Performance Evaluation of Concurrent Collections on High-Performance Multicore Computing Systems

Aparna Chandramowlishwaran, Richard Vuduc – Georgia Tech

Kathleen Knobe – Intel

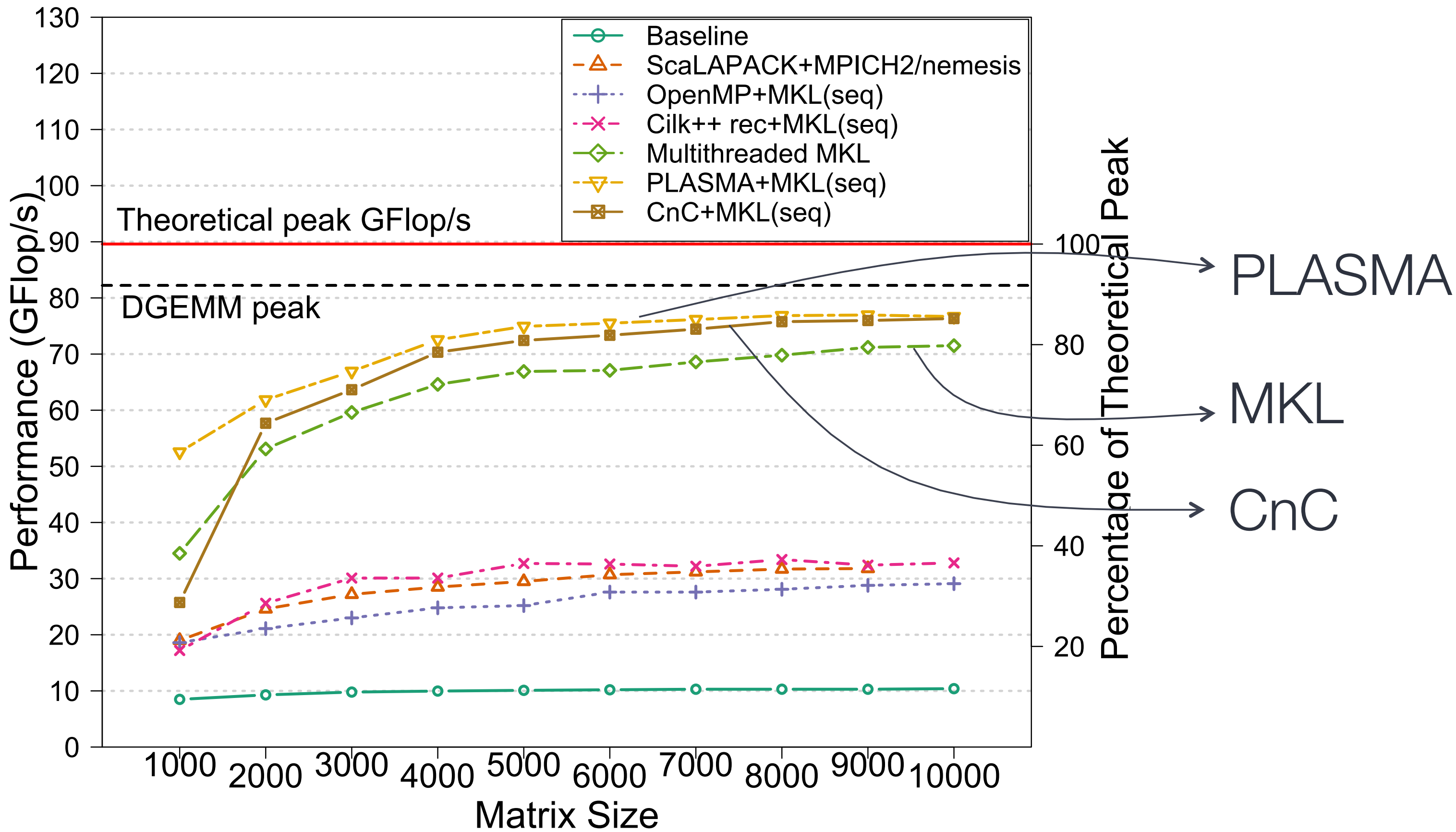
April 21, 2010

@ IPDPS, Atlanta

Georgia Tech College of Computing

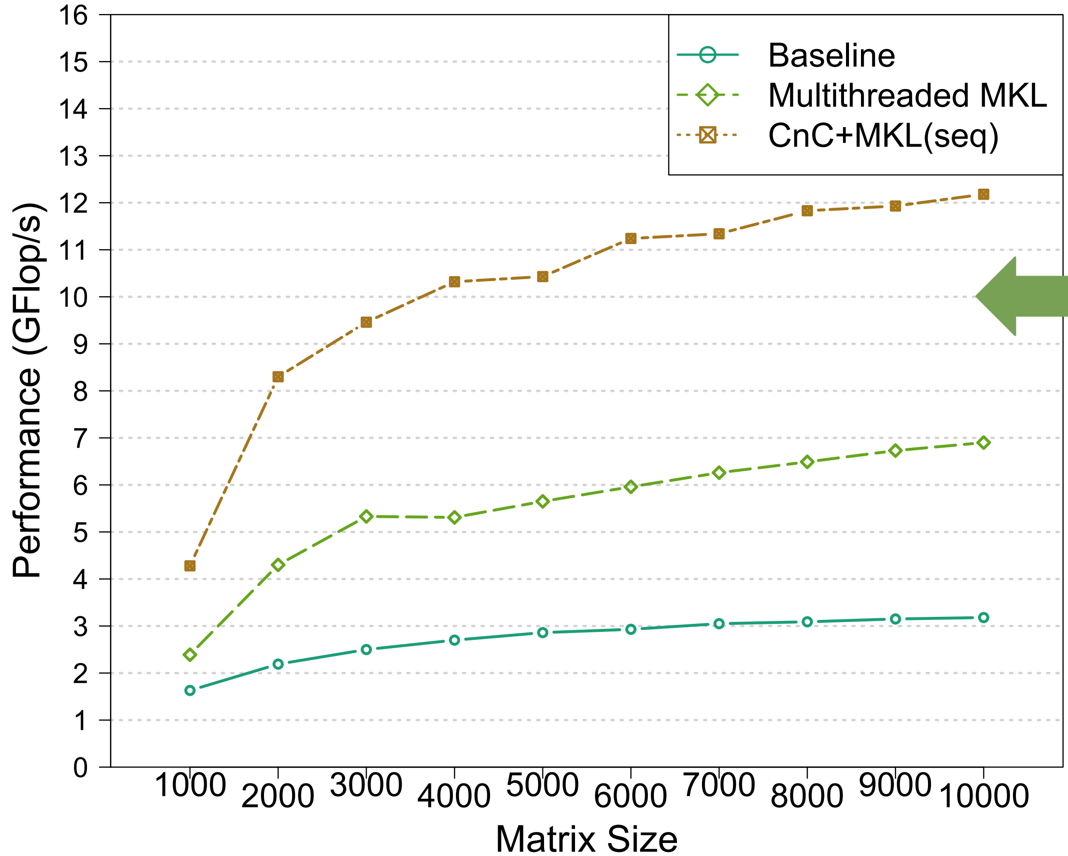
Computational Science and Engineering

hpcgarage



Why CnC? Cholesky Performance

Intel 2-socket x 4-core Nehalem @ 2.8 GHz + Intel MKL 10.2



~2x speedup over MKL

Why CnC? Eigensolver Performance

Intel 2-socket x 4-core Nehalem @ 2.8 GHz + Intel MKL 10.2

Motivation

- Fine-grained synchronization for multicore
  - Tile algorithms for DLA: *e.g.*, Buttari, *et al*. (2007); Chan, *et al*. (2007)
  - Manually tuned software for DLA: PLASMA, MAGMA, etc.
- No consensus on a programming model

- Our claim: Concurrent Collections (CnC) is suitable
- Study: Apply and evaluate CnC using PDLA examples

Main Contributions

- Evaluate performance potential of CnC for HPC
  - Analyze tiled (asynchronous) dense Cholesky in CnC
  - Present a novel, partially asynchronous dense symmetric eigensolver
- Compare performance against six alternative programming models
- Key observations
  - Feasible to express complex algorithms in CnC
  - CnC can match or exceed vendor and tuned library performance
  - Asynchronous algorithms perform better than bulk-synchronous variants

Outline

- Overview of the Concurrent Collections (CnC) language
- Asynchronous-parallel Cholesky and symmetric eigensolver in CnC
- Experimental results
- Summary

Concurrent Collections

CnC Model

Domain Expert: (person)
Only domain knowledge
No tuning knowledge

Application

- Semantic correctness
- Application constraints

Concurrent Collections

Tuning Expert: (person, runtime, static analysis)
No domain knowledge
Only tuning knowledge

- Architecture
- Parallelism
- Locality
- Overhead
- Load balancing
- Distribution among processors
- Scheduling within a processor

Mapping to target platform

Raises the level of abstraction for parallel programming

Concurrent Collections (CnC) programming model

- Program = components + scheduling constraints
  - Components: **Computation**, **control**, **data**
  - Constraints: **Relations** among components
  - No overwriting of data, no arbitrary serialization, and no side-effects
- Combines tuple-space, streaming, and dataflow models

To download CnC, see: whatif.intel.com

CnC example: Outer product

$$Z \leftarrow x \cdot y^T$$

CnC example: Outer product

$$Z \leftarrow x \cdot y^T$$

$$z_{i,j} \leftarrow x_i \cdot y_j$$

Example only; coarser grain may be more realistic in practice.

CnC example: Outer product

$$z_{i,j} \leftarrow x_i \cdot y_j$$

Collections: Static representation of dynamic ***instances***

CnC example: Outer product

$$z_{i,j} \leftarrow x_i \cdot y_j$$

Collections: Static representation of dynamic ***instances***

Step Unit of execution

*

Set of all (dynamic) multiplications

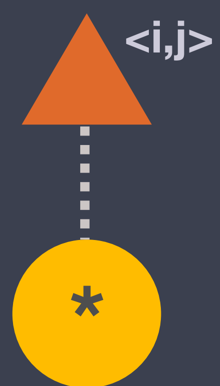
CnC example: Outer product

$$z_{i,j} \leftarrow x_i \cdot y_j$$

Collections: Static representation of dynamic ***instances***

Step Unit of execution

Tag Control



<*a*, *b*, …> = tuple of tag components

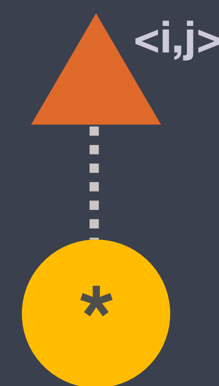
CnC example: Outer product

$$z_{i,j} \leftarrow x_i \cdot y_j$$

Collections: Static representation of dynamic ***instances***

Step Unit of execution

Tag Control



Says ***whether***, not ***when***, step executes

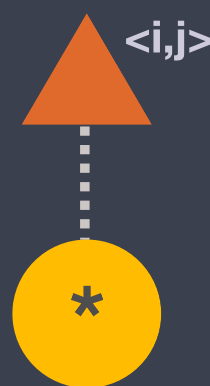
CnC example: Outer product

$$z_{i,j} \leftarrow x_i \cdot y_j$$

Collections: Static representation of dynamic ***instances***

Step Unit of execution

Tag Control



Tags ***prescribe*** steps

CnC example: Outer product

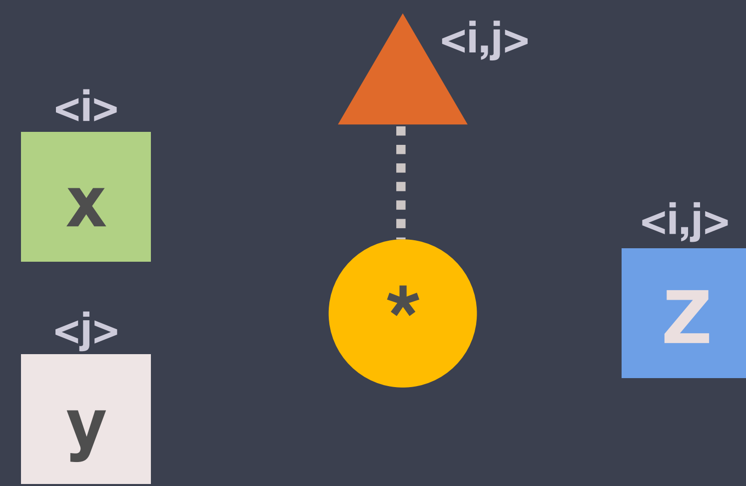
$$z_{i,j} \leftarrow x_i \cdot y_j$$

Collections: Static representation of dynamic ***instances***

Step — Unit of execution

Tag — Control

Item — Data



CnC example: Outer product

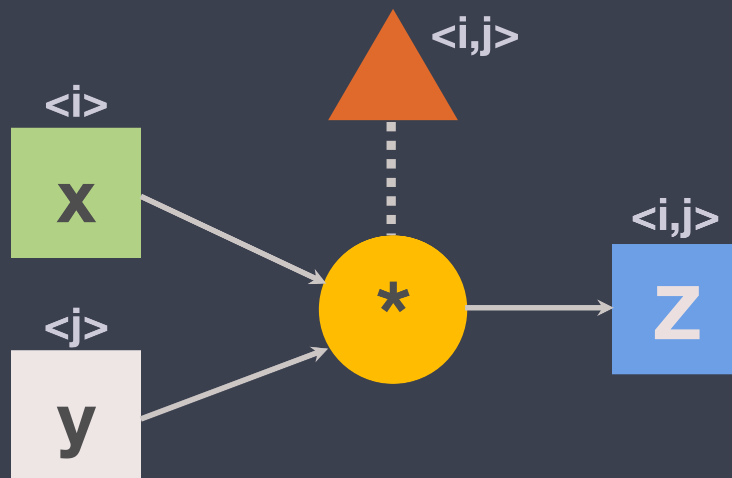
$$z_{i,j} \leftarrow x_i \cdot y_j$$

Collections: Static representation of dynamic ***instances***

Step — Unit of execution

Tag — Control

Item — Data



→ shows producer/consumer relations

CnC example: Outer product

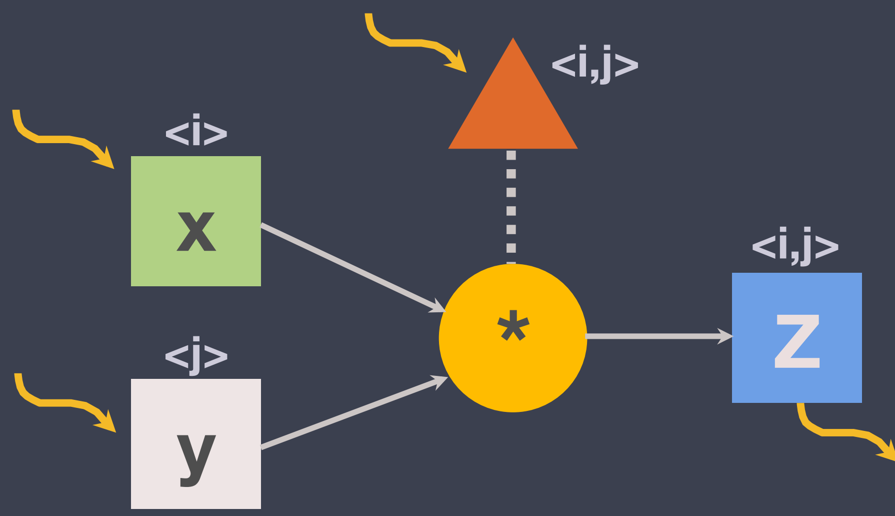
$$z_{i,j} \leftarrow x_i \cdot y_j$$

Collections: Static representation of dynamic ***instances***

Step Unit of execution

Tag Control

Item Data

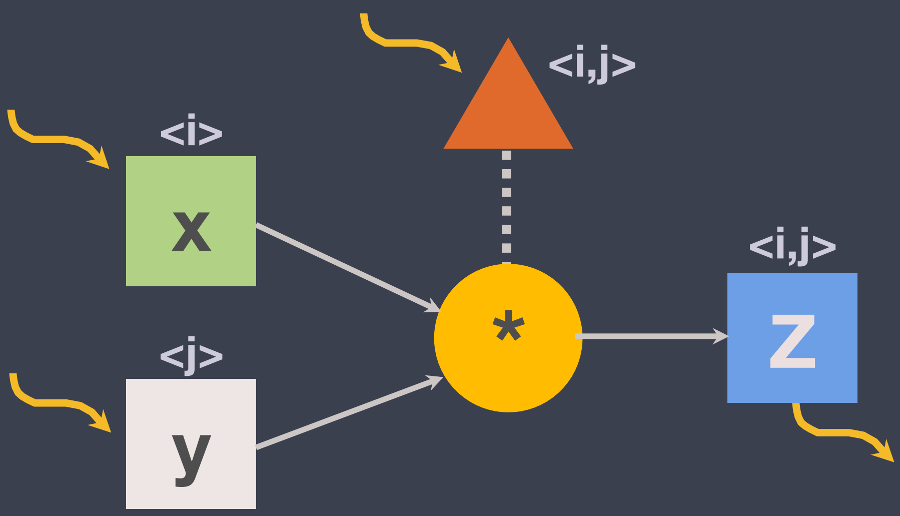


“**Environment**” may produce/consume

Essential properties of a CnC program

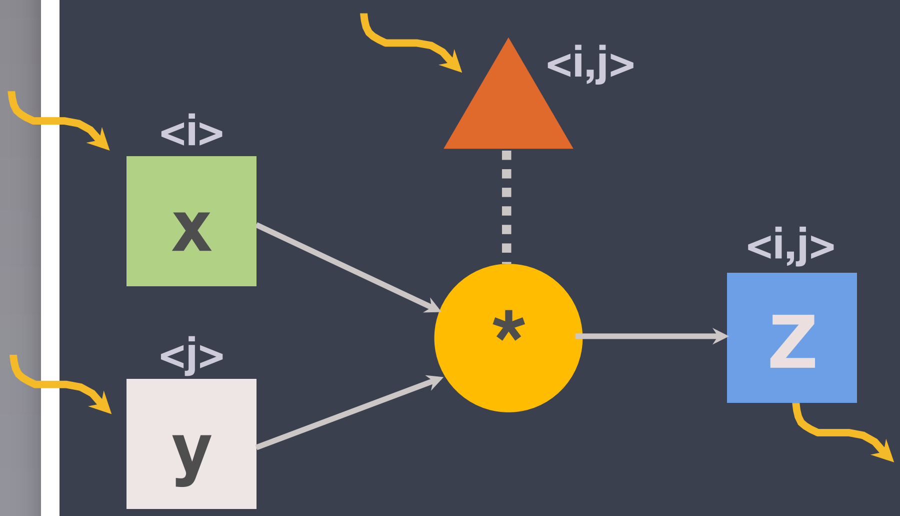
- No overwriting ⇒ race-free (dynamic single assignment)
- No arbitrary serialization (avoids analysis)
- Steps are side-effect free (functional)

$$z_{i,j} \leftarrow x_i \cdot y_j$$



Execution model

$$z_{i,j} \leftarrow x_i \cdot y_j$$



Recall: Outer product example

Execution model

$$z_{i,j} \leftarrow x_i \cdot y_j$$

- Tag <i=2, j=5> ***available***

Execution model

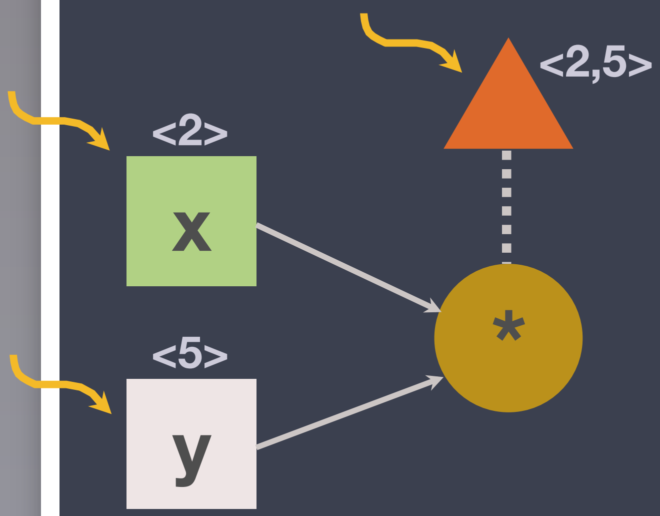
$$z_{i,j} \leftarrow x_i \cdot y_j$$

- Tag <i=2, j=5> *available*

⇒ Step ***prescribed***

Execution model

$$z_{i,j} \leftarrow x_i \cdot y_j$$

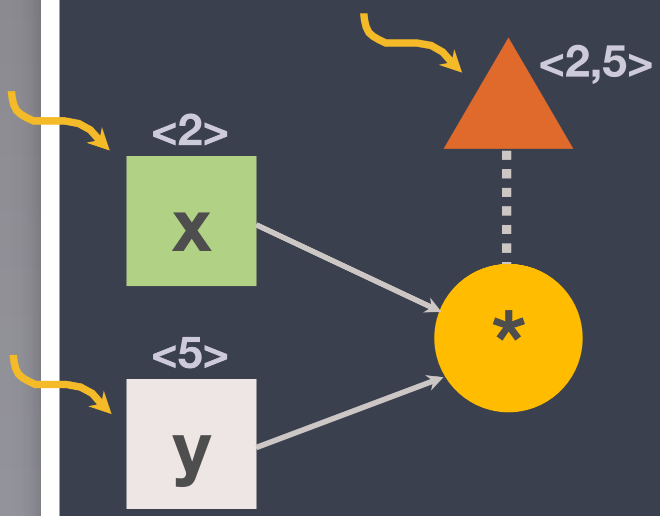


- Tag <2,5> *available*
  ⇒ Step *prescribed*

- Items x:<2>, y:<5> available
  ⇒ Step ***inputs-available***

Execution model

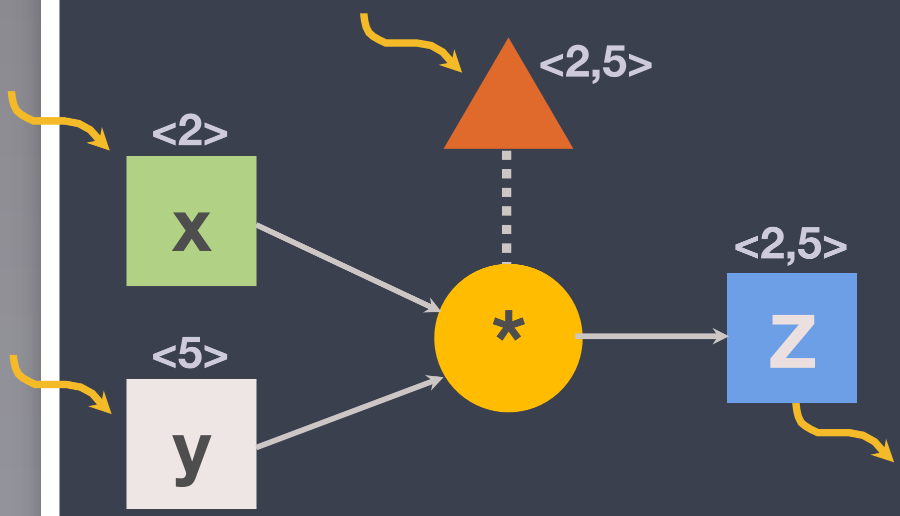
$$z_{i,j} \leftarrow x_i \cdot y_j$$



- Tag <2,5> *available*
  ⇒ Step *prescribed*
- Items x:<2>, y:<5> available
  ⇒ Step *inputs-available*
- *Prescribed* + *inputs-available* ⇒ ***enabled***

