Batched Tile Low-Rank GEMM 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, KSA
[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 most of the 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 the 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 retains 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 it can exploit the data sparsity structure of the matrix while leveraging the underlying TLR compression format. The main idea consists in aggregating all operations onto 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 advance 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 paradigm (i.e., simulation and big data), large-scale scientific applications, such as climate/weather forecasting [29], require a profound and mandatory 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 cut down the constraints on the memory footprint and the algorithmic complexity. For instance, hierarchical matrices or $\mathscr{H}$-matrices have been recently resurrected for high-performance computing [28, 30] 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 HiCMAlibrary an approximation method that exploits the natural data sparsity of the off-diagonal tiles while exposing parallelism to the fore [7]. Based on the tile low-rank (TLR) data format, the off-diagonal tiles of the dense covariance matrix are compressed up to a specific accuracy threshold, without unilaterally compromising the model fidelity. The resulting data structure, much smaller than the original dense tiles, stands as 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 like NVIDIA GPUs maintain high occupancy and would require developers to drift 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* [17], executes many smaller operations in parallel to make efficient use of the hardware and its instruction-level parallelism. To our knowledge, this paper 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: $\mathscr{H}$-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, perhaps one of the most important operations for high-performance numerical libraries, on TLR data format.

The remainder of the paper is organized as follows. Section 2 presents related work and details our research contributions. Section 3 recalls the batched linear algebra community effort and gives a general overview of the hierarchical low-rank matrix approximation. Section 4 introduces the new TLR GEMM and its variants. The implementation details of the various TLR GEMM kernels are given in Section 5. Section 6 assesses the accuracy, memory footprint and performance of TLR GEMM on the latest NVIDIA GPU hardware generation and compares it against the state-of-the-art high-performance batched dense GEMM, as implemented in [2]. We conclude in Section 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 [16] operations and their usage in dense linear algebra algorithms. With the need to solve large-scale 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. 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 [5], while batched dense LAPACK factorizations [14] are needed in sparse direct solvers for the Poisson equation using cyclic reduction [15]. Moreover, with the emergence of artificial intelligence, batched dense GEMM of even smaller sizes are needed in tensor contractions [3, 4, 31] and in deep learning frameworks [26, 27, 18]. To facilitate the dissemination of all these efforts, a new standard has been proposed to homogenize the various batched API [17]. While the literature is quite rich in leveraging batched executions for dense and sparse linear algebra operations on x86 and hardware accelerators, this trend has faced challenges and has not penetrated data-sparse applications yet, involving large hierarchically low-rank matrices (i.e., $\mathscr{H}$-matrices). In fact, there are mainly three points to consider, when designing batched operations for $\mathscr{H}$-matrices. First, there should be an efficient batched compression operations for $\mathscr{H}$-matrices on x86 and on hardware accelerators. Second, the inherent recursive formulation of $\mathscr{H}$-matrix resulting from nested dissections should be replaced, since it is not compliant with batched operations. And third, a strong support is eventually required to handle batched matrix operations on the data-sparse compressed format, such as $\mathscr{H}^2$, Hierarchically Semi-Separable representation (HSS) and Hierarchical Off-Diagonal Low-Rank (HODLR) matrix. More recently, a block/tile low-rank (TLR) compressed format has been recently introduced on x86 [9, 7], 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 [25] for $\mathscr{H}^2$ matrix compression has been ported to hardware accelerators [13]. The aforementioned three bottlenecks may now be unblocked to deploy TLR matrix computations on GPUs. Our contribution lies at the major confluence of tile low-rank (TLR) matrix and batched operations research trends. Not only does it consolidate the independent developments of two distinct communities, but it also permits to deploy the first TLR general matrix-matrix multiplication (TLR GEMM) operating on data sparse matrix structures using GPU hardware accelerators. Since TLR GEMM is a building block for data sparse linear algebra, it is essential for future deployments of more advanced numerical algorithms, i.e., matrix factorizations and solvers 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, as long as 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 CPU to GPU memory through the thin PCIe pipe may not be a worthwhile effort and 2) the overhead of the kernel launches, which does not get compensated by the low computation complexity. High-performance frameworks for batched operations [1, 2, 14] try to overcome both challenges by stitching together multiple operations occurring on independent data structures. This batched mode of execution increases hardware occupancy to hit higher sustained peak bandwidth while launching a single kernel to remove the kernel launch overheads all together. Fig. 1(a) sketches batched operations of small

