SAND2020-6291C

Performance Portable Supernode-based Sparse Triangular Solver for Manycore Architectures

Ichitaro Yamazaki, Sivasankaran Rajamanickam, and Nathan David Ellingwood
Sandia National Laboratories, Albuquerque, New Mexico, USA

ABSTRACT

Sparse triangular solver is an important kernel in many computational applications. However, a fast, parallel, sparse triangular solver on a manycore architecture such as GPU has been an open issue in the field for several years. In this paper, we develop a sparse triangular solver that takes advantage of the supernodal structures of the triangular matrices that come from the direct factorization of a sparse matrix. We implemented our solver using Kokkos and Kokkos Kernels such that our solver is portable to different manycore architectures. This has the additional benefit of allowing our triangular solver to use the team-level kernels and take advantage of the hierarchical parallelism available on the GPU. We compare the effects of different scheduling schemes on the performance and also investigate an algorithmic variant called the partitioned inverse. Our performance results on an NVIDIA V100 or P100 GPU demonstrate that our implementation can be 12.4× or 19.5× faster than the vendor optimized implementation in NVIDIA's CuSPARSE library.

ACM Reference Format:
Ichitaro Yamazaki, Sivasankaran Rajamanickam, and Nathan David Ellingwood. 2018. Performance Portable Supernode-based Sparse Triangular Solver for Manycore Architectures. In *ICPP '20: International Conference on Parallel Processing August 17–20, 2020, Edmont, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Sparse triangular solver is an important kernel in several computational science or engineering applications. However, the sparsity pattern of the triangular matrix limits the amount of the parallelism available for the solver to exploit. As a result, it is notoriously challenging to parallelize the sparse triangular solve on a manycore architecture such as a GPU. Instead of developing a sparse triangular solver that targets a general sparsity pattern, we focus on the case when a direct sparse matrix factorization is used to compute the triangular matrix. In this particular case, the triangular matrix typically has dense blocks called *supernodes*. We exploit this supernodal structure to accelerate the triangular solves. This use case covers a broad set of applications to warrant designing a triangular solve just for it.

For instance, computation frameworks such as SIERRA Structural Dynamics (SIERRA-SD) [24] implements finite element analysis for structural dynamics on distributed-memory computers. Computational frameworks such as these rely on domain-decomposition based linear solvers [9, 11], where a sparse direct factorization is used to solve each of the local problems. In a typical distributed-memory simulation, each process applies the sparse triangular solve $\sim 10^4$ times for each factorization. As a result, sparse triangular solves often dominate the simulation time. Hence, the performance improvement in the local triangular solve (used by each process) can directly impact the simulation time.

A second use case arises in computation simulations such as a low Mach fluids simulation that uses multigrid preconditioners on a distributed-memory computer [17]. In this case, a local sparse triangular solver is used as part of the coarse grid solve and as a smoother for the multigrid methods. A sparse direct factorization is typically used for the coarse grid, while either an incomplete or a complete factorization is used for the smoother, depending on the problem.

In this paper, to enhance the performance of the supernode-based sparse triangular solver, we study the effects of different techniques, including the level-set and dynamic scheduling, on the solver performance. We also investigate an algorithmic variant called the partitioned inverse [1], which partitions the triangular matrix and expresses its inverse as the product of the inverse of the partitioned matrices. This technique from two decades ago has not yet been investigated on a manycore architecture such as GPU. We revisit this technique where the triangular matrix is partitioned based on the supernodal level set, and demonstrate that it is a valid option on a GPU.

Our implementation is based on Kokkos [10] and is now available in Kokkos Kernels library [19]. Kokkos is a programming model that allows implementing performance portable applications on different manycore architectures. Hence, our solver can run either on the shared-memory CPUs or on the current NVIDIA GPU. It will also allow our solver to be available on other types of GPUs such as those from AMD or Intel in the future.

Our solver uses several key features of Kokkos Kernels including *team-level* linear algebra kernels that allow the batched operations on the independent supernodes in parallel (independent supernodes are processed in parallel, using a team of threads on each supernode). Hence, the team-level kernels can effectively map the supernodes to the hierarchical

ICPP '20, August 17–20, 2020, Edmonton, Canada

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

parallelism available on the GPU. This also aims to cover an important gap in the current algorithmic space. In order to effectively utilize manycore architectures, the algorithm has to exploit more parallelism. The supernode-based algorithm exposes such parallelism launching a batch of threaded linear algebra kernels on the independent supernodal blocks on top of a traditional parallel algorithm (e.g. level scheduling).

We present experimental results to demonstrate the trade-offs between the different approaches in term of stability, storage, and performance, and study the effects of different scheduling schemes on the GPU performance. Overall, the new sparse triangular solver obtains the respective speedups of up to 12.4× or 19.5× over the vendor optimized implementation on the NVIDIA V100 or P100 GPU. The main contributions of the papers include:

- A novel implementation of the supernode-based sparse triangular solver on the GPU that exploits hierarchical parallelism using team-level kernels;
- An investigation of using the partitioned inverse method for the supernode-based sparse-triangular solve on the GPU, where the matrix is partitioned based on the supernodal level sets; and
- A performance study of eight different implementations of the sparse-triangular solve, with various scheduling and algorithmic choices, on the GPU, and their impacts on the memory usage, performance, and stability.

The rest of the paper is organized as follows. After listing related works in Section 2, we review the sparse supernode-based factorization in Section 3. We then describe the algorithms and implementations of our sparse-triangular solver in Section 4 and 5, respectively. Finally, we present our experimental setups in Section 6 and results in Section 7. Since we focus on the supernodal block algorithms, we use $L_{i,j}$ to denote the (i,j)th supernodal **block** of the matrix L and $\mathbf{x}_i$ to refer to the ith **block** of the vector $\mathbf{x}$, while n_s is the number of supernodal columns in L.

2 RELATED WORK

Since sparse triangular solve is a critical kernel in many applications, significant efforts have been made to improve the parallel performance of a general-purpose sparse triangular solver both on CPUs [3, 5, 22] and on GPUs [14, 15, 18, 20, 23]. On shared-memory CPUs, the best known algorithm is the Hybrid Triangular Solve (HTS), which uses level-set scheduling for relatively sparse portions of the triangular matrix and uses recursive blocking for the denser portions [5]. HTS does not utilize the supernodal structure of the matrix and relies on fine-grained synchronizations that may not scale well on manycore architectures. We evaluate an option similar to this approach (called *dynamic* scheduling) that uses such synchronizations on the block columns instead of columns.

Supernode-based triangular solvers are implemented along with supernode-based sparse matrix factorization packages such as SuperLU [16] and CHOLMOD [6]. Recently, one-sided communication is used for sparse triangular solve on distributed memory architectures [8]. However, these supernodal solvers primarily target the CPUs. They could offload the dense block operations to the GPUs using a vendor-provided BLAS package. This is our *default* approach. In our experiments using one GPU, it was slower than our other approaches (Section 7).

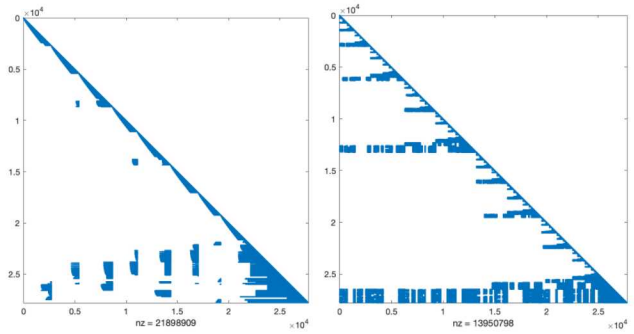
On a GPU, the NVIDIA's CuSPARSE library provides the vendor-optimized sparse-triangular solver that uses a level-set scheduling and hardware-specific optimizations to handle the "chains" of dependencies and to reduce the number of GPU kernel calls [18]. This implementation has been optimized by NVIDIA as the GPU architectures evolved. In Section 7, we compare our performance with this implementation.

