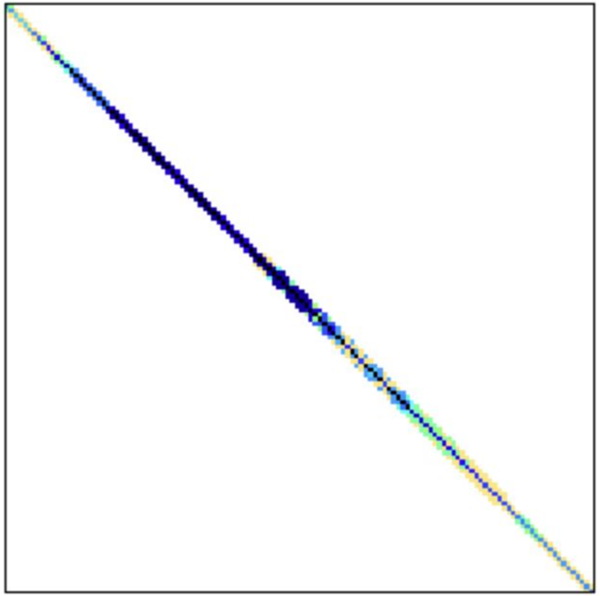
symPACK: A GPU-Capable Fan-Out Sparse Cholesky Solver

Julian Bellavita[1], Mathias Jacquelin[2], Esmond G. Ng[1], Dan Bonachea[1], Johnny Corbino[1], and Paul H. Hargrove[1]

[1]Lawrence Berkeley National Laboratory
[2]Cerebas Systems

Introduction

- Sparse symmetric positive-definite systems of equations are ubiquitous
- Sparse direct solvers use Cholesky Factorization to efficiently solve such systems
- Parallel sparse Cholesky codes are essential
- But, modern HPC is heterogeneous
    - Codes need to exploit CPUs and GPUs



Introduction

symPACK is a parallel sparse Cholesky solver that effectively utilizes heterogeneous processing units and employs a novel one-sided communication algorithm

https://github.com/symPACK/symPACK

Cholesky Basics

- Goal: Solve Ax=b, where A is spd
- Factorize step: $A = LL^T$
  - Proceed one column at a time
  - Compute each column of L using column of A
  - Update trailing lower triangular region of A
- Solve step: Solve Ly=b for y, then solve $L^Tx=y$ for x
  - Forward/Backward substitution

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

Sparse Cholesky

- Group contiguous columns of A into "supernodes"
  - Group rows into dense blocks
  - Lets you use dense matrix operations
- Derive an elimination tree from the supernodes
  - This gives you a de-facto task graph
- Factorize each supernode according to the elimination tree
- Fill-in: nonzeros in L that were zero in A
  - Reduce fill-in this by reordering A with permutation matrices P, P^T

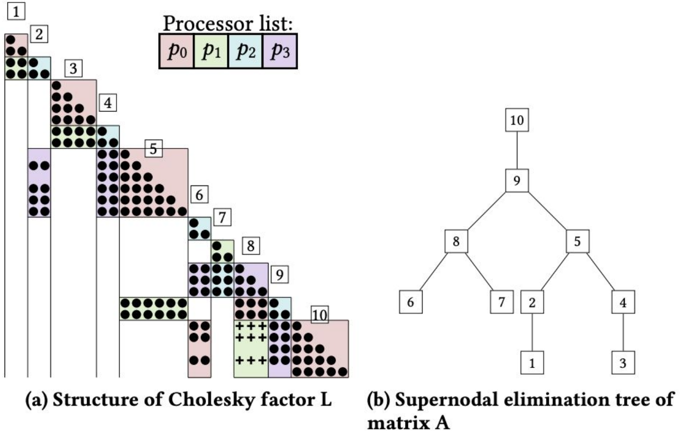


Figure 1: Sparse matrix A partitioned into supernodes and dense blocks. $\boxed{i}$ denotes the i-th supernode, • represents original nonzero elements in A, while + denotes fill-in entries. Colors correspond to the four distributed-memory nodes onto which blocks are mapped in a 2D block-cyclic way.

symPACK Implementation

symPACK Task Formulation

- Formulate Cholesky factorization as tasks that operate on dense blocks of A
    - Supernode partitioning, then block partitioning
    - Computation is done using BLAS 3/LAPACK operations to achieve superior performance

