GPU-based LU Factorization and Solve on Batches of Matrices with Band Structure

Ahmad Abdelfattah
University of Tennessee
Knoxville, USA
ahmad@icl.utk.edu

Stan Tomov
University of Tennessee
Knoxville, USA
tomov@icl.utk.edu

Piotr Luszczek
University of Tennessee
Knoxville, USA
luszczek@icl.utk.edu

Hartwig Anzt
University of Tennessee
Knoxville, USA
hanzt@icl.utk.edu

Jack Dongarra
University of Tennessee
Knoxville, USA
dongarra@icl.utk.edu

ABSTRACT

This paper presents a portable and performance-efficient approach to solve a batch of linear systems of equations using Graphics Processing Units (GPUs). Each system is represented using a special type of matrices with a band structure above and/or below the diagonal. Each matrix is factorized using an LU factorization with partial pivoting for numerical stability. Subsequently, the factors are used to find the solution for as many right hand sides as needed. The width of the band is often small enough that performing a fully dense LU factorization results in poor performance. We follow the standard LAPACK specifications for addressing this type of problems and develop a dedicated solver that runs efficiently on GPUs. No similar solver is currently available in the vendor's software stack, so performance results are shown on both NVIDIA and AMD GPUs relative to a parallel CPU solution utilizing OpenMP for thread-level parallelization.

KEYWORDS

Band matrix, LU factorization, batch solvers, GPU computing, performance portability

ACM Reference Format:
Ahmad Abdelfattah, Stan Tomov, Piotr Luszczek, Hartwig Anzt, and Jack Dongarra. 2023. GPU-based LU Factorization and Solve on Batches of Matrices with Band Structure. In *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3624062.3624247

1 INTRODUCTION AND RELATED WORK

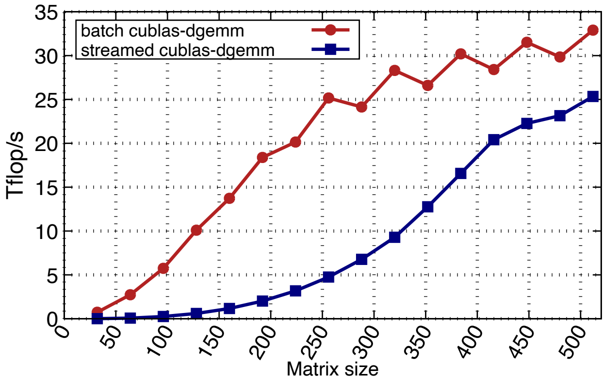
For decades, high performance implementations of dense linear algebra (DLA) algorithms have been defined using the standard specifications of BLAS [1] (Basic Linear Algebra Subprograms) and LAPACK [2] (Linear Algebra Package). The widespread adoption of these specifications, especially the user-level API, has led to performance portability for applications utilizing DLA functionality on almost every hardware platform. Most of the vendors providing high performance computing (HPC) processors ensure that their BLAS and LAPACK libraries are up-to-date, in terms of performance.

The past decade witnessed an increased interest in providing high performance BLAS and LAPACK functionality for batches of relatively small and independent matrices [13]. While the then-existing solutions could provide an adequate functionality (e.g. using concurrent executions of many BLAS/LAPACK instances), it has been proven in many past contributions that dedicated implementations achieve significantly higher performance [6]. For example, Figure 1 shows the performance gap between dedicated batch linear algebra kernels and a solution launching kernels concurrently on multiple streams. The figure shows that both compute-bound kernels (like matrix-multiply at the top) and memory-bound kernels (like matrix-vector multiply at the bottom) benefit from dedicated designs for batch workloads. As a result, there have been numerous contributions to provide batch BLAS/LAPACK functionality on every hardware platform, such as batch matrix multiply [6, 16], one-sided matrix factorizations [5, 8], and singular value decomposition [10]. In most cases, it is assumed that the batch is uniform, meaning that all matrices have the same dimensions. However, some recent contributions have also addressed non-uniform batches [4].

In this paper, we focus on a new functionality that performs the LU factorization and solves on batches of matrices with band structure. No equivalent implementation currently exists in the software offering of the two major GPU vendors (AMD and NVIDIA). Even a solution based on concurrent streams is not possible since the band matrix processing is absent from the single matrix API. While some previous contributions address the batch LU factorization of dense matrices [5, 19], we are unaware of previous efforts that address band matrices on modern GPUs, especially with arbitrary number of sub- and super-diagonals.

2 USE CASES ACROSS APPLICATIONS AND SOFTWARE LIBRARIES

Since its establishment as the *de facto* standard, Batched BLAS and LAPACK [3] continue enjoying deployment in scientific applications [15]. Here however, we focus on the high performance



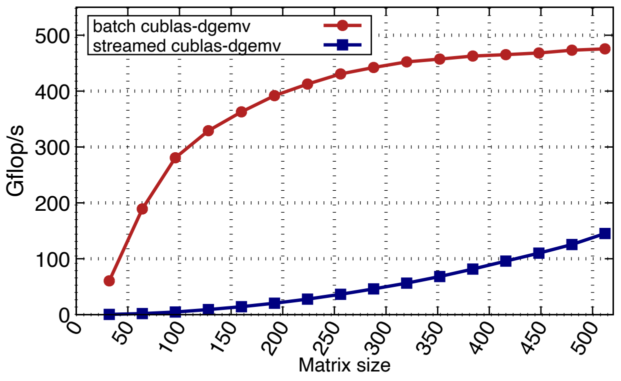


Figure 1: Batch matrix multiply (top) and matrix-vector multiply (bottom) versus concurrent stream execution using 16 streams. Results are shown in double precision for batches of size 500 with uniform dimensions. The GPU is an NVIDIA Tesla H100-PCIe GPU, tested using CUDA-12.1.

computing scenarios that require further extension of what may be regarded broadly as a *batched processing* paradigm.

