Batched Triangular Dense Linear Algebra Kernels for Very Small Matrix Sizes on GPUs

ALI CHARARA, DAVID KEYES, and HATEM LTAIEF, Extreme Computing Research Center, King Abdullah University of Science and Technology

Batched dense linear algebra kernels are becoming ubiquitous in scientific applications, ranging from tensor contractions in deep learning to data compression in hierarchical low-rank matrix approximation. Within a single API call, these kernels are capable of simultaneously launching up to thousands of similar matrix computations, removing the expensive overhead of multiple API calls while increasing the occupancy of the underlying hardware. A challenge is that for the existing hardware landscape (x86, GPUs, etc.), only a subset of the required batched operations is implemented by the vendors, with limited support for very small problem sizes. We describe the design and performance of a new class of batched triangular dense linear algebra kernels on very small data sizes (up to 256) using single and multiple GPUs. By deploying two-sided recursive formulations, stressing the register usage, maintaining data locality, reducing threads synchronization and fusing successive kernel calls, the new batched kernels outperform existing state-of-the-art implementations.

CCS Concepts: •**Mathematics of computing → Mathematical software;** •**Computing methodologies → Massively parallel algorithms;** •**Theory of computation → Divide and conquer;**

Additional Key Words and Phrases: KBLAS, Recursive Formulation, Batched BLAS Kernels, Dense Linear Algebra, Hardware Accelerators

ACM Reference format:
Ali Charara, David Keyes, and Hatem Ltaief. 2017. Batched Triangular Dense Linear Algebra Kernels for Very Small Matrix Sizes on GPUs. *ACM Trans. Math. Softw.* 9, 4, Article 39 (March 2017), 26 pages.
DOI: 0000001.0000001

1 INTRODUCTION

The Basic Linear Algebra Subprograms (BLAS) [Dongarra et al. 1988] and the Linear Algebra Package (LAPACK) [Anderson et al. 1999] libraries have been the foundation of the usual high-performance computing (HPC) software stack chain for decades. In fact, most of the today's scientific applications rely on a *de facto* highly optimized BLAS implementation, often provisioned by the vendor chip manufacturers, to extract performance from the underlying processing units. These architecture-dependent libraries (e.g., available in the Intel MKL [Intel 2017] for x86 or the NVIDIA cuBLAS/cuSOLVER libraries [NVIDIA 2017a,c] for GPUs) may allow applications developers to achieve close to bandwidth or floating-point performance sustained peak for memory-bound or compute-bound kernel workloads, respectively, across various hardware systems.

Author's addresses: Author's addresses: A. Charara (Ali.Charara@kaust.edu.sa), D. Keyes (David.Keyes@kaust.edu.sa) and H. Ltaief (Hatem.Ltaief@kaust.edu.sa), Extreme Computing Research Center, King Abdullah University of Science and Technology, Thuwal 23955, Saudi Arabia,

DOI: 0000001.0000001

However, some of the most critical applications currently of high interests in the HPC community, especially in data analytics, face a major performance bottleneck due to the inadequacy of legacy BLAS/LAPACK frameworks. For instance, tensor contractions [Abdelfattah et al. 2016] for deep learning and hierarchical low rank data-sparse matrix computations [Hackbusch 1999; Hackbusch and Khoromskij 2000] are key operations for solving partial differential equations. Interestingly, the bulk of the computation of these operations typically resides in performing thousands of independent dense linear algebra operations on *very* small sizes (usually less than 100). Even the highly vendor-optimized sequential implementations may not cope with the overhead of the memory latency at these tiny sizes. Moreover, calling the sequential version of the dense linear algebra functions within an embarrassingly parallel OpenMP loop may not be an option, due to the API overhead (i.e., parameters sanity check, memory initialization, etc.), which does not get compensated in return because of the low arithmetic intensity of the kernel operations. This is further exacerbated by hardware with a large number of threads, such as GPUs with many streaming multiprocessors, for which high occupancy may not be reached, bandwidth may not get saturated, and thread parallelism may not be exploited given the small workloads. At present, vendors currently provide only a subset of the overall batched linear algebra operations, with limited support for very small problem sizes.

This paper describes the high-performance implementations on GPUs of various batched triangular dense linear algebra operations targeting very small sizes (up to 256 in dimension), which are currently either poorly supported or not at all. There are two main algorithmic adaptations, which may address this challenge: designing synchronization-reducing (i.e., strong scaling) and communication-reducing (i.e., data motion avoiding) algorithms. Although both features are important moving forward with extreme scale simulations on future exascale systems, they also turn out to be crucial to get performance out of small workloads on highly parallel GPU devices. Our fundamental strategy consists in using a recursive formulation of the linear algebra operation, which inherently encompasses both algorithmic features, by recasting most of the memory-bound computations into compute-bound operations. In fact, performance optimizations and modeling of numerical kernels based on matrix-matrix multiplications (*GEMM*) have been well studied [Goto and Van De Geijn 2008; Igual et al. 2012; Kågström et al. 1998] due to the fact that the *GEMM* operation is highly parallel and maps well to the hierarchical memory of modern CPUs and accelerators, which also performs very close to the peak performance of such hardware devices. Recursive formulations leverage the performance of LAPACK/dense linear algebra (DLA) kernels by converting them into mostly *GEMMs*. Employing a recursive formulation is not a new technique in DLA [Andersen et al. 2001; Eliahu et al. 2015; Elmroth et al. 2004; Elmroth and Gustavson 2000; Kågström 2006; Peise and Bientinesi 2016]. Recursive DLA has been employed to minimize data motion across the hierarchical memory layers on x86 architectures [Elmroth et al. 2004; Kågström 2006]. It has also been applied to speed up large bandwidth-limited workloads, seen during the panel computation during factorizations of dense matrices. Recursion has been recently employed [Charara et al. 2016a,b] on large workloads to greatly enhance the performance of some already compute-bound triangular DLA kernels operating over NVIDIA GPUs. For GPU-based DLA kernels operating on much smaller workloads, the recursive scheme is even more of a necessity as it allows to maintain the data freshly fetched in from global memory at the level of the caches. Furthermore, to reduce vertical data motion and diminish the overheads of going to shared-memory back and forth, we stress register usage on the GPUs and utilize the CUDA *shuffle* instruction, which enables threads within a warp to communicate without having to rendezvous at the level of shared-memory. Additionally, we reduce synchronizations by operating on a whole matrix within a single warp.

We target some of the Level 3 BLAS triangular operations (symmetric rank-k update, triangular matrix-matrix multiplication and solve) as well as some triangular matrix computations from LAPACK (Cholesky factorization, corresponding linear solvers and matrix inversions). Some of our batched LAPACK functions reuse internally our batched BLAS kernels in a nested recursion fashion and create further optimizations opportunities by fusing the batched BLAS inner kernels for better performance. Our new implementations of single and nested batched kernels outperform existing state-of-the-art open-source and commercial implementations on GPUs.

The remainder of the paper is organized as follows. Section 2 presents related work. Section 3 outlines the operations currently supported in the KBLAS library. Section 4 recalls the challenges of designing triangular BLAS and LAPACK operations. Section 5 describes the fundamental algorithmic techniques to extract performance when dealing with a batch of matrix operations of very small sizes. The implementation details of the various batched GPU kernels are given in Section 6. Section 7 provides the performance results of various herein introduced batched triangular dense linear algebra operations on GPUs and compares them against existing state-of-the-art implementations. We conclude in Section 8.

2 RELATED WORK

The need for batched operations on BLAS routines – as well as on LAPACK routines – has been under thorough investigation lately [Dongarra et al. 2015]. The effectiveness of batched operations has been demonstrated in several applications [Abdelfattah et al. 2016; King et al. 2014]. Both vendors and academic institutions have contributed batched operations in their library recent releases. For instance, NVIDIA and Intel provide only a subset of batched operations in cuBLAS [NVIDIA 2017a] and the MKL [Intel 2017] / LIBXSMM [Heinecke et al. 2016] libraries, respectively, while MAGMA [MAGMA 2017] from the University of Tennessee supports further batched BLAS and LAPACK kernel operations.

In particular, batched *GEMM* has probably drawn the most attention since it is ubiquitous in emerging wide range of applications, e.g., related to the convolution operations in deep learning. cuBLAS [NVIDIA 2017a] provides a batched *GEMM* implementation on NVIDIA GPUs with both strided and non-strided forms. MKL [Intel 2017] provides an implementation for batched *GEMM* on Intel CPUs, in addition to the LIBXSMM [Heinecke et al. 2016] library, which is based on low-level code generation. MAGMA also provides an implementation for batched *GEMM* on both NVIDIA GPUs and on CPUs [Abdelfattah et al. 2016c; Masliah et al. 2016] for mid-range as well as very small matrix sizes, which are optimized at the register operations level. Besides batched *GEMM*, MAGMA implements other Level 3 BLAS batched operations, such as triangular solves (*TRSM*) and symmetric rank-k update (*SYRK*).

For advanced dense linear algebra algorithms from LAPACK, MAGMA leads the quest for developing high performance batched matrix factorizations: (1) batched Cholesky factorizations [Dong et al. 2014b], where three algorithms, i.e., non-blocked, blocked, and recursive blocked, are examined in contrast to the traditional hybrid CPU-GPU based factorization, (2) superseded later by batched Cholesky/LU/QR factorizations [Dong 2015; Dong et al. 2014a; Haidar et al. 2015a,b,c,d] using batched BLAS as building blocks and more recently, (3) revisited batched Cholesky factorization [Kurzak et al. 2016] using an auto-tuning framework. All these aforementioned batched kernels operate on matrices with same sizes. In [Abdelfattah et al. 2016a,b,d], a newer implementation in MAGMA is proposed for handling batched matrix factorizations with variable sizes, which has also been of interests in the context of accelerating sparse linear algebra [SuiteSparse 2017] during the Schur complement calculations. More recently, some of the authors have proposed new batched QR

and SVD kernels for very small matrix sizes with applications in the compression of hierarchical matrices [Boukaram et al. 2017].

In this paper, we focus our attention on batched Level 3 BLAS operations that involve triangular matrices (i.e., *TRSM*, *SYRK*, *TRMM*). We then derive several batched LAPACK algorithms that involve symmetric positive definite (SPD) matrices (Cholesky based factorization, solve, and inversion) using these aforementioned Level 3 BLAS triangular kernels. We have previously discussed and evaluated alternative formulations for the standard (non-batched) Level 3 BLAS triangular operations on large matrix sizes [Charara et al. 2016a,b] and have shown that recursive formulations may provide superior performance by optimizing the memory access patterns. Herein, we aim at extending this strategy to very small matrix sizes and improving parallel performance by means of batched operations, with a careful treatment on memory accesses. Moreover, at the time of writing, we only consider uniform sizes, which may be critical for accelerating low-rank matrix approximations and arithmetics during the preconditioning phase of sparse iterative solvers.

3 THE KBLAS LIBRARY

KBLAS[1] is an open-source library providing highly optimized kernel implementations of a subset of BLAS operations on NVIDIA GPUs [Abdelfattah et al. 2012, 2016e; Charara et al. 2016a,b]. All four standard precisions are supported.

3.1 Current Features

KBLAS currently provides support for a subset of the Level 2 and 3 BLAS kernels, i.e., the general and symmetric matrix-vector multiply (*GEMV* and *SYMV*) as well as the triangular matrix multiply and triangular solve with multiple right-hand sides (*TRMM* and *TRSM*), respectively, on single and multiple GPUs. The single GPU variant of these aforementioned Level 2 and 3 BLAS routines have been integrated into NVIDIA's cuBLAS library version 6.0 and 8.0, respectively [NVIDIA 2017b]. We also provide support for matrix-matrix multiplication *GEMM* on multiple GPUs.

