Parallel Symbolic Cholesky Factorization

Tobias Ribizel[1], Hartwig Anzt[1,2]

[1] Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
[2] Innovative Computing Laboratory (ICL), University of Tennessee, Knoxville, USA

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 the Helmholtz Impuls und Vernetzungsfond VH-NG-1241.
This work was performed on the HoreKa supercomputer and the NHR@KIT Future Technologies Partition testbed funded by the Ministry of Science, Research and the Arts Baden-Württemberg and by the Federal Ministry of Education and Research.

Sparse Direct Solvers

- Sparse linear system $Ax = b$, solved via factorization $A = L \cdot U$
- Factorization adds *fill-in* to sparsity pattern of A
- Fill-in reducing heuristics like Approximate Minimum Degree (AMD) and Nested Dissection reordering
- Reordered matrix PAP^T has fewer fill-in entries → less storage, faster factorization
- Reordering → Symbolic Factorization → Numerical Factorization → Triangular Solve

Sparse Direct Solvers

- Sparse linear system $Ax = b$, solved via factorization $A = L \cdot U$
- Factorization adds *fill-in* to sparsity pattern of A
- Fill-in reducing heuristics like Approximate Minimum Degree (AMD) and Nested Dissection reordering
- Reordered matrix PAP^T has fewer fill-in entries → less storage, faster factorization
- Reordering → **Symbolic Factorization** → Numerical Factorization → Triangular Solve

Sparse Factorizations: Symbolic Phase

- General criterion: For non pattern-symmetric A, $(L+U)_{\mathrm{ij}}$ is nonzero if and only if there is a path $i \to k_1 \to \cdots \to k_n \to j,\ k_l < \min(i,j)$ through A
- Symmetric case: L_{ij} nonzero if and only if there is a path $i \to k_1 \to \cdots \to k_n \to j, k_l < j$
- L has a compact representation through the Elimination Tree T (transitive reduction)
- T can be computed in almost linear time (size and number of nonzeros of A)
- Sparsity pattern of row L_{i*} consists of all pairwise paths between A_{i*} through T

Symbolic Cholesky: Enumerating Fill-in

Theorem: The entry l_{ij} in L is (symbolically) nonzero if and only if the row subtree of i contains j, i.e. there is a nonzero a_{ik} in A such that j lies on the path from k to i in the elimination tree T

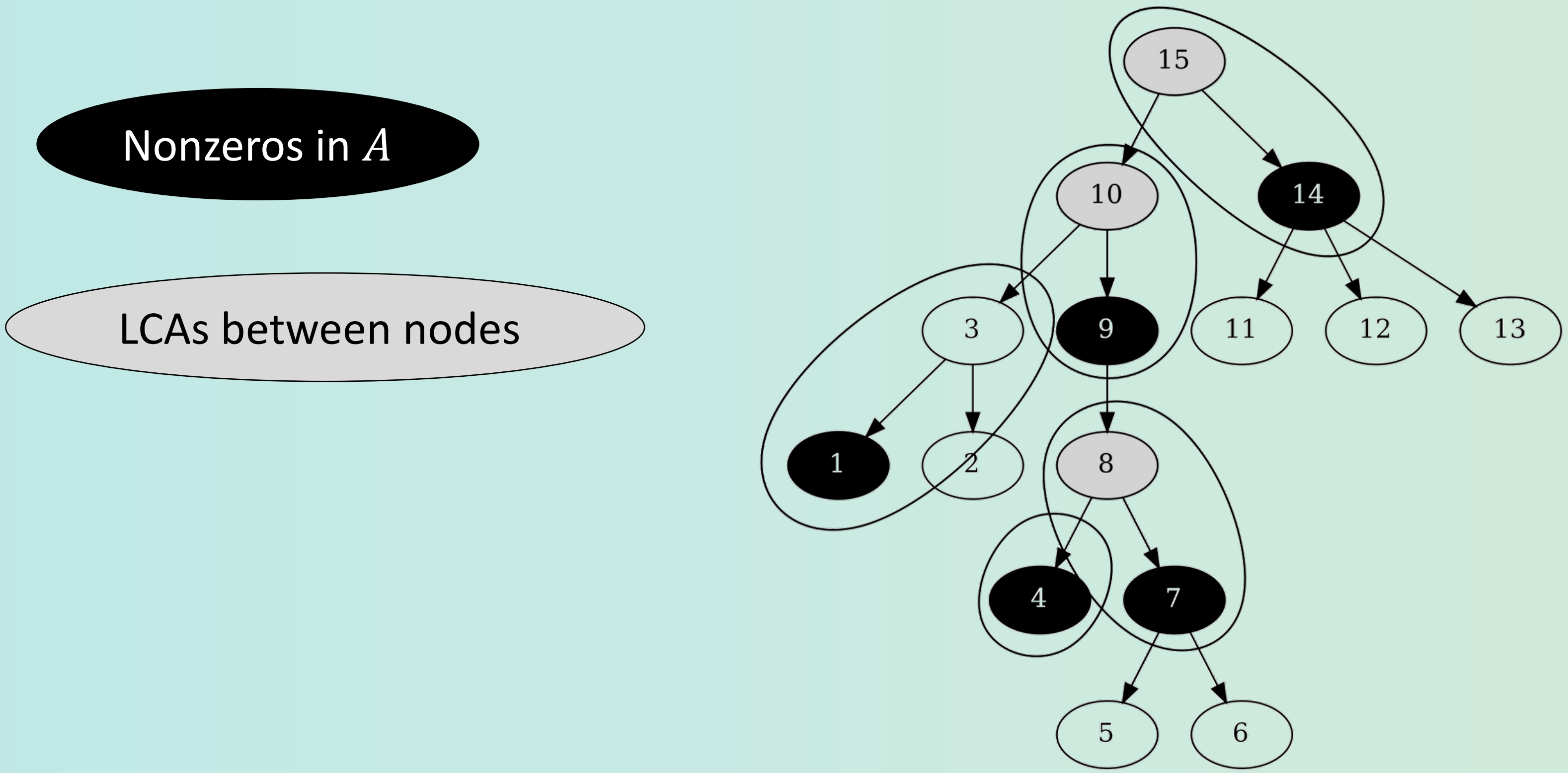
→ we need to identify lowest common ancestors (LCAs) between pairs k_1, k_2 belonging to nonzeros a_{ik_1}, a_{ik_2}

