Lawrence Berkeley National Laboratory

LBL Publications

Title

A graphics processing unit accelerated sparse direct solver and preconditioner with block low rank compression

Permalink

https://escholarship.org/uc/item/7tn9n67r

Authors

Claus, Lisa
Ghysels, Pieter
Boukaram, Wajih Halim
et al.

Publication Date

2024

DOI

10.1177/10943420241288567

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

International Journal on High Performance Computing Applications
XX(X):1–11
©The Author(s) 2016
Reprints and permission:
sagepub.co.uk/journalsPermissions.nav
DOI: 10.1177/ToBeAssigned
www.sagepub.com/
SAGE

A GPU accelerated sparse direct solver and preconditioner with block low rank compression

Lisa Claus[1], Pieter Ghysels[2], Wajih Halim Boukaram[2] and Xiaoye Sherry Li[2]

Abstract

We present the GPU implementation efforts and challenges of the sparse solver package STRUMPACK. The code is made publicly available on github with a permissive BSD license. STRUMPACK implements an approximate multifrontal solver, a sparse LU factorization which makes use of compression methods to accelerate time to solution and reduce memory usage. Multiple compression schemes based on rank-structured and hierarchical matrix approximations are supported, including hierarchically semi-separable, hierarchically off-diagonal butterfly, and block low rank.
In this paper, we present the GPU implementation of the block low rank (BLR) compression method within a multifrontal solver. Our GPU implementation relies on highly optimized vendor libraries such as cuBLAS and cuSOLVER for NVIDIA GPUs, rocBLAS and rocSOLVER for AMD GPUs and the Intel oneAPI Math Kernel Library (oneMKL) for Intel GPUs. Additionally, we rely on external open source libraries such as SLATE (Software for Linear Algebra Targeting Exascale), MAGMA (Matrix Algebra on GPU and Multi-core Architectures), and KBLAS (KAUST BLAS). SLATE is used as a GPU-capable ScaLAPACK replacement. From MAGMA we use variable sized batched dense linear algebra operations such as GEMM, TRSM and LU with partial pivoting. KBLAS provides efficient (batched) low rank matrix compression for NVIDIA GPUs using an adaptive randomized sampling scheme.
The resulting sparse solver and preconditioner runs on NVIDIA, AMD and Intel GPUs. Interfaces are available from PETSc, Trilinos and MFEM, or the solver can be used directly in user code. We report results for a range of benchmark applications, using the Perlmutter system from NERSC, Frontier from ORNL, and Aurora from ALCF. For a high frequency wave equation on a regular mesh, using 32 Perlmutter compute nodes, the factorization phase of the exact GPU solver is about $6.5\times$ faster compared to the CPU-only solver. The BLR-enabled GPU solver is about $13.8\times$ faster than the CPU exact solver. For a collection of SuiteSparse matrices, the STRUMPACK exact factorization on a single GPU is on average $1.9\times$ faster than NVIDIA's cuDSS solver.

Keywords

Linear Solvers, Preconditioning, Low-Rank Approximation, Sparse Direct Solver, Multifrontal Method

Introduction

We present a sparse approximate LU factorization solver for multi-GPU systems, as implemented in the STRUMPACK solver library. The sparse solver is a multifrontal LU method (Duff et al. 2017), where the triangular factors are compressed using a block low rank (BLR) scheme applied to the largest dense blocks (frontal matrices). Sparse direct solvers are popular for a variety of application areas because of their robustness and lack of tuning parameters. However, the main bottleneck is their memory usage and asymptotic scaling with the problem size. This poor scaling is due to the problem of fill-in, the triangular factors typically have many more nonzero entries compared to the original sparse matrix. Our focus is on the efficient implementation, on modern HPC systems, of the multifrontal solver with BLR compression.

Based on the STRUMPACK multifrontal sparse solver, we have previously shown nearly linear complexity for the factorization of sparse systems from a range of partial differential equations (PDEs). This earlier work relied on hierarchically semi-separable (HSS) compression (Ghysels et al. 2017), hierarchically off-diagonal butterfly (HODBF) compression (Liu et al. 2021) or a hybrid of HODBF and block low rank (BLR) compression (Claus et al. 2023) within a sparse multifrontal solver. However, the reliance of these methods on hierarchical matrix partitioning makes them challenging to implement efficiently on GPU architectures since the hierarchical representations rely on irregular tree data structures. In order to keep the complexity nearly linear these methods avoid explicitly constructing the largest frontal matrices (dense sub-blocks in the sparse factors). This then requires (partially) matrix-free matrix construction techniques which rely on random projection and matrix sampling or matrix element extraction. We have found that especially the matrix element extraction forms a bottleneck for the parallel implementation (Ghysels et al. 2016). Moreover, the HSS and hierarchically off-diagonal low rank (HODLR (Aminfar and Darve 2016)) schemes both use a weak admissibility structure (easier to implement in parallel than strong admissible $\mathcal{H}$), but leads to larger off-diagonal ranks. The work presented here uses

[1]National Energy Research Scientific Computing Center, Lawrence Berkeley National Laboratory, USA
[2]Applied Mathematics and Computational Research Division, Lawrence Berkeley National Laboratory, USA
Email: lclaus@lbl.gov

BLR compression, a non-hierarchical scheme with strong admissibility that is relatively straightforward to implement. Although not asymptotically optimal, our sparse solver with BLR is applicable to a wider range of problems, and gives significant speedups for medium to large scale problems over the sparse exact solver and over the sparse approximate solvers with HSS or HODBF compression.

The STRUMPACK solver can be used directly in user code or through a higher level math library such as PETSc (Portable, Extensible Toolkit for Scientific Computation) (Balay et al. 2023), Trilinos (Heroux et al. 2005), or MFEM (Anderson et al. 2021). STRUMPACK is also part of the xSDK (Extreme-scale Scientific Software Development Kit), which tests the integration between these various ECP (Exascale Computing Project) sponsored mathematical libraries, and releases them together using a Spack (https://spack.io/) based installation script. We highly recommend use through a library like PETSc, which provides robust and scalable primitives for sparse matrix assembly, sparse iterative solvers, logging and profiling tools, etc. Moreover, PETSc allows the user to compose multiple solver components into a single more advanced solver strategy, and it allows to easily swap out solver components. For instance, through PETSc one can use STRUMPACK as a direct solver, as a preconditioner for a Krylov method, as a subdomain solver in a domain decomposition method, as a coarse grid solver for algebraic multigrid, as a preconditioner for a LOBPCG, etc. Since our approximate solver is derived from a direct solver and uses rank structured approximation, the accuracy and preconditioner strength can be controlled well through the low rank compression tolerance. This fact can be exploited to minimize overall time to solution, spending more or less time in preconditioner setup versus solve, for instance when the solver is used repeatedly – in timestepping, in a non-linear solver, within a preconditioner – or with a large number of right-hand sides. The compression schemes introduce a number of performance tuning parameters which are not present in standard sparse direct solvers, such as the compression tolerance, the BLR block size, and the minimum front size for compression. Some of these have good default values – which might be set differently for CPU or GPU execution – for most practical problems, while for others application specific tuning might be required. The algorithm description presented here provides the user with background information to identify the important tuning parameters.

There exists a number of sparse direct solvers, including SuperLU_DIST (Li and Demmel 2003), SuiteSparse (UMFPACK (Davis 2004) and Cholmod (Chen et al. 2008)), PaStiX (Hénon et al. 2002), STRUMPACK (Ghysels et al. 2016; Ghysels and Synk 2022), MUMPS (Amestoy et al. 2001), WSMP (Gupta 2000), and cuSOLVER/cuDSS*. Out of these, SuperLU_DIST, Cholmod (symmetric positive definite), PaStiX, cuSOLVER and STRUMPACK have GPU support. Most of these solvers are supernodal, meaning that they group factor columns with the same sparsity pattern in order to exploit more efficient dense linear algebra kernels. SuperLU_DIST, PaStiX and Cholmod are supernodal; MUMPS, WSMP, UMFPACK and STRUMPACK are supernodal and multifrontal. MUMPS (Amestoy et al. 2019) and PaStiX (Hénon et al. 2002; Nies and Hoelzl 2019) have support for block low rank (BLR) compression. (Aminfar and Darve 2016) shows a multifrontal solver with hierarchically off-diagonal low rank approximation.