2.1 PELE Suite: Simulation of Chemical Kinetics

PELE is a scientific application representative of combustion simulation codes. Its major computational workload is concentrated in solving many occurrences of small linear system, that exhibit varying patterns of structural sparsity and may differ in numerical properties with a large range of condition numbers. Even one of these characteristics would make such matrix batches problematic for most sparse direct or iterative solvers. To be more precise, the structural sparsity affects the fill-in and the main memory data traffic, while the numerical conditioning affects the behavior of numerical stability measures such as threshold pivoting or the iteration count required for convergence. Using a band dense solver resolves both of these problems within the same computational framework with known numerical estimates and bounds.

The typical matrix sizes in batches do not exceed 150 but many are sized 50 or less thus stressing the design and focusing on taking advantage of massive parallelism of the compute units at very fine grain. The motivation behind using a dense solver stems from the structural sparsity of the problem, where approximaely 90% of entries are non-zero, with only a few entries dipping down to arround 30%. Moreover, after accounting for fill-in in direct solvers, most entries become non-zeros and no matrices' L and U factors have less than 90% non-zero entries.

2.2 Plasma Containment in WDMApp and XGC

The XGC (Exascale Gyrokinects) framework [21] aims to solve particle-in-cell (PIC) formulation for gyrokinetic simulations occurring in physics applications designed specifically for fusion reactions and plasma containment as represented by the Whole Device Model Application (WDMApp) project. More specifically, the code is an implementation of a fast and fully conservative Landau collision operator in structure-preserving methods for fusion plasma simulations. Recently, it was used in the recent 10 species models optimized for the WDMApp's milestone of ITER project simulations with tungsten impurities.

The matrices used for simulating the phenomena of interest to the domain scientists are notably larger in size. For example, a single species solver resolves 2D domain with Q3 finite elements and AMR (adaptive mesh refinement). This results in 512 sparse linear systems in a single batch, each having M=N=193 equations. In real world scenarios, multi-species setups are used, representing even greater importance of batched solvers that could increase their efficiency due to higher arithmetic intensity for larger problem sizes.

2.3 SUNDIALS Software Library

The SUNDIALS library focuses on solving ODEs (ordinary differential equations) and the connection to batched solvers occurs in its ReactEval test program, a part of the Pele physics from the AMREx application (Exascale AMR)[23]. The test advances only the reaction equations from a given initial state unlike the full Pele application problems featuring complete chemical kinetics (reactions), that are coupled with the compressible (PeleC)[14, 22] or incompressible (PeleLM[9, 12, 17, 18, 20] and PeleLMeX) Navier-Stokes equations. In this limited test case, the initial state comes from a sinusoidal temperature profile. However, ReactEval can also be initialized with an input file with states produced by PeleLM(eX). Such a setup is representative of the reaction evolution that typifies a full run of a Pele simulation with an external source term. Thus, ReactEval can be considered an excellent benchmark for both time-integration as well as linear solvers for combustion codes. This capability to mirror the full PeleLM(eX) setup is its key feature, enabling the variation of input sets. This results in a range of the simulated system's key metrics such as numerical stiffness and conditioning. Furthermore, it makes possible the use of variety of chemical mechanisms, which changes both the size of the ODE and the size of batch. Controlling the total number of linear systems and the number of batches occurs by changing the AMR parameters. Only at the moment the batches are formed, the control is passed to an efficient band batched solver that is brought to fully exploit the hardware accelerator's compute capacity.

The following sections provide the design details of the batch linear solver on band matrices. Performance results are shown in double precision for incremental design improvements. The performance metric is time-to-solution. It is not trivial to estimate the rate

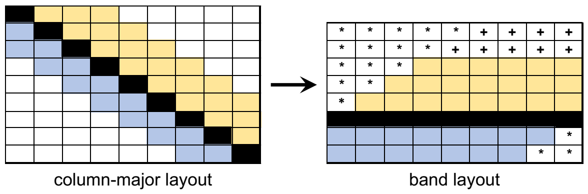


Figure 2: An example of standard layout for band matrices in LAPACK: with kl= 2 and ku= 3, the lower and upper band-width, respectively.

of execution (e.g., Gflop/s), since the operation count per matrix depends on the pivoting pattern, as described later in this section.

3 BAND MATRIX LAYOUT

We assume here that all matrices have the same dimensions and the same band structure. Also, the data layout is identical to the standard LAPACK representation of band matrices. Figure 2 shows a 9×9 band matrix in the column-major layout and its representation using the LAPACK's band storage. A band matrix is characterized by the number of super-diagonals (upper bandwidth ku) and the number of sub-diagonals (lower bandwidth kl). A band storage simply treats every diagonal as a row in the band data layout.

LAPACK defines a special routine (GBSV) for solving Ax = B for a band matrix A. The routine internally factorizes the matrix using an LU factorization (GBTRF) followed by forward/backward triangular solves (GBTRS). This paper adopts the same convention by designing these routines for a batch execution on the GPU. Note that the band storage requires extra kl rows at the top, marked by '+' in Figure 2, for the fill-in resulting from the partial pivoting during the factorization. Elements marked by '$*$' are not referenced.

4 BAND ROUTINES' USER INTERFACE

Our solution accepts an array of pointers specifying the locations of each matrix, its pivot vector, and its right hand side(s). Each operation has a dedicated return code in the info array. A user-defined GPU stream or queue is also required. Below are the interface declarations of the three main functions described in this paper.

```
/* band LU factorization */
void dgbtrf_batch(int m, int n, int kl, int ku,
  double**  A_array, int lda, int** pv_array,
  int* info, int batch, gpu_stream_t stream );
/* forward/backward solve */
void dgbtrs_batch(transpose_t transA,
  int n, int kl, int ku, int nrhs,
  double** A_array, int lda, int **pv_array,
  double** B_array, int ldb, int *info,
  int batch, gpu_stream_t stream);
/* top-level API (factorize and solve) */
void dgbsv_batch(int n, int kl, int ku, int nrhs,
  double** A_array, int lda, int**pv_array,
  double** B_array, int ldb, int *info,
  int batch, gpu_stream_t stream);
```

