






Tile Low-Rank GEMM Using Batched Operations on GPUs

Ali Charara^(✉), David Keyes,
and Hatem Ltaief



Extreme Computing Research Center, Division of Computer, Electrical,
and Mathematical Sciences and Engineering, King Abdullah University of Science
and Technology, Thuwal Jeddah 23955, Kingdom of Saudi Arabia
{Ali.Charara,David.Keyes,Hatem.Ltaief}@kaust.edu.sa

Abstract. Dense General Matrix-Matrix (GEMM) multiplication is a core operation of the Basic Linear Algebra Subroutines (BLAS) library, and therefore, often resides at the bottom of the traditional software stack for many scientific applications. In fact, chip manufacturers give a special attention to the GEMM kernel implementation since this is exactly where most of the high-performance software libraries extract hardware performance. With the emergence of big data applications involving large data-sparse, hierarchically low-rank matrices, the off-diagonal tiles can be compressed to reduce the algorithmic complexity and the memory footprint. The resulting tile low-rank (TLR) data format is composed of small data structures, which retain the most significant information for each tile. However, to operate on low-rank tiles, a new GEMM operation and its corresponding API have to be designed on GPUs so that the data sparsity structure of the matrix can be exploited while leveraging the underlying TLR compression format. The main idea consists of aggregating all operations into a single kernel launch to compensate for their low arithmetic intensities and to mitigate the data transfer overhead on GPUs. The new TLR-GEMM kernel outperforms the cuBLAS dense batched GEMM by more than an order of magnitude and creates new opportunities for TLR advanced algorithms.

Keywords: Hierarchical low-rank matrix computations
Matrix multiplication · GEMM · High performance computing
GPU Computing · KBLAS

1 Introduction

With the convergence of the third and fourth paradigms (i.e., simulation and big data), large-scale scientific applications, such as climate/weather forecasting [31], require a profound redesign to reduce the memory footprint as well as the overall algorithmic complexity. When considering multi-dimensional problems, with a large number of unknowns, n , the resulting covariance matrix may

render its structure fully dense. To overcome the curse of dimensionality without violating the fidelity of the physical model, application developers rely on approximation methods, which drastically reduce the constraints on the memory footprint and the algorithmic complexity. For instance, hierarchical matrices or \mathcal{H} -matrices have been recently resurrected for high-performance computing [29, 32] as a potential algorithmic solution to tackle the aforementioned challenge. Because of their inherent recursive formulations, they are not amenable to massively parallel hardware systems such as GPUs.

We have designed, investigated, and implemented, on x86 shared-memory systems within the HiCMA library, an approximation method that exploits the natural data sparsity of the off-diagonal tiles while exposing parallelism to the fore [8]. Based on the tile low-rank (TLR) data format, the off-diagonal tiles of a dense covariance matrix are compressed up to a specific accuracy threshold, without compromising the model fidelity. The resulting data structure, much smaller than the original dense tiles, represents the new building blocks to pursue the matrix computations. Since main memory is a scarce resource on GPUs, TLR should enable solving even larger GPU-resident problems and eventually fall back to out-of-core algorithms. Although this TLR scenario may look utterly GPU friendly and more compliant, there are still some lingering performance bottlenecks. Indeed, decomposing an off-diagonal low-rank matrix problem into tasks may lead to a computational mismatch between the granularity of the task and the computational power of the underlying hardware. In particular, heavily multi-threaded accelerators such as NVIDIA GPUs need to maintain high occupancy and would require developers to move away from the current model, where tasks occupy all hardware computing elements, and, instead, simultaneously execute multiple smaller tasks, each spanning across a subset of hardware resources. This mode of operation, called *batched execution* [19], executes many smaller operations in parallel to make efficient use of the hardware and its instruction-level parallelism. To our knowledge, this work introduces the first TLR general matrix-matrix multiplication (TLR-GEMM) operating on data sparse matrix structures using GPU hardware accelerators. Our research contribution lies at the intersection of two concurrent and independent efforts happening in the scientific community: \mathcal{H} -matrix and batched operations for BLAS/LAPACK. Our TLR-GEMM leverages the current batched execution kernels in BLAS and LAPACK to support the matrix-matrix multiplication operation, which is perhaps one of the most important operations for high-performance numerical libraries, in TLR data format. In this paper, we focus on compressing data-sparse matrices and operating on them with uniform rank sub-blocks. Non-uniform rank compression and operation is a subject under investigation and beyond the scope of this paper. Our TLR-GEMM implementation is available in the open source KBLAS library maintained at <https://github.com/ecrc/kblas-gpu>.

The remainder of the paper is organized as follows: Sect. 2 presents related work and details our research contributions; Sect. 3 recalls the batched linear algebra community effort and gives a general overview of the hierarchical low-rank matrix approximation; Sect. 4 introduces the new TLR-GEMM and its vari-

ants; the implementation details of the various TLR-GEMM kernels are given in Sect. 5; Sect. 6 assesses the accuracy and performance of TLR-GEMM on the latest NVIDIA GPU hardware generation and compares it to the state-of-the-art high-performance batched dense GEMM, as implemented in [2]; we conclude in Sect. 7.