System	Perlmutter	Frontier	Aurora
GPU	NVIDIA A100	AMD MI250X	Intel X^e
API	CUDA	HIP/ROCm	SYCL
CPU	AMD EPYC	AMD EPYC	Intel Xeon
TOP500	#12	#1	#2
Nodes	1,792 +3,072 (CPU)	9,472	10,624
RPeak	113.00	1,679.82	1,059.33
Location	NERSC / LBNL	OLCF / ORNL	ALCF / ANL
Vendor	HPE	HPE	Intel

Table 1. The three systems operated by the DOE leadership compute facilities, with three different GPU vendors and programming models. Top500 rankings are from the November 2023 list. RPeak is the theoretical peak performance in PFlop/s.

Through its leadership compute facilities (LCF), the Department of Energy (DOE) Advanced Scientific Computing Research (ASCR) program operates some of the most powerful computers in the world, see Table 1. DOE has pledged to work with multiple hardware vendors to prevent potential vendor lock-in. As a result, the three DOE compute facilities run on GPUs from three different vendors, each with their own programming model. With our solver, we target each of these machines: Perlmutter at NERSC using NVIDIA GPUs running CUDA, Frontier at OLCF with AMD GPUs running HIP/ROCm and Aurora at ALCF with Intel GPUs running SYCL.

Our main contribution is the development of a robust, efficient and scalable, fully algebraic preconditioner with support for GPUs from multiple vendors. The solver and preconditioner are available from the STRUMPACK sparse solver library, written in modern C++, and released with a permissive BSD license.

The rest of this article is structured as follows. We first describe the multifrontal solver and its GPU implementation in STRUMPACK. The next section describes the BLR format, followed by a discussion on how to incorporate the BLR scheme in the sparse multifrontal factorization. Then we present performance results.

Multifrontal LU Factorization

We consider the solution of a sparse linear system $Ax = b$ with $A \in \mathbb{C}^{N\times N}$. To achieve this, we compute a decomposition of A as $P(D_r A D_c Q)P^T = LU$, where P and Q are permutation matrices, D_r and D_c are diagonal scaling matrices, and L and U are sparse lower and upper triangular factors. The permutation P aims to reduce the number of nonzeros in the triangular factors, and we provide a number of options, based on nested dissection (including METIS (Karypis and Kumar 1998), ParMETIS (Schloegel

*https://docs.nvidia.com/cuda/cusolver/, https://developer.nvidia.com/cudss

et al. 1997), Scotch (Pellegrini and Roman 1996) and PT-Scotch (Chevalier and Pellegrini 2008)), or based on minimum degree or minimum fill/deficiency. The optional permutation Q, and the scaling factors D_r and D_c are computed using the MC64 (Duff and Koster 1999) matching code. The goal of Q is to maximize the product of the diagonal elements, which are then scaled by D_r and D_c such that all diagonal entries are 1 and all off-diagonal entries are less than 1 in absolute value. Since MC64 is a sequential code we need to gather the sparse input matrix on a single MPI rank in order to call it, which can be a scaling bottleneck. However, the MC64 reordering is optional, and can safely be disabled for some problems, such as diagonally dominant matrices. Alternatively, we interface to a parallel code, Combinatorial BLAS (Azad et al. 2021), to find the permutation Q, which is based on the heavy-weight perfect matching algorithm from (Azad et al. 2020).

When we are not using MC64, the row and column scalings are computed using an algorithm based on LAPACK's `xgeequ`, adapted for sparse input as in SuperLU_DIST.

One can typically distinguish three separate phases for the direct solution of sparse linear systems:

1. Reordering and symbolic analysis
2. Numerical factorization
3. Solve using forward and backward substitution

In phase 1, the permutation and scaling vectors are computed, the sparsity pattern is analyzed, and data-structures are initialized to guide the numerical factorization phase. After computation of the sparse factorization (computationally the most expensive phase), a linear system can be solved efficiently by applying the permutations and scalings, and performing the sparse triangular solves with the L and U factors.

To illustrate the numerical factorization phase, we consider a nested dissection permutation P with only two levels, and a single vertex separator:

$$PAP^T = \begin{bmatrix} A_1 & & X_{1S} \\ & A_2 & X_{2S} \\ X_{S1} & X_{S2} & S \end{bmatrix}. \quad (1)$$

The lower-right sub-block S corresponds to a separator in the graph of A, effectively splitting the problem in two unconnected components, represented by A_1 and A_2. We can now construct three frontal matrices

$$F_1 = \begin{bmatrix} A_1 & \hat{X}_{1S} \\ \tilde{X}_{S1} & \end{bmatrix}, \; F_2 = \begin{bmatrix} A_2 & \hat{X}_{2S} \\ \tilde{X}_{S2} & \end{bmatrix}, \; F_0 = S, \quad (2)$$

where $\hat{X}_{1S}/\tilde{X}_{S1}$ is the matrix consisting of only the columns/rows of X_{1S}/X_{S1} which contain nonzero elements. These fronts are put in a binary tree with F_0 as the root and F_1 and F_2 as the children. The numerical phase of the multifrontal LU factorization algorithm then traverses this binary tree from the leaves to the root. At each front $F_\tau = [F_{11}\ F_{12};\ F_{21}\ F_{22}]$ (except the root), the following steps are performed:

- $P_\tau L_\tau U_\tau \leftarrow LU(F_{11})$
- $F_{12} \leftarrow U_\tau^{-1} L_\tau^{-1} P_\tau^T F_{12}$
- $F_{22} \leftarrow F_{22} - F_{21} F_{12}$

At the root front, only the first of these steps needs to be performed. The first step computes a dense LU factorization with partial pivoting. Note that this restricts the pivoting to the diagonal blocks of the (permuted) sparse matrix. However, for most problems, especially when combined with the permutation Q from the MC64 matching, this is sufficient to ensure numerical stability, and it greatly simplifies the implementation, compared to for instance the more robust approach with dynamic delayed pivots as used in MUMPS (Amestoy et al. 2001). We also provide the option to add a small perturbation to the diagonal elements if they are below a certain threshold (set to $\sqrt{\epsilon_{\text{mach}}}\|A\|_1$). The permutation Q, combined with the diagonal perturbation, is referred to as static pivoting, as introduced by (Li and Demmel 1998). After these operations are applied to front F_τ, the Schur complement $F_{\tau;22}$ is added into the parent frontal matrix. However, since the parent front is typically larger than the Schur complement of its child, this requires a scatter operation defined by the sparsity pattern, referred to as the extend-add phase.

The described approach can be generalized by recursively applying the nested dissection heuristic to the subdomains A_1 and A_2, leading to a binary tree, referred to as the assembly tree, with $\mathcal{O}(\log N)$ levels. Going down the tree from the root to the leaves, subdomains and separators become smaller, leading to smaller frontal matrices, while the tree becomes wider. Note that all fronts in a given level can be handled concurrently. Likewise, non-overlapping subtrees can be handled concurrently. For a general permutation P (for instance from a minimum degree ordering), the assembly tree can be constructed from the elimination tree and a supernode detection algorithm.

The CPU multi-core implementation relies on OpenMP tasking to traverse the assembly tree and to exploit parallelism within the larger frontal matrices, for instance in BLAS, LAPACK and ScaLAPACK. For the distributed memory parallel code, the assembly tree is split in multiple subtrees, each of which is assigned to a single MPI rank, while the top $\log P$ levels (P the total number of MPI processes) of the tree are distributed using a 2D block cyclic layout and then processed using ScaLAPACK. The root front is distributed over a 2D process grid constructed from the user specified MPI communicator. The dimensions of the process grid are $P_r \times P_c$, with $P_c = \lfloor \sqrt{P} \rfloor$ and $P_r = \lfloor P/P_c \rfloor$, which leaves at most $\left\lfloor \sqrt{P} \right\rfloor - 1$ processes idle. For the next level in the assembly tree (children of the root), the MPI communicator is split in two distinct subcommunicators, sized proportionally to the estimated work (or memory) for each of the subtrees.