5 BAND LU FACTORIZATION

5.1 Reference Implementation

We first begin with a reference implementation that supports any matrix size and any combination of lower/upper bandwidths. This approach is not expected to be performance efficient, but it guarantees the same numerical behavior regardless of the input characteristics. Our design is based on the memory-bound building blocks of the column-wise factorization (GBTF2). A pseudo code is shown below:

```
kv = kl + ku;
ju = 0;
for(j = 0; j < min(m, n); j++) {
  // length of current column
  km = 1 + min( kl, m-j-1 );
  pivot = IAMAX( km, A(kv, j) );
  ju = GET_UPDATE_BOUND(kl, ku, j, pivot, ju);
  SET_FILLIN(m, n, kl, ku, A, j, ju);
  // swap to the right only
  SWAP(m, n, kl, ku, A(kv, j), j, ju, pivot);
  // scale the current column
  SCAL( km-1, A(kv+1, j), 1/A(kb,j) );
  // rank-1 update
  RANK_ONE_UPDATE(m, n, kl, ku, A(kv, j), ju );
}
```

There are two main differences between the band LU factorization and a fully dense one. First, the lower factor is not stored in its final form. In order to maintain kl rows for the lower factor, the row-swapping step (due to partial pivoting) affects only the trailing submatrix, unlike a fully dense factorization which affects the whole matrix. Second, the update step does not affect the entire trailing matrix. Depending on the pivot location, the functions GET_UPDATE_BOUND and SET_FILLIN identify the columns affected in the current iteration, and set the necessary fill-in elements. The CPU manages the factorization loop, and launches the corresponding GPU kernels at each iteration. As a reference implementation following a fork-join parallel model, this approach's performance is slower than a multicore CPU solution in most cases. However, further performance optimizations are discussed below.

5.2 Fully fused factorization

Perhaps the simplest approach for achieving high performance is a fully fused factorization in the shared memory of the GPU. Fully fused factorizations have the advantage of an optimal memory traffic, since each matrix is read/written exactly once from/to the global memory. While storing the matrix in the register file could yield a faster data access, it is not straightforward to map the matrix in the band layout into the register file in an algorithm-friendly manner. First, thread ownership of matrix elements is often designed as one row per thread in one-sided factorizations [7], which enables coalesced memory accesses. However, a band layout stores a matrix row in the reverse-diagonal direction. Second, unlike dense factorizations, the length of a row is dependent on the (kl, ku) pair, and an efficient use of the register file would probably require this pair to be known at compile time. This is why we opt for a in-shared-memory factorization, where referencing the input matrix is more flexible. Our fused solution works for any (kl, ku) pair, and does not require dedicated compilation for specific band sizes. It also works for any matrix sizes, as long as the matrix fits in the shared memory of the GPU.

Since the shared memory of the GPU is as fast as an L1 cache, the factorization can be efficiently implemented by factorizing one-column at a time (no blocking techniques necessary). The design requires a minimum number of threads, not less than (kl + 1). Figure 3 shows the execution time for square matrices for (kl, ku) = (2, 3) and (10, 7), respectively.

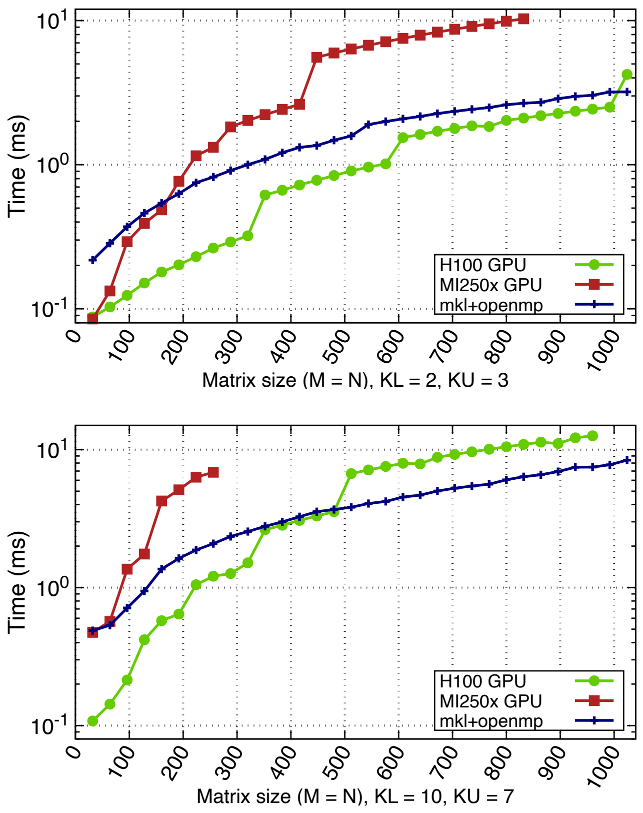


Figure 3: Execution time for the fully fused band LU factorization for (kl, ku) = (2, 3) and (10, 7), on a batch of 1,000 matrices in double precision. Results are shown for an NVIDIA H100-PCIe GPU using CUDA-12.1, an AMD MI250x GPU (single GCD) using ROCM-5.5.1, and an Intel Xeon Gold 6140 CPU (Skylake) using MKL-2023.0.1.

Despite some performance gains against the CPU solution for small sizes, larger sizes do not enjoy the same behavior. Note that the bands are relatively thin, which means that each problem lacks fine-grain parallelsim. In fact, the advantage of the GPU solution is mainly in the factorization throughput (i.e., parallelism across the batch) rather than the amount of parallel work per matrix. A fully fused solution increases the shared memory pressure for larger sizes, which in turn limits the number of resident factorizations per multiprocessor/compute-unit, thus leading to bad throughput. This explains the staircase-like behavior of our solution on both GPUs. As the sizes grow, the shared memory requirements affect the occupancy, which leads to sudden jumps in the execution time. As an example, the performance drops by almost a factor of 2× on the MI250x GPU for (kl, ku) = (2, 3) from size 416 × 416 to size 448 × 448. This is in inflection point where our fused design begins to consume more than half the available shared memory, leading to a drop in occupancy from 2 to 1, thus slowing down the performance by a similar factor. The same behavior is true for the H100 GPU, but the latter has a much larger shared memory (≈ 224 KB, compared to 64 KB on the MI210 GPU), which helps maintain a better throughput. On both GPUs the fused solution ends up being slower than the CPU solution, and even failing to run due to exceeding the shared memory capacity. This issue is addressed in the next section.

