Hierarchical Algorithms on Hierarchical Architectures

Hatem Ltaief, George Turkiyyah & David Keyes

Extreme Computing Research Center (ECRC)

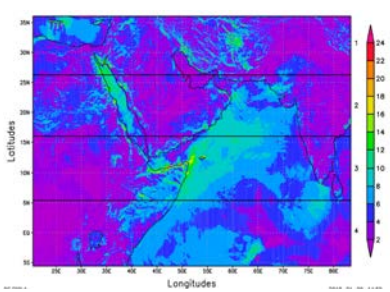
King Abdullah University of Science and Technology

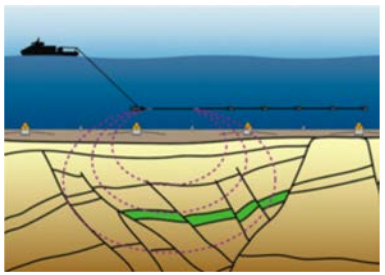
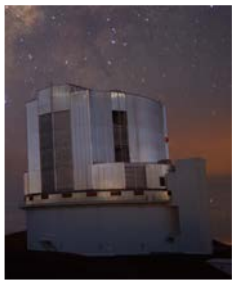
To take away (1)

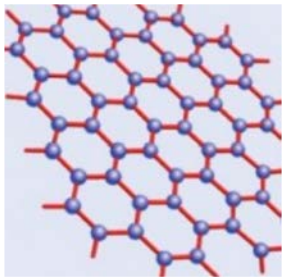
- To better exploit emerging architectures, we need new implementations of linear, least squares, eigenvalue, and singular value solvers that
  - offer tunable accuracy-time tradeoffs
  - exploit hierarchy of precisions
  - may require more flops but offer more concurrency (and thus complete faster)
- Besides exposing more concurrency, we must
  - remove synchrony and over-ordering
  - dwell as high as possible on the memory hierarchy

To take away (2)

- **With such new solvers, we can extend many applications that possess**
  - **memory capacity constraints (e.g., geospatial statistics, PDE-constrained optimization)**
  - **energy constraints (e.g., remote telescopes)**
  - **real-time constraints (e.g., wireless commun)**
  - **running time constraints (e.g., chem, materials, genome-wide associations)**

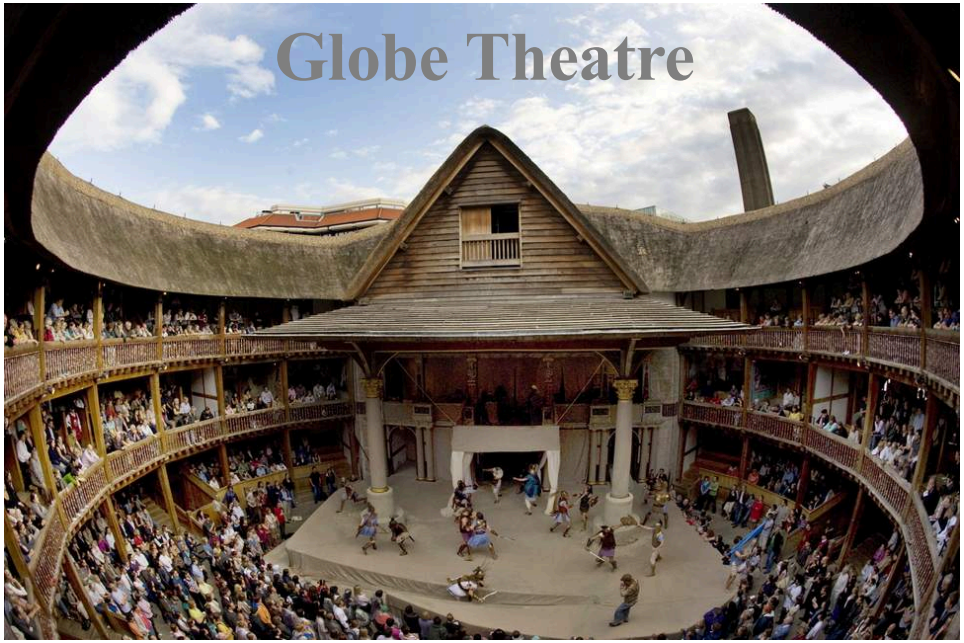





To take away (3)

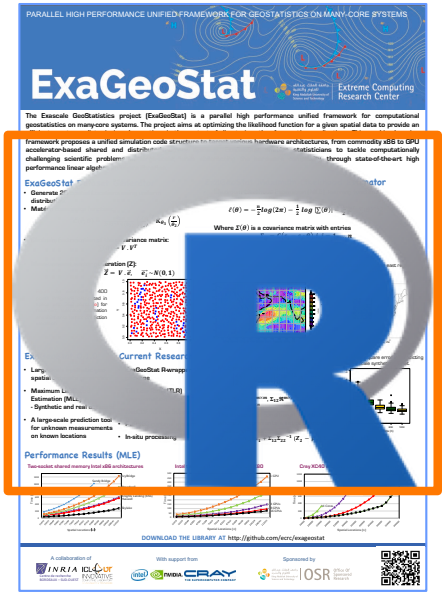
- **If you can speed up linear algebra kernels, *"the world's your oyster, which you with sword will open"* ***



