symPACK: A GPU-Capable Fan-Out Sparse Cholesky Solver

Julian Bellavita
Lawrence Berkeley National Lab
Berkeley, California, USA
jb2695@cornell.edu

Mathias Jacquelin
Cerebras Systems Inc.
Sunnyvale, California, USA
mathias.jacquelin@cerebras.net

Esmond G. Ng
Lawrence Berkeley National Lab
Berkeley, California, USA
egng@lbl.gov

Dan Bonachea
Lawrence Berkeley National Lab
Berkeley, California, USA
dobonachea@lbl.gov

Johnny Corbino
Lawrence Berkeley National Lab
Berkeley, California, USA
jcorbino@lbl.gov

Paul H. Hargrove
Lawrence Berkeley National Lab
Berkeley, California, USA
phhargrove@lbl.gov

ABSTRACT

Sparse symmetric positive definite systems of equations are ubiquitous in scientific workloads and applications. Parallel sparse Cholesky factorization is the method of choice for solving such linear systems. Therefore, the development of parallel sparse Cholesky codes that can efficiently run on today's large-scale heterogeneous distributed-memory platforms is of vital importance. Modern supercomputers offer nodes that contain a mix of CPUs and GPUs. To fully utilize the computing power of these nodes, scientific codes must be adapted to offload expensive computations to GPUs.

We present symPACK, a GPU-capable parallel sparse Cholesky solver that uses one-sided communication primitives and remote procedure calls provided by the UPC++ library. We also utilize the UPC++ "memory kinds" feature to enable efficient communication of GPU-resident data. We show that on a number of large problems, symPACK outperforms comparable state-of-the-art GPU-capable Cholesky factorization codes by up to 14x on the NERSC Perlmutter supercomputer.

CCS CONCEPTS

• **Mathematics of computing** → **Solvers**; • **Computing methodologies** → **Linear algebra algorithms**; **Hybrid symbolic-numeric methods**; • **Software and its engineering** → *Parallel programming languages*; *Distributed programming languages*; **Software libraries and repositories**.

ACM Reference Format:
Julian Bellavita, Mathias Jacquelin, Esmond G. Ng, Dan Bonachea, Johnny Corbino, and Paul H. Hargrove. 2023. symPACK: A GPU-Capable Fan-Out Sparse Cholesky Solver. In *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023), November 12–17, 2023, Denver, CO, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3624062.3624600

1 INTRODUCTION

Large symmetric positive definite systems of linear equations arise in the solution of many scientific and engineering problems. Such linear systems are solved using Cholesky factorization. Efficient Cholesky factorization is therefore important for the overall performance of scientific application codes. Modern supercomputers have nodes that contain a mix of GPUs and CPUs, and it is critical for scientific codes to effectively utilize the heterogeneous computing resources of modern HPC nodes.

In this paper, we present symPACK, a GPU-capable parallel sparse Cholesky solver that utilizes an *asynchronous task paradigm* and one-sided communication functionality offered by the partitioned global address space (PGAS) library UPC++ [5]. By using a task-based formalism and dynamic scheduling techniques within a node, symPACK achieves good strong scaling on modern supercomputers. symPACK offloads sufficiently large computations to a GPU and uses the UPC++ "memory kinds" feature to streamline communication of GPU-resident data. This enables symPACK to achieve high performance and effectively take advantage of heterogeneous computing resources, such as the GPU nodes on the Perlmutter [20] supercomputer at NERSC [19].

An outline of the paper is as follows. In Section 2, we provide some background on sparse Cholesky factorization. In Section 3, we describe our implementation of symPACK. The GPU functionality of symPACK is described in Section 4. Some numerical results are presented in Section 5. We end with some directions for future work in Section 6 and concluding remarks in Section 7.

2 BACKGROUND ON CHOLESKY FACTORIZATION

In the following, we give some background on Cholesky factorization and on how symmetry and sparsity can be taken into account. We first review the basic Cholesky algorithm for dense matrices and then detail how it can be modified to handle sparse matrices efficiently. We also present fundamental notions on sparse matrix computations before reviewing the work related to sparse Cholesky factorization.

2.1 The basic algorithms

Let $\mathbf{A} = [a_{i,j}]$ be an n-by-n symmetric positive definite matrix. The Cholesky algorithm factors the matrix $\mathbf{A}$ into

$$\mathbf{A} = \mathbf{L}\mathbf{L}^T, \tag{1}$$

where $\mathbf{L} = [\ell_{i,j}]$ is a lower-triangular matrix, and $\mathbf{L}^T$ is the transpose of $\mathbf{L}$ and is upper-triangular. The factorization thus allows symmetry to be exploited, since only $\mathbf{L}$ needs to be computed and saved.

PAW-ATM'23, November 13, 2023, Denver, CO, USA

The basic Cholesky factorization algorithm, given in Alg. 1, can be described as follows:

(1) Current column j of $\mathbf{L}$ is computed using column j of $\mathbf{A}$.
(2) Column j of $\mathbf{L}$ is used to update the remaining columns of $\mathbf{A}$.

If $\mathbf{A}$ is a dense matrix, then every column k, $k > j$, is updated.

Once the factorization is computed, the solution to the original linear system ($\mathbf{A}x = b$) can be obtained by solving two triangular linear systems using the Cholesky factor $\mathbf{L}$ like so:

$$\mathbf{L}y = b, \mathbf{L}^T x = y \tag{2}$$

for *column* $j = 1$ *to* n **do**
  $\ell_{j,j} = \sqrt{a_{j,j}}$
  for *row* $i = j + 1$ *to* n **do**
    $\ell_{i,j} = a_{i,j}/\ell_{j,j}$
  end

  for *column* $k = j + 1$ *to* n **do**
    for *row* $i = k$ *to* n **do**
      $a_{i,k} = a_{i,k} - \ell_{i,j} \cdot \ell_{k,j}$
    end
  end
end

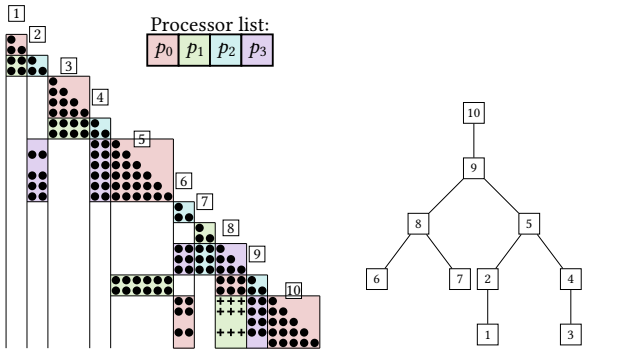
Algorithm 1: Basic Cholesky algorithm

2.2 Cholesky factorization of sparse matrices

For large-scale applications, $\mathbf{A}$ is often *sparse*, i.e., most of the entries of $\mathbf{A}$ are zero. It is well known that Cholesky factorization of a sparse matrix creates fill, i.e., some of the zero entries will become nonzero. One can reduce the memory and flops required to store and compute $\mathbf{L}$ by only performing computations involving nonzero elements. Techniques for controlling fill using reordering and efficient sparse Cholesky factorization can be found in [11]. Here we mention two components in sparse Cholesky factorization that are important in our discussion in this paper.

An important observation in sparse Cholesky factorization is that the columns of $\mathbf{L}$ will become denser and denser as the factorization proceeds from the left to the right. It is not uncommon to find groups of consecutive columns, referred to as *supernodes*, that share essentially the same zero-nonzero structure. If columns $i, i+1, \cdots, j$ form a supernode, then the diagonal block of these columns will be completely dense, and each row k, $j + 1 \leq k \leq n$, within the supernode is either entirely zero or entirely nonzero. Supernodes are further partitioned into blocks, which are collections of contiguous rows in a supernode that form a dense submatrix. *supernodes* and blocks of a sample symmetric matrix are depicted in Figure 1a.

The *elimination tree* of $\mathbf{L}$ is an important tool in sparse Cholesky factorization. It is a directed acyclic graph that has n vertices $\{v_1, v_2, \cdots, v_n\}$, with v_i corresponding to column i of $\mathbf{A}$. Suppose $i > j$. There is an edge between v_i and v_j in the elimination tree if and only if ℓ_{ij} is the *first* off-diagonal nonzero entry in column j of $\mathbf{L}$. Thus, v_i is called the *parent* of v_j and v_j is a *child* of v_i. The elimination tree contains useful information regarding the sparsity structure of $\mathbf{L}$ and dependencies among the columns of $\mathbf{L}$. This information can be used to construct a task graph for the factorization of $\mathbf{A}$. Such a task graph is typically composed of tasks that compute columns of $\mathbf{L}$ and tasks that update columns of $\mathbf{A}$. See [18] for details. By collapsing the columns of a supernode in the elimination tree into a single vertex, one gets the *supernodal elimination tree*. An example of such a tree is depicted in Figure 1b.