GPU Implementation

In contrast to the recursive tree traversal for the multi-core implementation, the GPU code traverses the tree level-by-level using batched algorithms for the dense linear algebra operations from the MAGMA library (e.g., for double precision):

- `magma_dgetrf_vbatched_max_nocheck_work` for the variable sized batched dense LU factorization

with partial pivoting for all F_{11} blocks on a given level. (The _max_nocheck_work version of this routine skips checking of the input, takes as input the maximum size of all matrices in the batch, and takes user allocated working memory).
- magmablas_dtrsm_vbatched_max_nocheck for triangular solves with all F_{12} on a level.
- magmablas_dgemm_vbatched_max_nocheck for the Schur complement update in F_{22}. Here, using the _max_nocheck version avoids synchronization in this routine, which allows this multiplication to run asynchronously and overlap with the copy of the factors from device to host memory.

These batched routines drastically reduce kernel launch overheads compared to, for instance, using vendor libraries like cuBLAS and cuSOLVER directly for each front separately, even when launched from multiple streams. Most of the work is performed in these dense linear algebra routines, and this allows the solver to achieve high floating point operation throughput. If STRUMPACK is configured without MAGMA support, it falls back to a combination of vendor BLAS/LAPACK libraries, such as cuBLAS/cuSOLVER or oneMKL, and a naive internal implementation of batched kernels for operations on fronts $\leq 32 \times 32$, see (Ghysels and Synk 2022).

The solver also relies on a small number of custom device kernels. These kernels include:

- The initial assembly of the fronts from elements of the sparse matrix.
- Extend-add operation, i.e., adding the Schur complement from one front into its parent front.
- Checking the diagonal elements of F_{11} and replacing small pivot elements.

These are implemented as batched kernels, separately for CUDA, HIP and SYCL. Duplicating the kernels for the different device backends is not ideal and a better long term strategy would be to use a portability layer such as Kokkos[†] or Raja[‡], which would introduce heavy external dependencies, or an industry standard like OpenMP device offloading. This would still not alleviate the burden of having to run regular integrated testing on the multiple vendor platforms.

The required amount of device memory is pre-computed. If sufficient device memory is available to store the entire factorization and the working memory on the device, the device memory is acquired with a single allocation for the factor memory and another allocation for the working memory. In this case, the triangular factors are kept on the device and the triangular solve phase can use the factors on the device. If the entire assembly tree does not fit in device memory, then the factorization is performed level-per-level, and frontal matrices are copied to host memory after a level is finished. If there is not enough device memory to store an entire level of the assembly tree, then the factorization is split in multiple traversals of subtrees that do fit in device memory. In this case, the computed factors also need to be transferred back from device to host memory. When the factors are stored in host memory, the triangular solve phase is performed on the CPU. Moving the factors to GPU first would be nearly as time consuming as performing the solve directly on the CPU. We also support multiple right-hand sides, but for now the number of right-hand sides does not affect the decision of where (on the CPU or the GPU) the solve is performed.

For the multi-GPU setting, we assume a single MPI rank per GPU. As for the CPU-only code, the parallel fronts are distributed with a 2D block cyclic layout, but they are now handled with SLATE (Gates et al. 2019), a GPU-capable ScaLAPACK alternative[§].

Vendor Agnostic GPU Implementation

The implementation of the multifrontal solver in STRUMPACK is GPU vendor agnostic, relying on a minimal interface to vendor specific APIs like CUDA for NVIDIA, HIP/ROCm for AMD and SYCL for Intel accelerators. The GPU backend is chosen when configuring STRUMPACK using CMake. The GPU interface layer includes the classes Stream, Event and Handle (in the strumpack::gpu namespace). The Stream class is a wrapper for cudaStream_t, hipStream_t or sycl::queue, for CUDA, HIP and SYCL respectively. The vendor specifics are internal and hidden from users of the library using the PIMPL (Pointer to Implementation) technique. The Event class is a wrapper for cudaEvent_t, hipEvent_t, or sycl::event, respectively. When targeting CUDA/NVIDIA, the Handle class is a wrapper to handles for cuBLAS (cublasHandle_t) and cuSOLVER (cusolverDnHandle_t), as well as MAGMA (magma_queue_t) and KBLAS (kblasHandle_t and kblasRandState_t), see the next section. For HIP, a hipblasHandle_t and a rocblas_handle are stored instead. These handles are associated with a Stream. For the SYCL implementation, the vendor BLAS/LAPACK functionality is available in the oneMKL library from Intel, which does not require any handles apart from the sycl::queue and the MAGMA queue.

The GPU interface further provides the routines device_malloc, device_free, host_malloc and host_free for device memory and pinned host memory allocation and deallocation, respectively. These routines are only used through RAII[¶] wrapper classes DeviceMemory and HostMemory.

Also provided are four copy routines, copy, copy_2D, copy_async, and copy_2D_async. The direction of the copy (e.g., cudaMemcpyKind) can be determined by the runtime (cudaMemcpyDefault). For convenience, these copy routines are overloaded for strumpack::DenseMatrix<T> objects.

The CUDA and HIP APIs are almost identical, apart from the cu/hip prefix. For SYCL however, there are some noteworthy differences. For instance, the copy and allocation routines are always associated with a queue,

[†] https://github.com/kokkos/kokkos

[‡] https://github.com/LLNL/RAJA

[§] Another alternative, cuSOLVERMp, is NVIDIA specific and is currently not available on NERSC's Perlmutter.

[¶] In RAII (Resource Allocation is Initialization) a resource is tied to an objects lifetime with the destructor releasing the resource.

while for CUDA and HIP, only the asynchronous copy routines are tied to a specific `Stream`. In our interface the allocation and (synchronous) copy routines do not take a `Stream`, but the (default) `sycl::queue` is stored as a globally accessible static instance. By default, a `cudaStream_t/hipStream_t` executes in issue order. A `sycl::queue` on the other hand defaults to out-of-order execution. We always construct `sycl::queues` using the `sycl::property::queue::in_order` flag to mimic the CUDA behavior. Moreover, a CUDA/HIP kernel launch or asynchronous copy cannot run concurrently with another operation launched on the default stream. In SYCL, there is no such notion of a default stream and all operations are asynchronous by default. We avoid running any asynchronous operations in a separate stream/queue concurrently with the default stream.

Block Low Rank Arithmetic

In the BLR format, a matrix is partitioned using a flat, non-hierarchical blocking. The flat partitioning is based on a clustering and permutation of the associated degrees of freedom. A BLR representation $\tilde{B}$ of a square dense matrix B with $p \times p$ blocks looks like

$$\tilde{B} = \begin{bmatrix} \tilde{B}_{11} & \dots & \tilde{B}_{1p} \\ \vdots & \ddots & \vdots \\ \tilde{B}_{p1} & \dots & \tilde{B}_{pp} \end{bmatrix} \approx B \,. \qquad (3)$$

The blocks of a BLR matrix can be represented/approximated as low rank depending on some admissibility condition as follows. Each sub-block $\tilde{B}_{\sigma\tau}$ in the matrix corresponds to the interaction between two clusters σ and τ. Diagonal blocks correspond to self-interactions, which we do not approximate. For the off-diagonal blocks, two admissibility conditions are considered. We can either compress every off-diagonal block, or only compress those blocks corresponding to well separated clusters, see the next section. A block $\tilde{B}_{\sigma\tau}$ of size $m_\sigma \times m_\tau$ and numerical rank $r_{\sigma\tau}$ is approximated by a low rank matrix $\tilde{B}_{\sigma\tau} = X_{\sigma\tau} Y_{\sigma\tau}^T \approx B_{\sigma\tau}$ at accuracy ε. $X_{\sigma\tau}$ is a $m_\sigma \times r_{\sigma\tau}$ matrix and $Y_{\sigma\tau}$ is a $m_\tau \times r_{\sigma\tau}$ matrix. However, if $r_{\sigma\tau}(m_\sigma + m_\tau) > m_\sigma m_\tau$, the tile $\tilde{B}_{\sigma\tau}$ is kept in its original dense non-compressed representation $\tilde{B}_{\sigma\tau} = B_{\sigma\tau}$.

The low rank approximation/compression can be performed in different ways. For the CPU code, truncated column pivoted QR is used. The truncation criteria checks both absolute and relative tolerance. On the GPU, an adaptive randomized approximation (ARA) algorithm is used for compression, see below.