- Three kinds of tasks
    - Diagonal Factorize D_j: Factorize diagonal block in supernode j
    - Factorize $F_{i,j}$: Factorize block i in supernode j
    - Update $U_{i,j,k}$: Update block i in supernode k using the factorized block i in supernode j ($j < k$)

Task Dependencies

- A diagonal block must be factorized before other blocks in the supernode can be factorized
- A block must be factorized before it can be used to update other blocks
- All updates must be applied to a block before it can be factorized

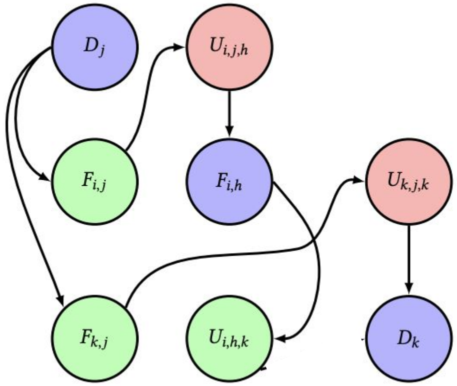


Figure 2: ***fan-out*** **task dependencies for four columns** j, i, k, **and** h

Task Scheduling

- Each processor has two task queues
  - Local task queue (LTQ): All tasks mapped to this processor
  - Ready task queue (RTQ): Tasks mapped to this processor that can be scheduled
- Tasks can be executed if all of their dependencies have been satisfied

- Tasks have a dependency counter

- Tasks are popped from the RTQ and executed, then they produce data used to satisfy dependencies between tasks
  - Once a task's dependency counter hits zero, it is moved from the LTQ to the RTQ

Parallel Algorithm

- Individual blocks are mapped to processors using a 2D block-cyclic mapping
- All tasks involving a block are mapped to the processor owning that block

- 2 kinds of messages
  - Diagonal factorized blocks need to be sent to remote processors for factorize tasks
  - Off-diagonal factorized blocks need to be sent to remote processors for update tasks

- Communication is handled with **one-sided RMA operations** and **remote procedure calls** provided by UPC++

Communication Paradigm

- Four UPC++ constructs are important here
  - **global address space:** region of memory that each processor owns a region of
    - Processors can access regions of the global address space owned by other processors
  - **rget**: reads data located in a remote processor's global address space using a global pointer
  - **Remote procedure call:** local processor enqueues a procedure on a remote processor
  - **progress:** advances internal UPC++ state, executes enqueued RPCs

- Example: Task T_1 produced data task T_2 needs
- Psource owns T_1, Ptarget owns T_2

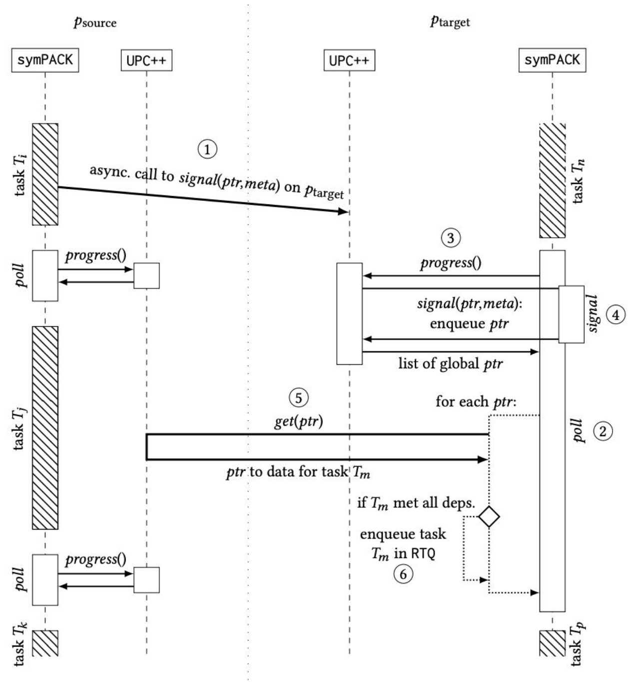


Figure 4: Data exchange protocol in symPACK. Notifications are performed using UPC++ asynchronous RPC, and the actual data is fetched using non-blocking one-sided RMA get.