(a) Structure of Cholesky factor L (b) Supernodal elimination tree of matrix A

Figure 1: Sparse matrix A partitioned into supernodes and dense blocks. i denotes the i-th supernode, • represents original nonzero elements in A, while + denotes fill-in entries. Colors correspond to the four distributed-memory nodes onto which blocks are mapped in a 2D block-cyclic way.

2.3 Parallel sparse Cholesky factorization

In the following, we discuss the scheduling of computation in the numerical factorization. The only constraints that have to be respected are the numerical dependencies among the columns: column k of $\mathbf{A}$ has to be updated by column j of $\mathbf{L}$, for any $j < k$ such that $\ell_{k,j} \neq 0$, but the order in which the updates occur is mathematically irrelevant, as long as the updates are performed before column k of $\mathbf{A}$ is factored. There is therefore significant freedom in the scheduling of computational tasks that factorization algorithms can exploit.

On sequential platforms, this has led to two well-known variants of the Cholesky factorization algorithm: *left-looking* and *right-looking* schemes, which have been introduced in the context of dense linear algebra [9]. In the *left-looking* algorithm, before column k of $\mathbf{A}$ is factored, all updates coming from columns i of $\mathbf{L}$ such that $i < k$ and $\ell_{k,i} \neq 0$ are first applied. In that sense, the algorithm is "looking to the left" of column k. In *right-looking*, after a column k has been factored, every column i such that $k < i$ and $\ell_{i,k} \neq 0$ is updated by column k. The algorithm thus "looks to the right" of column k.

Distributed memory platforms add the question of where the computations are going to be performed. Various parallel algorithms have been proposed in the literature for Cholesky factorization, such as `MUMPS` [1], which is based on the multifrontal approach (a variant of right-looking), and `PaStiX` [14], which is right-looking.

In [2], the author classifies parallel Cholesky algorithms into three families: *fan-in*, *fan-out* and *fan-both*.

The *fan-in* family includes all algorithms such that all updates from a column k to other columns i, for $k < i$ such that $\ell_{i,k} \neq 0$, are computed on the processor owning column k. When one of these columns, say i, will be factored, the processor owning i will have to "fan-in" (or collect) updates from previous columns.

The *fan-out* family includes algorithms that compute updates from column k to columns i, for $k < i$ such that $\ell_{k,i} \neq 0$, on processors owning columns i. This means that the processor owning column k has to "fan-out" (or broadcast) column k of the Cholesky factor.

The *fan-both* family generalizes these two families to allow these updates to be performed on any processor. This family relies on *computation maps* to map computations to processors.

In a *fan-both* algorithm, two kinds of messages can be exchanged throughout the factorization: *factors* and *aggregate vectors*. The first type of message corresponds to the entries in a column after it has been factorized, or in other words, to a portion of the output data of the algorithm. The second type of message is a temporary buffer in which a given p_{source} will accumulate all its updates to a remote target column residing on p_{target}.

3 SYMPACK IMPLEMENTATION

As mentioned in the previous section, there are many ways to schedule the computations as long as the precedence constraints are satisfied. The Cholesky factorization in symPACK is inspired by the *fan-out* algorithm. This section covers the symPACK factorization algorithm in detail.

3.1 Supernode and Block Partitioning

symPACK's symbolic factorization phase analyzes the sparsity structure of **L** in order to partition **A**'s columns into supernodes. After this is done, the rows of each supernode are partitioned so that they form *blocks*. Each block is essentially a dense submatrix of a supernode. The primary advantage of this additional step is that it allows computations to be performed via dense matrix-matrix operations on each block, meaning one can take advantage of the high performance offered by BLAS 3 and LAPACK routines. Block partitioning also allows a 2D block-cyclic mapping of tasks onto processors, which is expanded upon in Section 3.3.

The exact manner in which supernodes are partitioned into blocks is as follows. For the jth supernode, we first create a diagonal block denoted as $B_{j,j}$, which is composed of all rows of the supernode that contain a diagonal element. Let S denote a set that contains the indices of all such rows in **A**. Then, for supernode k such that $k < j$, if supernode k contains a nonzero element in any row $i \in S$, then rows $[min(S) : max(S)]$ form a block in supernode k denoted as $B_{j,k}$. Another way to think about this scheme is that for a block $B_{j,k}$, k denotes the supernode that the block is located in, and j denotes the supernode that contains the diagonal entries of the rows of the block. The block partitioning process is summarized in Algorithm 2, and Figure 1a shows an example block partitioning.

Input: supernodes[], N
Output: blocks $B_{*,*}$
for $j \leftarrow 1$ **to** N **do**
  $B_{j,j} \leftarrow []$
  $S \leftarrow \{\}$
  for $row \leftarrow 1$ **to** $size(supernodes[j])$ **do**
    $B_{j,j}$.append(supernodes[j][row])
    S.add(row)
  end
  for $k \leftarrow 1$ **to** $j-1$ **do**
    if $supernodes[k][i] \neq 0$ ***for any*** $i \in S$ **then**
      minRow $\leftarrow$ min(S)
      maxRow $\leftarrow$ max(S)
      $B_{j,k}$.append(supernodes[k][minRow : maxRow])
    end
  end
end

Algorithm 2: Partition N Supernodes into Blocks

3.2 Task-based formulation

symPACK uses three types of tasks during its numerical factorization phase: *diagonal factorization*, *factorization*, and *update*. Each task operates on a single block of a supernode. We let **A** be an n-by-n symmetric positive definite matrix, and denote these tasks using the following notation[1]:

- *Diagonal Factorization* D_i: factorize the diagonal block located in the ith supernode $B_{i,i}$ using the LAPACK routine POTRF.
- *Factorization* $F_{i,j}$: factorize block $B_{i,j}$ using the BLAS 3 routine TRSM to solve the equation $B_{i,j}L_{i,j} = L_{j,j}$, where $L_{i,j}$ denotes the block of the Cholesky factor corresponding to $B_{i,j}$ and $L_{j,j}$ is the diagonal block of the Cholesky factor located in the jth supernode.
- *Update* $U_{i,j,k}$: compute updates to block $B_{i,k}$ from block $B_{i,j}$, where $k > j$. If $B_{i,k}$ is located on the diagonal of **A**, this update is done using the BLAS 3 routine SYRK, otherwise the BLAS 3 routine GEMM is used.

These tasks cannot be scheduled arbitrarily. Before computing column k of the Cholesky factor **L**, all updates from the columns of **L** to the left of k must be applied to column k of **A**. Additionally, the diagonal of a column must be factorized before any other elements in that column can be factorized. When applied to a supernodal blocking scheme and the task descriptions provided earlier, this logic sets up the following dependencies between tasks:

- The Diagonal Factorization task D_i must be completed before Factorization tasks of the form $F_{*,i}$ can be scheduled. This is because the TRSM operation that is used to compute off-diagonal factorized blocks uses the factorized diagonal block as the RHS of the linear system.
- All Factorization tasks $F_{i,j}$ must be completed before Update tasks of the form $U_{i,j,*}$ can be computed. In other words, a block must be factorized before it can be used to update other blocks.
- All Update tasks of the form $U_{i,*,k}$ must be completed before Factorization tasks of the form $F_{i,k}$ can be computed. In other words, all possible updates that can be applied to a block must be applied before a block can be factorized. Following

[1] We use MATLAB notation in this paper.

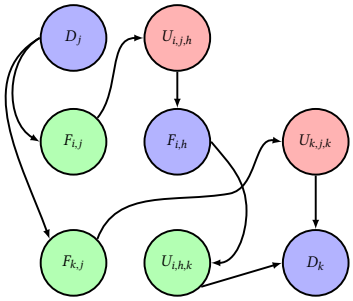


Figure 2: *fan-out* task dependencies for four columns j, i, k, and h

the same logic, all Update tasks of the form $U_{j,*,j}$ must be completed before Diagonal Factorization tasks of the form D_j can be computed.

An example of dependencies between Diagonal Factorization, Factorization, and Update tasks can be seen in Figure 2. symPACK derives a task graph of this form for a given problem using **L**'s supernodal elimination tree and proceeds to partition it among the given processors in a manner described in the following section.

3.3 Parallel algorithm

We now describe how a given task graph is partitioned among the P processes available in a distributed-memory environment. symPACK maps blocks to processes using a 2D block-cyclic distribution. Such a distribution has the advantage of reducing the presence of serial bottlenecks, as a 1D row or column cyclic distribution would assign excessive work to each process. Let P_k denote the kth process in our distributed-memory environment.

Recall that each block is denoted as $B_{i,j}$, where j is the supernode that the block is contained within, and i is the supernode that contains the diagonal entries of the rows within the block. $B_{i,j}$ is mapped to a process using the $map(i,j)$ function, which maps the block coordinates (i,j) to a process identifier P_k following a 2D block-cyclic distribution.