Several other techniques have been explored to improve the performance of a general-purpose triangular solve or when the triangular matrix comes from an incomplete sparse matrix factorization. For instance, the level scheduling and another scheduling scheme called element scheduling were studied in [15]. Also in [14, 23], graph coloring was used to reorder the matrix. Though this matrix reordering may increase the parallelism for the triangular solve, when the matrix was used as a preconditioner, it could increase the number of iterations needed by the iterative solvers. As our current focus is on direct factorizations, coloring is not an option for reordering the matrix. However, we used other matrix ordering techniques such as nested dissection or minimum degree ordering to reduce the number of nonzeros in the triangular factors. On the GPU, a graph partitioning algorithm was also used to find the subblocks and to improve the performance of the triangular solver [20]. This analysis phase can be expensive, but its cost may be amortized over multiple solves.

None of these recent studies considered the partitioned inverse method [1]. We revisit this approach from two decades ago on the current manycore architectures. The primary advantage of this method is that it transforms the triangular solves into a sequence of sparse matrix-vector multiply (SpMV) operations. Our implementation (called *InvertOff*) is a special case of this approach where we define the partitions based on the level set partition of the supernodal graph from the direct factorization. One concern is the stability of this approach. Higham and Pothen [12] showed that stability is guaranteed when the matrix is well-conditioned and has a small number of partitions. We show that this approach perform well for the problems of our interests.

3 SUPERNODAL FACTORIZATION

When considering a dense factorization, instead of factoring one column or row of a matrix at a time, it is a standard approach to factor one block column or row at a time. This block factorization performs the majority of the operations using BLAS-3 subroutines (instead of BLAS-2 for the column-wise factorization), improving the cache-reuse. Since the data access can be expensive on the current computers, the block factorization can obtain much higher performance than the



(a) Natural ordering. (b) Nested dissection ordering.

Figure 1: Sparsity pattern of lower-triangular matrix computed using SuperLU for the SIERRA-SD `A_20x20x20` matrix. We computed the nested dissection ordering using METIS. With or without nested dissection, we had 943 or 30 levels, respectively.

column-wise factorization can. Such block algorithms are implemented in software packages like LAPACK [2] for factoring a dense matrix.

The block factorization can be applied to a sparse matrix by grouping a set of consecutive rows or columns with a similar sparsity pattern into a supernodal block. During the factorization, new nonzero entries, referred to as *fill*, are introduced in the triangular factors, increasing both the storage and computational costs of factorization. To obtain high performance, it is critical to reorder the matrix before the factorization. For instance, Figure 1 shows that a proper matrix ordering can reduce the number of fill and also increase the supernodal block sizes in the triangular factor. For our experiments, we use the software packages SuperLU [16] and METIS [13] that implements the supernodal factorization of a sparse nonsymmetric general matrix, and the nested-dissection ordering to reduce the number of fill, respectively. Here, we only gave a short background of the sparse factorization and refer the interested reader to a recent survey [7].

The sparse triangular solver can also take advantage of the supernodal structures, allowing us to replace the BLAS-1 operations of the column-wise algorithm with the BLAS-2 operations on the blocks. In parallel execution, the supernodal approach can also reduce the number of synchronizations and in turn improve the scalability of the parallel factorization or solve. These two features make the block algorithm more suitable for GPUs.

4 ALGORITHMS

We now describe our implementations of the parallel sparse triangular solver including the scheduling schemes used for the implementations.

4.1 Solver Steps

We first provide the overview of our sparse triangular solver. Our solver consists of the following three steps:

(1) *Symbolic Analysis* uses just the sparsity structure of the triangular matrix. This step needs to be performed once for multiple solves with a fixed sparsity structure. Based on the sparsity structure, it first computes the level sets. It then sets up the internal data structures for storing the matrix on the GPU, and internally saves the sparsity structure and the scheduling information (either level-set or dynamic). If the algorithmic option to merge the supernodes is enabled, it merges the supernodal blocks with the same sparsity structure before computing the level-set. If the algorithm option for fine-grained dynamic scheduling is used, the task dependencies are also computed.
(2) *Numerical Setup* copies the numerical values of the triangular matrix into the internal data structures. It also performs any numerical steps such as explicitly inverting the diagonal blocks and applying the inverse of the diagonal blocks to the corresponding off-diagonal blocks if such options are enabled. This step is performed once for several solves with different right-hand-side vectors but with the same matrix.
(3) *Solve* performs the sparse-triangular solve based on the level sets. It can also integrate other optimizations at the implementation level such as using different kernels at each level (e.g., batched kernels or device level kernels with CUDA streams).

Figure 2 shows each step of the solver at high level. In the next three subsections, we cover three major algorithmic choices: level-set scheduling or dynamic scheduling, and the partitioned-inverse (`invertOff` in Figure 2). The rest of options in Figure 2 are addressed in Section 5.1.

4.2 Level-set Scheduling

Level-set scheduling [21] is a standard technique for implementing a parallel sparse-triangular solve. For each level, we identify a set of unknowns in the solution vector, which can be independently computed in parallel. In this paper, we focus on the *supernode-based level-set scheduling*, where the dependencies among the *supernodal blocks* are analyzed based on the underlying directed acyclic graph (DAG) associated with the triangular matrix. There is an edge from the ith to the jth supernodes in the DAG if the corresponding block $L_{i,j}$ is not empty. The solution for the jth supernode can be computed once all the solutions corresponding to the supernodes with the out-edges to the jth supernode are computed. Figure 4b shows a DAG, which happens to be a binary tree in this particular case. This is a common case when the matrix is reordered by a nested dissection ordering. For this matrix, the level-set scheduler will compute a set of the available solution blocks, referred to as *leaf blocks*, level by level, starting from the bottom to the top of the tree.

Figure 3 shows the pseudocodes of two different implementations of the block level-set triangular solve that we looked at ($\mathbf{x}_s$ is the s-th solution block corresponding to $L_{s,s}$). In Figure 3a, if $\mathbf{x}_s$ is one of the available leaf solution blocks at the current level, we first compute $\mathbf{x}_s$, and then

Symbolic phase:
  Inputs: sparsity structure of L in CSR (rowmap, colind), and number and sizes of supernodes (n_s, sizes)
  Options: **merge** and **levelset/dynamic**
  1. if **merge** then
  2. merge the supernodes and update (rowmap, colind) and (n_s, sizes)
  3. compute and store level-sets
  4. if **dynamic** then
  5. compute a counter for each task to keep track of its dependencies
  6. store the supernodal structure in a single CSC/CSR structure

Numeric phase:
  Inputs: numerical values of L
  Options: **gemv/spmv**, **invertDiag**, and **invertOff**
  1. copy numerical values into internal CSC/CSR structure
  2. if **invertDiag** then
  3. invert diagonal blocks
  4. if **invertOff** then
  5. apply inverse of diagonal blocks to off-diagonal blocks
  6. if **levelset** with **spmv** then
  7. store each L_ℓ in a separate CSR without zero entries
  8. if not **invertOff** then
  9. store diagonal blocks of L_ℓ separately from off-diagonal blocks

