Lawrence Berkeley National Laboratory

LBL Publications

Title

High performance sparse multifrontal solvers on modern GPUs

Permalink

https://escholarship.org/uc/item/7tv84567

Authors

Ghysels, Pieter

Synk, Ryan

Publication Date

2022-05-01

DOI

10.1016/j.parco.2022.102897

Copyright Information

This work is made available under the terms of a Creative Commons Attribution-NonCommercial License, available at https://creativecommons.org/licenses/by-nc/4.0/

Peer reviewed

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

High performance sparse multifrontal solvers on modern GPUs

Pieter Ghysels[*], Ryan Synk[†]

Abstract

We have ported the numerical factorization and triangular solve phases of the sparse direct solver `STRUMPACK` to GPU. `STRUMPACK` implements sparse LU factorization using the multifrontal algorithm, which performs most of its operations in dense linear algebra operations on so-called frontal matrices of various sizes. Our GPU implementation off-loads these dense linear algebra operations, as well as the sparse scatter-gather operations between frontal matrices. For the larger frontal matrices, our GPU implementation relies on vendor libraries such as cuBLAS and cuSOLVER for NVIDIA GPUs and rocBLAS and rocSOLVER for AMD GPUs. For the smaller frontal matrices we developed custom CUDA and HIP kernels to reduce kernel launch overhead. Overall, high performance is achieved by identifying submatrix factorizations corresponding to sub-trees of the multifrontal assembly tree which fit entirely in GPU memory. The multi-GPU setting uses SLATE (Software for Linear Algebra Targeting Exascale) as a modern GPU-aware replacement for ScaLAPACK. On 4 nodes of SUMMIT the code runs $\sim 10\times$ faster when using all 24 V100 GPUs compared to when it only uses the 168 POWER9 cores. On 8 SUMMIT nodes, using 48 V100 GPUs, the sparse solver reaches over 50TFlop/s. Compared to SuperLU, on a single V100, for a set of 17 matrices our implementation is faster for all but one matrix, and is on average $5\times$ (median $4\times$) faster.

1 Introduction

Sparse direct solvers are popular in both academia and industry because of their robustness. Unlike iterative solvers and preconditioners, sparse direct solvers do not suffer from convergence issues and do not require much tuning. However, correctly implementing a sparse direct solver in a scalable and high-performance way is very challenging. In this paper, we present an efficient multi-GPU algorithm for sparse multifrontal LU factorization and triangular solve. The work presented here is part of the Exascale Computing Project (ECP), which is making substantial investments in pursuit of application performance gains by refactoring codes and developing products to effectively utilize accelerated nodes.

In sparse matrix factorization, the sparse triangular factors are typically much more dense than the original input matrix due to fill-in, i.e., extra nonzeros that are generated during the Gaussian elimination process. However, with a proper ordering of the input matrix this fill-in can be minimized, and the fill-in entries will appear in the sparse triangular factors in dense sub-blocks. Hence, with a good ordering, most of the work in a sparse direct solver is performed using dense linear algebra operations on these dense sub-blocks, also referred to as supernodes or frontal matrices in the multifrontal setting. Hence, the numerical factorization phase can use higher level BLAS routines and achieve relatively high performance on modern multi-core architectures. However, since many of these blocks are small, it is hard to fully exploit the enormous performance potential of modern GPUs.

The GPU-accelerated multifrontal algorithm we present splits the problem into independent subtrees for which the factorization fits entirely in device (GPU) memory. Then this sub-tree is traversed level by level, using hand coded device kernels for the smallest matrices and vendor optimized libraries for the larger ones. With this approach, we minimize data movement between the CPU and the GPU, and reduce kernel launch overheads. We also present a multi-GPU version of the code which uses SLATE [1] – Software for Linear Algebra Targeting Exascale – a modern replacement for ScaLAPACK funded through the ECP project. SLATE can use the traditional ScaLAPACK 2D block-cyclic layout but adds GPU acceleration. A natural solution for the efficient factorization of all dense matrices on a single level of the sparse assembly tree would be the use of batched dense linear algebra routines. Several libraries provide batched BLAS operations, for instance MAGMA [2], cuBLAS, Kokkos kernels and MKL. MAGMA even provides batched LU with piv-

[*]pghysels@lbl.gov, Lawrence Berkeley National Laboratory, Computational Research Division, 1 Cyclotron Road, Berkeley, CA 94720-8150

[†]University of Maryland

oting. However, none of these libraries provide variable sized batched LU factorization with pivoting, which is what our solver needs.

Several high quality sparse direct solvers are available. For instance MUMPS [3] is a well-known parallel multifrontal solver, but has no GPU support. The non-symmetric multifrontal solver UMFPACK [4] is used in Matlab, but has no GPU or MPI support. The Watson Sparse Matrix Package (WSMP) [5, 6] has no officially released version with GPU support. UMFPACK, MUMPS and WSMP are so-called multifrontal solvers, see [7, 8] and Section 2. Other sparse direct solvers include PasTiX [9] and SuperLU_Dist [10]. Both SuperLU_Dist [11] and PaStiX [12] have support for distributed memory and GPU acceleration. However, since both SuperLU_Dist and PaStiX are supernodal, but not multifrontal, they typically operate on dense linear blocks that are smaller than the dense blocks found in multifrontal methods. The smaller blocks make it hard to fully exploit the potential of GPU accelerated nodes. In the main SuperLU factorization code, the only operations performed on the GPU are the dense matrix-matrix multiplications for the Schur updates. SuperLU provides a 3D factorization algorithm in which also certain sparse gather/scatter operations are off-loaded, but the dense LU factorization for the supernodes is still performed on the CPU. Furthermore since our implementation operates on relatively large distributed dense matrices, we can make use of ScaLAPACK, or the modern ScaLAPACK alternative SLATE, along with all the optimizations that go into these libraries. SuperLU uses a very different parallel layout of the sparse triangular factors, and needs to implement this functionality internally.

The outline of the paper is as follows. We give a brief overview of the sparse multifronal LU factorization in Section 2 where we also discuss the MPI+OpenMP parallelization. In Section 3 we describe a modified parallelization scheme for the multifrontal algorithm that is more amenable to GPU acceleration. Section 4 shows performance results for a number of different GPUs and a range of linear systems. Section 5 discusses the `STRUMPACK` software library in more detail and gives some pointers that should help with the installation. We conclude the paper with Section 6, where we also discuss how the work presented here can be a first step towards the development of a class of high performance, GPU-aware and robust preconditioners based on rank-structured approximate multifrontal factorization.

2 Sparse Multifrontal LU Decomposition

We consider the LU factorization of a sparse matrix $A \in \mathbb{C}^{N \times N}$, as $P\,(D_r A D_c Q_c)\,P^\top = LU$, where P and Q_c are permutation matrices, D_r and D_c are diagonal row and column scaling matrices and L and U are sparse lower and upper-triangular respectively. D_r, D_c and Q_c are optional and are applied for numerical stability. Q_c aims to maximize the magnitude of the elements on the matrix diagonal. D_r and D_c scale the matrix such that the diagonal entries are one in absolute value and all off-diagonal entries are less than one. This step is implemented using the sequential MC64 code [13] or the parallel method – without the diagonal scaling – described in [14]. The permutation P is applied symmetrically and is used to minimize the fill, i.e., the number of non-zero entries in the sparse factors L and U. This permutation is computed from the symmetric sparsity structure of $A + A^\top$. For large problems the preferred ordering is typically based on the nested dissection heuristic, as implemented in METIS [15] or Scotch [16].

The multifrontal method [7] relies on a graph called the assembly tree to guide the computation. Each node τ of the assembly tree corresponds to a dense frontal matrix, with the following 2×2 block structure: $F_\tau = \begin{bmatrix} F_{11} & F_{12} \\ F_{21} & F_{22} \end{bmatrix}$, with F_{11} of dimension $\#I_\tau^{\mathrm{s}}$ and F_{22} of dimension $\#I_\tau^{\mathrm{u}}$ (the superscripts$s$ and u stand for *separator* and *update* respectively), see also Fig. 1. Let $n_\tau = \#I_\tau^{\mathrm{s}} + \#I_\tau^{\mathrm{u}}$ denote the dimension of F_τ. A frontal matrix is an intermediate dense submatrix in sparse Gaussian elimination. The rows and columns corresponding to the F_{11} block are called the fully-summed variables because when the front is constructed, these variables have received all their Schur complement updates. The fully-summed variables correspond to I_τ^s, which are mutually exclusive, with $\bigcup_\tau I_\tau^s = \{1, \ldots, N\}$. In the context of nested dissection, the sets I_τ^s correspond to individual vertex separators. The I_τ^u index sets define the temporary Schur complement update blocks F_{22}. The use of this temporary F_{22} block to pass Schur complement updates up along the tree is the main difference between multifrontal methods and other sparse direct solvers, such as for instance SuperLU, in which Schur updates are passed up in the tree not just to the parent, but to multiple ancestors. Note that the frontal matrices tend to get bigger toward the root of the assembly tree. Furthermore, if ν is a child of τ in the assembly tree, then $I_\nu^u \subset \{I_\tau^s \cup I_\tau^u\}$. For the root node t, $I_t^u \equiv \emptyset$. When considering a single front, we will omit the τ subscript.