The LU factorization can be implemented in different ways, STRUMPACK has different option implemented that we can make use of, for instance a left-looking versus a right-looking algorithm. In addition, different pivoting procedures can be performed. For BLR, the algorithm can behave very different depending on when the compression is performed. Performing the compression at a later stage in the algorithm typically leads to a more accurate but more expensive factorization, since more operations are still performed with the original dense sub-blocks (Higham and Mary 2022). For the GPU implementation, STRUMPACK uses a right-looking algorithm which compresses a block as soon as it has received all of its Schur complement updates, see Algorithm 1. For the CPU code, STRUMPACK implements both left-looking and right-looking variants, with the left-looking variant performing the compression earlier, and the updates are done into low rank blocks with a low rank update, accumulate and recompress (LUAR) operation, see (Claus et al. 2023; Mary 2017). We found the right-looking version to be better suited for GPU implementation.

Algorithm 1 Right-Looking Block Low Rank Compression and Factorization

1: $\tilde{B}_{\sigma\tau} \leftarrow B_{\sigma\tau}, \forall\, \sigma, \tau$
2: **for** $\sigma = 1$ to p **do**
3: $\quad L_{\sigma\sigma}, U_{\sigma\sigma}, P_{\sigma\sigma} \leftarrow \mathrm{LU}(\tilde{B}_{\sigma\sigma})$ ▷ partial piv.
4: $\quad$ broadcast $L_{\sigma\sigma}$ and $P_{\sigma\sigma}$ along row-comm
5: $\quad$ broadcast $U_{\sigma\sigma}$ along col-comm
6: $\quad$ **for** $\tau = 1$ to $\sigma - 1$ **do**
7: $\quad\quad$ ▷ depending on admissibility
8: $\quad\quad \begin{cases} X_{\sigma\tau} \leftarrow L_{\sigma\sigma}^{-1} P_{\sigma\sigma} X_{\sigma\tau} \\ \tilde{B}_{\sigma\tau} \leftarrow L_{\sigma\sigma}^{-1} P_{\sigma\sigma} \tilde{B}_{\sigma\tau} \end{cases}$
9: $\quad$ **end for**
10: $\quad$ **for** $\tau = \sigma + 1$ to p **do**
11: $\quad\quad$ **if** $\sigma \times \tau$ is admissible **then**
12: $\quad\quad\quad X_{\sigma\tau}, Y_{\sigma\tau} \leftarrow \mathrm{ARA}(\tilde{B}_{\sigma\tau})$ ▷ $\tilde{B}_{\sigma\tau} \approx X_{\sigma\tau} Y_{\sigma\tau}^T$
13: $\quad\quad\quad X_{\sigma\tau} \leftarrow L_{\sigma\sigma}^{-1} P_{\sigma\sigma} X_{\sigma\tau}$
14: $\quad\quad\quad$ broadcast $X_{\sigma\tau}$ and $Y_{\sigma\tau}$ along col-comm
15: $\quad\quad$ **else**
16: $\quad\quad\quad \tilde{B}_{\sigma\tau} \leftarrow L_{\sigma\sigma}^{-1} P_{\sigma\sigma} \tilde{B}_{\sigma\tau}$
17: $\quad\quad\quad$ broadcast $\tilde{B}_{\sigma\tau}$ along col-comm
18: $\quad\quad$ **end if**
19: $\quad\quad$ **if** $\tau \times \sigma$ is admissible **then**
20: $\quad\quad\quad X_{\tau\sigma}, Y_{\tau\sigma} \leftarrow \mathrm{ARA}(\tilde{B}_{\tau\sigma})$
21: $\quad\quad\quad Y_{\tau\sigma}^T \leftarrow Y_{\tau\sigma}^T U_{\sigma\sigma}^{-1}$
22: $\quad\quad\quad$ broadcast $X_{\tau\sigma}$ and $Y_{\tau\sigma}$ along row-comm
23: $\quad\quad$ **else**
24: $\quad\quad\quad \tilde{B}_{\tau\sigma} \leftarrow \tilde{B}_{\tau\sigma} U_{\sigma\sigma}^{-1}$
25: $\quad\quad\quad$ broadcast $\tilde{B}_{\tau\sigma}$ along row-comm
26: $\quad\quad$ **end if**
27: $\quad$ **end for**
28: $\quad$ **for** $\tau = \sigma + 1$ to p **do**
29: $\quad\quad$ **for** $\delta = \sigma + 1$ to p **do**
30: $\quad\quad\quad$ ▷ depending on admissibility
31: $\quad\quad\quad \begin{cases} \tilde{B}_{\delta\tau} \leftarrow \tilde{B}_{\delta\tau} - \tilde{B}_{\delta\sigma} \tilde{B}_{\sigma\tau} \\ \tilde{B}_{\delta\tau} \leftarrow \tilde{B}_{\delta\tau} - X_{\delta\sigma}(Y_{\delta\sigma}^T \tilde{B}_{\sigma\tau}) \\ \tilde{B}_{\delta\tau} \leftarrow \tilde{B}_{\delta\tau} - (\tilde{B}_{\delta\sigma} X_{\sigma\tau}) Y_{\sigma\tau}^T \\ \tilde{B}_{\delta\tau} \leftarrow \tilde{B}_{\delta\tau} - \begin{cases} (X_{\delta\sigma}(Y_{\delta\sigma}^T X_{\sigma\tau})) Y_{\sigma\tau}^T \\ X_{\delta\sigma}((Y_{\delta\sigma}^T X_{\sigma\tau}) Y_{\sigma\tau}^T) \end{cases} \end{cases}$
32: $\quad\quad$ **end for**
33: $\quad$ **end for**
34: **end for**

ARA is implemented as a batched routine, so lines 12 and 20 are executed simultaneously for all $\tau = \sigma + 1, \dots, p$. Likewise, the Schur complement update in line 31 is implemented using three calls to MAGMA's `magmablas_dgemm_vbatched` (for all $\tau = \sigma + 1$ to p and $\delta = \sigma + 1$ to p). There are 4 cases depending on whether the tiles $\delta\sigma$ and $\sigma\tau$ are admissible. The first multiplication performs $Y_{\delta\sigma}^T X_{\sigma\tau}$, $Y_{\delta\sigma}^T \tilde{B}_{\sigma\tau}$ or $\tilde{B}_{\delta\sigma} X_{\sigma\tau}$; then depending on

the ranks, the result is multiplied with either $X_{\delta\sigma}$ or $Y_{\sigma\tau}^T$, or this step is skipped if one of the tiles was dense. Finally, one last (batched) multiplication performs the update into $\tilde{B}_{\delta\tau}$. The LU factorization in line 3 performs partial pivoting. However, this pivoting is limited to the diagonal blocks of $\tilde{B}$. For most problems, this seems good enough. In case of numerical issues, one can increase the BLR tile size, enable MC64 matching, or enable replacement of small pivot elements.

In the code, different behavior depending on whether a block is compressed or not is handled using polymorphism. A tile can be an object of either class `LRTile` or class `DenseTile`, which are both subclasses of `BLRTile`.

In the distributed memory setting, the BLR matrix is distributed using a 2D block cyclic layout, similar to the ScaLAPACK layout but with non-uniform block sizes (blocks correspond to BLR tiles), see (Claus et al. 2023) for more details including a communication cost analysis. This distribution does not rely on the BLACS (the ScaLAPACK communication library). Sub-communicators are constructed for each of the P_r processor rows and P_c processor columns; these are called row-comm and col-comm in Algorithm 1. The broadcast operations are combined as much as possible to minimize the number of messages. When supported by the system, this MPI communication is performed directly from device memory (GPU-aware MPI communication).

The CPU code does not use batched algorithms, but relies on OpenMP tasks with dependencies specified between tasks using the `depend` clause.

