and fourfold speedup, for single and double precision, respectively, when compared to using cuBLAS TRMM on a single GPU, as seen in Figure 9(a). Moreover, the symmetric matrix inversion operation drives a computational astronomy



(a) **Performance speedup of matrix inversion in MAGMA library (POTRI) using KBLAS TRMM vs using cuBLAS TRMM.**

(b) **Performance speedup of POTRS with low RHS (512) in MAGMA library using KBLAS TRSM vs. using cuBLAS TRSM.**

Fig. 9: Sample impact of using KBLAS TRMM and TRSM in MAGMA library.

application [10], which designs the future instruments of the European Extremely Large Telescope (E-ELT). This simulation has a strong constraint on close to real-time computations and, therefore, can highly benefit from the efficient KBLAS TRMM and TRSM introduced in this paper, when running on NVIDIA GPUs.

Finally, solving a system of linear equations with a positive definite matrix (POSV) is a very common operation in scientific computations. It can be achieved by first factorizing the matrix A with Cholesky method (POTRF), then solving with POTRS. Figure 9(b) shows the impact of linking the MAGMA library with the KBLAS library. With KBLAS TRSM, MAGMA POTRS gets up to twofold and 30% speedup, for single and double precision, respectively, compared to linking with cuBLAS TRSM.

6 Conclusions and Future Work

Recursive formulations of the Level 3 BLAS triangular matrix-matrix multiplication (TRMM) and triangular solve (TRSM) have been implemented on GPUs, which allow reducing memory traffic from global memory and expressing most of the computations in GEMM sequences. They achieve up to eightfold and twofold speedups, respectively, for asymptotic dense matrix sizes against the existing state-of-the-art TRMM/TRSM implementations from NVIDIA cuBLAS v7.5, across various GPU generations. After integrating them into high-level Cholesky-based dense linear algebra algorithms, performance impact on the overall application demonstrates up to fourfold and twofold speedups against the equivalent vendor implementations, respectively. The new TRMM and TRSM implementations on NVIDIA GPUs are available in the open-source KAUST BLAS (KBLAS) library [8] and are scheduled for integration into the NVIDIA cuBLAS library in its future release. Future work includes the performance analysis on tall and skinny matrices, which is critical for some dense linear algebra algorithms (e.g., the generalized symmetric eigenvalue problem), as well as looking at multi-GPU support. The authors would like also to study the general applicability of such recursive formulations on other Level 3 BLAS operations as well as targeting other hardware architectures (e.g., Intel/AMD x86, Intel Xeon Phi, AMD APUs, ARM processors, etc.).

Acknowledgments. We thank NVIDIA for hardware donations in the context of the GPU Research Center Award to the Extreme Computing Research Center at the King Abdullah University of Science and Technology and KAUST IT Research Computing for hardware support on the GPU-based system.

References

1. BLAS: Basic Linear Algebra Subprograms. Available at http://www.netlib.org/blas.
2. Engineering and Scientific Subroutine Library (ESSL) and Parallel ESSL. Available at http://www-03.ibm.com/systems/power/software/essl/.
3. Intel MKL Library. Available at http://software.intel.com/en-us/articles/intel-mkl.
4. NVIDIA CUDA Toolkit. http://developer.nvidia.com/cuda-toolkit.
5. The NVIDIA CUDA Basic Linear Algebra Subroutines (CUBLAS). Available at http://developer.nvidia.com/cublas.
6. CUDA C best practices guide.
7. 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. *J. Phys.: Conf. Ser.*, 180(1), 2009.
8. A. Ahmad, H. Ltaief, and D. Keyes. KBLAS: An Optimized Library for Dense Matrix-Vector Multiplication on GPU Accelerators. *ACM Trans. Math. Softw.*, 2016. To appear.
9. B. S. Andersen, F. G. Gustavson, A. Karaivanov, M. Marinova, J. Waniewski, and P. Y. Yalamov. Lawra: Linear algebra with recursive algorithms. In *Proc. of 5th Int. Work. App. Par. Comp., New Paradigms for HPC in Industry and Academia*, PARA '00, pages 38–51, London, UK, 2001. Springer-Verlag.
10. A. Charara, H. Ltaief, D. Gratadour, D. E. Keyes, A. Sevin, A. Abdelfattah, E. Gendron, C. Morel, and F. Vidal. Pipelining Computational Stages of the Tomographic Reconstructor for Multi-Object Adaptive Optics on a Multi-GPU System. In Trish Damkroger and Jack Dongarra, editors, *Int. Conf. High Perf. Comp., Netw., Stor. and Anal., SC 2014*, pages 262–273. IEEE, 2014.
11. E. Elmroth, F. Gustavson, I. Jonsson, and B. Kågström. Recursive blocked algorithms and hybrid data structures for dense matrix library software. *SIAM Review*, 46(1):3–45, 2004.
12. K. Goto and R. Van De Geijn. High-performance Implementation of the Level 3 BLAS. *ACM Trans. Math. Softw.*, 35(1):4:1–4:14, July 2008.
13. B. Hadri and M. R. Fahey. Mining Software Usage with the Automatic Library Tracking Database (ALTD). In H. Pfeiffer, D. Ignatov, J. Poelmans, and N. Gadiraju, editors, *ICCS*, volume 18 of *Lect. Notes Comp. Sci.*, pages 1834–1843. Elsevier, 2013.
14. N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, Philadelphia, 2002.
15. F. D. Igual, G. Quintana-Ort, and R. A. van de Geijn. Level-3 BLAS on a GPU: Picking the Low Hanging Fruit. *AIP Conference Proceedings*, 1504(1):1109–1112, 2012.
16. B. Kågström, P. Ling, and C. van Loan. GEMM-Based Level 3 BLAS: High-performance Model Implementations and Performance Evaluation Benchmark. *ACM Trans. Math. Softw.*, 24(3):268–302, 1998.
17. R. Nath, S. Tomov, and J. Dongarra. An Improved Magma GEMM For Fermi Graphics Processing Units. *Int. J. High Perform. Comput. Appl.*, 24(4):511–515, 2010.
18. R. Nath, Stan. Tomov, T. Dong, and J. Dongarra. Optimizing symmetric dense matrix-vector multiplication on GPUs. In *Proc. 2011 Int. Conf. High Perf. Comp., Netw., Stor. and Anal.*, SC'11, pages 6:1–6:10, New York, 2011. ACM.
19. V. Volkov and J. W. Demmel. Benchmarking GPUs to Tune Dense Linear Algebra. In *Proc. 2008 ACM/IEEE Conf. Supercomp.*, SC'08, pages 31:1–31:11, Piscataway, 2008. IEEE.