Communication Paradigm

- Step 1: Enqueue RPC to `signal()` on Ptarget
- One argument to `signal()` is a global pointer to the data T_2 needs

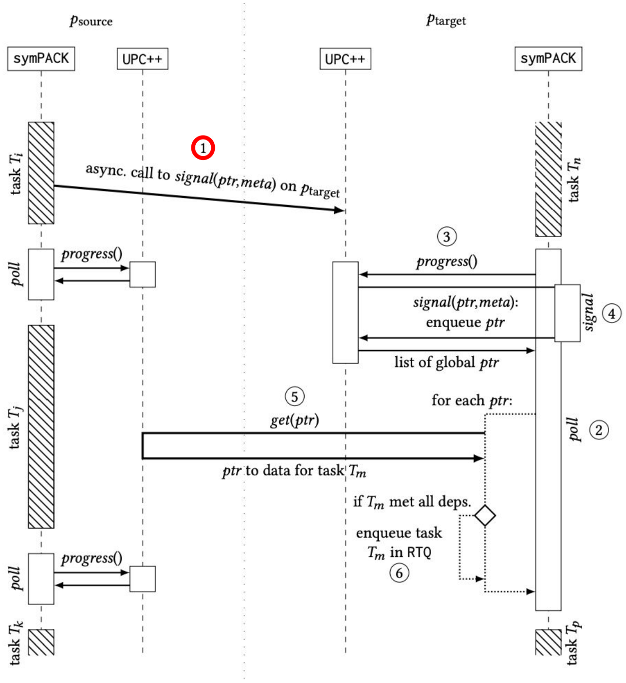


Figure 4: Data exchange protocol in symPACK. Notifications are performed using UPC++ asynchronous RPC, and the actual data is fetched using non-blocking one-sided RMA get.

Communication Paradigm

- Step 2: call `poll()` function on Ptarget, which dispatches a call to `upcxx::progress()`

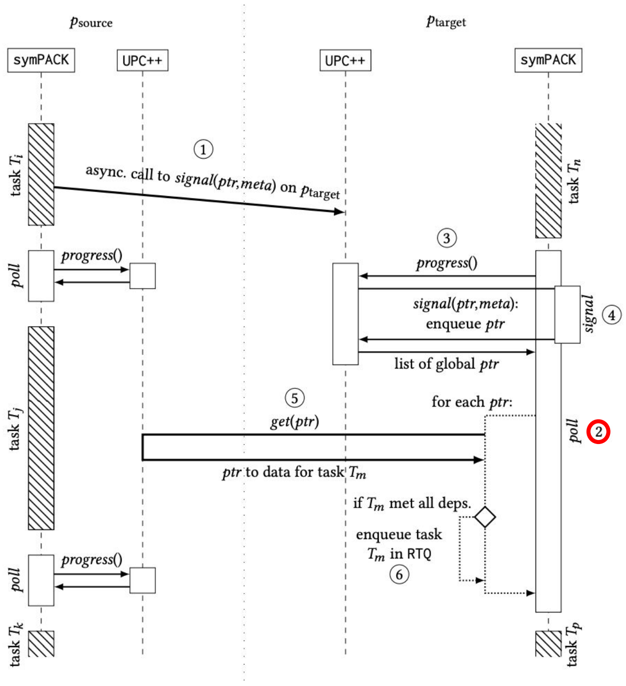


Figure 4: Data exchange protocol in symPACK. Notifications are performed using UPC++ asynchronous RPC, and the actual data is fetched using non-blocking one-sided RMA get.