3.2 Kernel API

We try —whenever possible— to be consistent in the KBLAS API with the standard/legacy BLAS API for easy code integration. This is especially true for single GPU routines, which operate on the default CUDA stream, and thus are considered synchronous ones. Corresponding asynchronous routines are provided, which accept a CUDA stream as a parameter and are suffixed with *_async*. Similarly, all KBLAS routines are prefixed with *kblas_*, in accordance with the naming conventions in MAGMA and cuBLAS. Level 2 and 3 BLAS routines assume their parameter data already reside on the device memory, thus, categorized as GPU API. KBLAS provides corresponding CPU API routines which assume data is resident on the host memory and will implicitly handle the data transfer. These routines accept CPU data pointers and are suffixed with *_cpu*. Routines for multiple GPUs support are suffixed with *_mgpu*, and accept an extra parameter specifying the number of devices to run on, with an optional parameter specifying the device ID to use.

3.3 New Features

The new batched Level 3 BLAS kernels supported in KBLAS and targeted in this paper are *TRSM*, *TRMM*, and the symmetric rank-k update kernel *SYRK*. We further provide the following batched operations of high-level LAPACK/DLA, which inherently depend on batched *TRSM*, *TRMM* and *SYRK*: the Cholesky factorization (*POTRF*), the positive definite triangular solve (*POTRS*), the solution to system of linear equations with positive definite matrix (*POSV*), the inverse of a real

[1]Available online at http://github.com/ecrc/kblas-gpu.

upper or lower triangular matrix (*TRTRI*), the product of upper or lower triangular matrix with its transpose (*LAUUM*), the inverse of a real symmetric positive definite matrix (*POTRI*), and a new LAPACK routine resulting from the merge of the Cholesky factorization and inversion of symmetric positive definite matrices (*POTI*).

These new batched routines follow the naming and API conventions adopted in MAGMA/cuBLAS, and are suffixed with *_batch*. Besides the legacy BLAS and LAPACK parameters, batched routines have two flavors: one accepting an array of pointers to the batch of matrices assuming the input data is scattered in memory and another that assumes the data have contiguous memory allocation, thus, accepting a single memory address pointer and a stride value between consecutive input matrices.

4 LIMITATIONS OF TRIANGULAR DLA OPERATIONS

It is critical, for adequately designing the needed batched CUDA kernels, to study, understand and expose the level of parallelism inherent in the triangular DLA operations. In fact, the data dependency in the algorithm of each operation dictates the degree of parallelism of such operation, and consequently, may limit or enrich the level of parallelism. In the context of our targeted triangular DLA operations, two key factors play the main role in determining the level of inherent parallelism: the triangular matrix shape and in-place (IP) nature of the operation.

The IP nature of triangular DLA operations encounters some data read-write dependency hazards, since computing some elements need the values of other elements before the latter get updated (Write-After-Read or WAR hazard) or after they get updated (Read-After-Write or RAW hazard). Such dependencies limit the parallelism inherent in the DLA operation and may allow processing only one row or column of elements at a time. To alleviate these data dependency impacts on parallelism, one option is to use an out-of-place (OOP) variant at the cost of extra memory allocations and data transfers, and therefore, limiting the problem size that can be processed within the limited GPU memory, but most importantly, not conforming to the legacy BLAS API.

The other key factor, i.e., the triangular or symmetric shape of involved matrices, although only half of the matrix elements are processed, engenders load imbalance among threads that compute in parallel a row or a column of entries, and they may have to diverge as they cross the diagonal entries. Since threads within a CUDA warp execute the same instructions in a lock-step fashion, thread divergence within a warp is expensive, because it forces diverging threads to idle status while other lock-step synchronized threads compute, thus wasting valuable core cycles. The authors in [Abdelfattah et al. 2016d] design two variants of batched Cholesky factorization to cope with the triangular nature of the matrix: loop-inclusive and loop-exclusive. In the first, all factorization iterations are executed in one kernel to maximize chances of data reuse. In the later, each iteration is executed in a separate kernel launch to optimize resource utilization.

Instead, our approach to remedy the effect of both factors, triangular shapes and IP nature, is based on a holistic set of techniques to increase parallelism while enhancing memory accesses: a) two-sided recursive formulations, with nested recursion, which help optimize resource allocation and relieve WAR and RAW data dependency hazards, b) register blocking, which relaxes shared memory limitation constraint and operates at faster register memory, c) fused kernels, which improve data reuse, and d) nested batches, which increase concurrency in computation.

5 FUNDAMENTAL ALGORITHMIC TECHNIQUES

In this section, we describe the techniques to improve the performance of batched triangular BLAS and LAPACK operations for very small sizes on GPUs. Although these operations differ in their processing algorithms and their outcome, they share a common trait, that is, involving triangular

matrices. We thus operate on them in a uniform approach, at two levels: 1) for tiny matrices (size up to 16), we design highly optimized CUDA kernels 2) for matrices of larger sizes (up to 256), we operate on them recursively, converting most of the computations into GEMM operations, then stop at the diagonal blocks of size 16 on which we apply the aforementioned CUDA kernels. The techniques discussed below apply to either of these two levels.

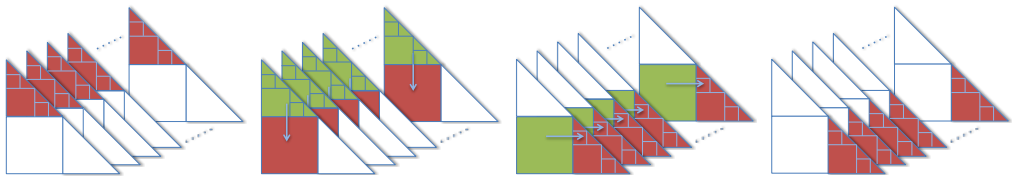
5.1 Two-Sided Recursive Blocking

Upon processing matrices of medium to large sizes, blocking is a necessary practice because such matrices cannot fit in the small (first level) caches available in modern CPUs or GPUs. Blocking helps in stressing the hierarchical memory of both CPU and GPU architectures.

In a recent work [Charara et al. 2016a,b], we have illustrated, for large matrix sizes, how two-sided recursion reduces data transfer across the GPU memory hierarchy and increases parallel performance, when applied to triangular Level 3 BLAS operations, by casting most of the computations in terms of *GEMM* operations.

Such blocking may be not necessary when processing small matrices that fit in cache. However, when processing a large number of small matrices, caches may quickly saturate, and one should still consider a form of blocking for this case. Therefore, we extend our previous technique for handling batched operations on small matrix sizes. Fig. 1 illustrates the successive steps for the batched *DPOTRF* operation using two-sided recursive blocking. In our design, the two-sided recursive formulations resort at the recursion base —for processing of diagonal blocks— to highly optimized CUDA kernels, which store and operate on data at the level of registers, as shown in Fig. 1(a), Fig. 1(b) and Fig. 1(c) for the recursive *POTRF*, *TRSM* and *SYRK* kernels, respectively.

Note that the two-sided recursive blocking we employ here is fundamentally different from a one-sided recursive blocking (otherwise known as block algorithm), where a panel with predefined width is factorized followed by an update to the right side, then single sided recursion is applied to the right side. Our two-sided recursive formulation starts by splitting the triangular matrix at (usually) half size, then operates recursively on each side either independently or sequentially based on the involved triangular operation. It is worth mentioning that, with these recursive formulations,



(a) Rec. Batched POTRF (b) Rec. Batched TRSM (c) Rec. Batched SYRK (d) Rec. Batched POTRF

Fig. 1. Illustrating recursive batched *POTRF* with nested recursion.

the only tuning parameter needed is the recursion stopping case, which is affected empirically by the register demand of each kernel and the effect it has on the kernel's occupancy. One may compare this tuning approach to that adopted by [Kurzak et al. 2016], which requires extensive tuning through the BEAST (Bench-testing Environment for Automated Software Tuning) [BEAST 2017] auto-tuning framework.

5.2 Register Hosted Computations

Similar to the CPU memory hierarchy, an NVIDIA GPU features memory hierarchy structured as follows, mentioned in their ascending order of access speed: 1) a device on-chip memory that

resides on the card, with high bandwidth (relative to SDRAM), which is a few GBs in size, 2) a non-programmable on-chip L2-cache that is shared among all Streaming Multiprocessors (SM), 3) a local non-programmable L1 / read-only texture cache dedicated to each SM, with a (usually 64KB) programmable shared memory, and 4) a 32-bit register file accessible by threads running on the SM's cores.

Note that registers are the fastest memory, on which it is desirable to make direct computations. Note also that CUDA imposes a limitation on how much a thread block (TB) can consume of shared memory. In our case, since the processing of each matrix (of the batched operations) is independent of the others and no data re-use is possible across the input matrices, it would be necessary to cache each matrix into shared memory to operate on it. However, saturating the shared memory per TB prevents other TBs from loading into the same SM and, consequently, sharply decreases the occupancy of the SMs and prevents proper latency hiding. For that reason, we adopt a different strategy, which relies on caching data in registers only.

Recall that recursive formulations —explained above— convert off-diagonal block processing into batched *GEMM* calls, while diagonal block processing is handled by our CUDA implementations for the BLAS or LAPACK batched kernels. We design highly optimized kernels that operate on tiny diagonal blocks (up to 16 in size). We fit the data on the registers of the threads cooperating to perform each matrix operation. Two key advantages can be mentioned for fitting data in registers only. First, we avoid —in most of the kernel implementations– the need for more expensive shared memory accesses. Second, we avoid saturating shared memory and consequently avoid limiting the number of concurrent TBs executing on the same SM. However, since the register file is shared among the executing threads, saturating register usage may also limit the number of concurrently executing TBs in an SM. For that reason, careful treatment and pressure on registers is needed to avoid register spilling into much slower local memory (which is hosted on the device memory).

On the other hand, loading data only in registers limits the possibilities of sharing data among threads (data sharing is possible only between threads of the same warp), unless we share data through shared memory, which we are trying to avoid. For that reason, we make sure that processing each matrix is limited to threads of the same warp. Whenever data sharing is needed among threads processing the same matrix, we utilize the *shuffle* instruction. The shuffle instruction allows registers of the same warp to share their values in four possible ways: shuffle up, shuffle down, butterfly and indexed shuffle [NVIDIA 2017d]. It is also possible to share data among sub-groups of the warp threads by defining the shuffle width, which allows us, by specializing thread sub-groups within a warp, to process multiple matrix operations with one warp. Moreover, processing each matrix operation within one warp provides the extra advantage of removing any artificial synchronizations since threads in a warp are already lock-step synchronized. By avoiding synchronizations, the warp scheduler can do a better job hiding the data fetch latency, which becomes the main bottleneck. All in all, by using register blocking and shuffle instruction, we avoid shared-memory limitations and latency, can share data between threads at the cost of no extra cycles, and reduce synchronizations among warps.

5.3 Nested Batching Calls

As explained above, recursion breaks the computation of triangular BLAS or LAPACK routines into a set of sub-operations involving diagonal or off-diagonal matrix blocks. In some operations, e.g., *SYRK* or *TRTRI*, computation of some or all of the diagonal or off-diagonal blocks may be independent and thus may be computed in parallel, resulting in an additional level of parallelism, referred to as nested batching calls. Indeed, while a batched sub-call operates on the same corresponding sub-block of each input matrix, a nested batching call can combine several sub-calls to batched

routines operating on several sub-blocks —from the same recursion level— into one batched call with 2D thread-block (TB) grid, or by issuing parallel batched calls on multiple CUDA streams. Figure 2 illustrates this concept for the batched *SYRK*. This is especially effective when the number of batched matrices does not provide enough workload to saturate the GPU occupancy needed to utilize all the GPU cores and to hide data fetching latency. This low occupancy situation is typically encountered when running strong scaling experiments on multiple GPUs and nested batching calls may remedy such a bottleneck by further increasing concurrency.