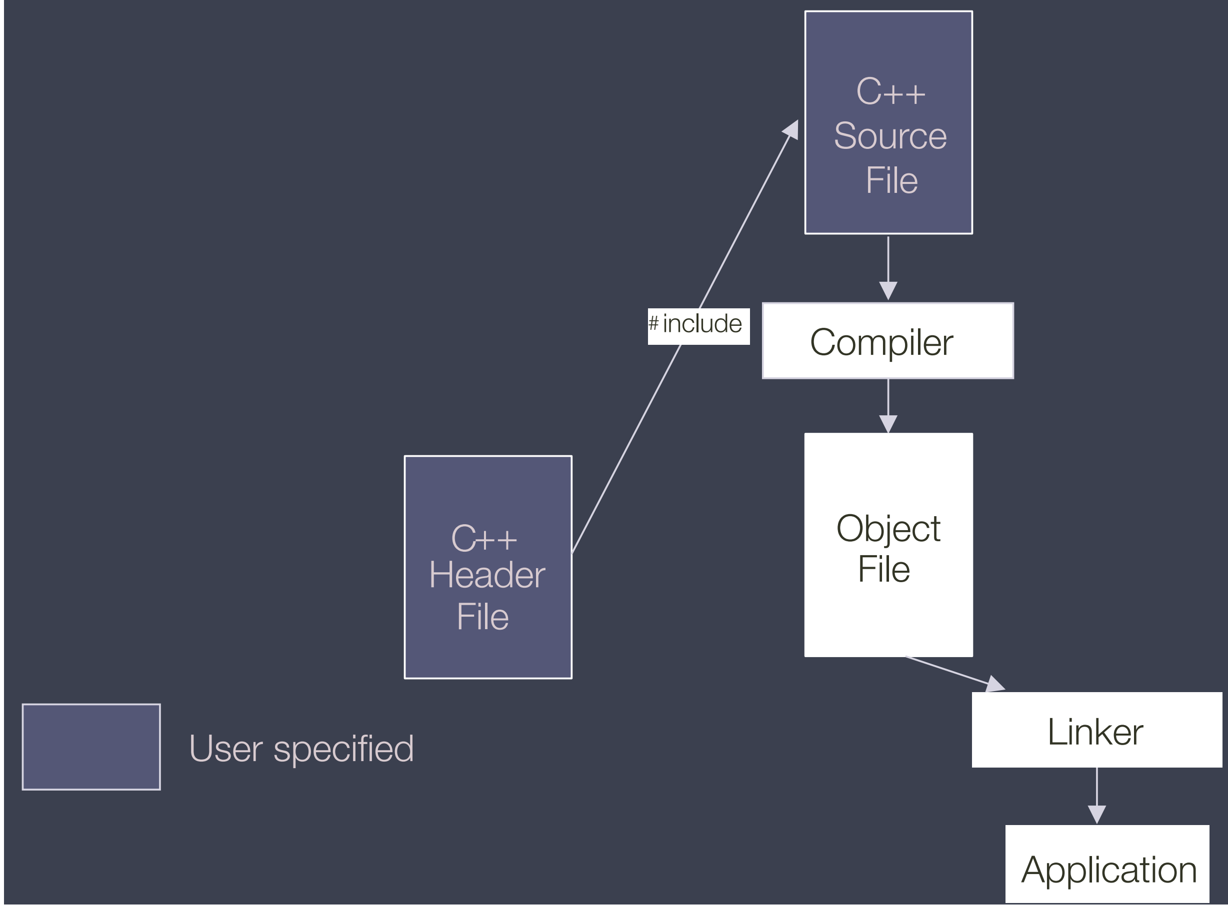
Execution model

$$z_{i,j} \leftarrow x_i \cdot y_j$$

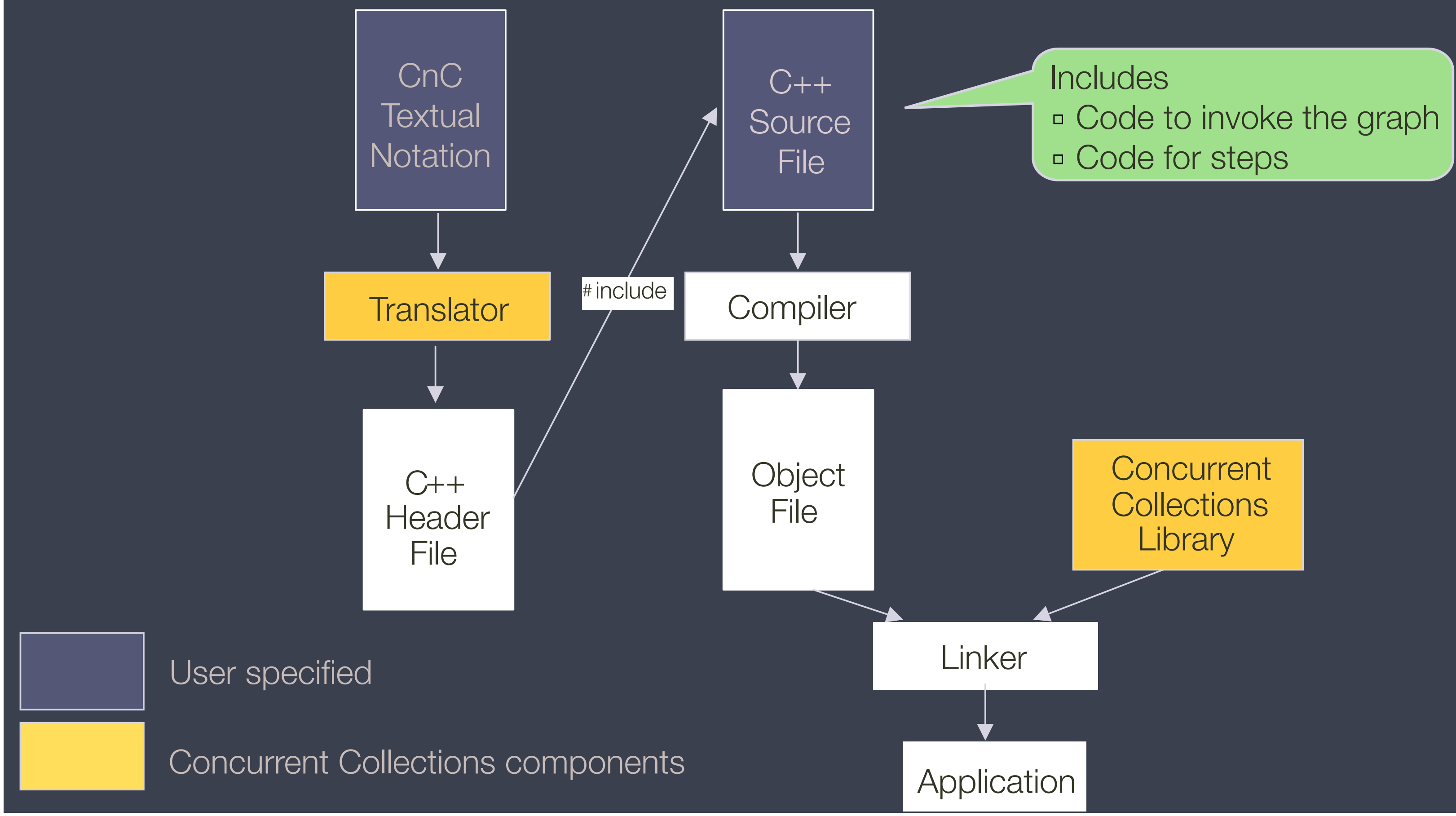


- Tag <2,5> *available*
  ⇒ Step *prescribed*
- Items x:<2>, y:<5> available
  ⇒ Step *inputs-available*
- *Prescribed + inputs-available* ⇒ *enabled*
- Executes ⇒ Z:<2,5> ***available***

Conventional Build Model



CnC Build Model



CnC Run-time System

- Built on top of Intel Threading Building Blocks (TBB)
  - Implements Cilk-style work stealing scheduler
  - Work queues use LIFO, but FIFO and other strategies in development
- Other run times possible
  - DEC/HP TStreams on MPI; Rice U. Habanero uses Java threads
  - Intel-specific issues with queuing (more later)

We use Intel CnC v0.3 in this study

Cholesky Factorization

Tile Cholesky: $A \rightarrow L \cdot L^T$

Buttari, *et al*. (2007)

Iteration *k*: // Over diagonal tiles

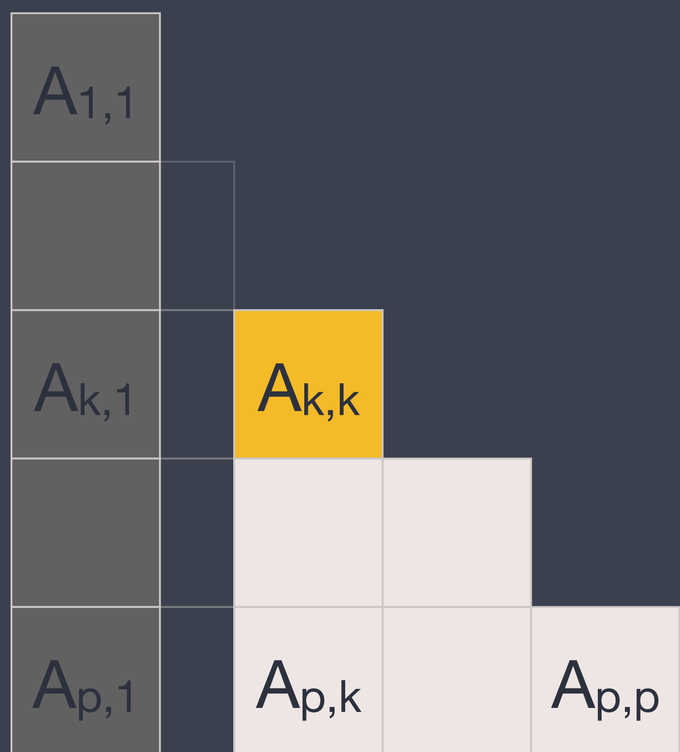
SeqCholesky ($L_{k,k} \leftarrow A_{k,k}$)

Trisolve ($L_{k+1:p,k} \leftarrow A_{k+1:p,k},\ L_{k,k}$)

Update ($A_{k+1:p,k+1:p} \leftarrow L_{k+1:p,k},\ A_{k+1:p,k+1:p}$)

Tile Cholesky: $A \to L \cdot L^T$

Buttari, *et al*. (2007)



Iteration *k*: // Over diagonal tiles

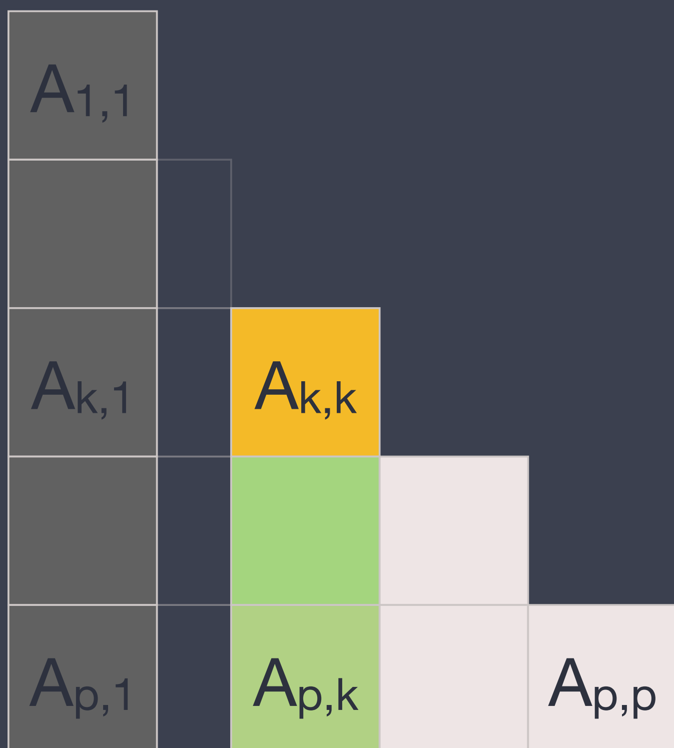
SeqCholesky ($L_{k,k} \leftarrow A_{k,k}$)

Trisolve ($L_{k+1:p,k} \leftarrow A_{k+1:p,k},\ L_{k,k}$)

Update ($A_{k+1:p,k+1:p} \leftarrow L_{k+1:p,k},\ A_{k+1:p,k+1:p}$)

Tile Cholesky: $A \to L \cdot L^T$

Buttari, *et al*. (2007)



Iteration k: // Over diagonal tiles

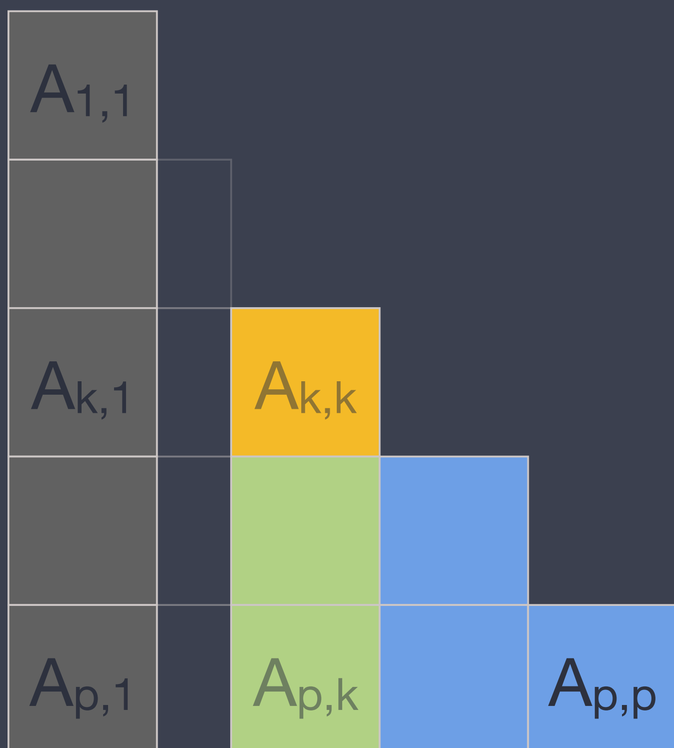
SeqCholesky ($L_{k,k} \leftarrow A_{k,k}$)

Trisolve ($L_{k+1:p,k} \leftarrow A_{k+1:p,k}, L_{k,k}$)

Update ($A_{k+1:p,k+1:p} \leftarrow L_{k+1:p,k}, A_{k+1:p,k+1:p}$)

Tile Cholesky: $A \to L \cdot L^T$

Buttari, *et al*. (2007)



Iteration *k*: // Over diagonal tiles

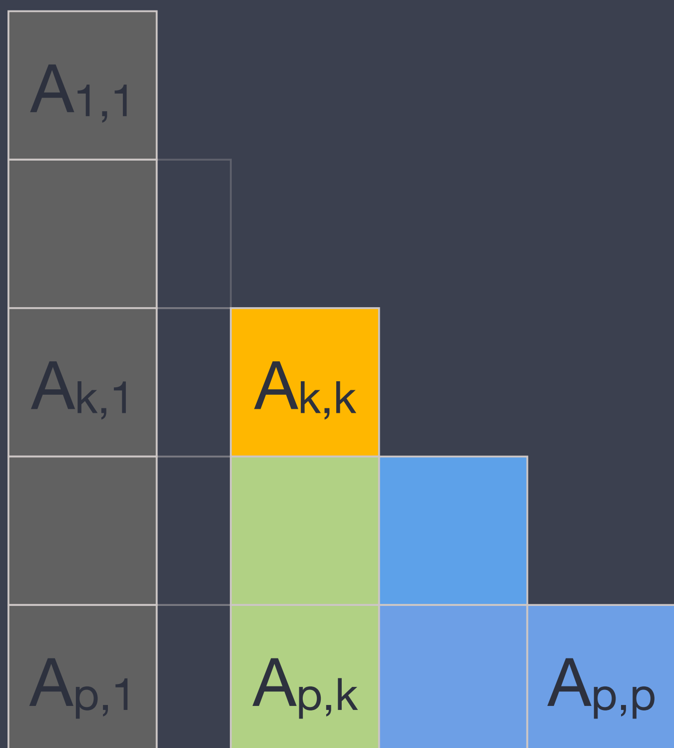
SeqCholesky ($L_{k,k} \leftarrow A_{k,k}$)

Trisolve ($L_{k+1:p,k} \leftarrow A_{k+1:p,k},\ L_{k,k}$)

Update ($A_{k+1:p,k+1:p} \leftarrow L_{k+1:p,k},\ A_{k+1:p,k+1:p}$)

Tile Cholesky: $A \to L \cdot L^T$

Buttari, *et al*. (2007)



Iteration *k*: // Over diagonal tiles

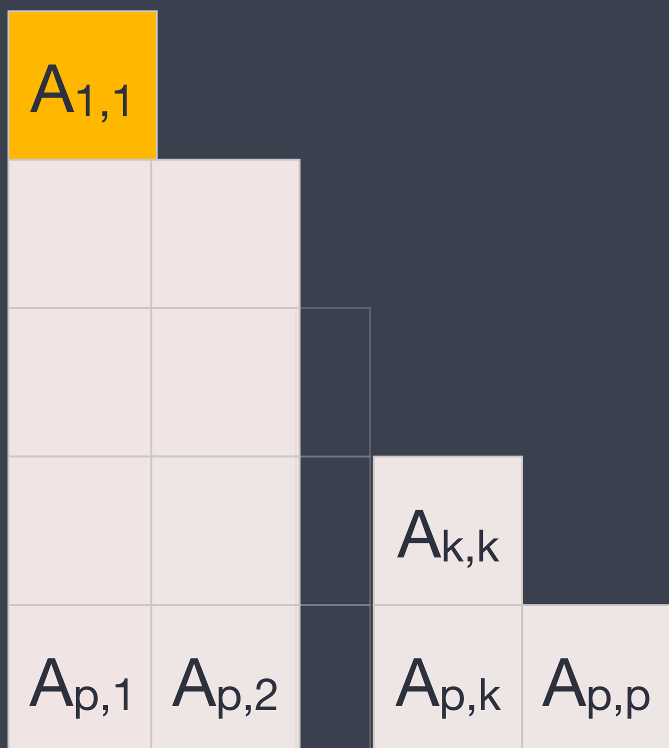
SeqCholesky ($L_{k,k} \leftarrow A_{k,k}$)

Trisolve ($L_{k+1:p,k} \leftarrow A_{k+1:p,k},\ L_{k,k}$)

Update ($A_{k+1:p,k+1:p} \leftarrow L_{k+1:p,k},\ A_{k+1:p,k+1:p}$)

Asynchronous Tile Cholesky

Buttari, *et al*. (2007)



Iteration *k*: // Over diagonal tiles

SeqCholesky ($L_{k,k} \leftarrow A_{k,k}$)

Trisolve ($L_{k+1:p,k} \leftarrow A_{k+1:p,k}, L_{k,k}$)

Update ($A_{k+1:p,k+1:p} \leftarrow L_{k+1:p,k}, A_{k+1:p,k+1:p}$)

Consider the case k=1 and number of threads is 2

Asynchronous Tile Cholesky

Buttari, *et al*. (2007)

Iteration k: // Over diagonal tiles

SeqCholesky ($L_{k,k} \leftarrow A_{k,k}$)

Trisolve ($L_{k+1:p,k} \leftarrow A_{k+1:p,k}, L_{k,k}$)

Update ($A_{k+1:p,k+1:p} \leftarrow L_{k+1:p,k}, A_{k+1:p,k+1:p}$)

Consider the case k=1 and number of threads is 2

Asynchronous Tile Cholesky

Buttari, *et al*. (2007)

Iteration *k*: // Over diagonal tiles

SeqCholesky ($L_{k,k} \leftarrow A_{k,k}$)

Trisolve ($L_{k+1:p,k} \leftarrow A_{k+1:p,k}, L_{k,k}$)

Update ($A_{k+1:p,k+1:p} \leftarrow L_{k+1:p,k}, A_{k+1:p,k+1:p}$)

Consider the case k=1 and number of threads is 2

Asynchronous Tile Cholesky

Buttari, *et al*. (2007)

Iteration k: // Over diagonal tiles

SeqCholesky ($L_{k,k} \leftarrow A_{k,k}$)

Trisolve ($L_{k+1:p,k} \leftarrow A_{k+1:p,k}, L_{k,k}$)

Update ($A_{k+1:p,k+1:p} \leftarrow L_{k+1:p,k}, A_{k+1:p,k+1:p}$)

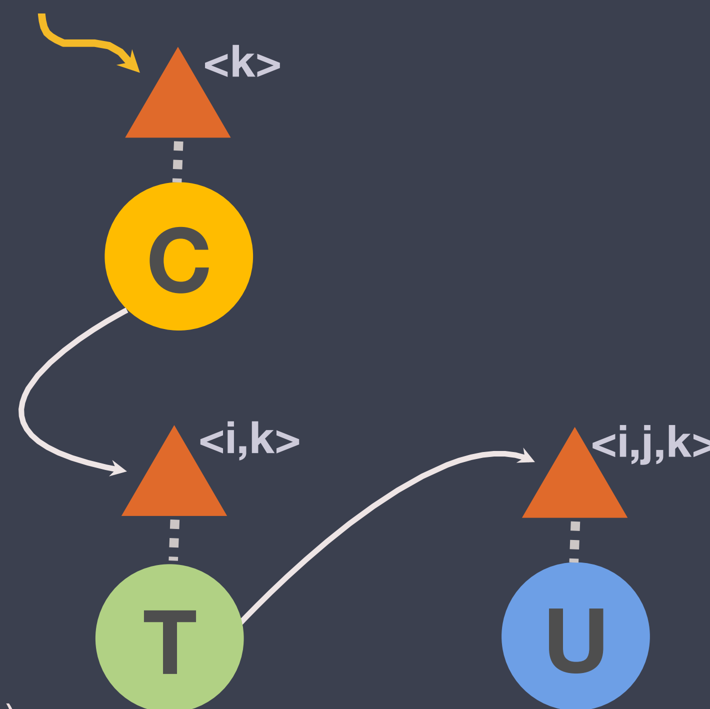
Consider the case k=1 and number of threads is 2