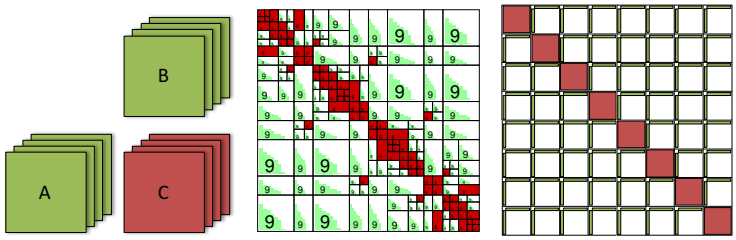
dense GEMM operations $C += A \times B$. Following the same community effort for the legacy BLAS, a community call for standardizing the batched API [17] has been initiated, gathering hardware vendors and researchers.This standardization effort enhances software development productivity, while the batched API gets maturity in the scientific community.

Hierarchical Low-Rank Matrix Computations Hierarchically low-rank matrix or $\mathscr{H}$-matrix [32, 20, 23, 24, 22] is a low-rank block approximation of a dense (sub)matrix, whose off-diagonal blocks can be each represented by an outer product of rectangular bases obtained from compression, e.g., via orthogonal transformations with singular value decomposition. $\mathscr{H}$-matrix captures on each block the most significant singular values and their corresponding singular vectors up to an application-dependent accuracy threshold. Fig. 1(b) highlights the structure of an $\mathscr{H}$-matrix resulting from a boundary element method. Each off-diagonal green block has been approximated up to a similar rank equal 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 to traverse low-rank off-diagonal blocks and to compress them using an adequate data storage format such as $\mathscr{H}$ [28], Hierarchical Off-Diagonal Low-Rank (HODLR) [10], $\mathscr{H}$ [21], Hierarchically Semi-Separable representation (HSS) [8, 30] and $\mathscr{H}^2$ [12]. All these data compression formats belong to the family of $\mathscr{H}$-matrices and can be differentiated by the type of their bases i.e., nested (HSS and $\mathscr{H}^2$) or non-nested ($\mathscr{H}$ and HODLR), as well as the type of admissibility conditions, i.e., strong ($\mathscr{H}$ and $\mathscr{H}^2$) 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) [9, 7] data format is yet another case of $\mathscr{H}$ data format with non-nested bases and strong admissibility condition. The matrix is logically split into tiles, similar to the tile algorithms from the PLASMA library [6]. The off-diagonal tiles may then be compressed using the rank-revealing QR once the whole dense matrix has been generated [9] or *on-the-fly* using the randomized singular value decomposition [25] while performing the matrix computations [7]. Fig. 1(c) shows an illustration of such a TLR matrix composed of an 8x8 logical tile. Thanks to its simplicity, TLR permits to flatten the inherent recursive formulation. Although TLR may not provide as 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), $\mathscr{H}$-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

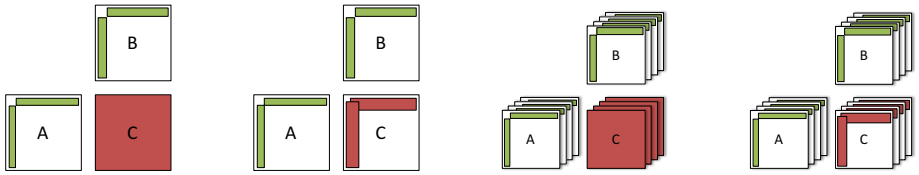
This section describes the design of the TLR GEMM kernel and identifies its variants using a bottom-up approach: from the single GEMM kernel operating on tile low-rank (TLR) data format to the corresponding batched operations, and then all the way up to the actual TLR GEMM driver.



(a) Batched dense GEMM execution. (b) $\mathscr{H}$-matrix from BEM. (c) TLR data format.

Fig. 1: Consolidating batched operations and $\mathscr{H}$-matrix through TLR data format.

Low-rank data format. Low-rank approximation consists of compressing a dense (sub)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 choices for compression algorithms are typically Rank-Revealing QR (RRQR), adaptive cross-approximation (ACA), (randomized) singular value decomposition (SVD), etc. The randomized Jacobi-based SVD [25] maps well on SIMD architectures because it does not involve pivoting nor element sweeping, as in RRQR and ACA methods, respectively. It is perhaps the most optimized compression algorithms on GPUs, as implemented in [13].