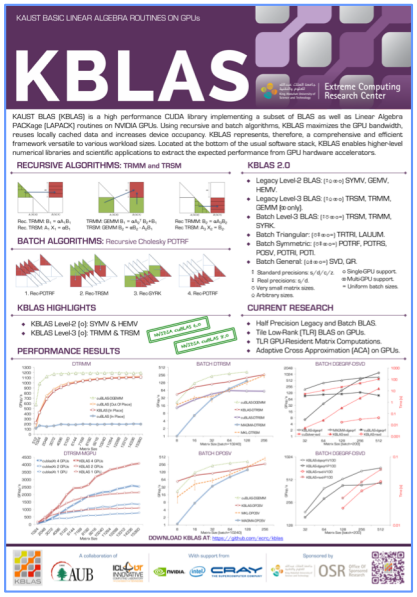
- **This can all be illustrated in applications**
  - **but not all in 30 minutes ☺**
- **Hope it highlights the relevance of this workshop**

*** Shakespeare (1600), *The Merry Wives of Windsor*, Act II, Scene II**

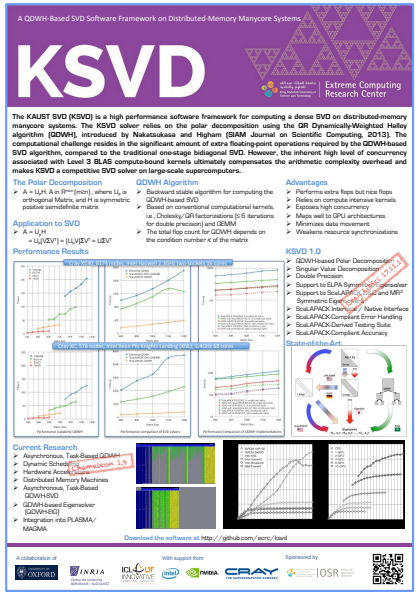
https://github.com/ecrc/



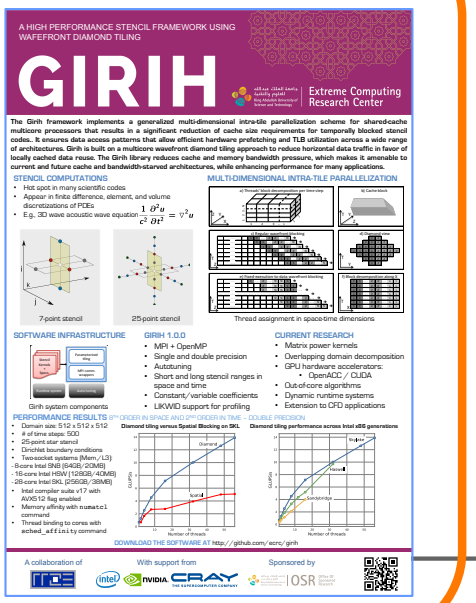
in NVIDIA cuBLAS

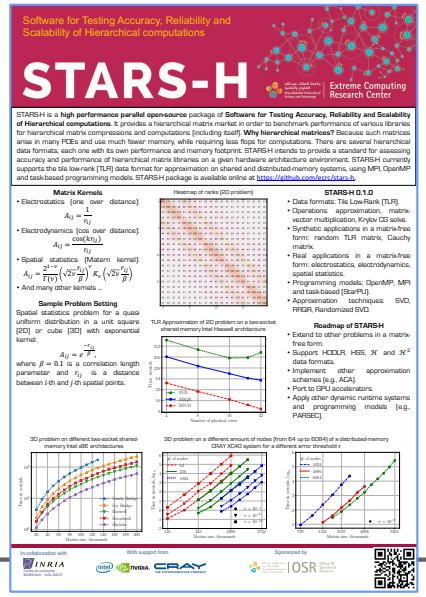


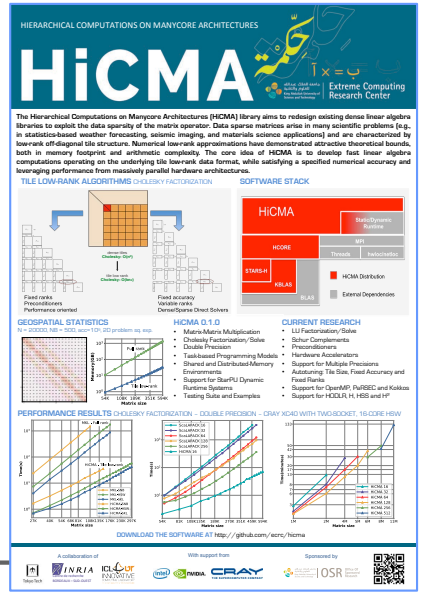
in Cray LibSci



Intel s/w for Aramco







Two universes of NLA exist side-by-side

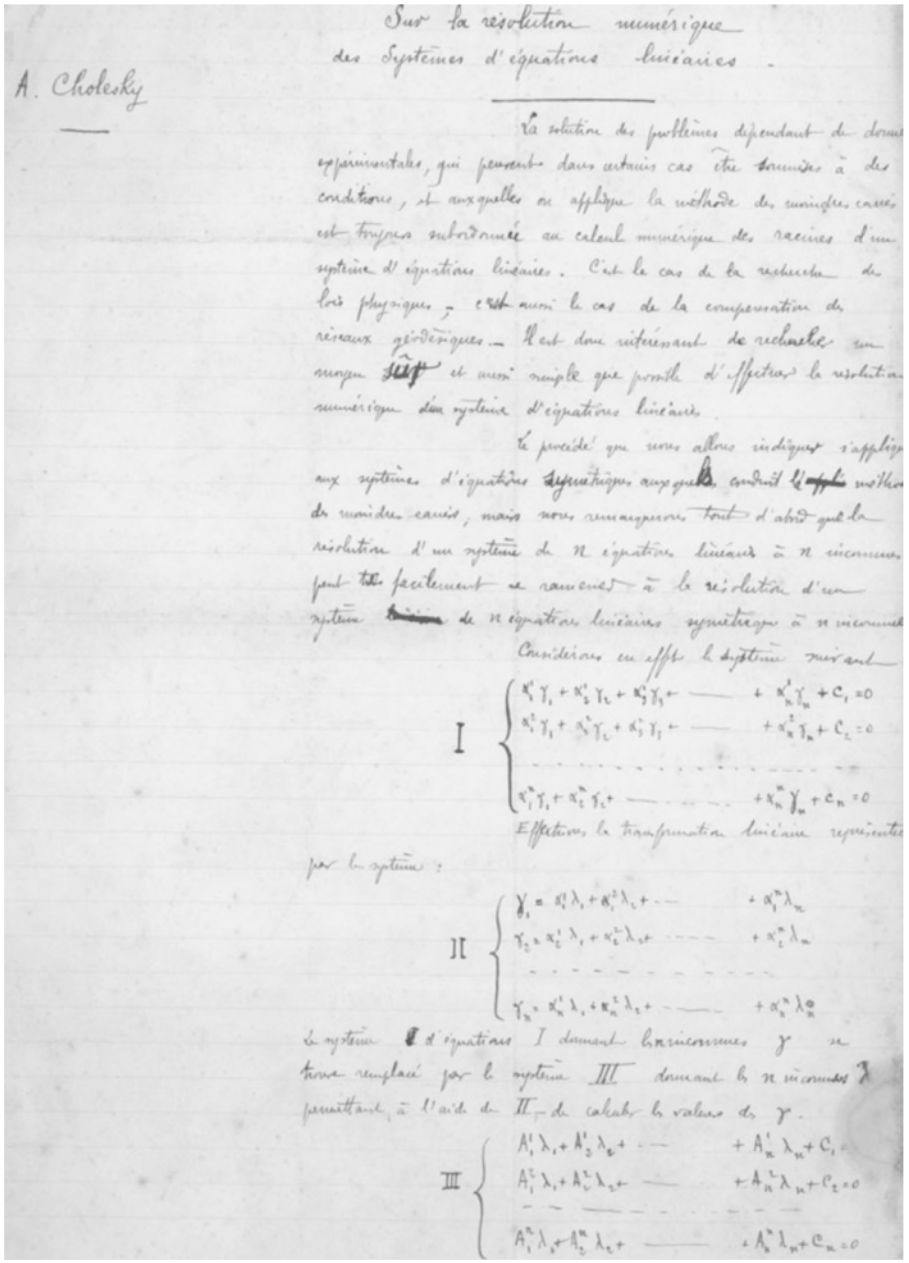
Flat

```
* Global indices *
do i {
    do j {
        for (i,j) in S do op
    }
}
```

Hierarchical

```
* Local indices *
for matrix blocks (k,l)
  do i {
      do j {
          for (i,j) in S_{k,l} do op
      }
  }
```

Algorithms were once flat (Cholesky, 1910)



Sur la résolution numérique
des Systèmes d'équations linéaires.

A. Cholesky

La solution des problèmes dépendant de données expérimentales, qui peuvent dans certains cas être soumises à des conditions, et auxquelles on applique la méthode des moindres carrés, est toujours subordonnée au calcul numérique des racines d'un système d'équations linéaires. C'est le cas de la recherche des lois physiques, c'est aussi le cas de la compensation des réseaux géodésiques. Il est donc intéressant de rechercher une marche aussi simple que possible d'effectuer la résolution numérique d'un système d'équations linéaires.