Theorem: In a post-ordered tree T, the LCA between any pair of nodes $u < v$ is the first node w on the path from u towards the root that fulfills $v \leq w$

→ we can limit ourselves to LCA search between consecutive nonzeros of A in postorder

Theorem: After postordering the matrix, the ordered lower nonzeros of a row a_{ij_k}, $j_1 < \cdots < j_n = i$ give a path decomposition of the row subtree of row i via $[j_k, LCA(j_k, j_{k+1}))$, plus the root i.

Symbolic Cholesky: Enumerating Fill-in

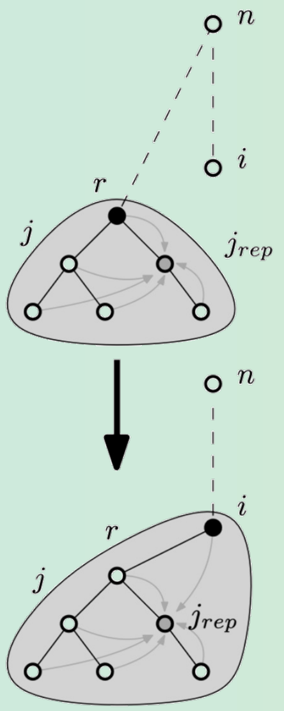


Symbolic Cholesky: Algorithmic Framework

- Copy A to the CPU (optional)
- CPU: Compute Elimination Tree T
- CPU: Compute post-ordering of T
- Copy T to the device (optional)
- Device: Reorder rows of A with post-order column indices
- Device: Count number of nonzeros for each row
- Device: Allocate memory
- Device: Generate nonzeros for each row
- Device: Sort rows by column index (optional)

Symbolic Cholesky: Computing the Elimination Tree

```
make_sets(n);
for row i = 0, . . . , n do
    subtree_roots[i] = i;
    i_rep = i; parent[i] = n;
    for column j < i with a_ij ≠ 0 do
        j_rep = find(j);
        r = subtree_roots[j_rep];
        if parent[r] = n and r ≠ i then
            parent[r] = i;
            i_rep = union(i_rep, j_rep);
            subtree_roots[i_rep] = i;
        end
    end
end
children = [0, . . . , n − 1];
child_ptrs, children =
  transpose(children, parents);
```



Symbolic Cholesky: Enumerating

```
// Map column indices to postorder
parfor j = 0, ..., m − 1 do
    post_columns[j] = postorder^{−1}[columns[j]];
end
// Sort postorder column indices
parfor i = 0, ..., n − 1 do
    sort(post_columns[row_ptrs[i], ..., diag_idx[i]]);
end
// Traverse row subtrees
parfor i = 0, ..., n − 1 do
    parfor j = row_ptrs[i], ..., diag_idx[i] do
        u_post ← post_columns[j];
        v_post ← post_columns[j + 1];
        while u_post < v_post do
            u = postorder[u_post];
            u_post = post_parent[u_post];
            // Output or count nonzero l_iu
        end
    end
end
```

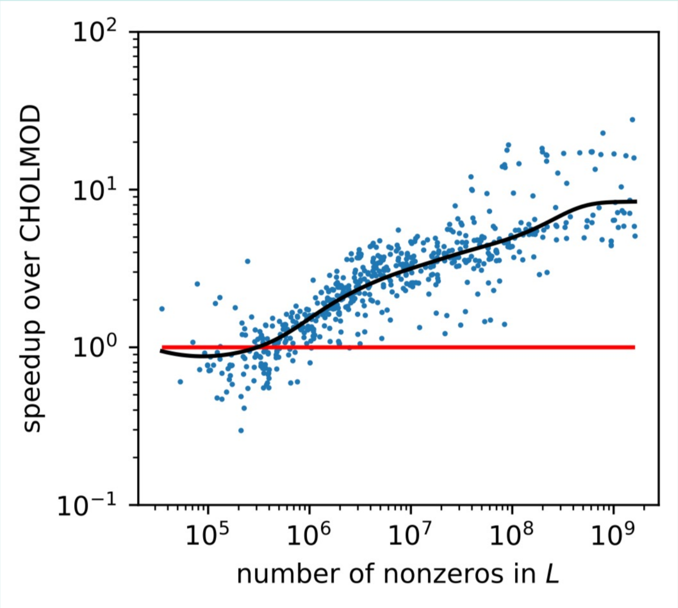
Performance Evaluation: Setup

- Code available in the Ginkgo HPC library https://github.com/ginkgo-project/ginkgo
- Comparison against symbolic part of CHOLMOD
- Compiled using gcc 11.3.0, CUDA 11.8, ROCM 5.1.1 with –O3 flags
- Inputs: (almost) all pattern-symmetric matrices from SuiteSparse with between 10^4 and 10^7 rows/columns
- Benchmarks with input ordering (*natural*), AMD and Nested Dissection (*nd*) ordering
- Removed all inputs that overflow 32 bit indices in row pointers
- Validated as correct for small sample of matrices