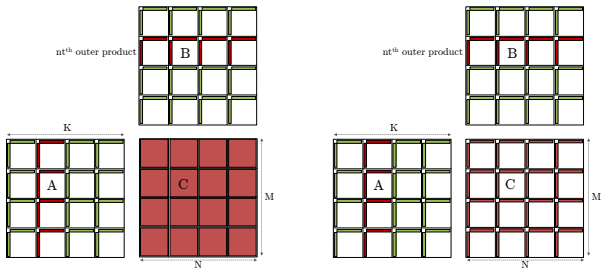
(a) *GEMM − LLD*. (b) *GEMM − LLL*. (c) Batched *GEMM − LLD*. (d) Batched *GEMM − LLL*.

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

Low-rank GEMM variants. We can identify four possible variants, based on the input data format of the involved matrices A, B, and C. We use the following notation: $GEMM - T_A T_B T_C$ is the GEMM kernel for a given input type (Dense or Low-rank) for the matrix 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 solely focus on the last two variants in this paper, i.e., $GEMM - LLD$ and $GEMM - LLL$, since they are the basis for supporting the Schur complement calculation and for matrix factorization, in the context of sparse direct solvers [9] and data-sparse matrix solvers [7], 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 kernel is now defined as a single kernel, i.e., $Batched - GEMM - LLD$ and $Batched - GEMM - LLL$, which simultaneously executes independent $GEMM - LLD$ and $GEMM - LLL$ operations, as demonstrated in Figures 2(c) and 2(d), respectively. This batched kernel is used as a building block for the main driver performing the batched TLR GEMM on large TLR matrices, as described in the next section.

Batched TLR GEMM (driver). In the driver of the batched TLR GEMM operation, the data-sparse matrices A, B, and C 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 3(b) represent the batched 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 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 enough work, concurrent processing of these independent low-rank tiles is necessary to increase the GPU occupancy. To overcome this challenge, we process these outer product GEMM calls in a batched mode, successively, and update 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 batched low-rank GEMM.



(a) Updating dense C with a call to batched $GEMM - LLD$ routine.

(b) Updating TLR C with a call to batched $GEMM - LLL$ routine.

Fig. 3: Processing the batched TLR GEMM as a series of nt outer product 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 of m-rows, C and B of n-columns, A

and B of k-columns and k-rows respectively, 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 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 [25]. We assume 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 in 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 $\dot{C} = \dot{C_u} \times \dot{C_v}$ with bloated rank $\dot{r_c}$. As such, low-rank matrix addition, as described by Grasedyck [19], 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 to restore 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 [13], in the context of matrix compression for H^2 data format, to the TLR data format.

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

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

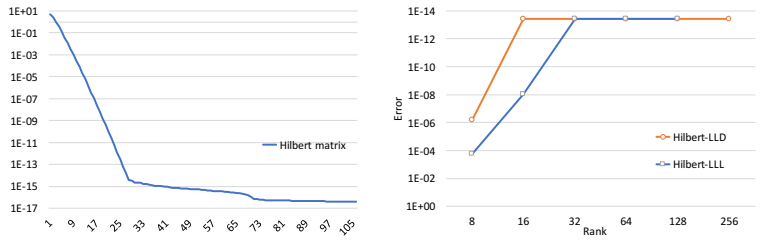
Batched Low-rank GEMM. When it comes to batching the two $GEMM - LLD$ and $GEMM - LLL$ variants, the challenges are quite different. Batched $GEMM - LLD$ is straightforward to implement on GPUs (and even on x86), thanks to existing fast batched GEMM implementations on small sizes [2, 26]. The task is 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 implementations or simply inexistent. We have further leveraged the batched GEQRF and ORGQR from [13] and integrated into the batched $GEMM-LLL$. For the batched GESVD on the tiny k-by-k matrix, we have two options. The first one is again based on the randomized SVD itself, while the second one 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 batched mode of execution.