Tile Cholesky in CnC

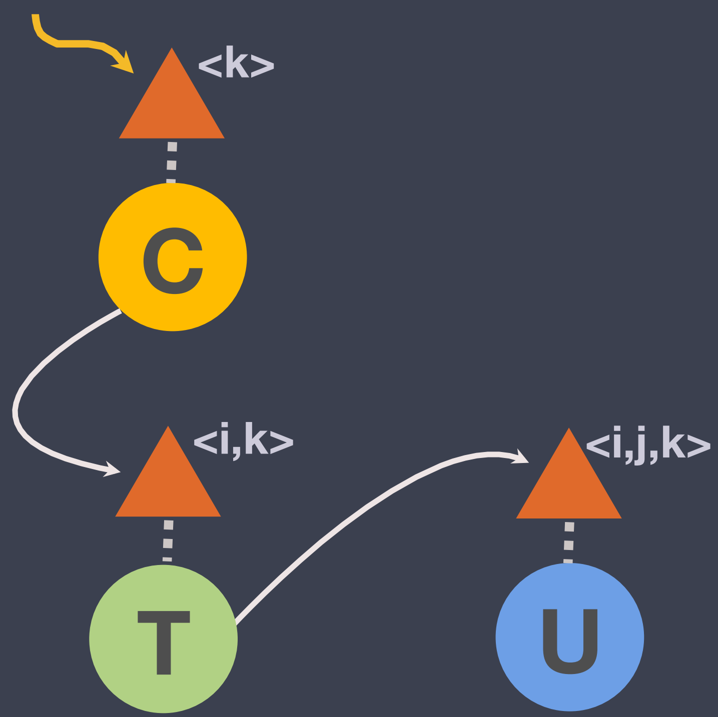
SeqCholesky ($L_{k,k} \leftarrow A_{k,k}$)

Trisolve ($L_{k+1:p,k} \leftarrow A_{k+1:p,k},\ L_{k,k}$)

Update ($A_{k+1:p,k+1:p} \leftarrow L_{k+1:p,k},\ A_{k+1:p,k+1:p}$)

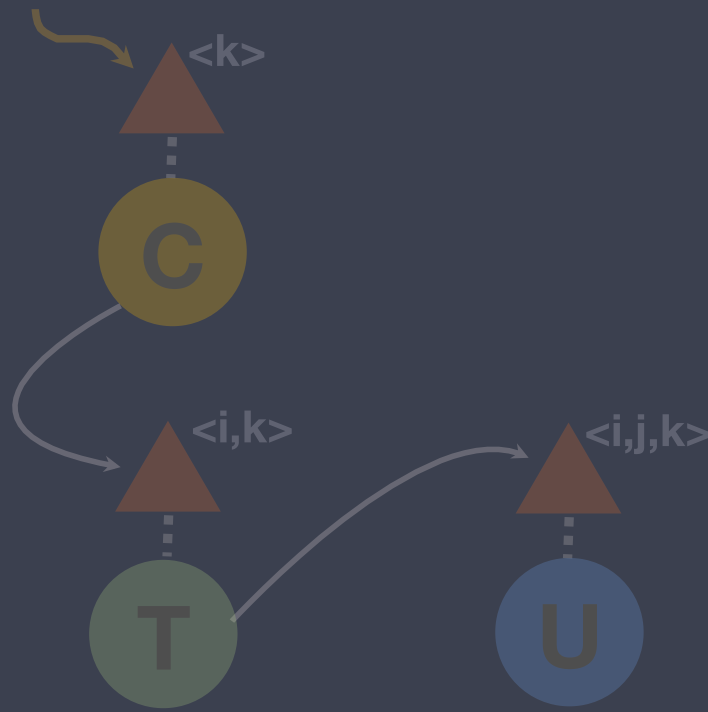


Other arrangements possible, *e.g.*, pre-generate all tags.



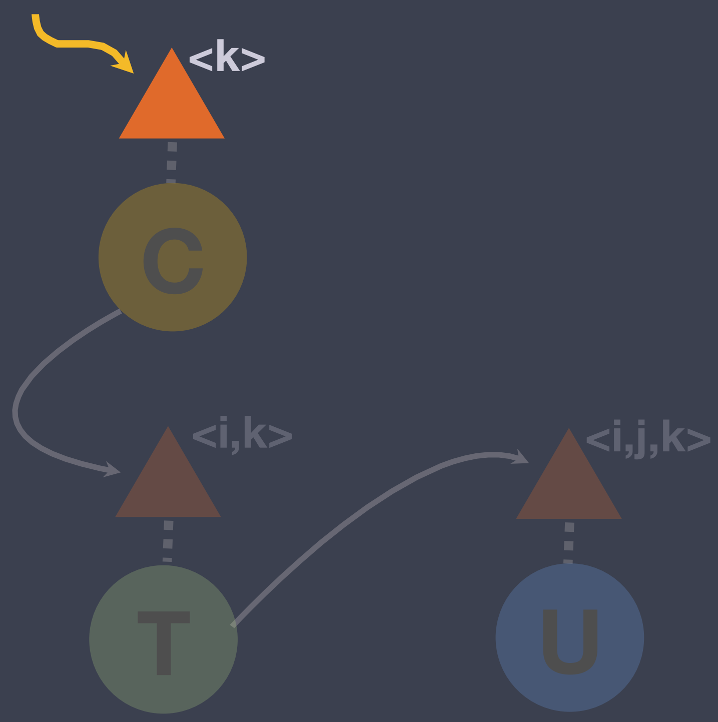
CnC Textual Notation

```
// Item: Matrix L, tagged by <k, j, i>
   [BlockedMatrix<double>* L: int, int, int];
```



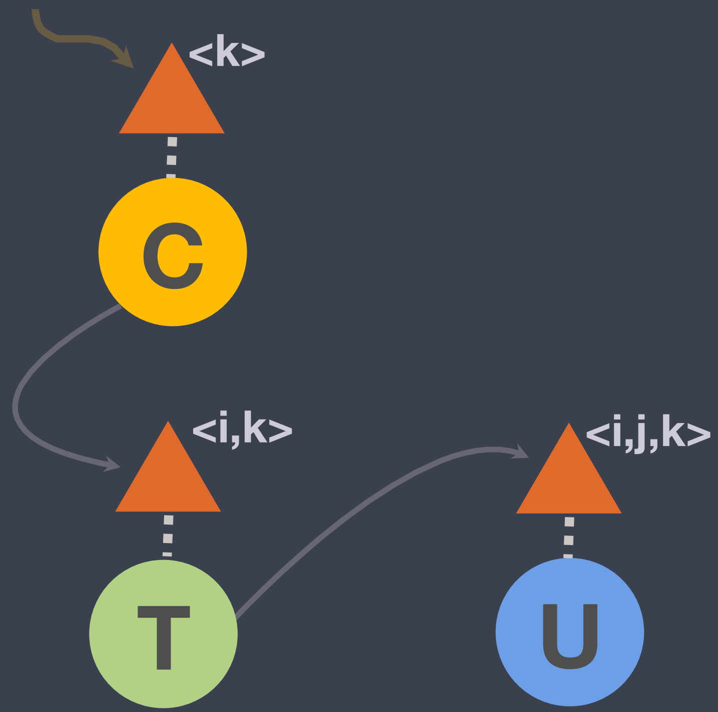
CnC Textual Notation

```
// Item: Matrix L, tagged by <k, j, i>
   [BlockedMatrix<double>* L: int, int, int];

// Input:
   env → [L], <C_tag: k>;
```



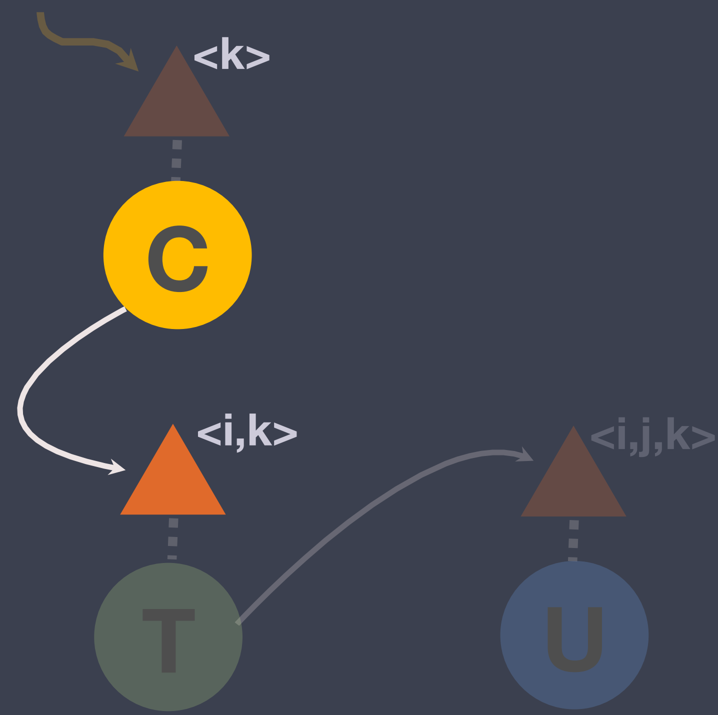
CnC Textual Notation

```
// Item: Matrix L, tagged by <k, j, i>
   [BlockedMatrix<double>* L: int, int, int];

// Input:
   env → [L], <C_tag: k>;

// Prescription relations:
   <C_tag: k> :: (C: k);
   <T_tag: i, k> :: (T: i, k);
   <U_tag: j, i, k> :: (U: j, i, k);
```



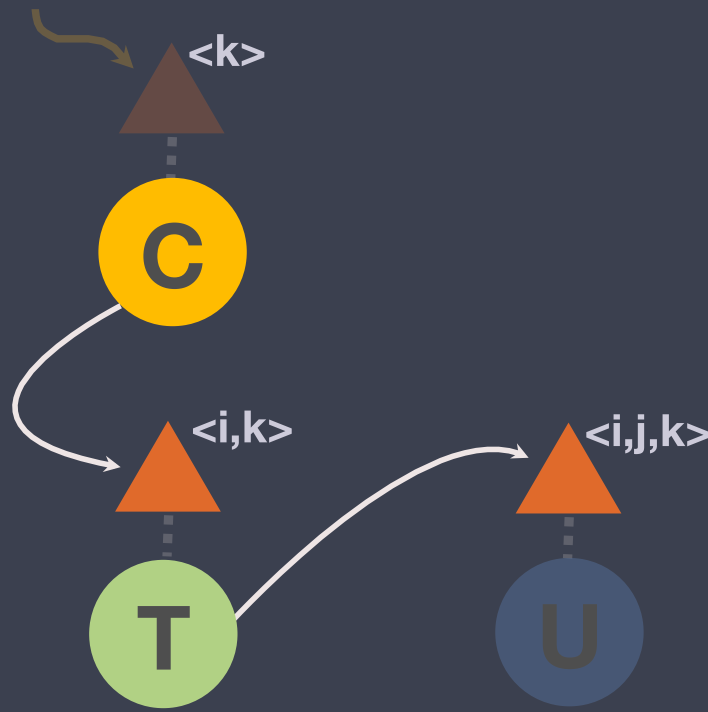
CnC Textual Notation

```
// Item: Matrix L, tagged by <k, j, i>
   [BlockedMatrix<double>* L: int, int, int];


// Input:
   env → [L], <C_tag: k>;


// Prescription relations:
   <C_tag: k> :: (C: k);
   <T_tag: i, k> :: (T: i, k);
   <U_tag: j, i, k> :: (U: j, i, k);


// Producer/consumer relations:
   [L] → (C: k);
   (C: k) → [L], [T_tag: i, k];
```



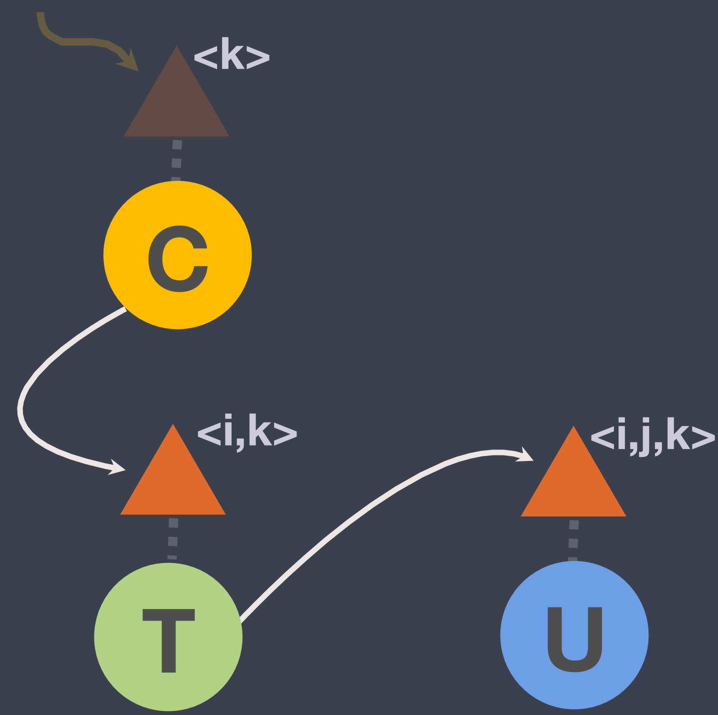
CnC Textual Notation

```
// Item: Matrix L, tagged by <k, j, i>
   [BlockedMatrix<double>* L: int, int, int];


// Input:
   env → [L], <C_tag: k>;


// Prescription relations:
   <C_tag: k> :: (C: k);
   <T_tag: i, k> :: (T: i, k);
   <U_tag: j, i, k> :: (U: j, i, k);


// Producer/consumer relations:
   [L] → (C: k);
   (C: k) → [L], [T_tag: i, k];

   [L] → (T: i, k);
   (T: i, k) → [L], [U_tag: j, i, k];
```



CnC Textual Notation

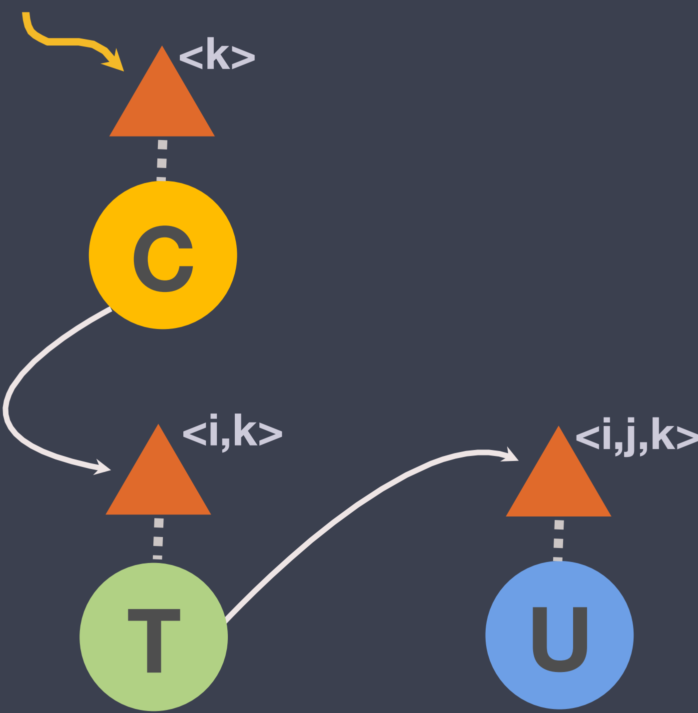
```
// Item: Matrix L, tagged by <k, j, i>
   [BlockedMatrix<double>* L: int, int, int];


// Input:
   env → [L], <C_tag: k>;


// Prescription relations:
   <C_tag: k> :: (C: k);
   <T_tag: i, k> :: (T: i, k);
   <U_tag: j, i, k> :: (U: j, i, k);


// Producer/consumer relations:
   [L] → (C: k);
   (C: k) → [L], [T_tag: i, k];

   [L] → (T: i, k);
   (T: i, k) → [L], [U_tag: j, i, k];

   [L] → (U: j, i, k);
   (U: j, i, k) → [L];
```



CnC Textual Notation

CnC Textual Notation

```
// Item: Matrix L, tagged by <k, j, i>
   [BlockedMatrix<double>* L: int, int, int];

// Input:
   env → [L], <C_tag: k>;

// Prescription relations:
   <C_tag: k> :: (C: k);
   <T_tag: i, k> :: (T: i, k);
   <U_tag: j, i, k> :: (U: j, i, k);

// Producer/consumer relations:
   [L] → (C: k);
   (C: k) → [L], [T_tag: i, k];

   [L] → (T: i, k);
   (T: i, k) → [L], [U_tag: j, i, k];

   [L] → (U: j, i, k);
   (U: j, i, k) → [L];

// Output:
   [L] → env;
```



```
StepReturnValue_t T (
     cholesky_graph_t& graph,
     const Tag_t& TS_tag)
{
   // For each input item in this step
   // retrieve the item using the proper tag
   BlockedMatrix<double>* A_block =
           graph.L.Get (Tag_t ());
   BlockedMatrix<double>* Li_block =
           graph.L.Get (Tag_t ());

   // Step implementation logic goes here

   // For each output item for this step
   // put the new item using the proper tag
   graph.L.Put (Tag_t (),);

   return CNC_Success;
}
```

Step code written in a sequential base language

- Grey color denote automatically generated stubs

Intel's implementation uses C++; Rice University's uses Java (Habanero)

```
StepReturnValue_t T (
     cholesky_graph_t& graph,
     const Tag_t& TS_tag)
{
   char uplo = 'l', side = 'r'
   char transa = 't', diag = 'n';
   double alpha = 1;



   // For each input item in this step
   // retrieve the item using the proper tag
   BlockedMatrix<double>* A_block =
           graph.L.Get (Tag_t ());
   BlockedMatrix<double>* Li_block =
           graph.L.Get (Tag_t ());



   // Step implementation logic goes here
   dtrsm (&side, &uplo, &transa, &diag, &b, &b,
           &alpha, Li_block, &b, A_block, &b);

   // For each output item for this step
   // put the new item using the proper tag
   graph.L.Put (Tag_t (),);

   return CNC_Success;
}
```

Step code written in a sequential base language

- Grey color denote automatically generated stubs
- User fills in the blue text
  - Sequential step code

Intel's implementation uses C++; Rice University's uses Java (Habanero)

```
StepReturnValue_t T (
     cholesky_graph_t& graph,
     const Tag_t& TS_tag)
{
   char uplo = 'l', side = 'r'
   char transa = 't', diag = 'n';
   double alpha = 1;

   int k = TS_tag[0];
   int j = TS_tag[1];

   // For each input item in this step
   // retrieve the item using the proper tag
   BlockedMatrix<double>* A_block =
           graph.L.Get (Tag_t (j, k, k));
   BlockedMatrix<double>* Li_block =
           graph.L.Get (Tag_t (k, k, k+1));

   // Get block size
   int b = A_block->getBlockSize ();

   // Step implementation logic goes here
   dtrsm (&side, &uplo, &transa, &diag, &b, &b,
           &alpha, Li_block, &b, A_block, &b);

   // For each output item for this step
   // put the new item using the proper tag
   graph.L.Put (Tag_t (j, k, k+1), A_block);

   return CNC_Success;
}
```

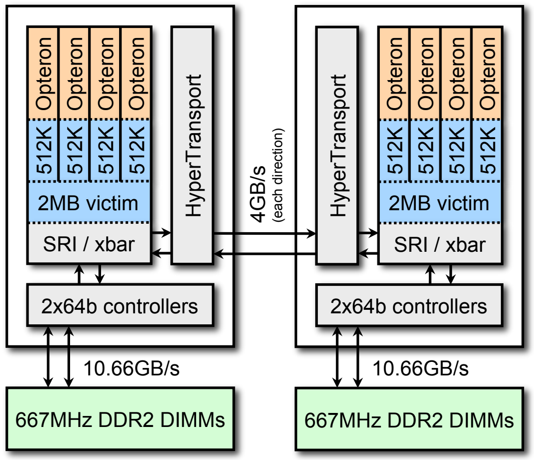
Step code written in a sequential base language

- Grey color denote automatically generated stubs
- User fills in the blue/green text
  - Input: *Get* API
  - Output: *Put* API
  - Sequential step code

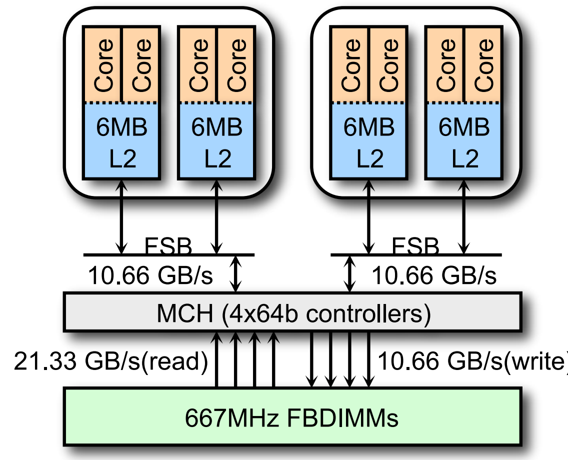
Intel's implementation uses C++; Rice University's uses Java (Habanero)

Architectural Summary

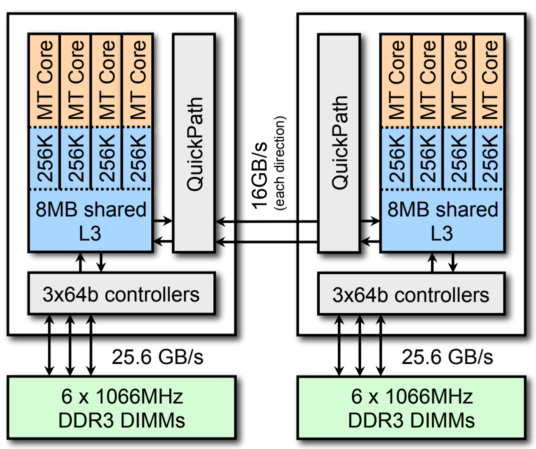
AMD Opteron 8350
(Barcelona)



Intel Xeon E5405
(Harpertown)