Le procédé que nous allons indiquer s'applique aux systèmes d'équations symétriques auxquels conduit la méthode des moindres carrés, mais nous remarquerons tout d'abord que la résolution d'un système de n équations linéaires à n inconnues peut très facilement se ramener à la résolution d'un système de n équations linéaires symétriques à n inconnues.

Considérons en effet le système suivant

$$\text{I}\quad \begin{cases} a_1^1\gamma_1 + a_2^1\gamma_2 + a_3^1\gamma_3 + \cdots + a_n^1\gamma_n + c_1 = 0 \\ a_1^2\gamma_1 + a_2^2\gamma_2 + a_3^2\gamma_3 + \cdots + a_n^2\gamma_n + c_2 = 0 \\ \cdots\cdots\cdots\cdots\cdots\cdots\cdots \\ a_1^n\gamma_1 + a_2^n\gamma_2 + \cdots\cdots + a_n^n\gamma_n + c_n = 0 \end{cases}$$

Effectuons la transformation linéaire représentée par le système :

$$\text{II}\quad \begin{cases} \gamma_1 = a_1^1\lambda_1 + a_2^1\lambda_2 + \cdots + a_n^1\lambda_n \\ \gamma_2 = a_1^2\lambda_1 + a_2^2\lambda_2 + \cdots + a_n^2\lambda_n \\ \cdots\cdots\cdots\cdots\cdots\cdots\cdots \\ \gamma_n = a_1^n\lambda_1 + a_2^n\lambda_2 + \cdots + a_n^n\lambda_n \end{cases}$$

le système (I) d'équations I donnant les inconnues γ se trouve remplacé par le système III donnant les n inconnues λ permettant, à l'aide de II, de calculer les valeurs des γ.

$$\text{III}\quad \begin{cases} A_1^1\lambda_1 + A_2^1\lambda_2 + \cdots + A_n^1\lambda_n + C_1 = 0 \\ A_1^2\lambda_1 + A_2^2\lambda_2 + \cdots + A_n^2\lambda_n + C_2 = 0 \\ \cdots\cdots\cdots\cdots\cdots\cdots\cdots \\ A_1^n\lambda_1 + A_2^n\lambda_2 + \cdots\cdots + A_n^n\lambda_n + C_n = 0 \end{cases}$$

$$\begin{array}{l}
\ell_{11} = \sqrt{a_{11}}; \\
\textbf{for } j = 2, \cdots, n \textbf{ do} \\
\quad \ell_{j1} = a_{j1}/\ell_{11}; \\
\textbf{end} \\
\textbf{for } i = 2, \cdots, n-1 \textbf{ do} \\
\quad \ell_{ii} = (a_{ii} - \sum_{k=1}^{i-1} \ell_{ik}^2)^{1/2}; \\
\quad \textbf{for } j = i+1, \cdots, n \textbf{ do} \\
\quad\quad \ell_{ji} = \left(a_{ji} - \sum_{k=1}^{i-1} \ell_{jk}\ell_{ik}\right)/\ell_{ii}; \\
\quad \textbf{end} \\
\quad \ell_{nn} = (a_{nn} - \sum_{k=1}^{n-1} \ell_{nk}^2)^{1/2}; \\
\textbf{end}
\end{array}$$

triangular recurrence

Architectures were flat, as well (vN, 1945)