Batched TLR GEMM (driver). Putting all previous standard and batched kernels together, we present the batched TLR GEMM on GPUs. We leverage the batched low-rank GEMM and operate on TLR matrices. This modular approach allows to assess the performance of each component, while enhancing software development productivity. Indeed, the algorithm for batched TLR GEMM driver consists of a single loop of nt successive batched 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 turns out to be a simple yet effective approach, especially when considering hardware accelerators. For the algorithmic complexity of each variant, it is obvious that the batched 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 2.20GHz with 512GB of main memory, equipped with an NVIDIA GPU Volta V100 with 16GB 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, this kernel is representative of many matrix kernels in covariance-based scientific applications, such as climate/weather forecasting simulations. All calculations are performed in double precision arithmetics. The reported performance numbers are against cuBLAS batched dense GEMM. Fig. 4 draws 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). Around the first 30 are the most significant, while the remaining ones are close to machine precisions and can be safely ignored. Fig. 4(b) demonstrates the numerical robustness of the single $GEMM-LLD$ and $GEMM-LLL$ kernel variants using the same Hilbert matrix operator. $GEMM-LLD$ seems to reach expected accuracy for rank smaller than $GEMM-LLL$, due the rounding errors introduced by the additional floating-point operations from the recompression stage. Otherwise, both variants show correctness when truncating at ranks around the accuracy threshold shown in Fig. 4(a). Fig. 5 highlights the Speedups of batched $GEMM-LLD$ and $GEMM-LLL$, against cuBLAS batch dense GEMM with varying ranks and a fixed batch size of 1000. The speedups recorded for batched $GEMM-LLD$ are higher than $GEMM-LLL$, when comparing against cuBLAS batch dense GEMM, because of the recompression step. While speedups are obtained for all ranks for batched $GEMM-LLD$, batched

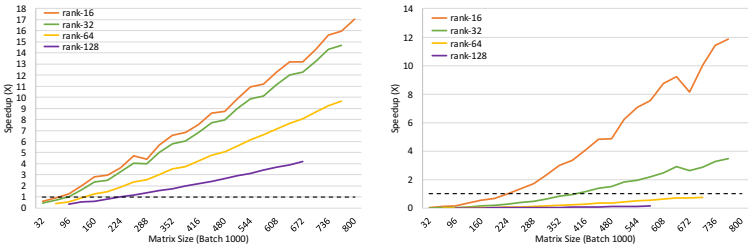


(a) Singular values decay for Hilbert Matrix of size 1024.

(b) Accuracy of the $GEMM - LLD$ and $GEMM - LLL$ variants with Hilbert.

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

$GEMM - LLL$ records speedups 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 to stretch the batched kernels and see when the crossover point occurs. For instance, in Fig. 5(a), 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(b), 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. Fig. 6 shows the speedups for batched $GEMM - LLD$

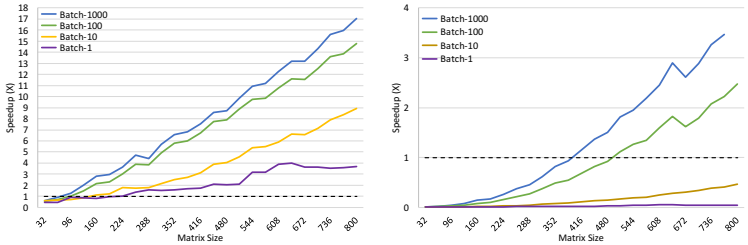


(a) GEMM-LLD: updating dense C.

(b) GEMM-LLL: updating low-rank C.

Fig. 5: Speedups of batched $GEMM - LLD$ and $GEMM - LLL$, with batch size 1000.

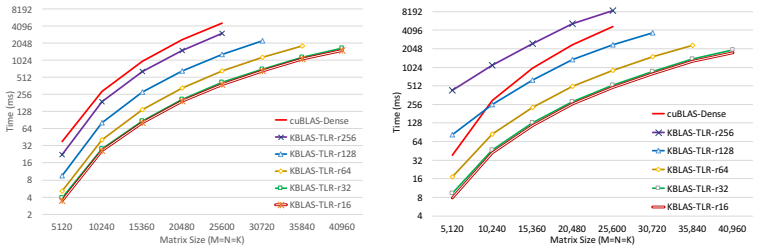
and $GEMM - LLL$, against cuBLAS batch dense GEMM, when using the ranks at which numerical accuracy is reached from Fig. 4(b), i.e., 16 and 32, respectively. The performance speedup increase as the batch count rises, which reveals how the device gets overwhelmed thanks to a high occupancy. Fig. 7 presents the elapsed time of batched TLR GEMM based on $GEMM - LLD$ against cuBLAS batch dense GEMM with various ranks. It outperforms cuBLAS batch dense GEMM by more than an order of magnitude when A and B are already compressed, as shown in Fig. 7(a). In case the matrices A and B are not compressed, the performance speedup slightly drops to eightfold, thanks to an efficient GPU compression implementation based on randomized SVD [13], as seen in Fig. 7(b). Fig. 8 highlights the elapsed time of batched TLR GEMM based on $GEMM - LLL$ against cuBLAS batch dense GEMM with various



(a) GEMM-LLD: updating dense C. (b) GEMM-LLL: updating low-rank C.