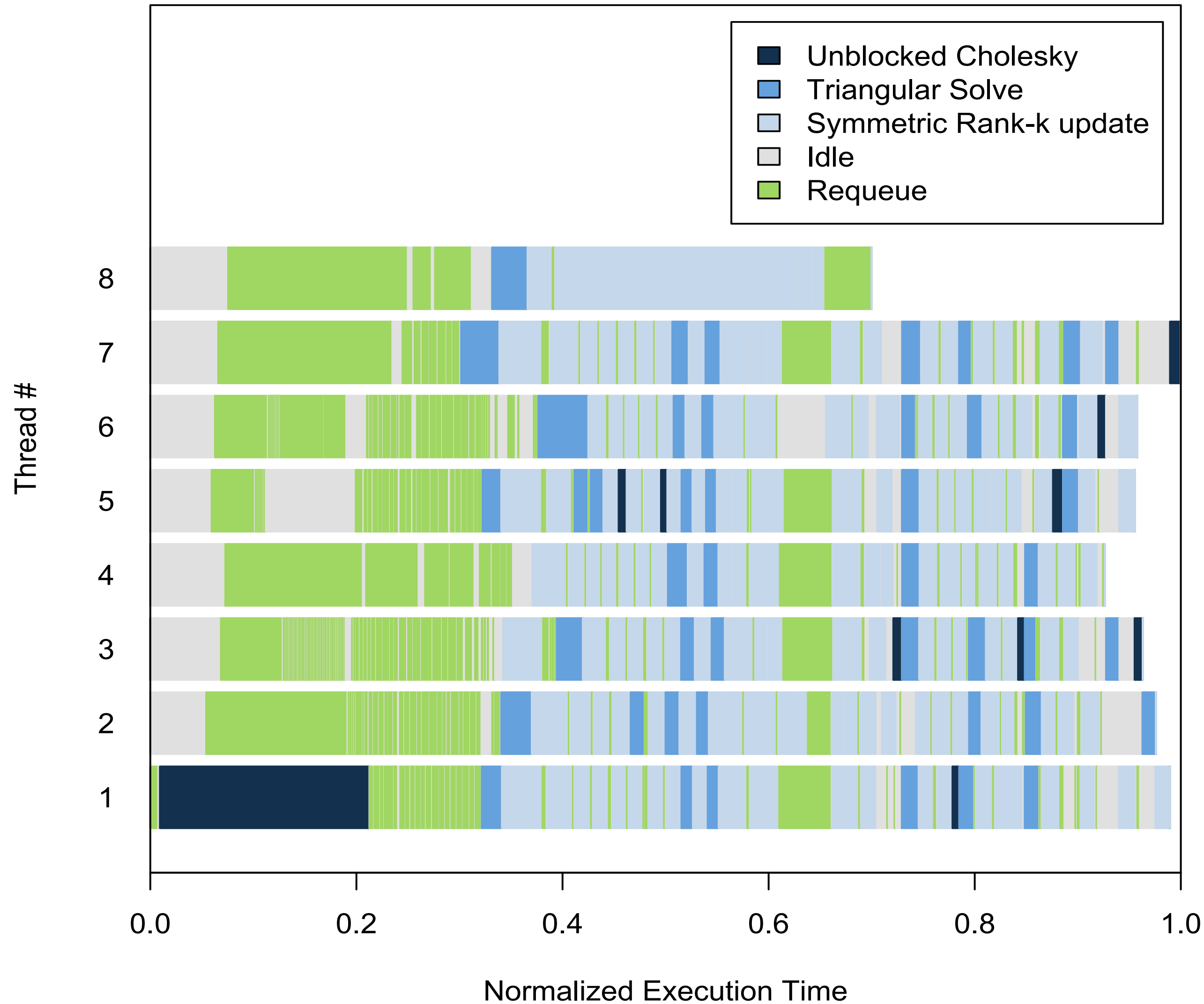
Intel Xeon X5560
(Nehalem)



Architectural Summary

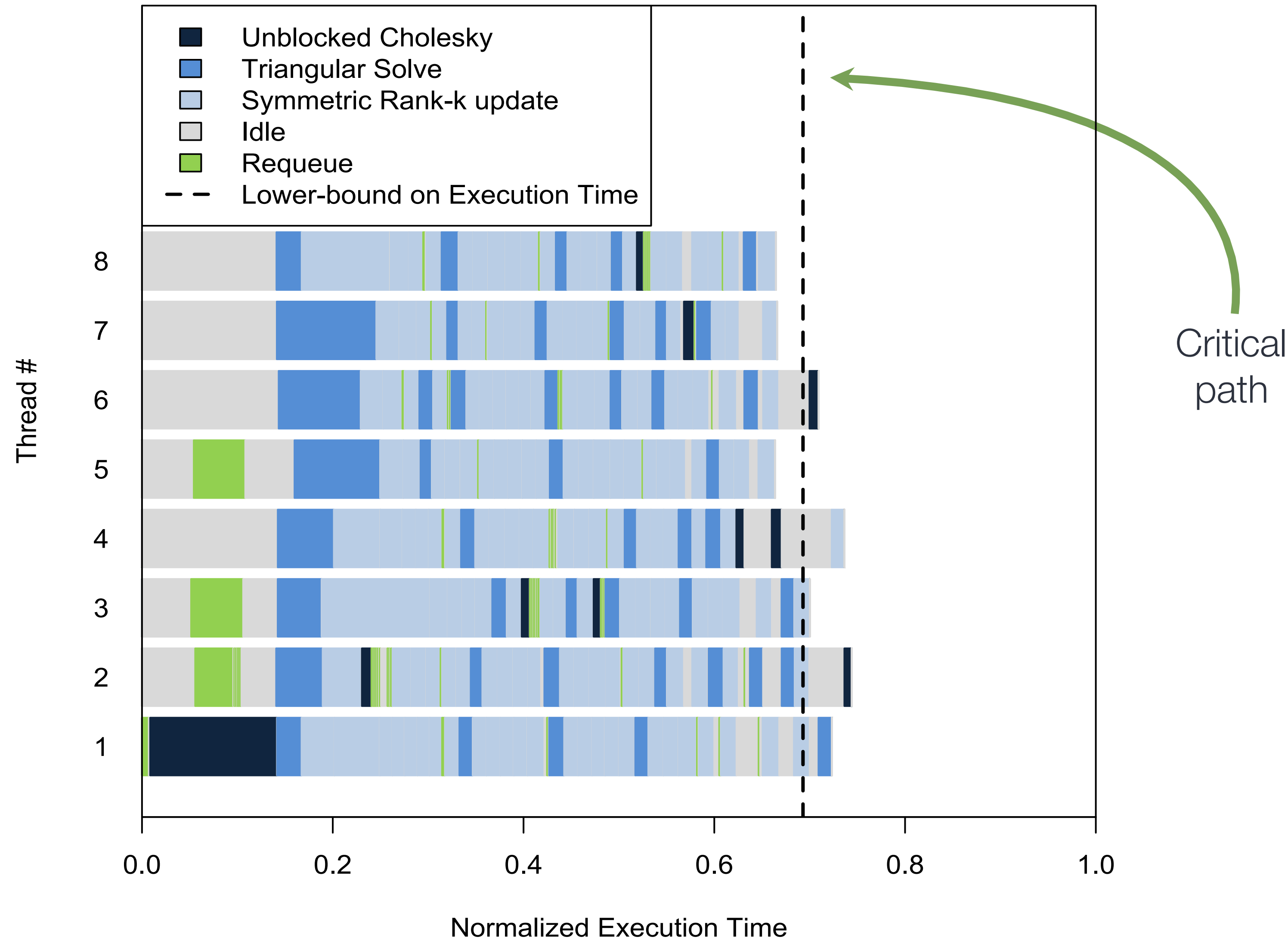
- Nehalem and Barcelona are NUMA
- Nehalem is hyper-threaded

Performance analysis of Cholesky Factorization



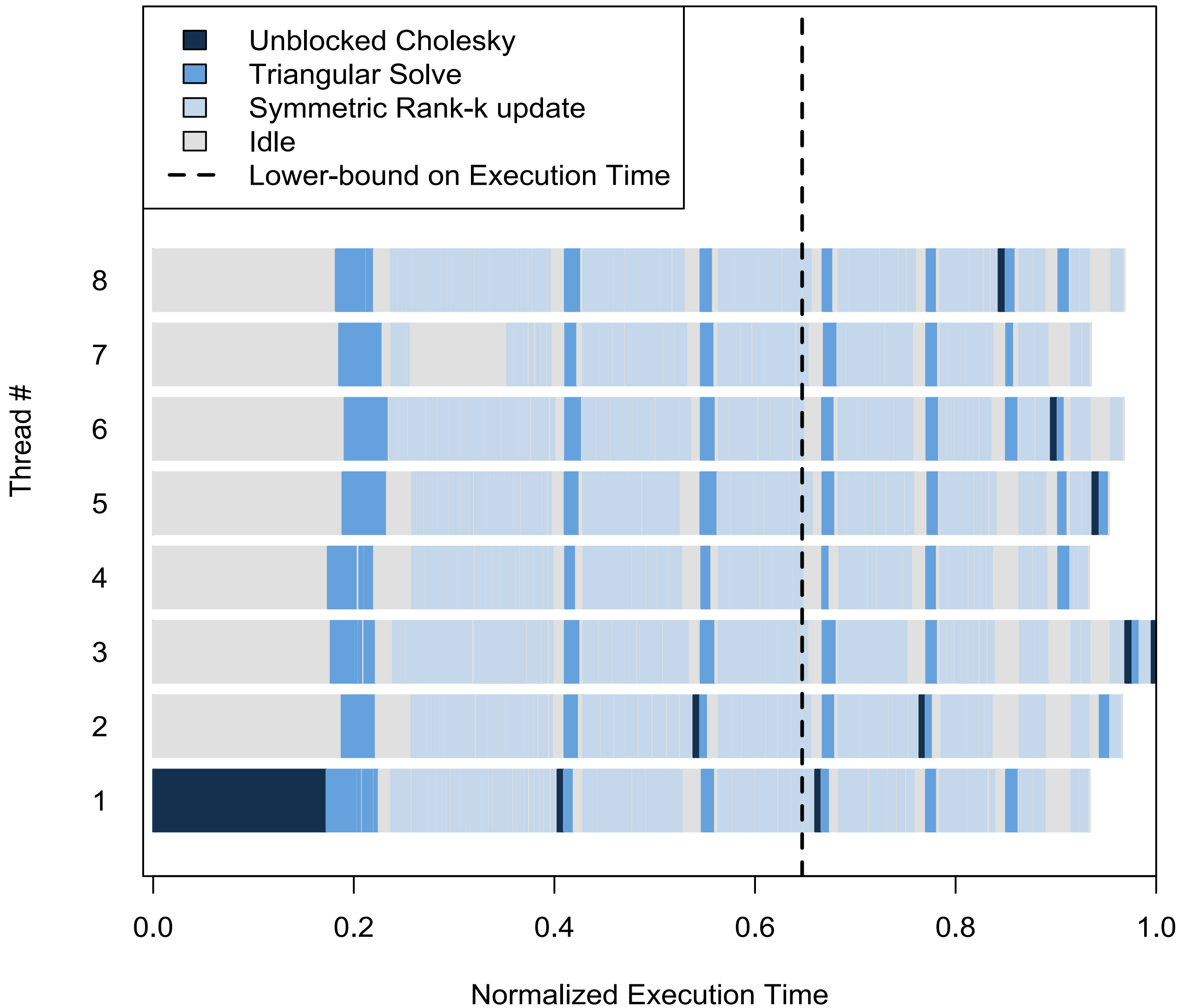
CnC-based Cholesky timeline (n=1000 with requeue):

Intel 2-socket x 4-core Harpertown @ 2 GHz + Intel MKL 10.1 for sequential components



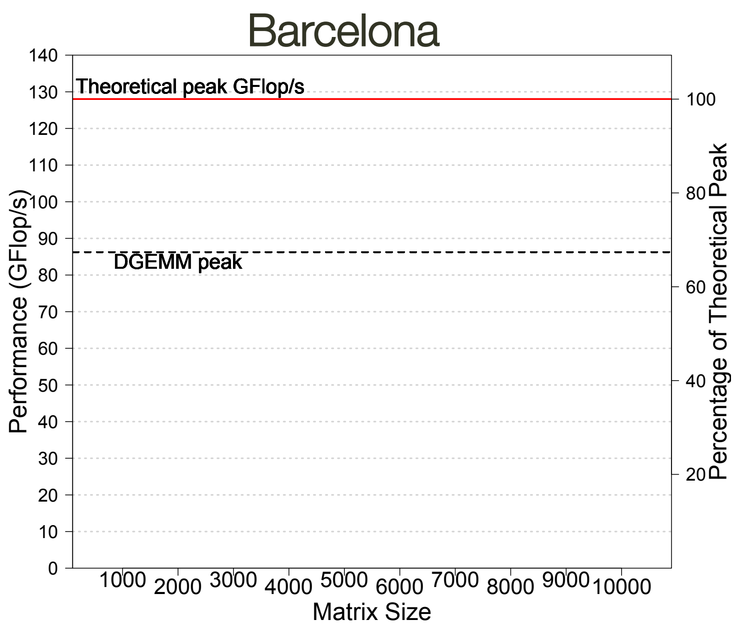
CnC-based Cholesky timeline (n=1000):

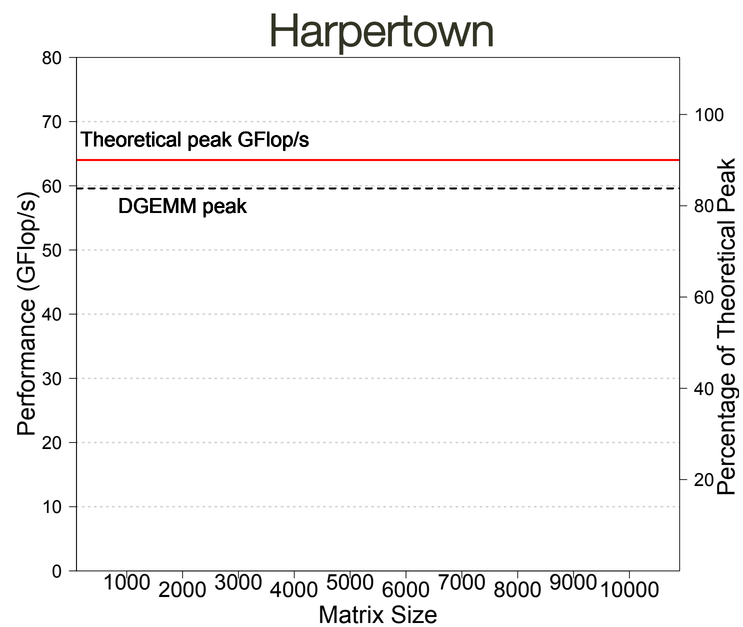
Intel 2-socket x 4-core Harpertown @ 2 GHz + Intel MKL 10.1 for sequential components

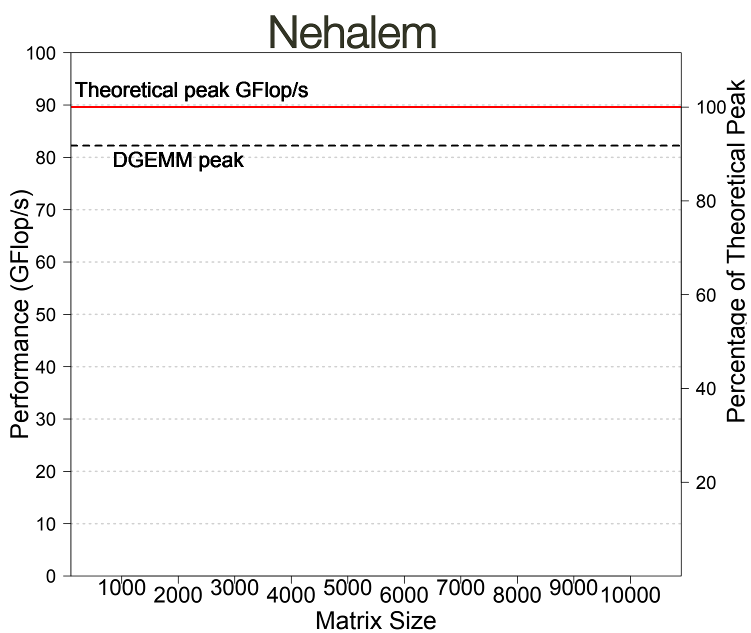


Cilk++-based Cholesky timeline (n=1000)

Intel 2-socket x 4-core Harpertown @ 2 GHz

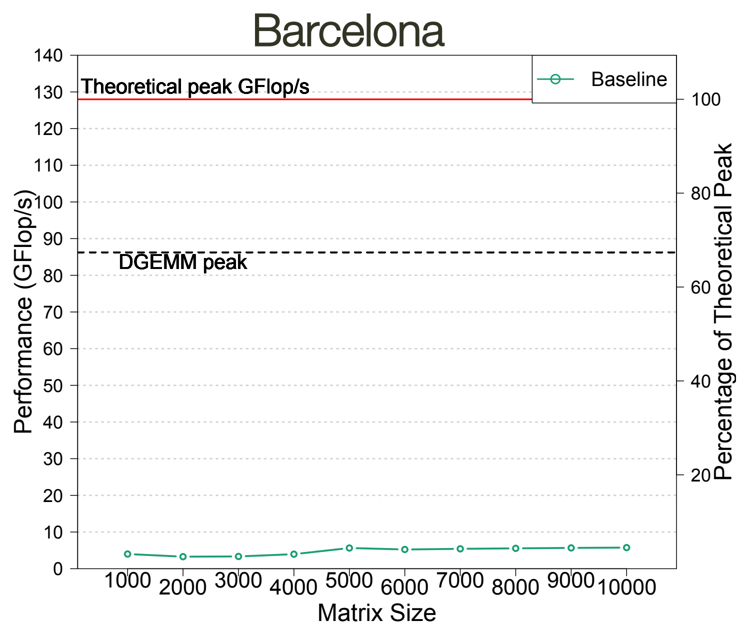


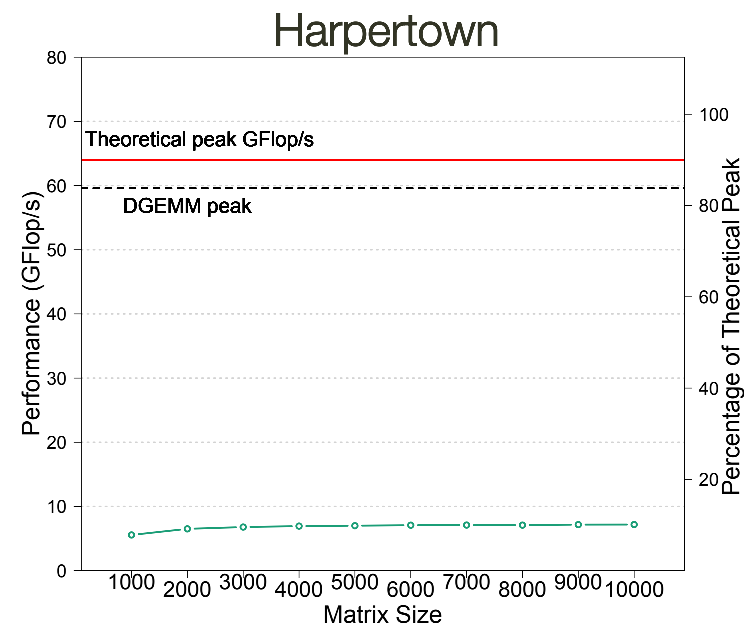


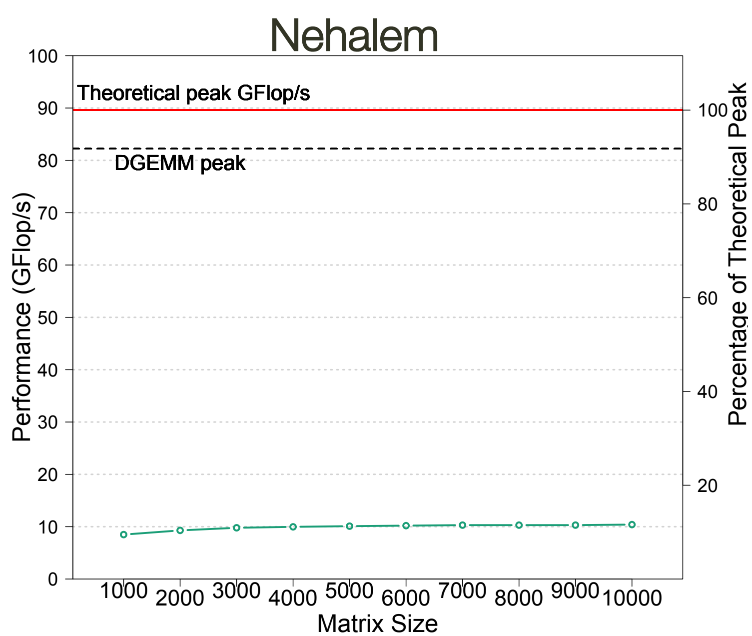


Cholesky Performance

- Theoretical peak: Double precision peak performance
- DGEMM peak: Performance of dense matrix multiplication

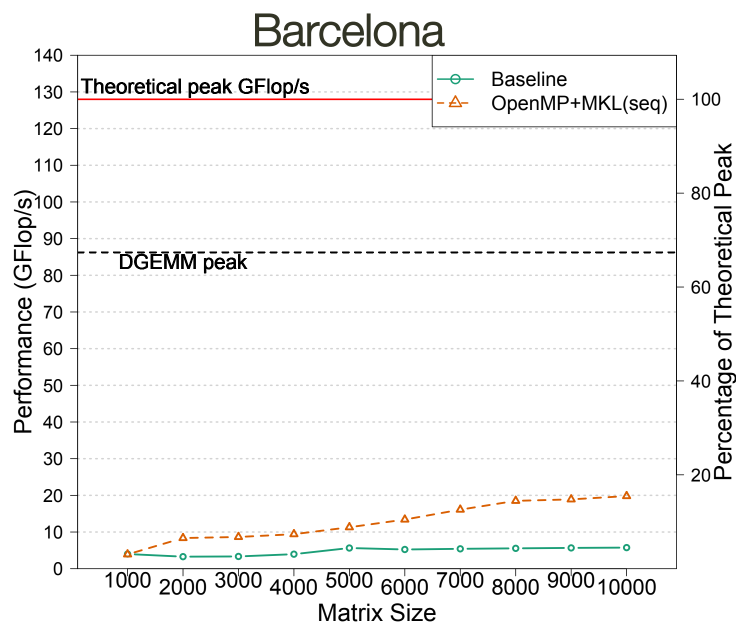


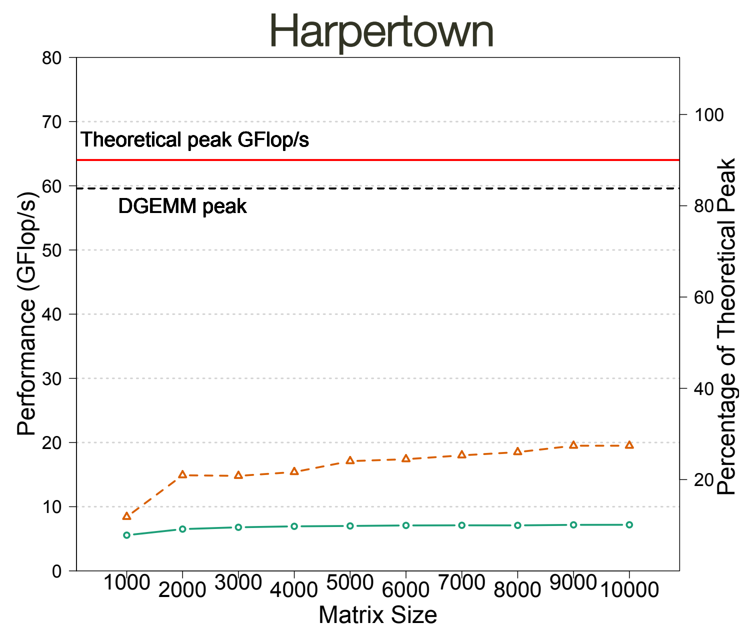


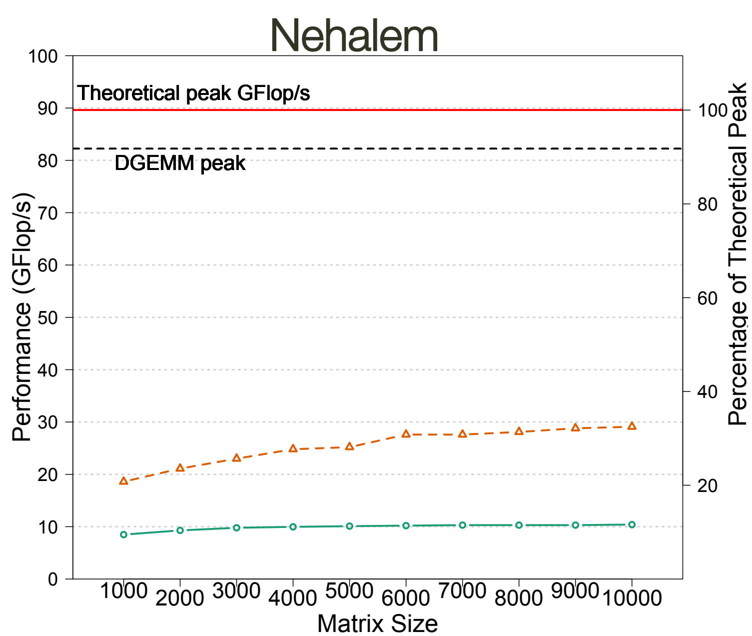


Cholesky Performance

- Baseline: Tuned sequential Math Kernel Library (MKL)