Adapative Randomized Approximation

For low rank compression, we use the adaptive randomized approximation (ARA) from (Boukaram et al. 2019). A fixed rank randomized approximation of a rank-k matrix $B \approx UV^T$ can be computed easily by first computing a random projection $Y = B\Omega$, with Ω a random matrix with $k+p$ columns where p is a small oversampling parameter. Let U be the orthonormal matrix Q from the QR decomposition of Y and $V = B^T U$. However, since the ranks are not known a-priori, ARA performs this adaptively by incrementing the number of random vectors in Ω, orthogonalizing with a blocked Gram-Schmidt scheme with reorthogonalization, all while tracking estimates for both absolute and relative tolerances. For the stopping criterion, we use a relative tolerance as well as an absolute tolerance scaled with the matrix norm (Higham and Mary 2022). ARA is implemented as a variable sized batched routine for CUDA GPUs in the KBLAS library (Abdelfattah et al. 2016). As the tiles are explicitly assembled as dense blocks before compression, the random projection (multiplication with Ω) can be performed using MAGMA's variable sized batched matrix multiplication. Other steps in the ARA algorithm are also implemented as batched routines in KBLAS. If KBLAS is not available (on AMD or Intel platforms), there are several less efficient fall back options. We can use vendor supplied SVD routines like `cusolverDn<t>gesvd` (QR) or `cusolverDn<t>gesvdj` (Jacobi), or `hipsolver<t>gesvdj`. MAGMA also provides singular value decomposition routines `magma_<t>gesvd` (QR) or `magma_<t>gesdd` (divide-and-conquer). However, these MAGMA routines take input from host memory. Alternatively, we are looking into MAGMA's column pivoted QR routine `magma_<t>geqp3_gpu`.

Approximate Sparse Solver

In STRUMPACK, the block low rank scheme is incorporated into the sparse multifrontal solver for compression of the larger frontal matrices, leading to an approximate sparse solver, which can be used as an efficient preconditioner. The three sub-blocks F_{11}, F_{12} and F_{21} are compressed, while for the GPU implementation the temporary matrix F_{22} is stored as dense. Keeping F_{22} dense avoids having to perform Schur complement updates into low rank blocks, and it also simplifies the extend-add operation, which scatters the Schur complement from one front to its parent in the assembly tree. The BLR partitioning for F_{11} is determined by a K-way graph partitioning of the adjacency graph of the sub-block of the sparse matrix corresponding to that supernode (or separator). The number of partitions is determined by the BLR block size.

The diagonal blocks of F_{11}, which are self-interactions, are not compressed in the BLR representation, and by default every block is compressed (but reverted to dense if the rank is too large). For F_{12} and F_{21}, every block is compressed. However, we also implement a strong admissibility condition, where blocks that contain a nonzero from the original sparse matrix are not compressed, since those blocks correspond to neighboring clusters in the adjacency graph of the sparse matrix. Trying to compress every block does not have significant overhead compared to using the stronger admissibility condition, and leads to sligthly better compression ratios.

As discussed before, for the sparse direct solver without BLR compression, the GPU implementation traverses the tree level-per-level, using MAGMA's batched LU, triangular solve and matrix-multiplication for each level. However, this assumes that each front is dense. If there are BLR fronts, those are handled separately. For instance, if there is a BLR front but all of its descendants are dense, then the subtrees of this BLR front can be handled on the GPU level-per-level using the batched routines. After that, the BLR front is constructed/factored, and finally its ancestors in the assembly tree can also be handled one by one on the GPU. For each fully dense subtree, and for each BLR front, device memory is allocated. When no longer needed this device memory is returned to a memory pool to potentially be reused. However, these memory allocations do add considerable overhead compared to the dense algorithm without BLR compression, where all memory can be pre-allocated with a single allocation.

Experiments

For the experiments reported in this section, the numerical factorization has a warmup before the actual run and only the second run is reported. For the first run there is – especially for the GPU code – often a large overhead from initializing the device, loading dynamic libraries, just-in-time compilation of device kernel code, etc. After factorization, a linear system with a single right-hand side

is solved. STRUMPACK supports multiple right-hand sides as well. All experiments are performed in double precision.

The approximate sparse solver with BLR compression is used as a preconditioner for GMRES with restart length 30, and a relative residual tolerance of 10^{-6}. Without compression, the solve is done within an iterative refinement loop, which usually only requires a single iteration.

The BLR leaf size is set to 512 for the GPU tests, and to 256 for the CPU tests. For the GPU tests, we make use of SLATE with a block size of 512 for the 2D block cyclic layout. For the CPU tests, ScaLAPACK is used with a block size of 32. For all tests we set a relative tolerance of $\varepsilon_{\text{rel}} = 10^{-2}$ for the BLR compression. This is clearly an important tuning parameter: a smaller ε_{rel} will give a more accurate preconditioner – fewer GMRES iterations – but at the cost of increased memory usage and more time spent in the factorization.

In the following subsections, we show results for the solution of the 3D visco-acoustic wave propagation, the Poisson equation as well as for a number of larger matrices from the SuiteSparse matrix collection (Davis and Hu 2011). The tests on regular meshes use a simple geometric nested dissection code that finds planes in the mesh, as implemented in STRUMPACK. For the tests using matrices from the SuiteSparse matrix collection the METIS routine `METIS_NodeNDP` is used (instead of the documented `METIS_NodeND`) as this gives more balanced trees for some problems.

Visco-Acoustic Wave Propagation

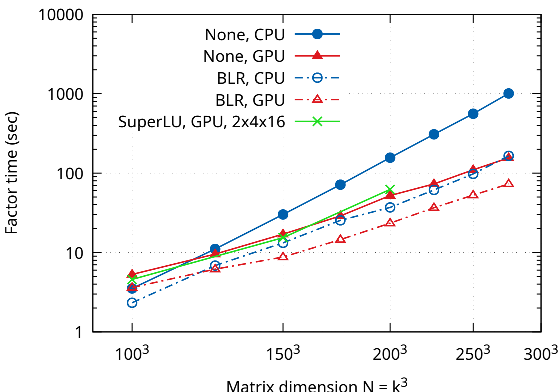
We first consider the 3D visco-acoustic wave propagation governed by the Helmholtz equation

$$\left(\sum_i \rho(\mathbf{x}) \frac{\partial}{\partial x_i} \frac{1}{\rho(\mathbf{x})} \frac{\partial}{\partial x_i} \right) p(\mathbf{x}) + \frac{\omega^2}{\kappa^2(\mathbf{x})} p(\mathbf{x}) = -f(\mathbf{x}). \tag{4}$$

Here $\mathbf{x} = (x_1, x_2, x_3)$, $\rho(\mathbf{x})$ is the mass density, $f(\mathbf{x})$ is the acoustic excitation, $p(\mathbf{x})$ is the pressure wave field, ω is the angular frequency, $\kappa(\mathbf{x}) = v(\mathbf{x})(1 - i/(2q(\mathbf{x})))$ is the complex bulk modulus with the velocity $v(\mathbf{x})$ and quality factor $q(\mathbf{x})$. We solve Eq. (4) by a finite-difference discretization on staggered grids using a 27-point stencil and 8 PML absorbing boundary layers (Operto et al. 2007). This requires direct solution of a sparse linear system where each matrix row contains 27 nonzeros, whose values depend on the coefficients and frequency in Eq. (4). We consider a cubed domain with $v(\mathbf{x}) = 4000$m/s, $\rho(\mathbf{x}) = 1$kg/m^3, $q(\mathbf{x}) = 10^4$. The frequency is set to $\omega = 8\pi$Hz and the grid spacing is set such that there are 15 grid points per wavelength. The problem is solved in double complex precision. Note that this relatively high-frequency problem is very challenging for many iterative solvers and preconditioners.

Figure 1 shows the time spent in the numerical factorization phase for this wave propagation problem, using 32 nodes of Perlmutter, with 4 GPUs per node. The solver with BLR compression on the GPU clearly outperforms the CPU BLR solver and the CPU and GPU exact sparse solvers ("None" means no compression). Note that Perlmutter has CPU-only nodes, but for this test we run the CPU only code on the same nodes as the GPU code. We run with

Figure 1. Time for factorization of the wave propagation problem on 32 nodes of Perlmutter, 128 NVIDIA A100 GPUs. "None" refers to no compression, i.e., the exact sparse solver.