Henceforth, we will use $P_{map(i,j)}$ to denote the process that $B_{i,j}$ is mapped to. $P_{map(i,j)}$ is responsible for all local computation involving $B_{i,j}$, meaning it must factorize $B_{i,j}$ and compute updates to $B_{i,j}$. Let T_k denote the set of tasks assigned to P_k. T_k is then defined according to the following:

$$T_k := \{F_{i,j} | map(i,j) = k\} \cup \{U_{i,*,l} | map(i,l) = k\} \cup \{D_i | map(i,i) = k\}$$

We now turn our attention to describing the communication required in this scheme. Two sorts of messages are required: factorized off-diagonal blocks of L, denoted $L_{i,j}$ where $i \neq j$, and factorized diagonal blocks of L, denoted $L_{j,j}$.

Off-diagonal factorized blocks are needed for processes to compute Update tasks involving said off-diagonal blocks, but it is not always the case that the necessary factorized block is local to the process computing the Update task. In this case, the process that owns the factorized block will send it to the block performing the update. More formally, let $P_{F_{(i,j)}}$ denote the set of processes that the factorized block computed by task $F_{(i,j)}$ will be sent to. $P_{F_{(i,j)}}$ is defined as follows:

$$P_{F_{(i,j)}} := \{map(i,k) \neq map(i,j) | U_{i,j,k} \text{ is an Update task}\}$$

Diagonal factorized blocks $L_{j,j}$ are used to compute all off-diagonal factorized blocks $L_{i,j}$ in supernode j. Therefore, if $L_{i,j}$ resides on a different process than $L_{j,j}$, then $L_{j,j}$ must be sent to the process owning $L_{i,j}$. Let P_{D_i} denote the set of processes that the factorized diagonal block computed by task D_i must be sent to:

$$P_{D_i} := \{map(i,j) \neq map(i,i) | F_{i,j} \text{ is a Factorize task}\}$$

The remainder of this section concentrates on our communication paradigm, implemented with functionality provided by the UPC++ library.

3.4 Communication Paradigm

symPACK uses the following data structures, where each process has:

- a *local task queue* (LTQ), containing all the tasks statically mapped onto this process and awaiting execution,
- a *ready task queue* (RTQ), containing all the tasks for which precedence constraints have been satisfied and that can therefore be processed.

This is illustrated in Figure 3.

A task $T_{s,t}$ is represented by a *source* block s and a *target* block t on which computations have to be applied. Each task also has an incoming dependency counter, initially set to the number of incoming edges in the task graph.

Similarly, a message $M_{s,t}$ exchanged to satisfy the dependence between tasks mapped onto distinct processes is labeled by the *source* block s of the sending task and the *target* block t of the receiving task.

The overall mechanism that we propose is the following: whenever a task is completed, processes owning dependent tasks are notified that new input data is now available. As soon as a process is done with its current computation, it periodically handles each incoming notification by issuing a corresponding one-sided RMA *get* to retrieve the relevant data. This communication proceeds asynchronously, and the incoming dependency counter of the corresponding task is decremented when the non-blocking communication later reaches completion.

When a task from the LTQ has all its dependencies satisfied (i.e. when its dependency counter reaches zero) then it is moved to the RTQ, and is now ready for execution. The process then picks a task from the RTQ and executes it. If multiple tasks are available in the RTQ, then the next task that will be processed is whichever one is at the top of the queue. Evaluating different scheduling policies will be a subject for future work.

We use the UPC++ PGAS library [5, 26] for communicating between distributed-memory compute nodes. UPC++ is built atop the GASNet-EX [6, 12] communication library, and introduces several parallel programming features useful for our implementation.

First, it provides *global pointers* for referencing memory locations on remote processes. Using the `upcxx::rget()` and `upcxx::rput()` functions, one can perform Remote Memory Access (RMA) that

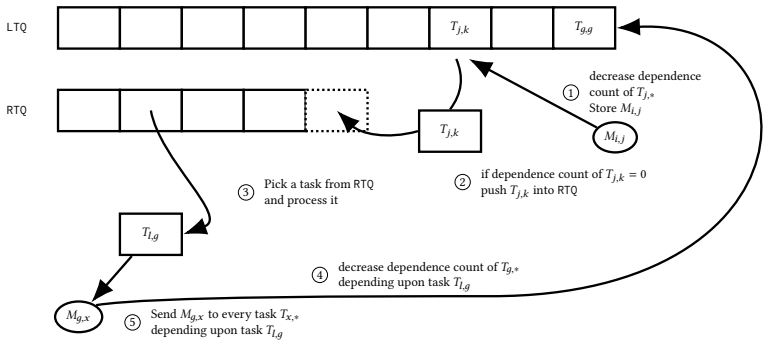


Figure 3: Task scheduling in symPACK. Update of task dependence in LTQ. Dynamic scheduling of local tasks in RTQ.

transfers the data between two processes in a one-sided way. Moreover, on modern HPC networks these transfers are handled by RDMA offload hardware in the NIC and are generally performed without any involvement from the remote processor.

Another useful feature is the ability to perform *remote procedure calls* (RPC), whereby a process can submit a function for execution on a remote process. This gets pushed into a queue that the remote process executes when calling the UPC++ function `upcxx::progress()`.

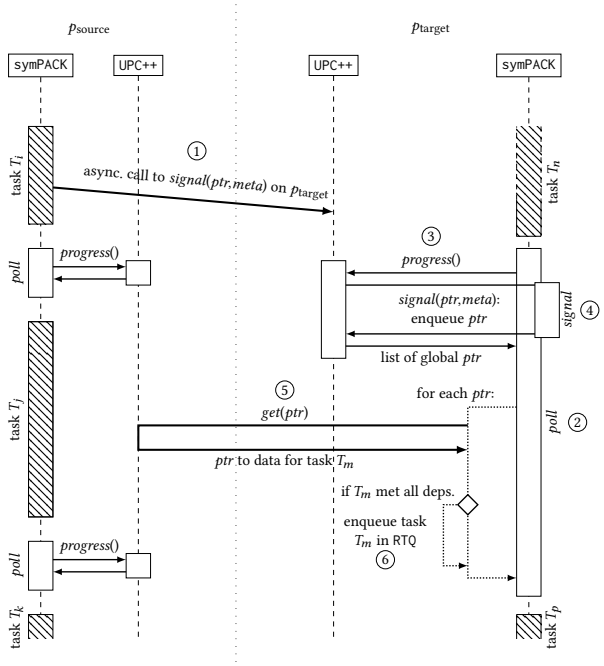


Figure 4: Data exchange protocol in symPACK. Notifications are performed using UPC++ asynchronous RPC, and the actual data is fetched using non-blocking one-sided RMA get.

symPACK uses a data notification and communication algorithm which is heavily based on these two features of UPC++. This algorithm is depicted in Figure 4, where we suppose that at the end of a computation, p_{source} has produced some data that needs to be sent out to p_{target}.

First, p_{source} notifies p_{target} that some data has been produced by sending it a global pointer *ptr* to the data along with some meta-data *meta*. This is done by performing an RPC to a *signal*(*ptr*,*meta*) function on p_{target} directly from p_{source}, and is referred to as step 1 in the diagram.

When p_{target} finishes its current computation, it calls a *poll* function (step 2), whose main role is to watch for incoming communications and do the bookkeeping of task dependencies. This function dispatches to the `upcxx::progress()` function to execute all enqueued RPCs to the *signal*(*ptr*,*meta*) function, which enqueues *ptr* and *meta* into a list. This corresponds to steps 3 and 4. The next step in the *poll* function is to go through that list of global pointers and issue a UPC++ RMA get operation to pull the data to the local host or local GPU memory, as appropriate for where the dependent computation will be scheduled (step 5). Once the RMA get operation is completed, the *poll* function updates the dependencies of every task T_m that will be using this data (which can be found by looking at the meta-data *meta*). If all dependencies of a task are met, that task is moved into the RTQ (at step 6). Finally, p_{target} resumes its work by selecting a task from RTQ.

In summary, symPACK functions by scheduling tasks for dense supernode blocks mapped to processes according to a 2D block-cyclic distribution. BLAS and LAPACK calls are used to perform local computation, and the communication necessary to satisfy dependencies between tasks is done via a combination of RPCs that enqueue global pointers to remote nodes into a list and UPC++ RMA get operations that fetch data from said global pointers. Once a task's dependencies are satisfied, it is placed on the RTQ, where it can be scheduled and executed. This process is repeated until all tasks on all processes have been executed, at which point the factorization is complete.

4 GPU FUNCTIONALITY

Modern distributed-memory HPC systems typically offer nodes that include both CPUs and GPUs. In certain cases, GPUs are able to achieve raw arithmetic performance far superior to that of CPUs. This is thanks to GPU architectures naturally supporting extremely high levels of parallelism. Thus, GPUs are crucial tools with which we can achieve better performance for computationally intensive scientific applications. symPACK offers GPU functionality, wherein

certain computations are offloaded to a GPU for faster execution. This section describes the GPU mode of symPACK.