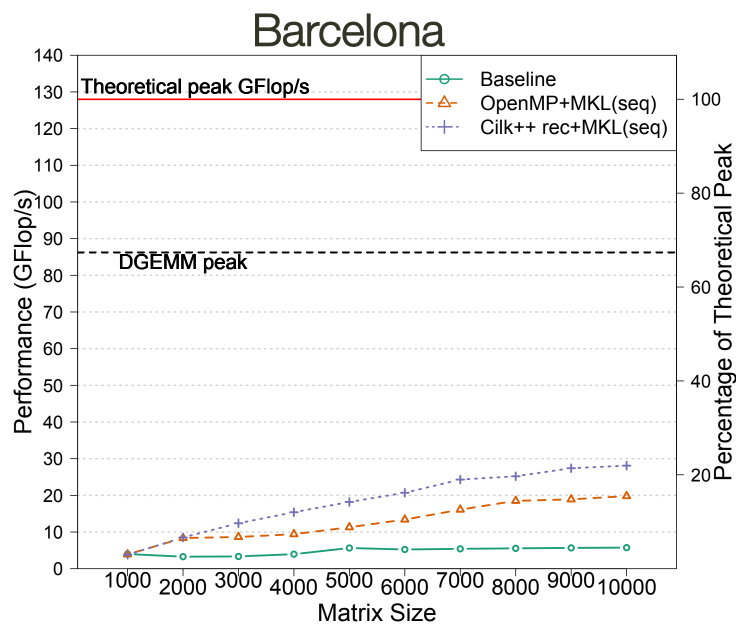


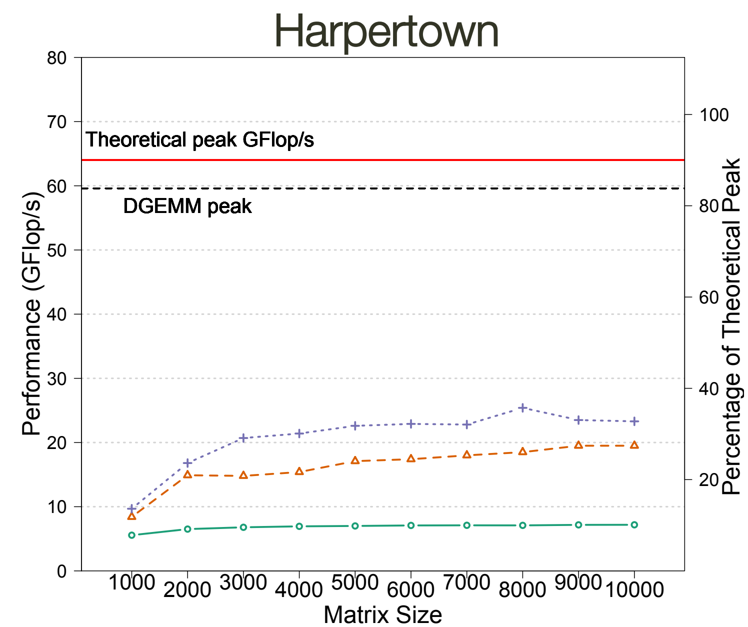


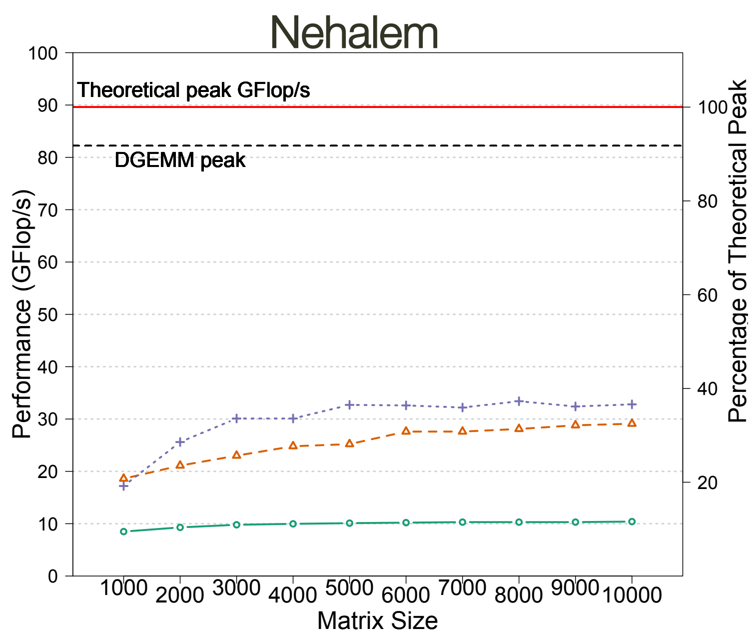


Cholesky Performance

- Sequential MKL for each block operation
- Block size chosen by exhaustive search

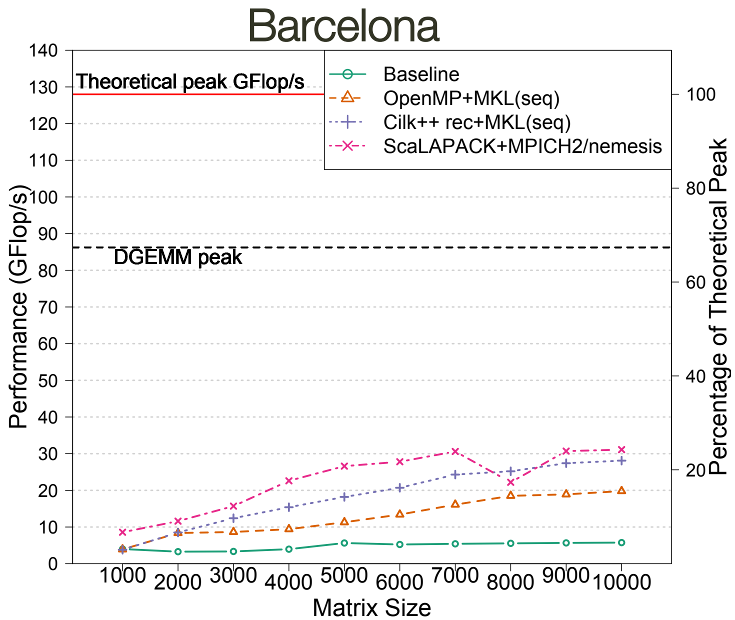


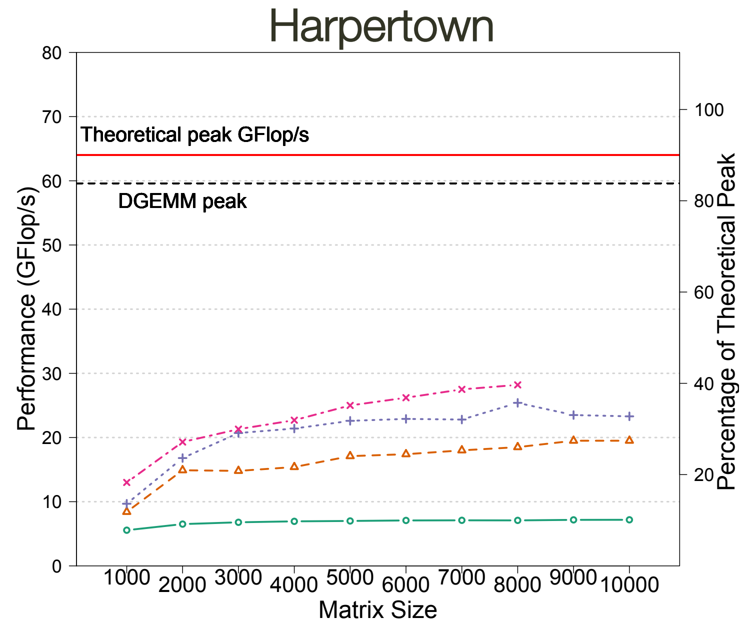


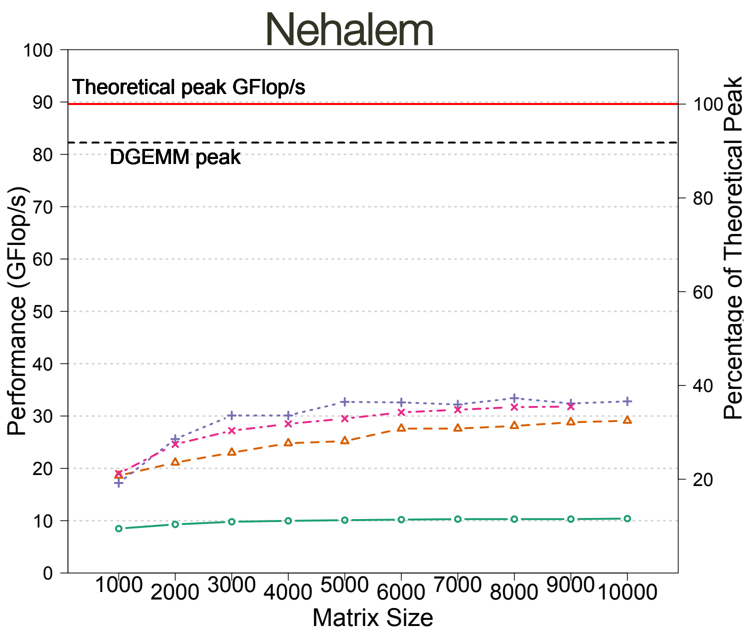


Cholesky Performance

- All steps of Cholesky implemented recursively

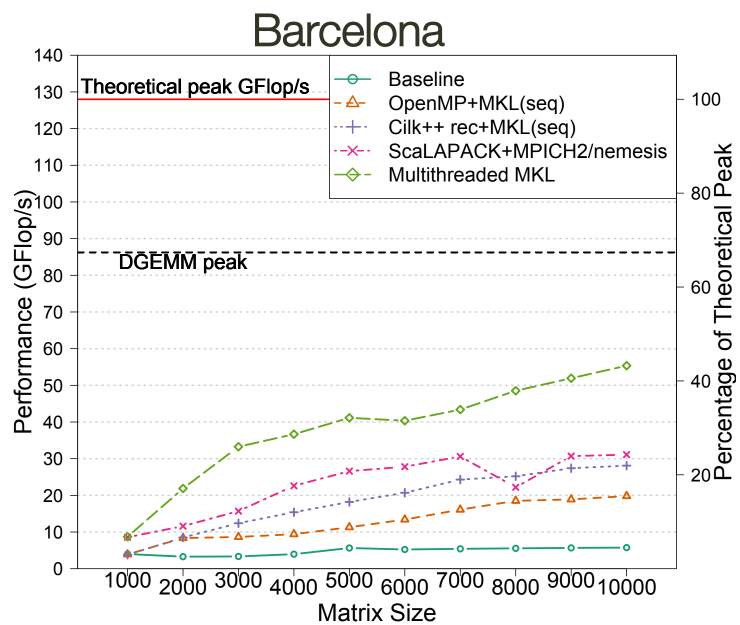


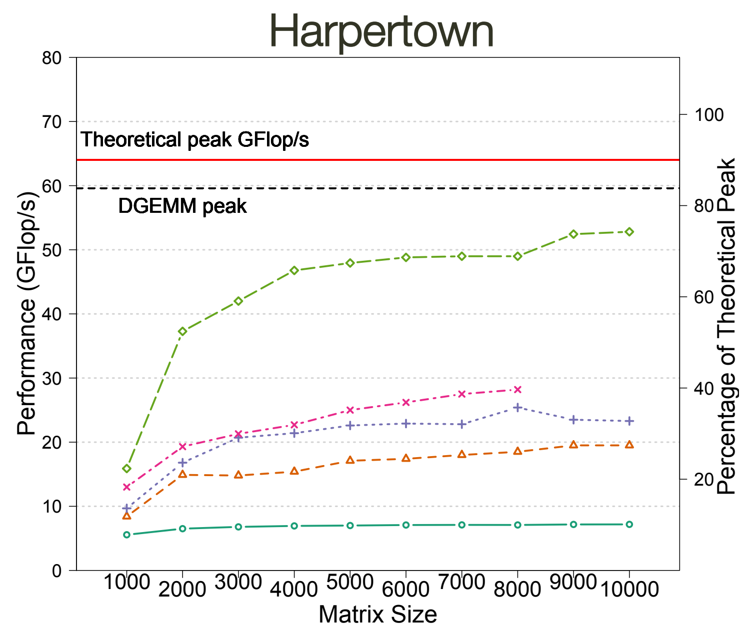


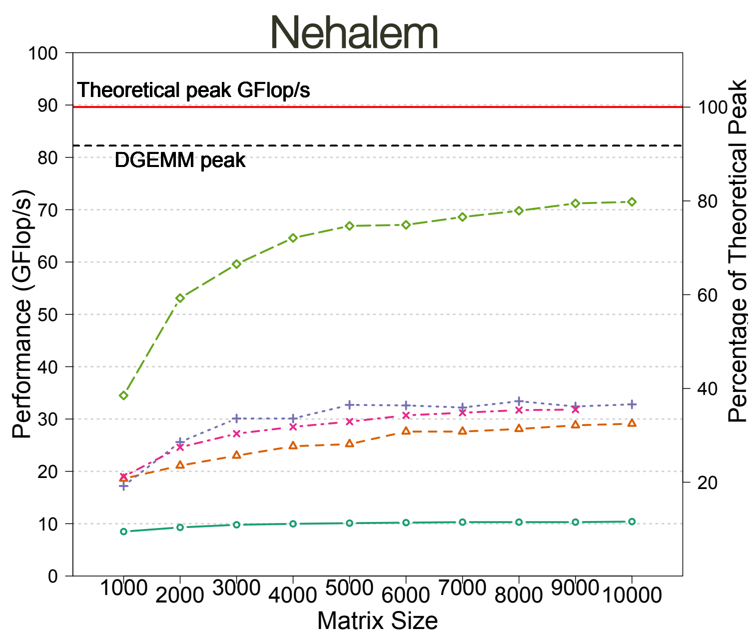


Cholesky Performance

- SCALAPACK 1.8.0 with MPI tuned for shared memory
- MPICH2 1.0.8 compiled with the Nemesis device

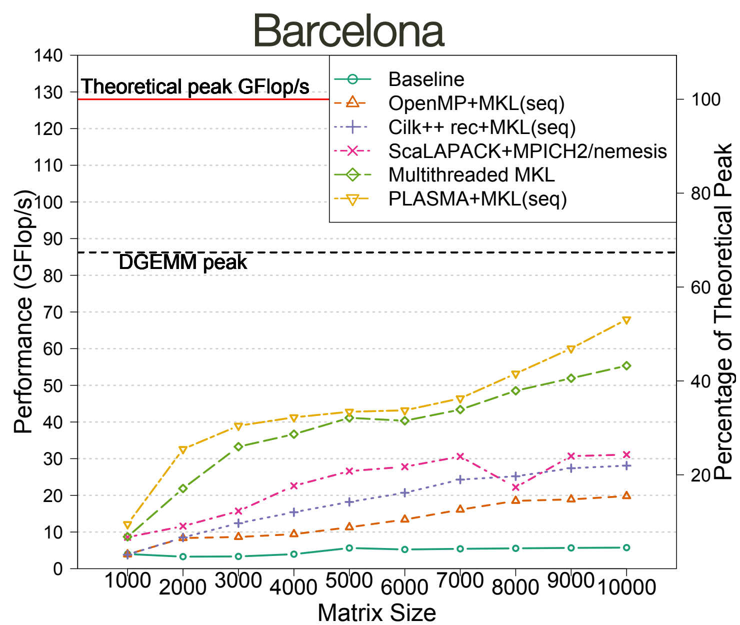


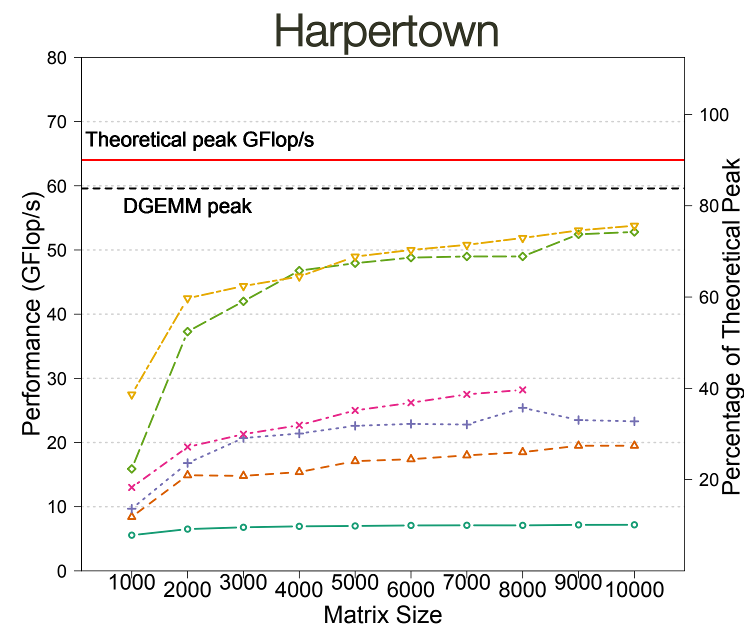


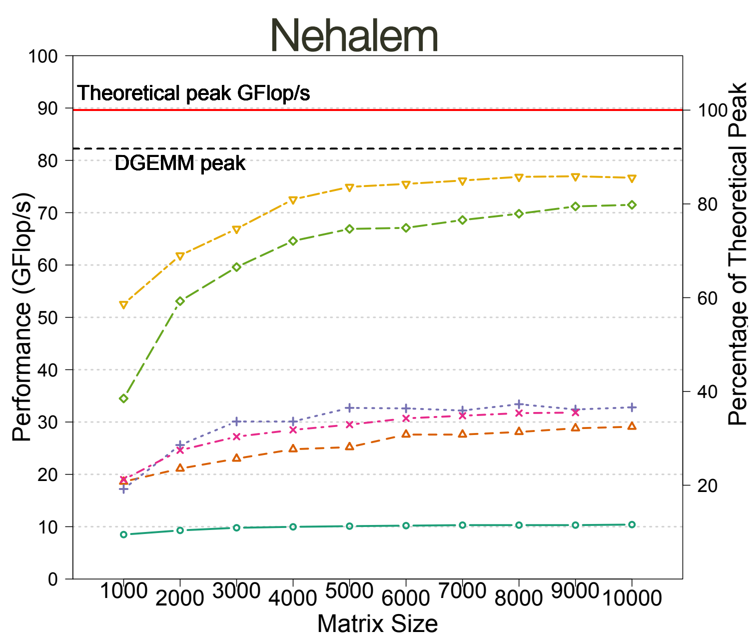


Cholesky Performance

- MKL implementation of LAPACK routine "dpotrf"
- Matrix stored in column major layout