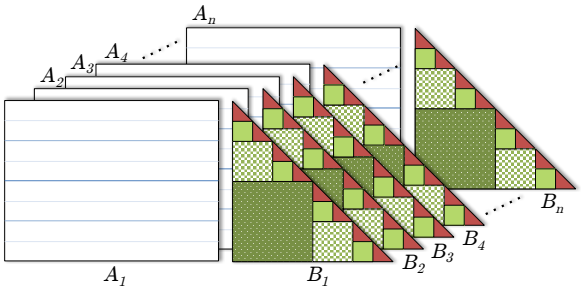


Fig. 2. Illustration of nested batching calls for the batched *SYRK*. Blocks of similar green shades can be processed in one batched *GEMM* call. Diagonal red-shaded blocks can be processed in one batched *SYRK* call.

5.4 Kernel Fusion

Processing thousands of matrices with very small sizes concurrently in batched mode may still fall into the category of a memory-bound operation because arithmetic intensity is very low, especially when operating on hardware with over-provisioned flops. It would be, therefore, of high importance to avoid data transfer, whenever possible, when designing our kernels. Indeed, particular attention should be made when two or more kernels operate successively on the same data block. As outlined above, we designed our kernels to operate on data that fit in registers. Thus, instead of launching multiple kernels, with its associated kernel launch overhead, and reading / writing the data multiple times, we fuse the multiple kernels into one code that performs all the corresponding operations in-place, as long as the data fits in registers and the successive operations occur on the same data block. As an example, this is the case when solving a system of linear equations (*POTRS*), which consists of two consecutive triangular solves.

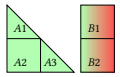
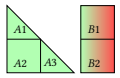
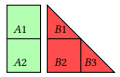
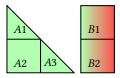
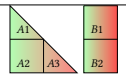
6 HIGH PERFORMANCE IMPLEMENTATION DETAILS

In this section, we present a detailed description of the implementations of the batched triangular operations, based on the guidelines outlined in Section 5. There are two main components for the studied batched kernels: the two-sided recursive formulations, and the required techniques for highly optimized CUDA kernels operating on tiny matrix sizes, which are invoked at the recursion stopping criterion.

6.1 Two-sided Recursive formulations

Table 1 shows the two-sided recursive formulations we have applied for each of the introduced batched operations. The first column of the table lists the mathematical representation of each operation. The table lists one variant for each of the operations (the second column); other

Table 1. Recursive formulations of the batched operations and the CUDA kernels called at the recursion stop.

Batched operation	Variant[a,b]		Recursive definition[c]		Split[d]	Kernel (max)[e]
$TRSM : A\,X = \alpha B$	LLN	*RecTRSM*:	$A_1\,X_1 = \alpha\,B_1$ $B_2 = \alpha\,B_2 - A_2\,B_1$ $A_3\,X_2 = B_2$	*RecTRSM* *GEMM* *RecTRSM*		*KerTRSM*(16)
$TRMM : B = \alpha\,A^T\,B$	LLT	*RecTRMM*:	$B_1 = \alpha\,A_1^T\,B_1$ $B_1 = \alpha\,A_2^T\,B_2 + B_1$ $B_2 = \alpha\,A_3^T\,B_2$	*RecTRMM* *GEMM* *RecTRMM*		*KerTRMM*(16)
$SYRK : B = \alpha AA^T + \beta B$	L _ N	*RecSYRK*:	$B_1 = \alpha A_1 A_1^T + \beta B_1$ $B_2 = \alpha A_2 A_1^T + \beta B_2$ $B_3 = \alpha A_2 A_2^T + \beta B_3$	*RecSYRK* *GEMM* *RecSYRK*		*KerSYRK*(16)
$POTRF : A = L\,L^T$	L _ _	*RecPOTRF*:	$A_1 = L_1\,L_1^T$ $A_1\,X = A_2$ $A_3 = -A_2\,A_2^T + A_3$ $A_3 = L_3\,L_3^T$	*RecPOTRF* *RecTRSM* *RecSYRK* *RecPOTRF*		*KerPOTRF*(16)
$POTRS : A\,X = B$[f]	L _ _	*RecPOTRS*:	$A_1\,X_1 = B_1$ $B_2 = B_2 - A_2\,B_1$ $A_3\,X_2 = B_2$ $B_2 = B_1 - A_2\,B_2$ $A_1\,X_1 = B_1$	*RecTRSM* *GEMM* *RecPOTRS* *GEMM* *RecTRSM*		*KerPOTRS*(16)
$POSV : A\,X = B$[g]	L _ _	*POSV*:	$A = L\,L^T$ $A\,X = B$	*RecPOTRF* *RecPOTRS*		*KerPOSV*(8)
$TRTRI : A = A^{-1}$[h]	L _ _	*RecTRTRI*:	$X\,A_1 = -A_2$ $A_3\,X = A_2$ $A_1 = A_1^{-1}$ $A_2 = A_2^{-1}$	*RecTRSM* *RecTRSM* *RecTRTRI* *RecTRTRI*		*KerTRTRI*(16)
$LAUUM : A = A\,A^T$	L _ _	*RecLAUUM*:	$A_1 = A_1\,A_1^T$ $A_1 = A_2^T\,A_2 + A_1$ $A_2 = A_3^T\,A_2$ $A_3 = A_3\,A_3^T$	*RecLAUUM* *RecSYRK* *RecTRMM* *RecLAUUM*		*KerLAUUM*(16)
$POTRI : A = A^{-1}$[i]	L _ _	*POTRI*:	$A = A^{-1}$ $A = A\,A^T$	*RecTRTRI* *RecLAUUM*		*KerPOTRI*(8)
$POTI : A = A^{-1}$[j]	L _ _	*POTI*:	$A = L\,L^T$ $A = A^{-1}$	*RecPOTRF* *RecPOTRI*		*KerPOTI*(8)

[a]Variant legend: Upper/Lower, Left/Right, Transpose/Non-transpose.
[b]Other variants bear similar recursive formulations, not mentioned for brevity.
[c]Rec-prefix denotes recursive call.
[d] denotes input matrix, denotes output matrix, denotes input-output matrix.
[e]Ker-prefix denotes the CUDA kernel. Max is the maximum size the kernel supports.
[f]*POTRS* assumes matrix A is factorized.
[g]*POSV* factorizes matrix A, then applies *POTRS*.
[h]*TRTRI* assumes matrix A is triangular (after factorization).
[i]*POTRI* assumes matrix A is factorized.
[j]*POSV* factorizes matrix A, then applies *POTRI*.

variants are supported and can be described in a similar recursive fashion. The table illustrates the input/output/input-output matrices and colors them accordingly. Note that, except for *SYRK*, all the targeted operations are in-situ (denoted by mixed shade). For example, in *TRSM*, the output matrix X overwrites the memory occupied upon input by the matrix B; therefore the second line of the recursive definition uses B_1 where, logically, X_1 should appear.

Two-Sided Recursive Implementation. The two-sided recursive algorithms for triangular matrices are generally described as follows. Assuming the lower case, split the triangular matrix A into three sub-matrices: an upper-left triangular matrix $A1$, a lower-left rectangular matrix $A2$, and a lower-right triangular matrix $A3$, as illustrated in Table 1. We split matrix B, if involved, at a corresponding index, into $B1$ and $B2$. This is translated as splitting the rows of B for left-sided operations, and the columns of B for right-sided ones. The split is preferred to be at an index that is a power of 2 since this produces fewer clean-up trailing matrices and, thus, helps generate better performance at regular sizes; however, to avoid operating on thin matrices produced upon encountering sizes that are slightly bigger than powers of 2, one may split at a power of 2 that is not the closest. We proceed by applying the same recursive scheme to the left side of the recursion involving $A1$, applying any middle updates involving $A2$, then applying the recursive scheme to the right side of the recursion involving $A3$. The recursive scheme for each operation is also detailed in Table 1. The recursion is stopped upon reaching a size that can be handled by our CUDA kernels as described in Section 6.2.

Nested Recursion. The main advantage of our two-sided recursive implementation of the batched DLA kernels is in relaxing the WAR or RAW data dependency hazards. For the case of batched BLAS operations, the recursive scheme achieves this relaxation by converting the IP BLAS operations into a set of calls to OOP *GEMMs*. However, the recursive scheme of some LAPACK operations does not involve direct conversion to *GEMMs*, as detailed in Table 1, e.g., in the case of Cholesky factorization (*POTRF*). To remedy this shortcoming, it is necessary to use nested recursion, that is, invoke the recursive formulations of the sub-components of the main recursive call. For example, *RecPOTRF* invokes itself as well as *RecTRSM* and *RecSYRK*, which, in turn, convert into batched *GEMM* calls.

6.2 CUDA Kernels

We discuss the design and implementation of the KBLAS CUDA kernels that process batched operations on matrices of tiny sizes (up to 16). Table 1 lists the CUDA kernels that are needed for the corresponding batched operations. Recall our motivation to store and process such tiny matrices in registers, as explained in Section 5.2, in order to avoid the latency of shared memory, since such data can fit in registers only. However, we need to carefully craft the kernel's pressure on registers in order to avoid register spillage into much slower local memory.

Note that, since the processing of each matrix in the batched operation is independent, there is, generally, enough parallelism in batched operations to involve all the cores of the underlying hardware in the computation, especially when the batch size is large enough. However, that is not enough to extract high performance from the involved operations for several reasons. First, due to the low arithmetic intensity of the targeted operations at very small matrix sizes, these operations are memory bound. Thus, proper utilization of the hierarchical GPU memory structure and bandwidth is needed. Second, since each matrix operation is independent of the other batched operations, each matrix will need to be loaded into shared memory or registers or both to be processed, which brings huge pressure on the low capacity shared memory and registers. The alternative option of accessing the main memory directly from the CUDA threads is not considered

due to the inherently slow global memory accesses. Such memory pressure would sharply decrease the occupancy of the GPU and consequently decrease the kernel's performance. Third, the triangular or symmetric shape may drive threads to an idle state, while other threads are computing.For these reasons, careful design of the CUDA thread blocks (TB) and TB grid is needed, as discussed in the next subsections.

6.2.1 Thread-Block Design. Multiple options can be explored for the design of TB and the distribution of computation on multiple threads, some of which has been explored by [Oreste et al. 2013] in the context of batched LU factorization. In the next few paragraphs, we explore and evaluate these options and justify our choice of TB design and TB grid layout.

Thread Level Parallelism. In this layout, one CUDA thread computes one operation. Such layout would allow arbitrary matrix sizes to be processed per thread with no idle cycles. However, processing each matrix operation will be serialized being served by one thread. Its main disadvantage is in having to fetch all data directly from main off-chip memory since neither single thread register capacity nor shared memory capacity would suffice for caching the involved matrices, unless very few threads are involved per TB, which in turn engenders very low occupancy, and consequently very low performance. Obviously, this approach is not the best configuration we can use.

Warp Level Parallelism. In this layout, one warp computes one operation. Since the parallelism available in all of our target kernels is limited to processing one row (or column) concurrently, due to the IP requirement, this TB layout allows one thread to compute all elements of a column (or row resp.) sequentially. This layout makes better utilization of parallel CUDA threads and may allow data of each matrix to be stored within registers, and eventually may allow supporting matrices of size up to 32 in dimension per kernel. However, due to the triangular shape of the matrices, a high percentage of the consumed cycles would be spent by idle threads. Moreover, the register pressure would be very high (demanding $32 * 32 * 2 = 2048$ registers per warp for the double precision case to process a matrix of size $32x32$), which causes register spillage to the slower local memory and sharply decreases the occupancy. Such configuration is far less than optimal for our purposes.