Communication Paradigm

- Step 3: `upcxx::progress()` executes the RPC on Ptarget

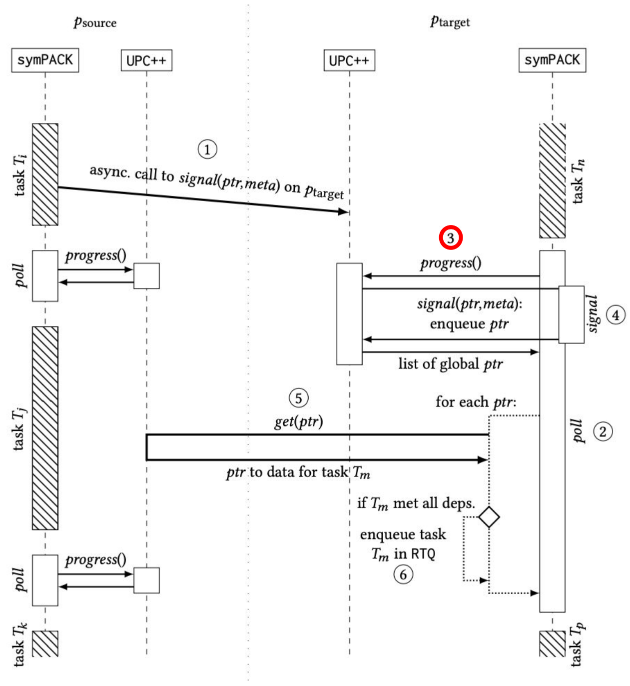


Figure 4: Data exchange protocol in symPACK. Notifications are performed using UPC++ asynchronous RPC, and the actual data is fetched using non-blocking one-sided RMA get.

Communication Paradigm

- Step 4: `signal()` enqueues global pointer in a list of global pointers local to processor owning T_2

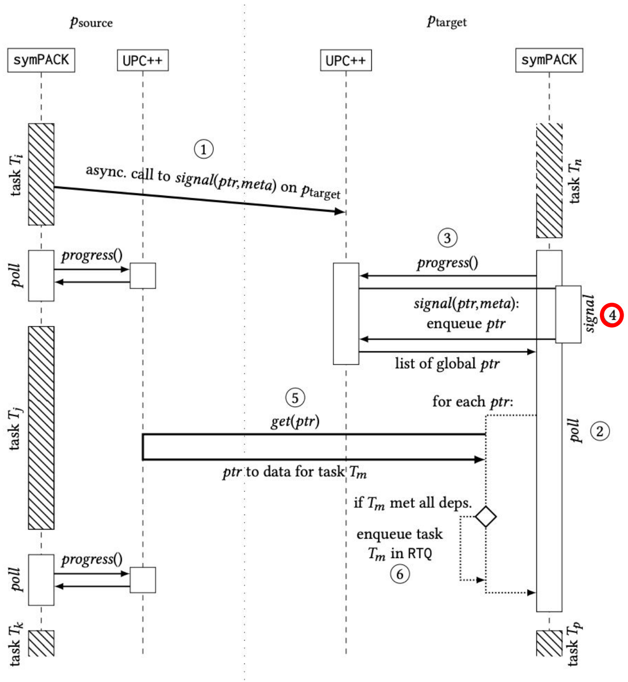


Figure 4: Data exchange protocol in symPACK. Notifications are performed using UPC++ asynchronous RPC, and the actual data is fetched using non-blocking one-sided RMA get.

Communication Paradigm

- Step 5: Iterate through list of global pointers, call `upcxx::rget()` on each one
- This actually satisfies the data dependency

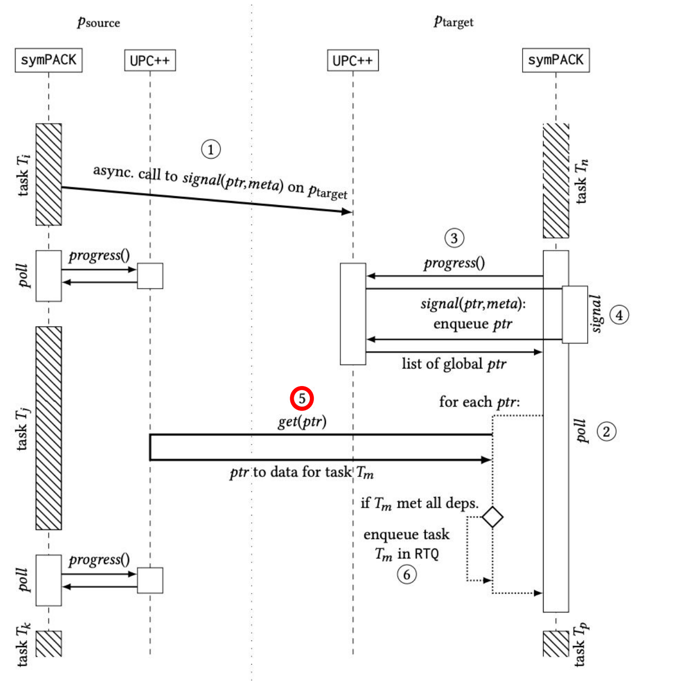


Figure 4: Data exchange protocol in symPACK. Notifications are performed using UPC++ asynchronous RPC, and the actual data is fetched using non-blocking one-sided RMA get.