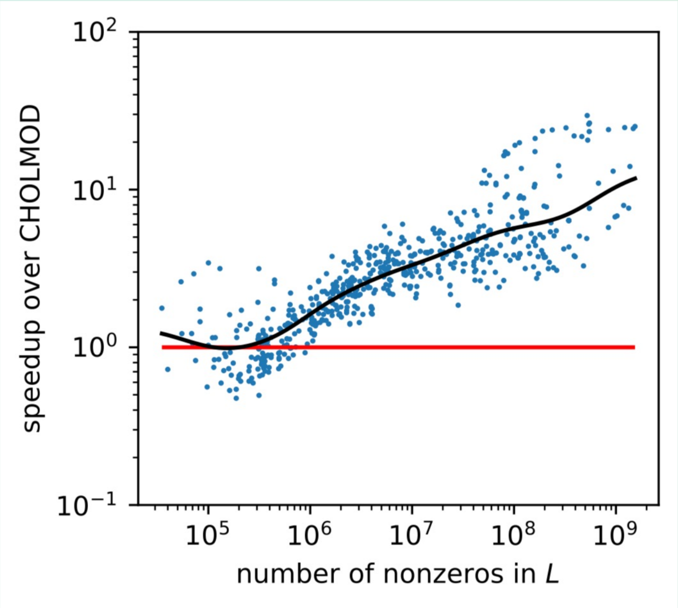
	natural	AMD	ND
#matrices	458	579	601
median fill-in	131x	8.5x	7.7x

	NVIDIA	AMD
	Intel	AMD
CPU	Xeon Platinum 8368	EPYC 7543
Sockets	2	2
Cores/Socket	38	32
L3 Cache/Socket	57 MB	256 MB
GPU	NVIDIA A100	AMD MI210
VRAM	40 GB	64 GB
Memory BW	1555 GB/s	1600 GB/s
FP32 FLOPS	19.5 TFLOPS/s	22.6 TFLOP/s

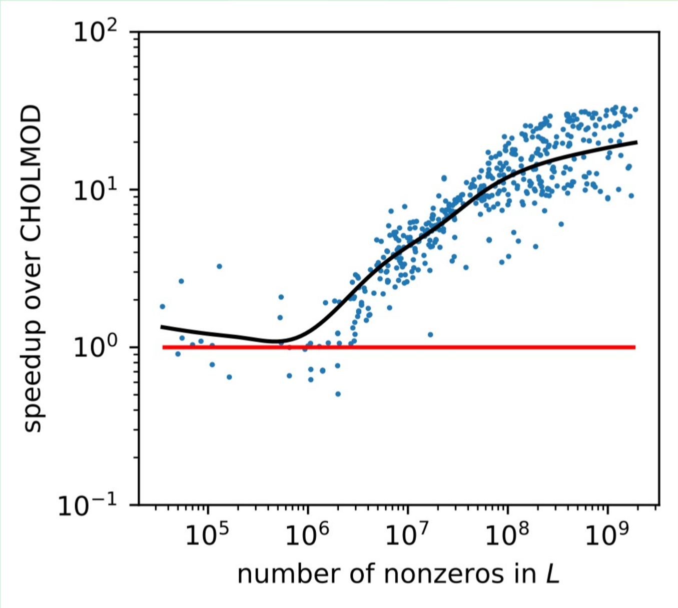
Performance Evaluation: Results Intel CPU (Speedup)