5.3 Sliding Window Factorization

In order to solve the occupancy problem, we developed a new design that significantly improves over the fully fused approach. It uses a "sliding window" technique, which caches only part of the matrix that will be affected by the current factorization iteration. This is an interesting property of the band LU factorization, which we exploit in this design. During the factorization of the j^{th} column, the last column affected by the factorization depends on the pivot location and can be expressed as ju = *max*(ju, *min*(j+ku+jp, N-1)), where jp is the current pivot location. The worst case scenario would be when jp = kl (based on a zero-based indexing), which means having, at maximum, (kv + 1) columns to update, where kv = kl + ku.

Figure 4 shows the concept of the sliding window design. It uses a tunable *blocking size* (nb), which denotes the number of columns to be factorized during a single iteration of the kernel. This "factor window" is highlighted in green in Figure 4. For a matrix with N columns, the total number of iterations required would be $\lceil \frac{N}{\text{nb}} \rceil$. These iterations can translate into either multiple kernel calls, or multiple iterations inside the same kernel while shifting the factor/update windows in shared memory. The latter approach has the better performance overall, since it avoids the kernel launch overheads, as well as some redundant global memory traffic. In terms of the shared memory requirements, our design accounts for the widest possible "update window" that contains all the columns affected by the current nb columns. However, the actual number of columns read per matrix still depends on the local value ju of every factorization. The sliding window of the kernel is, therefore, the concatenation of the factor window and the update window.

Instead of caching the entire matrix, the sliding window design needs to only cache (nb + kv + 1) columns at maximum. The sliding window size can be calculated as: (kv + nb + 1) × (kv + kl + 1), meaning that it is constant for a given band regardless of the matrix size. This is a significant reduction in resources, since we need to cache only the sliding window (a constant value) instead of the entire matrix.

The sliding window design requires a careful choice of two tuning parameters that greatly affect the performance of the factorization. The first is the blocking size (nb), and the second is the number

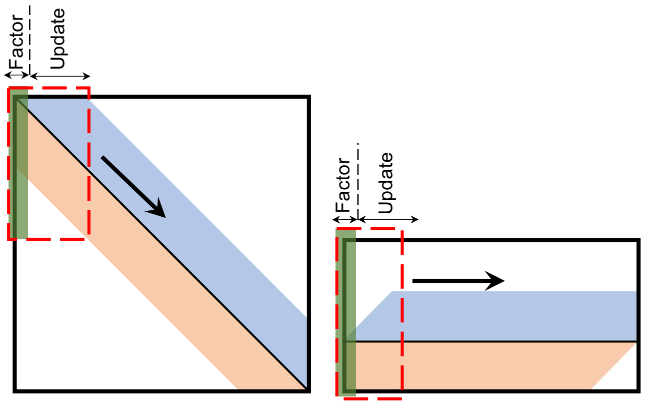


Figure 4: Illustration of the sliding window technique

of threads assigned to a single matrix. The latter has a minimum value of (kl+1), but has no upper limit. The choice of the tuning parameters mainly depends on the lower and upper bandwidths. In order to cover band sizes of interests to the applications mentioned in Section 2, we have conducted a benchmark sweep for square matrices up to 1024, for any kl/ku in the range [0:32]. The results of the benchmark sweep are then fed to a post-processing phase that extracts the best tuning parameters for a given band pattern. Separate test sweeps have been conducted for the H100 GPU and the AMD MI250x GPU.

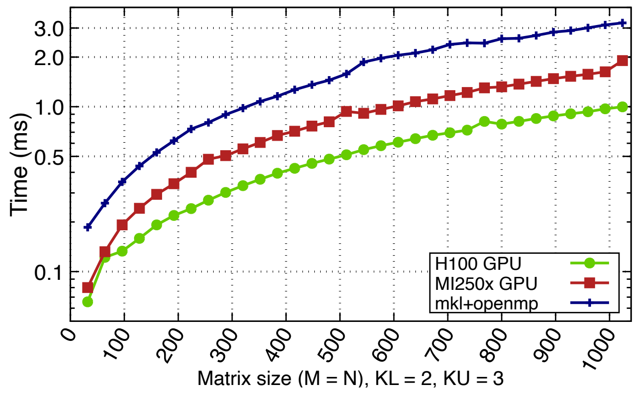
5.4 The Complete Picture

The three different designs of the band LU factorization are put together under the same interface in Section 4. In most cases, the sliding window approach is selected, since it covers a very wide range of band sizes regardless of the matrix size. However, for very small matrices (e.g., up to 64×64), the fully fused kernel has a slight advantage, since it does not have the overhead of shifting the sliding window in shared memory, which requires extra synchronization steps. The reference implementation is kept as a safe guard.

Fiure 5 shows the final performance results for the band LU factorization. For most band sizes of interest, a combination of the fused kernel and the sliding window kernel is used. The advantage of the sliding window kernel is apparent for larger sizes, maintaining an advantage over the parallel CPU solution. Table 1 shows the summary of speedups on the H100 and the MI250x GPUs against the CPU solution. We observe that larger band sizes have a greater impact on the performance of the AMD GPU, due to the small capacity of shared memory, which limits the number of resident factorizations per compute unit.

	H100-PCIe GPU			MI250x GPU		
(kl, ku)	min.	max.	avg.	min.	max.	avg.
(2, 3)	2.13×	3.43×	3.07×	1.67×	2.32×	1.88×
(10, 7)	3.07×	4.27×	3.56×	0.96×	2.01×	1.16×

Table 1: Minimum and maximum speedups of the batch band LU factorization against the parallel CPU solution.