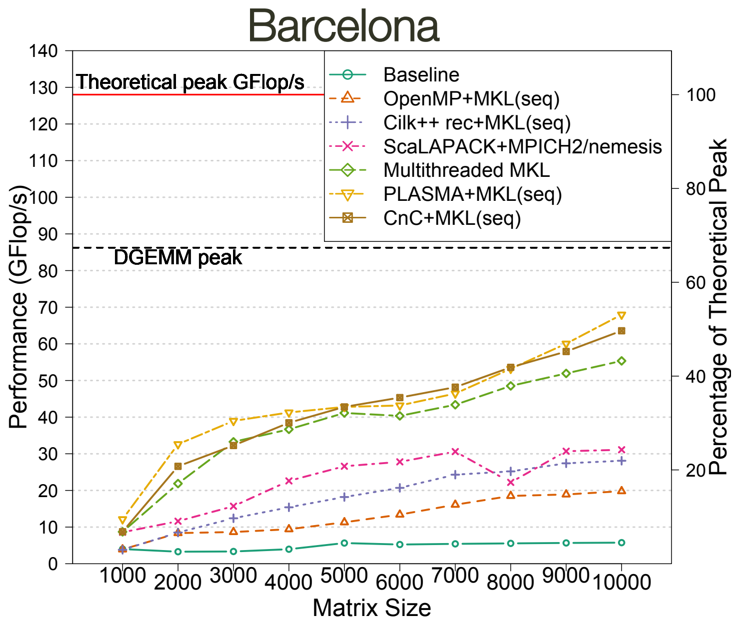


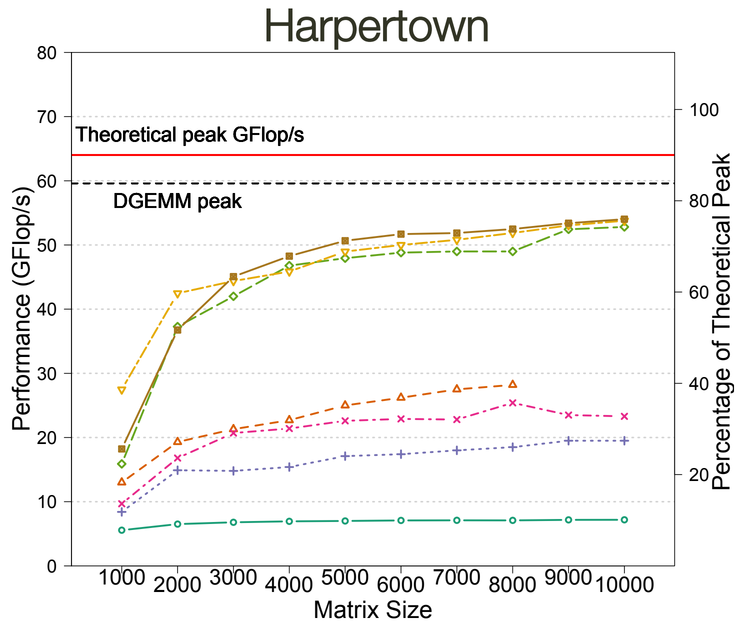


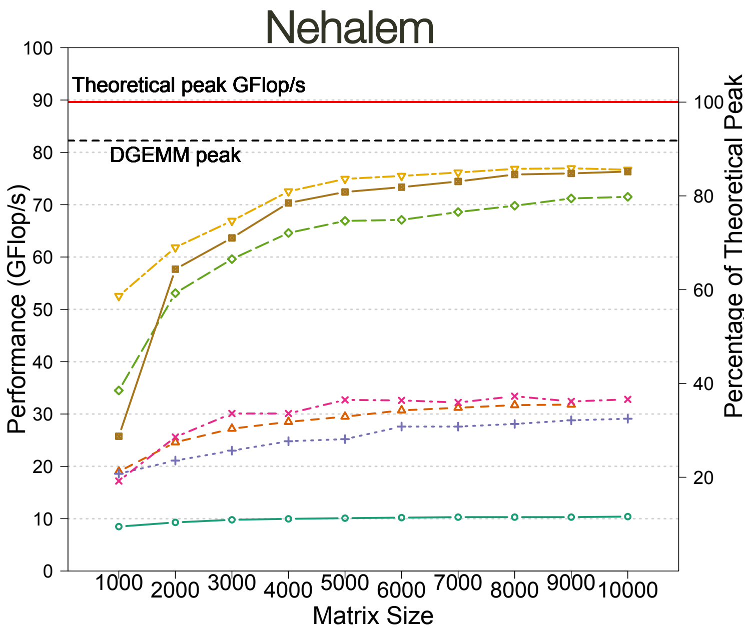


Cholesky Performance

- PLASMA 2.0
- Blocked size tuned separately for each input matrix

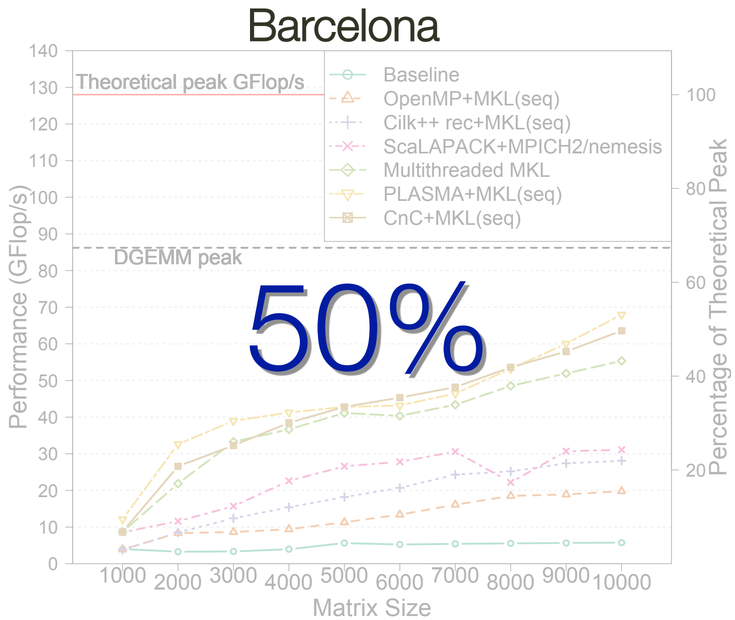


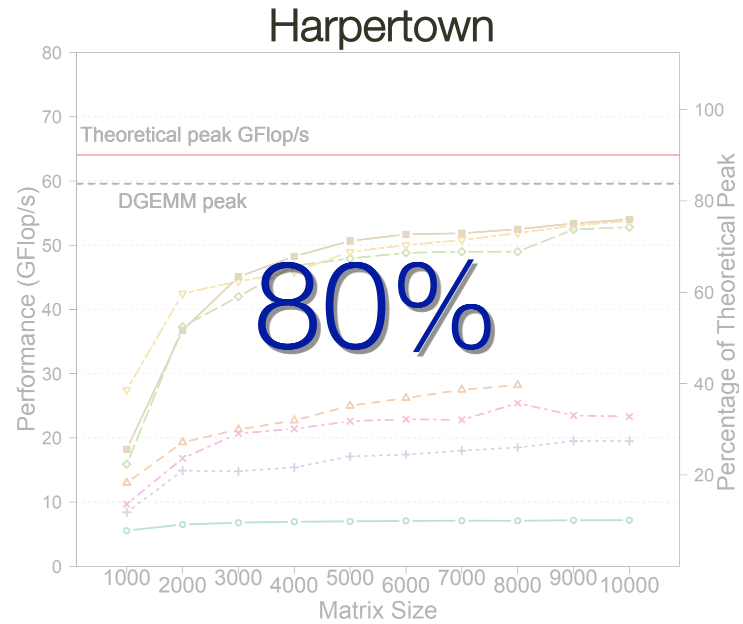


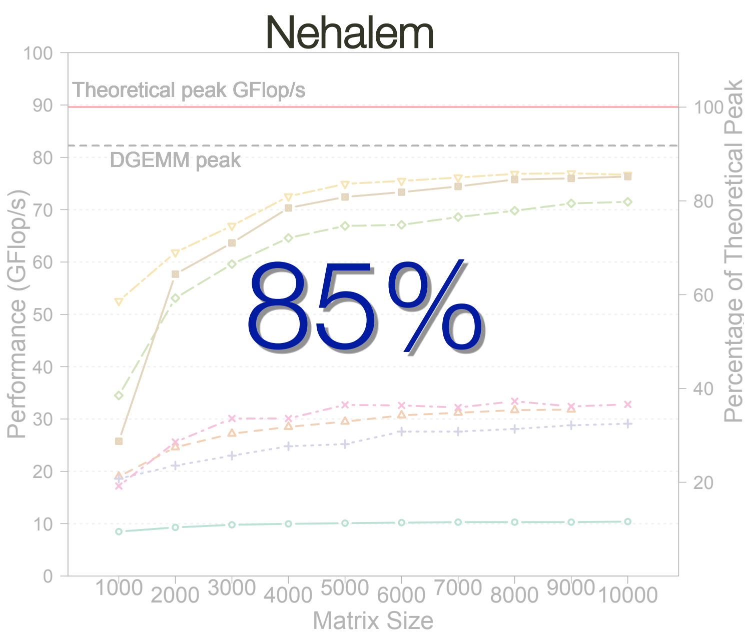


Cholesky Performance

- Sequential MKL for serial step code
- Matrix stored in blocked data layout

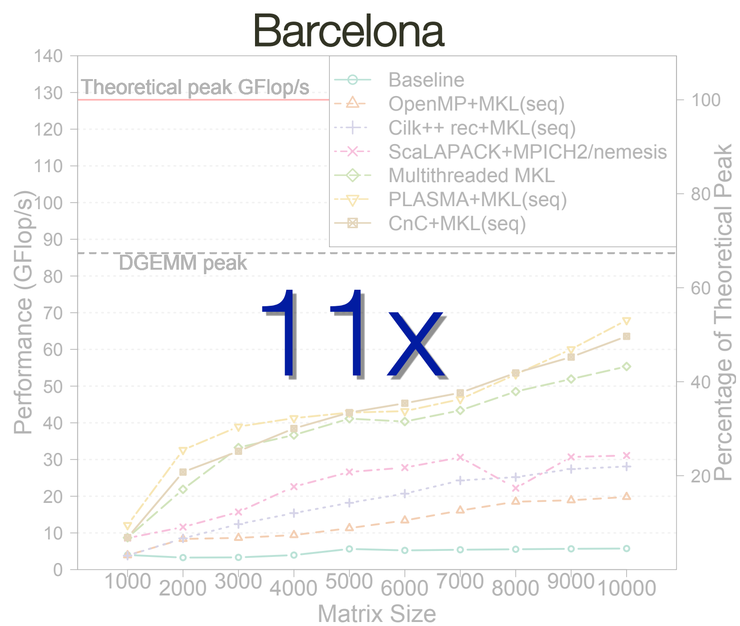


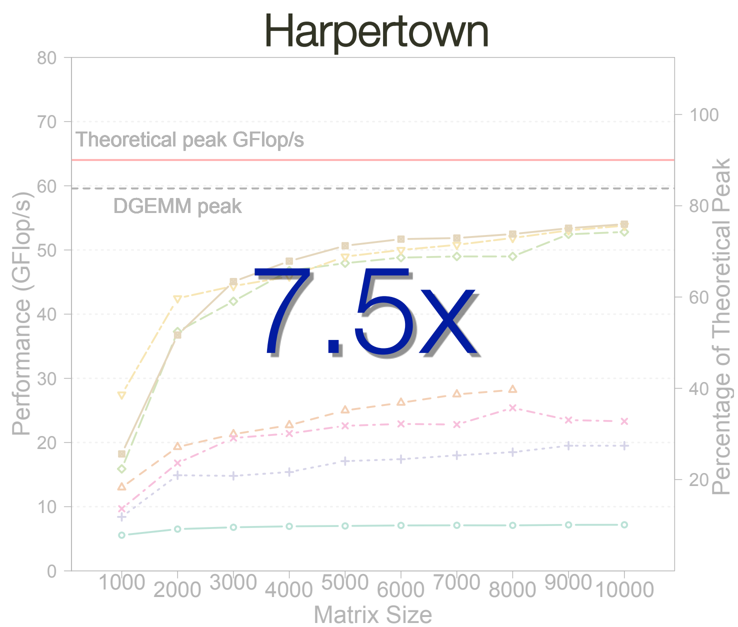


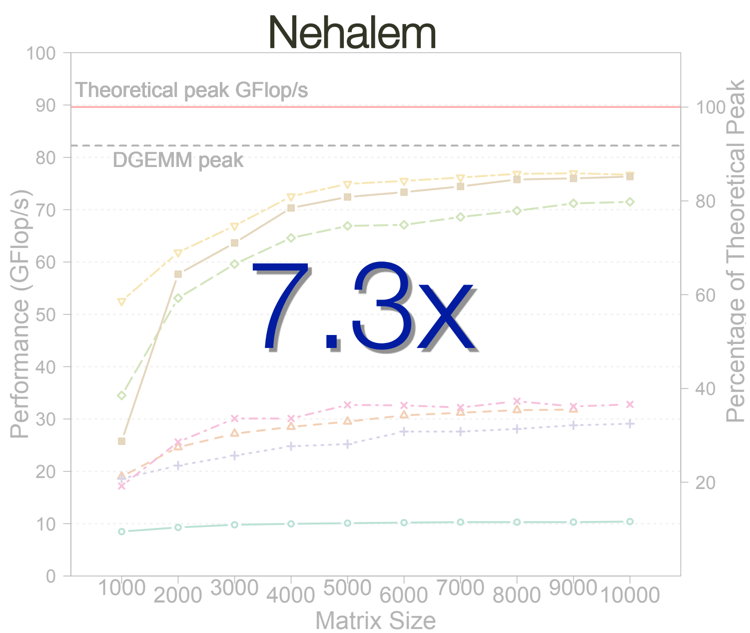


Achieved theoretical peak

- Theoretical peak performance achieved for matrix dimension, n = 10000





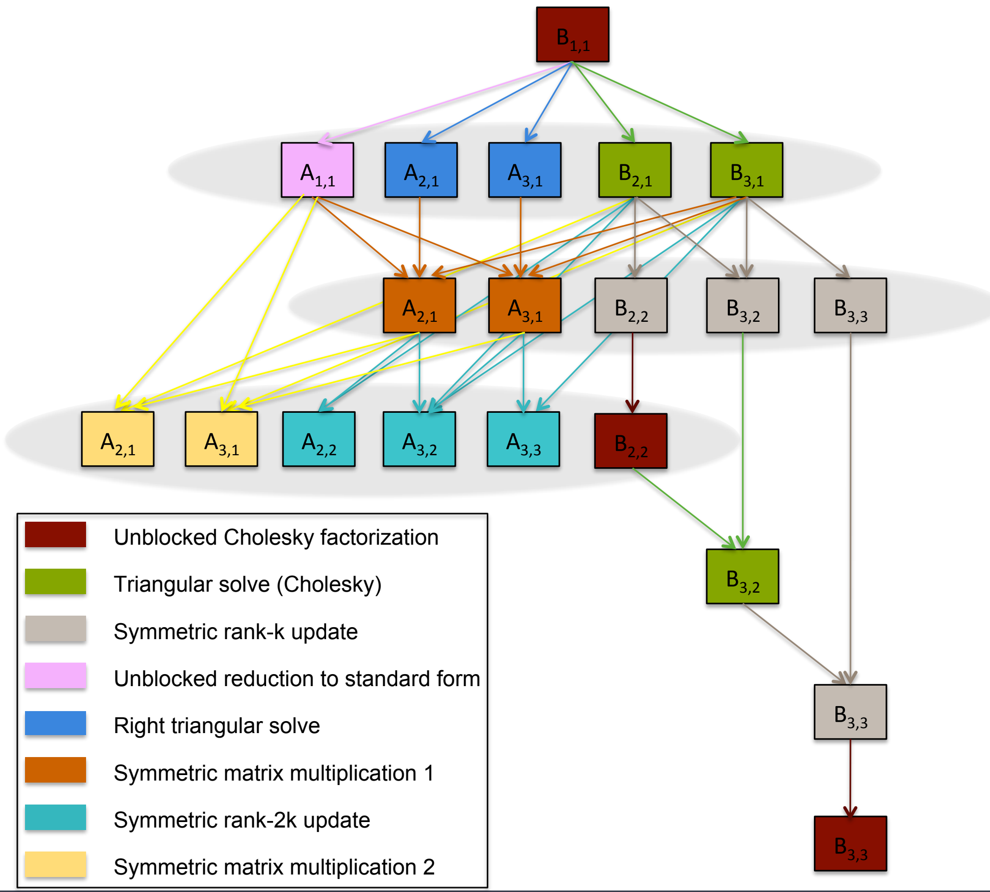


Speedup compared to baseline

Symmetric Eigensolver

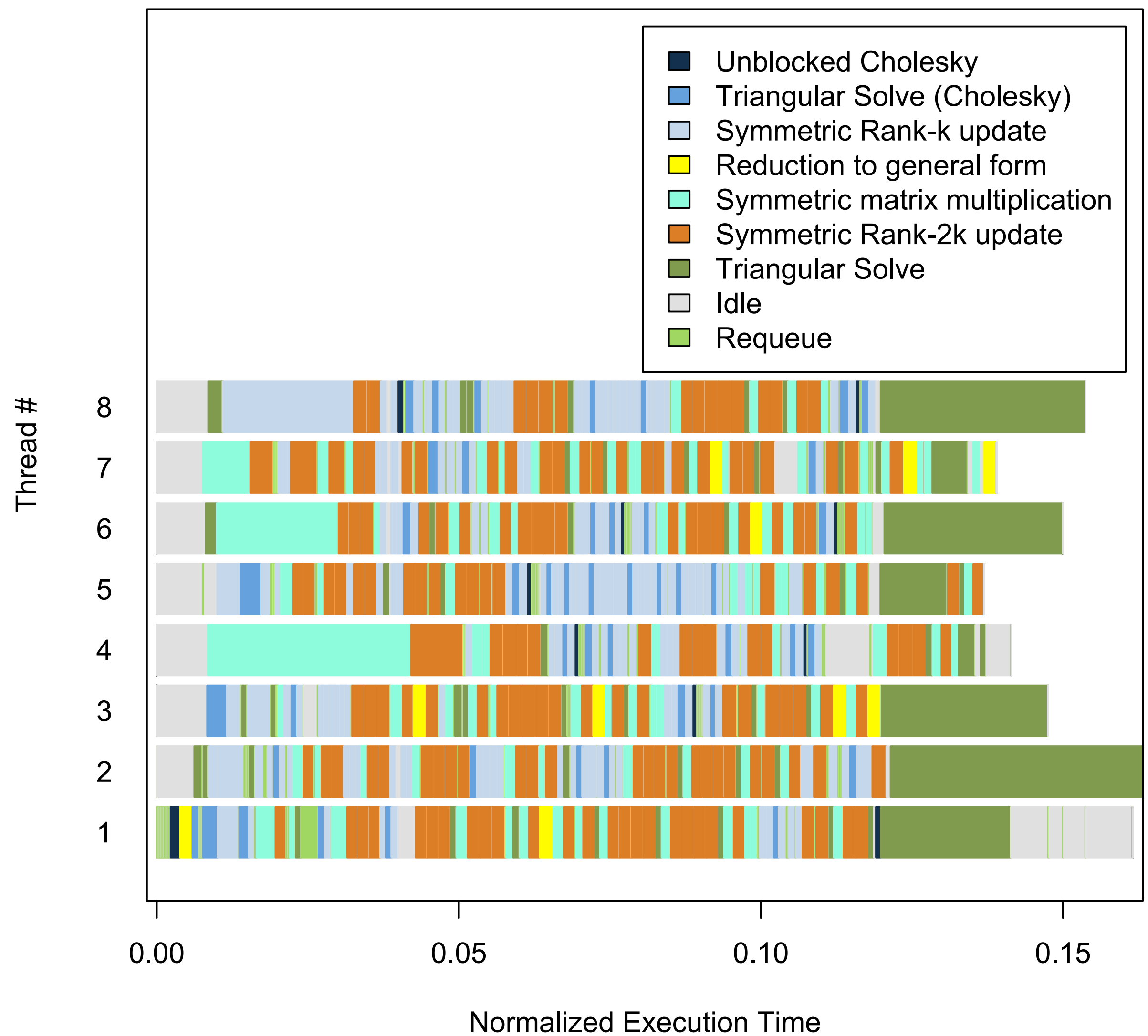
Dense symmetric generalized eigensolver

- Input
  - Symmetric matrix A
  - Symmetric positive definite matrix B
- We wish to compute λ and z such that $Az = \lambda Bz$
- "Straightforward" translation of LAPACK's `_sygvx`
  - Pieces: Cholesky / reduction to standard form; tridiag reduction, computing eigenvalues of the tridiag matrix
  - Only partly "asynchronous," but useful proof-of-concept
  - Performance limited by tridiagonal reduction step (BLAS-2)



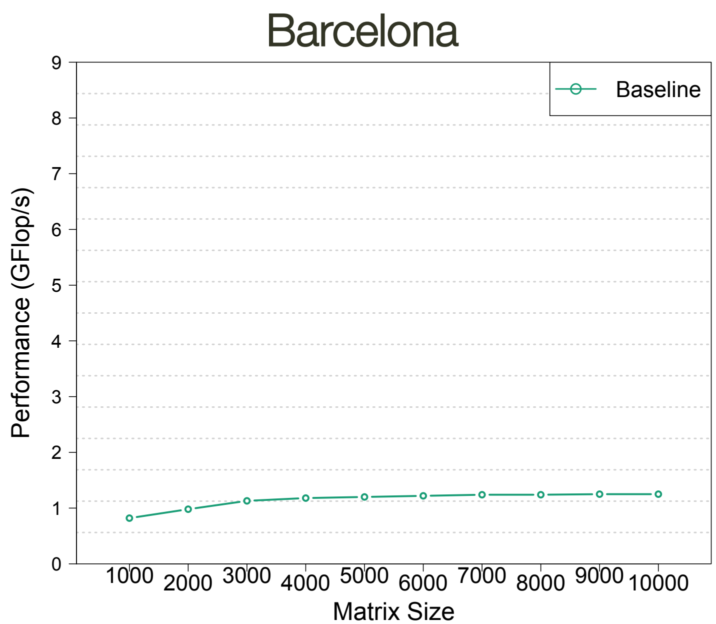
Partial DAG of eigensolver

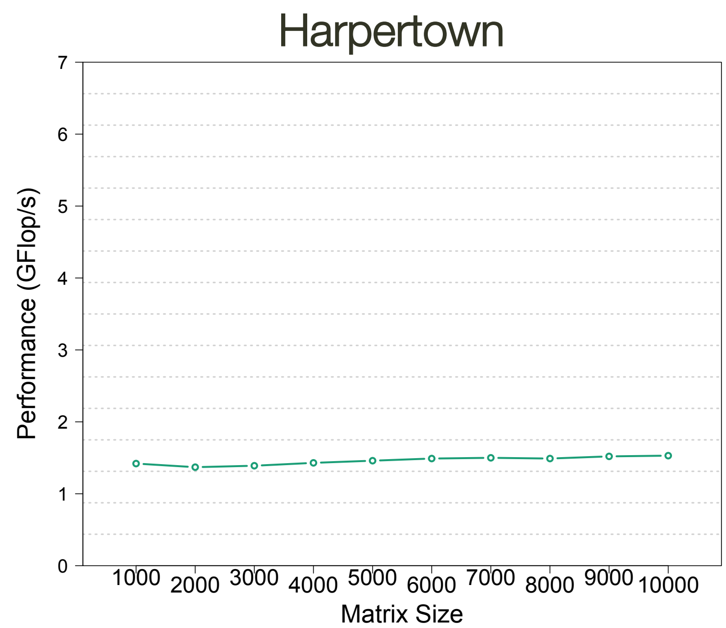
Performance analysis of Symmetric Eigensolver

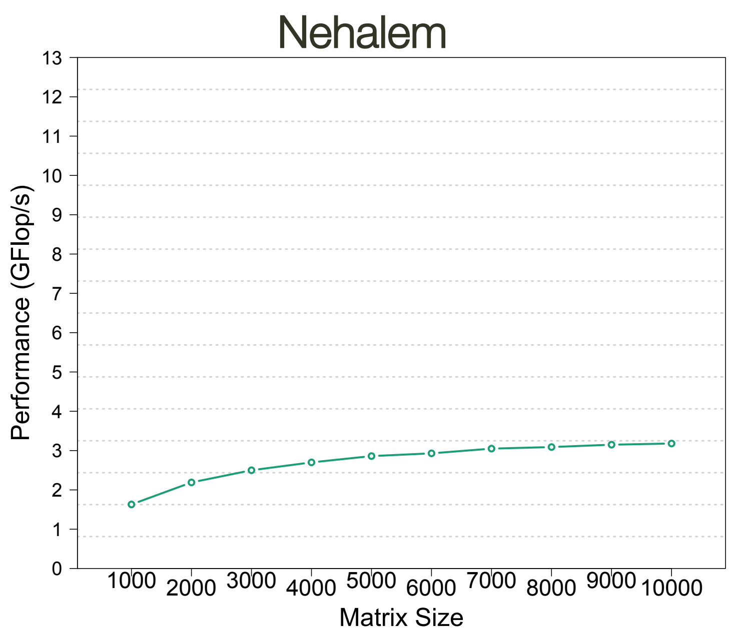


CnC-based Eigensolver timeline (n=1000):

Intel 2-socket x 4-core Harpertown @ 2 GHz + Intel MKL 10.1 for sequential components