The multifrontal method casts the factorization of a sparse matrix into a series of partial factorizations of many smaller dense matrices and Schur complement updates. It consists in a bottom-up traversal of the assembly tree following a topological order. Processing a node consists of four steps:

1. Assembling the frontal matrix F_τ, i.e., combining elements from the sparse matrix A with the children's (ν_1 and ν_2) Schur complement updates F_{22} into the (larger) F_τ. This involves a scatter operations and is called extend-add, denoted by $\mathbin{\leftrightarrow\hspace{-0.75em}\updownarrow}$:

$$F_\tau = \begin{bmatrix} A(I^s_\tau, I^s_\tau) & A(I^s_\tau, I^u_\tau) \\ A(I^u_\tau, I^s_\tau) & \end{bmatrix} \mathbin{\leftrightarrow\hspace{-0.75em}\updownarrow} F_{22;\nu_1} \mathbin{\leftrightarrow\hspace{-0.75em}\updownarrow} F_{22;\nu_2} \tag{1}$$

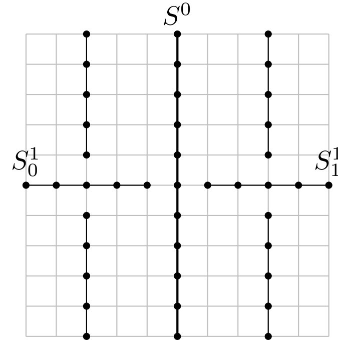
2. Elimination of the fully-summed variables in the F_{11} block, i.e., dense LU factorization with partial pivoting of F_{11}.
3. Updating the off-diagonal blocks F_{12} and F_{21}.
4. Compute the Schur complement update:

$$F_{22} \leftarrow F_{22} - F_{21} F_{11}^{-1} F_{12}.$$

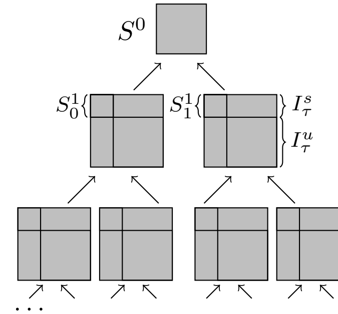
   We will also denote the Schur updated F_{22} for front τ as the contribution block CB_τ. F_{22} is temporary storage (pushed on a stack), and can be released as soon as it has been used in the front assembly (step (1)) of the parent node.

After the numerical factorization, the lower triangular sparse factor is available in the F_{21} and F_{11} blocks and the upper triangular factor in the F_{11} and F_{12} blocks. These can then be used to very efficiently solve linear systems, using forward and backward substitution. A high-level overview of the multifrontal algorithm for solving a sparse linear system is given in the pseudo-code in Algorithm 1.

Figure 1 illustrates the multifrontal algorithm for a sparse matrix resulting from the discretization of a partial differential equation using a 5-point finite difference stencil on a regular two-dimensional 11×11 mesh. Figure 1a shows the mesh and the top 3 levels of the nested dissection ordering. Nested dissection is a heuristic algorithm for the ordering of a sparse matrix to reduce the fill-in in the sparse factors. It is based on recursively finding vertex separators. The vertical line marked S^0 is the root separator and this separator corresponds to the root of the multifrontal assembly tree, see Fig. 1b. The next level separators, S^1_0 and S^1_1 correspond to the F_{11} blocks of the



(a)



(b)

Figure 1: (a) The top three levels of nested dissection for an 11^2 mesh. The root separator S^0 is a vertical 11 point line. The next level separators are S^1_0 and S^1_1. The root separator corresponds to the top level front in (b), and similarly for the next level down in the assembly/frontal tree. For the lower levels, the fronts are regular dense matrices. Note that the fronts in (b) are to scale, but from this figure it is not obvious that the fronts typically get smaller lower in the tree (except for the root front, which has no Schur complement).

Algorithm 1 Sparse multifrontal symbolic analysis, factorization and solve

Input: $A \in \mathbb{C}^{N\times N}$, $b \in \mathbb{C}^N$
Output: $x \approx A^{-1}b$

1: $A \leftarrow D_r A D_c Q_c$ ▷ (optional) col perm & scaling
2: $A \leftarrow P A P^\top$ ▷ symm fill-reducing reordering
3: Build assembly tree: define $I^{\rm s}_\tau$ and $I^{\rm u}_\tau$ for every frontal matrix F_τ
4: **for** nodes τ in assembly tree in topological order **do**
5: ▷ Build front F from sparse entries and child contributions (extend-add)
6: $F_\tau \leftarrow \begin{bmatrix} A(I^{\rm s}_\tau, I^{\rm s}_\tau) & A(I^{\rm s}_\tau, I^{\rm u}_\tau) \\ A(I^{\rm u}_\tau, I^{\rm s}_\tau) & 0 \end{bmatrix} \updownarrow\!\!\!\leftrightarrow F_{\nu_1;22} \updownarrow\!\!\!\leftrightarrow F_{\nu_2;22}$
7: $P_\tau L_\tau U_\tau \leftarrow F_{\tau;11}$ ▷ LU with partial pivoting
8: $F_{\tau;12} \leftarrow L_\tau^{-1} P_\tau^\top F_{\tau;12}$
9: $F_{\tau;21} \leftarrow F_{\tau;21} U_\tau^{-1}$
10: $F_{\tau;22} \leftarrow F_{\tau;22} - F_{\tau;21} F_{\tau;12}$ ▷ Schur update
11: **end for**
12: $x \leftarrow D_c Q_c P^\top$ bwd-solve (fwd-solve $(P D_r b)$)

next lower level in the assembly tree. Hence, typically the larger frontal matrices are found near the root of the assembly tree since the separators tend to get smaller further in the nested dissection recursion. For regular $N = k^2$ 2D or $N = k^3$ 3D grids, the cost of dense LU factorization on a front corresponding to the largest separator (k for 2D and $k\times k$ for 3D) determines the asymptotic complexity of the solver. This leads to $\mathcal{O}(N^{\frac{3}{2}})$ and $\mathcal{O}(N^2)$ asymptotic complexity for 2D and 3D problems respectively. STRUMPACK provides several approximate solvers with lower complexity, see also the discussion in Section 6.

For a more detailed discussion of the multifrontal method, see [8]. Since its introduction in [7], the multifrontal method has been widely used in sparse direct solvers. Popular multifrontal implementations are available through the MUMPS solver [3], in UMFPACK (unsymmetric) [4] and in WSMP [5]. We implemented the multifrontal method in the `STRUMPACK` library[1], using C++, MPI, OpenMP, CUDA and HIP. `STRUMPACK` supports real/complex arithmetic, single/double precision and 32/64-bit integers.

Pivoting Issues The row pivoting in `STRUMPACK` is restricted to the F_{11} blocks only. For most problems this works well, especially when it is used in combination with (MC64) matrix reordering and scaling. However, in certain cases this can still fail. We have observed this for instance for linear systems from the interior point method used in optimization (ExaSGD ECP project collaboration). In this setting the linear systems become gradually more and more ill-conditioned. Therefore, whenever our solver encounters a zero or very small pivot element, i.e., a diagonal entry of the upper-triangular factor, this entry is replaced by a larger value. For the threshold for small pivots the value $\tau = \sqrt{\epsilon_{\rm mach}}\|A\|_1$ is used. A small positive pivot element is replaced by τ, a negative one by $-\tau$. The resulting factorization is only approximate, but is often still a very good preconditioner, requiring only a few iterations of iterative refinement.

2.1 MPI+OpenMP Parallelization

Following the assembly tree, the distributed algorithm sets up nested MPI sub-communicators to facilitate the computation at each node and its subtree. At the root of the tree, we create a two-dimensional process grid using all the available processes in the main MPI communicator, and distribute the frontal matrix over this grid using the ScaLAPACK 2D block-cyclic data layout [17]. The process grid is $P_r \times P_c$ with $P_r = \left\lfloor \sqrt{P} \right\rfloor$ and $P_c = P/P_r$, with at most $\left\lfloor \sqrt{P} \right\rfloor - 1$ idle processes. Next, the root MPI communicator is split in two communicators proportionally to the number of flops required by the subtrees rooted at the children of the root node. Again each child can construct a ScaLAPACK BLACS context using a 2D process grid and distribute the child's frontal matrix over this subgrid. This is repeated recursively until the MPI communicator has only one process in it.

Numerical factorization requires traversing the assembly tree from the leafs to the root. For a local sub-tree, i.e., a tree assigned to a single MPI process, this is done using OpenMP. When moving up to the distributed part of the assembly tree, communication between fronts is required for the extend-add operation. This is implemented using an `MPI_Alltoallv` on the MPI communicator of the parent node, which includes all the processes from the two sub-communicators corresponding to the two children of that node.

The entire local tree traversal is done in a single OpenMP parallel region using task parallelism. The factorization of a node will first spawn two tasks, one for each of the sub-trees rooted at its children. When these two child tree traversals are completed – guaranteed by a `pragma omp taskwait` – the factorization of the node itself can proceed. In order to exploit both the parallelism from the assembly

[1] `http://portal.nersc.gov/project/sparse/strumpack/`

tree and the parallelism available in the dense linear algebra kernels at each node, we added OpenMP tasking to the BLAS and LAPACK routines called at each node in the OpenMP parallel region. Using dynamically scheduled OpenMP task parallelism ensures good performance and load balance within a node.