2 Related Work

The general matrix-matrix multiplication (GEMM) operation is the primitive kernel for a large spectrum of scientific applications and numerical libraries. GEMM has been optimized on various hardware vendors for large matrix sizes and constitutes the basic reference for Level-3 BLAS [18] operations and their usage in dense linear algebra algorithms. With the need to solve multicomponent partial differential equations, the resulting sparse matrix may have a dense block structure. The block size is relatively small and corresponds to the number of degrees of freedom per mesh element. The blocks are usually stored in a compressed block column/row data format. Matrix computations are then performed on these independent blocks by means of batched operations. For instance, batched dense matrix-vector multiplication is employed in sparse iterative solvers for reservoir simulations [6], while batched dense LAPACK factorizations [15] are required in sparse direct solvers for the Poisson equation using cyclic reduction [17]. Moreover, with the emergence of artificial intelligence, batched dense GEMM of even smaller sizes are needed in tensor contractions [4, 5, 33] and in deep learning frameworks [3, 27, 28]. To facilitate the adoption of all these efforts, a new standard has been proposed to homogenize the various batched API [19]. While the literature is rich in leveraging batched executions for dense and sparse linear algebra operations on x86 and hardware accelerators [4–6, 14, 15, 19], this trend has faced challenges and has not penetrated data-sparse applications yet, involving large hierarchically low-rank matrices (i.e., \mathcal{H} -matrices). In fact, there are three points to consider when designing batched operations for \mathcal{H} -matrices. First, there should be an efficient batched compression operation for \mathcal{H} -matrices on x86 and on hardware accelerators. Second, the inherent recursive formulation of \mathcal{H} -matrix resulting from nested dissections should be replaced, since it is not compliant with batched operations. Third, strong support is eventually required to handle batched matrix operations on the data-sparse compressed format, such as \mathcal{H}^2 , Hierarchically Semi-Separable representation (HSS), and the Hierarchical Off-Diagonal Low-Rank (HODLR) matrix. More recently, a block/tile low-rank (TLR) compressed format has been introduced on x86 [8, 11], which further exposes parallelism by flattening the recursion trees. The TLR data format may engender new opportunities and challenges for batched matrix compression and operation kernels on advanced SIMD architectures. Moreover, an effective implementation of the randomized SVD [26] for \mathcal{H}^2 matrix compression has been ported to hardware accelerators [14]. The aforementioned three bottlenecks may now be relieved to deploy TLR matrix computations on GPUs.

3 Background

Batched Dense Linear Algebra. GPUs are massively parallel devices optimized for high SIMD throughput. Numerical kernels with low arithmetic intensity may still take advantage of the high memory bandwidth, provided they operate on large data structures to saturate the bus bandwidth. When operating on relatively small workloads on GPUs, the GPU overheads are twofold: (1) the overhead of moving the data from the CPU to the GPU memory through the thin PCIe pipe may be not worthwhile, and (2) the overhead of launching the kernels is not compensated by the low computation complexity. High-performance frameworks for batched operations [1, 2, 15] attempt to overcome both challenges by stitching together multiple operations occurring on independent data structures. This batched mode of execution increases hardware occupancy to attain higher sustained peak bandwidth while launching a single kernel to remove the kernel launch overheads all together. Figure 1(a) sketches batched operations of small dense GEMM operations $C += A \times B$. Following the same community effort for the legacy BLAS, a community call for standardizing the batched API [19] has been initiated, gathering hardware vendors and researchers. This standardization effort enhances software development productivity, while the batched API gains maturity in the scientific community.

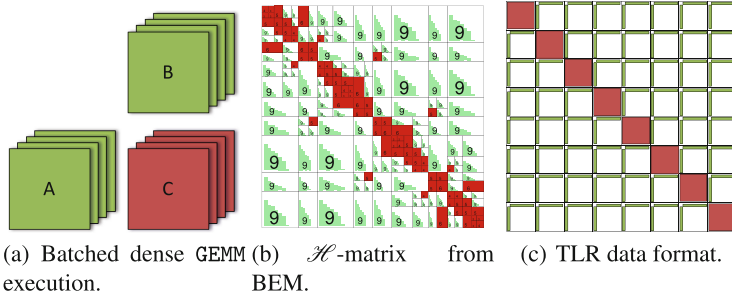


Fig. 1. Consolidating batched operations and \mathcal{H} -matrix through TLR data format. (Color figure online)

Hierarchical Low-Rank Matrix Computations. The hierarchically low-rank matrix, or \mathcal{H} -matrix [21, 23–25, 34] is a low-rank block approximation of a dense (sub)matrix, whose off-diagonal blocks may be each represented by an outer product of rectangular bases obtained from compression, e.g., via orthogonal transformations with singular value decomposition. An \mathcal{H} -matrix captures the most significant singular values and their corresponding singular vectors up to an application-dependent accuracy threshold for each block. Figure 1(b) highlights the structure of an \mathcal{H} -matrix resulting from a boundary element method. Each off-diagonal green block has been approximated to a similar rank up to 9. The red blocks are dense blocks and are mostly located around the diagonal structure of the matrix. This data sparse structure may be exposed by performing

nested dissection after proper reordering and partitioning. This recursive formulation allows traversing low-rank off-diagonal blocks and compressing them using an adequate data storage format such as \mathcal{H} [29], Hierarchical Off-Diagonal Low-Rank (HODLR) [12], \mathcal{H} [22], Hierarchically Semi-Separable representation (HSS) [10,32], and \mathcal{H}^2 [13]. All these data compression formats belong to the family of \mathcal{H} -matrices and can be differentiated by the type of their bases i.e., nested (HSS and \mathcal{H}^2) or non-nested (\mathcal{H} and HODLR), in addition to the type of admissibility conditions, i.e., strong (\mathcal{H} and \mathcal{H}^2) or weak (HODLR and HSS). Each of these compression data formats exhibits different algorithmic complexities and memory footprint theoretical upper-bounds. The tile low-rank (TLR) [8,11] data format is another case of the \mathcal{H} data format with non-nested bases and strong admissibility conditions. The matrix is logically split into tiles, similar to the tile algorithms from the PLASMA library [7]. The off-diagonal tiles may then be compressed using the rank-revealing QR once the whole dense matrix has been generated [11] or *on-the-fly* using the randomized singular value decomposition [26] while performing the matrix computations [8]. Figure 1(c) shows an illustration of such a TLR matrix composed of an 8×8 logical tile. Owing to its simplicity, TLR permits flattening the inherent recursive formulation. Although TLR may not provide such optimal theoretical bounds as for the nested-basis data formats, regarding algorithmic complexities and memory footprint, it is very amenable to advanced performance implementations and optimizations.