Communication Paradigm

- Step 6: Decrement T_2 dependency counter, push on RTQ if all dependencies are satisfied

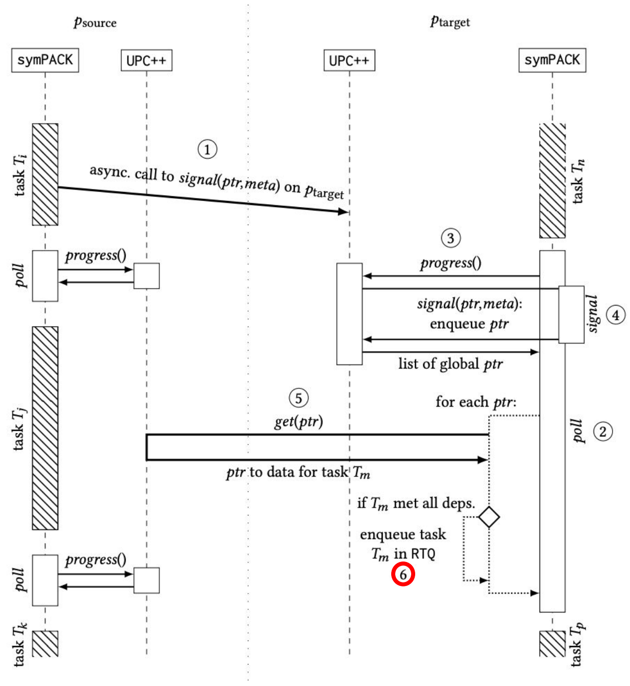


Figure 4: Data exchange protocol in symPACK. Notifications are performed using UPC++ asynchronous RPC, and the actual data is fetched using non-blocking one-sided RMA get.

symPACK GPU Functionality

GPU Functionality

- symPACK's GPU functionality is built with UPC++ memory kinds
  - Extends the global address space to include device memory
  - Allocate memory on devices with `upcxx::device_allocator`
  - Returns a global pointer to device memory

- `upcxx::copy()` moves data between any combination of hosts and devices
  - Local host <--> Remote device
  - Local device <--> Remote host
  - Local device <--> Remote device
  - Local Host <--> Remote host

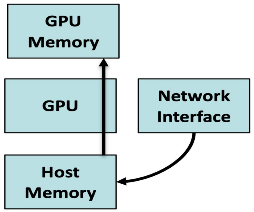
GPU Functionality

- It’s best to use the GPU only for tasks that have a high arithmetic intensity
  - This translates to tasks that operate on large blocks
  - Inevitable overheads mean computation has to be much faster to justify overhead

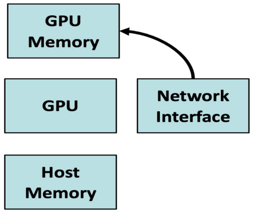
- For each BLAS/LAPACK operation, define a <u>size threshold</u> that determines whether we map the an instance of each operation to the GPU or the CPU
  - If an operation involves a block > size threshold, do it on the GPU, otherwise do it on the CPU
  - cuBLAS/cuSolver handles local computation

Optimizing GPU Communication with Memory Kinds

- Observation: If a block is large enough, all tasks involving the block will happen on the GPU
- Naive approach: fetch data from remote host onto local host, then copy it to local device
- Superior approach: fetch data from remote host directly to local device
    - Memory kinds enable the superior approach through GASNet-EX's support for GPUDirect RDMA (GDR)



Data movement without acceleration



Data movement with acceleration

Performance Evaluation

Performance Evaluation

- All experiments were run on NERSC Perlmutter GPU nodes
  - 1 AMD EPYC 7763 “Milan” CPU with 64 cores
  - 4 NVIDIA A100 “Ampere” GPUs
  - 4 HPE Slingshot 11 “Cassini” 200Gbps network cards