4.1 Memory Kinds

While introducing support for GPU operations, symPACK utilized a relatively new feature of UPC++ known as 'memory kinds'. Memory kinds allow the user to allocate buffers on devices using a `upcxx::device_allocator` object, which produces a global pointer to the allocated device memory that behaves similarly to a typical global pointer to host memory. Memory kinds make it possible to use UPC++ RMA operations to communicate using device memory in the same way one would handle communication involving host memory. A UPC++ programmer can use the `upcxx::copy()` function to uniformly move data between host or device memories located anywhere in the system, in a device-agnostic manner. A particularly significant benefit of memory kinds is the ability to move data directly between host memory that resides on one compute node and device memory that resides on a different node separated by a network, without the need for any intermediate copies staged through host memory. For distributed applications that involve many GPUs scattered across different nodes, this feature can significantly decrease the amount of communication overhead in the application.

Memory kinds are also highly portable. By using C++ templates to specify the device kind (e.g. `cuda_device`, `hip_device`) where one wishes to allocate memory, it is possible to use the same UPC++ code to interact with devices from different vendors. Code that is written to communicate data resident on NVIDIA GPUs can be modified to run on AMD GPUs by simply changing a template parameter, or by using a wildcard parameter. For more information on UPC++ memory kinds, the reader is referred to [4, 7, 27].

4.2 GPU Mode Functionality

When symPACK is initialized, each UPC++ process creates a device allocator object using the `upcxx::make_gpu_allocator()` function. Although symPACK is not locked into any particular binding of processes to devices, a recommended scheme is to bind processes to devices in a cyclic manner. Using this strategy, in a node with d devices, process p will be bound to device $p \ mod(d)$. All processes mapped to a given device allocate an equal portion of memory on the device.

As explained earlier, the computation in symPACK is implemented using calls to the BLAS routines GEMM, SYRK, and TRSM. Additionally, the LAPACK routine POTRF is used to factorize blocks located on the diagonal of $\mathbf{A}$. High-performance implementations of these routines for NVIDIA GPUs are available via the CuBLAS [21] and CuSolver [22] libraries. symPACK uses these libraries to perform the needed BLAS and LAPACK routines on the GPU.

The dimensions of different blocks in a supernode can vary dramatically, and as a result, the sizes of the buffers passed into the computation routines can also vary. There are overheads associated with using a CUDA kernel to perform computation, which include both the overheads of invoking and synchronizing the computational kernel and any overheads of moving data to and from the device. Because the former overheads are significant and relatively insensitive to problem size, the net performance benefit achievable using GPUs is more significant for larger computations. Therefore, it is most advantageous to utilize the GPU exclusively for computations involving adequately large buffers.

The GPU mode of symPACK uses a simple heuristic based on buffer size to determine if a computation should be offloaded to the GPU or not. If a given BLAS/LAPACK routine involves a sufficiently large buffer, data is moved from host memory to device memory using `upcxx::copy()`, and the corresponding GPU version of the matrix computation is invoked. If the buffers involved in a BLAS/LAPACK routine are small, then the CPU is used for the computation instead. Each operation has a different size threshold that determines when to offload the operation to the GPU. Different thresholds for each operation are necessary because each operation has a different non-asymptotic arithmetic intensity. These thresholds have default values that were determined via a simple brute-force manual tuning effort, but since the optimal values of these thresholds are likely to vary for different hardware, symPACK also allows the user to specify each threshold manually. These size thresholds ensure that the GPU is used only when doing so is expected to be beneficial to performance. If the GPU were used for every computation, the fixed overheads arising from many invocations of CUDA kernels on small buffers would eliminate the performance gains obtained from using the GPU for large computations. Thus, the GPU functionality is not a 'GPU-only' algorithm; it is instead a hybrid algorithm that takes advantage of the unique processing capabilities that each type of hardware offers.

As described earlier, memory kinds allow the programmer to move data between a host and a device that reside on different nodes. symPACK takes advantage of this feature when sending factorized diagonal blocks to remote nodes. If a factorized diagonal block is sufficiently large and is to be sent to a remote node, it could naively be fetched into host memory using `upcxx::rget()` followed by a use of `upcxx::copy()` to move the block to the device. However, since memory kinds make it possible to move data directly to a device on a remote node, placing the block in remote host memory before moving it to the device is unnecessary. Therefore, large-enough factorized diagonal blocks are marked as 'GPU blocks' and are copied directly to a GPU-resident buffer on the remote node via `upcxx::copy()`. This eliminates the need for the remote process to use `upcxx::rget()` to fetch the data into host memory before moving it to the device.

To handle scenarios where there is insufficient device memory to accommodate a given computation, symPACK provides multiple "fallback options" that allow the factorization to proceed. The default behavior is to simply perform the computation on the CPU. However, for certain problems, a user may prefer to instantly terminate the factorization and rerun it with more device memory allocated to each process. For this reason, symPACK also provides a fallback option that will throw an exception if a device allocation fails due to insufficient memory.

While our implementation of GPU operations for symPACK currently only supports NVIDIA GPUs, it would be relatively easy to introduce support for AMD or Intel GPUs, thanks to the portability offered by UPC++ memory kinds. One would only need to write a short function that initializes library handlers, and then replace the calls to CuBLAS/CuSolver with calls to the vendor equivalents of these libraries.

5 PERFORMANCE EVALUATION

In this section, we present the performance of the sparse Cholesky factorization and triangular solve routines implemented in symPACK. Our experiments were conducted on the GPU partition of the NERSC Perlmutter supercomputer [20]. Each Perlmutter GPU node contains one AMD EPYC 7763 "Milan" CPU with 64 cores, four NVIDIA A100 "Ampere" GPUs, and four HPE Slingshot 11 "Cassini" 200Gbps network cards connected to a 3-hop dragonfly network fabric (see §A.2.2 for details).

We evaluate the performance of symPACK using a set of matrices from the SuiteSparse Matrix Collection [8]. A description of each matrix can be found in Table 1. The Flan_1565 and boneS10 matrices were chosen because of their size, and thermal2 was chosen because of its irregular sparsity structure and its high level of sparsity. We chose to focus primarily on large problems for our experiments because large-scale multi-node runs are typically only done on large matrices. In this paper, we analyze the performance of symPACK in a distributed-memory setting only. In addition, all experiments are conducted without multi-threading (which is commonly referred to as "flat-MPI"). In our experiments, a fill-reducing ordering computed using Scotch [23] is applied to the original matrix in order to reduce the number of fill-in entries in **L**. The Scotch library contains an implementation of the nested dissection algorithm [10] to compute a permutation that reduces the number of fill-in entries in the Cholesky factor.

Matrices from SuiteSparse matrix collection			
Name	Description	n	nnz
Flan_1565	3D model of a steel flange	1,564,794	114,165,372
boneS10	3D trabecular bone	914,898	40,878,708
thermal2	steady state thermal	1,228,045	8,580,313

Table 1: Characteristics of symmetric matrices used in the experiments. n denotes the number of rows/columns in the matrix, and nnz denotes the number of nonzero elements in the matrix.

5.1 Impact of Memory Kinds

Use of memory kinds in RMA operations informs UPC++ (and the underlying GASNet-EX communication layer) that a given buffer resides in memory associated with a particular device type, allowing the library to ensure that the most efficient access methods are used for communication. The current GASNet-EX release (v2023.3.0) includes accelerated memory kinds support for platforms including GPUs from NVIDIA or AMD paired with Mellanox InfiniBand or HPE Slingshot network hardware. When pairing a supported network with an NVIDIA GPU, GASNet-EX can utilize the technology known as "GPUDirect RDMA" (GDR), which enables the network adapter to directly access the GPU memory (such as for RMA puts and gets) without the need to interrupt the CPU or use host memory to stage the transfer through any intermediate buffers. This zero-copy capability yields a significant acceleration of eligible transfers.

