Efficient solution of batched band linear systems on GPUs

Journal Title
XX(X):1–12

DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE

Pratik Nayak[1] Isha Aggarwal[1] Hartwig Anzt[2]

Abstract

The data-parallel solution of sparse linear systems sharing the same band structure with the help of batched band solvers is relevant in many applications. In this paper, we elaborate on three new GPU algorithms for the data-parallel direct solution of linear system batches, sharing a band structure. We develop algorithms for three types of matrices: tri-diagonal matrices, small bandwidth matrices, and wide bandwith matrices. We implement these algorithms for NVIDIA GPUs and study the performance for a large range of matrix sizes and number of batch items. We compare the performance of our implementations with the corresponding vendor implementations, with the state-of-the-art GPU library MAGMA, and with the optimized LAPACK implementation provided by Intel MKL on Intel Skylake CPUs. We also showcase the effectiveness of our batched band solvers for matrices originating from the plasma physics application, XGC. We show that our implementations are on average $\sim 2\times$ (for batched banded solvers) to $\sim 3\times$ (for batched tridiagonal solvers) faster than the state-of-the-art and the vendor provided implementations.

Keywords

batched algorithms, band linear systems, GPU computing, direct solvers, LU factorization, partial pivoting

1 Introduction

Many computational science applications such as plasma physics and combustion simulations use ODEs for modeling the physics and the chemical processes in a grid of cells covering the simulation domain. The solution of the ODEs for the distinct cells results in many linear systems of small size that all share the same sparsity pattern. Often, the discretization is based on finite difference schemes, resulting in this shared sparsity pattern being of band structure. To evolve the physics in space-time (Hindmarsh (2002)), a non-linear solver is required. For robustness and stability, implicit methods are usually preferred, and a linearization step necessitates the solution of a sequence of linear systems.

For applications that need to solve for independent, non-coupled variables such as concentration of chemical species, the linear systems for the distinct cells are independent. Therefore, in each non-linear solve, there is a need for solver functionality that can efficiently handle a large number of (relatively small) independent linear systems. For the efficient solution of these independent linear systems, batched methods have been developed (Dongarra et al. (2016)).

Batched methods are well-suited for the hierarchical parallelism provided by many-core architectures such as GPUs. Mapping the independent linear system solutions to a coarser level enables synchronization-free computing, while the finer level of parallelism can still be utilized to accelerate the solution of the individual linear systems (Nayak (2023)).

With GPUs providing most of the computing power in the latest supercomputers, it is important to deploy algorithms that harness the massive parallelism provided by these architectures. In this paper, we present three different algorithms that enable efficient solution of band-structure linear systems on GPUs. Our contributions are:

1. General batched band matrix solvers for GPUs that outperform existing state-of-the-art CPU and GPU solvers.
2. A new variant of a blocked band solver that enables the efficient solution for wide bandwith matrices.
3. A batched tri-diagonal solver that significantly outperforms the vendor libraries.
4. A detailed performance evaluation of the GPU band solvers for randomly generated matrices and an analysis investigating the performance impact of partial pivoting. Additionally, we consider matrices originating from a plasma physics application, XGC, to showcase the performance of our batched band solver in real-world applications.

In Section 2, we provide a brief background of band matrices, their relevance in scientific computing applications, and some related work. In Section 3, we provide details of our band and tri-diagonal solver algorithms, including implementation and optimization details that enable efficient computation on GPUs. We evaluate the performance of our band solvers and benchmark them against state-of-the-art band solvers in Section 4. We finally conclude with a brief summary in Section 5.

2 Background and Related Work

Direct methods for the solution of linear systems compute the solution via a fixed and pre-defined sequence of operations.

[1]Karlsruhe Institute of Technology, Germany;
[2]Technical University of Munich, Germany;.

Most direct solvers initially factorize the coefficient matrix in the factorization step and then solve the linear system with forward and backward substitution. For a general matrix of size n, the factorization step has a computational cost of $\mathcal{O}(n^3)$, which makes the approach expensive and thus unattractive for large problems. Iterative methods that compute a sequence of solution approximations with increasing accuracy are thus often preferred. However, direct solvers are still attractive if the linear system is of small size, very ill-conditioned, or if the factorized matrix can be re-used. The most common general direct solver is the LU factorization combined with the triangular solves (Duff et al. (2017)).

Algorithms for the factorization and subsequent solution can account for the sparsity pattern and symmetry of the matrix. While for small matrices, using a dense storage format is acceptable, storing large matrices in the dense format can be inefficient or even impossible if the memory requirements exceed the hardware resources. Efficient algorithms have been developed for these cases that handle the coefficient matrix and the factorization in sparse matrix formats such as CSR (Duff et al. (2017); Li and Demmel (2003)). When using sparse data formats, the factorization step is typically split into two stages, a symbolic phase where the row dependencies and fill-in are determined, and a numeric phase where the actual factorization and the numeric values are computed.

The sparsity pattern of the generated matrix depends on the application. Many applications require the solution of linear systems whose matrices have a band structure, for example, those that arise from a stencil discretization. Band matrices are characterized by two main parameters: the lower bandwidth, β_L, and the upper bandwidth, β_U, as shown in Figure 1 with $(\beta_L, \beta_U) = (3, 1)$.

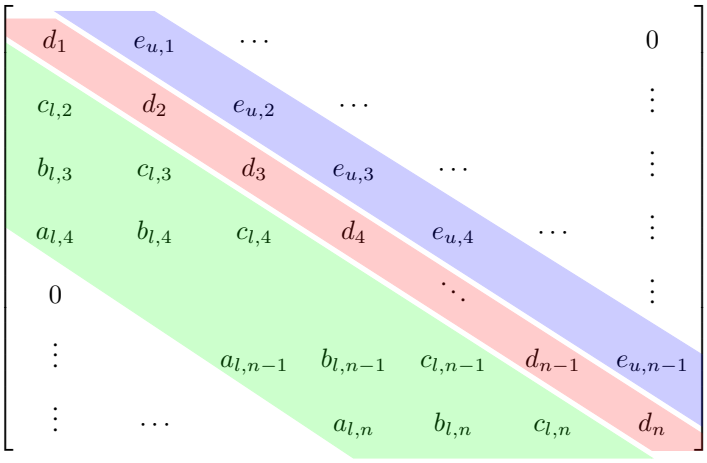


Figure 1. Band matrix ($\mathbb{R}^{n\times n}$) with β_L =3, β_U =1

There have been efforts in developing algorithms for data-parallel basic linear algebra functionality (BLAS) and advanced solver algorithms (LAPACK). The data-parallel versions of the functionality are generally called "batched" to indicate the data-parallel processing of a batch of linear algebra objects. A batched BLAS interface has also been recently proposed (Dongarra et al. (2016, 2017)) to standardize the interface to batched basic linear algebra functionality across hardware architectures and software stacks. This standardization effort for batched functionality has also been expanded to LAPACK, providing direct solvers for dense linear systems (Abdelfattah et al. (2021)).

In terms of batched direct solvers for sparse problems, there has been some work on tri-diagonal and penta-diagonal systems (Valero-Lara et al. (2018); Carroll et al. (2021); Gloster et al. (2019)), and the NVIDIA cuSPARSE library provides the **`gtsv2StridedBatch`** routine based on variants of cyclic reduction.

These methods aim to solve tri-diagonal and penta-diagonal systems and rely on Thomas' algorithm or cyclic reduction (Valero-Lara et al. (2018)). These implementations are not fine-grain parallel, but each GPU thread solves an entire linear system. The performance benefits, therefore, primarily come from storing the problem data in an interleaved fashion to enable coalesced data access. In the context of the Human Brain Project (Valero-Lara et al. (2017)), methods similar to *cuThomasBatch* (Valero-Lara et al. (2018)) have been developed to accelerate the solution of batched Hines systems on NVIDIA GPUs. There also exists some work on the efficient solution of batched penta-diagonal systems (Gloster et al. (2019); Carroll et al. (2021)), however, in these attempts, the factorization step is performed on the CPU, necessitating data transfer for the triangular solves. Within a non-batched setting, GPU-based tri-diagonal solvers have also received some attention (Klein and Strzodka (2021); Pérez Diéguez et al. (2018)).

The development of direct solvers for band matrices with an arbitrary number of sub- and super- diagonals, (β_L, β_U) has received very little attention. The only implementation that performs a GPU-resident factorization and solve for band matrices that the authors could find at the time of writing this paper is the implementation available in the MAGMA library (Abdelfattah et al. (2023)). For the CPU, LAPACK provides efficient routines (`xgbsv`) (Blackford and Dongarra (1991)) that can be launched in a data-parallel fashion by launching a solver routine on each of the CPU cores and each solver instance solving a subset of the linear systems in the batch. Additionally to the high-level parallelization across the systems in the batch, vendor implementations of LAPACK such as Intel MKL (Intel (2023)) may use SIMD vectorization to harness the available parallelism within a single CPU core for increasing the performance of the solver instances launched on the distinct cores.