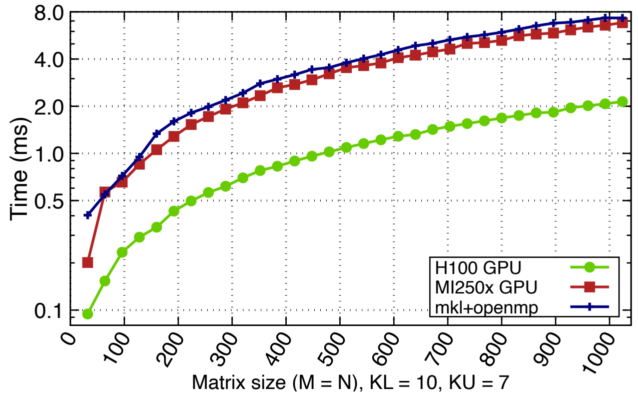


Figure 5: Execution time for the final band LU factorization for (kl, ku) = (2, 3) and (10, 7), on a batch of 1,000 matrices in double precision. Results are shown for an NVIDIA H100-PCIe GPU using CUDA-12.1, an AMD MI250x GPU (single GCD) using ROCM-5.5.1, and an Intel Xeon Gold 6140 CPU (Skylake) using MKL-2023.0.1.

6 BAND TRIANGULAR SOLVE (GBTRS)

We follow a similar approach by first providing a reference implementation that proceeds one column at a time. Note that the upper bandwidth is now equal to (kv = kl+ku). In addition, recall that the lower factor L is still stored in kl rows below the diagonal, but in order to reconstruct its dense form, each column $j \in \{0, 1, \cdots, N-1\}$ must undergo a number of row interchanges defined by the pivot entries in the interval [0:j]. However, recovering L in its final form is inefficient due to data movement and the need for an extra workspace. Instead, we apply the pivot entries progressively on the RHS matrix, coupled with rank-1 updates. For each column $j \in \{0, 1, \cdots, N-1\}$ in the lower factor, two GPU kernels perform a pair of (row swap, rank-1 updates) operations on the RHS matrix. For the upper factor, a column-wise backward triangular solver is developed.

We also develop two blocked versions of the forward/backward triangular solves. Figure 6 shows a combined view of the two solvers. Similar to the sliding window factorization, the optimized kernels perform $\lceil \frac{N}{nb} \rceil$ iterations, where at each iteration, nb columns of

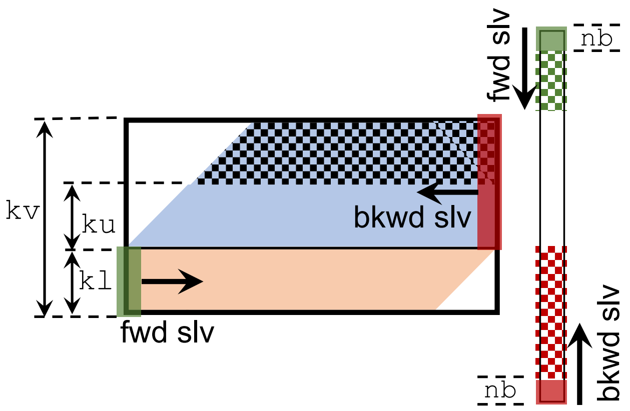


Figure 6: A combined view of the sliding window GBTRS kernels

the lower/upper factors are cached in the register file. Assuming one RHS for simplicity, the forward solver begins with the first nb columns of the lower factor and the top nb elements of the RHS vector. The solver needs to cache at most (nb + kl) elements from the RHS vector in shared memory to accommodate all the pivoting and rank-1 updates of the nb columns of L. After all the updates are performed, the top nb elements of the RHS vector are written back to the global memory, and the remaining elements are shifted up in shared memory for the next nb columns of L. For the backward solve, a similar approach is taken except that the solver begins with that last nb columns of the upper factor and the bottom nb elements of the RHS vector. At each iteration, it solves the bottom nb elements, but needs to cache at most (nb + kv) elements to accommodate the necessary updates. The kernel writes the bottom nb elements into global memory, shifts down the remaining elements from the RHS vector, and proceeds to the next iteration.

7 BAND FACTORIZATION AND SOLVE (GBSV)

The LAPACK standards define GBSV as a driver routine that calls the factorization (GBTRF) followed by the triangular solves (GBTRS). This is partially the case in our design, except that small sizes are handled using a single kernel that performs the factorization and the solve in a single context. This obviously maximizes the data reuse and the bandwidth utilization for very small sizes. Similar to a previous work on the fully dense GESV routine [11], the fused GBSV kernel performs the band LU factorization on the augmented matrix [A|B] in shared memory, which implicitly performs the forward triangular solve. After the factorization is complete, the backward solve is performed in the shared memory as well. Figure 7 shows the advantage of using a fused GBSV kernel against performing the factorization and solve separately. Note that, depending on the matrix size and the bandwidth, a fused implementation might not maintain its advantage over a standard approach. Based on our empirical results, we enable the fused kernel for systems with order 64 or less, and for a single right hand side.