Nested Dissection

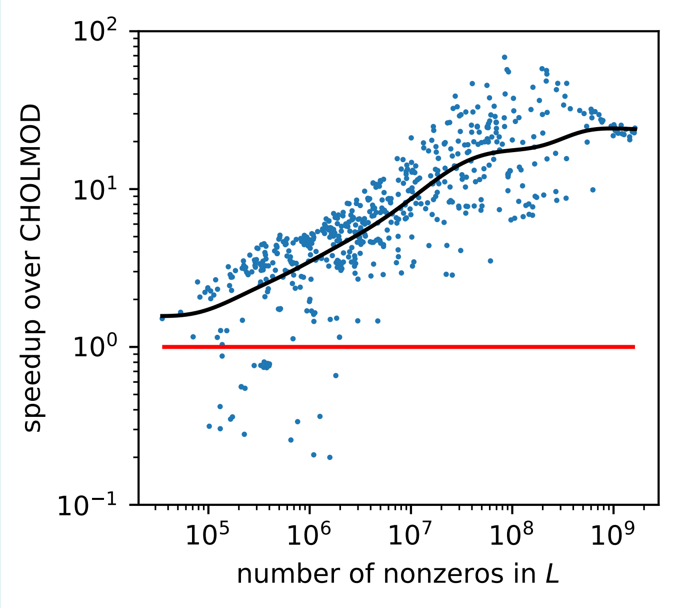


AMD

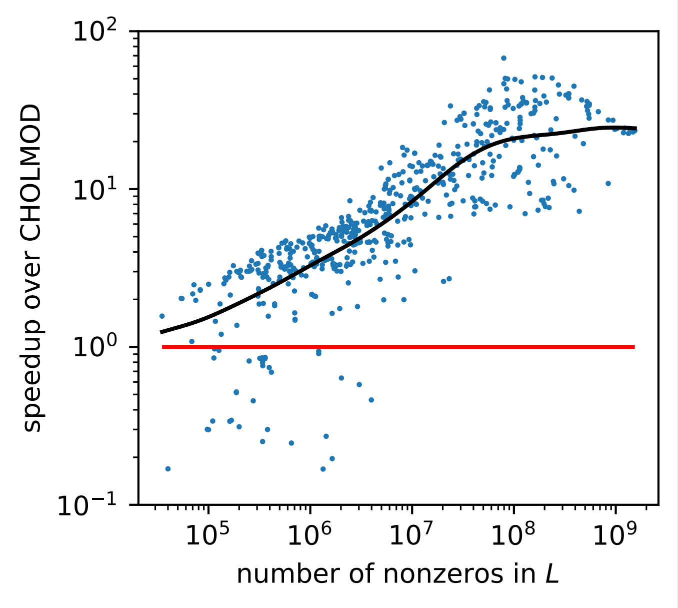


Natural

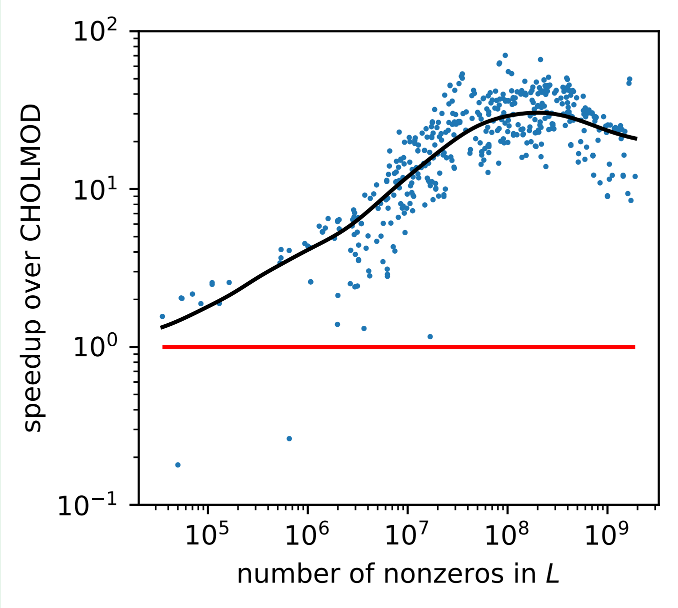
Performance Evaluation: Results AMD MI210 (Speedup)