First Draft of a Report
on the EDVAC

by

John von Neumann

Contract No. W-670-ORD-4926

Between the

United States Army Ordnance Department

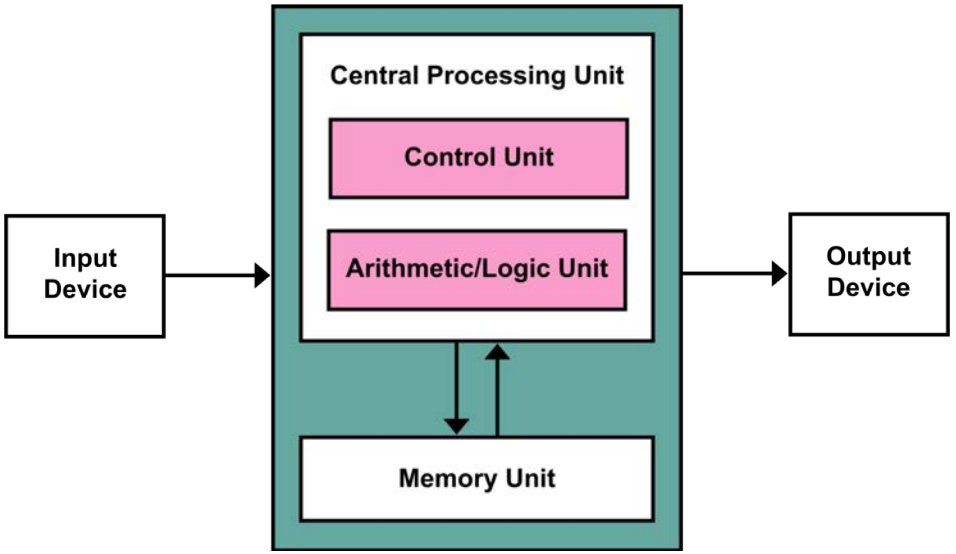
and the

University of Pennsylvania

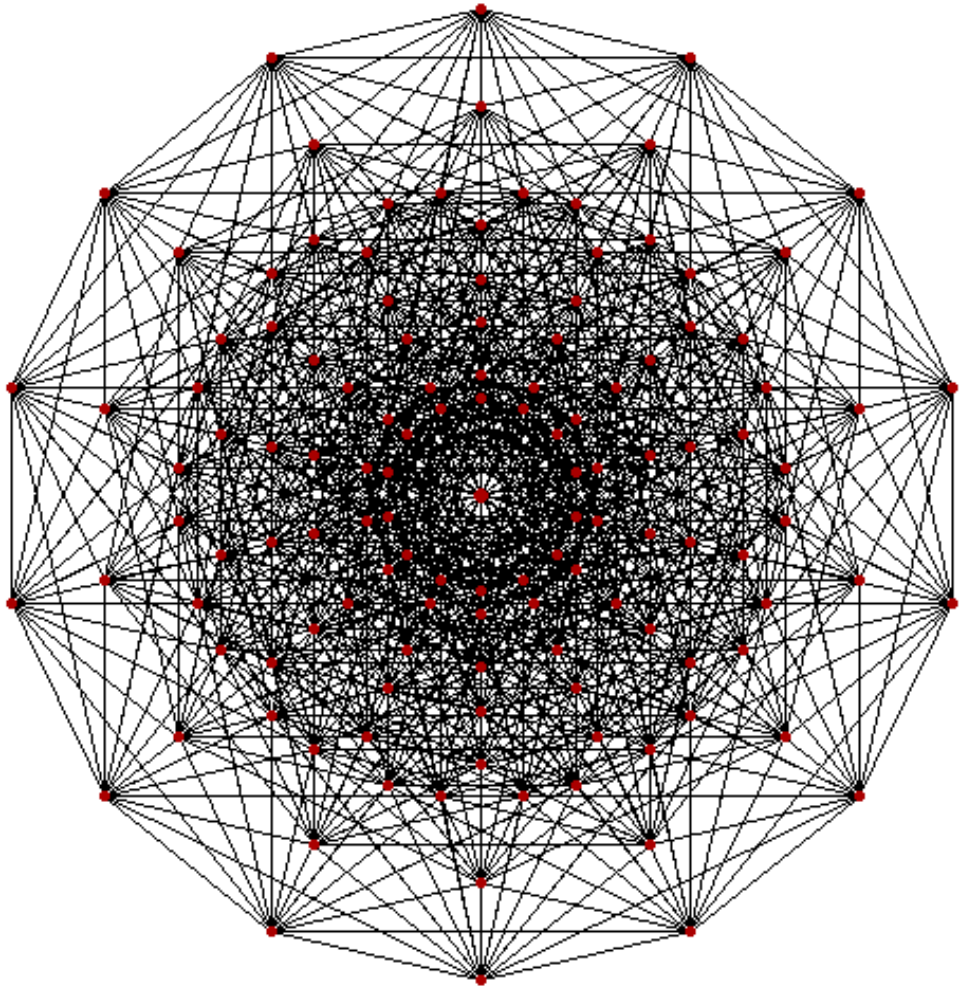
Moore School of Electrical Engineering
University of Pennsylvania