The main objective of this paper is to consolidate the three messages conveyed by the sketches of Fig. 1, i.e., batched operations (including matrix compression and computations), \mathcal{H} -matrix applications and TLR data format. This consolidation is the crux of the TLR-GEMM implementation on GPUs.

4 Design of Tile Low-Rank GEMM Kernels

This section describes the design of the TLR-GEMM kernel and identifies its variants using a bottom-up approach: from the single GEMM kernel operating on low-rank data format to the corresponding batched operations, and then all the way up to the actual TLR-GEMM driver.

Low-Rank Data Format. Low-rank approximation consists of compressing a dense matrix X of dimensions m -rows and n -columns and representing it as the product of two tall and skinny matrices, such that $X = X_u \times X_v^T$, with X_u and X_v of dimensions (m -rows, k -columns) and (n -rows, k -columns), respectively, and k the rank of X . The choice for the compression algorithms is typically Rank-Revealing QR (RRQR), adaptive cross-approximation (ACA), or (randomized) singular value decomposition (SVD), etc. The randomized Jacobi-based SVD [26] maps well on SIMD architectures because it does not involve pivoting nor element sweeping, as in the RRQR and ACA methods, respectively. It is perhaps the most optimized compression algorithm on GPUs, as implemented in [14].

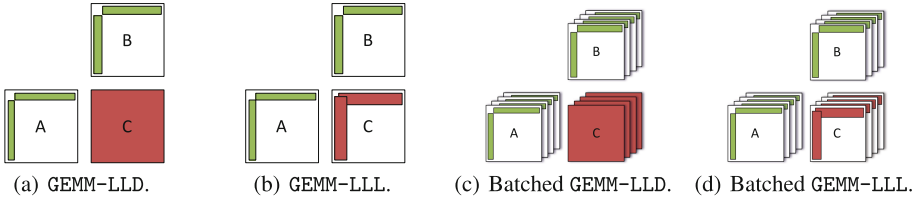


Fig. 2. Illustrating single and batched low-rank GEMM variants.

Low-Rank GEMM Variants. We identify four possible variants, based on the input data format of the involved matrices A, B , and C . We use the following notation: $\text{GEMM}-T_A T_B T_C$ is the GEMM kernel for a given input type (Dense or Low-rank) for the matrices A, B , and C . The variants are as follows: (1) either of A or B in low-rank format, C in dense format: GEMM-DLD , or GEMM-LDD , (2) either of A or B in low-rank format, C in low-rank format: GEMM-DLL , or GEMM-LDL , (3) A and B in low-rank format, C in dense format: GEMM-LLD , as illustrated in Fig. 2(a), and, (4) A, B and C in low-rank format: GEMM-LLL , as illustrated in Fig. 2(b). We focus solely on the last two variants in this paper, i.e., GEMM-LLD and GEMM-LLL , since they are the bases for supporting the Schur complement calculation and for matrix factorization, in the context of sparse direct solvers [11] and data-sparse matrix solvers [8], respectively. In fact, the other possible variants, i.e., when both A and B are in dense format and C in dense (GEMM-DDD) or in low-rank format (GEMM-DDL), are not considered because the first is the actual legacy GEMM operation, and the second is more expensive than regular GEMM in terms of flops.

Batched Low-Rank GEMM. We can then derive the batched low-rank GEMM routines from their corresponding single low-rank GEMM routines. The new batched low-rank GEMM kernels are now defined as single kernels, i.e., Batched-GEMM-LLD and Batched-GEMM-LLL , which simultaneously execute independent GEMM-LLD and GEMM-LLL operations, as demonstrated in Fig. 2(c) and (d), respectively. This batched kernel is used as a building block for the main driver performing the TLR-GEMM on large TLR matrices, as described in the following paragraph.

TLR-GEMM (driver). In the driver of the TLR-GEMM operation, the data-sparse matrices A, B , and C are subdivided into a grid of tiles, where each tile may individually be compressed into low-rank data form, as illustrated in Fig. 3. Indeed, Fig. 3(a) and (b) represent the TLR-GEMM operation, when T_C is tile dense or tile low-rank, respectively. In a standard GEMM operation, each tile of the matrix C is updated by an inner-product composed of a sequence of pair-wise GEMM operations of its corresponding row of tiles from matrix A and column of tiles from matrix B . However, when dealing with TLR data format, since the workload of each low-rank tile is too small to saturate a modern GPU with sufficient work, concurrent processing of these independent low-rank tiles inner-products is necessary to increase the GPU occupancy. To overcome this challenge, we need to process these inner-product low-rank GEMM calls in a batched mode.

However, available batched GEMM routines assume the batched operation is a primitive CUDA kernel rather than a sequence of calls; thus, we re-formulate the set of inner-products into a set of successive outer-products by means of loop re-ordering. Each outer-product is a call to batched GEMM-LLD or GEMM-LLL routine which updates all tiles of matrix C in parallel. This process is repeated nt times, nt being the number of tiles in a row of matrix A or a column of matrix B , as illustrated in Fig. 3 for both variants of TLR-GEMM.