1 MPI rank per GPU, and 16 OpenMP threads per MPI rank. The CPU only experiments also use 4 MPI ranks per node, each with 16 OpenMP threads. The CPU code makes efficient use of OpenMP through OpenBLAS[‖], used in ScaLAPACK, and OpenMP task parallelism for the local subtree computations. For the BLR compression tests in Figures 1 and 3, only those fronts for which F_{11} is larger than 2000×2000 are compressed. Note that one can also specify a minimum threshold for the size of the entire front $[F_{11} F_{12}; F_{21} F_{22}]$ instead. This minimum BLR size, together with the compression tolerance ε_{rel}, is an important tuning parameter. A smaller minimum BLR size will lead to better compression (if the BLR leaf size is small enough), but this can affect accuracy since the errors of low rank approximation propagate along the tree. The minimum BLR size also affects performance.

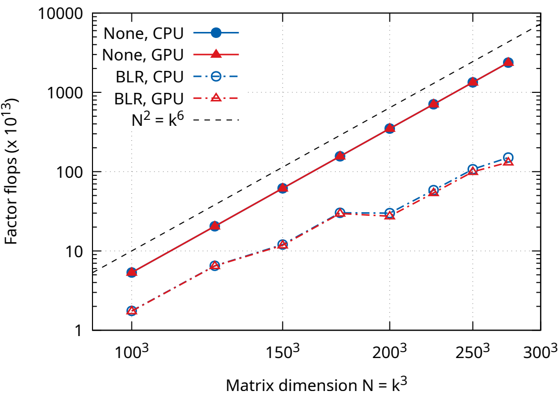
For the 275^3 problem, the factorization phase of the exact GPU solver is about $6.5\times$ faster than that of the CPU solver. The BLR-enabled GPU solver is about $13.8\times$ faster than the CPU exact solver.

Figure 1 also compares with the latest version of the SuperLU_DIST GPU enabled 3D (communication avoiding) factorization, with the optimal 3D processor grid for this setting. Note that SuperLU_DIST uses the METIS reordering, which produces slightly less fill-in than the geometric nested dissection used with STRUMPACK.

Figure 2 shows the scaling of the number of floating point operations for the numerical factorization. As expected, this scales as $N^2 = k^6$ (determined by the dense linear algebra (cubic) on the largest front, which for a 3D k^3 problem is a 2D k^2 plane). From Figure 1, the scaling for the GPU and BLR solvers looks better than for the CPU solver without compression. However, for the GPU solver, this is likely because the GPU solver can more efficiently use the GPUs for larger problems, leading to better floating point per second throughput. For BLR, the scaling of the factorization time looks better due to the irregular increase in the number

[‖] We ran into the occasional segmentation fault when using Cray LibSci, the vendor supplied BLAS, LAPACK and ScaLAPACK implementation, with multiple threads.

Figure 2. Number of floating point operations for the numerical factorization of the wave propagation problem on 32 nodes of Perlmutter, 128 NVIDIA A100 GPUs.



of BLR compressed frontal matrices. See also the discussion of Figure 6.

Figure 3. Time for the triangular solve phase for the wave propagation problem on 32 nodes of Perlmutter, 128 NVIDIA A100 GPUs. For the approximate factorization with BLR the number of iterations are also shown.

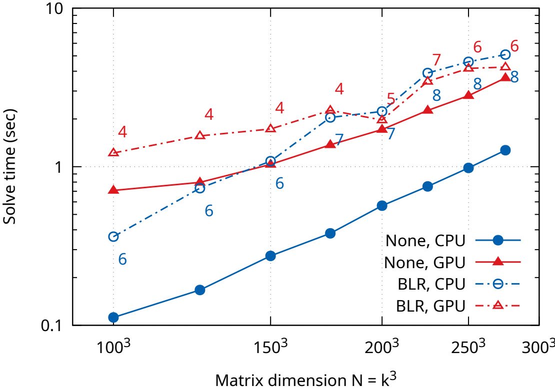


Figure 3 shows the time spent in the triangular solve phase after factorization of the wave propagation problem, and, for the BLR runs, the number of GMRES iterations. For the multi-GPU setting, we currently do not see any improvement from performing the triangular solver on the GPU compared to the CPU. Only when running on a single GPU and the entire factorization fits in device memory, then the factors are kept on the GPU, and the solve can be performed efficiently – without moving the factors first – directly on the GPU.

Poisson Equation

We solve the Poisson equation on a 3D mesh with $N = k^3$ degrees of freedom, using the standard 7-point stencil from a central second order finite difference scheme. Figure 4 shows the factorization time on 32 nodes of Perlmutter, with the same MPI+OpenMP settings as for the wave propagation problem. One can clearly see here, and in Figure 5 which shows the memory usage for the triangular factors, that thanks to the compression, larger problems can be solved, and can be solved faster.

Figure 4. Time for factorization of the Poisson problem on 32 nodes of Perlmutter, 128 NVIDIA A100 GPUs.

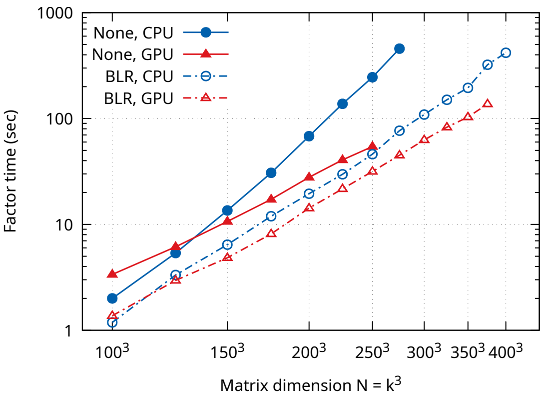


Figure 5. Memory usage for the matrix sparse triangular factors of the Poisson matrix on 32 nodes of Perlmutter, 128 NVIDIA A100 GPUs.

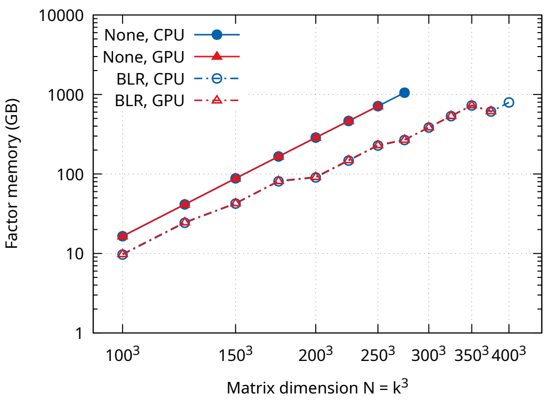


Figure 6. Floating point operations per second for the factorization phase of the Poisson problem on 32 nodes of Perlmutter, 128 NVIDIA A100 GPUs.

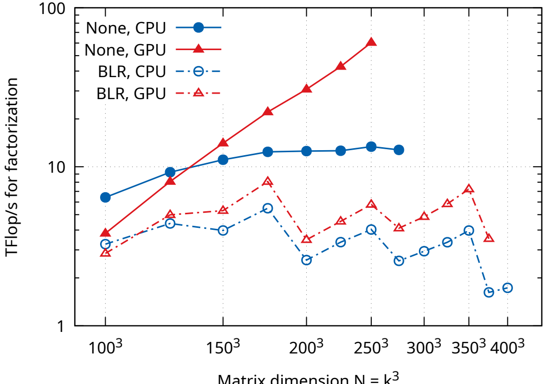


Figure 6 shows the floating point throughput for the numerical factorization of the Poisson problem. As expected, the exact solver (no compression) achieves higher performance, both on CPU and GPU. The drops in performance seen for the BLR experiments as the problem gets bigger can be explained by looking at the number of compressed fronts. For instance, going from 175^3 to 200^2,

the number of BLR fronts increases from 31 (top 5 levels of the assembly tree) to 127 (top 7 levels).

Figure 7. Floating point operations per second for the factorization phase of the Poisson problem on up to 256 nodes of Frontier, with 2048 AMD MI250X Graphics Compute Dies (GCDs).