Thread-block Level Parallelism. One TB (including one or more warps) to process one matrix is also not a suitable choice for the same reasons above. Additionally, only a small number of TBs can run concurrently on an SM by CUDA design, which definitely does not increase occupancy, especially when the batch size is huge.

Thread-group Level Parallelism. We propose and implement this TB design that makes better utilization of the SM resources and minimizes idle thread cycles. We introduce the notion of thread groups (TG). A TG is a subset of threads of a warp that can share register values through the CUDA shuffle instruction. The shuffle instruction can share data between threads within a thread-index range. This range is enforced by CUDA to be a power of 2 (i.e., only possible values are 2, 4, 8, 16, or 32). Consequently, we define a TG size (TGS) in a corresponding manner (i.e., 4, 8, or 16). For our purposes, a TG would store and compute one matrix operation. The amount of register storage needed is also dictated by the TGS for triangular matrices. We, thus, introduce the constant EPT (elements-per-thread), which is the size of an array statically declared by each thread to store the input/output matrices elements. The EPT for a triangular matrix has to be congruent to TGS; however, EPT for a rectangular matrix can be tuned for optimal register pressure. We also introduce the tunable parameter $WPTB$, representing the number of warps per TB. Consider one more constant WS representing the warp size (which is fixed by CUDA to 32, but may change with future GPU architectures). With these notions in place, we design our thread blocks as two-dimensional blocks (TBx, TBy), where vertical dimension $TBx = TGS$, and horizontal dimension

$TBy = WPTB \times WS \,/\, TBx$. Consequently, one warp stores and processes multiple matrices. Figure 3 illustrates a typical example case where the matrix dimension of the batched matrices is 8 or less; we use $TGS = 8$, $EPT = 8$; consequently, a warp is serving four matrices, in this particular example, (demanding $8x8x2x4 = 512$ registers to host four $8x8$ matrices). The choice of $WPTB$ is empirically influenced by the register demand of the threads, which in turn affect occupancy. The options are limited and range from 1 to 4. Not that these design parameters (TGS, EPT, $WPTB$) are templatized and automatically selected at runtime based on input matrix size, operation variant, and kernel traits.

6.2.2 TB-Grid Design. CUDA thread blocks can be organized in 1, 2, or 3-dimensional grids. Given the TB design outlined previously, we use 1D grids for the general cases. The 1D grid size (GS) is affected by two values: the batch size (BS), and the number of operations served by a TB, which is equivalent to TBy. Thus, the grid size is determined by $GS = \lceil BS \,/\, TBy \rceil$. However, there are two cases where we employ 2D grids to enable further optimizations as explained in following paragraphs.

Nested Batching Calls. We discussed in Section 5.3 the concept of nested batching calls. This can be achieved by deploying multiple CUDA streams to execute in parallel the independent batched calls within a recursion level. It can also be achieved in a simpler way by deploying 2D grid of the involved batched call, the x-dimension is used as usual across the batch size, and the y-dimension is used to batch across the parallel computations of sub-blocks from the same matrix. This is the case, for instance, for the kernels $KerSYRK$ and $KerTRTRI$. Figure 2 illustrates this concept for the $KerSYRK$ kernel, applied to the red diagonal blocks, which can be batched within one whole $SYRK$ operation as well as across the batch size. The same can be said about similarly green-shaded blocks.

Enhancing Parallelism for Large Dimension. In this paper, we target matrices with very small sizes, i.e., up to 256 in dimension. However, some operations accept two inputs for dimensions, take for example the left-sided $TRSM$ operation, it takes as input M, the number of rows of the triangular matrix A, and $NRHS$, the number of right-hand sides for matrix B. In order to allow the $NRHS$ to grow beyond the small size, we split the $NRHS$ into chunks, and deploy a 2D grid of TBs, where the x-dimension works across the batch size as usual, while the y-dimension launches TBs across the $NRHS$ chunks. Such arrangement of a 2D grid is needed in $KerTRSM$, $KerTRMM$, $KerPOTRS$, and $KerPOSV$.

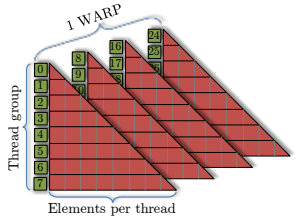


Fig. 3. A warp split into 4 thread-groups (TG). Each thread stores 8 matrix elements in registers. Each TG stores and processes a matrix.

Table 2. Fused kernels.

Kernel (Max Size)	Fused kernels
$KerSYRK(16)$	$KerSYRK(8) + KerGEMM(8) + KerSYRK(8)$
$KerPOTRF(16)$	$KerPOTRF(8) + KerTRSM(8) + KerSYRK(8)$
$KerPOTRS$	$KerTRSM + KerTRSM^T$
$KerPOSV$	$KerPOTRF + KerTRSM + KerTRSM^T$
$KerPOTRI$	$KerTRTRI + KerLAUUM$
$KerPOTI$	$KerPOTRF + KerTRTRI + KerLAUUM$

6.2.3 Additional Optimization Techniques.

Shared Memory Buffering. In [Abdelfattah et al. 2016e], we introduced the concept of double buffering at the register level. Double buffering is used to hide the latency of fetching data from main GPU memory by pipelining its instructions with other computation instructions. In this work, we employ the same concept by buffering at the shared memory level since register space is quite tight. We use shared memory buffering for two purposes. The first type of usage is for early fetching of data that is scheduled to be processed next. By the time the current block is processed, the next block is being fetched into the shared memory block in parallel, thanks to the possible pipelining of instructions made possible by the advanced *nvcc* compiler. When ready, the data at shared memory buffer would be copied to the register buffer for processing, and then the next block is fetched again. Shared memory buffering is needed in the kernels $KerTRSM$, $KerTRMM$, $KerPOTRS$, and $KerPOSV$. More important, we use shared memory buffering to avoid non-coalesced memory reads. When fetching data into register blocks, non-coalesced reads may be encountered, especially with transposed cases. To avoid these slow reads, we fetch the data into shared memory with coalesced reads, then transpose the data into register blocks for processing.

Kernel Fusion. In Section 5.4, we explain and justify the need and advantage for fusing batched kernels. We note here the operations where we apply batched kernel fusion within KBLAS implementation. Table 2 summarizes the fused batched kernels and the corresponding matrix sizes.

Loop Unrolling. Loop unrolling is an effective optimization technique that is widely used. It enhances instruction level parallelism (ILP) with the aid of the compiler. Loop unrolling can be automatically applied by the compiler based on the level of optimization flags, or by explicitly instructing the compiler to apply it with the special pragmas. Using the *nvcc* compiler from NVIDIA, loop unrolling can be applied only when the number of iterations of the loop is statically defined at compile time. For this reason, by parameterizing our kernel implementations with TGS, all the internal loops that use TGS may be unrolled.

7 PERFORMANCE RESULTS AND ANALYSIS

This section highlights the impacts of the optimization techniques on the batched DLA operations for small matrix sizes and compares the performance against existing CPU and GPU implementations (whenever available) on various hardware configurations.

7.1 Environment Settings

Experiments reported below have been conducted on single and multiple NVIDIA K40 GPUs across all subsequent GPU performance plots (unless noted otherwise). Experiments for CPU based performance have been conducted on an Intel multi-core Broadwell system with two sockets of 14 cores each and 64 GBs of memory. Since all of our implementations are equivalent in flop count to the native versions, we report performance results in flops per second, which are obtained by dividing the theoretical flop count of each algorithm by the execution time, over a range of input sizes. We use CUDA v7.5, Intel compilers v16, and MAGMA v2.2.0. Performance timing does not include data transfer to/from GPU since data is assumed to reside on the GPU memory. All results are reported with IEEE compliant compilation (without fast-math compiler optimization). Although single and double precision arithmetics (SP and DP) are only shown in the subsequent performance graphs, KBLAS is distributed with a support to all precisions. Batch sizes considered are enough to saturate the device and we generally use a batch size in SP twice larger than DP to maintain the same amount of data for both studied precisions.

CPU Batched DLA. Batched processing on CPU is handled with simple OpenMP parallel loop since the only library available on x86 architectures that perform batched processing (Intel's

MKL [Intel 2017] / LIBXSMM [Heinecke et al. 2016]) provides only batched GEMM so far. In our experiments, we found out that the best performance —for batched DLA processing on CPUs with small matrices— are achieved by processing each matrix with a single threaded MKL call embedded in an OpenMP parallel *for* loop. These findings are also confirmed by [Haidar et al. 2015a].

7.2 Performance Gain Breakdown

To properly assess the fundamental techniques introduced in Section 5, we examine the incremental performance gain of each optimization technique.

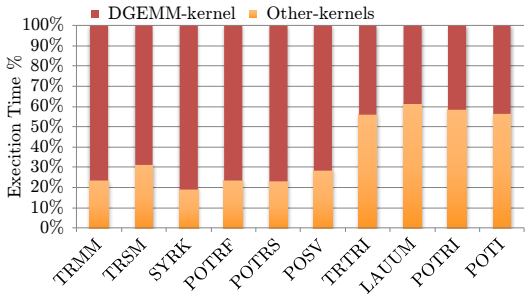
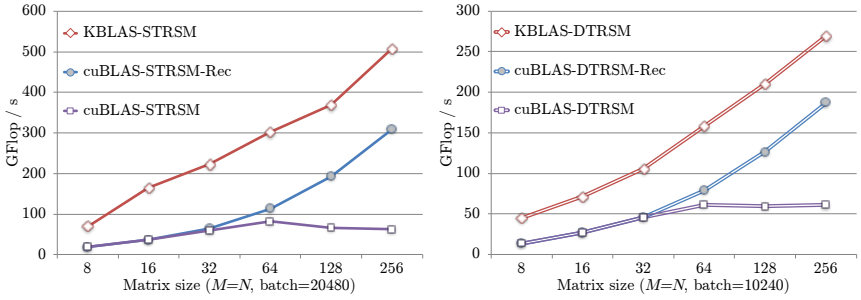


Fig. 4. Profiling of recursive batched operations, showing most of time is spent in batched *DGEMM*.

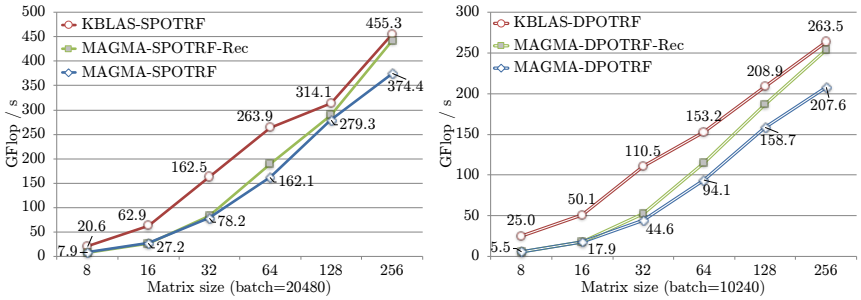
Two-sided Recursion. The two-sided recursive formulations play a major role in accelerating triangular operations by converting them into mostly *GEMMs* of (nearly) square sizes. Figure 4 shows a breakdown of the execution time of each kernel in all the triangular operations, expressed as a percentage of the total time for execution. The figure shows that time spent in *GEMM* operation, due to recursive formulations, is dominant with respect to the other kernels. For instance, thanks to nested recursion, *RecPOTRF* is decomposed into a set of batched *GEMM* calls, which, in turn, consumes 77% of the execution time. The performance impact over the non-batched version is demonstrated in Figure 5 for the case of batched *STRSM* and *DTRSM*. The curves *cuBLAS − xTRSM − Rec* show the effect of applying recursion over the corresponding cuBLAS *TRSM*, where we call cuBLAS batched *GEMM* kernel for off-diagonal blocks and cuBLAS batched *TRSM* kernel for recursion base. Since recursion in batched *RecTRSM* begins at size 32 and beyond, it does not affect at lower sizes. Indeed, the effect of recursion magnifies with the increase in matrix sizes as larger *GEMMs* are generated, reaching beyond 2.5X speedup at the higher end of the targeted matrix sizes. Similarly, The curves *MAGMA − xTRSM − Rec* in Figure 6 show the effect of applying recursion over the corresponding MAGMA *POTRF*, where we call KBLAS batched *TRSM* and *SYRK* for off-diagonal blocks and MAGMA batched *POTRF* kernel for recursion base.