Solve phase:
  Inputs: right-hand-side
  Options: **Stream**
  Parameters: threshold to switch between device/batched **gemv** (default is 250)
  use device **gemv** if average supernode size at a level > threshold
  1. for each level do
  2. if **invertOff** then
  3. if **spmv** or small **gemv** then
  4. launch one spmv kernel or one batched gemv kernel for all the supernodal columns at this level
  5. else
  6. launch one gemv for each of supernodal columns, on different streams **if Stream** using a default stream **otherwise**
  7. else
  8. if **invertDiag** then
  9. launch batched/streamed gemv or spmv with all the diagonal blocks
  10. else
  11. launch batched/streamed trsm for all the diagonal blocks
  12. launch batched/streamed gemv or spmv with all the off-diagonal blocks

Figure 2: Algorithmic flow at each solver phase with lower-triangular matrix L.

use $\mathbf{x}_s$ to update all the solution blocks that depend on $\mathbf{x}_s$ In contrast, in Figure 3b, we first update the leaf solution block $\mathbf{x}_s$ using the previously-computed solution blocks, and then compute $\mathbf{x}_s$. At each level of the scheduling, all the leaf solution blocks $\mathbf{x}_s$ can be computed in parallel.

Figure 4a illustrates these two approaches. Due to the way the lower-triangular matrix is traversed, we referred to the first approach as push-based (right-looking), while the second approach is called pull-based (left-looking). We use the column-major and row-major storages for the push-based and pull-based approach, respectively.

4.3 Partitioned Inverses

Our implementation exploits the two-level of parallelism: 1) all the available solution blocks $\mathbf{x}_s$ are computed in parallel

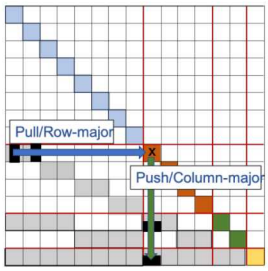
```
1. for each level
2.  for each s in this level
3.   // compute sth solution
4.   x_s := L_{s,s}^{-1} x_s
5.   // use sth solution
     // to update child RHS
6.   for i = s+1, s+2, ..., n_s
7.    x_i := x_i - L_{i,s} x_s
8.   end for
9.  end for
10.end for
```

(a) Push (col-major/left-look).

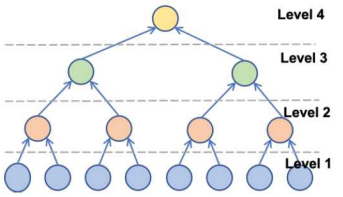
```
1. for each level
2.  for each s in this level
3.   // update sth RHS
     // with computed solution
4.   for j = 1, 2, ..., s-1
5.    x_s := x_s - L_{s,j} x_j
6.   end for
7.   // compute sth solution
8.   x_s := L_{s,s}^{-1} x_s
9.  end for
10.end for
```

(b) Pull (row-major/right-look).

Figure 3: Sparse-triangular solve based on level-set scheduling of supernode *blocks*. The for-loop for the block column or block row indexes, j or i, are executed with respect to the *block* sparsity of the matrix (we do not operate with the empty block $L_{s,j}$ or $L_{i,s}$).



(a) Push/Pull(column/row major) approach.



(b) DAG representation of block matrix.

Figure 4: Supernode-based level-set scheduling with nested-dissection ordering. In (a), only the nonzero block rows or columns in the supernodal block (indicated by black blocks in Figure 4a) are stored in the block row or column major scheme, respectively.

at each level and 2) each solution block $\mathbf{x}_s$ is computed using a threaded kernel. The dense triangular-solve `trsm`, which is needed to compute the solution block $\mathbf{x}_s$ (on Line 4 or 8 in Figure 3a or 3b), is a fundamentally sequential algorithm. Hence, it cannot exploit the thread parallelism as well as the matrix-vector multiply `gemv` used to update the solution vector (on Line 7 or 5 in Figure 3a or 3b).

To avoid the potential performance bottleneck with `trsm`, we explicitly compute the inverse of the diagonal blocks of L in the numeric phase and use `gemv` to compute the solution $\mathbf{x}_s$ in the solve phase. We call this approach *InvertDiag* in Figure 2. Moreover, we can apply the inverse of the diagonal blocks to the corresponding off-diagonal blocks in the numeric phase (i.e., if $L_{s,s} := L_{s:s}^{-1}$, then $L_{s+1:n_s,s} := L_{s+1:n_s,s} L_{s,s}^{-1}$). Then, in the solve phase, this allows us to combine `trsm` and `gemv` calls into a single `gemv` call to compute $\mathbf{x}_s$ and update the remaining solutions (i.e., $\mathbf{x}_{s:n_s} := L_{s:n_s,s} \mathbf{x}_s$ for Lines 4 through 7 in Figure 3a), halving the number of kernel launches. We call this approach *InvertOff* in Figure 2.

With this *InvertOff* scheme, we can write the inverse of the triangular matrix as the product of the partitioned inverses:

$$L^{-1} = \prod_{\ell=1}^{n_\ell} L_\ell^{-1},$$

where n_ℓ is the number of levels, and L_ℓ is an identity matrix except that the supernodal columns, which belong to the ℓth level, are replaced with the corresponding columns in L. Hence, we can apply the inverse of L to a vector by applying the sequence of the sparse-matrix vector products, which is often more efficient than the sparse-triangular solve on a manycore architecture. One trade-off is the increase in the memory (to store the additional nonzero entries introduced by applying the inverse of the diagonal blocks to the corresponding off-diagonal blocks). However, all the new nonzeros are introduced within the non-empty blocks of L, and this is a block version of *no-fill* partition [1].

Another trade-off of the partitioned inverse is the potential numerical instability. There are several studies on the numerical stability of the partitioned inverse method [12]. In our application, the triangular matrix is computed either by the LU factorization with partial pivoting, or by the Cholesky factorization. Hence, computing the inverse of the diagonal blocks or the partitioned matrix is often more stable, compared with a random triangular matrix.

4.4 Dynamic Scheduling

The level-set scheduling exposes the parallelism within the level, but there is a synchronization at the end of each level. As a result, if there is not enough computation at any level, we may not fully utilize the compute power of the manycore architecture. In order to exploit more parallelism, we also looked at a dynamic scheduling scheme, where each of the node in the DAG (see Figure 4b) are executed as soon as its child tasks were completed. Typically, supernodes provide enough computation at each level, but if supernodes have different sizes or do not provide enough computation at one level, the dynamic scheme may be beneficial. Our implementation of the dynamic scheduling is described in Section 5.4.

5 IMPLEMENTATIONS

We implemented our solver using Kokkos [10], which is a C++ library to provide portable performance across different manycore architectures. By writing our solvers in Kokkos, it can run on different machines with a single code base.

5.1 Algorithmic Options

Here, we list the different solver options, each of which we describe in the following subsections, and whose effects we study on the solver performance in Section 7:

(1) **Default** Our default implementation performs the supernodal triangular solve (Figure 3) using the device-level kernels (e.g., CuBLAS `gemv` or `trsm`) on one supernodal block column at a time.