Fig. 5 shows the bandwidth of `upcxx::copy()` on NERSC Perlmutter for one particular RMA transfer variant across various payload sizes on a log-log plot. The figure compares performance of UPC++ using a "Reference" implementation of memory kinds

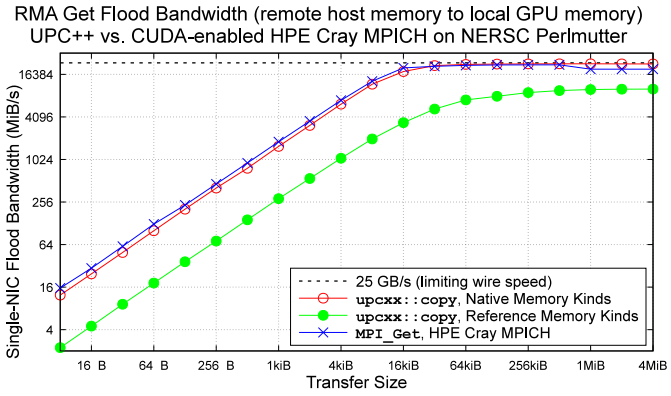


Figure 5: Microbenchmark comparison of one-way point-to-point communication bandwidth for non-blocking RMA gets involving GPU-resident buffers using GPUDirect RDMA technology versus the same transfer staged through an intermediate buffer in host memory.

that stages transfers involving GPU memory through intermediate buffers in host memory, versus the zero-copy GDR-accelerated implementation ("Native" series). An equivalent MPI RMA benchmark using GPU-enabled HPE Cray MPICH is also included for comparison. The results demonstrate that GASNet-EX memory kinds enable substantial improvement in the performance of `upcxx::copy()`; taking it from substantially under-performing relative to the MPI equivalent, to delivering comparable performance. The bandwidth ratio measured between Native and Reference memory kinds ranges from 5.9x (at 8 KiB payload) to 2.3x (for payloads over 1 MiB). The bandwidth gap measured between UPC++ Native memory kinds and MPI is much smaller and within 20% across the entire range of payloads measured.

For more information on UPC++ memory kinds acceleration and resulting performance improvements, the reader is referred to [13, 27].

5.2 Workload distribution between CPUs and GPUs

As mentioned in Section 4, a simple heuristic based on buffer size is used to determine whether a given operation should be computed on the CPU or on the GPU. Figure 6 shows how many operations are performed on each type of hardware for a factorization and triangular solve of the Flan_1565 matrix. These numbers were gathered using the default heuristic parameters on a run with 4 UPC++ processes and 4 GPUs on NERSC Perlmutter. Only data from rank 0 is shown, but the number and distribution of operations is roughly the same across all ranks, so the rank 0 data is generally representative of the entire workload.

For all four operation types, the majority of the operations happen on the CPU, indicating that the majority of the blocks obtained by examining the supernodal factorization of Flan_1565 are small- to medium-sized. The relatively few number of operations involving large blocks are offloaded to the GPU.

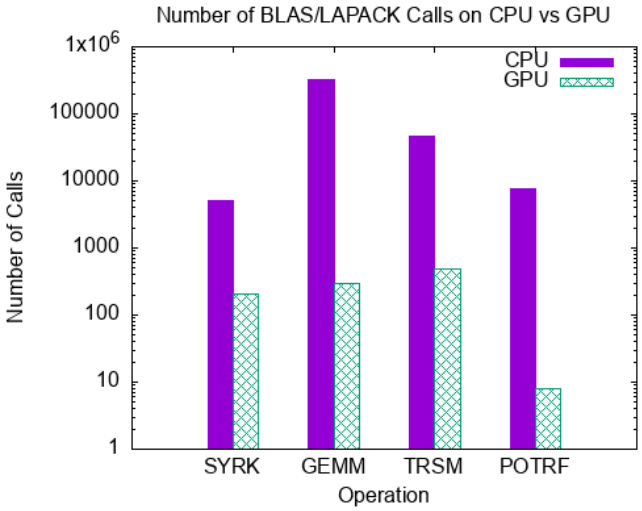


Figure 6: Number of times each operation is performed on the CPU versus on the GPU for a symPACK factorization and solve of the Flan_1565 matrix using 4 UPC++ processes and 4 GPUs. Only data from rank 0 is shown.

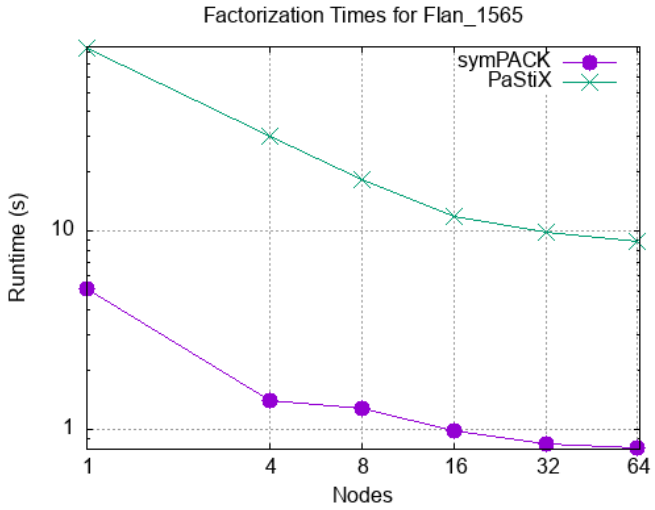


Figure 7: Strong scaling of symPACK's Cholesky factorization on Flan_1565

5.3 Strong scaling comparison of factorization and solve

In the next set of experiments, we evaluate the strong scaling of symPACK's GPU functionality on a set of large problems. We compare its performance to the GPU functionality of PaStiX 6.2.2 [14], a state-of-the-art parallel symmetric solver based on a right-looking supernodal formulation. Another well-known comparable solver is MUMPS [1], but it does not currently offer GPU functionality. Therefore, a comparison with PaStiX is more interesting. The same matrix ordering computed by Scotch is used for both solvers.

PaStiX was built using the runtime scheduler StarPU [3]. PaStiX also offers support for another runtime scheduler, PaRSEC, which we considered. However, we ran into difficulties compiling the PaStiX-specific fork of PaRSEC on Perlmutter, and the PaStiX documentation does not outline any explicit advantages to using PaRSEC over StarPU. Additionally, PaRSEC only works with PaStiX 6.0.2. Therefore, StarPU seemed to be a reasonable choice for our purposes.

Figures 7, 9, and 11 show factorization times for the Flan_1565, boneS10, and thermal2 matrices respectively. For each node count on the x-axis, the data point indicates the best performance achieved by that solver using all of the hardware resources available on that many Perlmutter nodes. For each node count, the experiments were run with a varying number of processes per node, and the result from the run that yielded the best performance for a given node count is reported. For certain problems, symPACK benefits from mapping more than one UPC++ process to each device because using more processes gives each core fewer small- to medium-sized computations to perform on the CPU, which can benefit performance substantially for certain sparsity structures. See the Artifact Description / Artifact Evaluation (AD/AE) for more information.

On each problem and for all node counts measured, symPACK significantly outperforms PaStiX, demonstrating the efficacy of

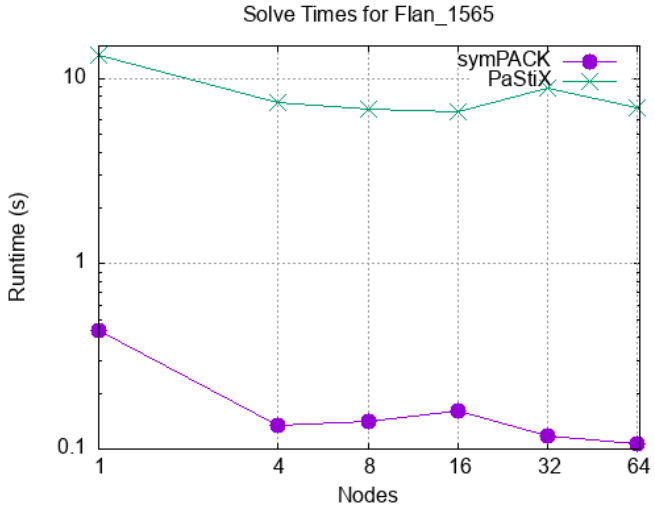


Figure 8: Strong scaling of symPACK's triangular solve on Flan_1565

our communication paradigm, our device offloading heuristic, and our utilization of memory kinds. Although PaStiX appears to exhibit better relative strong scaling that symPACK for some problems, this is likely only because the single node performance of PaStiX is significantly below that of symPACK, and not because of any algorithmic advantages that PaStiX offers. symPACK still exhibits superior absolute performance in all cases. Figures 8, 10, and 12 show solve times for each matrix. As with the factorization phase, symPACK outperforms PaStiX for all problems and node counts measured. Notably, for the thermal2 matrix, PaStiX's solve routine performs worse as the node count increases. This could be in part due to thermal2's highly irregular sparsity structure (for