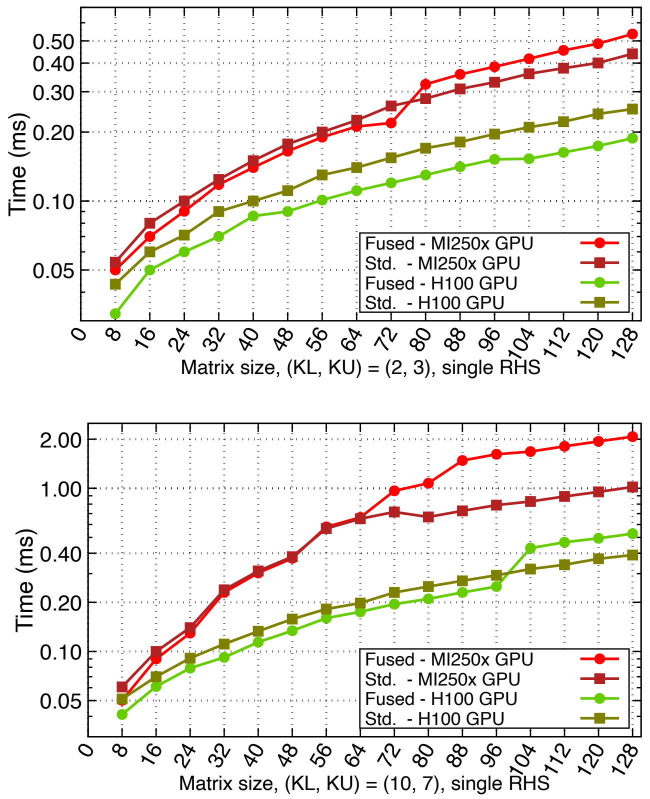
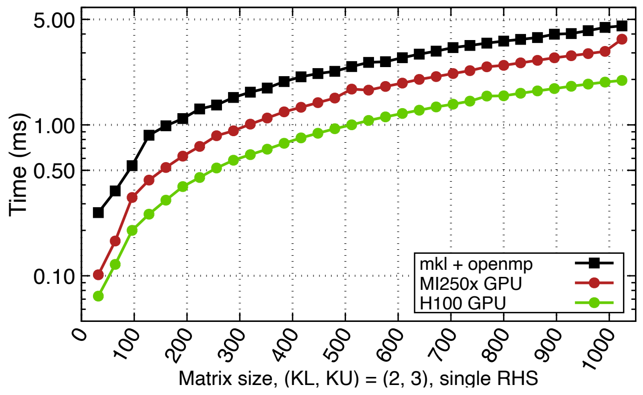


Figure 7: Performance comparison of a fully fused GBSV kernel versus the standard approach of separate factorization and solve. Results are shown for batch of 1,000 matrices in double precision on an NVIDIA H100-PCIe GPU using CUDA-12.1, and an AMD MI250x GPU (single GCD) using ROCM-5.5.1.

8 FINAL PERFORMANCE RESULTS FOR GBSV

Figure 8 shows the final GBSV performance when solving for a single right hand side. The relative speedups against the CPU solution are shown in Table 2. In most cases, the GPU solution is better than the CPU solution. However, the CPU remains a close competitor for AMD GPUs, especially for larger lower/upper bandwidths. The main reason is that band matrices do not have sufficient parallelism within a single problem. Most of the performance gain against the CPU is based on how many factorization/solves can be executed concurrently on the GPU. Our design choice is based on shared memory blocking, so the shared memory capacity plays a pivotal role on the level of concurrency. This explains the performance gap between the H100 GPU and the MI250x GPUs.

To emphasize the previous point, we compare the memory bandwidths of both GPUs. By running very large dense matrix vetor products (GEMV), we are able to estimate the sustained peak memory bound on both GPUs. The H100-PCIe GPU achieves 47% higher bandwidth, scoring about 1.92TB/s, versus 1.31TB/s for a single GCD of the MI250x GPU. For memory-bound kernels such as the



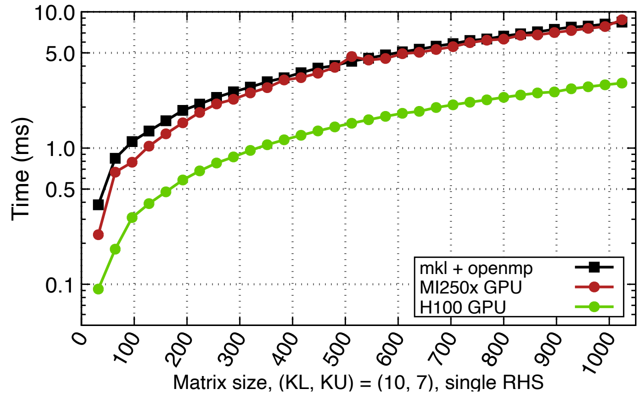


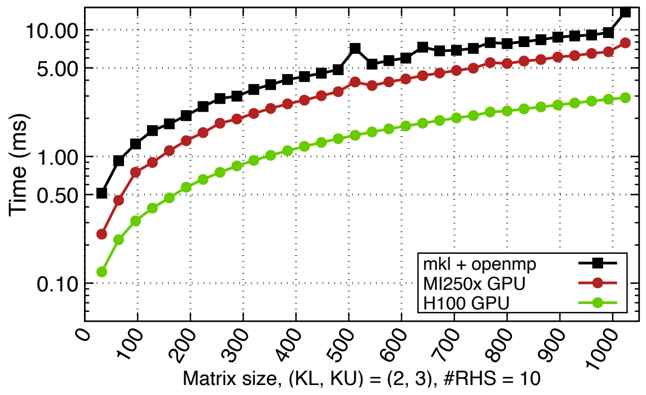
Figure 8: Final execution time of the GBSV routine solving for a single right hand side. Results are shown for batch of 1,000 matrices in double precision on an NVIDIA H100-PCIe GPU using CUDA-12.1, an AMD MI250x GPU (single GCD) using ROCM-5.5.1, and an Intel Xeon Gold 6140 CPU (Skylake) using MKL-2023.0.1.

batch GBTRF/GBSV, we can assume that the memory bandwidth is the main factor affecting the performance on GPUs. However, our solution on the H100 GPU is up to 1.88× faster than the MI250x GPU for (kl, ku) = (2, 3), and up to 3.68× for (kl, ku) = (10, 7). These speedups are much larger than the bandwidth difference, which indicates that another factor plays an important role in the performance gap. We believe that the shared memory capacity is the critical factor impacting the performance on the MI250x GPU, since its shared memory is 3.5× smaller than the H100 GPU. Other factors that could impact the performance include the shared memory bandwidth and the compiler overhead.

Figure 9 shows sample performance results when solving for multiple right hand sides (#RHS = 10 in this case). The relative speedups are shown in Table 3. Our best results remain on the H100 GPU. However, we observe that the MKL-based solution suffers a sharp increase in the execution time that averages around 2.18× for (kl, ku) = (2, 3) and 1.93× for (kl, ku) = (10, 7). In most cases, however, both GPUs do not experience the same level of performance drop. On average, the execution time on the H100 GPU has increased by 49% for (kl, ku) = (2, 3), and by 25% for (kl, ku) = (10, 7). On the MI250x GPU, while the average increase in execution time is 2.19× for (kl, ku) = (2, 3), it was recorded only at 1.33× for (kl, ku) = (10, 7).