3 GINKGO's batched band solvers

GINKGO is a high-performance numerical linear algebra library that implements efficient and performance portable linear solvers and preconditioners (Anzt et al. (2022)). Batched iterative solvers and preconditioners have been added recently to accelerate science applications such as combustion simulations (Aggarwal et al. (2021)) and plasma physics (Kashi et al. (2023)). In this paper, we implement GPU-resident batched band solvers for arbitrary band sizes and specializations for wide bands and tri-diagonal matrices.

3.1 Batched band matrix storage format

Unlike general sparse matrices, the fill-in for band-structure matrices is limited to within the band if no pivoting is

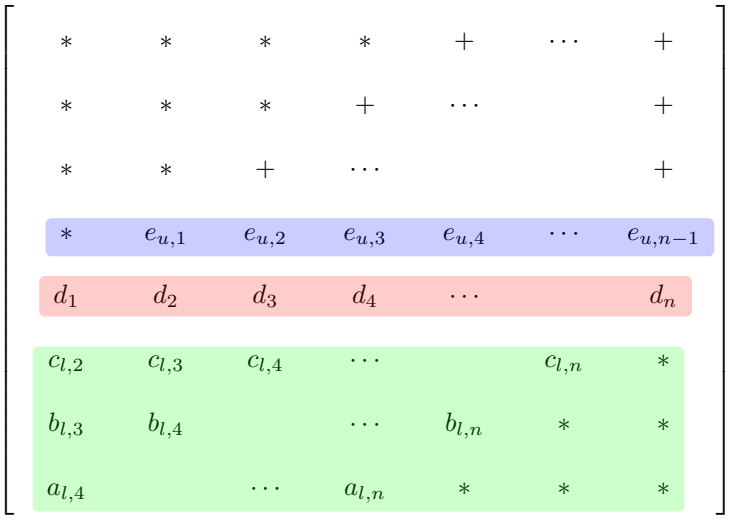


Figure 2. Band storage ($\mathbb{R}^{\beta_{st} \times n}$) : Number of elements stored = $\beta_{st} \times n$.

used, and the fill-in with partial pivoting is also well-defined, with given lower and upper bandwidths. Storing these band matrices in a tailored format and using their structure to compute the factorization can be advantageous. LAPACK (Anderson (1999)) introduced an efficient storage format for band-structure matrices and a customized solver operating on the band matrix storage format in the LU factorization and the subsequent triangular solves. We visualize the band matrix storage format in Figure 2. As partial pivoting is necessary to ensure stability for a general linear system solver, the storage format allocates additional memory for possible fill-in during the factorization. In Figure 2, we see these locations marked by "+". The extra padding (elements only stored to ensure uniform strided accesses) is marked with "∗". The total number of bands stored, each of size n, is equal to $\beta_{st} = 2\beta_L + \beta_U + 1$. This includes the extra storage required for possible fill-in introduced by pivoting.

GINKGO also employs the standard LAPACK band matrix format as shown in Figure 2. To ensure coalesced access, GINKGO stores the matrix band arrays in a column-major format without interleaving the elements of different matrices. This has proven to be efficient for both GPU and CPU implementations of band solvers (Abdelfattah et al. (2023)).

3.2 Batched direct solves for band matrices

With the band matrix stored as shown in Figure 2, the batched band direct solver consists of two steps: An LU factorization that decomposes a band matrix of the batch into triangular factors, and the subsequent triangular solves using the generated triangular band factors. The batched band direct solver is shown in Algorithm 1.

As our focus is on small problems and the goal is to solve thousands to hundreds of thousands of small problems in parallel, we map one linear system solve (factorization + triangular solves) to one workgroup (thread block) of the GPU. As workgroups have no data dependencies, this allows for the parallel and independent processing of the problems,

Algorithm 1 The batched band direct solver.

```
INPUT: A, b
OUTPUT: x
for i ∈ [0, n_batch_items) do
    [A_i, b_i, x_i] ← get_batch_item( i, A, b, x )
    [A_i(:= L_i U_i), pivots] ← lu_factorization(A_i)
    y_i ← lower_trsv(A_i, b_i, pivots)
    x_i ← upper_trsv(A_i, y_i)
end for
```

and global memory reads and writes are conflict-free, thereby reducing the global memory traffic.

Due to the reduced amount of parallelism available in the band solver, large work-group sizes would lead to warp-stalling. We therefore tune our workgroup sizes based on the problem size and the matrix bandwidth. An additional advantage of small workgroup sizes is that more batch items can be processed concurrently, allowing for higher throughput.

For the efficient solution of batched band linear systems, we perform the triangular solves for each system immediately after the factorization. That is, once a band matrix is loaded into the multiprocessor memory, we first perform in-place factorization and then the forward and backward triangular solves in shared memory without writing intermediate results to the GPU main memory. While this requires consolidating both functionalities into a single GPU kernel it radically reduces the main memory access. In the end, we launch a single kernel handling the solution of all linear systems in the batch.

Next, we provide more details on our generic band solver implementation. Afterward, we present the implementations tailored towards matrix batches with wide bands and batches composed of tri-diagonal matrices.

3.2.1 *Batched LU factorization for band matrices*

The LU factorization of a band matrix batch is shown in Algorithm 2. For each batch item, we process each column individually. First, we identify the pivot and process the row swaps. We then proceed by scaling the columns and finish by updating the trailing matrix. After completion, we have an in-place factorization of the input band-matrix. We note that as a result of the storage scheme shown in Figure 2, we do not need to perform a structural analysis or allocate additional memory to account for fill-in.

Templating each kernel on a compile-time subwarp size, we can maximize SIMD parallelism and ensure that accesses to the band matrix are coalesced.

Algorithm 2 The LU factorization for band-structure matrices.

```
INPUT: A_i, n_rows, β_L, β_U
OUTPUT: In-place factorization: A_i := LU, pivots
for col ∈ [0, n_rows) do
    piv ← find_pivot_row(A_i, col)
    end_col ← max(end_col, min(piv + β_U, n_rows − 1 ))
    if piv != diag then
        swap_rows(col, end_col, piv, A_i)
    end if
    scale_col(col, A_i)
    update_trailing_matrix(col, end_col, A_i)
end for
```

3.2.2 Batched triangular solves After computing a factorization, we need to perform two triangular solves.

$$\begin{aligned} & (LU)x = b \\ 1: \quad & Ly = b \ \text{, Lower } \texttt{trsv} \\ 2: \quad & Ux = y \ \text{, Upper } \texttt{trsv} \end{aligned} \tag{1}$$

With partial pivoting (only row interchanges), the lower triangular solver uses the pivoting array computed in the factorization step, while pivoting is not required for the upper triangular solve.

We process both triangular solves in a row-by-row fashion, with the upper triangular solve processing elements in a bottom-up fashion (backward substitution) and the lower triangular solve processing elements in a top-down order (forward substitution).

3.2.3 Efficient solution of wide-band matrices For matrices with small to medium bandwidths ($(\beta_L + \beta_U) < 32$), processing each column is efficient, as it enables coalesced access when performing the scaling and the trailing matrix updates. For matrices with a wide band, due to the larger number of elements that need to be updated in each row, performing a blocked factorization and processing ϕ columns at once, where ϕ is the panel size, can be advantageous. A schematic of our panel-based factorization is shown in Figure 3 for a panel size of 2.

We partition the matrix into block-diagonal ($A_{00} \in \mathbb{R}^{\phi \times \phi}$), trailing row block ($[A_{01}\, A_{02}] \in \mathbb{R}^{\phi \times (\phi_{1j}+\phi_{2j})}$), trailing column block ($\begin{bmatrix} A_{10} \\ A_{20} \end{bmatrix} \in \mathbb{R}^{(\phi_{1k}+\phi_{2k}) \times \phi}$) and a trailing block-diagonal block ($\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \in \mathbb{R}^{(\phi_{1k}+\phi_{2k}) \times (\phi_{1j}+\phi_{2j})}$) as shown in Equation (2), with $\phi_{1k}, \phi_{2k}, \phi_{1j}, \phi_{2j}$ as defined in Algorithm 3 (the trailing size of the blocks).

$$A = \begin{bmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{bmatrix} \tag{2}$$

The panel LU factorization algorithm, shown in Algorithm 3, proceeds by first factorizing the column panel $\begin{bmatrix} A_{00} \\ A_{10} \\ A_{20} \end{bmatrix}$. We compute the row pivots, and update the trailing block columns A_{10} and A_{20}. Using the computed pivots, we apply the necessary row interchanges in a block-column fashion. Using a trsm and trailing column update, we update the block columns for A_{01} and A_{02}.

We proceed in this fashion recursively for the trailing block diagonal matrix until the entire matrix has been factorized. As the factors replace the original matrix in memory, we consider it an in-place factorization.