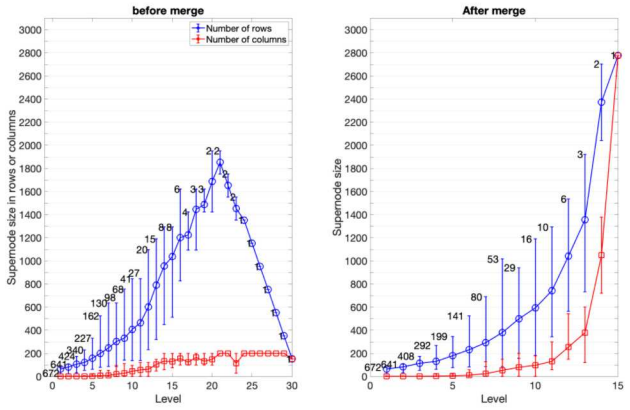


Figure 5: Supernode sizes of `A_20x20x20` from SierraSD with nested dissection. The error bars show the minimum, average, and maximum number of rows or columns of the supernode blocks at each level, while the numbers by the markers show the number of supernodes at each level. Rows/columns here are *not* block rows/columns. Merging the supernodes results in fewer larger supernodes (left vs right).

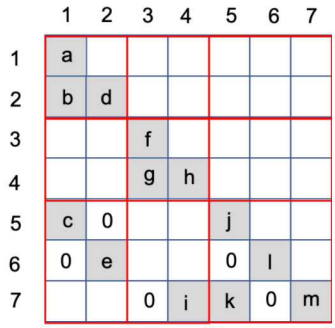
(2) **Stream** In order to more efficiently utilize the compute power of the manycore architecture (e.g., GPU), at each level, we execute each of the independent device-level kernel calls on a different stream in parallel.

(3) **Team** We replace the device-level kernels with the team-level (batched) kernels for `gemv` or `trsm`, hence launching the "batch" of independent team-level kernels (of variable-sizes) with a single kernel launch. This often exposes the hierarchical parallelism more effectively than `Stream` can. We use Kokkos Kernels team level gemv or trsm for this option.

(4) **Merge** We merge the supernodes that have the same supernodal structure. Figure 5 shows one example where merging the supernodes results in fewer larger supernodes (15 levels after merge instead of 30). This has the trade-off of increasing the memory usage (more explicit zeros may be needed to form a larger supernode). But, this often improves the performance by reducing the number of kernel launches (or the number of levels) and increasing the compute intensity at each level.

(5) **InvertDiag** At each level, instead of applying the triangular solve `trsm` with the diagonal blocks, we invert these blocks (during the numerical setup) and use the matrix-vector multiply `gemv` to apply the inverse of the diagonal blocks to the vector (for the solve phase).

(6) **InvertOff** At each level, we combine two matrix-vector multiplies: one to apply the inverse of the diagonal blocks and the other to update with the off-diagonal blocks. This is done by applying the inverse of the diagonal block to the corresponding off-diagonal blocks (during the numerical setup). This is equivalent to the



(a) Matrix with supernodal blocks shown with red lines.

colptr: 1 5 8 11 13 16 18 19
values: a b c 0 d 0 e f g 0 h i j 0 k l 0 m
rowind: 1 2 5 6 2 5 6 3 4 7 4 7 5 6 7 6 7 7

(b) CSC storing the supernodal blocks, where `values` and `rowind` store the numerical values and row indices of the entries in nonempty blocks, and `colptr` points to the beginning of each column in `values`.

Figure 6: Supernodal blocks stored in CSC.

partitioned inverse method (based on the level-set supernode partition), and may improve the performance.

(7) **SpMV** Instead of relying on the batched `gemv`, we store each partitioned inverse L_ℓ^{-1} in a separate sparse format and use the sparse-matrix vector product (`spmv`) kernel. This trades off the knowledge of the block structure, but `spmv` is often optimized on the specific hardware (e.g., like one from CuSPARSE). This also reduces the memory requirement since it removes the explicit zero entries used to form the supernodal blocks.

(8) **Dynamic** executes each task as soon as its data dependencies are satisfied. Hence, it removes the synchronization at each level of scheduling.

InvertOff may increase the storage cost for **SpMV** due to the extra fill, while for `gemv`, the fill is introduced only within the non-empty blocks and the storage cost stays the same.

5.2 Basic Components

For all the solver options listed in Section 5.1, we store the triangular matrix either in the Compressed Sparse Column (CSC) or in the Compressed Sparse Row (CSR) format, respectively, for our push-based or pull-based approaches in Figure 3. If we choose to use `gemv` (instead of `spmv`), then we potentially store explicit zero entries to form the supermodel blocks, but this allows us to compute the solution using the dense matrix kernels like BLAS. In this case, we only store the union of the nonzero rows or columns in each supernodal block. The empty rows or columns in the block are not stored. Figure 6 shows an example of supernodal blocks stored in the CSC format.

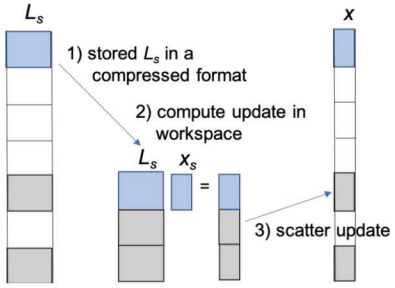


Figure 7: Illustration of block update, where the supernodal block L_s in the lower-triangular matrix is stored in CSC format, and the updates are computed in the workspace before being scattered into the solution.

In the row-major pull-based approach, to update the solution $\mathbf{x}_s$ with the multiple off-diagonal blocks (on Line 5 of Figure 3b), we first gather the previously-computed solutions, corresponding to the nonzero columns in $L_{s,:}$, into a workspace, then we update $\mathbf{x}_s$ with a single `gemv` call. Similarly, in the column-major pull-based approach, to update the multiple solution blocks (on Line 7 of Figure 3a), we first call `gemv` to accumulate all the updates in a workspace then the updates are scattered into $\mathbf{x}$ (see Figure 7 for an illustration). Even with the overhead of gathering or scattering the vector, it is often more efficient to compute the update with one `gemv` call rather than using one `gemv` call per block.

At each level of the column-major push approach, different block columns may update the same part of the solution in parallel. Hence, for updating the remaining solutions, we use atomic operations among the teams. In contrast, in the row-major pull approach, we do not need the atomic operations among the teams, but to update each supernodal block of the solution, the kernel performs the reduction operation among the threads within its team. This could lead to performance differences, which we will study later in Section 7.

5.3 Level-set Scheduling

At each level, we compute all the available leaf solution blocks in parallel. The corresponding updates are also executed in parallel. To exploit as much parallelism as possible, we employ hierarchical parallelization:

- At the initial levels of the solve, we typically have a large number of relatively small supernodes (see Figure 5). If we launch one kernel to compute or update the small supernode, the overhead associated with the kernel launch can dominate the execution time (referred to as "Default" setup). To effectively utilize the manycore architecture, we rely on a batched kernel, which can launch a batch of the multiple independent kernels (e.g., `trsm` followed by `gemv` for each $\mathbf{x}_s$ in the level) in parallel with a single call. Then, a team of threads executes the team-level thread-parallel kernel, independently (referred to as "Team" setup).

```
1.  for ( size_type lvl = 0; lvl < num_levels; ++lvl ) {
2.   using lower_functor = LowerTriSupernodalFunctor
       <ColPtrType, RowIndType, ValuesType, SolType,
        NodeGroupType>;
3.
4.   // specify parallel execution policy:
5.   // uses num_blocks teams of threads (one team/block column)
6.   // use default number of threads per team (i.e., AUTO)
7.   size_type num_blocks = num_blocks_per_level(lvl);
8.   TeamPolicy<execution_space> team_policy (num_blocks,
                                              Kokkos::AUTO);
9.   // create triangular solve functor
10.  // (colptr, rowind, valus) stores L in CSC
11.  // suprcols(s) specifies column-offset to s-th supernode
12.  // block_id stores block-column ids, ordered by level-set
13.  lower_functor sptrsv_functor (
         supercols, colptr, rowind, values,
         lvl, sol, work, work_offset,
         block_id, num_blocks_processed);
14.
15.  // launch functor
16.  Kokkos::parallel_for ("lower_solve_supernode",
                          team_policy, sptrsv_functor);
17.  num_blocks_processed += num_blocks
18. }
```

Figure 8: Driver function for sparse lower-triangular solve by level-set scheduling.