Nested Dissection

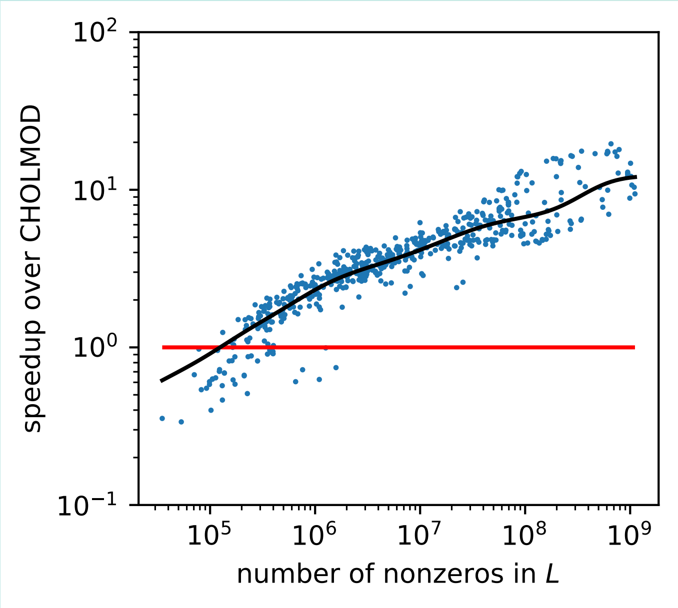


AMD

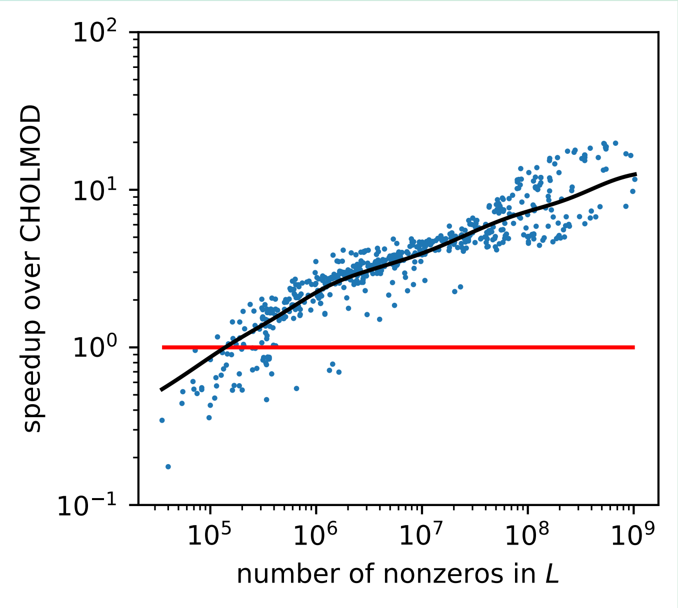


Natural

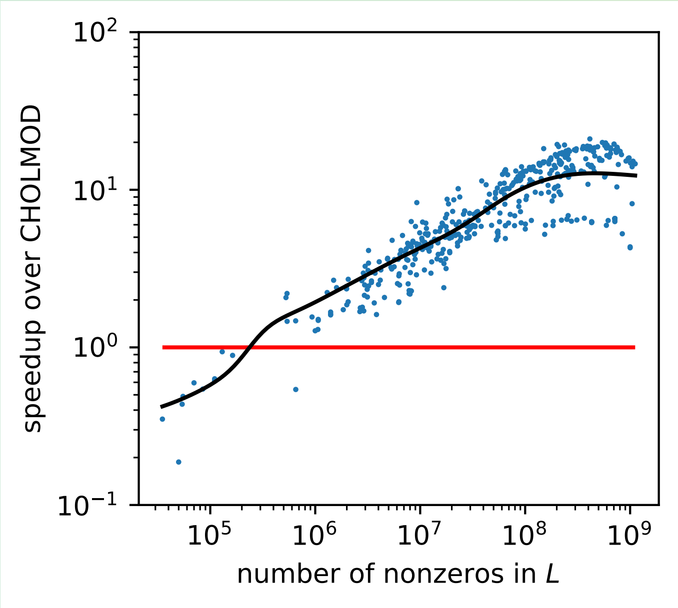
Performance Evaluation: Results NVIDIA A100 (Speedup)