To exploit thread parallelism at the distributed memory levels of the assembly tree, we use ScaLAPACK with multithreaded BLAS/LAPACK kernels. The local sub-trees, however, are traversed within an OpenMP region so we do not want multithreaded BLAS here as this would lead to over-subscription of cores and hence poor performance. Fortunately, the default behavior of most vendor supplied BLAS[2] implementations is to run sequentially when called from within a parallel region and multithreaded otherwise. We refer to [18] for more details about the threaded implementation.

3 GPU Off-Loading Strategy

Algorithm 2 shows our new GPU-accelerated multifrontal factorization algorithm. Compared to the OpenMP tasking parallelization scheme described in Section 2.1, now the tree is traversed level by level, from the leafs to the root. On a given level the algorithm needs to perform dense LU decomposition, LU solve and matrix multiplication for every front on that level. This is a perfect fit for so-called batched BLAS kernels, which have recently gained much attention, and are available in several libraries, such as MAGMA [2], cuBLAS, Kokkos kernels and MKL. However, none of these libraries provide variable sized batched LU factorization with pivoting. Therefore we rely on regular cuBLAS (and rocBLAS) calls from within multiple streams, and for the very small frontal matrices, we developed our own batched kernels for partial LU factorization of a front, see Section 3.2.

The input to Algorithm 2 is a sparse matrix A and the root node of the assembly tree. Lines 5 to 24 handle the case where the multifrontal factorization does not completely fit in device memory. On Line 5 the algorithm checks if the device has sufficient memory to hold the entire multifrontal factorization of the assembly tree rooted at node τ in device memory. If this is not the case, then the code will first traverse the two sub-trees rooted at the two children of node τ using recursive calls to the same algorithm. After these two smaller problems have completed, the Schur complements $F_{\nu_1;22}$ and $F_{\nu_2;22}$ corresponding the the

[2]For instance, Intel MKL and Cray LibSci behave this way.

Algorithm 2 GPU-accelerated multifrontal numerical factorization

Input: $A \in \mathbb{C}^{N \times N}$ in host memory; node τ of the assembly tree
Output: $LU \approx A$ in host memory

factor_GPU(A, τ):

1: **for** level ℓ from $\ell_{\max}$ to 0 **do**
2: $\quad M_\ell^F \leftarrow$ size of factors and piv vectors on level ℓ
3: $\quad M_\ell^S \leftarrow$ size of Schur and temp storage on level ℓ
4: **end for**
5: **if** $\max(M_\ell^F + M_\ell^S) >$ available device memory **then**
6: $\quad$ **for all** ν child of τ **do**
7: $\quad\quad$ **factor_GPU**(A, ν) ▷ recursive call
8: $\quad$ **end for**
9: $\quad F_\tau = [\ldots] \leftrightarrow\!\!\!\!\!\updownarrow F_{\nu_1;22} \leftrightarrow\!\!\!\!\!\updownarrow F_{\nu_2;22}$ ▷ F_τ on host
10: $\quad M^F \leftarrow \max(\text{size}(F_{11}, F_{12}), \text{size}(F_{12}, F_{21}, F_{22}))$
11: $\quad$ allocate_device(M^F)
12: $\quad$ copy_host_to_device(F_{11})
13: $\quad$ getrf_device$(F_{11}$, piv$)$ ▷ cuSOLVER
14: $\quad$ copy_device_to_host$(F_{11}$, piv$)$
15: $\quad$ **if** $\dim(F_{\tau;22}) \neq 0$ **then**
16: $\quad\quad$ copy_host_to_device(F_{12})
17: $\quad\quad$ getrs_device$(F_{11}$, piv, $F_{12})$ ▷ cuSOLVER
18: $\quad\quad$ delete_device$(F_{11}$, piv$)$
19: $\quad\quad$ copy_device_to_host(F_{12})
20: $\quad\quad$ copy_host_to_device(F_{21}, F_{22})
21: $\quad\quad$ gemm_device$(-1, F_{21}, F_{12}, 1, F_{22})$ ▷ cuBLAS
22: $\quad\quad$ copy_device_to_host(F_{22})
23: $\quad$ **end if**
24: $\quad$ **return**
25: **end if**
26: $dS_{\text{old}} \leftarrow$ `nullptr`
27: **for** level ℓ from $\ell_{\max}$ to 0 **do**
28: $\quad$ ▷ allocate factor and pivot vector memory
29: $\quad dF, piv \leftarrow$ allocate_device(M_ℓ^F)
30: $\quad$ ▷ allocate Schur and sparse matrix memory
31: $\quad dS, dA \leftarrow$ allocate_device(M_ℓ^S)
32: $\quad$ copy_host_to_device(dA) ▷ elems of A for assembly
33: $\quad$ front_assembly_device$(\ell, dA, dF, dS, dS_{\text{old}})$
34: $\quad$ swap(dS, dS_{old})
35: $\quad$ factor_small_fronts_device$< 8 >(\ell, dF, dS)$
36: $\quad$ factor_small_fronts_device$< 16 >(\ell, dF, dS)$
37: $\quad$ factor_small_fronts_device$< 24 >(\ell, dF, dS)$
38: $\quad$ factor_small_fronts_device$< 32 >(\ell, dF, dS)$
39: $\quad$ factor_large_fronts_device(ℓ, dF, dS)
40: $\quad$ synchronize_device()
41: $\quad$ copy_device_to_host$(dF$, piv$)$
42: **end for**
43: **if** $\dim(F_{\tau;22}) \neq 0$ **then** ▷ τ is not the root
44: $\quad$ copy_device_to_host$(F_{\tau;22})$
45: **end if**

two children of τ will be in host memory. Then, in Line 9, F_τ is assembled on the host, using $F_{\nu_1;22}$, $F_{\nu_2;22}$ and elements from the sparse matrix. This step is the so-called extend-add operation, denoted by the $\updownarrow$ symbol. Lines 12 to 22 perform the partial factorization of F_τ starting with F_τ in host memory. The copies to the device, as in Line 12, copy to memory allocated on the device on Line 11. This part of the code tries to minimize memory usage on the device, for instance by removing F_{11} from the device as soon as it is no longer needed there, or by not overlapping the transfer of F_{12} with the LU factorization of F_{11}. However, if there is not enough device memory to hold either $F_{\tau;11}$ and $F_{\tau;12}$ or $F_{\tau;12}$, $F_{\tau;21}$ and $F_{\tau;22}$, the algorithm will still fail. We plan to work around this restriction in a future implementation. However, in this case, one can simply use more GPUs, see Section 3.3. Moreover, if there is not enough device memory to hold a single front, it is quite likely that there will not be enough host memory to hold the rest of the factors either. Line 13 performs the LU factorization on the device and Line 14 copies both $F_{\tau;11}$, now containing its LU factors, and the corresponding pivot vector, back to the host. Lines 13, 17 and 21 use cuBLAS or cuSOLVER on NVIDIA and rocBLAS or rocSOLVER on AMD hardware.

3.1 GPU Sub-Tree Traversal

Lines 26 to 45 handle the case when the entire factorization of the assembly tree rooted at node τ fits in device memory. The assembly tree is traversed level by level. At every level, memory is allocated for the factors ($F_{\sigma;11}$, $F_{\sigma;12}$ and $F_{\sigma;21}$ blocks for all σ at level ℓ), for the pivot vectors, for the Schur complements (F_{22} blocks) and for the elements of the sparse matrix that are required for the assembly of the fronts at that level. Simultaneously, some memory is also allocated to hold the front metadata such as front sizes and pointers, but those details are omitted from Algorithm 2.

Then the frontal matrix assembly is performed on the device, Line 33, for all fronts on that level. Assembly of a frontal matrix requires elements from the sparse matrix, which are copied to dA on the device in Line 32, as well as the Schur complements, the F_{22} blocks, of the previous level, level $\ell + 1$. Assembly for all fronts on a level is done with a single CUDA/HIP kernel launch. After the assembly we keep dS in a separate pointer dS_{old} so the Schur complement at level ℓ can be used for the assembly of the next level, level $\ell - 1$. Next, partial factorization of all the fronts can start. The small fronts, i.e., fronts for which $\dim(F_{11}) \leq 32$, are treated differently then the larger ones. Section 3.2 discusses the custom CUDA/HIP device kernel we developed to handle the small fronts, called from Lines 35 to 38. For all the larger fronts, i.e., $\dim(F_{11}) > 32$, on level ℓ the code calls cuBLAS and cuSOLVER (or hipBLAS and rocSOLVER) directly using different streams. By default we use 4 stream, assigning fronts to streams in a round robin way. This is done in the `factor_large_fronts_device` routine (not shown in more detail), called on Line 39 of Algorithm 2.

After the GPU traversal of the assembly tree, the last Schur complement corresponding to node τ is copied back to host memory in Line 44. If τ is the root node of the tree, there is no Schur complement so this step can be skipped. Line 44 is only executed if τ is the root of a sub-tree, for instance when the **factor_GPU** routine was called recursively from Line 7, or when node τ is the root node of the sub-tree assigned to this GPU, or MPI rank, within a distributed memory instance of the solver, see Section 3.3.