- As the supernode becomes large enough at the later stage of solve, it is more efficient to rely on the standard kernel (like CuBLAS), instead of a batched kernel (which is optimized for small sized matrices). To exploit more parallelism, we may execute each of the independent kernel calls on a different stream in parallel (referred to as "Stream" setup).

As an illustration, Figure 8 shows our driver routine for the sparse lower-triangular solve using Kokkos. On line 9, we assign one team of threads for computing one supernodal block of the solution (i.e., `num_blocks` is the number of the available leaf blocks at the level `lvl`, and we use the keyword `AUTO` to let Kokkos decide how many threads to use per team, e.g., one thread on Intel Haswell CPUs and 64 threads on an NVIDIA P100 GPU). On line 13, we create a functor which is called by each team of threads. Finally, on line 16, we launch the parallel-for to compute the solution blocks in parallel.

Figure 9 then shows the functor called by each thread team. The first few lines of the functor extract the information about the thread and supernode, but the main computation is at Lines 29 and 34, where the team-level threaded `trsm` and `gemv` are called. Since each team executes both `trsm` and `gemv`, it can reuse the data more effectively, compared with launching the batched `trsm` followed by the batched `gemv`, separately. For the rest of the functor, this team's threads work together to scatter the solution back into the output vector (using the atomic operations).

For the partitioned inverse, instead of relying on the batched `gemv` at each level, we store L_ℓ^{-1} in the sparse storage format and use the sparse-matrix vector product `spmv` kernel (e.g., like one from CuSparse). This also reduces the storage requirement since the explicit zeros are not stored. If the diagonal inverse is not applied to the off-diagonal blocks, then

```
1.  KOKKOS_INLINE_FUNCTION
2.  void operator()(const member_type & team) const {
3.   // batch id
4.   const int team_rank = team.team_rank();
5.   const int team_size = team.team_size ();
6.   const int thread_rank = team.thread_rank ();
7.   // supernode sizes
8.   auto s = block_id (num_blocks_processed + team_rank);
9.   int j1 = supercols[s], j2 = supercols[s+1];
10.  int i1 = colptr (j1), i2 = colptr (j1+1);
11.  // number of columns in the s-th supernode column
12.  int nscol = j2 - j1 ;
13.  // total number of rows in the supernodes
14.  int nsrow  = i2 - i1;
15.  // total number of rows in off-diagonal supernodes
16.  int nsrow2 = nsrow - nscol;
17.
18.  // extract s-th supernocal column
19.  scalar_t *dataL = const_cast<scalar_t*> (values.data ());
20.  View<scalar_t**, LayoutLeft, memory_space, MemoryUnmanaged>
      viewL (&dataL[i1], nsrow, nscol);
21.  // workspace
22.  int workoffset = work_offset (s);
23.  auto Z = subview (work, range_type(offset+nscol,
24.                                     offset+nsrow));
25.  // call TRSM with diagonal
26.  auto Xj = subview (X, range_type(j1, j2));
27.  auto Ljj = subview (viewL, range_type (0, nscol),
                         ALL ());
28.  TeamTrsm::invoke(team, one, Ljj, Xj);
29.  // call GEMV with off-diagonal
30.  auto Y = subview (work, range_type(offset,
31.                                     offset+nsrow));
32.  auto Lij = subview (viewL, range_type (nscol, nsrow),
                         ALL ());
33.  TeamGemv::invoke(team, one, Lij, Xj, zero, Y);
34.
35.  /* scatter updated back into X */
36.  int k = i1 + nscol ;      // offset into rowind
37.  View<scalar_t*, memory_space,
38.       MemoryTraits<Unmanaged | Atomic>>
      Xatomic(X.data(), X.extent(0));
39.  for (int ii = thread_rank; ii < nsrow2; ii += team_size) {
40.   int i = rowind (k + ii);
41.   Xatomic (i) -= Z (ii);
42.  }
43.  team.team_barrier();
44. }
```

Figure 9: Kokkos-kernel functor (called by each team of threads) for column-major push-based sparse lower-triangular solve by level-set scheduling.

at each level, we call `spmv` twice (once to apply the inverses of all the diagonal blocks, followed by another to update the off-diagonal blocks).

5.4 Dynamic Scheduling

For the dynamic scheduling, we focus on the push-based scheme, where each task computes a solution block and performs the corresponding updates (i.e., the inner s-th step in Figure 3a). As a part of symbolic setup, we first figure out the number of tasks that each task depends on and then assign a set of tasks to a team of threads (where the number of team is either specified by the user or given as the largest number of tasks at a level). In our current implementation, we assign the tasks at each level to the teams in a round-robin fashion. Then, at the solve time, we launch these teams in parallel-for, where each team executes tasks from the queue

id	name	type	n	$\frac{nnz}{n}$	n_ℓ	error
1	ACTIVSg70K	power system grid	69,999	12.6	83	0.003
2	dawson5	structural problem	51,537	770.4	1277	3.512
3	qa8fk	acoustic problem	66,127	653.3	22	0.006
4	FEM3Dtherm	thermal problem	17,880	324.6	15	0.008
5	thermal1	thermal problem	82,654	58.7	27	0.002
6	apache1	3D finite difference	80,800	240.2	25	0.002
7	apache2	3D finite difference	715,176	53.6	32	0.001
8	helm2d03	2D problem	392,257	14.9	109	0.018

Figure 10: Test matrices from SuiteSparse matrix collection, where "n" is the matrix dimension, "$\frac{nnz}{n}$" is the ratio of the total number of nonzero entries in the lower and upper triangular factors, computed by SuperLU, over the matrix dimension, "n_ℓ" is the number of levels, and "error" is the backward error $\frac{\|\mathbf{b}-A\mathbf{x}\|}{\epsilon(\|\mathbf{b}\|+\|A\|\|\mathbf{x}\|)}$ using the partitioned inverse method (showing it was stable for all these matrices).

	symbolic	compute	L-solve CSC	U-solve CSR	U-solve CSC	fill-ratio
CuSparse	0.0487	0.2587	0.0087	0.0167	0.0185	13.7
Default	0.3000	0.2712	0.0888	0.0918	0.1343	28.9
Team	0.2921	0.3067	0.0038	0.0111	0.0051	28.9
Merge	0.5611	0.6444	0.0031	0.0083	0.0037	35.4
InvertDiag	0.5575	1.2576	0.0016	0.0076	0.0024	35.4
InvertOff	0.7067	7.2798	0.0015	–	0.0023	35.4
Stream(5)	0.7063	7.3001	0.0013	–	0.0020	35.4

(a) batched `gemv` based implementation.

	symbolic	compute	L-solve	U-solve	fill-ratio
InvertDiag	0.6939	1.6820	0.0021	0.0018	14.7
InvertOff	0.6912	7.8646	0.0010	0.0012	15.5

(b) `spmv` based implementation.