To ensure coalesced accesses when processing ϕ columns at a time, we utilize a workspace array to copy the matrix blocks that tend to have non-coalesced accesses, namely A_{02} and A_{20}. We omit this implementation detail from Algorithm 3 for simplicity.

3.3 Efficient solution of tri-diagonal matrices

Tri-diagonal matrices are a special case of band matrices with $(\beta_L, \beta_U) = (1, 1)$. For these systems, it is not necessary

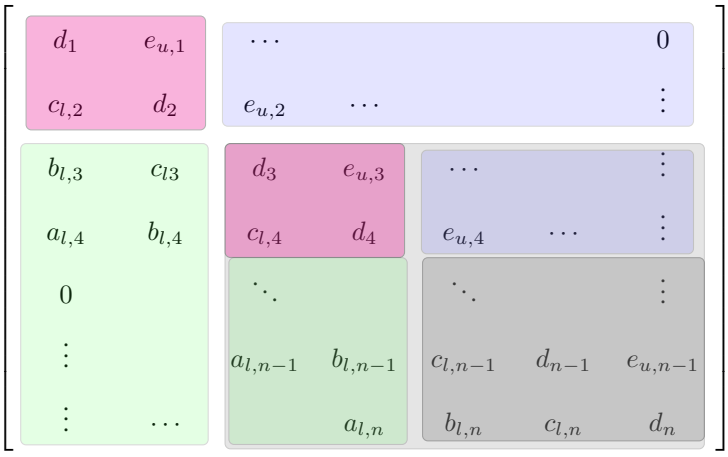


Figure 3. Panel factorization for band matrix ($\in \mathbb{R}^{n_{rows} \times n_{rows}}$) with β_L = 3, β_U = 1, ϕ = 2.

Algorithm 3 The panel-based LU factorization for band-structure matrices.

INPUT: $A_i, n_{rows}, \beta_L, \beta_U, \phi$
OUTPUT: $Factorized(in-place) A_i (= L_i U_i)$
for j ∈ $[0, \phi, n_{rows})$ **do**
  $\phi_j \leftarrow \texttt{min}(\phi, n_{rows} - j)$
  $\phi_{1k} \leftarrow \texttt{min}(\beta_L - \phi_j, n_{rows} - j - \phi_j)$
  $\phi_{2k} \leftarrow \texttt{min}(\phi_j, n_{rows} - j - \beta_L)$
  for k ∈ $[j, j + \phi_j)$ **do**
    piv ← find_pivot_row(A_i, k)
    end_col ← max(end_col, min(piv + β_U, $n_{rows} - 1$))
    if piv != k **then**
      swap_rows(k, end_col, piv, A_i)
    end if
    pivots[k] ← piv
    update_trailing_matrices(k, j+$\phi_j - 1$, $A_{i,10}$, $A_{i,20}$)
  end for
  $\phi_{1j} \leftarrow \texttt{min}(\text{end\_col} - j + 1, \beta_L + \beta_U) - \phi_j$
  $\phi_{2j} \leftarrow \texttt{max}(0, \text{end\_col} - j + 1 - \beta_L - \beta_U)$
  apply_row_interchanges(pivots, $A_{i,01}, A_{i,11}, A_{i,21}$)
  apply_row_interchanges(pivots, $A_{i,02}, A_{i,12}, A_{i,22}$)
  trsv_and_update($A_{i,01}, A_{i,11}, A_{i,21}$)
  trsv_and_update($A_{i,02}, A_{i,12}, A_{i,22}$)
end for

to compute an LU factorization, but we can directly use the Thomas algorithm to solve the linear system (Valero-Lara et al. (2018)). A recursive divide-and-conquer approach allows for a high level of concurrency (Wang and Mou (1991)). We use this approach for the batched tri-diagonal solver and show that our implementation is faster than both the vendor-provided tri-diagonal solver implementation and the general band solver implementation.

The algorithm we use (shown in Algorithm 4) is a variant of the standard parallel Gaussian Elimination (GE) for tri-diagonal matrices (Wang and Mou (1991)). The idea is to first merge adjacent rows into groups, with a partial GE, removing dependencies between the adjacent groups. We then obtain groups that can be eliminated in parallel with a full GE step. Once all groups have been eliminated with a forward GE, we can perform a backward substitution to obtain the final solution. A schematic of this process is shown in Figure 4 for $\rho = 2$ where ρ is the number of merge steps that is performed on the initial matrix and for a tile size, $t = 16$.

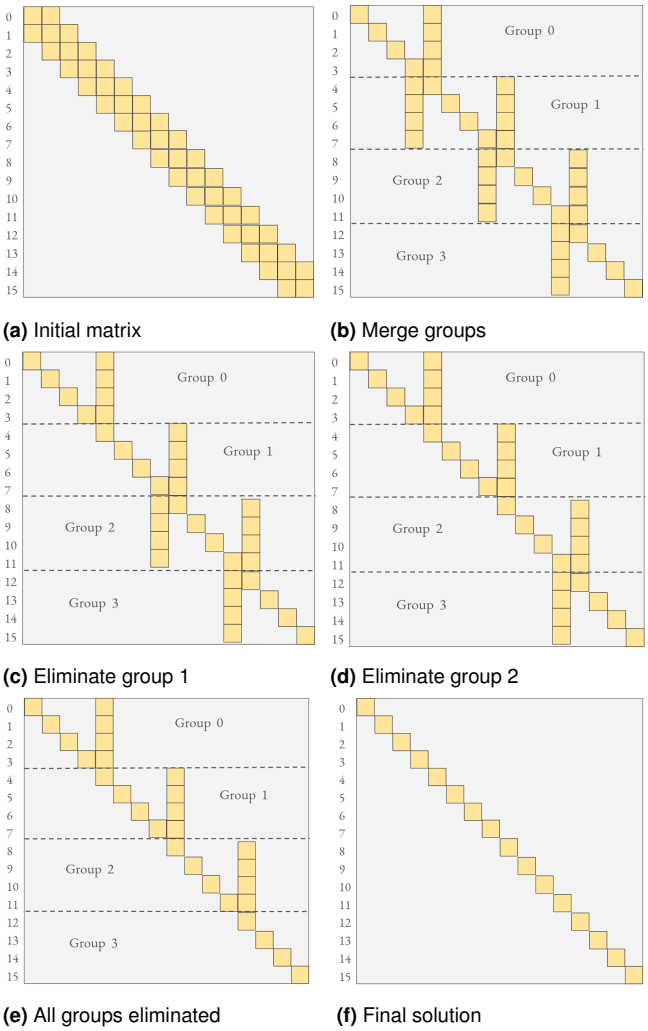
To optimize performance, we expose two control knobs to the user. The tile size, t, defines the size of the matrix tile to be handled by each subgroup, and the number of merge

steps, ρ, defines the total number of groups that are created and forward-eliminated in parallel. Additionally, the user can choose one of two strategies: Each thread in a subwarp handles one row of the matrix (`strat1`), or each thread in a subwarp handles 2 rows of the matrix (`strat2`). Therefore, we can obtain (ρ, t) from the constraints:

$$(\rho, t) \text{ s.t } \begin{cases} \texttt{strat1} & : 2^\rho \leq t \\ \texttt{strat2} & : 2^\rho \leq 2t \end{cases}$$

To simplify usage, we also provide a strategy, `auto`, that automatically sets (ρ, t) depending on the number of rows, n_{rows}, as that forms the central aspect when choosing (ρ, t).

To reduce global memory traffic, we utilize communicator-group functionality such as warp-shuffles and broadcasts (Tsai et al. (2021)). We instantiate kernels for all major subwarp sizes (1, 2, 4, 8, 16, 32) during compile time. The optimal subwarp size is selected at runtime, depending on the problem size (n_{rows}).



(a) Initial matrix **(b)** Merge groups

(c) Eliminate group 1 **(d)** Eliminate group 2

(e) All groups eliminated **(f)** Final solution

Figure 4. Divide and conquer parallel Gauss-Elimination algorithm for tri-diagonal matrices with number of merge steps, $\rho = 2$ and tile size, $t = 16$.

Algorithm 4 The batched tri-diagonal algorithm.