Algorithm 2 has two memory allocations per level of the assembly tree, in Lines 29 and 31. The number of levels is typically small. For instance for a balanced nested dissection ordering the number of levels is $\mathcal{O}(\log N)$. However, allocating device memory seems to be a very costly operation, and we observed that for relatively small problems the calls to `cudaMalloc` can still be a serious bottleneck. We plan to work on reducing this overhead in the future, for instance relying on the memory management API from Umpire [19].

3.2 Kernels for Small Fronts

To deal with the small fronts, we developed a GPU kernel that performs partial LU factorization for all the fronts, on a given level, with sizes in a certain range. This kernel combines the LU factorization of the F_{11} block, the LU solve $F_{12} \leftarrow F_{11}^{-1} F_{12}$ and the Schur complement update $F_{22} \leftarrow F_{22} - F_{21} F_{12}$. The kernel is implemented to use a single $\text{NT} \times \text{NT}$ threadblock per front, where the block size NT is a template parameter. We instantiate this kernel for $\text{NT} = \{8, 16, 24, 32\}$, such that the number of threads in a block is at least 64 and at most 1024. Hence, at each level, four lists are constructed corresponding to the fronts satisfying: $\dim(F_{11}) \leq 8$, $8 < \dim(F_{11}) \leq 16$, $16 < \dim(F_{11}) \leq 24$, and $24 < \dim(F_{11}) \leq 32$. The kernels are launched with a block size $\text{NT} \times \text{NT}$ and with a one-dimensional grid, where the block index refers to the front. Front metadata, such as sizes and pointers to the F_{11}, F_{12}, F_{21}, and F_{22} blocks and the pivot vectors, are currently stored in an array of structures. To improve

memory access coalescing, this could in the future be changed to a single structure of arrays. All fronts for which $\dim(F_{11}) \leq 32$ are handled by this manual kernel, even though $\text{cols}(F_{12}) \equiv \dim(F_{22})$ could be significantly larger than 32. For the LU solve $F_{12} \leftarrow F_{11}^{-1} F_{12}$, the columns of F_{12} are blocked and loaded in shared memory in steps of NT columns. Likewise for the Schur update $F_{22} \leftarrow F_{22} - F_{21}F_{12}$, the F_{22} matrix is updated in blocks of size NT × NT in shared memory.

3.3 SLATE

For the GPU acceleration of the distributed memory fronts, we rely on SLATE [1] (Software for Linear Algebra Targeting Exascale), a modern replacement for ScaLAPACK. SLATE aims to achieve high performance and maximum scalability on modern HPC machines with large numbers of cores and multiple accelerators per node. ScaLAPACK does not provide any accelerator support. SLATE is a complete rewrite of ScaLAPACK with a modern templated C++ API using standards like MPI and OpenMP, and also relying on CUDA libraries for high performance on NVIDIA GPUs.

Like ScaLAPACK, SLATE uses the proven 2D block cyclic layout. However, the memory layout and processor distribution of blocks (tiles) is more flexible in SLATE. A SLATE matrix is stored tile per tile, instead of with a local column major layout. This allows SLATE to efficiently represent different matrix types, such as symmetric, banded or triangular. Moreover, it allows for different distributions and non-uniform block sizes.

In `STRUMPACK`, we use SLATE with the traditional 2D block cyclic data distribution, and column major storage of the local matrix, as used by ScaLAPACK. Because of this, the changes to the code to replace ScaLAPACK with SLATE are very minimal. However, to achieve reasonable performance with SLATE when using GPU acceleration, it is crucial to properly tune the block size. When running with SLATE we set the block size to 256, while for traditional CPU ScaLAPACK, the block size is set to 32. For SLATE, a larger block size drastically reduces overheads associated to GPU off-loading, whereas taking the block size too large would lead to poor load balance. For ScaLAPACK, the performance does not depend as much on the block size, since many local BLAS operations can be performed on the entire local matrix, or on row or column blocks of the matrix, instead of on individual tiles.

SLATE takes care of the distributed memory dense linear algebra operations on the distributed fronts. The extend-add operation, which passes the Schur complement F_{22} from a front to its parent front, is still performed on the CPU. The extend-add operation is negligible in terms of floating point operations compared to the dense linear algebra. However, for the local sub-tree GPU traversal it is crucial that the extend-add operation is performed on the GPU in order to keep the fronts in device memory and avoid data movement. For the distributed code, the extend-add is performed using an `MPI_Alltoallv` call, which is performed from the CPU. By performing the distributed memory extend-add on the CPU we can also accommodate large distributed fronts that do not fit entirely in GPU memory.

3.4 Pivoting Issues

We developed a GPU kernel to replace small pivot elements on the diagonal of the U factors of the LU factorization of F_{11} by larger threshold values to prevent unsafe division. This kernel is called just after the LU factorization and right before the solve $F_{12} \leftarrow F_{11}^{-1} F_{12}$. The LAPACK routine `xgetrf` returns the column index of the first zero pivot, but it will still complete the entire LU factorization. However, the cuSOLVER routine `cusolverDnXgetrf` aborts as soon as it encounters a zero pivot. Hence, the result is useless for our use case where we still want to apply a slightly modified factorization as a preconditioner. We have reported this issue to NVIDIA. Likewise, the routine `magma_xgetrf_native` in MAGMA 2.5.4 fails due to an illegal memory access when it encounters a zero pivot.

3.5 Triangular Solve

After numerical factorization the sparse lower triangular factor is stored in the F_{21} blocks and in the lower triangular part of the F_{11} blocks. Likewise, the sparse upper triangular factor is stored in the upper triangular part of F_{11} and in F_{12}. We also ported the multifrontal forward (L) and backward (U) triangular solve phases to GPU, using a similar strategy as in Algorithm 2. However, where the performance of the multifrontal factorization is bound by floating point operations, the triangular solve is memory bandwidth bound, at least for a single right-hand side. For multiple right-hand sides the arithmetic intensity of the solve increases. If the factors are initially in host memory, and one solves a linear system with a single right-hand side, there is no potential gain from execution on the GPU since copying the factors to the GPU would take about as long as performing the solve. However, there are use cases in which the

sparse solver is used repeatedly or with multiple right-hand sides. Hence, we also provide routines to copy the sparse factors to device memory and remove them from the device again. This way the cost of copying the factors to device memory can be amortized over multiple solves. If the factors do not fit in device memory, the code still works correctly, but the factors have to be moved to device memory for every solve.

3.6 Porting to AMD Hardware

We originally developed the GPU code in `STRUMPACK` on NVIDIA hardware using CUDA and the CUDA Toolkit libraries cuBLAS, cuSOLVER. For AMD hardware the CUDA code was 'hipified'. A tool[3] is available for converting CUDA code to HIP code. However, the process is typically quite straightforward, modifying the kernel launch syntax, and simply changing calls like `cudaDeviceSynchronize()` and `cudaMalloc()` to `hipDeviceSynchronize()` and `hipMalloc()` for instance. The `hipcc` compiler can target both AMD and NVIDIA hardware, and the HIP library hipBLAS is a thin marshalling layer that supports both cuBLAS for NVIDIA and rocBLAS for AMD as backends. Hence, in theory one could use HIP to target both NVIDIA and AMD. However, since at the moment there is no hipSOLVER marshalling library (this would be a thin layer on top of cuSOLVER and rocSOLVER), and because the HIP CMake support is still under active development, we are keeping both the CUDA and HIP code paths in `STRUMPACK`.

`STRUMPACK` also supports complex arithmetic, relying on the C++ `std::complex<T>` datatype. However, standard C++ complex number arithmetic is not supported in CUDA, so these numbers are reinterpreted as CUDA `thrust::complex<T>` numbers. The problem with `thrust::complex<T>` numbers is that they do not have a default constructors so they cannot be used in CUDA shared memory. Therefore we need one more reinterpration cast between `thrust::complex<float>` and CUDAs builtin `float2` (or `double2`). On AMD hardware we use rocThrust.

4 Experimental Evaluation

We start this section with a brief description of the test setup in Section 4.1. The tests in Section 4.2 and Section 4.3 show single GPU results for regular mesh PDE problems and test matrices from the SuiteSparse collection respectively. In Section 4.4, we look at the performance of the multi-GPU code. It should be noted that in all experiments, the timing for numerical factorization includes all data transfers from and to the device. The factorization starts with the sparse matrix in host memory and ends with the sparse triangular factors in host memory.

4.1 Test Setup

We consider four different test systems, with four different GPUs. The first system is the SUMMIT cluster[4], a large scale HPC cluster operated by the Oak Ridge Leadership Computing Facility, and which used to be the number one ranked system on the TOP500 in 2018 and 2019. The entire SUMMIT machine has 4,608 nodes, with a combined peak performance of over 200TFlop/s. Every compute node has two 21-core POWER9 CPUs, 42 cores per node, 512GB of DDR4 memory and 6 NVIDIA Volta V100 GPUs. The second system is a typical workstation or gaming rig, equipped with a 16-core AMD Ryzen 9 3950X, 128GB DDR4 and an NVIDIA GeForce RTX 2060 SUPER[5]. The final system is a small cluster operated by HPE with a handful of nodes. Some of the nodes have an AMD EPYC 7601 32-core processor, 256GB memory and four MI60 AMD GPUs each. This HPE system also has several nodes each equipped with two 64-core AMD ROME CPUs and four AMD Instinct MI100 GPUs. Since this is an early access system used to mimic the setup of the future Frontier system – scheduled to be installed at the Oak Ridge Leadership Computing Facility in 2021 – all four of the AMD Instinct MI100 GPUs are connected to the second CPU. Therefore, we only use this second CPU and pin all threads to the second socket, but still allow processes to use memory connected to the first socket.