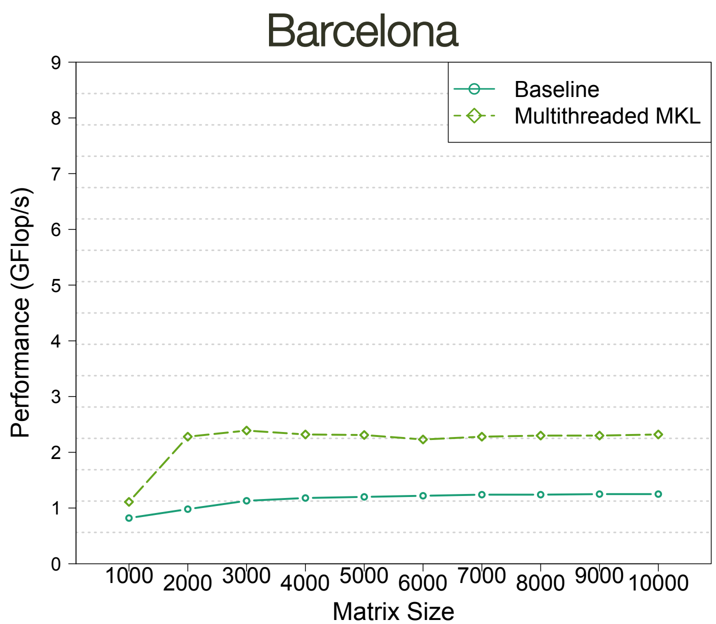


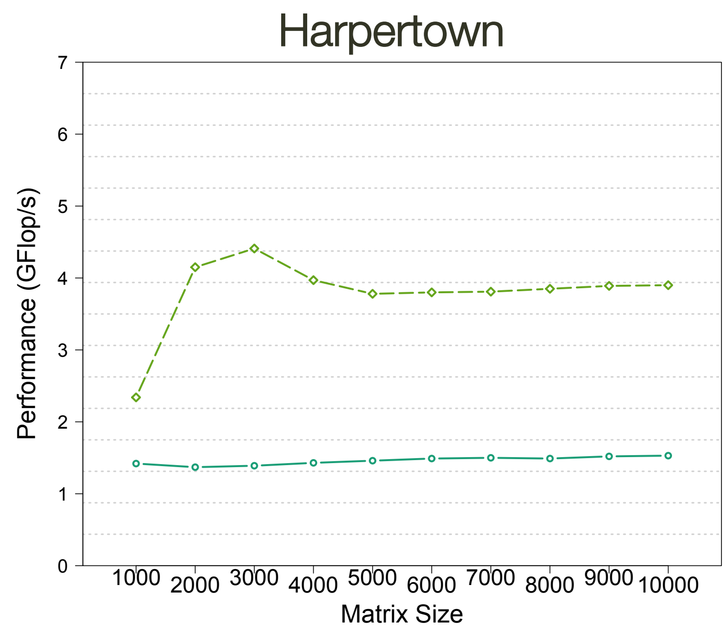


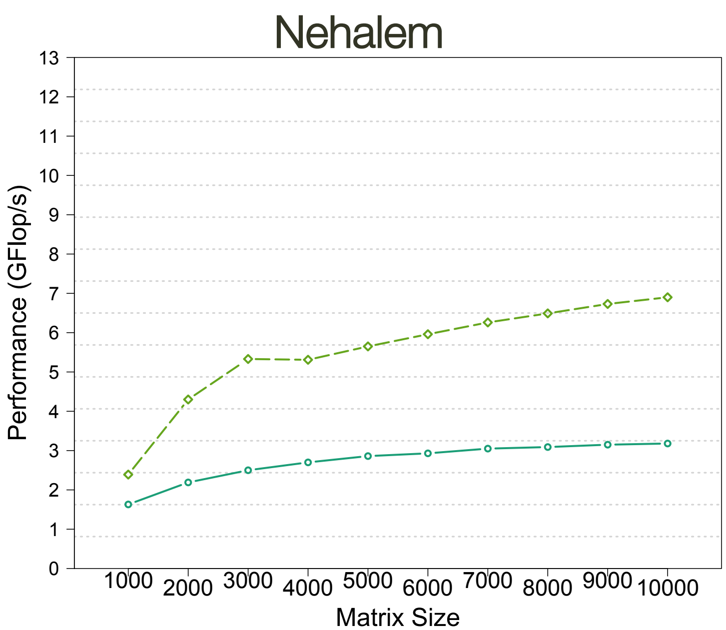


Eigensolver Performance

- Baseline is tuned sequential MKL

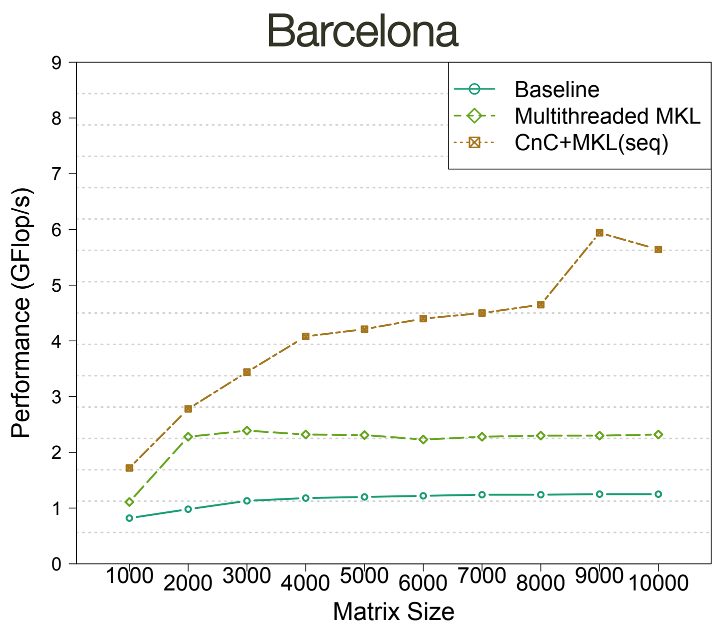


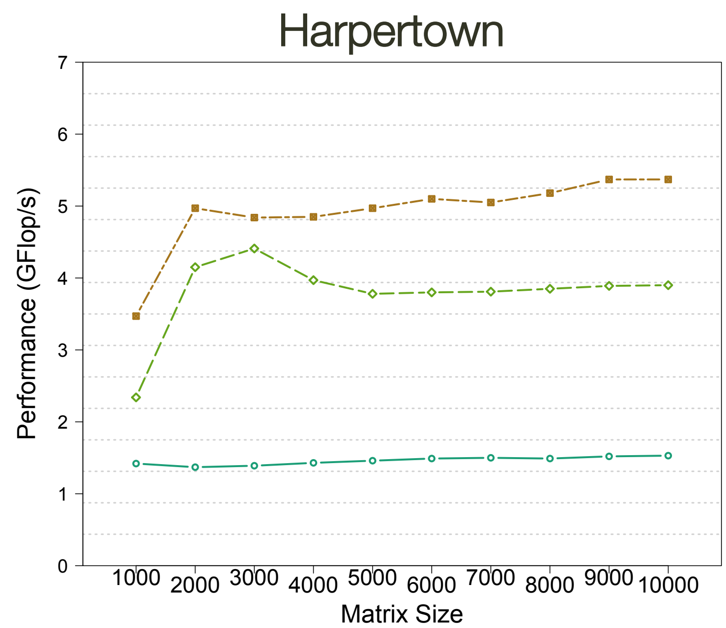


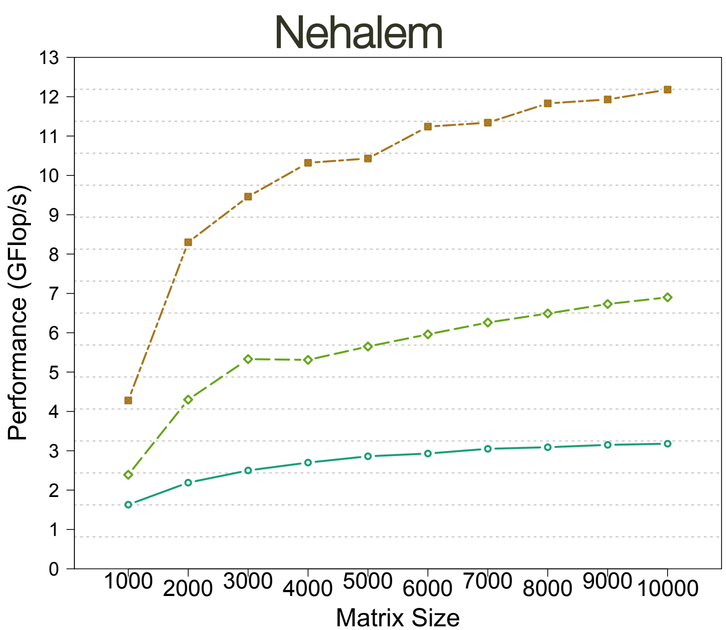


Eigensolver Performance

- MKL implementation of LAPACK routine "dsygvx"
- MKL implementation does not scale beyond 1 socket on Nehalem/Barcelona

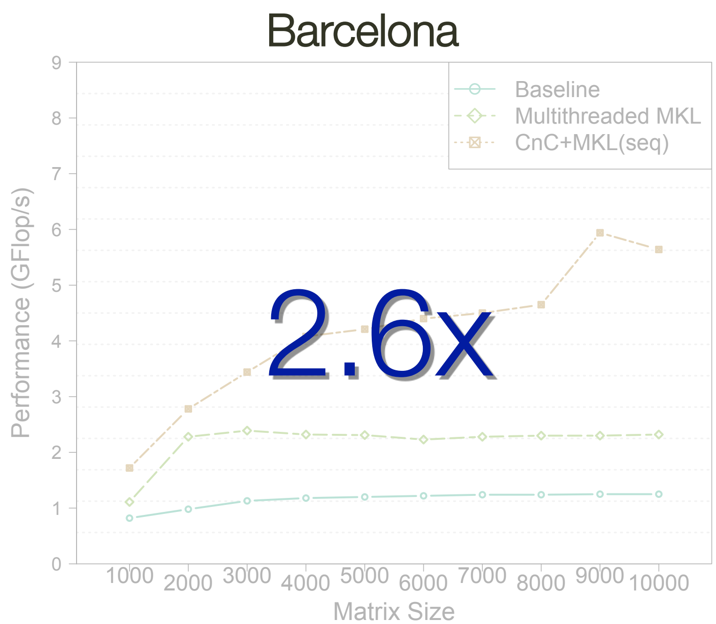


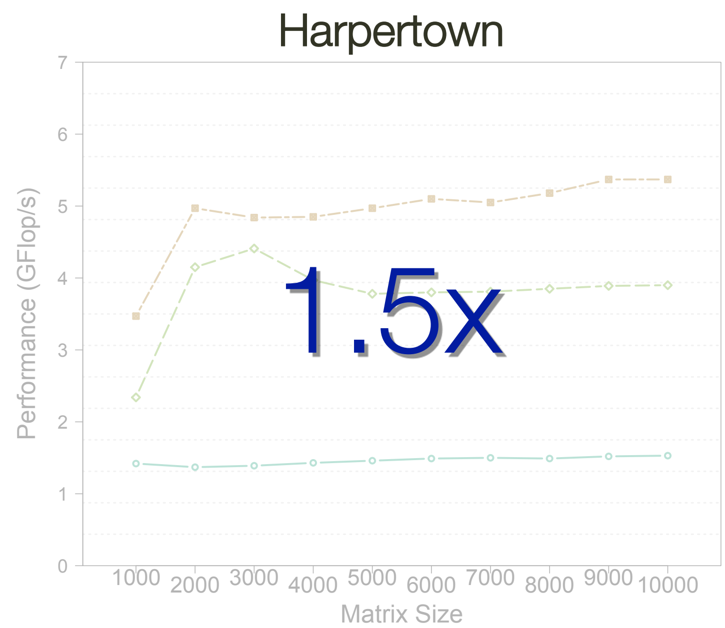


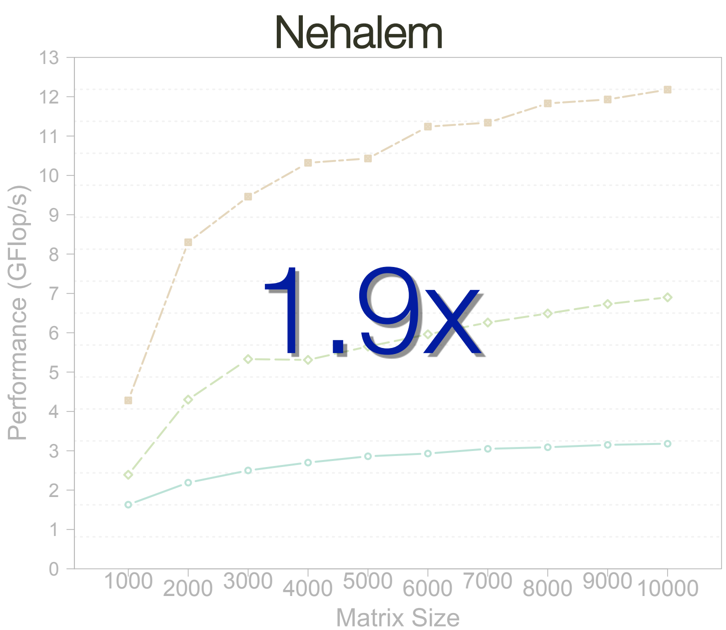


Eigensolver Performance

- Scales to the maximum number of cores
- Manually parallelized symmetric matrix vector multiply routine







Speedup over multithreaded MKL

- Smaller critical path than MKL
- Symmetric matrix vector multiply not parallelized by MKL
- NUMA effects on Barcelona/Nehalem

Summary

Limitations

- Requeue events (current runtime implementation has a “solution”)
- No support for in-place algorithms (static-single assignment)
- Current run-time scheduling is limited to LIFO
- Cannot handle continuous (streaming) input
- Tag types: integers only
- Tools, *e.g.*, debugging

Summary

- PDLA in CnC
  - Complements existing approaches for expressing and scheduling asynchronous parallel computations.
  - We can match or exceed a highly tuned vendor library (*e.g., MKL*) and state-of-the-art domain-specific library (*e.g., PLASMA*) for Cholesky.
  - Achieve significant speedups (1.1-2.6x) on a complex eigensolver.
  - In short, we can achieve high performance in CnC but there is scope for improvement.

Backup slides

dsygvx algorithm

- Cholesky factorization, $B \longrightarrow L\,L^T$
- Reduction of symmetric generalized eigenvalue problem to standard form
  - $(L^{-1} A\, L^{-T})z = \lambda z$

dsygvx algorithm

- Cholesky factorization, $B \longrightarrow L\,L^T$
- Reduction of symmetric generalized eigenvalue problem to standard form
  - $(L^{-1} A\, L^{-T})z = \lambda z$
- Now the problem has been transformed from $Az = \lambda Bz$ to $Cz = \lambda z$

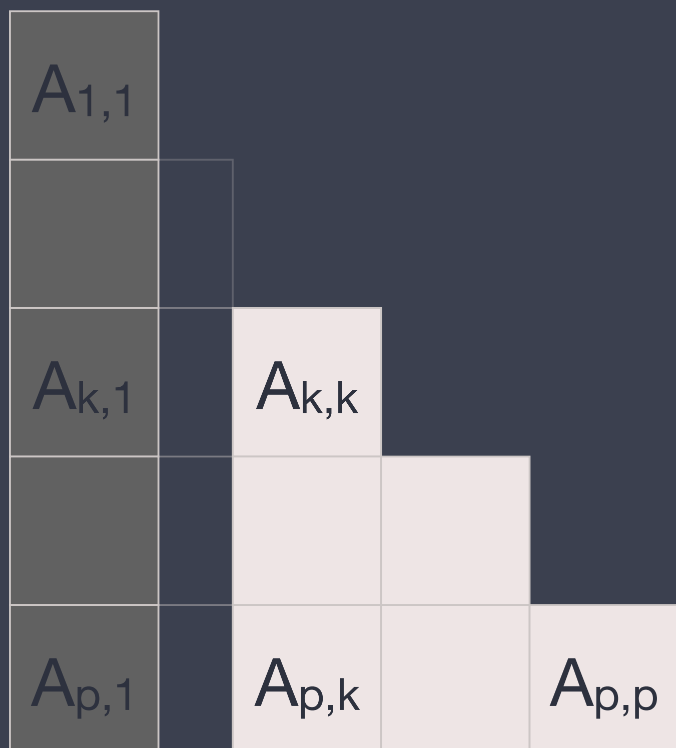
dsygvx algorithm

- Cholesky factorization, $B \longrightarrow L\,L^T$
- Reduction of symmetric generalized eigenvalue problem to standard form
  - $(L^{-1} A\, L^{-T})z = \lambda z$
- Now the problem has been transformed from $Az = \lambda Bz$ to $Cz = \lambda z$
- Reduction of the symmetric matrix to symmetric tridiagonal form
  - Orthogonal similarity transformation
  - $T = Q^T C\, Q$

dsygvx algorithm

- Cholesky factorization, $B \longrightarrow L\, L^T$
- Reduction of symmetric generalized eigenvalue problem to standard form
  - $(L^{-1} A\, L^{-T})z = \lambda z$
- Now the problem has been transformed from $Az = \lambda Bz$ to $Cz = \lambda z$
- Reduction of the symmetric matrix to symmetric tridiagonal form
  - Orthogonal similarity transformation
  - $T = Q^T C\, Q$
- Find the eigenvalues of T using a modified QR method

Symmetric eigensolver



Iteration *k*: // Over diagonal tiles

SeqCholesky ($L_{k,k} \leftarrow B_{k,k}$)

Trisolve ($L_{k+1:p,k} \leftarrow B_{k+1:p,k}, L_{k,k}$)

Update ($B_{k+1:p,k+1:p} \leftarrow L_{k+1:p,k}, B_{k+1:p,k+1:p}$)

SeqReduction ($M_{k,k} \leftarrow A_{k,k}, L_{k,k}$)

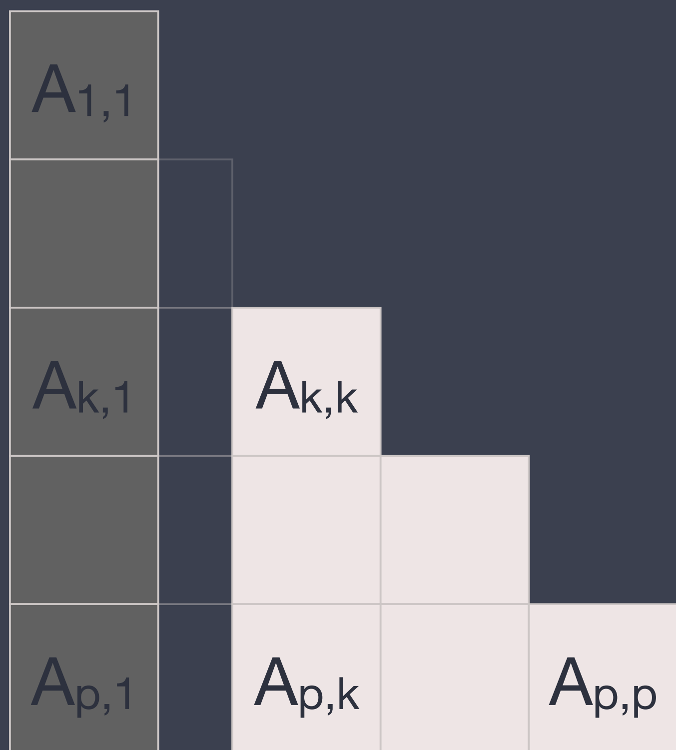
RightTriSolve ($A_{k+1:p,k} \leftarrow A_{k+1:p,k}, L_{k,k}$)

SymmMatmul ($A_{k+1:p,k} \leftarrow A_{k+1:p,k}, L_{k+1:p,k}, M_{k,k}$)

Update2 ($A_{k+1:p,k+1:p} \leftarrow L_{k+1:p,k}, A_{k+1:p,k}, A_{k+1:p,k+1:p}$)

SymmMatmul ($A_{k+1:p,k} \leftarrow A_{k+1:p,k}, L_{k+1:p,k}, M_{k,k}$)

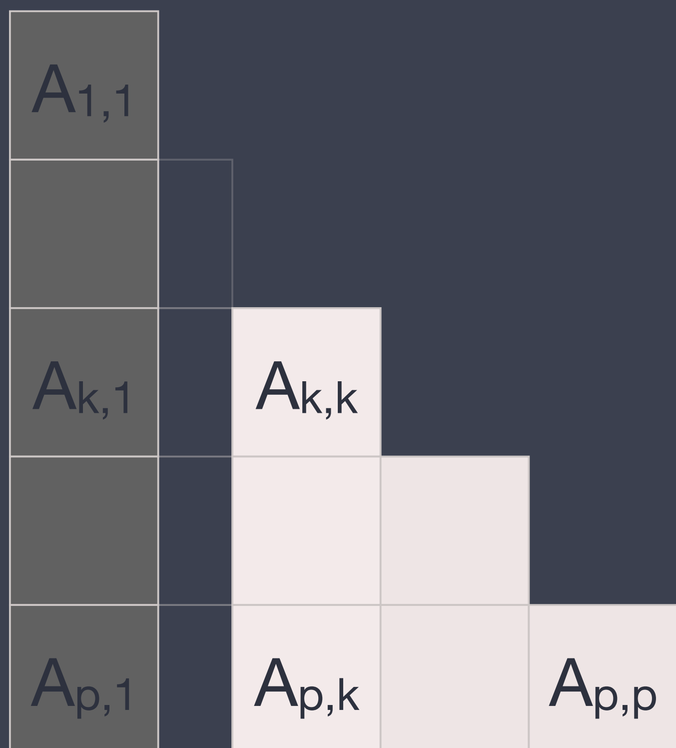
Symmetric eigensolver



Iteration *k*: // Over diagonal tiles

- **SeqCholesky ($L_{k,k} \leftarrow B_{k,k}$)**
- Trisolve ($L_{k+1:p,k} \leftarrow B_{k+1:p,k}, L_{k,k}$)
- Update ($B_{k+1:p,k+1:p} \leftarrow L_{k+1:p,k}, B_{k+1:p,k+1:p}$)
- SeqReduction ($M_{k,k} \leftarrow A_{k,k}, L_{k,k}$)
- RightTriSolve ($A_{k+1:p,k} \leftarrow A_{k+1:p,k}, L_{k,k}$)
- SymmMatmul ($A_{k+1:p,k} \leftarrow A_{k+1:p,k}, L_{k+1:p,k}, M_{k,k}$)
- Update2 ($A_{k+1:p,k+1:p} \leftarrow L_{k+1:p,k}, A_{k+1:p,k}, A_{k+1:p,k+1:p}$)
- SymmMatmul ($A_{k+1:p,k} \leftarrow A_{k+1:p,k}, L_{k+1:p,k}, M_{k,k}$)

Symmetric eigensolver



Iteration *k*: // Over diagonal tiles

SeqCholesky ($L_{k,k} \leftarrow B_{k,k}$)

Trisolve ($L_{k+1:p,k} \leftarrow B_{k+1:p,k}, L_{k,k}$)