Fig. 6: Speedups of batched $GEMM - LLD$ and $GEMM - LLL$, with varying batch count, while fixing ranks to 16 and 32, respectively.

ranks. It outperforms cuBLAS batch dense GEMM by more than an order of magnitude when A and B are already compressed, as shown in Fig. 8(a). In case the matrices A and B are not compressed, the performance speedup stays almost the same, since the (re)compression is the most time consuming part of the batch kernels (Fig. 8(b)).



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

Fig. 7: Elapsed time of batched TLR GEMM-LLD with various ranks.

Fig. 9 highlights the performance enhancements when using the Hilbert matrix kernel to perform the batched TLR GEMM with the appropriate ranks for $GEMM - LLD$ and $GEMM - LLL$, 16 and 32, respectively. Although the number of floating-point operations are different, the objective is to achieve the expected numerical accuracy. Batched TLR GEMM-LLD and TLR GEMM-LLL kernels score a speedup of more than an order of magnitude and fourfold, respectively.

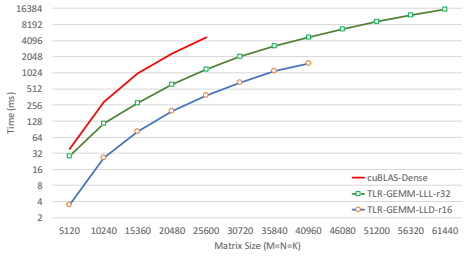
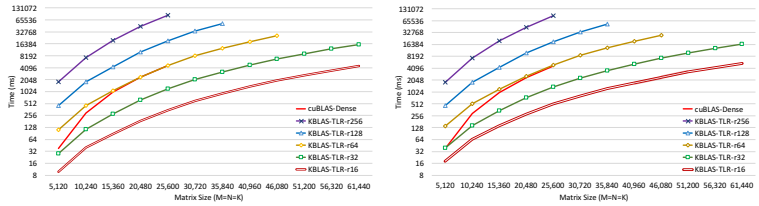


Fig. 9: Elapsed time of TLR $GEMM - LLD$ and TLR $GEMM - LLL$ with rank 16 and 32, respectively, with tile size 1024.



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

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

Fig. 8: Elapsed time of batched TLR GEMM-LLL with various ranks.

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, thanks to its simplicity and modularity, TLR data format may facilitate the port to multiple GPUs of batched low-rank matrix operations.

References

1. Matrix Algebra on GPU and Multicore Architectures. Innovative Computing Laboratory, University of Tennessee. Available at http://icl.cs.utk.edu/magma/.
2. The NVIDIA CUDA Basic Linear Algebra Subroutines (CUBLAS). Available at http://developer.nvidia.com/cublas.
3. A. Abdelfattah, M. Baboulin, V. Dobrev, J. Dongarra, C. Earl, J. Falcou, A. Haidar, I. Karlin, Tz. Kolev, I. Masliah, and S. Tomov. High-performance tensor contractions for gpus. *Procedia Computer Science*, 80:108 – 118, 2016. International Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA.
4. A. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra. Performance, design, and autotuning of batched gemm for gpus. In Julian M. Kunkel, Pavan Balaji, and Jack Dongarra, editors, *High Performance Computing*, pages 21–38, Cham, 2016. Springer International Publishing.
5. A. Abdelfattah, H. Ltaief, D. E. Keyes, and J. J. Dongarra. Performance optimization of sparse matrix-vector multiplication for multi-component pde-based applications using gpus. *Concurrency and Computation: Practice and Experience*, 28(12):3447–3465, 2016.
6. E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical Linear Algebra on Emerging Architectures: The PLASMA and MAGMA Projects. *J. Phys.: Conf. Ser.*, 180(1), 2009.
7. K. Akbudak, H. Ltaief, A. Mikhalev, and D. Keyes. Tile Low Rank Cholesky Factorization for Climate/Weather Modeling Applications on Manycore Architectures. In *Proceedings of ISC'17 International Conference on Supercomputing*, Francfort, Germany, June 2017.
8. S. Ambikasaran and E. Darve. An $\mathscr{O}(N \log N)$ fast direct solver for partial Hierarchically Semiseparable matrices. *Journal of Scientific Computing*, 57(3):477–501, Dec 2013.
9. P. R. Amestoy, C. Ashcraft, O. Boiteau, A. Buttari, J.-Y. L'Excellent, and C. Weisbecker. Improving Multifrontal Methods by Means of Block Low-Rank Representations. *SIAM Journal on Scientific Computing*, 37(3):A1451–A1474, 2015.