INPUT: $A = (trid(a_l, a_d, a_u)), b, \rho, t$
OUTPUT: x
$n_{tiles} \leftarrow$ `ceildiv`(n_{rows}, t)
for j $\in [0, n_{tiles})$ **do**
  curr_tile = `get_tile`$(a_l, a_d, a_u$, j, t, $n_{tiles})$
  curr_grp_size $\leftarrow 1$
  ▶ **Merge groups**
  if !(last_tile) `or size`(last_tile) == t **then**
    for s $\in [0, \rho)$ **do**
      `merge_adjacent_groups`(curr_tile, curr_grp_size, b_k)
      curr_grp_size *= 2
    end for
  end if
  ▶ **Forward full Gaussian elimination**
  $n_{groups} \leftarrow$ t / curr_grp_size
  for g $\in [0, n_{groups})$ **do**
    `eliminate_bottom_spike`(curr_tile, curr_grp_size, g, b_k)
  end for
end for
▶ **Backward substitution**
for j $\in [n_{tiles} - 1, 0)$ **do**
  curr_tile = `get_tile`$(a_l, a_d, a_u$, j, t, $n_{tiles})$
  final_grp_size $\leftarrow 1$
  if !(last_tile) `or size`(last_tile) == t **then**
    final_grp_size $\leftarrow$pow(2, ρ)
  end if
  $n_{groups} \leftarrow$ t / final_grp_size
  for g $\in [n_{groups} - 1, 0)$ **do**
    `backward_substitution`(curr_tile, final_grp_size, g, x_k, b_k)
  end for
end for

4 Benchmarking and performance evaluation

We use the HoreKa* supercomputer for the performance evaluation of our batched band solvers. It consists of 4 NVIDIA A100 GPUs per node and 2 sockets of Intel Xeon Platinum (Icelake) CPUs with a total of 76 CPU cores per node. The relevant hardware characteristics are shown in Table 1.

4.1 Dataset and evaluation metrics

Our dataset consists of artificially generated band matrices with random values (from a normal distribution) that are different along three main characteristics: the lower bandwidth β_L, the upper bandwidth β_U, and the diagonal dominance. For each of the experiments†, we run 2 warmup iterations to minimize the effects of startup artifacts such as library and symbol loading, data transfer, etc., and gather average timings over 5 kernel executions. We verified (post-solve) that all variants of the solvers (our solvers and the ones compared with) converged to machine precision (10^{-16}).

To ensure a fair comparison of the solvers, we exclude the setup and analysis timings and only report the timings for the solution process (factorization and solve).

We present the performance of GINKGO'S batched solvers on one A100 GPU with CUDA 11.8. For comparison with the CPU-based banded solvers, we use the LAPACK implementation from Intel MKL 2022.0.2 with the Intel C/C++ compiler 2021.5, and parallelize over all the available CPU cores (the full CPU node, 76 cores) using OpenMP. Additionally, we compare our performance with

*This work was performed on the HoreKa supercomputer funded by the Ministry of Science, Research and the Arts Baden-Württemberg and by the Federal Ministry of Education and Research.

†Reproducibility artifact: Nayak et al. (2024)

Table 1. Hardware characteristics (NVIDIA (2020) and Intel (2021)).

Arch	*Peak FP64* (TFlops)	*BW* (GB/s)	*(L1+SM) /CU* (KB)	*L2+L3* (MB)	*# of SMs /CUs*
A100-40GB	9.7	1555	192	40	108
Intel Xeon 8368 (1CPU)	1.9	204	64	95	38

MAGMA (Dongarra et al. (2014)), a state-of-the-art software that implements both batched and non-batched BLAS and LAPACK routines for GPUs.

We use MAGMA's asynchronous batched strided interface. As mentioned previously, we do not include the timings for allocation and deallocation of MAGMA's workspace. Both the MAGMA and MKL interfaces perform an in-place factorization, while GINKGO'S factorization ensures the immutability of the input matrix and copies the band matrix into a workspace before each solve. The time needed for this explicit copy is included in the overall runtime.

The number of batch items in the experiments varies with the number of nonzeros in each of the batch items. To minimize effects of the CUDA runtime, we set the number of batch items to a sufficiently high enough count that fills the GPU memory and is reflective of the application use-case.

4.2 Batched band solvers on GPUs

We first present the performance of GINKGO's batched band solvers for two sets of bandwidths, $(\beta_L,\ \beta_U)$ = (2, 3) and $(\beta_L,\ \beta_U)$ = (15, 5), representing matrices with narrow and medium bandwidths. Figure 5 shows the time to solution for the three batched band solvers: The CPU-based solver using Intel MKL, GINKGO'S batched band solver, and MAGMA's batched band solver (Abdelfattah et al. (2023)), for matrix sizes ranging from (32×32) to (1024×1024) for a total of 10^4 linear systems. Both GINKGO'S and MAGMA's GPU batched band solvers outperform the batched CPU band solver. The MAGMA implementation is dependent on the amount of shared memory available on the GPU and explicitly performs the factorization and the solve after copying the band arrays to the GPU shared memory. Therefore, we observe the jump in timings at a matrix size of (128×128) where the shared memory capacity is exceeded and only part of the work vectors can be stored in shared memory. We do not observe this behavior for GINKGO's implementation because it caches only the pivot vector in shared memory but keeps the band array in global memory. This strategy allows for increasing the occupancy of the GPU, allowing the CUDA runtime to utilize the caches effciently, based on the thread block sizes.

Due to the limited amount of parallelism available in the band LU factorization, large thread block sizes are inefficient as they leave most warps unutilized. For the non-blocked factorization, GINKGO's solver uses only a single workgroup for each system in the batch. This allows leveraging the coarse-grained parallelism of the GPU, enabling better scaling when solving batches with large cardinality. This comes at the cost of lower performance for large problems as seen in Figure 5a, where more data needs to be fetched from global memory.

Increasing the width of the bands of the matrices, as shown in Figure 5b, with $(\beta_L,\ \beta_U)$ = (15, 5), we see a similar behavior: MAGMA's batched band solvers perform the best, particularly for matrix sizes larger than (256×256). There, in addition to increased shared memory usage, it is beneficial to increase the amount of fine-grained parallelism available.

Figure 6 shows the solver runtime for GINKGO'S batched band solver in comparison to Intel's MKL LAPACK band solver parallelized with OpenMP. We see that the GPU based solvers are more efficient for small batch sizes, and hence have a higher relative speedup. The variance in the plot denotes the different linear system sizes (number of rows in the system matrix), with the solid line showing the median runtime over the different matrix sizes, ranging from (64×64) to (1024×1024).

4.3 Wide band matrix optimizations

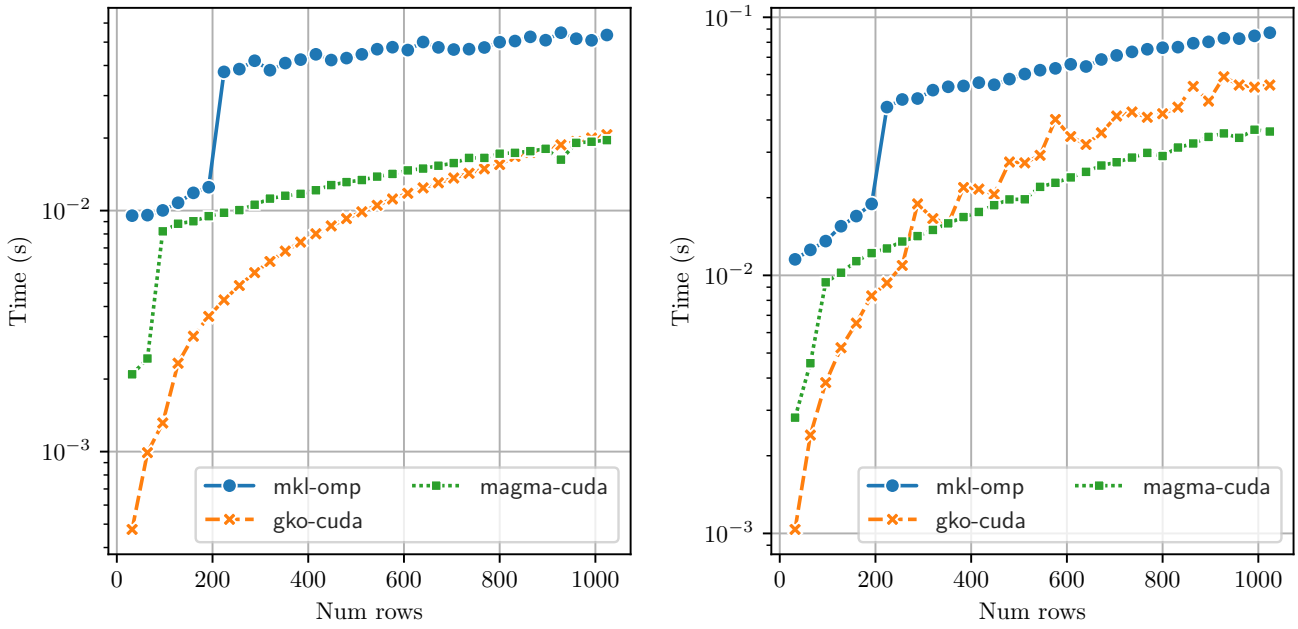
In Figure 7, we run the batched band solvers for matrices with a large bandwidth, $(\beta_L,\ \beta_U)$ = (32, 32). These wide bands can occur in applications such as the XGC plasma simulations (Ku et al. (2009)). As noted before, using a non-blocked version of the factorization does not allow for harnessing the parallelism available, and we see in Figure 7 that the non-blocked version on the GPU achieves lower performance than CPU-based MKL solver. With a blocked version, GINKGO's GPU solver outperforms both the MKL and MAGMA solver.