Register Hosted Computations. The CUDA kernels —which operate at the register level— are most effective at the tiny sizes (below recursion base). Their impact on performance gradually decreases with the increase in matrix size since other kernel calls (mainly *GEMM*) become dominant. The curves *KBLAS − xTRSM* in Figure 5 illustrate the additional gain in performance achieved by replacing the corresponding cuBLAS batched *TRSM* kernel with a register-hosted KBLAS batched *TRSM* kernel for the recursion base, on top of that achieved by recursive formulations. Such gains range from 1.6X and 1.4X, at the upper spectrum of matrix sizes, up to 3.7X and 3.3X, at the lower spectrum, for SP and DP respectively. The curves *KBLAS − xPOTRF* in Figure 6 also highlight the additional gain in performance achieved by replacing MAGMA batched *PORTF* kernel with a



(a) Batched *STRSM*. (b) Batched *DTRSM*.

Fig. 5. Effect of recursion on performance of batched cuBLAS *TRSM* running on NVIDIA K40 GPU.



(a) Batched *SPOTRF*. (b) Batched *DPOTRF*.

Fig. 6. Effect of recursion on performance of batched MAGMA *PORTF* running on NVIDIA K40 GPU.

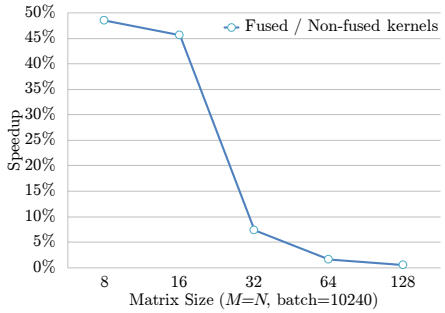


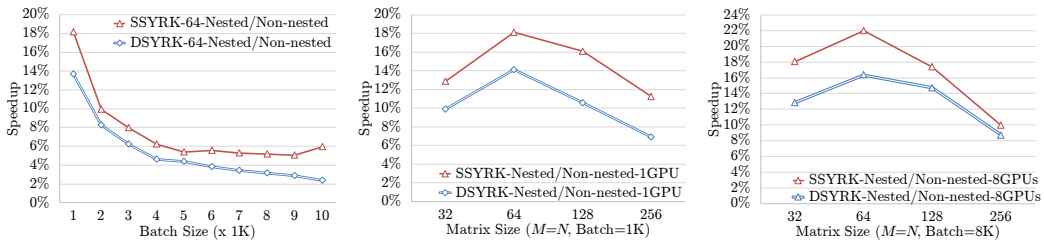
Fig. 7. Speedup gained by fusing two kernels of *DPOTRS*.

register-hosted KBLAS batched *POTRF* kernel for the recursion base. Such gains range from 1.1X, at the upper spectrum, up to 2.8X and 4.3X, at lower spectrum, for SP and DP, respectively.

Kernel Fusion. Figure 7 shows a sample of performance gain obtained by fusing the two kernels of *POTRS* running on a K40 GPU with 10240 batch size. Note that the gain is maximum (reaching close to 1.5X) when the execution involves the fused CUDA kernels only (for sizes up to the

recursion base); It decreases when recursion takes over by getting invoked on top of the fused kernels, involving *GEMM* calls for off-diagonal blocks.

Nested Batching Calls. Figure 8 shows the effect of nested batching calls on the batched *SYRK* kernel. The speedups gained for these sample runs range from 11% and 7% up to 18% and 14%, for SP and DP, respectively. Recall that nested batching calls combine several partial batch calls into one call —when possible– to improve occupancy and minimize kernel launch overhead. This is most effective when the batch size is small, where the workload would not saturate the GPU. Figure 8(a) demonstrates the speedup gained with nested batching calls for a matrix of size 64 while varying the batch size. The plot shows that, indeed, the gain is considerable when the batch size is small. Such gain gradually decreases with the increase in batch size as expected, since each partial batch call would bring enough workload to the GPU device. Figure 8(b) shows the speedup gain as the matrix size changes on a single K40 GPU. Note that the speedup decreases with the increase of matrix size as larger *GEMM* calls dominate the execution time. However, this technique is still effective in a strong scaling setup. Figure 8(c) shows the speedups gained with nested batching calls on 8 GPUs with batch size of 8K. The distributed load on each GPU is rather small where the nested batching becomes effective, thus exposing further parallelism and resulting in better GPU utilization. The resulting boost in performance reaches up to 22% and 16%, for SP and DP, respectively.



(a) Varying batch size for square matrices of size 64.

(b) Varying matrix size with 1K batch size on single K40 GPU.

(c) Varying matrix size with 8K batch size on 8 K40 GPUs.

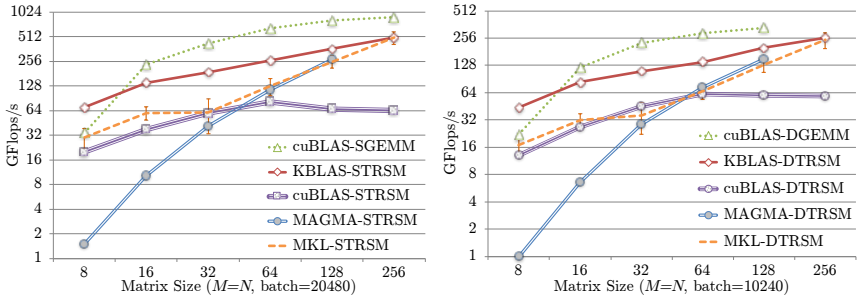
Fig. 8. Speedup gained by nested batching calls for batched *SYRK*, with SP and DP, running on NVIDIA K40 GPUs.

From now on, we report and evaluate the performance of our KBLAS implementations assuming all previously mentioned techniques have been activated.

7.3 Performance Comparisons of Batched BLAS Kernels

In this section, we compare our results against MAGMA, cuBLAS, MKL, and Kurzak et. al [Kurzak et al. 2016], whenever the relevant operations are available within such libraries. We report cuBLAS batched *GEMM* (shown as dashed curves) as a sustained upper-bound for the batched BLAS and DLA operations, as a more realistic bound than the theoretical peak performance of the hardware, given the low arithmetic intensity of our memory-bound batched kernels. These upper-bound curves are not always appropriate for small matrix sizes, due to the additional memory transfers required for the matrix-matrix multiply kernel, but they are still considered valid reference bounds. Furthermore, we do not check on the definiteness property of the matrices in KBLAS (and so is the implementation of Kurzak et. al [2016]), and we have disabled it in MAGMA for a fair comparison.

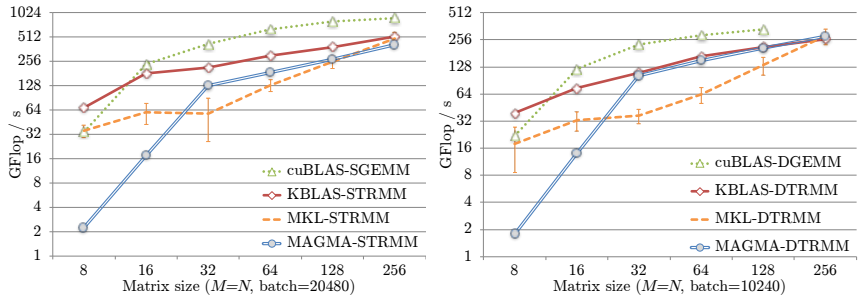
The overhead for such sanity check is rather minimal, and we intend to have it in KBLAS as well eventually.



(a) Batched *STRSM*. (b) Batched *DTRSM*.

Fig. 9. Performance comparison of KBLAS batched *TRSM* against MAGMA and cuBLAS (running on NVIDIA K40 GPU) and MKL (running on Broadwell 28 cores). cuBLAS DGEMM is shown as an upper-bound.

7.3.1 Single GPU. Figures 9, 10, and 11 show the performance comparisons of KBLAS batched *TRSM*, *TRMM*, and *SYRK* with both SP and DP against MAGMA, cuBLAS (when available) on single NVIDIA K40 GPU, and against MKL on Broadwell system. The performance gain for batched *TRSM* against MAGMA (Figures 9(a) and 9(b)) ranges from 1.3X and 1.4X (for relatively small sizes) up to 47X and 43X (for very small sizes), in SP and DP, respectively. Note that our KBLAS

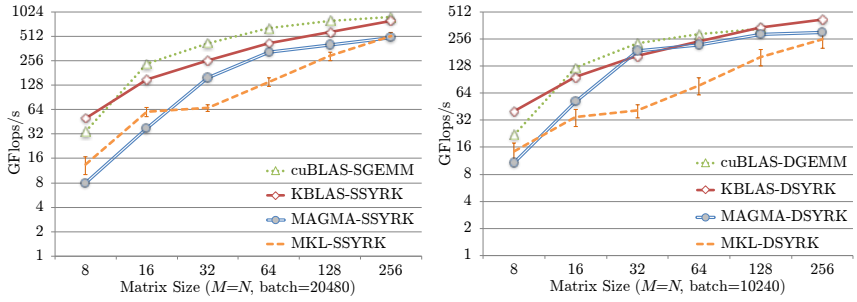


(a) Batched *STRMM*. (b) Batched *DTRMM*.

Fig. 10. Performance comparison of KBLAS batched *TRMM* against MAGMA and cuBLAS (running on NVIDIA K40 GPU) and MKL (running on Broadwell 28 cores).

RecTRSM implementation is in-place (IP), while MAGMA's implementation is out-of-place (OOP), thus requiring extra memory allocations for the output matrix and a workspace. The speedup ranges from 3.7X and 2.2X until 7.8X and 4.4X against the IP cuBLAS implementation and from 3X until 1.5X against the IP MKL implementation on Broadwell system, in SP and DP, respectively.

On the other hand, the gain in performance for batched *TRMM* (Figures 10(a) and 10(b)) is high for tiny sizes that are handled directly by KBLAS CUDA kernels, reaching up to 22X. However, since *TRMM* is basically implemented as a *GEMM* in MAGMA, speedups for matrix sizes beyond the recursion base (when recursion is invoked) are minimal and reach up to 1.68X and 1.2X, for SP



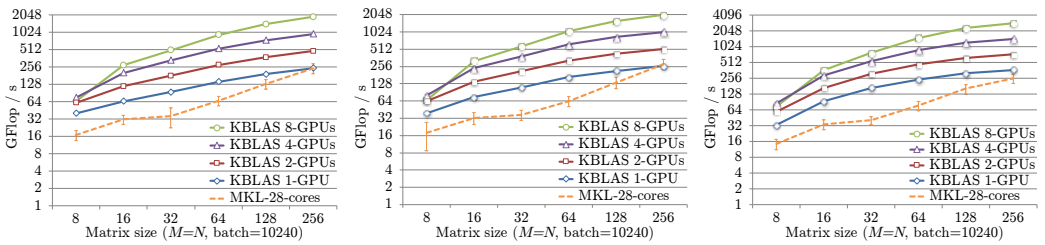
(a) Batched $SSYRK$. (b) Batched $DSYRK$.