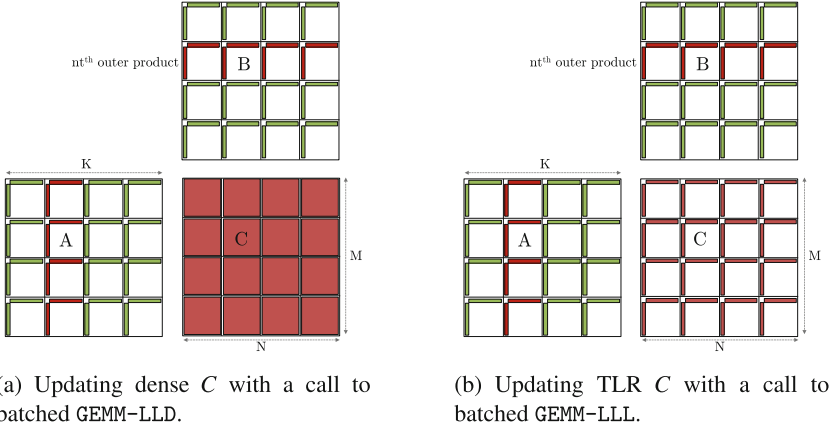


Fig. 3. Processing TLR-GEMM as a series of nt outer-products using batched GEMM-LLD or GEMM-LLL kernels.

5 Implementation Details

Update Dense C: GEMM-LLD. This operation is performed as a sequence of three small GEMM calls. Assuming matrices C and A are of m -rows, C and B of n -columns, A and B of k -columns and k -rows respectively, and A and B are of ranks r_a , and r_b , respectively, the operation $C = \alpha A \times B + \beta C$ is equivalent to $C = \alpha A_u \times A_v^T \times B_u \times B_v^T + \beta C$.

Update Low-Rank C: GEMM-LLL. We describe the second variant of the low-rank GEMM operation when updating C in low-rank format, as outlined in the corresponding Algorithm 1. In fact, this algorithm corresponds to the randomized SVD, as described in [26]. We assume the non-transpose case for both matrices A and B . The matrix-matrix multiplication $C = \alpha A \times B + \beta C$ involves two sub-stages, where matrices A, B , and C are represented by their low-rank format $(A_u, A_v), (B_u, B_v)$, and (C_u, C_v) , respectively. The first stage consists of the multiplication of low-rank matrices A and B , as shown in steps 2 – 3 of Algorithm 1. The second stage highlights the final addition of the intermediate matrix with the low-rank matrix C , as demonstrated in steps 4 – 6. This second stage produces low-rank $\hat{C} = \hat{C}_u \times \hat{C}_v$ with bloated rank \hat{r}_c . As such,

Algorithm 1. GEMM-LLL($m, n, k, \alpha, A_u, A_v, r_a, B_u, B_v, r_b, \beta, C_u, C_v, r_c, W_a, W_b$).

Input: A_u, A_v, B_u, B_v, C_u , and C_v are $m \times r_a, k \times r_a, k \times r_b, n \times r_b, m \times r_c$, and $n \times r_c$ matrices, respectively. W_a and W_b are workspaces of size $r_a \times r_b$ and $r_a \times n$ respectively.

```

1 Setup work-space buffer.;
2 //Multiply A and B;
3 GEMM(Trans, noTrans, r_a, r_b, k, alpha, A_v, B_u, 0, W_a):  W_a ← alpha A_v^T × B_u;
4 GEMM(noTrans, Trans, r_a, n, r_b, 1, W_a, B_v, 0, W_b):  W_b ← W_a × B_v^T;
5 //Add to C r_c ← r_c + r_a;
6 C_u ← C_u | A_u; // Concat C_u and A_u into one buffer
7 C_v ← beta(C_v | W_b); // Concat C_v and W_b into one buffer, and scale by beta
8 //Recompression of C_u and C_v;
9 GEQRF(m, r_c, C_u, tau): C_u ← QR(C_u); // factorize C_u
10 GEQRF(n, r_c, C_v, tau): C_v ← QR(C_v); // factorize C_v
11 R_u = upper triangular of C_u;
12 R_v = upper triangular of C_v;
13 GEMM(noTrans, Trans, r_c, r_c, 1, R_u, R_v, 0, R): R = R_u × R_v^T;
14 GESVD(r_c, r_c, R, S, R_u, R_v);
15 Pick r_c based on threshold of accuracy or maximum rank.;
16 Scale R_v by S;
17 ORGQR(m, r_c, C_u, tau); // extract Q factors
18 ORGQR(n, r_c, C_v, tau); // extract Q factors
19 GEMM(noTrans, noTrans, m, r_c, r_c, 1, C_u, R_u, 0, C_u): C_u = C_u × R_u; // final C_u
20 GEMM(noTrans, noTrans, n, r_c, r_c, 1, C_v, R_v, 0, C_v): C_v = C_v × R_v; // final C_v
21 return;

```

low-rank matrix addition, as described by Grasedyck [20], requires a process of recompression based on QR factorization to restore a minimal rank for the product matrix as well as the orthogonality of its components. This recompression is achieved by reforming the product $\dot{C}_u \times \dot{C}_v$ in terms of its SVD representation, i.e., its singular values and their corresponding right and left singular vectors. By factorizing $\dot{C}_u = Q_u \times R_u$, and $\dot{C}_v = Q_v \times R_v$, we can then represent $\dot{C} = Q_u \times R_u \times (Q_v \times R_v)^T = Q_u \times (R_u \times R_v^T) \times Q_v^T$, as the SVD of \dot{C} . Recompressing the result of the tiny product $R_u \times R_v^T$ using SVD or ACA, enables restoration of the rank of \dot{C} to a minimum value based on a predetermined fixed accuracy threshold or fixed rank truncation. This process of re-compression is described in steps 7–18 of Algorithm 1. The implementation of this variant leverages the randomized SVD on GPUs from [14], in the context of matrix compression for H^2 data format, to the TLR data format.