Nested Dissection

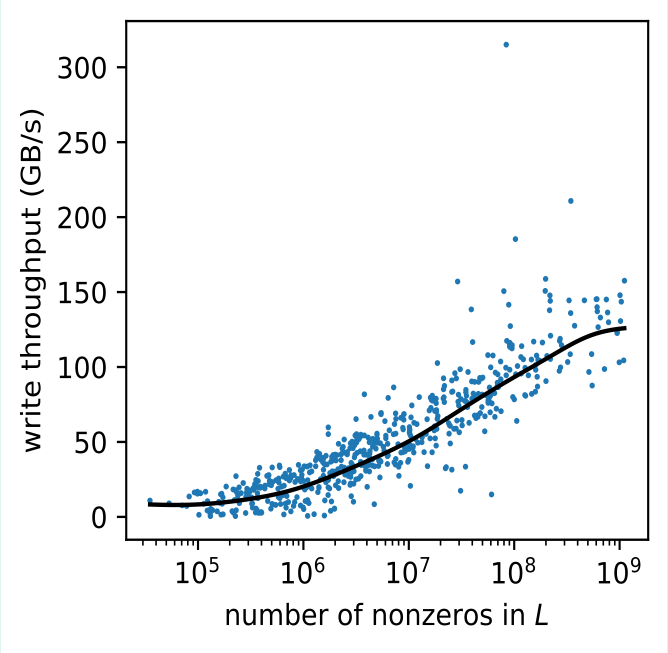


AMD

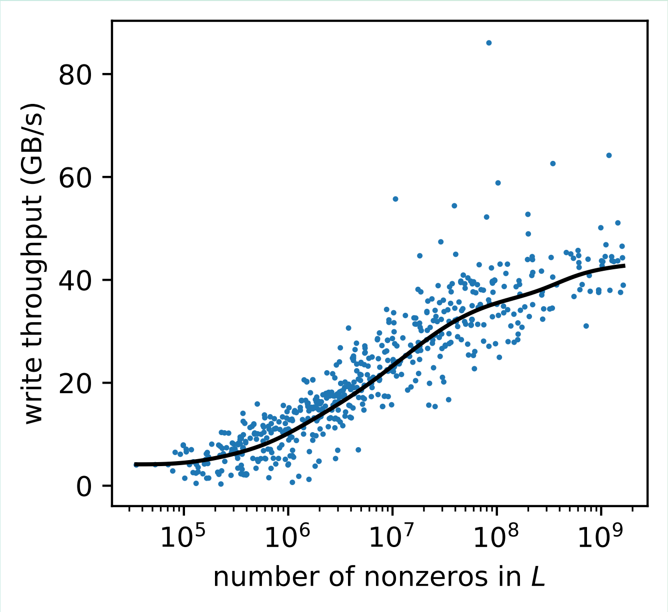


Natural

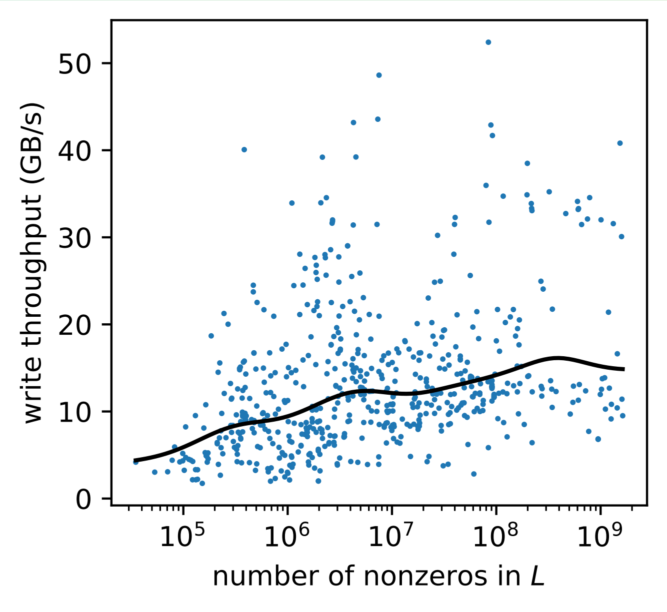
Performance Evaluation: Throughput



A100



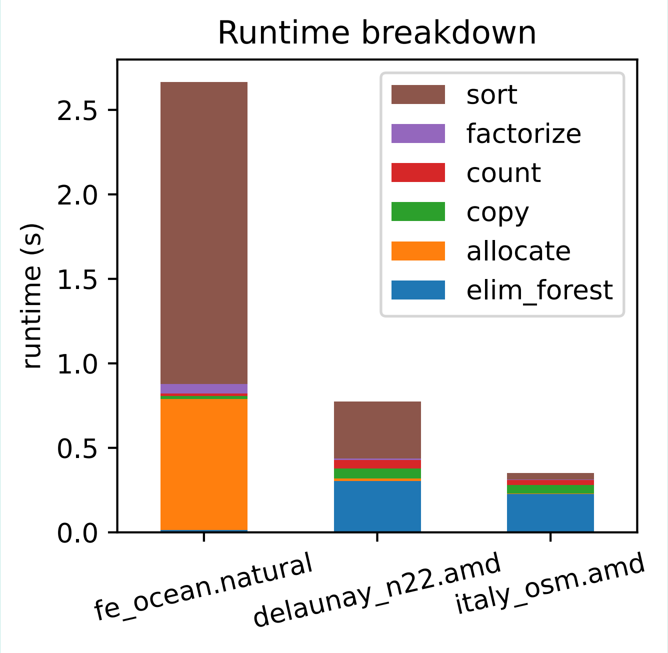
MI210



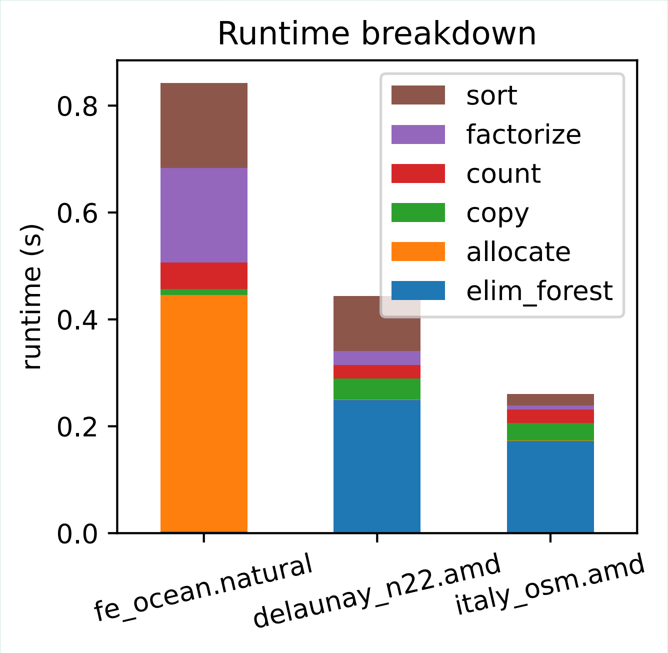
Intel CPU

Sparsity Pattern generation only on Nested Dissection ordering

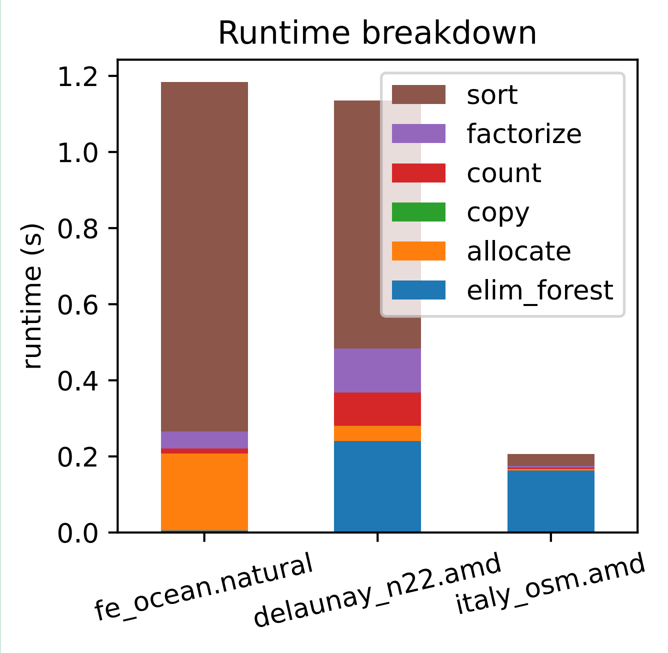
Performance Evaluation: Breakdown



A100

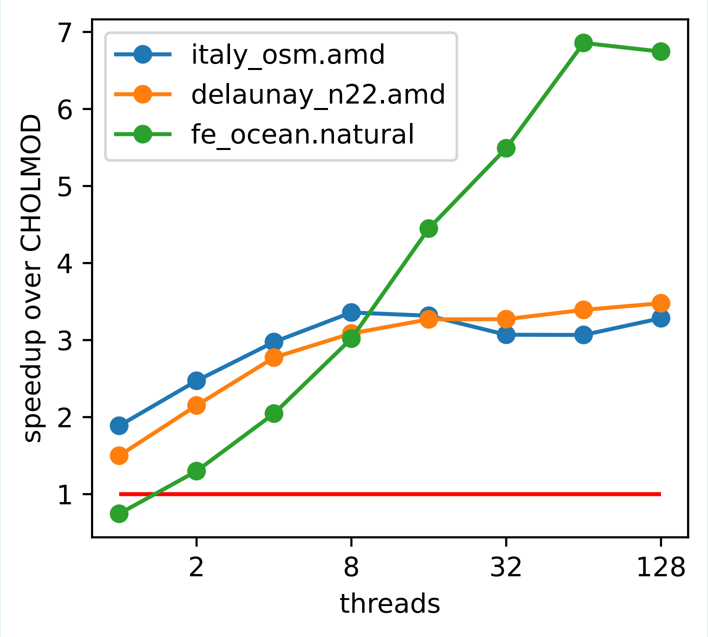


MI210



Intel CPU

Performance Evaluation: Scaling on Intel CPU



Future Work in Progress: Near-Symmetric LU

- If we symmetrize via $A_{symm} = A + A^T$, then the fill-in of A is a subset of the factors $L_{symm} + L_{symm}^T$ of A_{symm}