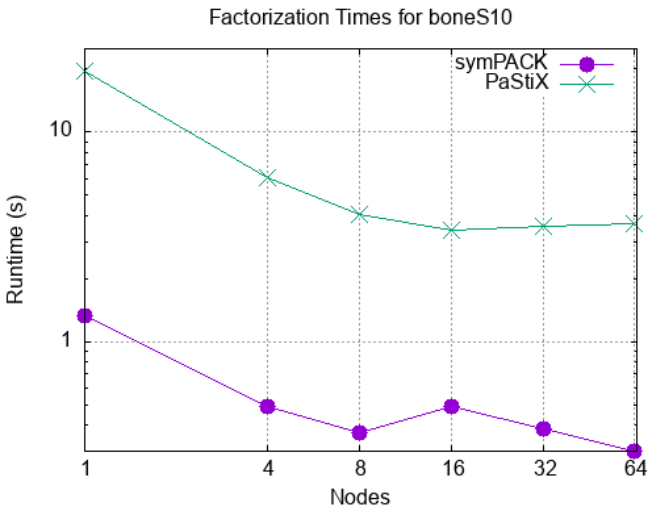


Figure 9: Strong scaling of symPACK's Cholesky factorization on boneS10

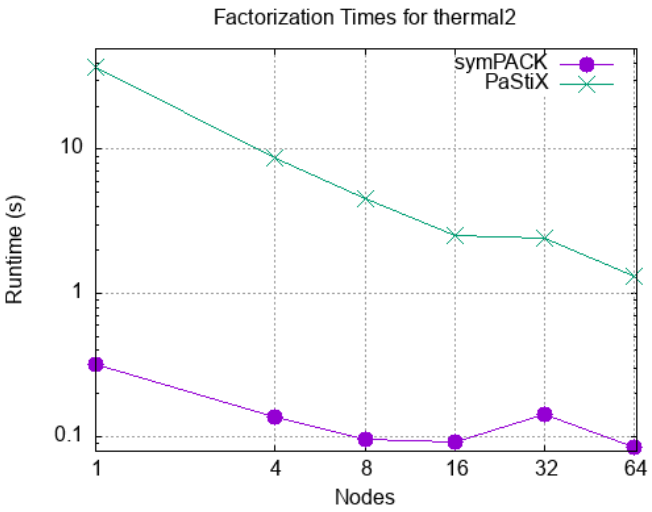


Figure 11: Strong scaling of symPACK's Cholesky factorization on thermal2

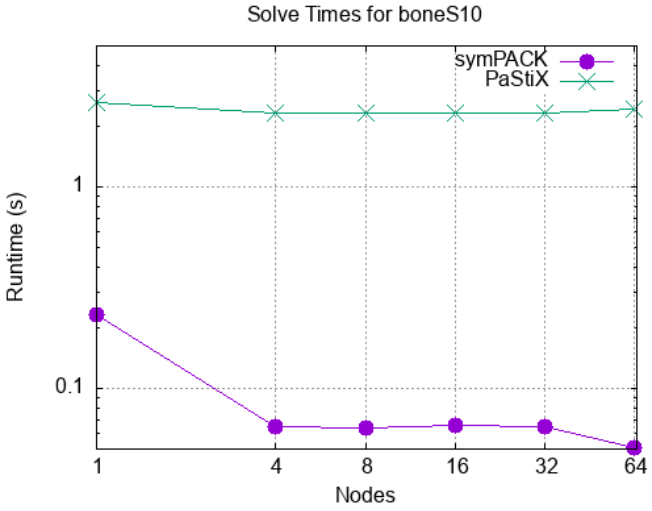


Figure 10: Strong scaling of symPACK's triangular solve on boneS10

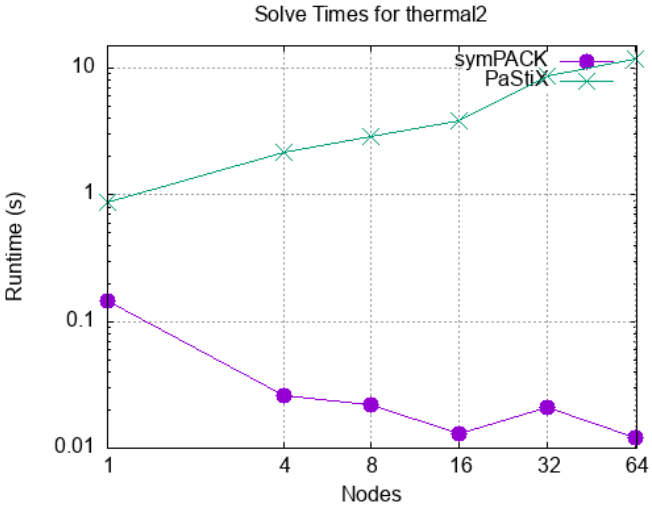


Figure 12: Strong scaling of symPACK's triangular solve on thermal2

details, see [25]). Overall, symPACK demonstrates significant, consistent performance improvements over PaStiX for the Cholesky factorization and triangular solve routines.

The speedups symPACK offers over PaStiX for a single factorization and solve are on the order of seconds. However for an application that needs multiple factorizations in succession, the overall benefit imparted by symPACK could be substantial. One such application is the method for solving eigenvalue problems described in [24]. Another such application is PEXSI [16, 17], a library that can be used for electronic structure calculations and for evaluating specific elements of a matrix inverse without explicitly inverting the matrix.

6 FUTURE WORK

There are several areas where future work is warranted. In order to allow symPACK to encompass a broader range of GPU hardware, it is essential to consider introducing support for AMD and Intel GPUs to complement the currently offered support for NVIDIA GPUs. Additionally, there is a need to conduct experiments with various intra-node scheduling heuristics. Fine-tuning the mechanisms responsible for task distribution and management within a node seems likely to improve performance and efficiency. It is also of interest to benchmark symPACK on a wider variety of input problems. In particular, it will be interesting to see how symPACK performs on smaller problem sizes, as well as on problems with

varying sparsity levels. Lastly, it is worth exploring the development of a hardware-agnostic analytical framework for determining the optimal GPU threshold sizes for each operation, and it is also worth investigating the potential use and benefits of autotuning in this area.

7 CONCLUSION

In this paper, we presented a GPU-capable solver, symPACK, which uses features of the UPC++ PGAS library to implement a novel communication paradigm and optimized data movement strategy for transferring data between hosts and devices on remote nodes. We describe a task-based formalism for Cholesky factorization that considers dense blocks of individual supernodes as the fundamental units of computation. We show that on a number of large problems, including one with an irregular structure, symPACK significantly outperforms a comparable GPU-capable state-of-the-art Cholesky solver by up to 14x, validating the efficacy of our approach.

ACKNOWLEDGMENTS

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, and by the U.S. Department of Energy, Office of Science, Scientific Discovery through Advanced Computing (SciDAC) program under Contract No. DE-AC02-05CH11231 at Lawrence Berkeley National Laboratory.

This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

With permission from the authors, some sections of this paper are based on [15].

REFERENCES

[1] P. Amestoy, I. Duff, J.-Y. L'Excellent, and J. Koster. 2001. A Fully Asynchronous Multifrontal Solver Using Distributed Dynamic Scheduling. *SIAM J. Matrix Anal. and Appl.* 23 (2001), 15–41. https://doi.org/10.1137/S0895479899358194

[2] C Cleveland Ashcraft. 1996. *A taxonomy of column-based Cholesky factorizations.* Ph. D. Dissertation. Yale University.

[3] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2011. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *CCPE - Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009* 23 (Feb. 2011), 187–198. Issue 2. https://doi.org/10.1002/cpe.1631

[4] John Bachan, Scott B. Baden, Dan Bonachea, Johnny Corbino, Max Grossman, Paul H. Hargrove, Steven Hofmeyr, Mathias Jacquelin, Amir Kamil, Brian van Straalen, and Daniel Waters. 2023. *UPC++ v1.0 Programmer's Guide, Revision 2023.3.0.* Technical Report LBNL-2001517. Lawrence Berkeley National Laboratory. https://doi.org/10.25344/S43591

[5] John Bachan, Scott B. Baden, Steven Hofmeyr, Mathias Jacquelin, Amir Kamil, Dan Bonachea, Paul H. Hargrove, and Hadia Ahmed. 2019. UPC++: A High-Performance Communication Framework for Asynchronous Computation. In *Proceedings of the International Parallel & Distributed Processing Symposium (IPDPS)* (Rio de Janeiro, Brazil). IEEE, USA, 11 pages. https://doi.org/10.25344/S4V88H

[6] Dan Bonachea and Paul H. Hargrove. 2018. GASNet-EX: A High-Performance, Portable Communication Library for Exascale. In *Proceedings of Languages and Compilers for Parallel Computing (LCPC'18) (LNCS, Vol. 11882).* Springer, Cham, 138–158. https://doi.org/10.25344/S4QP4W

[7] Dan Bonachea and Amir Kamil. 2023. *UPC++ v1.0 Specification, Revision 2023.3.0.* Technical Report LBNL-2001516. Lawrence Berkeley National Laboratory. https://doi.org/10.25344/S46W2J

[8] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *Transactions on Mathematical Software* 38, 1, Article 1 (Dec 2011), 25 pages. https://doi.org/10.1145/2049662.2049663

[9] Jack J Dongarra, Sven J Hammarling, and Danny C Sorensen. 1987. *LAPACK Working Note #2.* Argonne National Laboratory, Mathematics and Computer Science Division, Technical Memorandum No. 99.