Fig. 11. Performance comparison of KBLAS batched *SYRK* against MAGMA and cuBLAS (running on NVIDIA K40 GPU) and MKL (running on Broadwell 28 cores).

and DP, respectively. Speedups of KBLAS batched *TRMM* against MKL on Broadwell system range between 1.5X and 3.7X in both SP and DP.

Similar observations can be made about the batched *SYRK* operation, which is also implemented as a batched *GEMM* in MAGMA. Consequently, speedups for sizes below recursion base (32) are high, (Figures 11(a) and 11(b)) reaching up to 6.3X and 3.7X for SP and DP, respectively, and relatively lower for sizes beyond recursion base, reaching up to 1.6X and 1.3X for SP and DP, respectively. Speedups of KBLAS batched *SYRK* against MKL range from 1.5X and 1.6X up to 3.8X and 4X, for SP and DP, respectively.

7.3.2 Multiple GPUs. We benchmark and report performance of our implementations on multiple NVIDIA K40 GPUs. Data is assumed to reside on the devices main memories; consequently, data transfer time is not accounted for. We distribute the data and corresponding operations across devices equally. Figure 12 shows the performance of KBLAS batched *DSYRK*, *DTRSM* and *DTRMM* on multiple NVIDIA K40 GPUs, for DP arithmetic. We show the performance of same kernels on multi-core CPU with MKL as a reference. With equivalent work-load on the GPU devices, the plots demonstrate a decent strong scaling in performance on 1 to 8 GPU devices.



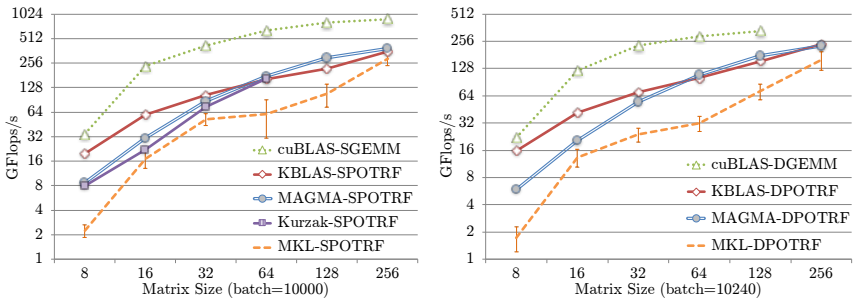
(a) Multi-GPUs batched $DTRSM$. (b) Multi-GPUs batched $DTRMM$. (c) Multi-GPUs batched $DSYRK$.

Fig. 12. Performance scalability of KBLAS batched BLAS operations with 10240 batch size running on multiple NVIDIA K40 GPUs.

7.4 Performance Comparisons of Batched High-Level Triangular DLA / LAPACK Operations

7.4.1 Single GPU. The series of Figures 13–16 show the performance comparisons of KBLAS batched high-level triangular DLA (*POTRF*, *POTRS*, *POSV*, *TRTRI*, *LAUUM*, *POTRI* and *POTI*) against MAGMA on single NVIDIA K40 GPU and MKL on Broadwell 28-core system.

The performance of KBLAS batched Cholesky factorization (*POTRF*) —shown in Figures 13(a) and 13(b)— achieves up to 2.6X and 2.9X speedups against MAGMA for very small sizes in SP and DP, respectively, and meets MAGMA's performance at sizes beyond 64. Figure 13(a) also shows the performance of batched *POTRF* in SP as implemented and reported by Kurzak et al [2016]. Kurzak's implementation employed an exhaustive tuning approach for each matrix size using the BEAST framework. Note that we extracted their *POTRF* performance from their corresponding publication (which reports results on a batch of size 10000 with SP only running on similar hardware), because the authors do not provide a software release. The performance gain against Kurzak's implementation is very close to that against MAGMA. Speedups of KBLAS batched *POTRF* against MKL on Broadwell system ranges between 2X for moderately small and 9X for very small matrix sizes in both SP and DP.

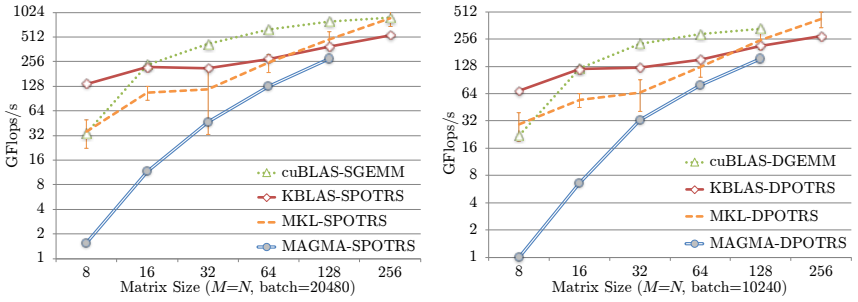


(a) Batched *SPOTRF*. (b) Batched *DPOTRF*.

Fig. 13. Performance comparison of KBLAS batched *POTRF* against MAGMA (running on NVIDIA K40 GPU) and MKL (running on Broadwell with 28 cores).

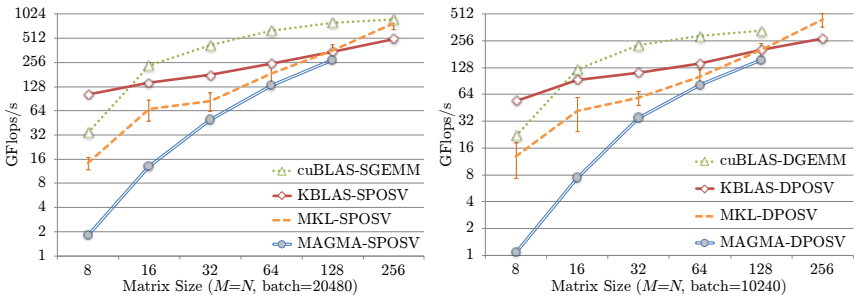
The positive definite triangular solve (*POTRS*) is effectively two consecutive triangular solves (*TRSM*s); consequently, it inherits the massive speedup brought by the *RecTRSM* kernel. The performance gain, as shown in Figures 14(a) and 14(b), ranges from 1.4X and 1.3X, for moderately small matrix sizes, up to 87X and 68X, for very small matrix sizes, in SP and DP, respectively. On the other hand, the larger CPU cache memory offers high level of data locality; consequently, the speedup of KBLAS batched *POTRS* over MKL on Broadwell system is less aggressive and ranges from 1.1X and 1.2X up to 3.8X and 2.3X for SP and DP, respectively. The MKL performance curve catches up with KBLAS at matrix sizes beyond 64.

Figure 15 shows performance comparisons of the solution to a system of linear equations with positive definite matrix (*POSV*), which decomposes into a Cholesky factorization (*POTRF*) and two consecutive triangular solves (*TRSM*s). Therefore, it inherits the corresponding performance gains. The speedups of KBLAS batched *POSV* against MAGMA ranges from 1.2X and 1.3X, for relatively small matrices, up to 56X and 49X for very small matrices, in SP and DP, respectively. Last but not least, Figure 16 draws the performance comparisons of other batched triangular DLA operations in KBLAS against MKL corresponding implementations (MAGMA does not provide support for these DLA operations). These additional DLA kernels are critical in the context of preconditioners for finite element discretizations of elliptic partial differential equations using balancing domain decomposition by constraints (BDDC) algorithm [Dohrmann 2003; Zampini 2016]. Thanks to the



(a) Batched *SPOTRS*. (b) Batched *DPOTRS*.

Fig. 14. Performance comparison of KBLAS batched *POTRS* against MAGMA (running on NVIDIA K40 GPU) and MKL (running on Broadwell with 28 cores).



(a) Batched *SPOSV*. (b) Batched *DPOSV*.

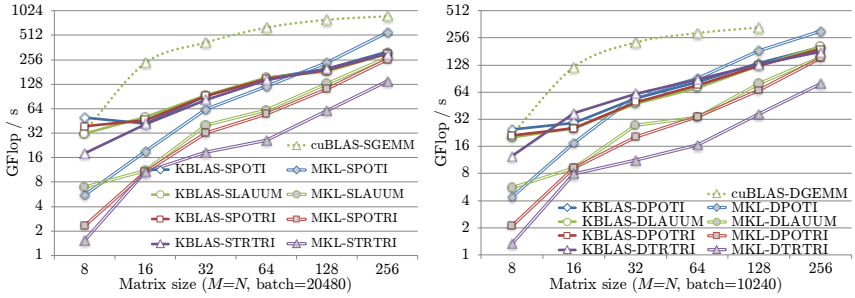
Fig. 15. Performance comparison of KBLAS batched *POSV* against MAGMA (running on NVIDIA K40 GPU) and MKL (running on Broadwell with 28 cores).

new technique optimizations described in Section 5, KBLAS batched DLA kernels run close to the sustained batched cuBLAS *GEMM* and outperform the MKL implementations for most of the matrix size ranges.

7.4.2 Multiple GPUs. Figure 17 shows the performance of KBLAS batched *DPOTRF*, *DPOTRS*, *DPOSV*, *DLAUUM*, *DTRTRI*, *DPOTRI* and *DPOTI* on multiple NVIDIA K40 GPUs. Similarly to the setup described in section 7.3.2, we distribute the load on devices equally and show MKL performance as a reference. Again, the plots show a decent strong scaling on 1 to 8 GPU devices.

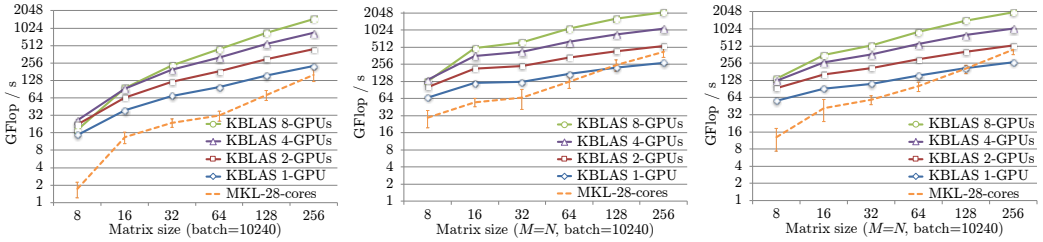
7.5 Portability Across GPU Architectures

Figure 18 demonstrates the performance speedup brought by KBLAS against MAGMA and cuBLAS (whenever available) across various generations of NVIDIA GPUs. Note that no extensive tuning was performed for any of the setups on various GPU architectures. The performance gain is significant for matrices with very small sizes and competitive for larger sizes, across all architectures and batched operations, and therefore, emphasizes that our optimization techniques are portable. It is noteworthy that the newly released Pascal P100 architecture brings some major changes in its core technical specifications, e.g., register file size, cores per SM, shared memory capacity, etc.

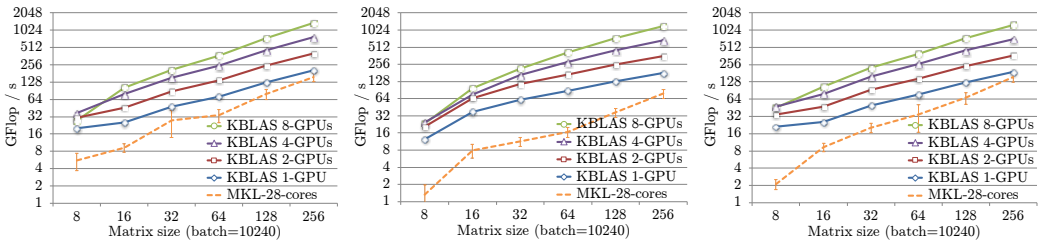


(a) Batched SP *LAUUM-TRTRI-POTRI-POTI*. (b) Batched DP *LAUUM-TRTRI-POTRI-POTI*.