10. A. Aminfar and E. Darve. A fast sparse solver for Finite-Element matrices. *arXiv:1403.5337 [cs.NA]*, pages 1–25, 2014.
11. S. Börm. *Efficient numerical methods for non-local operators: $\mathscr{H}^2$-Matrix compression, algorithms and analysis*, volume 14 of *EMS Tracts in Mathematics*. European Mathematical Society, 2010.
12. W. H. Boukaram, G. Turkiyyah, H. Ltaief, and D. E. Keyes. Batched qr and svd algorithms on gpus with applications in hierarchical matrix compression. *Parallel Computing*, 2017.
13. A. Charara, D. Keyes, and H. Ltaief. Batched Triangular Dense Linear Algebra Kernels for Very Small Matrix Sizes on GPUs. *Submitted to ACM Trans. Math. Softw. (under review, http://hdl.handle.net/10754/622975)*, 2017.
14. G. Chávez, G. Turkiyyah, S. Zampini, H. Ltaief, and D. Keyes. Accelerated cyclic reduction: A distributed-memory fast solver for structured linear systems. *Parallel Computing*, 2017.
15. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of Fortran basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14:1–17, 1988.
16. J. Dongarra, I. Duff, M. Gates, A. Haidar, S. Hammarling, N. J. Higham, J. Hogg, P. Valero-Lara, S. D. Relton, S. Tomov, and M. Zounon. A Proposed API for Batched Basic Linear Algebra Subprograms. Mims preprint, University of Manchester, 2016.
17. M. Abadi et. al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.
18. L. Grasedyck and W. Hackbusch. Construction and arithmetics of $\mathscr{H}$-matrices. *Computing*, 70(4):295–334, Aug 2003.
19. W. Hackbusch. A sparse matrix arithmetic based on $\mathscr{H}$-matrices. part i: Introduction to $\mathscr{H}$-matrices. *Computing*, 62(2):89–108, 1999.
20. W. Hackbusch. *Hierarchical matrices: Algorithms and analysis*, volume 49. Springer, 2015.
21. W. Hackbusch and S. Börm. Data-sparse Approximation by Adaptive $\mathscr{H}^2$-Matrices. *Computing*, 69(1):1–35, March 2002.
22. W. Hackbusch, S. Börm, and L. Grasedyck. HLib 1.4. 2012. Max-Planck-Institut, Leipzig.
23. W. Hackbusch, B. Khoromskij, and S.A. Sauter. On H2-Hatrices. pages 9–29, 2000.
24. N. Halko, P. G. Martinsson, and J. A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Review*, 53(2):217–288, 2011.
25. A. Heinecke, G. Henry, M. Hutchinson, and H. Pabst. LIBXSMM: accelerating small matrix multiplications by runtime code generation. In John West 0001 and Cherri M. Pancake, editors, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016*, page 84. ACM, 2016.
26. K. Kim, T. B. Costa, M. Deveci, A. M. Bradley, S. D. Hammond, M. E. Guney, S. Knepper, S. Story, and S. Rajamanickam. Designing vector-friendly compact blas and lapack kernels. In Bernd Mohr and Padma Raghavan, editors, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2017, Denver, CO, USA, November 12 - 17, 2017*, pages 55:1–55:12. ACM, 2017.
27. R. Kriemann. LU factorization on many-core systems. *Computat. and Visualiz. in Science*, 16(3):105–117, 2013.
28. G. R. North, J. Wang, and M. G. Genton. Correlation models for temperature fields. *Journal of Climate*, 24:5850–5862, 2011.
29. F.-H. Rouet, X. S. Li, P. Ghysels, and A. Napov. A distributed-memory package for dense hierarchically semi-separable matrix computations using randomization. *ACM Trans. Math. Softw.*, 42(4):27:1–27:35, June 2016.
30. Y. Shi, U. N. Niranjan, A. Anandkumar, and C. Cecka. Tensor contractions with extended blas kernels on cpu and gpu. In *HiPC*, pages 193–202. IEEE Computer Society, 2016.
31. E. Tyrtyshnikov. Mosaic-skeleton approximations. *Calcolo*, 33(1):47–57, 1996.