On SUMMIT we use GCC 9.1.0 and CUDA 11.0.221. On the desktop with the RTX2060, we use Ubuntu 20.04, with GCC 9.3.0 and CUDA 10.1.243. On the MI60 equipped nodes, we use GCC 9.2.0 and ROCm 3.8.0 with `hipcc` based on Clang 11.0.0. Table 1 lists the different accelerators or GPUs we use for the experiments, along with some key characteristics, such as their power usage, number of compute units, theoretical peak performance etc.

For BLAS and LAPACK, we use OpenBLAS 0.3.10, currently the latest released stable version, compiled with `make USE_OPENMP=1`. We use OpenBLAS on all test systems, even if on the POWER9

[3] https://github.com/ROCm-Developer-Tools/HIPIFY

[4] https://www.olcf.ornl.gov/summit/

[5] https://www.techpowerup.com/gpu-specs/geforce-rtx-2060-super.c3441

GPU	Memory	Power	Freq	SMs/CUs	SP Peak	DP Peak	Arch	Vendor
V100	16GB	300W	1530MHz	80	15.7 TFlop/s	7.8 TFLOP/s	Volta	NVIDIA
RTX2060-S	8GB	175W	1470MHz	34	7.181 TFlop/s	224.4 GFlop/s	Turing	NVIDIA
MI60	32GB	225W	1294MHz	64	12 TFlop/s	5.3 TFlop/s	GCN5	AMD
MI100	32GB	300W	1054MHz	120	32/15 TFlop/s	8 TFlop/s	CDNA	AMD

Table 1: Graphics processing units (GPUs) used in the experiments. NVIDIA count Streaming Multiprocessor (SMs), while AMD refers to them as Compute Units (CUs). The single precision floating point achievable peak performance for the AMD Instinct MI100 is 32TFlop/s for GEMM, 15 for more general workloads not using instructions optimized for GEMM specifically.

system one might recommend IBM's optimized ESSL library. The main issue for our use case is that ESSL provides separate sequential and multithreaded libraries. `STRUMPACK` relies on the behavior as implemented in for instance MKL, OpenBLAS and Cray LibSci, where the BLAS and LAPACK routines run sequentially when called from within an OpenMP parallel region, and exploit multiple threads otherwise.

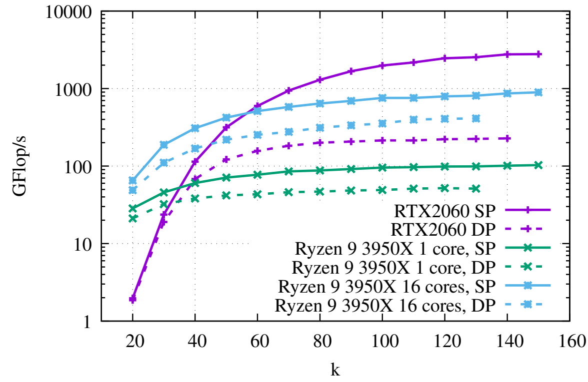
We construct an exact solution vector/matrix $\hat{x}$ with entries sampled from a normal distribution with mean zero and standard deviation 1. The corresponding right hand side is computed as $b = A\hat{x}$. We then solve $Ax = b$ for x, which is always compared to the exact solution $\hat{x}$ to verify correctness.

4.2 Single GPU, PDE Problems on Regular Grids

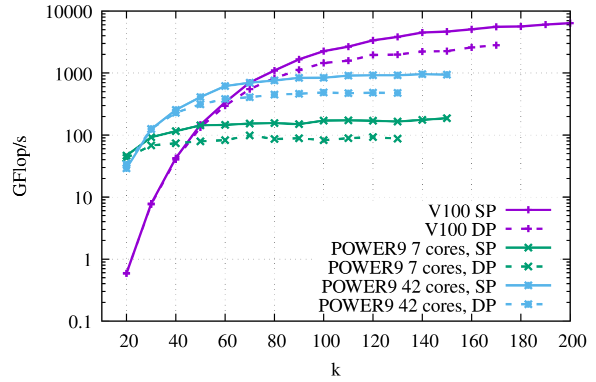
We first consider the discretization of the Poisson equation $\nabla^2 u = f$ on a cubic $N = k^3$ domain using a second order 7-point finite difference stencil and Dirichlet boundary conditions. This leads to a linear system with N rows and approximately $7N$ nonzeros. Since the problem is defined on a regular mesh, we can easily perform a nested dissection fill-reducing reordering of the matrix without having to call a library such as METIS. Figure 2 shows the performance achieved during the numerical factorization on the four different hardware platforms. The number of mesh points per dimension is incremented from $k = 20$, in steps of 10, until the system runs out of memory or until the job exceeds the time limit set by the queue policy. The results are presented for both single (SP) and double precision (DP). Figures 2a to 2d show the results for the RTX2060, V100, MI60 and MI100 systems respectively. These figures also compare with the respective CPUs. For the RTX2060 system, which has a single GPU, we compare with a single CPU core and with all 16 CPU cores utilized using OpenMP (a single MPI process is used). For the V100 system, which is part of a SUMMIT node with two 21-core POWER9 processors and 6 V100 GPUs, we compare with 7 POWER9 cores – since there is one V100 GPU for every 7 POWER9 CPU cores – and with all 42 POWER9 cores. Likewise, for the MI60 system, we compare with 8 and 32 cores of the EPYC 7601 CPU.

Using single precision, the numerical factorization reaches 43% of the peak performance on the RTX2060, 41% on the V100, 24% on the MI60 and 26% on the MI100. For the MI60, MI100 and V100, the difference in performance between the single and double precision is very close to a factor two. For the RTX2060 however, theoretically the difference is 1: 32, although we only observed a factor $\sim 12\times$. Note that for smaller problems, the cheaper RTX2060 outperforms the other cards, possibly because this one is optimized for low latency – important in games – whereas the others are targeting HPC workloads and are optimized for throughput rather than latency. Finally, Fig. 3a compares the different GPUs directly. We should note that at the time of writing, the AMD Instinct MI100 card has not yet been officially released and we still expect the software – the ROCm libraries and our `STRUMPACK` solver – to mature further and eventually achieve significantly higher performance on this card. The AMD Instinct MI100 has specific operations to accelerate matrix multiplication. Currently, our kernels for small matrices do not exploit these SGEMM operations yet. However, the small matrix kernels are not the main bottleneck in the code, as can be seen from Fig. 4a.

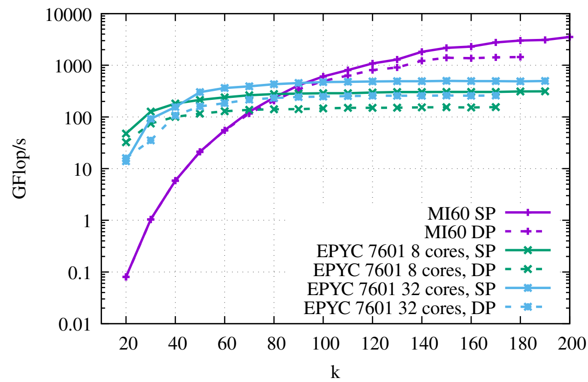
Figure 3b illustrates the same experiment as Fig. 2a, but this time the wall clock time is shown. Also here it is clear that there is a large overhead for smaller problems for the GPU code. For larger problems, the asymptotic scaling of $\mathcal{O}(N^2)$ becomes clear, and the mid-range $\sim$ 400$/175W RTX2060 SUPER gaming GPU outperforms even the 16-core Ryzen 9 3950X processor ($\sim$ 700$/105W), at least for single precision. Figure 4a takes a closer look at the 100^3 Poisson problem, in single precision, which is the largest one for which the factorization fits entirely