Fig. 16. Performance comparison of KBLAS batched LAPACK operations against MAGMA (running on NVIDIA K40 GPU) and MKL (running on Broadwell with 28 cores).



(a) MultiGPUs batched *DPOTRF*. (b) MultiGPUs batched *DPOTRS*. (c) MultiGPUs batched *DPOSV*.



(d) MultiGPUs batched *DLAUUM*. (e) MultiGPUs batched *DTRTRI*. (f) MultiGPUs batched *DPOTRI*.

Fig. 17. Performance scalability of KBLAS batched high-Level DLA operations with 10240 batch size running on multiple NVIDIA K40 GPUs.

Minimal tuning on the register hosted computation technique of the KBLAS code may be necessary to further increase the performance gain for very small matrix sizes.

7.6 Performance Profiling

To further justify and assess the performance gains recorded in the previous section, we profile in detail our implementations in regards to data transfer, arithmetic intensity, and bandwidth saturation. We compare the profiling results to those on MAGMA. FLOPs and data transfer sizes are measured using NVIDIA profiler (*nvprof*). Figure 19 show the ratio of MAGMA performed FLOPs and memory transactions to that performed by KBLAS for the equivalent operations and

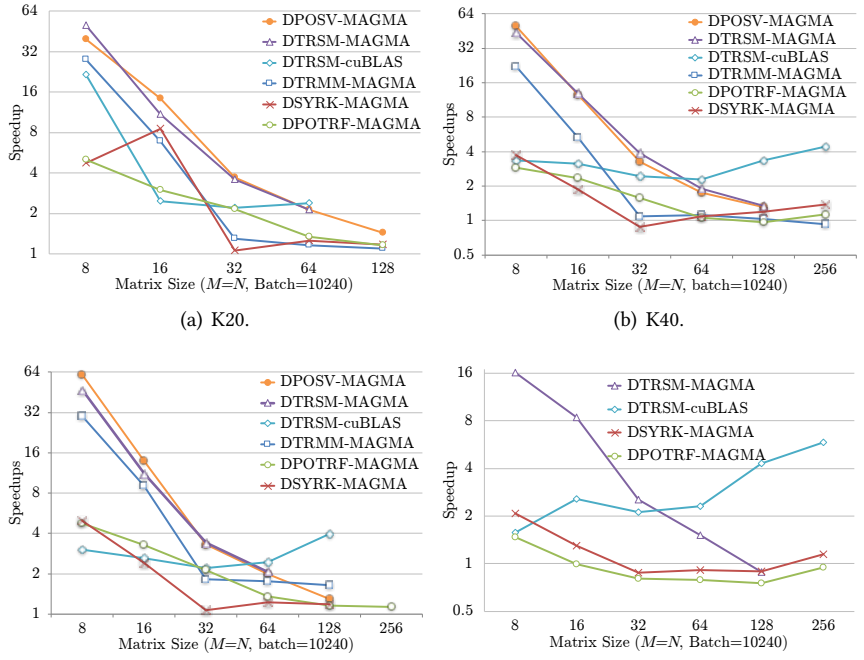


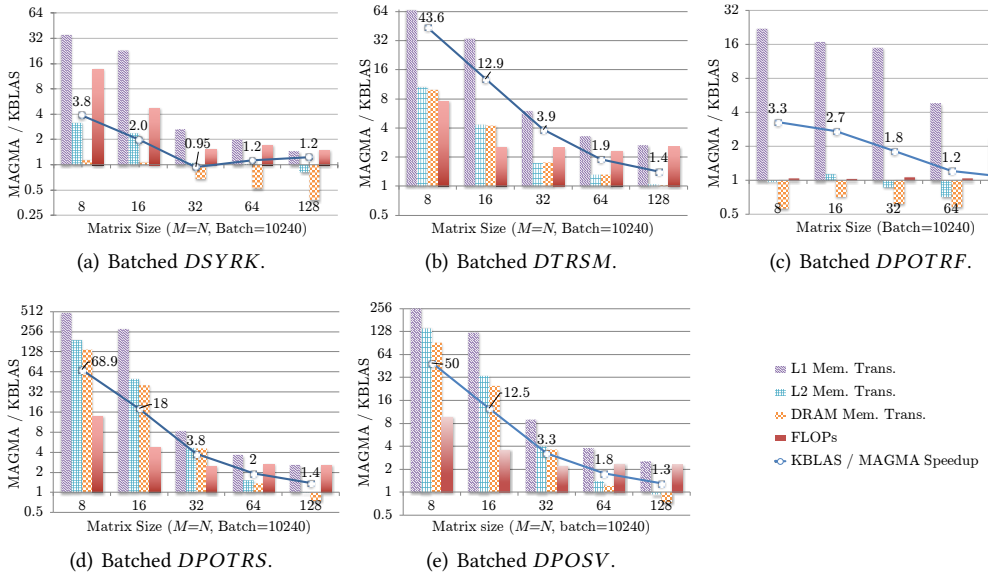
Fig. 18. Performance speedups of KBLAS batched BLAS and LAPACK operations against MAGMA and cuBLAS (whenever available) across various NVIDIA GPU generations.

matrix and batch sizes at all memory levels: L1 (including shared memory), L2, and global memory. The plots show that KBLAS is consistently performing many fewer flops than MAGMA since MAGMA inverts the diagonal blocks instead for the batched *TRSM*, *POSV* and *POTRS* kernels and operates on the full diagonal symmetric blocks for the batched *SYRK* kernel. In addition, KBLAS performs many fewer data transfers than MAGMA at the targeted matrix sizes in various triangular operations, thanks to the optimization techniques described in Section 5. The plots also show the recorded speedups by KBLAS against MAGMA, which very much align with the ratios of data transfer rates.

Figure 20(a) show the roofline performance model [Ofenbeck et al. 2014] of various KBLAS batched operations based on measured FLOPS and data transfer sizes, with a batch size of 10240 running on NVIDIA K40 GPU, on square matrices of sizes 128. The figure shows that performance of KBLAS implementations is very close to the sustained performance of the underlying hardware. Figure 20(b) shows the ratio of achieved bandwidth in GB/s of KBLAS batched operations using various matrix sizes to the sustained bandwidth of the used GPU device. Although FLOPs performance is way below the theoretical peak of the device, achieved bandwidth ranges from 60% to 80% of the sustained device bandwidth.

8 CONCLUSION AND FUTURE WORK

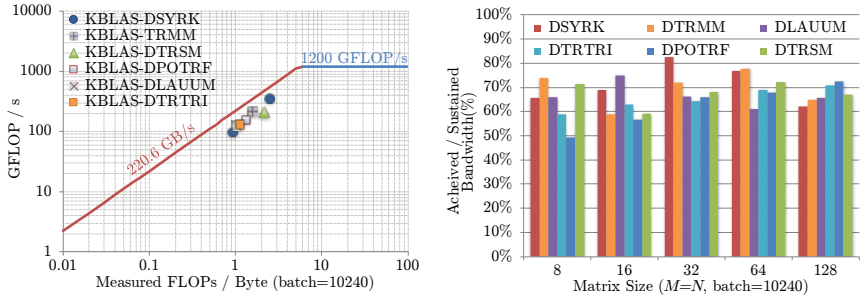
This paper presents a new set of high performance batched BLAS and LAPACK/DLA operations on GPUs with support for very small matrix sizes. Thanks to two-sided recursive algorithmic formulations and hardware/software-related optimizations (i.e., register pressure, kernel fusions and



(a) Batched *DSYRK*. (b) Batched *DTRSM*. (c) Batched *DPOTRF*.

(d) Batched *DPOTRS*. (e) Batched *DPOSV*.

Fig. 19. Profiling of FLOPs and memory transactions of MAGMA vs KBLAS batched operations with 10240 batch size running on NVIDIA K40m GPU.



(a) Roofline performance model of KBLAS batched operations in double precision and batched size of 10240 running on NVIDIA K40 GPU, on square matrices of size 128.

(b) Ratio of achieved to sustained bandwidth of various KBLAS batched operations in double precision on a K40 GPU with batch size of 10240.

Fig. 20

nested batching calls), these new GPU batched BLAS and LAPACK kernels outperform existing GPU and CPU implementations. They also require minimal performance tuning to extract performance on various GPU generations. These new kernels represent critical building blocks for designing efficient sparse direct solvers on hardware accelerators [Amestoy et al. 2011; Hénon et al. 2002] as well as in the context of machine learning applications with low-rank matrix approximations [Akbudak et al. 2017]. Future work includes extending these kernels to support non-uniform matrix sizes and a lightweight auto-tuning framework to optimize these batched operations transparently.

ACKNOWLEDGMENT

The authors would like to thank the NVIDIA for their hardware donations and remote access to their systems in the context of the NVIDIA GPU Research Center awarded to the Extreme Computing Research Center at KAUST.

REFERENCES

A. Abdelfattah, M. Baboulin, V. Dobrev, J. Dongarra, C. Earl, J. Falcou, A. Haidar, I. Karlin, Tz. Kolev, I. Masliah, and S. Tomov. 2016. High-performance tensor contractions for GPUs. *Procedia Comput. Sci.* 80 (2016), 108 – 118. DOI: http://dx.doi.org/10.1016/j.procs.2016.05.302 International Conference on Computational Science 2016, {ICCS} 2016, 6-8 June 2016, San Diego, California, {USA}.

A. Abdelfattah, J. Dongarra, D. Keyes, and H. Ltaief. 2012. Optimizing memory-bound SYMV kernel on GPU hardware accelerators. In *High Performance Computing for Computational Science - VECPAR 2012, 10th International Conference, Kobe, Japan, July 17-20, 2012, Revised Selected Papers*. 72–79. DOI: http://dx.doi.org/10.1007/978-3-642-38718-0_10

A. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra. 2016a. Fast Cholesky factorization on GPUs for batch and native modes in MAGMA. *J. Comput Sci.* (2016), –. DOI: http://dx.doi.org/10.1016/j.jocs.2016.12.009

A. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra. 2016b. On the development of variable size batched computation for heterogeneous parallel architectures. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1249–1258. DOI: http://dx.doi.org/10.1109/IPDPSW.2016.190

A. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra. 2016c. Performance, design, and autotuning of batched GEMM for GPUs. In *High Performance Computing - 31st International Conference, ISC High Performance 2016, Frankfurt, Germany, June 19-23, 2016, Proceedings (Lecture Notes in Computer Science)*, Julian M. Kunkel, Pavan Balaji, and Jack Dongarra (Eds.), Vol. 9697. Springer, 21–38. http://dx.doi.org/10.1007/978-3-319-41321-1

A. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra. 2016d. Performance tuning and optimization techniques of fixed and variable size batched Cholesky factorization on GPUs. *Procedia Comput. Sci.* 80 (2016), 119 – 130. DOI: http://dx.doi.org/10.1016/j.procs.2016.05.303 International Conference on Computational Science 2016, {ICCS} 2016, 6-8 June 2016, San Diego, California, {USA}.