4.4 Tri-diagonal matrix optimizations

A pathological case for the band solvers is a tri-diagonal matrix. With only one sub- and super- diagonal, it does not offer enough parallelism to be solved with a general band solver algorithm like in LAPACK. We therefore implement a specialized tri-diagonal solver as explained in Section 3, based on the parallel divide-and-conquer approach (Wang and Mou (1991)). Figure 8 (a) shows the runtimes of GINKGO'S batched tri-diagonal solver compared with both cuSPARSE's batched tri-diagonal solver, and with GINKGO's generic band solver, setting (β_L, β_U) = (1, 1).

GINKGO automatically selects the best parameters (tile size, t and number of merge steps, ρ) for the tri-diagonal solver based on the number of rows of the matrix to maximize the available parallelism on the GPU. We see that there is a significant benefit in using a tailored tri-diagonal solver rather than using the band solver for tri-diagonal matrices, with the tri-diagonal solver outperforming the GPU-based band solver by almost $10\times$. The merge-tiling-based approach outperforms the cuSPARSE batched tri-diagonal solver by $1.7\times$ on average.

Figure 8 (b) shows the speedup obtained by GINKGO'S batched tri-diagonal solver over cuSPARSE for different batch sizes. We note that for the cuSPARSE solver, we



(a) β_L =2, β_U =3 **(b)** β_L =15, β_U =5

Figure 5. Ginkgo banded solver on A100 v/s MKL+OpenMP on Intel Icelake (76), 10^4 batch items.

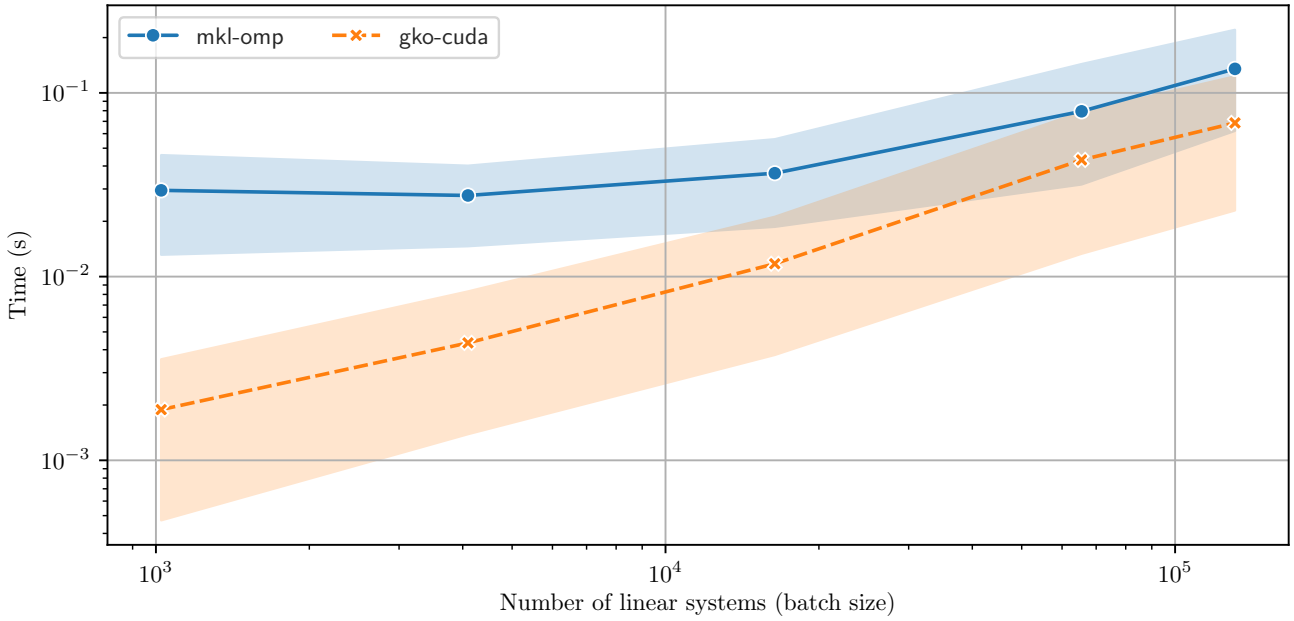


Figure 6. Runtime comparison of GINKGO band solver and OMP+LAPACK solver for different batch sizes, β_L = 2, β_U = 3.

time only the solve phase and do not consider the analysis phase. We observe that the merge-based tiling approach is more efficient for large batch sizes due to the better latency hiding of the warp scheduler for these larger batch sizes. Additionally, for small problems, GINKGO performs significantly better, for example with an average speedup of over $3\times$ for a matrix size of (8×8). For larger problems, GINKGO still outperforms the cuSPARSE solver with an average speedup of $1.5\times$. We note that the cuSPARSE solver exceeds the memory capacity of the NVIDIA A100 GPU, and hence some of the bars are missing.

4.5 Effects of pivoting

To ensure the robustness of the LU factorization, pivoting is usually necessary. For almost all cases, partial pivoting (row interchanges) is sufficient to ensure the stability of the factorization (Duff et al. (2017)). The state-of-the-art banded solvers such as MKL and MAGMA also implement only partial pivoting.

The dataset consists of matrices with all elements sampled from a normal distribution with variance of 0.1. All band solver implementations are equipped with partial pivoting. In Figure 9 (a), we consider four experiment runs of the banded solver and measure the number of row interchanges in each

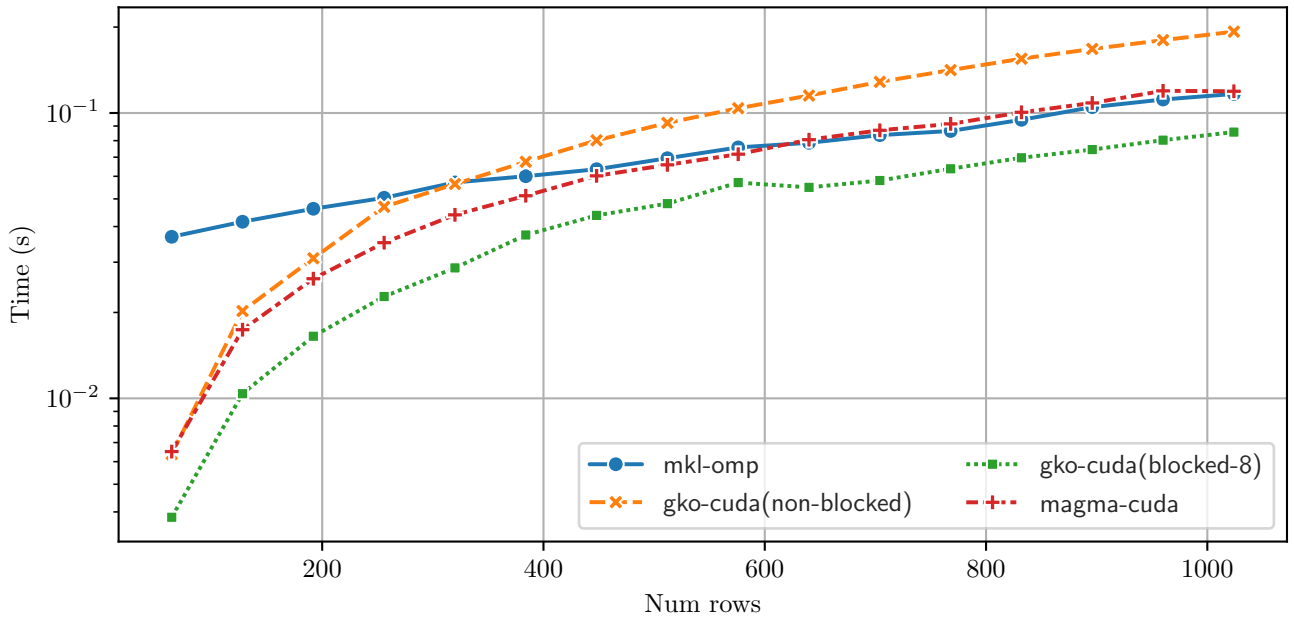


Figure 7. Ginkgo's panel based band solver, β_L =32, β_U =32, 5000 systems, block size=8×8,