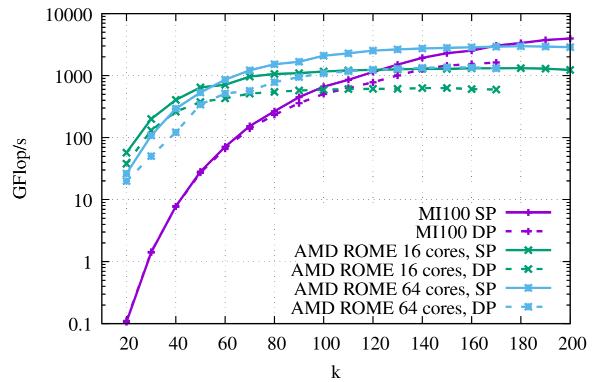
(a) NVIDIA RTX2060



(b) NVIDIA V100

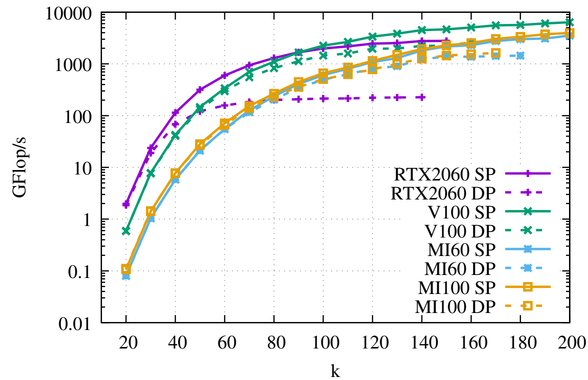


(c) AMD Instinct MI60

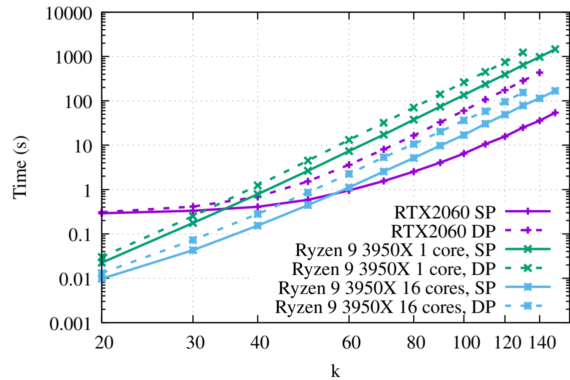


(d) AMD Instinct MI100

Figure 2: Achieved performance in GFlop/s for the numerical factorization phase for sparse linear systems derived from the discretization of the 3D Poisson equation on a regular k^3 mesh, for different hardware systems.



(a) RTX2060 vs V100 vs MI60 vs MI100



(b) Scaling of factorization time with k

Figure 3: Figure 3a compares the different GPU results from Fig. 2 directly. Figure 3b shows the $\mathcal{O}(N^2)$ scaling of the factorization time with $N = k^3$.

in GPU memory. Figure 4a shows the time spent on each level of the assembly tree, the floating point rate achieved at each level (axis on the right), and the overall average floating point performance for the entire numerical factorization. On level 15 for instance, there are 32768 nodes in the assembly tree (fronts), of which $32,643$ have a top-left block smaller than 8×8, and 125 have the top-left block smaller than 16×16. From level 10, which has 1024 nodes, and upwards towards the root at level 0, all fronts are larger than 32×32. Hence all nodes on level $10 - 0$ are handled by cuBLAS/cuSOLVER directly. At the root level there is a single front of size $10,000 \times 10,000$. Level 1 has two nodes, level 2 has 4 nodes and so on. It is clear that the reasonably good performance of the solver is achieved by the use of the GPU for the fewer, much larger, frontal matrices near the root of the assembly tree. The experiment shown in Fig. 4a was performed on the RTX2060, which has a single precision peak performance of 6.451TFlop/s. At level 1 of the assembly tree, we achieve 3.418TFlop/s, which is 53% of the peak performance.

4.3 Single GPU Results for SuiteSparse Test Matrices

Table 2 lists a number of selected matrices from the online SuiteSparse matrix collection [20]. The table shows the number of rows/columns in the matrices, the number of nonzeros, the type – SPD for symmetric positive definite, sym for symmetric, non-sym for non-symmetric – and a brief description of the application domain. We solve a linear system with each of these matrices, with the results shown in Table 3. For these tests, we run single precision on the RTX2060-SUPER, and double precision on the other cards. We do this simply because the double precision performance of the RTX2060 is so poor. For the RTX2060, we compare with all 16 cores of the CPU. For the two other systems, we compare with the number of CPU cores that are available per GPU, i.e., 7 POWER9 cores per V100, 8 AMD EPYC 7601 cores per MI60, and 16 AMD ROME cores per MI100. The main takeaway from Table 3 is again that the GPU code performs well compared to the CPU code as long as the problem is large enough. For the smaller problems, the CPU outperforms the GPU code. There is too much overhead for the smaller problems from GPU memory allocation, copies between host and device, kernel launch latency, and synchronization. For larger problems, or more precisely, for problems for which there is a lot of fill-in in the sparse triangular factors, such as cage13 and nlpkkt80, speed-up from the GPU code over the CPU code is considerable. Compared to SuperLU, on the V100 our implementation is faster for all but one matrix, and is on average $5\times$ faster (median speedup is $4\times$). On the POWER9, SuperLU is about $5\times$ slower, but with the median slowdown only $1.57\times$.

All the tests in Table 3 used the nested dissection fill-reducing ordering from METIS 5.1.0 through the `METIS_NodeNDP` routine rather than `METIS_NodeND`. `METIS_NodeNDP` is an undocumented routine that returns the permutation vector as well as the separator tree information. In our experience the assembly tree from `METIS_NodeNDP` is often better balanced than the tree derived from the permutation returned by `METIS_NodeND`. For the SuperLU_dist results in Table 3, we used the 3D factorization algorithm using the `pdgssvx3d` driver, with a single MPI process. The 3D algorithm off-loads more operations to the GPU than the default driver `pdgssvx`. After consulting the SuperLU developers, we settled on the following tuning parameters: NREL=100, NSUP=256, NUM_CUDA_STREAMS=1, MAX_BUFFER_SIZE=500000000, N_GEMM=100, LOOKAHEAD=2. However, in Table 3 we report the minimum of the times obtained with either these or the default parameters.

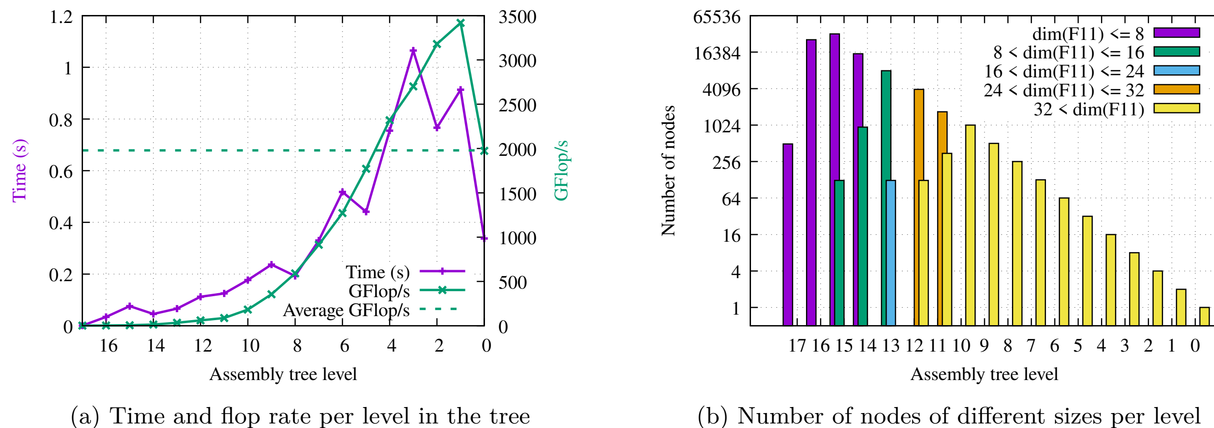
Figure 5 attempts to give more insight into the performance of the solver, on both GPU (the RTX2060) and CPU (16 core Ryzen) for the problems from Table 3. This shows that performance, in terms of floating point operations per second improves as the problem size increases. Here we measure the problem size as the total number of floating point operations required during the numerical factorization (horizontal axis on Fig. 5). The GPU only outperforms the CPU (in single precision) when the problem requires $\gtrsim 5 \cdot 10^{13}$ floating point operations. This also shows that, at least for single precision, the factorization gets very close to the theoretical peak on the RTX2060.

4.4 Multi-GPU Using SLATE

We now solve the visco-acoustic wave propagation problem as described by the Helmholtz equation

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

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. (2) by a finite-difference discretization on stag-



(a) Time and flop rate per level in the tree

(b) Number of nodes of different sizes per level

Figure 4: A closer look at the 100^3 problem on the RTX2060. Figure 4a shows the time spent on each level of the assembly tree as well as the floating point rate achieved on each level and the overall average achieved floating point performance for the whole sparse factorization. Figure 4b shows the number of nodes on each level, split into different sizes.

name	N	nnz	type	origin
atmosmodd	1,270,432	8,814,880	non-sym	Computational Fluid Dynamics Problem
c-73	169,422	1,279,274	sym	Optimization Problem Sequence
cage13	445,315	7,479,343	non-sym	Directed Weighted Graph
Freescale1	3,428,755	17,052,626	non-sym	Circuit Simulation Problem
Geo_1438	1,437,960	60,236,322	SPD	Structural Problem
Hook_1498	1,498,023	59,374,451	SPD	Structural Problem
memchip	2,707,524	13,343,948	non-sym	Circuit Simulation Problem
ML_Geer	1,504,002	110,686,677	non-sym	Structural Problem
nlpkkt80	1,062,400	28,192,672	sym	Optimization Problem
ohne2	181,343	6,869,939	non-sym	Semiconductor Device Problem
PFlow_742	742,793	37,138,461	SPD	2D/3D Problem
pwtk	217,918	11,524,432	SPD	Structural Problem
scircuit	170,998	958,936	non-sym	Circuit Simulation Problem
Serena	1,391,349	64,131,971	SPD	Structural Problem
torso3	259,156	4,429,042	non-sym	2D/3D Problem
Transport	1,602,111	23,487,281	non-sym	Structural Problem
xenon2	157,464	3,866,688	non-sym	Materials Problem

Table 2: Set of test matrices, taken from the SuiteSparse matrix collection.