	H100-PCIe GPU			MI250x GPU		
(kl, ku)	min.	max.	avg.	min.	max.	avg.
(2, 3)	2.23×	3.58×	2.54×	1.22×	2.58×	1.59×
(10, 7)	2.79×	4.65×	3.03×	0.92×	1.66×	1.11×

Table 2: Speedup summary of the GPU-accelerated GBSV design versus the parallel CPU solution. Results are shown for (kl, ku) = (2, 3) and (10, 7) using a single right hand side.



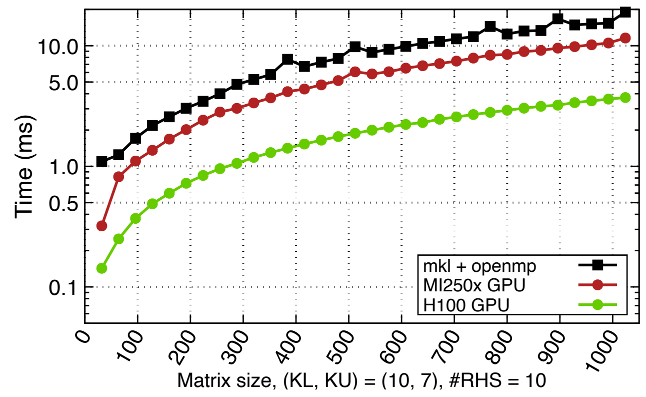


Figure 9: Final execution time of the GBSV routine solving for ten right hand sides. Results are shown for batch of 1,000 matrices in double precision on an NVIDIA H100-PCIe GPU using CUDA-12.1, an AMD MI250x GPU (single GCD) using ROCM-5.5.1, and an Intel Xeon Gold 6140 CPU (Skylake) using MKL-2023.0.1.

8.1 Discussion

According to the description and the performance of the proposed solver, it is clear that it does not take full advantage of the register file, although it is usually larger than the shared memory on modern GPUs. This indeed becomes a limiting factor on hardware with relatively small shared memory. As mentioned earlier,

	H100-PCIe GPU			MI250x GPU		
(kl, ku)	min.	max.	avg.	min.	max.	avg.
(2, 3)	3.33×	4.85×	3.69×	1.40×	2.11×	1.57×
(10, 7)	4.12×	7.67×	4.64×	1.42×	3.41×	1.61×

Table 3: Speedup summary of the GPU-accelerated `GBSV` design versus the parallel CPU solution. Results are shown for (`kl`, `ku`) = (2, 3) and (10, 7) using ten right hand sides.

caching the matrix in the register file would be a non-trivial task, and would probably require (`kl`, `ku`) to be known at compile-time in order to guarantee efficient indexing and avoid spilling. However, it is impractical to compile (**`KL`**×**`KU`**) kernel instances for every pair (`kl`∈[0:**`KL`**-1], `ku`∈[0:**`KU`**-1]). For example, if **`KL`** = **`KU`** = 15, there are 256 kernel instances to compile. Instead, we can use Just-In-Time (JIT) compiler technology, such as `nvrtc` and `hiprtc` to provide the capability of building a more optimized kernel for a specific band structure. This, however, requires more intervention from the user, who now has to create/destroy kernel instances based on the requirement of a given application. This can be a potential extension for this paper.

9 CONCLUSION AND FUTURE WORK

This paper presented an efficient use of GPUs to solve a batch of linear systems that are described using matrices with band structures. The solver uses a band LU factorization with partial pivoting, and supports arbitrary problem sizes and bandwidths. To the best of our knowledge, no similar functionality exists in the vendor's software stack, despite the existence of applications that would benefit from it. Performance results are shown on both NVIDIA and AMD GPUs, with reasonable speedups observed against a parallel CPU solution. Future directions include a more robust tuning framework, investigating more optimizations through just-in-time compilation technology, and adding support for non-uniform batches of different sizes and/or different bandwidths.

ACKNOWLEDGMENTS

This research is supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

REFERENCES

[1] 1980-2023. *BLAS (Basic Linear Algebra Subprograms)*. http://www.netlib.org/blas

[2] 1992-2023. *LAPACK - Linear Algebra PACKage*. http://www.netlib.org/lapack

[3] Ahmad Abdelfattah, Timothy Costa, Jack Dongarra, Mark Gates, Azzam Haidar, Sven Hammarling, Nicholas J. Higham, Jakub Kurzak, Piotr Luszczek, Stanimire Tomov, and Mawussi Zounon. 2021. A Set of Batched Basic Linear Algebra Subprograms and LAPACK Routines. *ACM Trans. Math. Softw.* 47, 3, Article 21 (June 2021), 23 pages. https://doi.org/10.1145/3431921

[4] Ahmad Abdelfattah, Pieter Ghysels, Wajih Boukaram, Stanimire Tomov, Xiaoye Sherry Li, and Jack J. Dongarra. 2022. Addressing Irregular Patterns of Matrix Computations on GPUs and Their Impact on Applications Powered by Sparse Direct Solvers. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis, Dallas, TX, USA, November 13-18, 2022*, Felix Wolf, Sameer Shende, Candace Culhane, Sadaf R. Alam, and Heike Jagode (Eds.). IEEE, 26:1–26:14. https://doi.org/10.1109/SC41404.2022.00031

[5] Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack Dongarra. 2018. Batched one-sided factorizations of tiny matrices using GPUs: Challenges and countermeasures. *J. Comput. Sci.* 26 (2018), 226–236. https://doi.org/10.1016/j.jocs.2018.01.005

[6] Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack J. Dongarra. 2016. Performance, Design, and Autotuning of Batched GEMM for GPUs. In *ISC High Performance 2016, Frankfurt, Germany, June 19-23, 2016, Proceedings.* 21–38. https://doi.org/10.1007/978-3-319-41321-1_2

[7] Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, and Jack J. Dongarra. 2017. Factorization and Inversion of a Million Matrices using GPUs: Challenges and Countermeasures. In *International Conference on Computational Science, ICCS 2017, 12-14 June 2017, Zurich, Switzerland.* 606–615. https://doi.org/10.1016/j.procs.2017.05.250