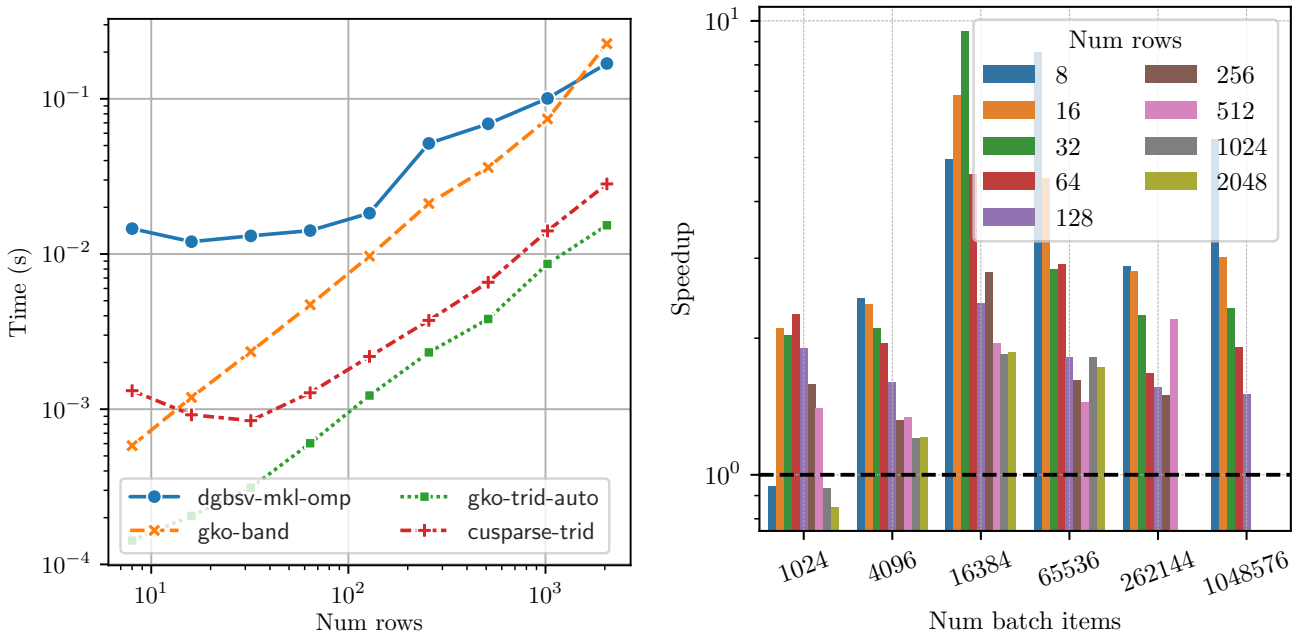


Figure 8. Performance of GINKGO'S tri-diagonal solver on A100 (a) v/s MKL+OpenMP on Intel Icelake (76) and cuSPARSE, 65536 batch items and (b) v/s cuSPARSE for different batch sizes.

of the batch items for two matrix sizes. We see that with a random matrix generation approach with approximately equal elements sampled from a normal distribution, all batch items need to perform partial pivoting and the number of row interchanges across the batch items ranges from $(n_{rows} - \frac{\sqrt{n_{rows}}}{2})$ to n_{rows}, where n_{rows} denotes the number of rows in each matrix of the batch.

Figure 9 (b) shows the runtime of the solvers with and without partial pivoting. To ensure that the band solvers do not apply row exchanges, we explicitly make the matrices strictly diagonally dominant by increasing the weight of the diagonal to be such that $|a_{jj}| > \sum_i^{n_{rows}} |a_{ij}|$. For the CPU-based MKL solver, the runtime overhead of pivoting is small – which is expected, given the significantly larger and coherent caches (L2 and L3). The GPU solvers from MAGMA and GINKGO both execute faster when no pivoting is necessary. Conversely, if pivoting is necessary, the communication across warps and induced synchronization introduces significant performance penalties. With partial pivoting, we also incur more cache misses, leading to more fetches from the main memory resulting in reduced performance for both MAGMA and GINKGO solvers. Figure 10 shows the speedup of the solvers without pivoting over the ones that need pivoting for different matrix sizes. As expected, the benefits of omitting pivoting grow with the matrix size.

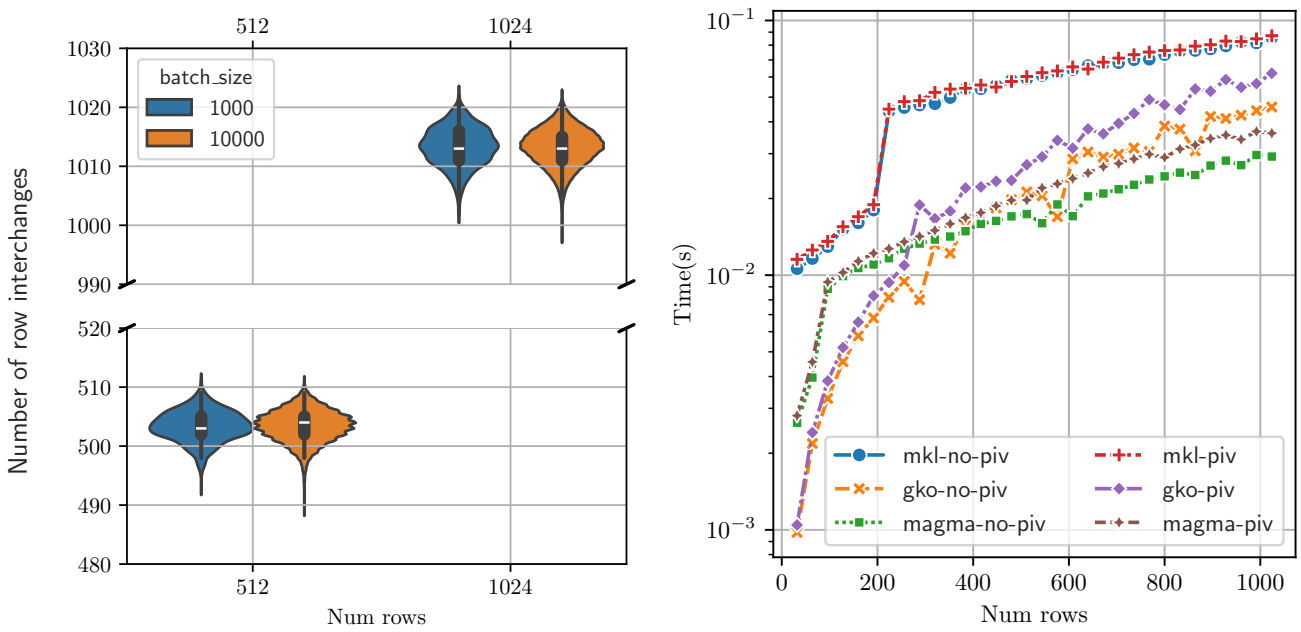


Figure 9. Effect of pivoting on performance (a) variation in number of row interchanges in a batch, (b) Comparing performance of the band solvers with and without pivoting, β_L = 15, β_U = 5, 10^4 batch items.

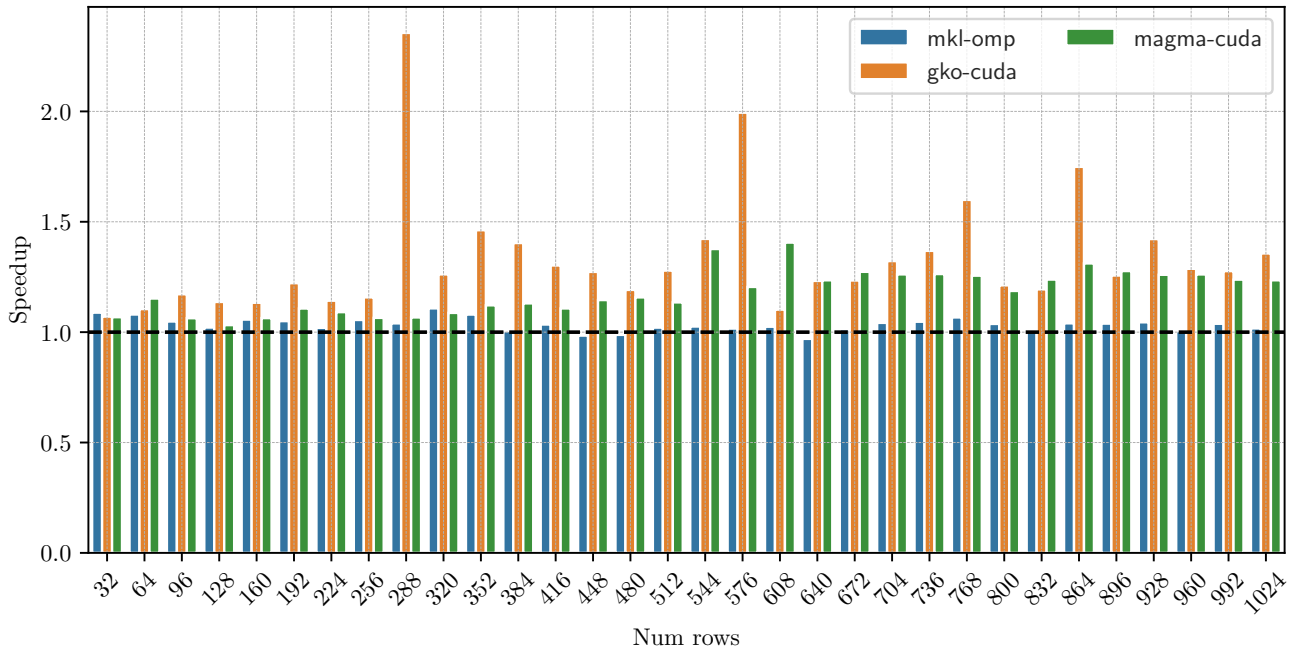


Figure 10. Band solver speedup without pivoting, β_L = 15, β_U = 5, 10^4 batch items.