Update ($B_{k+1:p,k+1:p} \leftarrow L_{k+1:p,k}, B_{k+1:p,k+1:p}$)

SeqReduction ($M_{k,k} \leftarrow A_{k,k}, L_{k,k}$)

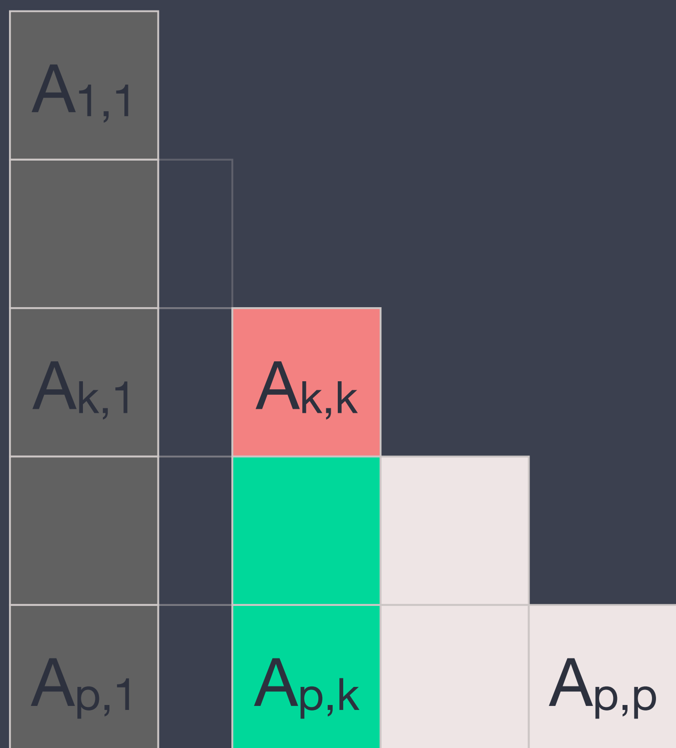
RightTriSolve ($A_{k+1:p,k} \leftarrow A_{k+1:p,k}, L_{k,k}$)

SymmMatmul ($A_{k+1:p,k} \leftarrow A_{k+1:p,k}, L_{k+1:p,k}, M_{k,k}$)

Update2 ($A_{k+1:p,k+1:p} \leftarrow L_{k+1:p,k}, A_{k+1:p,k}, A_{k+1:p,k+1:p}$)

SymmMatmul ($A_{k+1:p,k} \leftarrow A_{k+1:p,k}, L_{k+1:p,k}, M_{k,k}$)

Symmetric eigensolver



Iteration *k*: // Over diagonal tiles

SeqCholesky ($L_{k,k} \leftarrow B_{k,k}$)

Trisolve ($L_{k+1:p,k} \leftarrow B_{k+1:p,k}, L_{k,k}$)

Update ($B_{k+1:p,k+1:p} \leftarrow L_{k+1:p,k}, B_{k+1:p,k+1:p}$)

SeqReduction ($M_{k,k} \leftarrow A_{k,k}, L_{k,k}$)

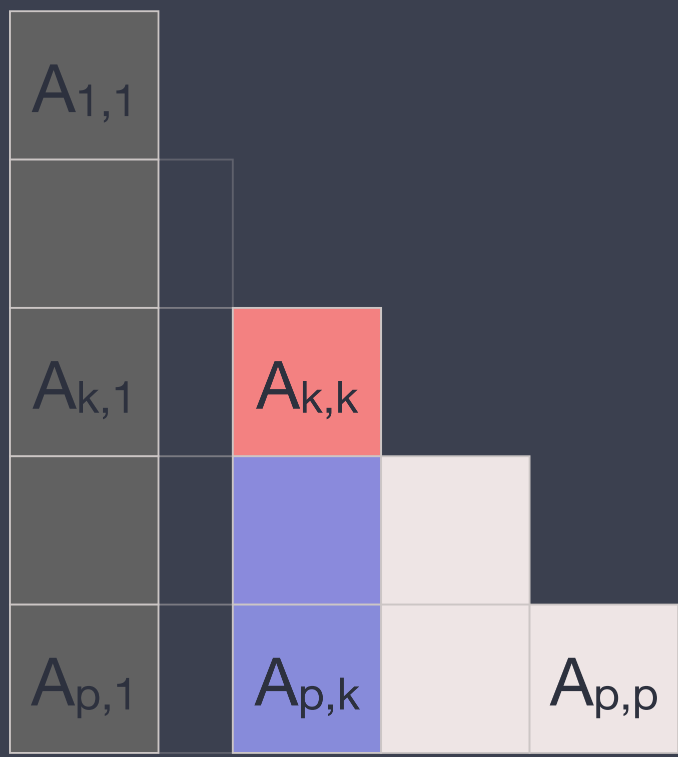
RightTriSolve ($A_{k+1:p,k} \leftarrow A_{k+1:p,k}, L_{k,k}$)

SymmMatmul ($A_{k+1:p,k} \leftarrow A_{k+1:p,k}, L_{k+1:p,k}, M_{k,k}$)

Update2 ($A_{k+1:p,k+1:p} \leftarrow L_{k+1:p,k}, A_{k+1:p,k}, A_{k+1:p,k+1:p}$)

SymmMatmul ($A_{k+1:p,k} \leftarrow A_{k+1:p,k}, L_{k+1:p,k}, M_{k,k}$)

Symmetric eigensolver



Iteration *k*: // Over diagonal tiles

SeqCholesky ($L_{k,k} \leftarrow B_{k,k}$)

Trisolve ($L_{k+1:p,k} \leftarrow B_{k+1:p,k}, L_{k,k}$)

Update ($B_{k+1:p,k+1:p} \leftarrow L_{k+1:p,k}, B_{k+1:p,k+1:p}$)

SeqReduction ($M_{k,k} \leftarrow A_{k,k}, L_{k,k}$)

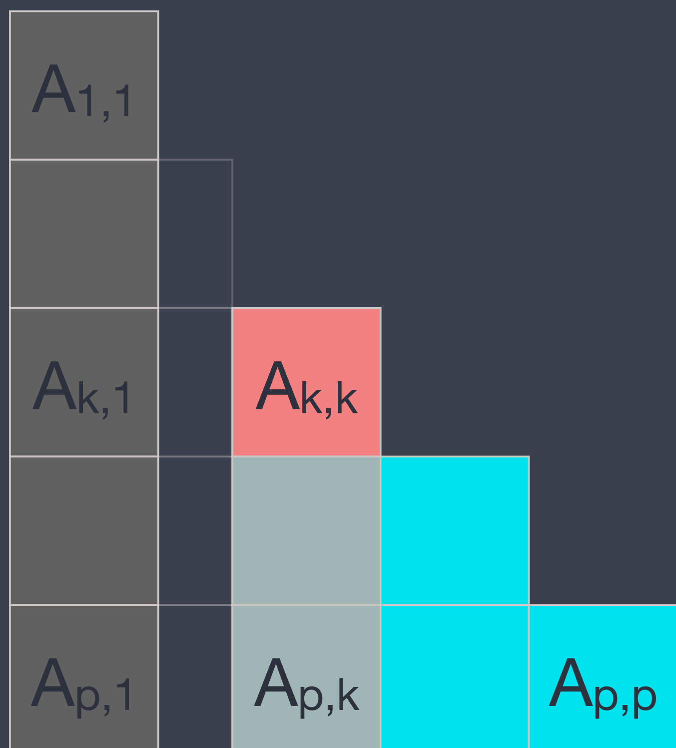
RightTriSolve ($A_{k+1:p,k} \leftarrow A_{k+1:p,k}, L_{k,k}$)

SymmMatmul ($A_{k+1:p,k} \leftarrow A_{k+1:p,k}, L_{k+1:p,k}, M_{k,k}$)

Update2 ($A_{k+1:p,k+1:p} \leftarrow L_{k+1:p,k}, A_{k+1:p,k}, A_{k+1:p,k+1:p}$)

SymmMatmul ($A_{k+1:p,k} \leftarrow A_{k+1:p,k}, L_{k+1:p,k}, M_{k,k}$)

Symmetric eigensolver



Iteration k: // Over diagonal tiles

SeqCholesky ($L_{k,k} \leftarrow B_{k,k}$)

Trisolve ($L_{k+1:p,k} \leftarrow B_{k+1:p,k}, L_{k,k}$)

Update ($B_{k+1:p,k+1:p} \leftarrow L_{k+1:p,k}, B_{k+1:p,k+1:p}$)

SeqReduction ($M_{k,k} \leftarrow A_{k,k}, L_{k,k}$)

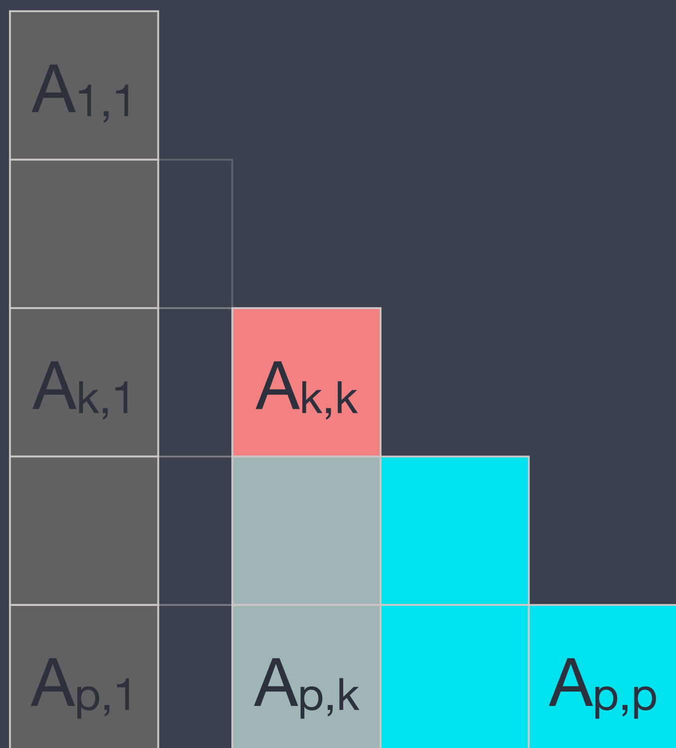
RightTriSolve ($A_{k+1:p,k} \leftarrow A_{k+1:p,k}, L_{k,k}$)

SymmMatmul ($A_{k+1:p,k} \leftarrow A_{k+1:p,k}, L_{k+1:p,k}, M_{k,k}$)

Update2 ($A_{k+1:p,k+1:p} \leftarrow L_{k+1:p,k}, A_{k+1:p,k}, A_{k+1:p,k+1:p}$)

SymmMatmul ($A_{k+1:p,k} \leftarrow A_{k+1:p,k}, L_{k+1:p,k}, M_{k,k}$)

Symmetric eigensolver



Iteration *k*: // Over diagonal tiles

SeqCholesky ($L_{k,k} \leftarrow B_{k,k}$)

Trisolve ($L_{k+1:p,k} \leftarrow B_{k+1:p,k}, L_{k,k}$)

Update ($B_{k+1:p,k+1:p} \leftarrow L_{k+1:p,k}, B_{k+1:p,k+1:p}$)

SeqReduction ($M_{k,k} \leftarrow A_{k,k}, L_{k,k}$)

RightTriSolve ($A_{k+1:p,k} \leftarrow A_{k+1:p,k}, L_{k,k}$)

SymmMatmul ($A_{k+1:p,k} \leftarrow A_{k+1:p,k}, L_{k+1:p,k}, M_{k,k}$)

Update2 ($A_{k+1:p,k+1:p} \leftarrow L_{k+1:p,k}, A_{k+1:p,k}, A_{k+1:p,k+1:p}$)

SymmMatmul ($A_{k+1:p,k} \leftarrow A_{k+1:p,k}, L_{k+1:p,k}, M_{k,k}$)

Tile Cholesky in CnC

SeqCholesky ($L_{k,k} \leftarrow A_{k,k}$)

Trisolve ($L_{k+1:p,k} \leftarrow A_{k+1:p,k}, L_{k,k}$)

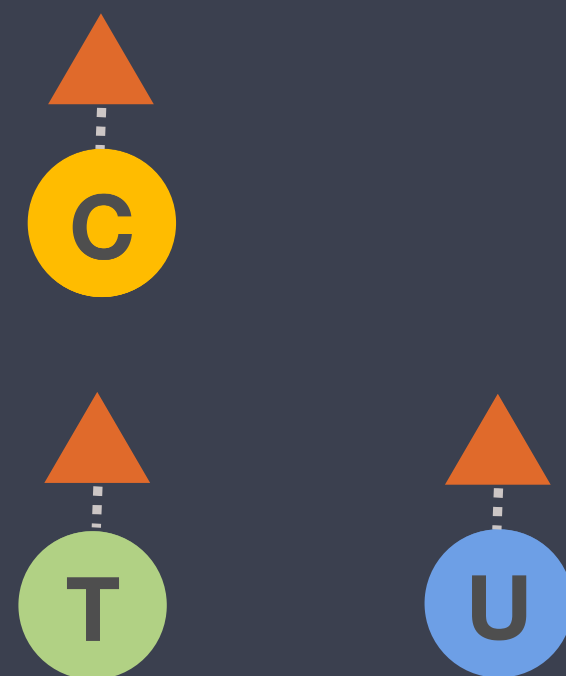
Update ($A_{k+1:p,k+1:p} \leftarrow L_{k+1:p,k}, A_{k+1:p,k+1:p}$)

Tile Cholesky in CnC

SeqCholesky ($L_{k,k} \leftarrow A_{k,k}$)

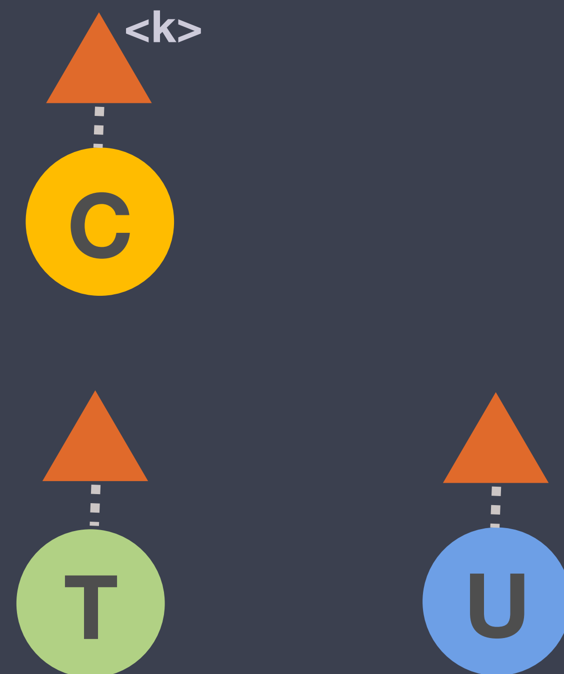
Trisolve ($L_{k+1:p,k} \leftarrow A_{k+1:p,k},\ L_{k,k}$)

Update ($A_{k+1:p,k+1:p} \leftarrow L_{k+1:p,k},\ A_{k+1:p,k+1:p}$)



Omitted: Items

Tile Cholesky in CnC



SeqCholesky ($L_{k,k} \leftarrow A_{k,k}$)

Trisolve ($L_{k+1:p,k} \leftarrow A_{k+1:p,k}, L_{k,k}$)

Update ($A_{k+1:p,k+1:p} \leftarrow L_{k+1:p,k}, A_{k+1:p,k+1:p}$)

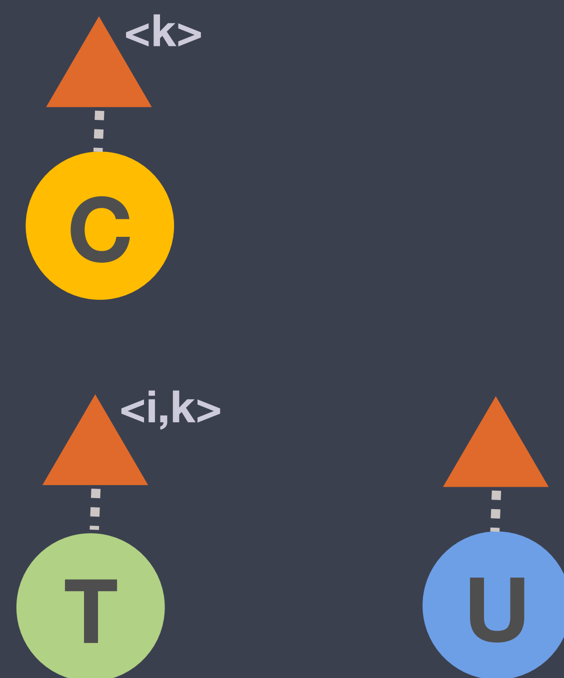
Iteration index is a natural tag

Tile Cholesky in CnC

SeqCholesky ($L_{k,k} \leftarrow A_{k,k}$)

Trisolve ($L_{k+1:p,k} \leftarrow A_{k+1:p,k},\ L_{k,k}$)

Update ($A_{k+1:p,k+1:p} \leftarrow L_{k+1:p,k},\ A_{k+1:p,k+1:p}$)



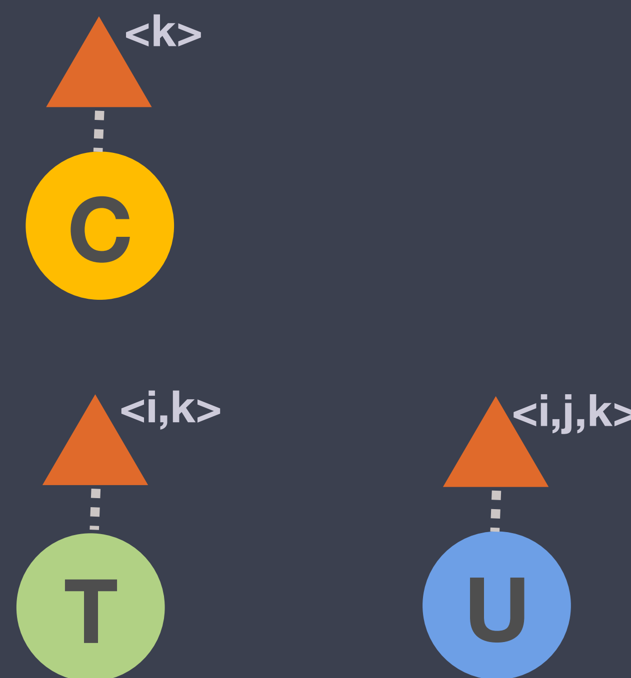
Given *k*, multiple *T* steps could go ⇒ 2-D tag

Tile Cholesky in CnC

SeqCholesky ($L_{k,k} \leftarrow A_{k,k}$)

Trisolve ($L_{k+1:p,k} \leftarrow A_{k+1:p,k}, L_{k,k}$)

Update ($A_{k+1:p,k+1:p} \leftarrow L_{k+1:p,k}, A_{k+1:p,k+1:p}$)



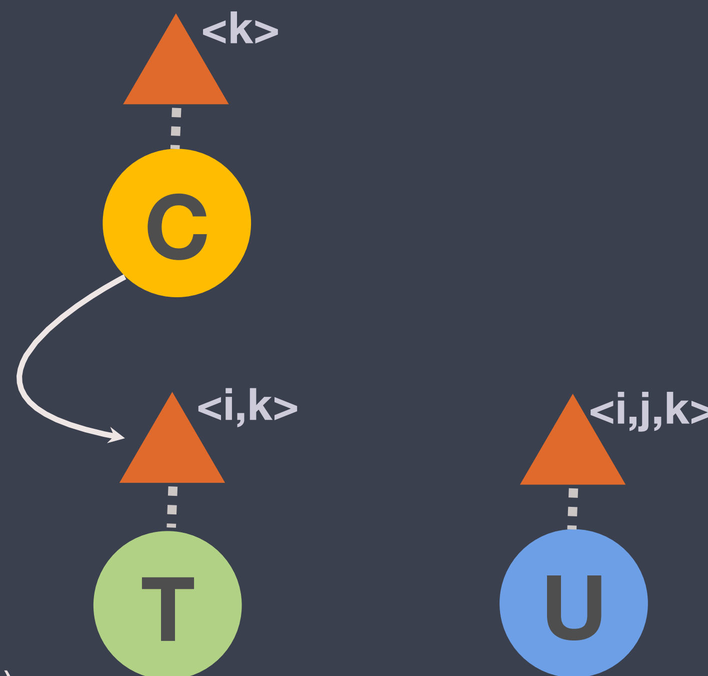
Given *k*, 2-D iteration space of update steps could go

Tile Cholesky in CnC

SeqCholesky ($L_{k,k} \leftarrow A_{k,k}$)

Trisolve ($L_{k+1:p,k} \leftarrow A_{k+1:p,k}, L_{k,k}$)

Update ($A_{k+1:p,k+1:p} \leftarrow L_{k+1:p,k}, A_{k+1:p,k+1:p}$)



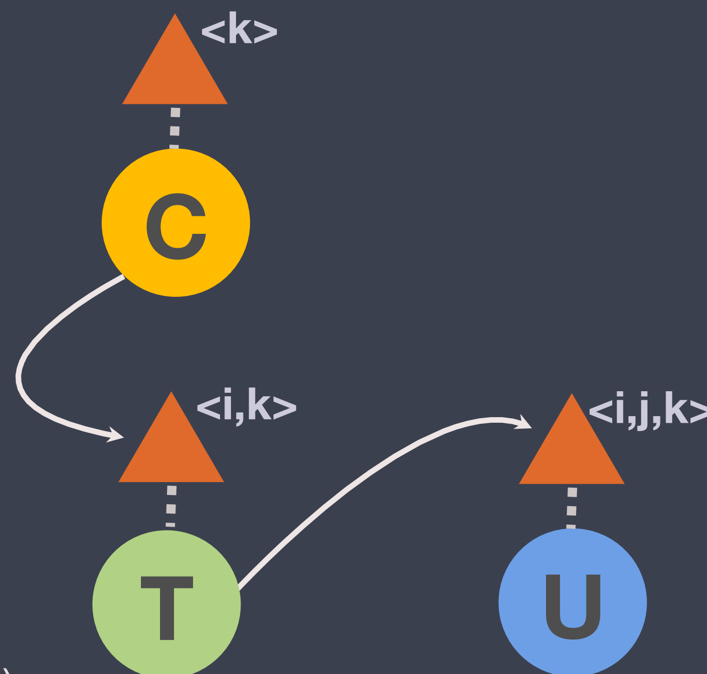
Sequential Cholesky step enables Trisolve steps

Tile Cholesky in CnC

SeqCholesky ($L_{k,k} \leftarrow A_{k,k}$)

Trisolve ($L_{k+1:p,k} \leftarrow A_{k+1:p,k},\ L_{k,k}$)

Update ($A_{k+1:p,k+1:p} \leftarrow L_{k+1:p,k},\ A_{k+1:p,k+1:p}$)



Similarly, Trisolve step enables Update steps

Coding and execution

[1] Write the specification (graph).

[2] Implement steps in a “base” language (C/C++).

[3] Build using CnC translator + compiler.

[4] Run-time system maintains collections and schedules step execution.