	STRUMPACK								SuperLU	
		Ryzen		EPYC		ROME		POWER9		POWER9
matrix	RTX2060	16 core	MI60	8 core	MI100	16 core	V100	7 core	V100	7 core
atmosmodd	**6.58**	17.41	**24.48**	85.87	23.09	**21.76**	**8.40**	141.82	**85.17**	217.77
c-73	2.08	**0.24**	10.61	**0.68**	8.23	**0.36**	5.10	**1.66**	**2.81**	4.06
cage13	**30.05**	116.99	**53.10**	672.59	**40.70**	160.55	**34.13**	1123.67	-	-
Freescale1	1.75	**0.39**	17.35	**1.03**	14.14	**0.73**	3.91	**1.00**	**12.78**	15.03
Geo_1438	**17.81**	48.00	**54.31**	250.50	**52.83**	61.01	**21.41**	414.18	**165.12**	440.57
Hook_1498	**10.89**	24.25	**39.56**	121.76	39.19	**29.61**	**13.72**	254.76	**101.67**	237.44
memchip	1.52	**0.37**	12.64	**0.88**	11.02	**0.58**	6.58	**1.07**	**11.05**	46.70
ML_Geer	7.70	**7.06**	36.65	**30.90**	38.42	**8.06**	**10.15**	50.20	**50.46**	78.98
nlpkkt80	**14.64**	42.23	**40.42**	210.27	**39.19**	52.57	**16.66**	344.12	**160.34**	435.15
ohne2	1.13	**0.70**	10.75	**2.77**	9.01	**0.74**	**2.13**	4.00	**5.90**	7.00
PFlow_742	**4.31**	5.01	23.31	**22.25**	21.68	**5.76**	**6.32**	37.34	**24.33**	42.71
pwtk	0.77	**0.18**	9.09	**0.56**	7.27	**0.21**	1.61	**0.87**	3.68	**2.11**
scircuit	0.36	**0.03**	7.47	**0.04**	5.43	**0.04**	1.13	**0.05**	1.26	**1.15**
Serena	**23.80**	75.51	**59.32**	409.88	**54.37**	98.64	**27.59**	662.93	**260.61**	759.34
torso3	1.01	**0.79**	10.81	**3.12**	8.58	**0.93**	**2.30**	5.09	**9.76**	12.14
Transport	**8.06**	15.95	**35.25**	72.71	35.67	**18.21**	**10.10**	118.90	**78.71**	160.72
xenon2	0.65	**0.31**	8.81	**0.95**	6.77	**0.31**	**1.41**	1.42	**3.89**	-

Table 3: Time (in seconds) for numerical factorization for each of the matrices listed in Table 2. The RTX2060 results are for single precision, while the V100, MI60 and MI100 results are for double precision. For each matrix and each system, the faster time is emphasized. For SuperLU, there were failures for xenon2 and cage13 which need further investigation.

Figure 5: Performance of the factorization in terms of floating point operations per second for the matrices from Table 2 and Table 3, using the total number of floating point operations on the horizontal axis.

gered grids using a 27-point stencil and 8 PML absorbing boundary layers [21], which requires the solution of a complex sparse linear system where each matrix row contains 27 nonzeros whose values depend on the coefficients and frequency in Eq. (2). We set $v(\mathbf{x}) = 4000\text{m/s}$, $\rho(\mathbf{x}) = 1\text{kg/m}^3$, $q(\mathbf{x}) = 10^4$. The frequency is set to $\omega = 8\pi\text{Hz}$ and the grid spacing is set such that there are 15 grid points per wavelength. This is a high frequency wave propagation problem and is extremely hard to solve with iterative methods. Direct solvers are usually required for this type of problem, although domain specific preconditioners exist – see also the discussion in [22] where we present a preconditioner based on approximate multifrontal factorization.

Table 4 shows the timings for the numerical factorization of the linear system resulting from the discretization of Eq. (2) for $k = \{100, 150, 200, 250\}$ grid points per dimension on a regular 3-dimensional cubic domain. The distributed memory fronts are handled by ScaLAPACK (with block size 32) for the CPU only tests, and by SLATE (block size 256) for the GPU accelerated tests. For the $k = 100, 150, 200$ CPU tests, we use flat MPI, ie., one MPI rank per core, since this gives the best performance. However, the $k = 250$ test with flat MPI on 8 nodes ran out of memory. On 4 nodes of SUMMIT the code runs $\sim 10\times$ faster

when using all 24 V100 GPUs compared to when it only uses the 168 POWER9 cores. On 8 SUMMIT nodes, using 48 V100 GPUs, the sparse solver reaches over 50TFlop/s.

All distributed memory tests are performed on the SUMMIT system, with POWER9 CPUs and V100 GPUs. These tests use ESSL instead of OpenBLAS for BLAS and LAPACK due to some difficulties in building SLATE with OpenBLAS, and because we prefer to use IBM's optimized ScaLAPACK library which links to ESSL. The ESSL threading issue mentioned earlier is irrelevant for the ScaLAPACK results shown here since those use flat MPI, i.e., one MPI rank per CPU core.

5 Getting the Code

We developed a GPU-accelerated sparse multifrontal solver as part of the `STRUMPACK` software library, which is available with a 3-clause BSD licence from `github.com/pghysels/strumpack`. Up-to-date documentation, including configuration and installation instructions, as well as several recent publications can be found at the `STRUMPACK` website: `portal.nersc.gov/project/sparse/strumpack/master/`. Configuration of `STRUMPACK` is done using modern CMake, with exported targets. Modern CMake has excellent support for CUDA, treating it as a language just as it does with C/C++ and Fortran. With CMake version 3.18, the support for HIP is not at the same level yet. Dependencies required to build `STRUMPACK` are C++11, C and Fortran compilers, BLAS, LAPACK, and METIS [15]. MPI is optional, but when it is enabled, then ScaLAPACK is required. Other optional dependencies are OpenMP ($\geq$ 3.1), ZFP [23], CUDA[6] (including cuBLAS and cuSOLVER), HIP[7] (including hipBLAS, rocBLAS, rocSOLVER and rocThrust), MAGMA and Scotch [16]. Furthermore, if MPI is enabled, then also ParMETIS [24], PTScotch [25], ButterflyPACK[8] [26], SLATE[9], and Combinatorial BLAS [27] offer additional optional functionality.

Given all these dependencies, it can be very challenging to properly configure and build the code on a local desktop or on a large scale HPC cluster. To ease the installation process, we developed a `spack` installation script. Spack [28] is a package manager specifically targeting HPC software, and is funded by the ECP project. Moreover, `STRUMPACK` is part of the Extreme-scale Scientific Software Development Kit (xSDK) [29], and the Extreme-scale Scientific Software Stack (E4S). Both are ecosystems of reusable scientific software and can be installed through `spack`. New releases of xSDK thoroughly test compilation of all its packages on a variety of systems, as well as verify compatibility between the different packages. For instance the solvers in `STRUMPACK` can be called directly or through interfaces from PETSc [30], MFEM [31] or Trilinos [32], which are all also included in the xSDK.

[6] `https://developer.nvidia.com/cuda-zone`
[7] `https://github.com/ROCm-Developer-Tools/HIP`
[8] `https://github.com/liuyangzhuan/ButterflyPACK`
[9] `https://icl.utk.edu/slate/`

6 Conclusion and Outlook

We have ported the `STRUMPACK` sparse direct multifrontal solver to GPU, targeting both NVIDIA and AMD hardware. By off-loading numerical factorization for larger sub-trees of the multifrontal assembly tree that fit entirely in GPU memory our algorithm avoids a lot of data movement, and a lot of kernel launch overhead. We use a combination of CUDA kernels and cuBLAS/cuSOLVER calls to achieve good performance. The code is also ported to AMD hardware, using HIP and rocBLAS, rocSOLVER. For the distributed memory and multi-GPU setting the code depends on the SLATE library, a modern GPU-aware replacement for ScaLAPACK. On 4 nodes of SUMMIT the code runs $\sim 10\times$ faster when using all 24 V100 GPUs compared to when it only uses the 168 POWER9 cores. On 8 SUMMIT nodes, using 48 V100 GPUs, the sparse solver reaches over 50TFlop/s.

However, bottlenecks still remain which become especially apparent for the factorization of smaller linear systems. Since we have observed that the (de-)allocation of memory on the device can be a bottleneck for small systems, we plan to improve the memory management in the solver [19]. Another possible area of improvement is to use more tricks developed specifically for batched dense matrix operations, as for instance in [33, 2, 34]. Moreover, since we rely on several third party libraries, we can expect performance improvements just by using newer versions of these libraries. For instance, the SLATE project is relatively new and has initially focused mostly on feature completeness. We are hopeful that SLATE performance will improve with future releases. Likewise, we presented results for the AMD Instinct MI100, which at the time of writing is not officially released yet. We expect the performance of rocBLAS and rocSOLVER to improve with coming releases of the ROCm software stack.