[10] Alan George. 1973. Nested dissection of a regular finite element mesh. *SIAM J. Numer. Anal.* 10, 2 (1973), 345–363. https://doi.org/10.1137/0710032

[11] A. George and J. W-H. Liu. 1981. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice-Hall Inc., Englewood Cliffs, New Jersey.

[12] Paul H. Hargrove and Dan Bonachea. 2022. GASNet-EX RMA Communication Performance on Recent Supercomputing Systems. In *IEEE/ACM Parallel Applications Workshop, Alternatives To MPI+X (PAW-ATM)* (Dallas, TX, USA). eScholarship, California, USA, 7 pages. https://doi.org/10.25344/S40C7D

[13] Paul H. Hargrove, Dan Bonachea, Colin Maclean, and Daniel Waters. 2021. GASNet-EX Memory Kinds: Support for Device Memory in PGAS Programming Models. In *International Conference for High Performance Computing, Networking, Storage, and Analysis* (St. Louis, MO, USA) *(SC).* ACM/IEEE, USA, 6. https://doi.org/10.25344/S4P306

[14] Pascal Hénon, Pierre Ramet, and Jean Roman. 2002. PASTIX: a high-performance parallel direct solver for sparse symmetric positive definite systems. *Parallel Comput.* 28, 2 (2002), 301–321. https://doi.org/10.1016/S0167-8191(01)00141-7

[15] Mathias Jacquelin, Yili Zheng, Esmond Ng, and Katherine Yelick. 2016. An Asynchronous Task-based Fan-Both Sparse Cholesky Solver. arXiv manuscript. https://doi.org/10.48550/arXiv.1608.00044 arXiv:1608.00044 [cs.MS]

[16] L. Lin, M. Chen, C. Yang, and L. He. 2013. Accelerating atomic orbital-based electronic structure calculation via pole expansion and selected inversion. *J. Phys. Condens. Matter* 25, 29 (2013), 14 pages. https://doi.org/10.1088/0953-8984/25/29/295501

[17] L. Lin, J. Lu, L. Ying, R. Car, and W. E. 2009. Fast algorithm for extracting the diagonal of the inverse matrix with application to the electronic structure analysis of metallic systems. *Comm. Math. Sci.* 7, 3 (2009), 755–777. https://doi.org/10.4310/CMS.2009.v7.n3.a12

[18] J. W-H. Liu. 1990. The role of elimination trees in sparse factorization. *SIAM. J. Matrix Anal. & Appl.* 11 (1990), 134–172. https://doi.org/10.1137/0611010

[19] National Energy Research Scientific Computing Center (NERSC). 2023. https://www.nersc.gov

[20] NERSC Perlmutter System Architecture. 2023. https://docs.nersc.gov/systems/perlmutter/architecture/#gpu-nodes

[21] NVIDIA Corporation. 2023. cuBLAS. https://docs.nvidia.com/cuda/cublas.

[22] NVIDIA Corporation. 2023. cuSOLVER. https://docs.nvidia.com/cuda/cusolver.

[23] François Pellegrini and Jean Roman. 1996. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *High-Performance Computing and Networking*, Vol. 1067. Springer, Berlin, Heidelberg, 493–498. https://doi.org/10.1007/3-540-61142-8_588

[24] Tetsuya Sakurai and Hiroshi Sugiura. 2003. A projection method for generalized eigenvalue problems using numerical integration. *J. Comput. Appl. Math.* 159, 1 (2003), 119–128. https://doi.org/10.1016/S0377-0427(03)00565-X

[25] SuiteSparse Matrix Collection. 2023. https://sparse.tamu.edu/.

[26] UPC++ website. 2023. https://upcxx.lbl.gov.

[27] Daniel Waters, Colin A. MacLean, Dan Bonachea, and Paul H. Hargrove. 2021. Demonstrating UPC++/Kokkos Interoperability in a Heat Conduction Simulation. In *IEEE/ACM Parallel Applications Workshop, Alternatives To MPI+X (PAW-ATM)* (St. Louis, MO, USA). eScholarship, California, USA, 5 pages. https://doi.org/10.25344/S4630V

Appendix: ARTIFACT DESCRIPTION / ARTIFACT EVALUATION (AD/AE)

This section describes the methodology used for the experiments presented in this paper, in accordance with the SC23 Reproducibility Initiative.

A.1 Artifact Identification

Title: symPACK: A GPU-Capable Fan-Out Sparse Cholesky Solver

Authors: Julian Bellavita[†], Esmond G. Ng[†], Mathias Jacquelin[*], Dan Bonachea[†], Johnny Corbino[†], Paul H. Hargrove[†]

† Lawrence Berkeley National Laboratory

∗ Cerebras Systems, Inc.

Description: This paper presents `symPACK`, a parallel sparse GPU-capable Cholesky solver built using the `UPC++` PGAS programming model. The primary contribution of the paper is a description and evaluation of the computational artifact presented in this work, namely the `symPACK` solver. The `symPACK` source code was used in combination with its library dependencies to produce the performance data shown in the experiments presented in this paper. Performance results are compared to a baseline established using the `PaStiX` solver running on the same hardware and inputs.

A.2 Reproducibility of Experiments

A.2.1 Software Requirements.

The following artifacts were used to perform the experiments documented in this paper:

- `UPC++` library, version 2022.3.0 or later: https://upcxx.lbl.gov
  - Experiments for the paper used `UPC++` version 2023.3.0
- `symPACK` source code: https://github.com/symPACK/symPACK
  - See commit `9c24de6` on the master branch, also available here: https://doi.org/10.25344/S4PAWATM23_SYMPACK.TAR.GZ
  - The benchmarking program is `driver/run_sympack2D`.
- `PaStiX` 6.2.2 source code: https://solverstack.gitlabpages.inria.fr/pastix/
  - The benchmarking program is `example/simple.c`. The iterative refinement option present in the default version of the benchmarking program was deactivated prior to compilation (comment out line 125).
  - `StarPU` version 1.4.1 was used as the runtime scheduler: https://starpu.gitlabpages.inria.fr/
- `Scotch` version 7.0.3 : https://www.labri.fr/perso/pelegrin/scotch/
- OSU MPI Microbenchmarks, version 7.2 : https://mvapich.cse.ohio-state.edu/benchmarks/

A.2.2 Experimental Platform.

All experiments presented in the paper were run on the Perlmutter supercomputer at NERSC, using Perlmutter's GPU nodes [20]. Here is relevant hardware information for each GPU node:

- Single-socket 64-core 2.45GHz AMD EPYC 7763 "Milan" CPU
- 4x HPE Slingshot 11 NICs
- 256 GB of DDR4 DRAM
- 4x NVIDIA A100 (Ampere) GPUs

The following system modules were loaded on Perlmutter during execution of the experiments:

- `contrib upcxx-cuda/2023.3.0`
- `craype-x86-milan`
- `craype-network-ofi`
- `libfabric/1.15.2.0`
- `PrgEnv-gnu/8.3.3`
- `cray-libsci/23.02.1.1`
- `cray-dsmml/0.2.2`
- `cray-mpich/8.1.25`
- `craype/2.7.20`
- `gcc/11.2.0`
- `cpe/23.03`
- `xalt/2.10.2`
- `cudatoolkit/11.7`
- `craype-accel-nvidia80`
- `gpu/1.0`

A.2.3 Memory Kinds Microbenchmark Experiment Workflow (Figure 5).
The OSU Microbenchmarks were built according to its install instructions to use HPE Cray MPICH, notably including configure options: `CC=cc CXX=CC –enable-cuda`. Performance was obtained from the `osu_get_bw` test ("MPI uni-directional get flood bandwidth").
`UPC++` was built according to its HPE-Cray-EX-specific install instructions (embedded in INSTALL.md), notably including configure options `CC=cc CXX=CC –with-ofi-provider=cxi –with-cuda`. The "Reference" memory kinds series was obtained from a separate `UPC++` install with additional configure option `–disable-kind-cuda-uva` that disables GDR support in `GASNet-EX` and activates the reference implementation of `UPC++` memory kinds. Performance for both `UPC++` series was obtained from the `bench/gpu_microbenchmark.cpp` microbenchmark included in the `UPC++` source distribution.
The following environment variables were used while running the experiment:

```
export MPICH_GPU_SUPPORT_ENABLED=1
export UPCXX_SHARED_HEAP_SIZE=16G
export FI_MR_CACHE_MAX_SIZE=-1
```

These respectively enable GPU support in HPE Cray MPI, configure the `UPC++` shared heap to a size sufficient for the needs of the benchmark program, and enable a defect workaround in the system-level HPE CXI libfabric provider that underpins both MPI and `UPC++`/`GASNet-EX`. All other variables and settings used the center-provided defaults.
Benchmarks were run within a single batch job on two nodes with one process per node, with each process using one GPU and one NIC (the latter is a current limitation in both software stacks). In each test the active process issued RMA get operations into local GPU memory, from remote host memory located on the passive peer. Each test used a window size of 64 RMA gets per synchronization (i.e., `MPI_Win_flush()` or `upcxx::future::wait()`), with a total of 40 windows per payload size. Launch commands were of the form:

```
srun -n2 -N2 -c8 --gpus-per-task=1 --gpu-bind=closest \
     osu_get_bw -m 8:4194304 -w create -s flush -d cuda -i 2560 H D
srun -n2 -N2 -c8 --gpus-per-task=1 --gpu-bind=closest \
     gpu_microbenchmark -c -g -uni -flood -gg -gs -sg -ss -t 40 -w 64
```

`gpu_microbenchmark` reports bandwidth results in units of GiB/s (2^{30} bytes/second), whereas `osu_get_bw` reports bandwidth results in units of MB/s (10^6 bytes/second). All results were converted to units of MiB/s (2^{20} bytes/second) for the graph.
The execution time for this experiment workflow should be no more than 30 minutes (excluding queuing delays).