Figure 11: Sparse-triangular time for `A_20x20x20` on an NVIDIA V100 GPU ($n = 27,783$ and $nnz/n = 73.5$). Our solver options are listed in Section 5.1, and the fill-ratio is defined as the number of nonzeros in the factor over the number of the nonzeros in the original matrix.

containing the set of tasks that are ready for execution. Once the team completes a task, it decrements the counters for the tasks that depend on the completed task, using an atomic. Once the counter reaches zero, the corresponding task is moved to the ready queue. Hence, there are some overhead of figuring out the dependencies at the solve time.

6 EXPERIMENT SETUP

For our experiments, we used matrices from the SIERRA-SD simulations [24] and from the SuiteSparse Matrix collection. Table 10 lists our test matrices from the sparse collection, covering different types of applications and sparsity patterns. To reduce the number of fill in the factors, we used the nested dissection ordering (computed by METIS [13]). We report the average performance of 100 runs.

The main inputs to our solver are the sparse triangular matrix (stored in a standard format such as CSR) and the information about the supernodes (i.e., the number and the sizes of the supernodes). Hence, our solver does not depend on a particular factorization package. Nevertheless, our solver is currently interfaced with SuperLU [16] and CHOLMOD [6]. Our symbolic and numerical phases read the LU factors from SuperLU or CHOLMOD into Kokkos graph and sparse matrix data structures, respectively. Internally, our solver currently uses either the CSR or the CSC format for storing the triangular matrices. For a symmetric matrix (e.g., with CHOLMOD), it is possible to store the lower and upper triangular matrices in the CSC and CSR formats, respectively, and hence to store only one copy of the factor.

For our GPU experiments, we use an NVIDA V100 or P100 GPU (with IBM Power9 or Power8 CPUs as host, respectively).[1] For our V100 or P100 runs, we compiled our codes using the GNU `g++` compiler version 6.40 or 5.4.0 and the NVIDIA `nvcc` compiler version 10.1.243 or 10.0.130, respectively. We compare the performance of our solver with that of the vendor optimized `cusparseDcsrsv2_solve` from the NVIDIA's CuSPARSE library. To call this NVIDIA's sparse triangular solver, we only store the nonzero entries of the factors in the CSC format (all the explicit zero entries were removed). We run the CuSPARSE solver with the level-based scheduling by calling `cusparseDcsrsv2_analysis` with `CUSPARSE_SOLVE_POLICY_USE_LEVEL`.

To demonstrate the portability of our code, we also show the performance on the shared-memory two 10-core Intel Haswell CPUs with 64GB of main memory. We compiled our codes using GNU compiler version 7.3.0 and the optimization flag `-O3`, and linked to OpenBLAS version 0.3.5. We use these setups for comparing the performance of SuperLU and our triangular solve implementations on the CPUs.

7 PERFORMANCE RESULTS

We now study the performance of our supernode-based sparse-triangular solver.

7.1 SIERRA-SD matrices on a GPU

In Figures 11a and 12a, we compare the performance of our sparse triangular solve with that of CuSPARSE for a matrix from a SIERRA-SD 3D simulation on an NVIDIA V100 and P100 GPU, respectively. In these tables, we used the batched `gemv` at each level (instead of `spmv`). For all the SIERRA-SD matrices, all the methods, including the partitioned inverse, obtained the backward errors in the order of machine epsilon.

We currently perform both symbolic and numerical setups on the CPU, sequentially, where the numerical time is dominated by the time to compute the inverse of diagonal blocks (using LAPACK's `trtri`) and the time to apply the inverse to the corresponding off-diagonal blocks (using BLAS' `trmm`) if we choose the InvertDiag or InvertOff options. Our current focus is on the solve time, which is the bottleneck in the simulations of our interests. These two routines are not yet available in Kokkos Kernels as team-level options, but we plan to perform `trtri` and `trmm` on the GPU to reduce the setup time once they become available. After the setup, the

[1] Our V100 results are on the Summit supercomputer at the Oak Ridge Leadership Computing Facility.

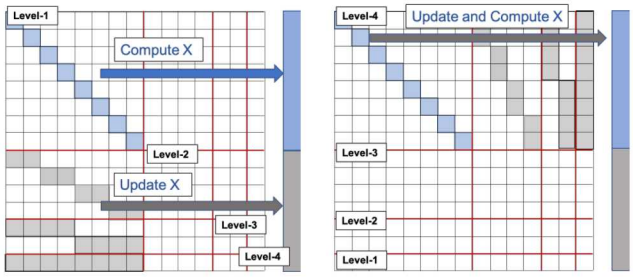
	symbolic	compute	L-solve CSC	U-solve CSR	U-solve CSC	fill-ratio
CuSparse	0.0446	0.2597	0.0227	0.0355	0.0359	13.7
Default	0.2147	0.2796	0.1163	0.1067	0.0922	13.7
Team	0.4420	0.2823	0.0041	0.0108	0.0046	28.9
Merge	0.7401	0.5331	0.0033	0.0097	0.0043	35.4
InvertDiag	0.7430	1.6134	0.0022	0.0077	0.0025	35.4
InvertOff	0.7416	7.0458	0.0020	–	0.0024	35.4
Stream(5)	0.7148	7.0445	0.0014	–	0.0022	35.4
Dynamic	0.7447	7.0496	0.0015	–	–	

(a) batched `gemv` based implementation.

	symbolic	compute	L-solve	U-solve	fill-ratio
InvertFull	3.3638	529.40	0.0022	0.0019	64.4
InvertDiag	0.7506	2.0961	0.0028	0.0027	14.7
InvertOff	0.7399	7.9446	0.0013	0.0017	15.5

(b) `spmv` based implementation.

Figure 12: Sparse-triangular time for `A_20x20x20` on an NVIDIA P100 GPU ($n = 27,783$ and $nnz/n = 73.5$). For full-inversion in (b), we computed the inverse using the dense matrix kernel `trtri`, and the symbolic and compute times are just for reference.



(a) Lower-triangular update at 1st level. **(b) Upper-triangular update at 4th level.**

Figure 13: Illustration of CSC-based lower-triangular and CSR-based upper-triangular update.

lower and upper triangular matrices are copied to the GPU in either the CSR or CSC format.

For the "default" and "team" setups in the table, we did not merge any supernode nor compute the inverse of the diagonal blocks. Hence, we simply apply the level-set scheduling to the supernodes computed by SuperLU. The only difference is that at each level, the default setup calls the device-level kernel (i.e., CuBLAS) on each supernode, while the team setup uses the team-level kernels. We clearly see the benefit of using the team-level kernels. Furthermore, even without further optimizations, our team-level implementation takes advantage of the supernodal structures, and obtains a significant speedup over CuSPARSE (on V100 and P100, we obtained 2.3× and 5.5× speedups for the lower-triangular solve, and 1.5× or 3.6× and 3.3× or 7.8× speedups for the upper-triangular solve in CSR or CSC, respectively).