Batched Low-Rank GEMM. For batching the two GEMM-LLD and GEMM-LLL variants, the challenges are quite different. Batched GEMM-LLD is straightforward to implement on GPUs (and even on x86), due to existing fast batched GEMM implementations on small sizes [2, 27]. The task is far more complex for GEMM-LLL, since the recompression involves numerical kernels (GEQRF, ORGQR and GESVD), which are not as regular as standard GEMMs, e.g., in terms of memory accesses.

The support from vendor numerical libraries for batched versions of these routines is limited with poorly performing or simply inexistent implementations. We have further leveraged the batched GEQRF and ORGQR from [14] and integrated this into the batched GEMM-LLL. For the batched GESVD on the tiny $k \times k$ matrix, there are two options. The first is again based on the randomized SVD itself, while the second uses a novel ACA implementation on GPUs. Although ACA may require an expensive element sweeping procedure, this overhead is mitigated by the small matrix size. The resulting algorithm for batched low-rank is very similar to Algorithm 1, except that each call is now performed in a batched mode of execution.

TLR-GEMM (driver). Putting all previous standard and batched kernels together, we present TLR-GEMM on GPUs. We leverage the batched low-rank GEMM and operate on TLR matrices. This modular approach allows assessing the performance of each component, while enhancing software development productivity. The algorithm for TLR-GEMM driver consists of a single loop of nt successive outer-products, each corresponding to a batched GEMM-LLD or GEMM-LLL call, as depicted in Fig. 3. Compared to a GEMM operation on matrices with non-TLR data formats (involving recursion and tree traversals), TLR proves to be a simple yet effective approach, especially when considering hardware accelerators. For the algorithmic complexity of each variant, it is obvious that TLR-GEMM based on GEMM-LLL is more expensive than the one based on GEMM-LLD, because of the recompression stage.

6 Experimental Results

The benchmarking system is a two-socket 20-core Intel Broadwell running at 2.20 GHz with 512 GB of main memory, equipped with an NVIDIA GPU Volta V100 with 16 GB of main memory and PCIe 16x. We use a data-sparse matrix kernel (i.e., Hilbert) with singular values following an exponential decay. In fact, such decay in singular values is frequently observed in many matrix kernels in covariance-based scientific applications, such as climate/weather forecasting simulations [9]. All calculations are performed in double precision arithmetics. The reported performance numbers are compared to cuBLAS batched dense GEMM. Figure 4 illustrates the singular value distribution and the numerical accuracy assessment. The singular values of the Hilbert matrix kernel exponentially decay, as seen in Fig. 4(a). Approximately the first 30 are the most significant, while the remainder are close to machine precision and can be safely ignored. Figure 4(b) demonstrates the numerical robustness of the single GEMM-LLD and GEMM-LLL kernel variants using the same Hilbert matrix operator. GEMM-LLD approaches expected accuracy for rank smaller than GEMM-LLL, due to the rounding errors introduced by the additional floating-point operations from the recompression stage. Otherwise, both variants show correctness when truncating at ranks close to the accuracy threshold shown in Fig. 4(a).

Figure 5 highlights the speedups of batched GEMM-LLD and GEMM-LLL, against cuBLAS batch dense GEMM considering various ranks and a fixed batch size of

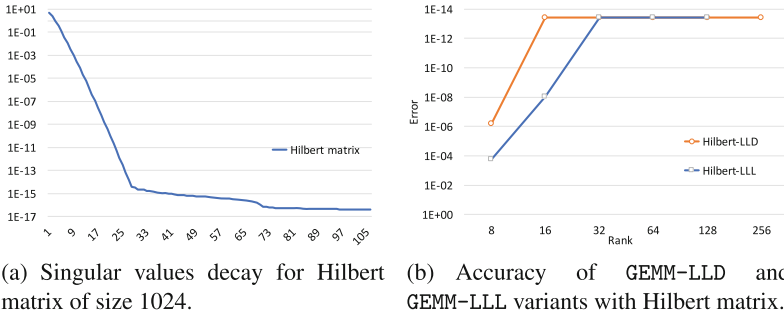
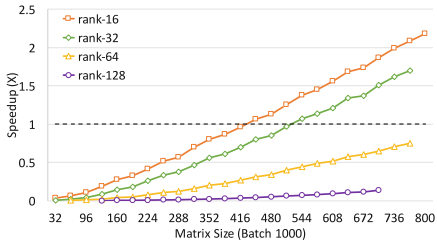
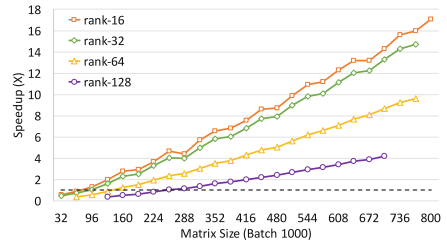
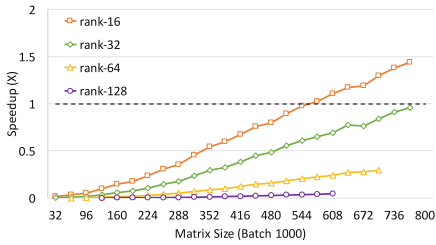
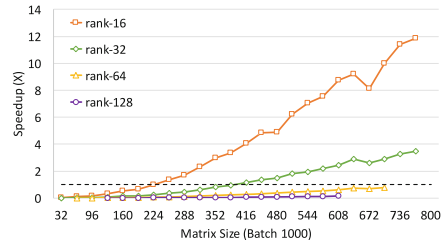
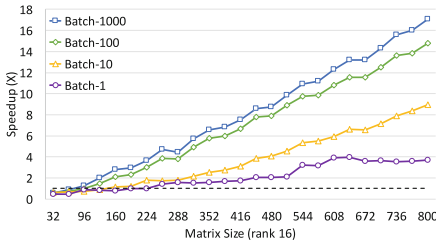
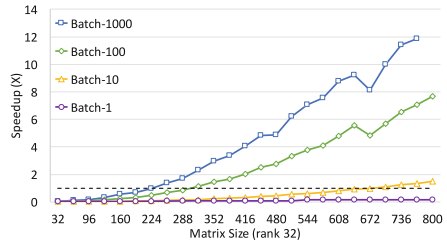