  → Follow Symbolic Cholesky on A_{symm} with ”numerical factorization” with A stored inside 0/1 binary A_{symm}
  → Result tells us which fill-in entries in $L_{symm} + L_{symm}\hat{}T$ are actually present in factors $L + L^T$ of A
- First results (A100 vs. sequential baseline):
  - 3.5x speedup for AMD-ordered matrix
  - 18x speedup for input-ordered matrix

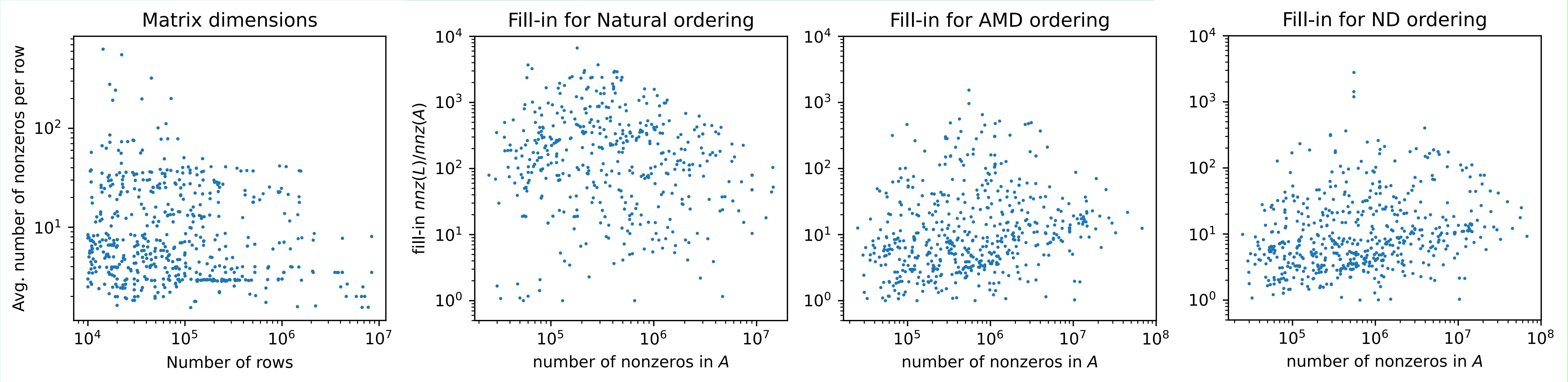
Future Work

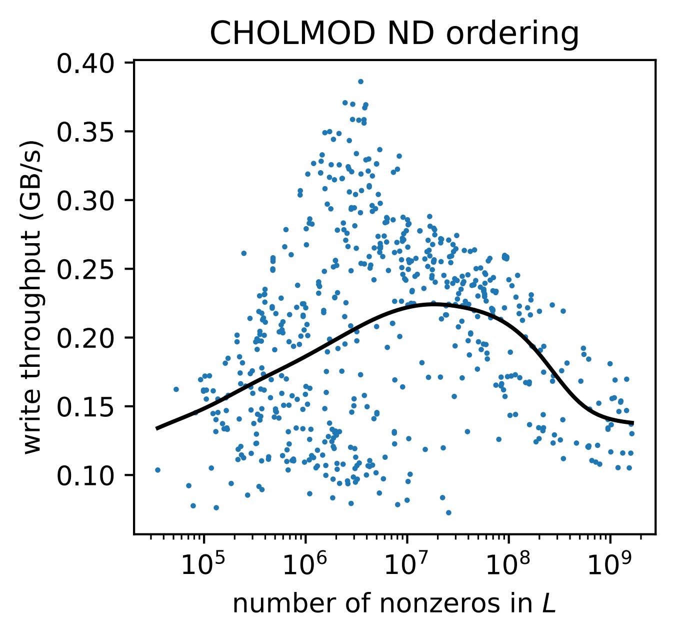
- Performance optimization of fill-in kernels (faster LCA lookup, parallel path traversal)
- Reduced data movement cost by computing skeleton graph of A
- Fully on-GPU elimination tree computation
- Fully on-GPU symbolic factorization
- Tuning numerical factorizations