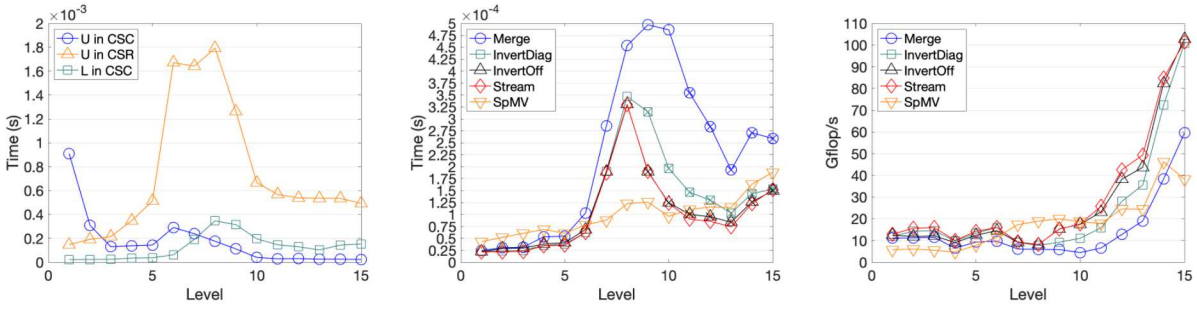
The upper-triangular solve performed better using the CSC format than using the CSR format. Figures 14b and 14c illustrate the performance difference at each level of scheduling. Using the CSC format, the upper-triangular solve was about as fast as the lower-triangular solve using the CSC format, except for the first couple of levels, where the upper-triangular solve needs to update the remaining solution using relatively large supernodes.

In contrast, the upper-triangular solve using the CSR format obtained significantly lower performance compared with the lower-triangular solve using the CSC format. Figure 13 illustrates these two solves, where the matrix is assumed to be symmetric, and hence, the upper-triangular factor is the transpose of the lower-triangular factor (without pivoting to maintain the numerical stability of the factorization). In this case, compared with the CSC-based lower-triangular solve, the CSR-based upper-triangular solve traverses the level-set DAG in the reverse order but operates on the same set of supernodes at the corresponding level (at the jth level, the upper-triangular solve operates on the same supernodes as the lower-triangular solve's $(n_\ell - j + 1)$th level). However, in the upper-triangular solve, each `gemv` kernel computes the solution primarily based on the dot-products, or the reductions, operations, while the lower-triangular solve scatters the updates to the remaining solutions (i.e., $\mathbf{x}_\ell := U_{\ell,\ell:n_\ell}\mathbf{x}_{\ell:n_\ell}$, compared with $\mathbf{x}_{\ell:n_\ell} := L_{\ell:n_\ell,\ell}\mathbf{x}_\ell$). This dot-product used for the upper-triangular solve is a notoriously challenging kernel to be optimized. We also looked at the performance of the lower-triangular solve using the CSR format, but it often performed better using the CSC format.

The rest of the rows in Figures 11a and 12a show the trade-offs between the performance and the storage requirement (and the setup costs). Compared with CuSPARSE, our implementation requires more memory since it explicitly stores the zero entries to form the supernodes. The memory requirement increases when the supernodes are merged, but it does not increase when the inverses are applied since all the fill occur within the blocks. Overall, for this problem, the supernode-based solver was up to 8.0× and 16.3× faster than CuSPARSE on a V100 and P100 GPU, respectively.

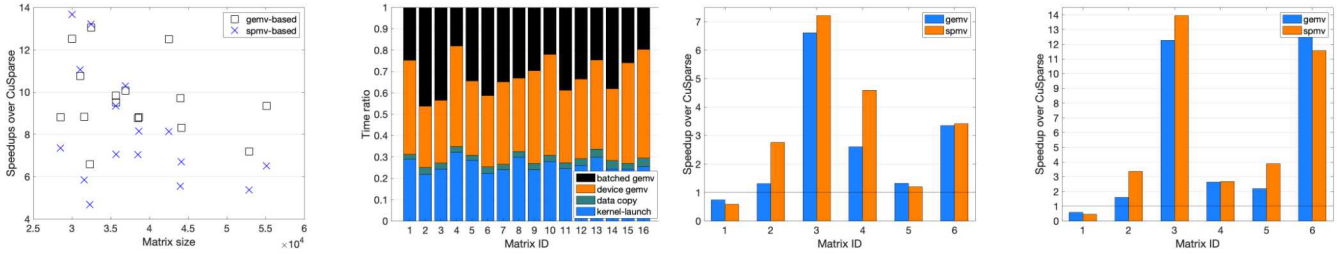
Figures 11b and 12b then show the performance of the partitioned inverse using `spmv` at each level. More fill could occur when the supernodes were merged or the diagonal inverses are applied to the off-diagonal blocks, but since only the nonzero entries are stored, using `spmv` reduces the memory requirement from when using `gemv` on the supernodal blocks. We also looked at computing the exact inverse of the triangular matrix. The full inverse can be applied without the level-set scheduling, but it often results in much more fill than the partitioned inverse. As a result, the partitioned inverse often outperformed the full inverse. Overall, the partitioned inverse obtained the respective speedups of up to 12.4× or 19.5× over CuSPARSE on a V100 or P100 GPU.

In our experiments, the best performance was obtained using the CSC format and the partitioned inverse (InvertOff) calling either `spmv` or batched `gemv` at each level. Hence, for the rest of the experiments, we focus on these two options. Figure 15a shows the speedups over CuSPARSE for the matrices of different sizes from the distributed SIERRA-SD



(a) Time (s) per level. (b) Time (s) per level (L-solve). (c) Gflop/s at each level (L-solve).

Figure 14: Performance per level for SIERRA-SD 3D problem (`A_20x20x20`) on an NVIDIA P100 GPU.



(a) Speedups over CuSPARSE. (b) Breakdown of solver time. (a) V100. (b) P100.

Figure 15: Performance of the partitioned inverse on an NVIDIA P100 GPU, for the local interior problems from a distributed SIERRA-SD run.

Figure 16: Speedups over CuSparse for the matrices from SuiteSparse collection on a GPU. For these matrices 1 through 6 (listed in Figure 10), CuSPARSE took 0.002, 0.180, 0.032, 0.008, 0.004, and 0.018 seconds on V100, and 0.002, 0.220, 0.061, 0.014, 0.007, and 0.039 seconds on P100.

simulation (these are the local interior submatrices on different processes). Compared with the 3D problem `A_20x20x20`, these local matrices typically had smaller supernodes, and obtained smaller speedups. However, we still see the effectiveness of our implementation over these different sizes of the matrices (i.e., $4.8\times \sim 13.7\times$ speedups). Figure 15b shows the breakdown of the solver time for the same matrices (showing 29% of time could be spent in the kernel launch).

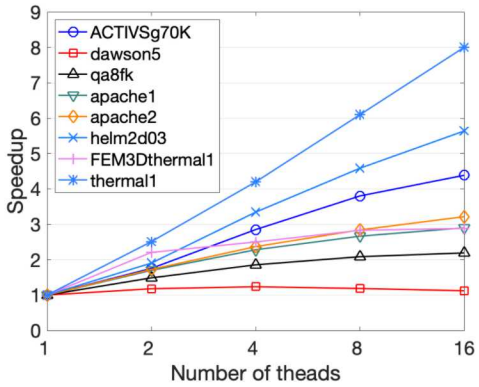
7.2 SuiteSparse matrices on GPU / CPUs

We now, in Figure 16, look at the GPU performance of the sparse-triangular solve for a few matrices from the SuiteSparse matrix collection (see Figure 10). The performance of our solver clearly depends on the sizes and structures of the supernodes. Our solver typically performs well when the supernodes are well separated such that we obtain a small number of levels, or have supernodes of large sizes.

Then, in Figure 17, we compare the performance of our sparse triangular solver (using batched `gemv`) with the SuperLU's sparse triangular solver on one CPU, and show the thread scalability of our solver. There are a few differences between SuperLU and our implementations. For instance, SuperLU stores the off-diagonal blocks of the upper-triangular matrix in a sparse format, and hence operates only with non-zero entries. In contrast, we operate on the supernodes with explicit zeros. Also, our solver relies on the team-level kernels for the batched operations, while SuperLU performs the block operation using the standard BLAS, whose vendor-optimized implementations are often available. Finally, the atomic operations, which our solver relies on, may have performance overhead, especially on the CPUs. These difference in the implementations could lead to their different sequential performance. SuperLU's sparse-triangular solve relies on the threaded BLAS/LAPACK to exploit the thread parallelism, and it performs similar to our default setup.