Fig. 4. Singular value distribution and accuracy assessment of the Hilbert matrix kernel.

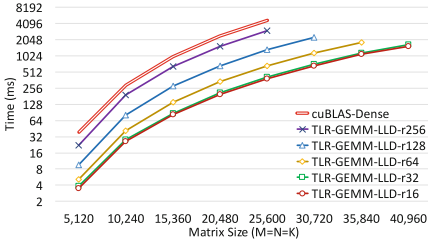
1000. Figure 5(a) and (b) illustrate the speedups of batched GEMM-LLD with and without compression overhead, respectively. Similarly, Fig. 5(c) and (d) illustrate the speedups of batched GEMM-LLL with and without compression overhead, respectively. Obviously, compression may turn out to be an expensive operation, which may slow down the the performance of batched GEMM-LLD (Fig. 5(a)) and batched GEMM-LLL (Fig. 5(c)); however, this overhead is usually occurring once, since the compressed form of the corresponding matrices may be used repeatedly (see TLR-GEMM in Fig. 5(b) and (d)). The speedups recorded for batched GEMM-LLD are higher than those for GEMM-LLL, when comparing to the cuBLAS batch dense GEMM, because of the recompression step. While speedups are obtained for all ranks for batched GEMM-LLD (Fig. 5(b)), batched GEMM-LLL (Fig. 5(d)) records speedups only for relatively small rank sizes. Although the Hilbert matrix kernel has an exponential singular value decay, we also assess performance for larger ranks. These extra flops, although unnecessary, allow stretching of the batched kernels and observing when the crossover point occurs. For instance, in Fig. 5(b), the batched GEMM-LLD with rank 128 runs out of memory, due to the dense storage of the matrix C , while still outperforming the cuBLAS batch dense GEMM. In Fig. 5(d), the batched GEMM-LLL with rank 128 runs out of memory, due to the temporary memory space required by the recompression stage, while not being able to outperform the cuBLAS batch dense GEMM.

Figure 6 shows the speedups for batched GEMM-LLD and GEMM-LLL, with varying batch count, against cuBLAS batch dense GEMM, when using the ranks at which numerical accuracy is reached from Fig. 4(b), i.e., 16 and 32, respectively. The performance speedup increase as the batch count rises reveals how the device becomes overwhelmed due to high occupancy.

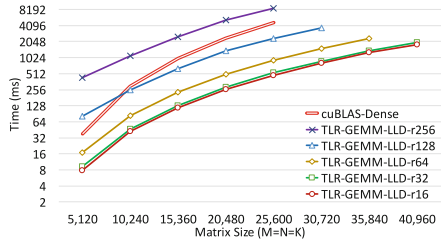
Figure 7 presents the elapsed time of TLR-GEMM based on batched GEMM-LLD, named TLR-GEMM-LLD (Fig. 7(a) and (b)), and based on batched GEMM-LLL, named TLR-GEMM-LLL (Fig. 7(c) and (d)), considering various ranks, against cuBLAS dense GEMM. TLR-GEMM-LLD (solid line plots) outperforms cuBLAS dense GEMM (double line plot) by more than an order of magnitude when A and B are already compressed, as shown in Fig. 7(a). When the matrices A and B are

(a) GEMM-LLD: updating dense C with compression overhead.(b) GEMM-LLD: updating dense C without compression overhead.(c) GEMM-LLL: updating low-rank C with compression overhead.(d) GEMM-LLL: updating low-rank C without compression overhead.**Fig. 5.** Speedups of batched GEMM-LLD and GEMM-LLL against batched dense cuBLAS GEMM, with batch size 1000.(a) GEMM-LLD: updating dense C .(b) GEMM-LLL: updating low-rank C .**Fig. 6.** Speedups of batched GEMM-LLD and GEMM-LLL against batched dense cuBLAS GEMM, with varying batch count, while fixing ranks to 16 and 32, respectively.

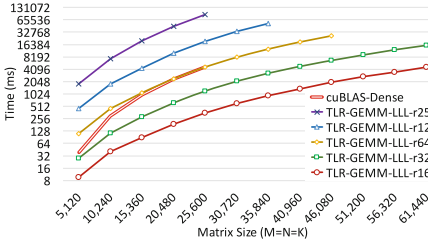
not compressed, the performance speedup slightly drops to eightfold, as seen in Fig. 7(b). Indeed, the expensive compression of matrices A and B is only performed once, followed by successive outer-products, in the form of batched GEMM-LLD calls. This allows to mitigate the compression overhead, discussed earlier in the section. TLR-GEMM-LLL (solid line plots) outperforms cuBLAS dense GEMM (double line plot) by more than an order of magnitude when A and B are already compressed, as shown in Fig. 7(c). When the matrices A and B are not compressed, the performance speedup remains almost the same, since



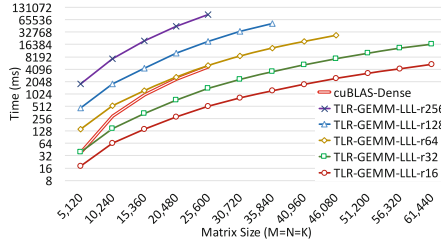
(a) TLR-GEMM-LLD with input matrices A and B already compressed.