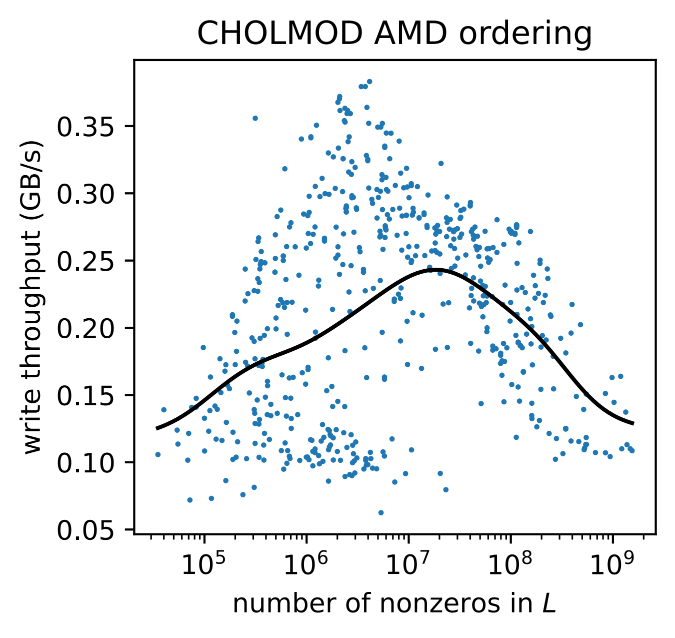
References

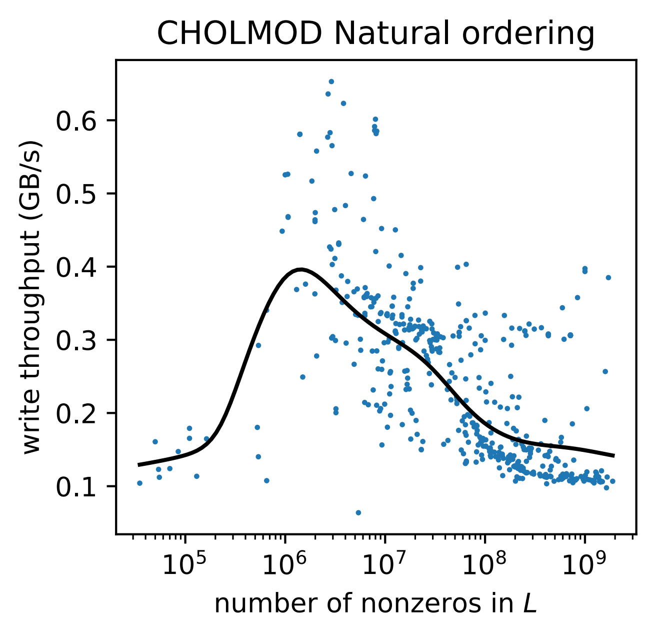
- Joseph W.H. Liu. 1990. The Role of Elimination Trees in Sparse Factorization, SIAM J. Matrix Anal. Appl, https://doi.org/10.1137/0611010
- Timothy A. Davis. 2006. Direct methods for sparse linear systems, SIAM
- Yanqing Chen, Timothy A. Davis, William W. Hager, and Sivasankaran Rajamanickam. 2008. Algorithm 887: CHOLMOD, Supernodal Sparse Cholesky Factorization and Update/Downdate. ACM TOMS, https://doi.org/10.1145/1391989.1391995
- Timothy A. Davis and Yifan Hu. 2011. The University of Florida sparse matrix collection. ACM TOMS, https://doi.org/10.1145/2049662.2049663
- Hartwig Anzt, Terry Cojean, Goran Flegar, Fritz Göbel, Thomas Grützmacher, Pratik Nayak, Tobias Ribizel, Yuhsiang Mike Tsai, and Enrique S. Quintana-Ortí. 2022. Ginkgo: A Modern Linear Operator Algebra Framework for High Performance Computing. ACM TOMS, https://doi.org/10.1145/3480935

Backup Slides


Matrix dimensions
Avg. number of nonzeros per row
Number of rows
Fill-in for Natural ordering
fill-in nnz(L)/nnz(A)
number of nonzeros in A
Fill-in for AMD ordering
number of nonzeros in A
Fill-in for ND ordering
number of nonzeros in A


CHOLMOD ND ordering
write throughput (GB/s)
0.40
0.35
0.30
0.25
0.20
0.15
0.10
10^5
10^6
10^7
10^8
10^9
number of nonzeros in L


CHOLMOD AMD ordering
write throughput (GB/s)
0.35
0.30
0.25
0.20
0.15
0.10
0.05
10^5
10^6
10^7
10^8
10^9
number of nonzeros in L


CHOLMOD Natural ordering
write throughput (GB/s)
0.6
0.5
0.4
0.3
0.2
0.1
10^5
10^6
10^7
10^8
10^9
number of nonzeros in L