June 30, 1945

National Bureau of Standards
Division 12
Data Processing Systems



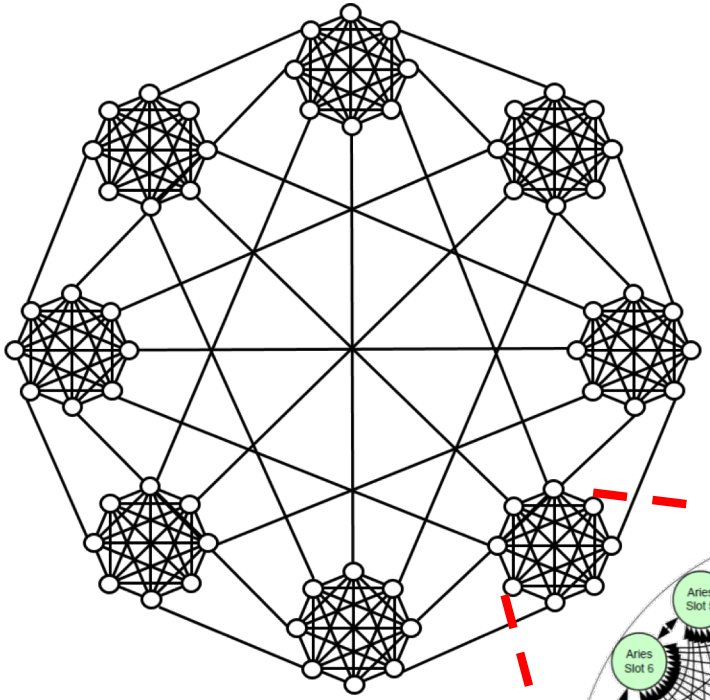
Since 1985: “horizontal” structure…

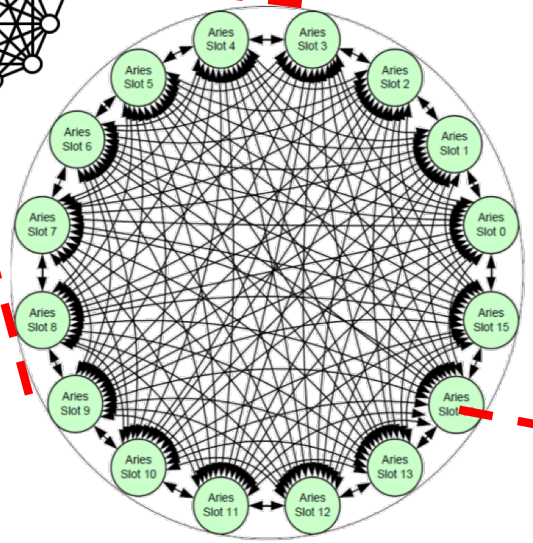


128-node hypercube

c/o Computer History Museum, Mountain View, CA

Today's "horizontal" structure…

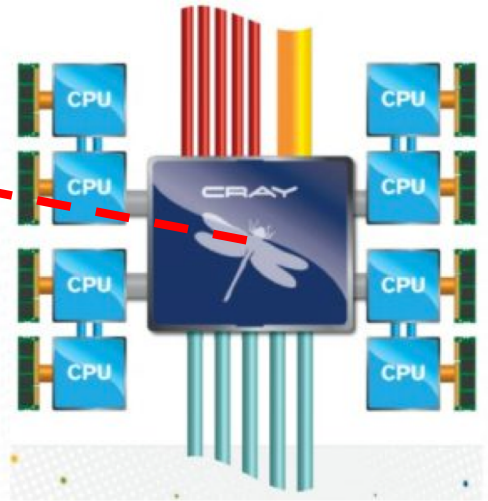




dragonfly network

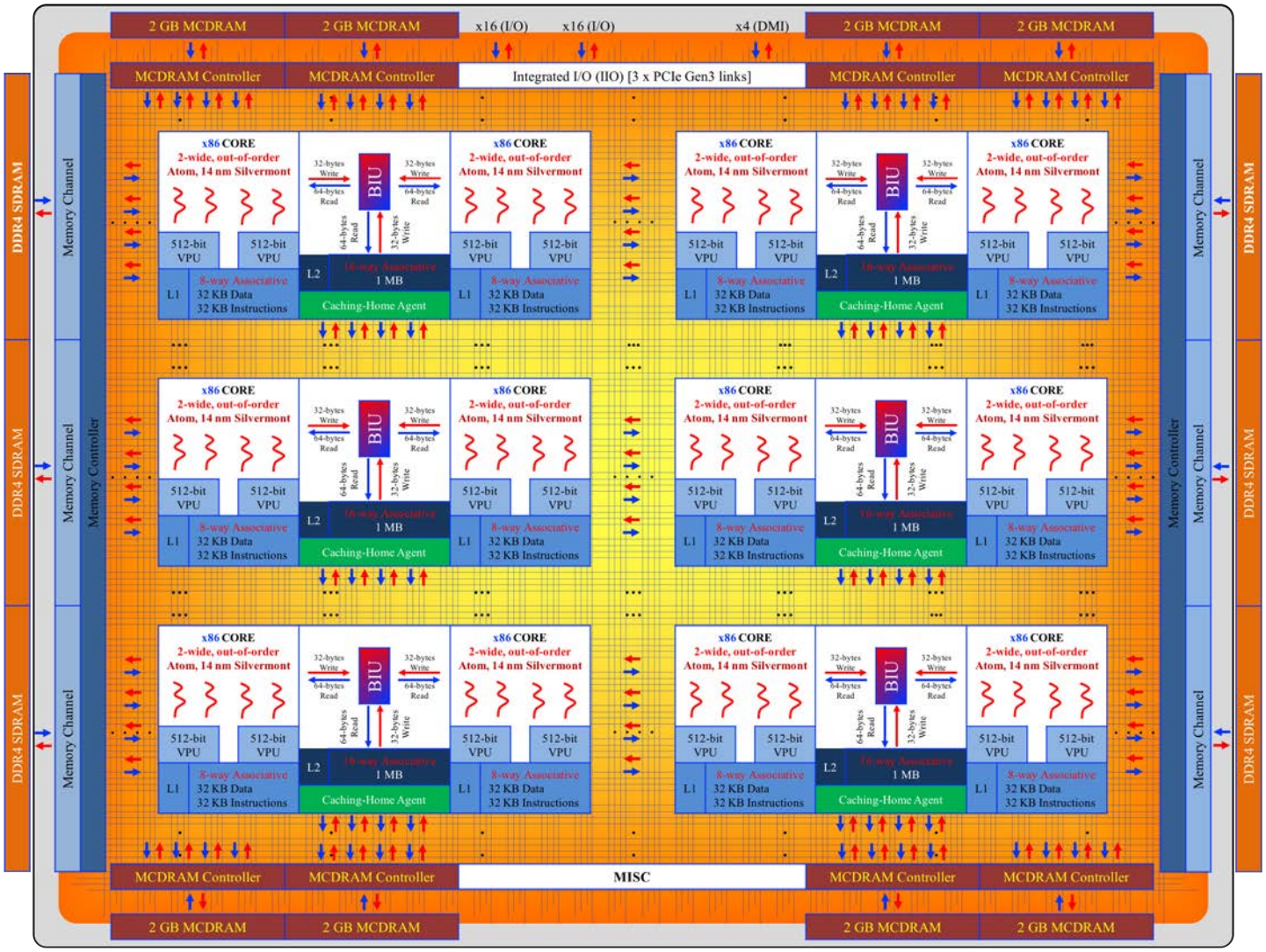
Cray's "Aries" network

- copper *within* a cabinet
- optical *between* cabinets
- scalability of a fat tree
- cost of a torus
- maximum of three hops between any pair of the 200,000 Xeon cores in KAUST's Cray XC40



c/o Cray

And now add: “vertical” structure



c/o M. Farhan, KAUST

Intel’s 256-core Knights Landing Processor

- **nested levels of cache**
- **MCDRAM**
- **DDR4 SDRAM**

all within a 3 Tflop/s node

Two decades of evolution

1997

2017

ASCI Red at Sandia

1.3 TF/s, 850 KW

Cavium ThunderX2

~ 1.1 TF/s, ~ 0.2 KW

3.5 orders of magnitude

Hierarchies do not necessarily match!

As humans managing implementation complexity, we would all prefer:

- **hierarchical algorithms on flat architectures**

or even (suboptimally)

- **flat algorithms on hierarchical architectures**

Reality

We go to exascale with the architectures we have, not with the architectures we want. *

- A 4,000-node subset of ORNL's Summit sustains 1.88 ExaOp/s of mixed precision on a genomics application
- Majority of these operations are half-precision (16-bit floating point) NVIDIA tensor-core matrix-matrix multiplies

* paraphrase of D. Rumsfeld, Cable News Network, 8 Dec 2004.