(b) TLR-GEMM-LLD including compression of input matrices A and B .



(c) TLR-GEMM-LLL with matrices A, B and C already compressed.



(d) TLR-GEMM-LLL including compression of matrices A, B and C .

Fig. 7. Elapsed time of TLR-GEMM-LLD and TLR-GEMM-LLL with various ranks.

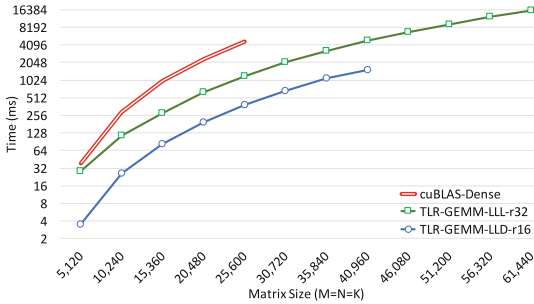


Fig. 8. Elapsed time of TLR-GEMM-LLD and TLR-GEMM-LLL with rank 16 and 32, respectively, with tile size 1024, compared to elapsed time of cuBLAS dense GEMM.

the (re)compression is the most time consuming part of the batched GEMM-LLL operations (Fig. 7(d)).

Figure 8 highlights the performance enhancements when using the Hilbert matrix kernel to perform TLR-GEMM with appropriate ranks for GEMM-LLD and GEMM-LLL, 16 and 32, respectively. Although the number of floating-point operations varies, the objective is to achieve the expected numerical accuracy. TLR-GEMM-LLD and TLR-GEMM-LLL kernels (solid line plots) score a speedup of more than an order of magnitude and fourfold, respectively, against cuBLAS dense GEMM (double line plot).

7 Conclusions and Future Work

This paper presents a novel batched tile low-rank (TLR) GEMM kernel on GPUs, which is a core operation of large-scale data sparse applications. Results demonstrate the numerical robustness and manyfold performance speedups against cuBLAS batched dense GEMM on the latest NVIDIA V100 GPU generation. This work represents a pathfinder toward enabling advanced hierarchical matrix computations on GPUs. Moreover, owing to its simplicity and modularity, the TLR data format may facilitate the port to multiple GPUs of batched low-rank matrix operations. Future work includes supporting non-uniform ranks for compression and operations to further reduce the memory footprint and flop count cost, in addition to supporting the other BLAS routines. We would like also to integrate the TLR compression and the TLR-GEMM operation in the Multi-Object Adaptive Optics application [30] in the context of computational astronomy and assess its real-time performance impact.

Data Availability Statement and Acknowledgments. The datasets and code generated during and/or analysed during the current study are available in the figshare repository: <https://doi.org/10.6084/m9.figshare.6387623> [16]. We would like to acknowledge Paris Observatory (LESIA, France) for giving us remote access to their Volta-based system, sponsored through a grant from project #671662 (Green Flash), funded by European Commission under program H2020-EU.1.2.2 coordinated in H2020-FETHPC-2014.

References

1. Matrix Algebra on GPU and Multicore Architectures. Innovative Computing Laboratory, University of Tennessee. <http://icl.cs.utk.edu/magma/>
2. The NVIDIA CUDA Basic Linear Algebra Subroutines (CUBLAS). <http://developer.nvidia.com/cublas>
3. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., et al.: TensorFlow: large-scale machine learning on heterogeneous distributed systems. arXiv preprint [arXiv:1603.04467](https://arxiv.org/abs/1603.04467) (2016)
4. Abdelfattah, A., et al.: High-performance tensor contractions for GPUs. *Procedia Comput. Sci.* **80**, 108–118 (2016). International Conference on Computational Science 2016, ICCS 2016, San Diego, California, USA, 6–8 June 2016
5. Abdelfattah, A., Haidar, A., Tomov, S., Dongarra, J.: Performance, design, and autotuning of batched GEMM for GPUs. In: Kunkel, J.M., Balaji, P., Dongarra, J. (eds.) *ISC High Performance 2016*. LNCS, vol. 9697, pp. 21–38. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41321-1_2
6. Abdelfattah, A., Ltaief, H., Keyes, D.E., Dongarra, J.J.: Performance optimization of sparse matrix-vector multiplication for multi-component PDE-based applications using GPUs. *Concurr. Comput.: Pract. Exp.* **28**(12), 3447–3465 (2016)
7. Agullo, E., et al.: Numerical linear algebra on emerging architectures: the PLASMA and MAGMA projects. *J. Phys: Conf. Ser.* **180**(1), 012037 (2009)
8. Akbudak, K., Ltaief, H., Mikhalev, A., Keyes, D.: Tile low rank cholesky factorization for climate/weather modeling applications on manycore architectures. In: Kunkel, J.M., Yokota, R., Balaji, P., Keyes, D. (eds.) *ISC 2017*. LNCS, vol. 10266, pp. 22–40. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-58667-0_2