`STRUMPACK` aims to provide a modern collection

	100^3		150^3		200^3		250^3	
	P9	V100	P9	V100	P9	V100	P9	V100
SUMMIT nodes	1		2		4		8	
P9 cores	42		84		168		336	
# of V100s	-	6	-	12	-	24	-	48
MPI ranks	42	6	84	12	168	24	336	48
OpenMP/rank	1	7	1	7	1	7	1	7
Fact. time (sec)	105.58	18.63	581.45	61.29	1612.67	157.67	OOM	266.96
Speedup	**5.7**×		**9.5**×		**10.2**×		-	
TFlop/s	0.51	2.86	1.06	10.09	2.17	22.14	-	50.18
Flops	$5.3 \cdot 10^{13}$		$6.2 \cdot 10^{14}$		$3.5 \cdot 10^{15}$		$1.3 \cdot 10^{16}$	
Fact. mem. (GB)	35.25		185.89		598.77		1482.38	

Table 4: Statistics for the numerical factorization phase of complex linear systems from the discretization of the 3D Helmholtz equation, a difficult problem for (unpreconditioned) iterative solvers such as (restarted) GMRES and BiCGStab, and many preconditioners. On 4 SUMMIT nodes, and using all 24 available V100 GPUs, the code runs $\sim 10\times$ faster compared to when only the POWER9 CPU cores are used. The 250^3 CPU test ran out of memory.

of dense and sparse solvers and preconditioners relying on rank-structured matrix formats. Rank-structured or data-sparse matrix representations exploit on the presence of matrix sub-blocks that are of low rank. Such blocks appear for instance in the off-diagonal blocks of large dense matrices from the discretization of integral equations using the boundary element method. In sparse multifrontal solvers – as has been the focus of this article – low rank blocks are also found in the off-diagonal parts of the frontal matrices arising in the multifrontal solution of sparse systems coming from the discretization of many partial differential equations. Hence, `STRUMPACK` provides several rank-structured approximate multifrontal solvers/preconditioners. The different rank-structured matrix formats currently supported in `STRUMPACK` include hierarchically semi-separable (HSS), hierarchically off-diagonal low rank (HODLR), butterfly, hierarchically off-diagonal butterfly (HODBF) [22], and block low rank (BLR). Alternatively, `STRUMPACK` also provides lossless or lossy compression of the sparse triangular factors using the ZFP library, which provides compression algorithms specifically designed for floating point data. However, currently none of the rank-structured solvers or preconditioners available in `STRUMPACK` run on GPUs. Porting these advanced solvers to modern accelerators is a major task that we have only just started to tackle.

7 Acknowledgements

This research was supported in part 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, and in part by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program through the FASTMath Institute under Contract No. DE-AC02-05CH11231 at Lawrence Berkeley National Laboratory.

References

[1] M. Gates, J. Kurzak, A. Charara, A. YarKhan, J. Dongarra, 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, 2019, pp. 1–18.

[2] A. Abdelfattah, A. Haidar, S. Tomov, J. Dongarra, Novel HPC techniques to batch execution of many variable size BLAS computations on GPUs, in: Proceedings of the International Conference on Supercomputing, 2017, pp. 1–10.

[3] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, J. Koster, MUMPS: a general purpose distributed memory sparse solver, in: International Workshop on Applied Parallel Computing, Springer, 2000, pp. 121–130.

[4] T. A. Davis, Algorithm 832: UMFPACK V4. 3—an unsymmetric-pattern multifrontal method, ACM Trans. Math. Softw. 30 (2) (2004) 196–199.

[5] A. Gupta, WSMP: Watson sparse matrix package (Part-I: direct solution of symmetric sparse systems), IBM TJ Watson Research Center, Yorktown Heights, NY, Tech. Rep. RC 21886 (2000).

[6] A. Gupta, WSMP: Watson Sparse Matrix Package (Part-II: direct solution of general sparse systems), Tech. rep., Citeseer (2000).

[7] I. S. Duff, J. K. Reid, The multifrontal solution of indefinite sparse symmetric linear, ACM Trans. Math. Softw. 9 (3) (1983) 302–325.

[8] T. A. Davis, S. Rajamanickam, W. M. Sid-Lakhdar, A survey of direct methods for sparse linear systems, Acta Numer. 25 (2016) 383–566. doi:10.1017/S0962492916000076.

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

[10] X. S. Li, J. W. Demmel, Superlu_dist: A scalable distributed-memory sparse direct solver for unsymmetric linear systems, ACM Transactions on Mathematical Software (TOMS) 29 (2) (2003) 110–140.

[11] P. Sao, R. Vuduc, X. S. Li, A distributed CPU-GPU sparse direct solver, in: European Conference on Parallel Processing, Springer, 2014, pp. 487–498.

[12] X. Lacoste, Scheduling and memory optimizations for sparse direct solver on multi-core/multi-gpu duster systems, Ph.D. thesis (2015).

[13] I. S. Duff, J. Koster, The design and use of algorithms for permuting large entries to the diagonal of sparse matrices, SIAM J MATRIX ANAL A. 20 (4) (1999) 889–901.

[14] A. Azad, A. Buluc, X. S. Li, X. Wang, J. Langguth, A distributed-memory algorithm for computing a heavy-weight perfect matching on bipartite graphs, SIAM J. Scientific Computing (2020 (to appear)).

[15] G. Karypis, V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, SIAM J. Sci. Comput. 20 (1) (1998) 359–392.

[16] F. Pellegrini, J. Roman, Sparse matrix ordering with scotch, in: International Conference on High-Performance Computing and Networking, Springer, 1997, pp. 370–378.

[17] J. Choi, J. J. Dongarra, R. Pozo, D. W. Walker, Scalapack: A scalable linear algebra library for distributed memory concurrent computers, in: The Fourth Symposium on the Frontiers of Massively Parallel Computation, IEEE Computer Society, 1992, pp. 120–121.

[18] P. Ghysels, X. S. Li, F.-H. Rouet, S. Williams, A. Napov, An efficient multicore implementation of a novel HSS-structured multifrontal solver using randomized sampling, SIAM J. Sci. Comput. 38 (5) (2016) S358–S384.

[19] D. A. Beckingsale, M. J. Mcfadden, J. P. Dahm, R. Pankajakshan, R. D. Hornung, Umpire: Application-focused management and coordination of complex hierarchical memory, IBM Journal of Research and Development 64 (3/4) (2019) 00–1.

[20] S. P. Kolodziej, M. Aznaveh, M. Bullock, J. David, T. A. Davis, M. Henderson, Y. Hu, R. Sandstrom, The Suitesparse matrix collection website interface, Journal of Open Source Software 4 (35) (2019) 1244.

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

[22] Y. Liu, P. Ghysels, L. Claus, X. S. Li, Sparse Approximate Multifrontal Factorization with Butterfly Compression for High Frequency Wave Equations, arXiv preprint arXiv:2007.00202 (2020).

[23] P. Lindstrom, Fixed-rate compressed floating-point arrays, IEEE transactions on visualization and computer graphics 20 (12) (2014) 2674–2683.

[24] G. Karypis, V. Kumar, A parallel algorithm for multilevel graph partitioning and sparse matrix ordering, J PARALLEL DISTR COM 48 (1) (1998) 71–95.

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

[26] H. Guo, Y. Liu, J. Hu, E. Michielssen, A butterfly-based direct integral-equation solver using hierarchical LU factorization for analyzing scattering from electrically large conducting objects, IEEE Trans. Antennas Propag. 65 (9) (2017) 4742–4750.

[27] A. Buluç, J. R. Gilbert, The Combinatorial BLAS: Design, Implementation, and Applications, Int. J. High Perform. C. 25 (4) (2011) 496–509.

[28] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, S. Futral, The Spack package manager: bringing order to HPC software chaos, in: SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2015, pp. 1–12.

[29] R. Bartlett, I. Demeshko, T. Gamblin, G. Hammond, M. Heroux, J. Johnson, A. Klinvex, X. Li, L. C. McInnes, J. D. Moulton, et al., xSDK foundations: Toward an extreme-scale scientific software development kit, arXiv preprint arXiv:1702.08425 (2017).

[30] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W. D. Gropp, D. Karpeyev, D. Kaushik, M. G. Knepley, D. A. May, L. C. McInnes, R. T. Mills, T. Munson, K. Rupp, P. Sanan, B. F. Smith, S. Zampini, H. Zhang, H. Zhang, PETSc Web page, `https://www.mcs.anl.gov/petsc` (2019).
URL `https://www.mcs.anl.gov/petsc`

[31] R. Anderson, J. Andrej, A. Barker, J. Bramwell, J.-S. Camier, J. Cerveny, V. Dobrev, Y. Dudouit, A. Fisher, T. Kolev, et al., MFEM: a modular finite element methods library, arXiv preprint arXiv:1911.09220 (2019).

[32] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, et al., An overview of the trilinos project, ACM Transactions on Mathematical Software (TOMS) 31 (3) (2005) 397–423.

[33] A. Abdelfattah, A. Haidar, S. Tomov, J. Dongarra, Performance, design, and autotuning of batched gemm for gpus, in: International Conference on High Performance Computing, Springer, 2016, pp. 21–38.

[34] A. Charara, D. Keyes, H. Ltaief, Batched triangular dense linear algebra kernels for very small matrix sizes on GPUs, ACM Transactions on Mathematical Software (TOMS) 45 (2) (2019) 1–28.