We note that the code for the two experiments is identical. The only difference is in the input data: the matrices are generated such that pivoting is not necessary. Therefore, the end-user automatically gets improved performance when the matrix does not require any pivoting.

4.6 *Linear systems from a plasma physics application*

Finally, we study the performance of GINKGO's batched band solvers for matrices from the XGC plasma physics application. XGC (Hager et al. (2016)) is a 5D gyrokinetic particle-in-cell (PIC) application code that numerically simulates fusion plasma for realistic geometries, and has been optimized for boundary plasma. A nonlinear collision operator for the Coulomb collisions is required to accurately model edge plasma. Using a Fokker-Planck-Landau operator in the 2D guiding-center velocity space for multiple particle species, XGC employs a backward Euler time integrator to evolve the distribution functions for the multiple species in time.

Each time-step therefore requires multiple non-linear steps and each non-linear step requires multiple independent linear solves, giving us a batched linear system for each species. XGC currently uses the band solver from vendor-provided

LAPACK, mapping one linear system to one CPU core using Kokkos (Carter Edwards et al. (2014)).

The size and characteristics of the band matrix are dependent on the velocity grid used. For example, using a velocity grid of size (31×32) gives a matrix of size (992×992). Additionally, if multiple species are involved, we need to solve a linear system for each of these species. For our purposes, we use a velocity grid of size (31×32) and two species: ions and electrons. The characteristics of our test matrices (obtained from the XGC application) are shown in Figure 11. We note that the matrix is derived from a 9-point stencil, therefore the upper and lower bandwidths are given by $(\beta_L, \beta_U) = (33, 33)$, dictated by the number of points in the x-direction of the velocity grid. The matrix therefore has a moderately large bandwidth. The number of batch items varies with the number of species, and can depend on the available GPU memory. In a typical run of XGC (on A100-40GB), it can vary from 256 to 1024. It is expected that for newer GPUs with larger memory capacities, this can be significantly larger.

In Figure 12, we compare the performance of our batched band solvers with that of the state-of-the-art CPU-based band solver from MKL on Intel CPUs and with the GPU batched band solvers from MAGMA. Given the wide bandwidth of this matrix, we observe that the non-blocked strategy is inefficient and each subwarp gets assigned too much work, and hence it is slower than both the MKL version and MAGMA. Using a blocked version, with a panel size $\phi = 2$, we can improve our performance significantly, outperforming MKL solver on average by $1.5\times$. The matrix has lower and upper bandwidths of 33, and we observe that a panel size of 8 gives the best performance, providing a good balance between the amount of work assigned to each subwarp while providing enough parallelism to have enough concurrent warps resident on the compute unit. On average, with a panel size of 8, we get a speedup of $2.5\times$ over the MKL solver. We did not observe any performance benefits beyond a panel size of 8.

5 Conclusion

In this paper, we have presented GINKGO'S batched band solvers which are useful in many computational physics applications. Using a matrix storage scheme initially proposed in LAPACK, we designed two algorithms for the solution of batched band solvers. The first band solver is efficient for small bandwidth matrices. The second algorithm, a blocked version, uses a panel-based factorization to enable an efficient factorization for matrices with a wide band. We showcased the performance of our batched band solvers for randomly generated band matrices for different bandwidths, and demonstrated competitiveness with the state-of-the-art batched band solvers for CPU and GPU architectures. We included an analysis of the performance impact of partial pivoting, and showed that the GPU solvers can benefit from matrices that do not require partial pivoting.

The third algorithm we presented employed a divide-and-conquer approach for the solution of batched tri-diagonal systems. This strategy proved to be significantly faster than implementations available in the software stack provided by hardware vendors.

Finally, we investigated the performance of the developed batched band solvers when applied to linear systems originating from the XGC plasma physics application. We demonstrated attractive runtime savings over the solvers available in MKL and the GPU library MAGMA.

Acknowledgements

This research was 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. This work was performed on the HoreKa supercomputer funded by the Ministry of Science, Research and the Arts Baden-Württemberg and by the Federal Ministry of Education and Research, Germany.

References

Abdelfattah A, Costa T, Dongarra J, Gates M, Haidar A, Hammarling S, Higham NJ, Kurzak J, Luszczek P, Tomov S and Zounon M (2021) A Set of Batched Basic Linear Algebra Subprograms and LAPACK Routines. *ACM Transactions on Mathematical Software* 47(3): 21:1–21:23. DOI:10/gnn4cz.

Abdelfattah A, Tomov S, Luszczek P, Anzt H and Dongarra J (2023) GPU-based LU Factorization and Solve on Batches of Matrices with Band Structure. In: *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, SC-W '23. New York, NY, USA: Association for Computing Machinery. ISBN 9798400707858, pp. 1670–1679. DOI: 10.1145/3624062.3624247.

Aggarwal I, Kashi A, Nayak P, Balos CJ, Woodward CS and Anzt H (2021) Batched Sparse Iterative Solvers for Computational Chemistry Simulations on GPUs. In: *2021 12th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*. pp. 35–43. DOI:10/gn3xcg.

Anderson E (ed.) (1999) *LAPACK Users' Guide*. Software, Environments, Tools, 3rd ed edition. Philadelphia: Society for Industrial and Applied Mathematics. ISBN 978-0-89871-447-0.

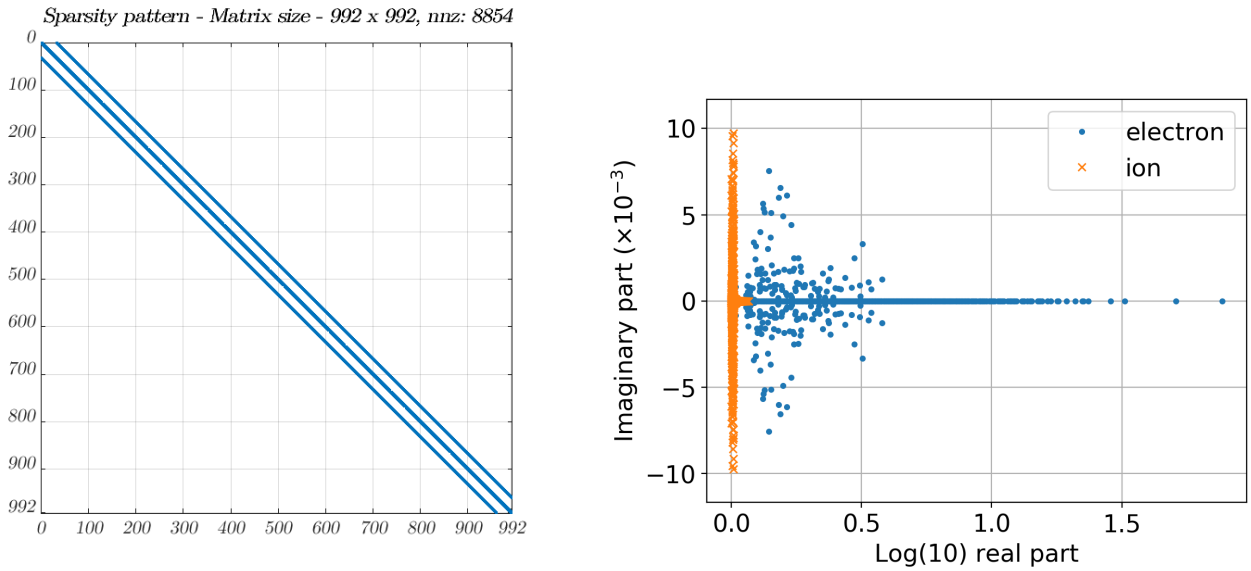
Anzt H, Cojean T, Flegar G, Göbel F, Grützmacher T, Nayak P, Ribizel T, Tsai YM and Quintana-Ortí ES (2022) Ginkgo: A Modern Linear Operator Algebra Framework for High Performance Computing. *ACM Transactions on Mathematical Software* 48(1): 2:1–2:33. DOI:10/gphfsq.

Blackford S and Dongarra J (1991) LAPACK Working Note 41 Installation Guide for LAPACK.

Carroll E, Gloster A, Bustamante MD and Náraigh LÓ (2021) A Batched GPU Methodology for Numerical Solutions of Partial Differential Equations. *arXiv:2107.05395 [physics]* .