9. Akbudak, K., Ltaief, H., Mikhalev, A., Charara, A., Keyes, D.: Exploiting data sparsity for large-scale matrix computations. In: Aldinucci, M., et al. (eds.) Euro-Par 2018. LNCS, vol. 11014, pp. xx–yy. Springer, Cham (2018)
10. Ambikasaran, S., Darve, E.: An $\mathcal{O}(N \log N)$ fast direct solver for partial hierarchically semiseparable matrices. *J. Sci. Comput.* **57**(3), 477–501 (2013)
11. Amestoy, P.R., Ashcraft, C., Boiteau, O., Buttari, A., L’Excellent, J.Y., Weisbecker, C.: Improving multifrontal methods by means of block low-rank representations. *SIAM J. Sci. Comput.* **37**(3), A1451–A1474 (2015). <https://doi.org/10.1137/120903476>
12. Aminfar, A., Darve, E.: A fast sparse solver for Finite-Element matrices. [arXiv:1403.5337](https://arxiv.org/abs/1403.5337) [cs.NA], pp. 1–25 (2014)
13. Börm, S.: Efficient numerical methods for non-local operators: \mathcal{H}^2 -Matrix compression, algorithms and analysis. EMS Tracts in Mathematics, vol. 14. European Mathematical Society (2010)
14. Boukaram, W.H., Turkiyyah, G., Ltaief, H., Keyes, D.E.: Batched QR and SVD algorithms on GPUs with applications in hierarchical matrix compression. *Parallel Comput.* **74**, 19–33 (2017)
15. Charara, A., Keyes, D., Ltaief, H.: Batched triangular dense linear algebra kernels for very small matrix sizes on GPUs. *ACM Trans. Math. Softw.* (2017, submitted). (under review, <http://hdl.handle.net/10754/622975>)
16. Charara, A., Keyes, D., Ltaief, H.: Software artifact for Euro-Par 2018: Tile Low-Rank GEMM Using Batched Operations on GPUs. *figshare. Code.* (2018). <https://doi.org/10.6084/m9.figshare.6387623>
17. Chávez, G., Turkiyyah, G., Zampini, S., Ltaief, H., Keyes, D.: Accelerated cyclic reduction: a distributed-memory fast solver for structured linear systems. *Parallel Comput.* **74**, 65–83 (2017)
18. Dongarra, J., Du Croz, J., Hammarling, S., Hanson, R.J.: An extended set of Fortran basic linear algebra subprograms. *ACM Trans. Math. Softw.* **14**, 1–17 (1988)
19. Dongarra, J., et al.: A proposed API for batched basic linear algebra subprograms. Mims preprint, University of Manchester (2016). <http://eprints.maths.manchester.ac.uk/id/eprint/2464>
20. Grasedyck, L., Hackbusch, W.: Construction and arithmetics of \mathcal{H} -matrices. *Computing* **70**(4), 295–334 (2003). <https://doi.org/10.1007/s00607-003-0019-1>
21. Hackbusch, W.: A sparse matrix arithmetic based on \mathcal{H} -matrices. part i: introduction to \mathcal{H} -matrices. *Computing* **62**(2), 89–108 (1999). <https://doi.org/10.1007/s006070050015>
22. Hackbusch, W.: Hierarchical Matrices: Algorithms and Analysis. Springer Series in Computational Mathematics, vol. 49. Springer, Heidelberg (2015). <https://doi.org/10.1007/978-3-662-47324-5>
23. Hackbusch, W., Börm, S.: Data-sparse approximation by adaptive \mathcal{H}^2 -matrices. *Computing* **69**(1), 1–35 (2002). <https://doi.org/10.1007/s00607-002-1450-4>
24. Hackbusch, W., Börm, S., Grasedyck, L.: HLib 1.4. Max-Planck-Institut, Leipzig (2012)
25. Hackbusch, W., Khoromskij, B., Sauter, S.: On \mathcal{H}^2 -matrices. In: Bungartz, H.J., Hoppe, R.H.W., Zenger, C. (eds.) Lectures on Applied Mathematics, pp. 9–29. Springer, Heidelberg (2000). https://doi.org/10.1007/978-3-642-59709-1_2
26. Halko, N., Martinsson, P.G., Tropp, J.A.: Finding structure with randomness: probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Rev.* **53**(2), 217–288 (2011). <https://doi.org/10.1137/090771806>

27. Heinecke, A., Henry, G., Hutchinson, M., Pabst, H.: LIBXSMM: accelerating small matrix multiplications by runtime code generation. In: 0001, J.W., Pancake, C.M. (eds.) Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, 13–18 November 2016, p. 84. ACM (2016)
28. Kim, K., et al.: Designing vector-friendly compact BLAS and LAPACK kernels. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2017, pp. 55:1–55:12. ACM, New York (2017). <https://doi.org/10.1145/3126908.3126941>
29. Kriemann, R.: LU factorization on many-core systems. *Comput. Vis. Sci.* **16**(3), 105–117 (2013). <https://doi.org/10.1007/s00791-014-0226-7>
30. Ltaief, H., et al.: Real-time massively distributed multi-object adaptive optics simulations for the european extremely large telescope. In: 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), accepted, May 2018
31. North, G.R., Wang, J., Genton, M.G.: Correlation models for temperature fields. *J. Clim.* **24**, 5850–5862 (2011)
32. Rouet, F.H., Li, X.S., Ghysels, P., Napov, A.: A distributed-memory package for dense hierarchically semi-separable matrix computations using randomization. *ACM Trans. Math. Softw.* **42**(4), 27:1–27:35 (2016)
33. Shi, Y., Niranjan, U.N., Anandkumar, A., Cecka, C.: Tensor contractions with extended BLAS kernels on CPU and GPU. In: HiPC, pp. 193–202. IEEE Computer Society (2016)
34. Tyrtyshnikov, E.: Mosaic-skeleton approximations. *Calcolo* **33**(1), 47–57 (1996)