[8] Ahmad Abdelfattah, Stan Tomov, and Jack J. Dongarra. 2022. Batch QR Factorization on GPUs: Design, Optimization, and Tuning. In *Computational Science - ICCS 2022 - 22nd International Conference, London, UK, June 21-23, 2022, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13350)*, Derek Groen, Clélia de Mulatier, Maciej Paszynski, Valeria V. Krzhizhanovskaya, Jack J. Dongarra, and Peter M. A. Sloot (Eds.). Springer, 60–74. https://doi.org/10.1007/978-3-031-08751-6_5

[9] A. S. Almgren, J. B. Bell, P. Colella, L. H. Howell, and M. L. Welcome. 1998. A Conservative Adaptive Projection Method for the Variable Density Incompressible Navier-Stokes Equations. *J. Comp. Phys.* 142 (1998), 1–46.

[10] Wajih Halim Boukaram, George Turkiyyah, Hatem Ltaief, and David E. Keyes. 2017. Batched QR and SVD algorithms on GPUs with applications in hierarchical matrix compression. *Parallel Comput.* (2017). https://doi.org/10.1016/j.parco.2017.09.001

[11] Chiang-Heng Chien, Hongyi Fan, Ahmad Abdelfattah, Elias P. Tsigaridas, Stanimire Tomov, and Benjamin B. Kimia. 2022. GPU-Based Homotopy Continuation for Minimal Problems in Computer Vision. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2022, New Orleans, LA, USA, June 18-24, 2022.* IEEE, 15744–15755. https://doi.org/10.1109/CVPR52688.2022.01531

[12] M. S. Day and J. B. Bell. 2000. Numerical Simulation of Laminar Reacting Flows with Complex Chemistry. *Combust. Theory Model* 4, 4 (2000), 535–556.

[13] Azzam Haidar, Tingxing Dong, Piotr Luszczek, Stanimire Tomov, and Jack J. Dongarra. 2015. Batched matrix computations on hardware accelerators based on GPUs. *Int. J. High Perform. Comput. Appl.* 29, 2 (2015), 193–208. https://doi.org/10.1177/1094342014567546

[14] Marc T Henry de Frahan, Jon S Rood, Marc S Day, Hariswaran Sitaraman, Shashank Yellapantula, Bruce A Perry, Ray W Grout, Ann Almgren, Weiqun Zhang, John B Bell, and Jacqueline H Chen. 2022. PeleC: An adaptive mesh refinement solver for compressible reacting flows. *The International Journal of High Performance Computing Applications* OnlineFirst, Open Access (2022), 10943420221121151. https://doi.org/10.1177/10943420221121151

[15] Konstantin Herb and Pol Welter. 2022. Parallel time integration using Batched BLAS (Basic Linear Algebra Subprograms) routines. *Computer Physics Communications* 270 (2022), 108181. https://doi.org/10.1016/j.cpc.2021.108181

[16] Ian Masliah, Ahmad Abdelfattah, Azzam Haidar, Stanimire Tomov, Marc Baboulin, Joël Falcou, and Jack 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.* 659–671. https://doi.org/10.1007/978-3-319-43659-3_48

[17] A. Nonaka, J. B. Bell, and M. S. Day. 2018. A conservative, thermodynamically consistent numerical approach for low Mach number combustion. I. Single-level integration. *Combust. Theor. Model.* 22, 1 (2018), 156–184.

[18] A. Nonaka, J. B. Bell, M. S. Day, C. Gilet, A. S. Almgren, and M. L. Minion. 2012. A Deferred Correction Coupling Strategy for Low Mach Number Flow with Complex Chemistry. *Combust. Theory and Model* 16, 6 (2012), 1053–1088.

[19] Villa Oreste, Massimiliano Fatica, Nitin A. Gawande, and Antonino Tumeo. 2013. Power/Performance Trade-offs of Small Batched LU Based Solvers on GPUs. In *Euro-Par 2013 (Lecture Notes in Computer Science, Vol. 8097).* Aachen, Germany, 813–825.

[20] R. B. Pember, L. H. Howell, J. B. Bell, P. Colella, W. Y. Crutchfield, W. A. Fiveland, and J. P. Jessee. 1998. An Adaptive Projection Method for Unsteady, Low-Mach Number Combustion. *Comb. Sci. Tech.* 140 (1998), 123–168.

[21] A. Y. Sharma, M. D. J. Cole, T. Görler, Y. Chen, D. R. Hatch, W. Guttenfelder, R. Hager, B. J. Sturdevant, S. Ku, A. Mishchenko, and C. S. Chang. 2022. Global gyrokinetic study of shaping effects on electromagnetic modes at NSTX aspect ratio with ad hoc parallel magnetic perturbation effects. *Physics of Plasmas* 29, 11 (Nov. 2022), 112503. https://doi.org/10.1063/5.0106925 arXiv:https://pubs.aip.org/aip/pop/article-pdf/doi/10.1063/5.0106925/16627444/112503_1_online.pdf

[22] Hariswaran Sitaraman, Shashank Yellapantula, Marc T. Henry de Frahan, Bruce Perry, Jon Rood, Ray Grout, and Marc Day. 2021. Adaptive mesh based combustion simulations of direct fuel injection effects in a supersonic cavity flame-holder. *Combustion and Flame* 232 (2021), 111531. https://doi.org/10.1016/j.combustflame.2021.111531

[23] Weiqun Zhang, Ann Almgren, Vince Beckner, John Bell, Johannes Blaschke, Cy Chan, Marcus Day, Brian Friesen, Kevin Gott, Daniel Graves, Max P. Katz, Andrew Myers, Tan Nguyen, Andrew Nonaka, Michele Rosso, Samuel Williams, and Michael Zingale. 2019. AMReX: a framework for block-structured adaptive mesh refinement. *Journal of Open Source Software* 4, 37 (2019), 1370. https://doi.org/10.21105/joss.01370 https://github.com/AMReX-Codes/amrex.