Carter Edwards H, Trott CR and Sunderland D (2014) Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing* DOI:10.1016/j.jpdc.2014.07.003.

Dongarra J, Duff I, Gates M, Haidar A, Hammarling S, Higham N, Hogg J, Valero-Lara P, Relton SD, Tomov S and Zounon M (2016) A Proposed API for Batched Basic Linear Algebra Subprograms. Technical Report 2016.25, The University of Manchester.



(a) Sparsity pattern

(b) Eigenvalues

Figure 11. XGC example matrix characteristics for the two species, ion and electron (Kashi et al. (2022))

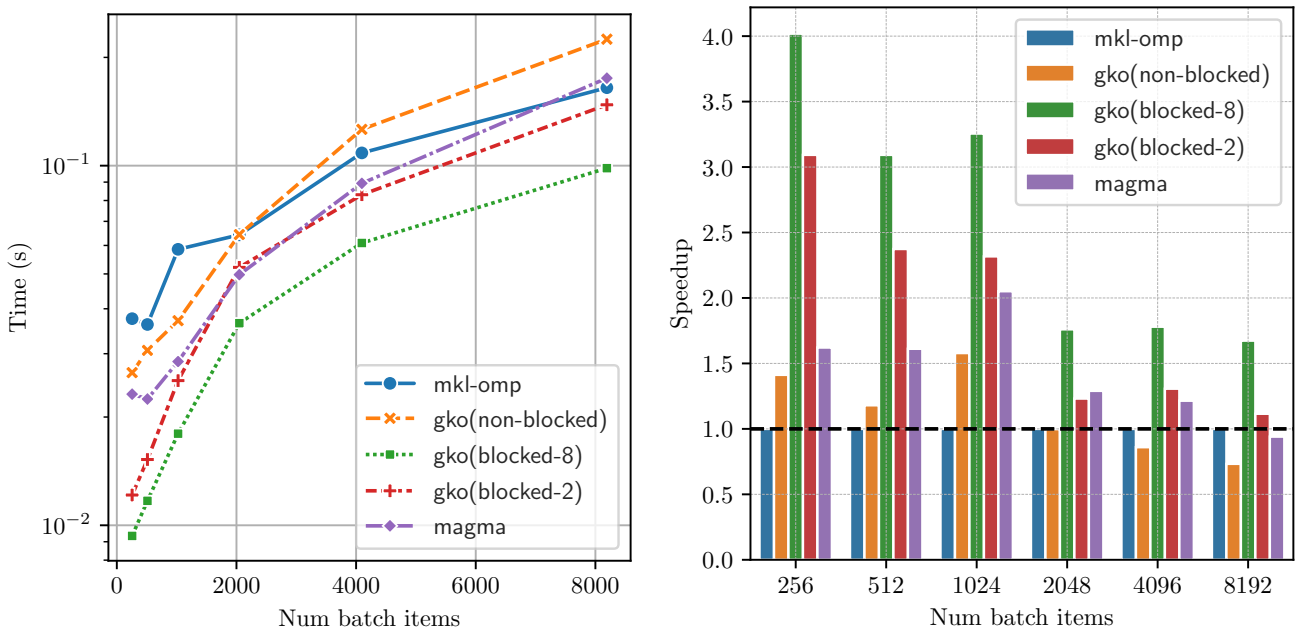


Figure 12. Performance of the batched band solvers with matrices and right hand sides from the plasma physics application, XGC.

Dongarra J, Gates M, Haidar A, Kurzak J, Luszczek P, Tomov S and Yamazaki I (2014) Accelerating Numerical Dense Linear Algebra Calculations with GPUs. In: Kindratenko V (ed.) *Numerical Computations with GPUs*. Cham: Springer International Publishing. ISBN 978-3-319-06548-9, pp. 3–28. DOI:10.1007/978-3-319-06548-9_1.

Dongarra J, Hammarling S, Higham NJ, Relton SD, Valero-Lara P and Zounon M (2017) The Design and Performance of Batched BLAS on Modern High-Performance Computing Systems. *Procedia Computer Science* 108: 495–504. DOI:10.1016/j.procs.2017.05.138.

Duff I, Erisman AM and Reid JK (2017) *Direct Methods for Sparse Matrices*. Second edition. Oxford University Press. ISBN 978-0-19-850838-0.

Gloster A, Ó Náraigh L and Pang KE (2019) cuPentBatch—A batched pentadiagonal solver for NVIDIA GPUs. *Computer Physics Communications* 241: 113–121. DOI:10.1016/j.cpc.2019.03.016.

Hager R, Yoon ES, Ku S, D'Azevedo EF, Worley PH and Chang CS (2016) A fully non-linear multi-species Fokker–Planck–Landau collision operator for simulation of fusion plasma. *Journal of Computational Physics* 315: 644–660. DOI:10.1016/j.jcp.2016.03.064.

Hindmarsh AC (2002) SUNDIALS: Suite of Nonlinear/Differential/Algebraic Equation Solvers. Technical Report UCRL-JC-149711, Lawrence Livermore National Lab. (LLNL), Livermore, CA (United States). DOI:10.1145/1089014.1089020.

Intel (2021) Intel Xeon 8368 specification. Technical report, Intel Corporation.

Intel (2023) oneAPI Math Kernel Library. Intel Corporation.

Kashi A, Nayak P, Kulkarni D, Scheinberg A, Lin P and Anzt H (2022) Batched sparse iterative solvers on GPU for the collision operator for fusion plasma simulations. In: *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. pp. 157–167. DOI:10.1109/IPDPS53621.2022.00024.

Kashi A, Nayak P, Kulkarni D, Scheinberg A, Lin P and Anzt H (2023) Integrating batched sparse iterative solvers for the collision operator in fusion plasma simulations on GPUs. *Journal of Parallel and Distributed Computing* DOI:10.1016/j.jpdc.2023.03.012.

Klein C and Strzodka R (2021) Tridiagonal GPU Solver with Scaled Partial Pivoting at Maximum Bandwidth. In: *Proceedings of the 50th International Conference on Parallel Processing*, ICPP '21. New York, NY, USA: Association for Computing Machinery. ISBN 978-1-4503-9068-2, pp. 1–10. DOI:10.1145/3472456.3472484.

Ku S, Chang CS and Diamond PH (2009) Full-f gyrokinetic particle simulation of centrally heated global ITG turbulence from magnetic axis to edge pedestal top in a realistic tokamak geometry. *Nuclear Fusion* 49(11): 115021. DOI:10.1088/0029-5515/49/11/115021.

Li XS and Demmel JW (2003) SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Transactions on Mathematical Software* 29(2): 110–140. DOI:10.1145/779359.779361.

Nayak P, Aggarwal I and Anzt H (2024) Reproducibility artifact for paper: Efficient batched band solvers on GPUs. Zenodo. DOI:10.5281/ZENODO.10871244.

Nayak PV (2023) *Synchronization-Free Algorithms for Exascale and beyond : A Study of Asynchronous and Batched Iterative Methods*. PhD Thesis, Karlsruhe Institute of Technology, Karlsruhe. DOI:10.5445/IR/1000165437.

NVIDIA (2020) NVIDIA A100 Tensor core GPU Architecture. Technical report, NVIDIA Corporation.

Pérez Diéguez A, Amor López M and Doallo Biempica R (2018) Solving Multiple Tridiagonal Systems on a Multi-GPU Platform. In: *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. pp. 759–763. DOI:10.1109/PDP2018.2018.00123.

Tsai YM, Cojean T, Ribizel T and Anzt H (2021) Preparing Ginkgo for AMD GPUs – A Testimonial on Porting CUDA Code to HIP. In: Balis B, B Heras D, Antonelli L, Bracciali A, Gruber T, Hyun-Wook J, Kuhn M, Scott SL, Unat D and Wyrzykowski R (eds.) *Euro-Par 2020: Parallel Processing Workshops*, Lecture Notes in Computer Science. Cham: Springer International Publishing. ISBN 978-3-030-71593-9, pp. 109–121. DOI:10.1007/978-3-030-71593-9_9.

Valero-Lara P, Martínez-Perez I, Peña AJ, Martorell X, Sirvent R and Labarta J (2017) cuHinesBatch: Solving Multiple Hines systems on GPUs Human Brain Project. *Procedia Computer Science* 108: 566–575. DOI:10.1016/j.procs.2017.05.145.

Valero-Lara P, Martínez-Pérez I, Sirvent R, Martorell X and Peña AJ (2018) cuThomasBatch and cuThomasVBatch, CUDA Routines to compute batch of tridiagonal systems on NVIDIA GPUs. *Concurrency and Computation: Practice and Experience* 30(24): e4909. DOI:10.1002/cpe.4909.

Wang X and Mou Z (1991) A divide-and-conquer method of solving tridiagonal systems on hypercube massively parallel computers. In: *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*. pp. 810–817. DOI:10.1109/SPDP.1991.218237.