- Benchmarked GPU mode of symPACK using GPU mode of PaStiX as a baseline
  - Matrices are from SuiteSparse matrix collection
  - symPACK and PaStiX both use the Scotch ordering library

Performance Evaluation

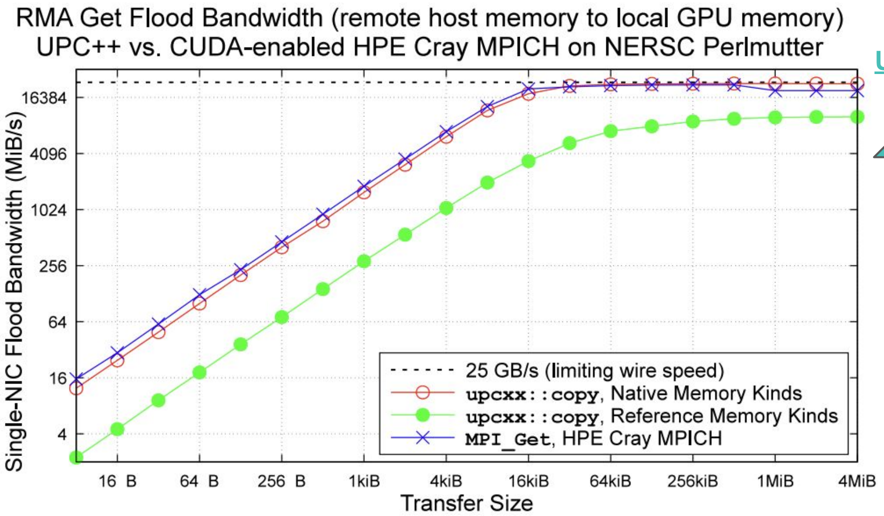
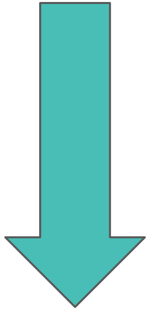


Figure 5: Microbenchmark comparison of one-way point-to-point communication bandwidth for non-blocking RMA gets involving GPU-resident buffers using GPUDirect RDMA technology versus the same transfer staged through an intermediate buffer in host memory.

Up is good

- Impact of memory kinds
- Bandwidth vs message size for MF one sided get, native `upcxx::copy()`, and reference `upcxx::copy()`

Performance Evaluation



Down is good

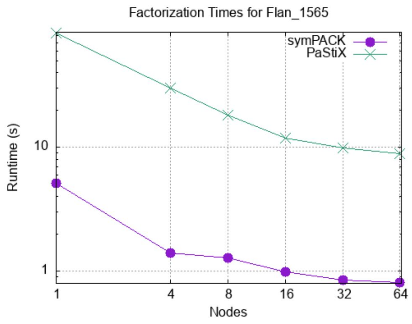


Figure 7: Strong scaling of symPACK's Cholesky factorization on Flan_1565

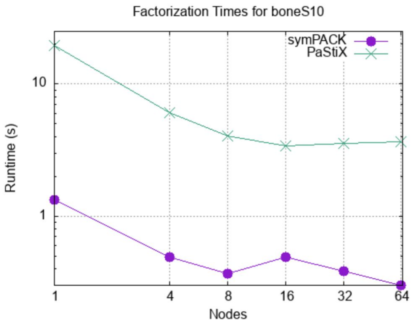


Figure 9: Strong scaling of symPACK's Cholesky factorization on boneS10

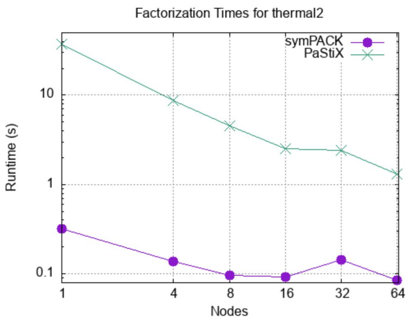


Figure 11: Strong scaling of symPACK's Cholesky factorization on thermal2

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.**

Future Work

- Develop a more sophisticated task scheduling policy
- Autotuning for GPU thresholds
- Supernode coalescing

Acknowledgements

This research was supported by the Exascale Computing Project (17- SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, 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.

Thank you!