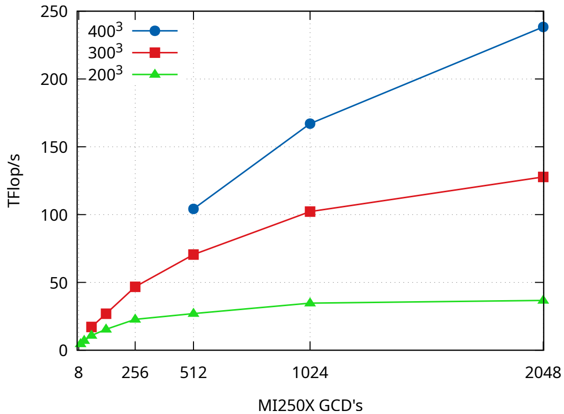


Figure 7 shows scaling with increasing number of nodes, and GPUs, on the Frontier system for the factorization phase of the Poisson problem, without BLR compression, on meshes with 200^3, 300^3 and 400^3 degrees of freedom. Each node of Frontier has 4 AMD MI250X devices, with each MI250X device having 2 Graphics Compute Dies (GCDs). In practice, each GCD is treated as a separate device and we run with 1 MPI rank per GCD. This shows that the code still scales, even up to thousands of GPUs/MPI ranks, as long as the problem is sufficiently large.

SuiteSparse Test Problems

Table 2 shows results for a number of larger matrices from the SuiteSparse matrix collection (Davis and Hu 2011). Since these problems are of medium size, a single GPU is used for the tests. An efficient single GPU solver can be used in a domain decomposition or block Jacobi preconditioner. The A100 GPU is from Perlmutter and the PVC GPU is an Intel Data Center GPU Max Series GPU (codename Ponte Vecchio) from Aurora. These single GPU results are compared with 8 cores from an AMD EPYC 7763 CPU from Perlmutter.

As can be seen from Table 2, the GPU implementation generally outperforms the corresponding CPU code. Without compression, the factorization is on average 12.0× faster on A100 compared to 8 CPU cores. Likewise, when BLR compression is enabled, the A100 is 3.6× faster than 8 CPU cores. However, note that the GPU code without compression is still faster than the BLR GPU code. This illustrates the raw power of the GPU when applied to large dense linear algebra operations. The GPU code without compression is also much simpler than the BLR code, with less overhead, fewer kernel launches, memory allocations, memory transfers, and synchronizations. Remember, in the previous sections we were able to show performance improvements of BLR GPU vs the GPU code without compression for large matrices, the matrix sizes in Table 2 are not large enough. We will work towards future optimizations in the BLR code to reduce the overhead. Note that there is some difference in terms of number of iterations and compression ratio between the CPU and the GPU code. This is due to the different low rank compression algorithms.

The A100 experiments both with and without compression rely on MAGMA. For Intel PVC however, MAGMA was not used, and instead the Intel oneAPI Math Kernel Library (oneMKL) was used. This is because, at the time of writing this, MAGMA does not have a stable release with SYCL support. The oneMKL library provides variable sized batched routines for dense LU, triangular solve and matrix multiplication. However, since these are not sufficiently optimized yet, we use our own naive kernel for fronts with F_{11} smaller than 32×32, and call the oneMKL routines `getrf`, `getrs` and `gemm` for the larger fronts. Despite these issues, the performance on PVC is quite competitive compared to A100.

Finally, Table 2 also compares with cuDSS**, also using LU factorization, on A100. For the exact factorization, the STRUMPACK solver is on average 1.87× faster than cuDSS for this test set. However, cuDSS's solve phase is more efficient. One possible explanation for this is that cuDSS expects the input and output arguments for the solver (right-hand side and solution) in device memory, while STRUMPACK takes host memory, and thus requires extra transfers. We plan to develop a fully GPU resident solve phase in the near future.

Conclusion

Thanks to the Exascale Computing Project (ECP)'s impactful structure of bringing people of various scientific focus areas together on the common vision of working towards exascale, we successfully ported the direct solver as well as the block low rank compressed approximate solver in STRUMPACK to three programming models, CUDA, HIP/ROCm and SYCL, which allows us to target the GPUs from all three vendors of each DOE compute facility, NVIDIA GPUs on Perlmutter at NERSC, AMD GPUs on Frontier at OLCF and Intel GPUs on Aurora at ALCF.

We studied the benefits of the GPU implementation of a block low rank (BLR) compressed approximate solver in STRUMPACK. Based on the experiments, we conclude that the BLR compression on GPUs is particularly beneficial for large problem sizes. For the visco-acoustic wave propagation and the Poisson equation, we notice a reduction of factorization time compared to the CPU-only version as well as GPU alternatives. We would like to note that the GPU code without compression is faster than the BLR GPU code for the medium sized matrices from the SuiteSparse collection. Further optimizations, e.g., reduce device memory allocations and device synchronizations, would likely make BLR beneficial for a wider range of problem sizes.

We gave a detailed description of our GPU implementation and approach to performance portability. The resulting code is modular and the higher level functionality can be extended without having to write much vendor specific

**Reported results are for cuDSS version 0.1.0 (`https://developer.nvidia.com/cudss-downloads`). Timings are nearly identical with version 0.2.0, but several cases terminate with a segmentation fault.

			no compression								BLR($\varepsilon_{\text{rel}} = 10^{-2}$)							
			CPU		A100		cuDSS A100		PVC		CPU				A100			
matrix	N $\times 10^3$	nnz $\times 10^3$	fact (sec)	solve (sec)	fact (sec)	solve (sec)	fact (sec)	solve (sec)	fact (sec)	solve (sec)	fact (sec)	solve (sec)	its	comp (%)	fact (sec)	solve (sec)	its	comp (%)
Serena	1,391	64,531	229.6	1.07	17.9	1.2	48.2	0.4	14.3	0.6	76.4	5.2	10	34.4	17.3	3.4	6	39.7
Geo_1438	1,437	63,156	151.9	1.04	12.7	1.0	33.0	0.3	10.6	0.6	60.4	7.2	13	45.6	16.9	4.5	7	54.2
Hook_1498	1,498	60,917	76.1	0.70	7.4	0.7	15.4	0.2	6.5	0.4	29.7	14.5	35	46.7	12.5	4.1	10	52.0
ML_Geer	1,504	110,879	23.6	0.51	2.0	0.3	5.4	0.1	4.5	0.3	11.5	10.1	27	64.6	8.7	4.0	11	66.6
Transport	1,602	23,500	40.9	0.63	3.2	0.3	10.8	0.2	5.0	0.3	21.3	10.8	25	52.0	8.8	4.5	11	58.1
Flan_1565	1,565	117,406	32.8	0.7	3.0	0.4	8.8	0.1	5.7	0.4	20.5	40.6	86	62.3	12.3	25.0	54	65.7
Cube_Coup_dt0	2,164	129,133	OOM	OOM	62.1	2.4	80.7	0.6	23.2	0.9	223.9	18.1	18	31.0	46.0	7.3	7	38.5

Table 2. Results for the numerical factorization and solve for a number of matrices from the SuiteSparse matrix collection. Compression ratio (comp %) refers to the size of the final LU factors relative to the exact solver without compression. CPU runs use 8 cores of an AMD EPYC 7763 with OpenMP parallelism, GPU runs use a single GPU (A100, PVC).

code. However, some kernels are still duplicated for various vendors. We plan to port those to a single portable framework such as OpenMP off-loading, Kokkos or Raja. We will also explore the use of runtime schedulers like PARSEC and StarPU, which could be especially beneficial for the rank-structured solvers, where the numerical ranks are not known a-priori.

Acknowledgements

We extend our gratitude to Yang Liu who collaborated with us on the SuperLU comparison results. His expertise and contributions have significantly enhanced the quality of this work. We are very grateful to Ahmad Abdelfattah and Stan Tomov for help with MAGMA, to Mark Gates for help with SLATE, to Abhishek Bagusetty and Dahai Guo for help with the SYCL implementation, and to Paul Lin for help with the installation of cuDSS. We also thank ECP management for their support.

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 research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231.

An award for computer time was provided by the U.S. Department of Energy's (DOE) Innovative and Novel Computational Impact on Theory and Experiment (INCITE) Program. This research used supporting resources at the Argonne and the Oak Ridge Leadership Computing Facilities. The Argonne Leadership Computing Facility at Argonne National Laboratory is supported by the Office of Science of the U.S. DOE under Contract No. DE-AC02-06CH11357. The Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory is supported by the Office of Science of the U.S. DOE under Contract No. DE-AC05-00OR22725.