A.2.4 symPACK Experiment Workflow (Figures 6 .. 12).
The following matrices used in our experiments were obtained from [8]:

- Flan_1565: https://sparse.tamu.edu/Janna/Flan_1565
- boneS10: https://sparse.tamu.edu/Oberwolfach/boneS10
- thermal2: https://sparse.tamu.edu/Schmid/thermal2

The Rutherford-Boeing format was used for the runs of `symPACK`, and the Matrix Market format was used for the runs of `PaStiX`. The Flan_1565, boneS10, and thermal2 matrices were used in our experiments. Additionally, both `symPACK` and `PaStiX` were built with and used the `Scotch` ordering library. All experiments were run using an exclusive batch allocation of 1..64 Perlmutter GPU compute nodes.
The following environment variables were set during all experimental runs to workaround a performance defect in the vendor's libfabric CXI provider underlying both MPI and `UPC++`/`GASNet-EX`:

```
export FI_MR_CACHE_MAX_SIZE=-1
export FI_MR_CACHE_MAX_COUNT=-1
```

All other variables and settings used the center-provided defaults.
The command line used to execute `symPACK` was of the form:

```
upcxx-srun -n <tasks> -N <nodes> --gpus-per-node 4 -shared-heap <heap_size> -- \
   ./run_sympack2D -in </path/to/matrix/> -nrhs 1 -ordering SCOTCH
```

Experiments were run with a varying number of tasks per node, and the best performance for each node count was reported as the final result for that node count. Figure 6 was generated by adding the `-gpu_v` command-line option that outputs work distribution statistics.
The command line used to execute `PaStiX` was of the form:

```
srun -n <tasks> -N <nodes> --gpus-per-task 1 -c <cores_per_task> \
   ./simple -f 0 -s 3 -t 1 -g 1 --mm </path/to/matrix/>
```

As with `symPACK`, experiments were run with varying number of tasks per node, and the best performance for each node count was reported. The plots in the paper show the solve and factorization times reported by `symPACK` and `PaStiX` for runs on 1-64 nodes for each of the three test matrices.
The execution time for the experiment workflow for a single matrix should be no more than 30 minutes (excluding queuing delays).

A.3 Artifact Dependencies and Requirements

A.3.1 System Requirements.
Here are the general hardware and software requirements for replicating the experiments performed in the paper on a different system:

- Recent x64_64-compatible CPU architecture
- Recent CUDA-compatible NVIDIA GPU accelerators
- Recent Linux operating system
- Working installation of an appropriate MPI library
- Working installation of appropriate BLAS, CBLAS and LAPACK libraries
- Working installation of the NVIDIA cuBLAS library
  - This is included in the NVIDIA HPC SDK and CUDA Toolkits
  - More details: https://developer.nvidia.com/cublas

A.3.2 Other Dependencies.
See Software Requirements section above for the list of software libraries and versions used in the experiments presented in the paper. See §A.2.4 above for the input data sets.

A.4 Artifact Installation and Deployment Process

A.4.1 Configure and install the `UPC++` *library with CUDA support.*
This step may be skipped on production systems where a CUDA-enabled `UPC++` installation is already provided: consult https://upcxx.lbl.gov/site for an incomplete list of installs on production facilities and system-specific instructions.
Installation details will vary depending on the given system. Follow the detailed instructions provided in the `UPC++` library archive, also available online here: https://upcxx.lbl.gov/wiki/INSTALL. Be sure to include `-enable-cuda` in the configure step.
This installation step should take no longer than 30 minutes, including time to read the relevant instructions.

A.4.2 Build and install the `Scotch` *library.*
To build and install `Scotch`, execute the following commands from the `Scotch` source directory:

```
mkdir build && cd build
cmake .. -DCMAKE_INSTALL_PREFIX=/path/to/install -DCMAKE_BUILD_TYPE=Release
make && make install
```

This installation step should take no longer than 10 minutes

A.4.3 Build the `symPACK` *library with CUDA support.*
To build and install `symPACK`, an installation of `Scotch` is required. Execute the following commands from the `symPACK` source directory:

```
mkdir build && cd build
export scotch_PREFIX=/path/to/scotch/install
cmake .. -DENABLE_CUDA=ON -DCMAKE_BUILD_TYPE=Release \
  -DENABLE_SCOTCH=ON -DSCOTCH_INCLUDE_DIR=$scotch_PREFIX/include
make
```

This installation step should take no longer than 15 minutes.

A.4.4 Build `StarPU` *(to replicate baseline comparison).*
This optional step is not required by symPACK, but is required in order to replicate the `PaStiX` runs presented in the paper. To build and install `StarPU`, execute the following commands from the `StarPU` source directory:

```
mkdir build && cd build
../configure --enable-fast --enable-mpi --prefix=/path/to/install \
    --with-cuda-dir=/path/to/cuda/dir
make && make install
```

Note the `StarPU` configure script expects to find both the CUDA library and cuBLAS library in the same provided CUDA directory. If these are actually installed in different places (as on Perlmutter), the recommended workaround is to embed this information using the `configure -with-cuda-lib-dir` option, eg:

```
../configure ... --with-cuda-dir=/path/to/cuda/ \
    '--with-cuda-lib-dir=/path/to/cuda/lib64/ -L/path/to/cublas/lib64/'
```

This installation step should take no longer than 20 minutes.

A.4.5 Build `PaStiX` *with CUDA support (to replicate baseline comparison).*
This optional step is not required by symPACK, but is required in order to replicate the `PaStiX` runs presented in the paper. To build and install `PaStiX`, installations of `StarPU` and `Scotch` are required. Execute the following commands from the `PaStiX` source directory:

```
mkdir build && cd build
export STARPU_DIR=/path/to/starpu
export SCOTCH_DIR=/path/to/scotch
export PATH=$PATH:$STARPU_DIR/bin:$SCOTCH_DIR/bin
export PKG_CONFIG_PATH=$PKG_CONFIG_PATH:\
    $STARPU_DIR/lib/pkgconfig:$SCOTCH_DIR/lib/pkgconfig
export LD_RUN_PATH=$LD_RUN_PATH:$STARPU_DIR/lib:$SCOTCH_DIR/lib
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$STARPU_DIR/lib:$SCOTCH_DIR/lib
export INCLUDE_PATH=$INCLUDE_PATH:$STARPU_DIR/include:$SCOTCH_DIR/include
cmake .. -DCMAKE_INSTALL_PREFIX=/path/to/install \
    -DPASTIX_WITH_STARPU=ON -DPASTIX_WITH_MPI=ON -DPASTIX_WITH_CUDA=ON \
    -DPASTIX_INT64=OFF -DPASTIX_ORDERING_SCOTCH=ON
make simple
```

On Perlmutter we observed the `PaStiX` CMake infrastructure did not correctly detect the vendor-provided CBLAS and LAPACK libraries provided by the `cray-libsci` module (which should be preferred to third-party BLAS libraries). If you encounter errors on an HPE Cray EX regarding `MORSE::LAPACKE` and `MORSE::CBLAS`, the recommended workaround is to ignore the CMake errors and proceed with the build using these commands:

```
make simple  # expected link error
perl -pi -e 's/-lMORSE::LAPACKE//;s/-lMORSE::CBLAS//' \
    example/CMakeFiles/simple.dir/link.txt
make simple
```

This installation step should take no longer than 15 minutes.

A.4.6 Run the experiments on the computational resources.
See §A.2.3 and §A.2.4 above for detailed commands suitable for use on the GPU partition of the NERSC Perlmutter supercomputer. Job launch commands will vary in system-specific ways on other systems.
The execution time for each experiment workflow (e.g. for a single matrix) should be no more than 30 minutes, excluding job queuing delays.

A.5 Other Notes

All experimental results presented in the paper used the GPU partition of NERSC Perlmutter, an HPE Cray EX supercomputer detailed in §A.2.2. The paper's central results are expected to generalize in a qualitative manner to sufficiently similar systems, but some variance should be expected in detailed quantitative results on other systems.