8 CONCLUSION

We studied the performance of a supernode-based sparse triangular solver, and the potential of the partitioned inverse method to improve its performance. Our code is based on Kokkos, and is portable to different manycore architecture, and uses the Kokkos-kernel's team-level kernels to enhance its performance. The solver is publicly available in the Kokkos-kernel library, and we are working to expose the functionality to the Trilinos users (e.g., through the Trilinos sparse direct solver Amesos2 [4]). We did not encounter any numerical instability with the partitioned inverse (combined with the



(a) Speedups over one thread.

name	SLU	number of threads 1	2	4	8	16
ACTIVSg70K	0.0043	0.0114	0.0065	0.0040	0.0030	0.0026
dawson5	0.1937	0.1244	0.1055	0.1007	0.1049	0.1109
qa8fk	0.1824	0.0892	0.0600	0.0481	0.0428	0.0407
apache1	0.0916	0.0618	0.0364	0.0271	0.0232	0.0213
apache2	1.1378	0.7283	0.4222	0.3075	0.2564	0.2265
helm2d03	0.2234	0.2012	0.1057	0.0602	0.0439	0.0357
FEM3Dthermal1	0.0244	0.0150	0.0068	0.0060	0.0053	0.0052
thermal1	0.0270	0.0256	0.0102	0.0061	0.0042	0.0032

(b) Time in seconds.

Figure 17: Performance with matrices from SuiteSparse Collection on Intel Haswell CPUs. In (b), "SLU" shows the sparse-triangular solve time using SuperLU sequential on one core.

stable factorization), but to ensure the stability, we are considering to integrate techniques such as iterative refinement.

We are working to further enhance its performance. In particular, the performance depends strongly on the performance of the underlying kernels (team-level or device-level). Hence, we are looking to improve their performance for our particular use cases (e.g., shapes of the dense blocks), or to interface with the vendor-optimized kernels (e.g., sparse-matrix vector multiply of CuSPARSE). We are also looking to amortize the kernel launch overhead, especially since we could have multiple kernel launches for each level, and the overhead could become significant in our solve time.

ACKNOWLEDGMENTS

The authors would like to thank Clark Dohrmann for providing SIERRA-SD matrices and for the valuable discussions. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy National Nuclear Security Administration under contract DE-NA0003525.

REFERENCES

[1] F. L. Alvarado, A. Pothen, and R. Schreiber. 1993. Highly Parallel Sparse Triangular Solution. In *Graph Theory and Sparse Matrix Computation. The IMA Volumes in Mathematics and its Applications*, A. George A, J. R. Gilbert, and J. W. H. Liu (Eds.). Springer, New York, NY, Chapter 56, 141–157.

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. 1999. *LAPACK User's Guide* (3 ed.). SIAM, Philadelphia, PA.

[3] E. Anderson and Y. Saad. 1989. Solving sparse triangular linear systems on parallel computers. *Int. J. High Speed Comput.* 1 (1989), 73–95.

[4] E. Bavier, M. Hoemmen, S. Rajamanickam, and H. Thornquist. 2012. Amesos2 and Belos: Direct and iterative solvers for large sparse linear systems. *Sci. Programming* 20 (2012), 241–255.

[5] A. M. Bradley. 2016. A hybrid multithreaded direct sparse triangular solver. In *2016 Proceedings of the Seventh SIAM Workshop on Combinatorial Scientific Computing*. SIAM, 13–22.

[6] Y. Chen, T. A. Davis, W. W. Hager, and S. Rajamanickam. 2008. Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate. *ACM Transactions on Mathematical Software (TOMS)* 35, 3 (2008), 1–14.

[7] Timothy A Davis, Sivasankaran Rajamanickam, and Wissam M Sid-Lakhdar. 2016. A survey of direct methods for sparse linear systems. *Acta Numerica* 25 (2016), 383–566.

[8] N. Ding, S. Williams, Y. Liu, and X. Li. 2020. Leveraging One-Sided Communication for Sparse Triangular Solvers. In *Proceedings of the SIAM Conference on Parallel Processing for Scientific Computing*.

[9] C. R. Dohrmann, A. Klawonn, and O. B. Widlund. 2008. Domain decomposition for less regular subdomains: Overlapping Schwarz in two dimensions. *SIAM J. Numer. Anal.* 46 (2008), 2153–2168.

[10] H. Edwards, C. Trott, and D. Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3202–3216.

[11] A. Heinlein, A. Klawonn, S. Rajamanickam, and O. Rheinbach. 2018. *FROSch: A Fast and Robust Overlapping Schwarz Domain Decomposition Preconditioner Based on Xpetra in Trilinos.* Technical Report. Sandia National Lab.(SNL-NM).

[12] N. J. Higham and A. Pothen. 1994. Stability of the partitioned inverse method for parallel solution of sparse triangular systems. *SIAM J. Sci. Comput.* 15 (1994), 139–148.

[13] G. Karypis. 2013. *METIS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices.* Technical Report.

[14] R. Li and Y. Saad. 2013. GPU-accelerated preconditioned iterative linear solvers. *Journal of Supercomputing* 63 (2013), 443–466.

[15] R. Li and C. Zhang. 2020. Efficient Parallel Implementations of Sparse Triangular Solves for GPU Architecture. In *Proceedings of SIAM Conference on Parallel Proc. for Sci. Comput.* 118–128.

[16] X. S. Li, J. W. Demmel, J. R. Gilbert, L. Grigori, P. Sao, M. Shao, and I. Yamazaki. 1999. *SuperLU Users' Guide.* Technical Report LBNL-44289.

[17] P. Lin, M. Bettencourt, S. Domino, T. Fisher, M. Hoemmen, J. Hu, E. Phipps, A. Prokopenko, S. Rajamanickam, C. Siefert, et al. 2014. Towards extreme-scale simulations for low mach fluids with second-generation trilinos. *Parallel processing letters* 24, 04 (2014), 1442005.

[18] M. Naumov. 2011. *Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU.* Technical Report Tech. Rep. NVR-2011.

[19] Kokkos Kernels Home Page. [n.d.]. https://github.com/kokkos/kokkos-kernels. [Online; accessed 2020].

[20] A. Picciau, G. E. Inggs, J. Wickerson, E. C. Kerrigan, and G. A. Constantinides. 2016. Balancing locality and concurrency: Solving sparse triangular systems on GPUs. In *Proceedings of the 23rd IEEE International Conference on High Performance Computing (HiPC)*. 183–192.

[21] Y. Saad. 2003. *Iterative Methods for Sparse Linear Systems* (2 ed.). SIAM, Philadelpha, PA.

[22] J. H. Saltz. 1990. Aggregation methods for solving sparse triangular systems on multiprocessors. *SIAM J. Sci. Comput.* 11 (1990), 123–144.

[23] B. Suchoski, C. Severn, M. Shantharam, and P. Raghavan. 2012. Adapting sparse triangular solution to GPUs. In *Proceedings of the 41st International Conference on Parallel Processing Workshops*. 140—-148.

[24] Sierra Structural Dynamics Development Team. 2017. *Sierra Structural Dynamics–User's Notes.* Technical Report SAND2018-2449.