References

Abdelfattah A, Keyes D and Ltaief H (2016) KBLAS: An Optimized Library for Dense Matrix-Vector Multiplication on GPU Accelerators. *ACM Transactions on Mathematical Software (TOMS)* 42(3): 1–31.

Amestoy PR, Buttari A, L'Excellent JY and Mary T (2019) Performance and Scalability of the Block Low-Rank Multifrontal Factorization on Multicore Architectures. *ACM Trans. Math. Softw.* 45(1). DOI:10.1145/3242094.

Amestoy PR, Duff IS, Koster J and L'Excellent JY (2001) A Fully Asynchronous Multifrontal Solver Using Distributed Dynamic Scheduling. *SIAM Journal on Matrix Analysis and Applications* 23(1): 15–41.

Aminfar A and Darve E (2016) A fast, memory efficient and robust sparse preconditioner based on a multifrontal approach with applications to finite-element matrices. *International Journal for Numerical Methods in Engineering* 107(6): 520–540.

Anderson R, Andrej J, Barker A, Bramwell J, Camier JS, Cerveny J, Dobrev V, Dudouit Y, Fisher A, Kolev T et al. (2021) MFEM: A modular finite element methods library. *Computers & Mathematics with Applications* 81: 42–74.

Azad A, Buluc A, Li X, Wang X and Langguth J (2020) A Distributed-Memory Algorithm for Computing a Heavy-Weight Perfect Matching on Bipartite Graphs. *SIAM J. Scientific Computing* 42(4): C143–C168.

Azad A, Selvitopi O, Hussain MT, Gilbert JR and Buluç A (2021) Combinatorial BLAS 2.0: Scaling combinatorial algorithms on distributed-memory systems. *IEEE Transactions on Parallel and Distributed Systems* 33(4): 989–1001.

Balay S, Abhyankar S, Adams MF, Benson S, Brown J, Brune P, Buschelman K, Constantinescu EM, Dalcin L, Dener A, Eijkhout V, Faibussowitsch J, Gropp WD, Hapla V, Isaac T, Jolivet P, Karpeev D, Kaushik D, Knepley MG, Kong F, Kruger S, May DA, McInnes LC, Mills RT, Mitchell L, Munson T, Roman JE, Rupp K, Sanan P, Sarich J, Smith BF, Zampini S, Zhang H, Zhang H and Zhang J (2023) PETSc Web page. https://petsc.org/. URL https://petsc.org/.

Boukaram W, Turkiyyah G and Keyes D (2019) Randomized GPU algorithms for the construction of hierarchical matrices from matrix-vector operations. *SIAM Journal on Scientific Computing* 41(4): C339–C366.

Chen Y, Davis TA, Hager WW and Rajamanickam S (2008) Algorithm 887: CHOLMOD, Supernodal Sparse Cholesky Factorization and Update/Downdate. *ACM Trans. Math. Softw.* 35(3). DOI:10.1145/1391989.1391995. URL https://doi.org/10.1145/1391989.1391995.

Chevalier C and Pellegrini F (2008) PT-Scotch: A tool for efficient parallel graph ordering. *Parallel computing* 34(6-8): 318–331.

Claus L, Ghysels P, Liu Y, Nhan TA, Thirumalaisamy R, Bhalla APS and Li S (2023) Sparse approximate multifrontal factorization with composite compression methods. *ACM*

Transactions on Mathematical Software 49(3): 1–28.

Davis TA (2004) Algorithm 832: UMFPACK V4.3—an Unsymmetric-Pattern Multifrontal Method. *ACM Trans. Math. Softw.* 30(2): 196—199. DOI:10.1145/992200.992206. URL https://doi.org/10.1145/992200.992206.

Davis TA and Hu Y (2011) The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)* 38(1): 1–25.

Duff IS, Erisman AM and Reid JK (2017) *Direct methods for sparse matrices*. Oxford University Press.

Duff IS and Koster J (1999) The Design and Use of Algorithms for Permuting Large Entries to the Diagonal of Sparse Matrices. *SIAM J. Matrix Anal. Appl.* 20(4): 889–901. DOI:10.1137/S0895479897317661. URL https://doi.org/10.1137/S0895479897317661.

Gates M, Kurzak J, Charara A, YarKhan A and Dongarra J (2019) SLATE: Design of a Modern Distributed and Accelerated Linear Algebra Library. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19. New York, NY, USA: Association for Computing Machinery. ISBN 9781450362290. DOI:10.1145/3295500.3356223. URL https://doi.org/10.1145/3295500.3356223.

Ghysels P, Li XS, Rouet FH, Williams S and Napov A (2016) An Efficient Multicore Implementation of a Novel HSS-Structured Multifrontal Solver Using Randomized Sampling. *SIAM J. Sci. Comput.* 38(5): S358–S384. DOI:10.1137/15M1010117. URL https://doi.org/10.1137/15M1010117.

Ghysels P and Synk R (2022) High performance sparse multifrontal solvers on modern GPUs. *Parallel Comput.* 110: 102897. DOI:https://doi.org/10.1016/j.parco.2022.102897. URL https://www.sciencedirect.com/science/article/pii/S0167819122000059.

Ghysels P, Xiaoye SL, Gorman C and Rouet FH (2017) A robust parallel preconditioner for indefinite systems using hierarchical matrices and randomized sampling. In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, pp. 897–906.

Gupta A (2000) WSMP: Watson Sparse Matrix Package Part II – direct solution of general systems. Technical report, IBM T. J. Watson Research Center. Https://s3.us.cloud-object-storage.appdomain.cloud/res-files/1331-wsmp2.pdf.

Hénon P, Ramet P and Roman J (2002) PaStiX: a High-Performance Parallel Direct Solver for Sparse Symmetric Positive Definite Systems. *Parallel Computing* 28(2): 301–321.

Heroux MA, Bartlett RA, Howle VE, Hoekstra RJ, Hu JJ, Kolda TG, Lehoucq RB, Long KR, Pawlowski RP, Phipps ET et al. (2005) An overview of the Trilinos project. *ACM Transactions on Mathematical Software (TOMS)* 31(3): 397–423.

Higham NJ and Mary T (2022) Solving block low-rank linear systems by LU factorization is numerically stable. *IMA Journal of Numerical Analysis* 42(2): 951–980.

Karypis G and Kumar V (1998) A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.* 20(1): 359–392. DOI:10.1137/S1064827595287997. URL https://doi.org/10.1137/S1064827595287997.

Li XS and Demmel JW (1998) Making sparse Gaussian elimination scalable by static pivoting. In: *Proceedings of SC98: High Performance Networking and Computing Conference*. Orlando, Florida.

Li XS and Demmel JW (2003) SuperLU_DIST: A Scalable Distributed-Memory Sparse Direct Solver for Unsymmetric Linear Systems. *ACM Trans. Math. Softw.* 29(2): 110—140. DOI:10.1145/779359.779361. URL https://doi.org/10.1145/779359.779361.

Liu Y, Ghysels P, Claus L and Li XS (2021) Sparse approximate multifrontal factorization with butterfly compression for high-frequency wave equations. *SIAM Journal on Scientific Computing* 43(5): S367–S391.

Mary T (2017) *Block Low-Rank multifrontal solvers: complexity, performance, and scalability*. PhD Thesis, l'Université de Toulouse.

Nies R and Hoelzl M (2019) Testing performance with and without block low rank compression in MUMPS and the new PaStiX 6.0 for JOREK nonlinear MHD simulations. *arXiv e-prints* : arXiv:1907.13442.

Operto S, Virieux J, Amestoy P, L'Excellent JY, Giraud L and Ali HBH (2007) 3D finite-difference frequency-domain modeling of visco-acoustic wave propagation using a massively parallel direct solver: A feasibility study. *Geophysics* 72(5): SM195–SM211.

Pellegrini F and Roman J (1996) Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In: *High-Performance Computing and Networking: International Conference and Exhibition HPCN EUROPE 1996 Brussels, Belgium, April 15–19, 1996 Proceedings 4*. Springer, pp. 493–498.

Schloegel K, Karypis G and Kumar V (1997) Multilevel diffusion schemes for repartitioning of adaptive meshes. *Journal of Parallel and Distributed Computing* 47(2): 109–124.