A. Abdelfattah, D. Keyes, and H. Ltaief. 2016e. KBLAS: an optimized library for dense matrix-vector multiplication on GPU accelerators. *ACM Trans. Math. Software* 42, 3, Article 18 (May 2016), 31 pages. DOI: http://dx.doi.org/10.1145/2818311

K. Akbudak, H. Ltaief, A. Mikhalev, and D. Keyes. 2017. Tile low rank Cholesky factorization for climate/weather modeling applications on manycore architectures. *International Supercomputing Conference* (2017).

P. Amestoy, A. Buttari, I. Duff, A. Guermouche, J.Y. L'Excellent, and B. Uçar. 2011. *MUMPS.* Springer US, Boston, MA, 1232–1238. DOI: http://dx.doi.org/10.1007/978-0-387-09766-4_204

B. S. Andersen, F. Gustavson, A. Karaivanov, M. Marinova, J. Waśniewski, and P. Yalamov. 2001. LAWRA linear algebra with recursive algorithms. In *Applied Parallel Computing. New Paradigms for HPC in Industry and Academia: 5th International Workshop, PARA 2000 Bergen, Norway, June 18–20, 2000 Proceedings*, T. Sørevik, F. Manne, A. H. Gebremedhin, and Randi Moe (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 38–51. DOI: http://dx.doi.org/10.1007/3-540-70734-4_7

E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. 1999. *LAPACK User's Guide* (3rd ed.). Society for Industrial and Applied Mathematics, Philadelphia.

BEAST. 2017. Bench-testing Environment for Automated Software Tuning. Innovative Computing Laboratory, University of Tennessee. (2017). Available at http://icl.cs.utk.edu/beast//.

W. H. Boukaram, G. Turkiyyah, H. Ltaief, and D. Keyes. 2017. Batched QR and SVD algorithms on GPUs with applications in hierarchical matrix compression. *Submitted to J. Parallel Comput., Special Edition* (2017).

A. Charara, D. Keyes, and H. Ltaief. 2016a. A framework for dense triangular matrix kernels on various manycore architectures. *Submitted to Concurr. Comput.: Prac. Experience* (2016). http://hdl.handle.net/10754/622077

A. Charara, H. Ltaief, and D. Keyes. 2016b. Redesigning triangular dense matrix computations on GPUs. In *Euro-Par 2016: Parallel Processing: 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings*, P. F. Dutot and D. Trystram (Eds.). Springer International Publishing, Cham, 477–489. DOI: http://dx.doi.org/10.1007/978-3-319-43659-3_35

C. R. Dohrmann. 2003. A preconditioner for substructuring based on constrained energy minimization. *SIAM J. Sci. Comput.* 25, 1 (Jan. 2003), 246–258. DOI: http://dx.doi.org/10.1137/S1064827502412887

T. Dong. 2015. *Batched linear algebra problems on GPU accelerators.* Ph.D. Dissertation. The University of Tennessee, Knoxville.

T. Dong, A. Haidar, P. Luszczek, J. A. Harris, S. Tomov, and J. Dongarra. 2014a. LU factorization of small matrices: accelerating batched DGETRF on the GPU. In *High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace*

Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICESS), 2014 IEEE Intl Conf on. IEEE, 157–160. DOI: http://dx.doi.org/10.1109/HPCC.2014.30

T. Dong, A. Haidar, S. Tomov, and J. Dongarra. 2014b. A fast batched Cholesky factorization on a GPU. In *2014 43rd International Conference on Parallel Processing*. IEEE Computer Society, 432–440. DOI: http://dx.doi.org/10.1109/ICPP.2014.52

J. Dongarra, M. Abalenkovs, A. Abdelfattah, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, I. Yamazaki, and A. YarKhan. 2015. Parallel programming models for dense linear algebra on heterogeneous systems. *Supercomput. Front. Innov.: Int. J.* 2, 4 (March 2015), 67–86. DOI: http://dx.doi.org/10.14529/jsfi150405

J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. 1988. An extended set of Fortran basic linear algebra subprograms. *ACM Trans. Math. Software* 14 (1988), 1–17.

D. Eliahu, O. Spillinger, A. Fox, and J. Demmel. 2015. *FRPA: A framework for recursive parallel algorithms.* Master's thesis. EECS Department, University of California, Berkeley. http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-28.html

E. Elmroth, F. Gustavson, I. Jonsson, and B. Kågström. 2004. Recursive blocked algorithms and hybrid data structures for dense matrix library software. *SIAM Rev.* 46, 1 (2004), 3–45. DOI: http://dx.doi.org/10.1137/S0036144503428693

E. Elmroth and F. G. Gustavson. 2000. Applying recursion to serial and parallel QR factorization leads to better performance. *IBM J. Res. Dev.* 44, 4 (July 2000), 605–624. DOI: http://dx.doi.org/10.1147/rd.444.0605

K. Goto and R. Van De Geijn. 2008. High-performance implementation of the level 3 BLAS. *ACM Trans. Math. Software* 35, 1, Article 4 (July 2008), 14 pages. DOI: http://dx.doi.org/10.1145/1377603.1377607

W. Hackbusch. 1999. A sparse matrix arithmetic based on $\mathcal{H}$-matrices. Part I: Introduction to $\mathcal{H}$-matrices. *Computing* 62, 2 (May 1999), 89–108. DOI: http://dx.doi.org/10.1007/s006070050015

W. Hackbusch and B. N. Khoromskij. 2000. A sparse $\mathcal{H}$-matrix arithmetic. Part II: Application to multi-dimensional problems. *Computing* 64, 1 (2000), 21–47. DOI: http://dx.doi.org/10.1007/PL00021408

A. Haidar, T. Dong, P. Luszczek, S. Tomov, and J. Dongarra. 2015a. Batched matrix computations on hardware accelerators based on GPUs. *Int. J. High Perform. Comput. Appl.* 29, 2 (May 2015), 193–208. DOI: http://dx.doi.org/10.1177/1094342014567546

A. Haidar, T. Dong, P. Luszczek, S. Tomov, and J. Dongarra. 2015b. Optimization for performance and energy for batched matrix computations on GPUs. In *Proceedings of the 8th Workshop on General Purpose Processing using GPUs, GPGPU@PPoPP 2015, San Francisco, CA, USA, February 7, 2015*, David R. Kaeli and John Cavazos (Eds.). ACM, 59–69. http://dl.acm.org/citation.cfm?id=2716282

A. Haidar, T. Dong, P. Luszczek, S. Tomov, and J. Dongarra. 2015c. Towards batched linear solvers on accelerated hardware platforms. *SIGPLAN Not.* 50, 8 (Jan. 2015), 261–262. DOI: http://dx.doi.org/10.1145/2858788.2688534

A. Haidar, T. T. Dong, S. Tomov, P. Luszczek, and J. Dongarra. 2015d. A framework for batched and GPU-resident factorization algorithms applied to block householder transformations. In *High Performance Computing: 30th International Conference, ISC High Performance 2015, Frankfurt, Germany, July 12-16, 2015, Proceedings*. Springer, 31–47.

A. Heinecke, G. Henry, M. Hutchinson, and H. Pabst. 2016. LIBXSMM: accelerating small matrix multiplications by runtime code generation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016*, John West 0001 and Cherri M. Pancake (Eds.). ACM, 84. http://dl.acm.org/citation.cfm?id=3014904

P. Hénon, P. Ramet, and J. Roman. 2002. PASTIX: A high-performance parallel direct solver for sparse symmetric positive definite systems. *Parallel Comput.* 28, 2 (2002), 301–321.

F. D. Igual, G. Quintana-Ort, and R. A. van de Geijn. 2012. Level-3 BLAS on a GPU: Picking the low hanging fruit. *AIP Conf. Proc* 1504, 1 (2012), 1109–1112. DOI: http://dx.doi.org/10.1063/1.4772121 arXiv:http://aip.scitation.org/doi/pdf/10.1063/1.4772121

Intel. 2017. Math Kernel Library (MKL). (2017). Available at http://software.intel.com/en-us/articles/intel-mkl.

B. Kågström. 2006. Management of deep memory hierarchies – recursive blocked algorithms and hybrid data structures for dense matrix computations. In *Applied Parallel Computing. State of the Art in Scientific Computing: 7th International Workshop, PARA 2004, Lyngby, Denmark, June 20-23, 2004. Revised Selected Papers*, Jack Dongarra, Kaj Madsen, and Jerzy Waśniewski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 21–32. DOI: http://dx.doi.org/10.1007/11558958_3

B. Kågström, P. Ling, and C. van Loan. 1998. GEMM-based level 3 BLAS: high-performance model implementations and performance evaluation benchmark. *ACM Trans. Math. Software* 24, 3 (1998), 268–302. DOI: http://dx.doi.org/10.1145/292395.292412

J. King, S. Yakovlev, Z. Fu, R. M. Kirby, and S. J. Sherwin. 2014. Exploiting batch processing on streaming architectures to solve 2D elliptic finite element problems: a Hybridized Discontinuous Galerkin (HDG) case study. *J. Sci. Comput.* 60, 2 (2014), 457–482. DOI: http://dx.doi.org/10.1007/s10915-013-9805-x

J. Kurzak, H. Anzt, M. Gates, and J. Dongarra. 2016. Implementation and tuning of batched Cholesky factorization and solve for NVIDIA GPUs. *IEEE Trans. Parallel and Distrib. Syst.* 27, 7 (July 2016), 2036–2048. DOI: http://dx.doi.org/10.1109/TPDS.2015.2481890

MAGMA. 2017. Matrix Algebra on GPU and Multicore Architectures. Innovative Computing Laboratory, University of Tennessee. (2017). Available at http://icl.cs.utk.edu/magma/.

I. Masliah, A. Abdelfattah, A. Haidar, S. Tomov, M. Baboulin, J. Falcou, and J. Dongarra. 2016. High-performance matrix-matrix multiplications of very small matrices. In *Euro-Par 2016: Parallel Processing: 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings*, P. F. Dutot and D. Trystram (Eds.). Springer International Publishing, Cham, 659–671. DOI: http://dx.doi.org/10.1007/978-3-319-43659-3_48

NVIDIA. 2017a. The CUDA Basic Linear Algebra Subroutines (cuBLAS). (2017). Available at http://developer.nvidia.com/cublas.

NVIDIA. 2017b. The CUDA Basic Linear Algebra Subroutines (cuBLAS). (2017). Available at http://docs.nvidia.com/cuda/cublas/index.html#appendix-acknowledgements.

NVIDIA. 2017c. The CUDA solver library (cuSOLVER). (2017). Available at http://developer.nvidia.com/cusolver.

NVIDIA. 2017d. CUDA C Programming Guide. (2017). http://docs.nvidia.com/cuda/cuda-c-programming-guide/

G. Ofenbeck, R. Steinmann, V. Caparros, D. G. Spampinato, and M. Pschel. 2014. Applying the roofline model. In *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*. IEEE Computer Society, 76–85. DOI: http://dx.doi.org/10.1109/ISPASS.2014.6844463

V. Oreste, M. Fatica, N. A. Gawande, and A. Tumeo. 2013. Power/performance trade-offs of small batched LU based solvers on GPUs, Lecture Notes in Computer Science (Ed.). *19th International Conference on Parallel Processing, Euro-Par* 8097 (August 2013).

E. Peise and P. Bientinesi. 2016. Recursive algorithms for dense linear algebra: the ReLAPACK collection. *ArXiv e-prints* (Feb. 2016). arXiv:cs.MS/1602.06763

SuiteSparse. 2017. A suite of sparse matrix software. (2017). Available at http://faculty.cse.tamu.edu/davis/SuiteSparse/.

S. Zampini. 2016. PCBDDC: a class of robust dual-primal methods in PETSc. *SIAM J. Sci. Comput.* 38, 5 (2016), S282–S306.

Received March 2017; revised March 2017; accepted March 2017