Now: hierarchy of precisions

SIAM J. SCI. COMPUT.
Vol. 40, No. 2, pp. A817–A847

ACCELERATING THE SOLUTION OF LINEAR SYSTEMS BY ITERATIVE REFINEMENT IN THREE PRECISIONS*

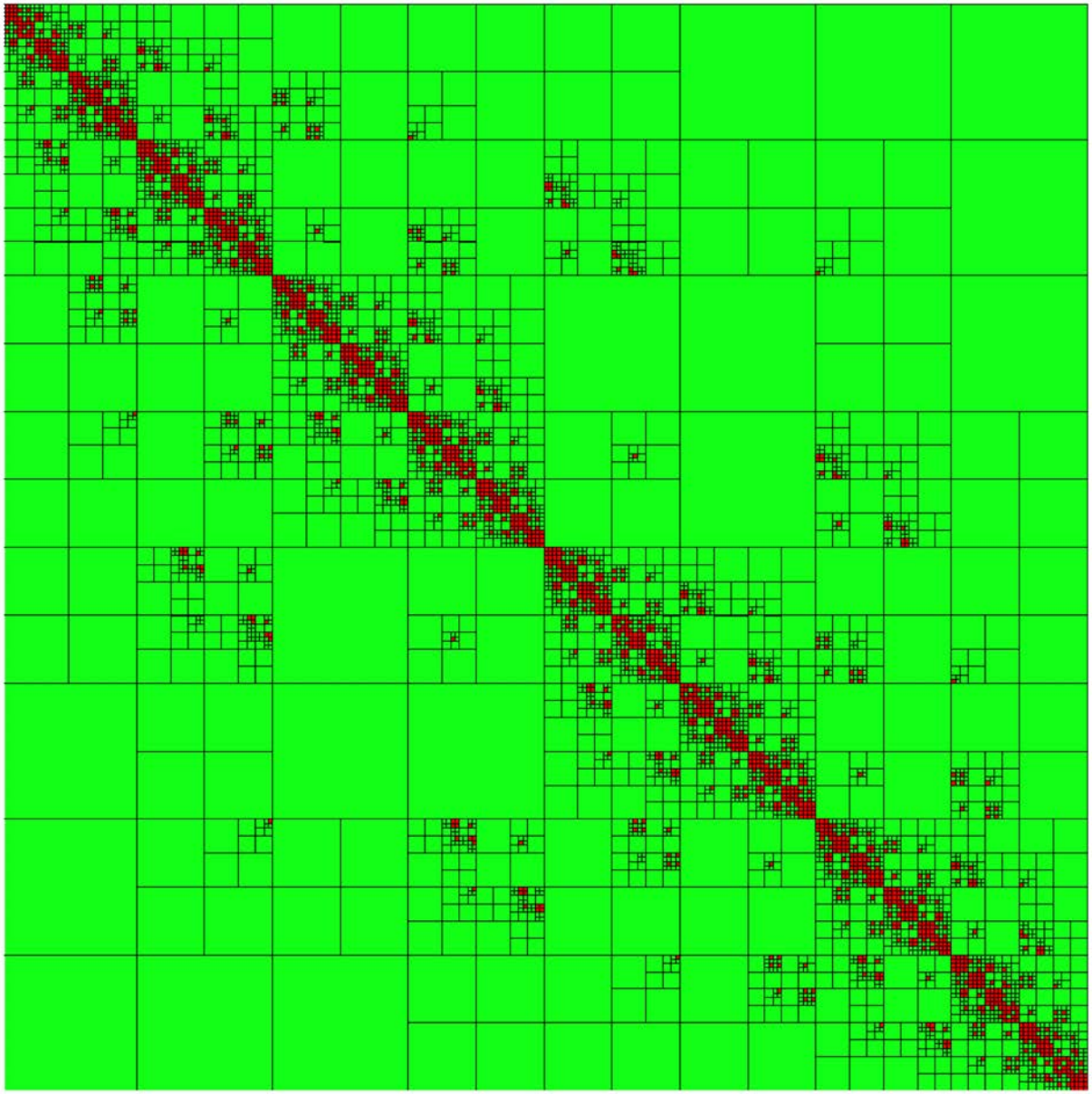
ERIN CARSON† AND NICHOLAS J. HIGHAM‡

TABLE 1.1

Summary of existing rounding error analyses for iterative refinement in floating point arithmetic indicating (a) *whether the analyses apply to LU factorization only or to an arbitrary solver,* (b) *whether the backward or forward error analyses are componentwise ("comp") or normwise ("norm"), and* (c) *the assumptions on the precisions* u_f, u_s, u, u_r *in Algorithm* 1.1 ($u_f = u$ *and* $u_s = u_f$ *unless otherwise stated*).

	Year	Solver	Forward error	Backward error	Precisions
Moler [27]	1967	LU	norm	–	$u \geq u_r$
Stewart [36]	1973	LU	norm	–	$u \geq u_r$
Jankowski et al. [22]	1977	arb.	norm	norm	$u = u_r$
Skeel [34]	1980	LU	comp	comp	$u \geq u_r$
Higham [17]	1991	arb.	comp	comp	$u = u_r$
Higham [18], [19]	1997	arb.	comp	comp	$u \geq u_r$
Tisseur [37]	2001	arb.	norm	norm	$u \geq u_r$
Langou et al. [24]	2006	LU	norm	norm	$u_f \geq u = u_r$
Carson and Higham [9]	2017	arb.	comp	–	$u \geq u_r$
This work	2017	arb.	comp	comp, norm	$u_f \geq u_s \geq u \geq u_r$

Now: hierarchy of ranks



Architectural challenge

- **Memories are hierarchical in an increasing number of levels**
  - **stronger-than-ever incentive to tune algorithms for register & cache reuse**
  - **additional flop/s cost little, within a given workingset of data that fits in highest level cache**
  - **more computation leading to less communication and/or synchronization may be a good trade-off**

It's not just bandwidth; it's energy

- **Access SRAM (registers, cache) ~ 10 fJ/bit**
- **Access DRAM on chip ~ 1 pJ/bit**
- **Access HBM/MCDRAM (few mm) ~ 10 pJ/bit**
- **Access DDR3 (few cm) ~ 100 pJ/bit**

~ 10^4 advantage in energy for staying in cache!

similar ratios for *latency* as for *bandwidth* and *energy*

Algorithmic imperatives

1) Reside “high” on the memory hierarchy
   - as close as possible to the processing elements
2) Reduce communication and synchrony
   - in frequency and/or span
3) … SIMT-style batching … algorithm-based fault tolerance … etc.

Widely applicable strategies

1) **Employ dynamic runtime systems based on directed acyclic task graphs (DAGs)**
   - **e.g., Charm++, Quark, StarPU, Legion, OmpSs, HPX, ADLB, Argo, ParSec**
   - **dynamic scheduling capabilities in OpenMP**
2) **Exploit data sparsity of hierarchically low-rank type**
   - **meet the "curse of dimensionality" with the "blessing of low rank"**
3) **Code libraries to various architecture while presenting high-level application programmer interface**

1) Taskification based on DAGs

- **Advantages**
  - **remove artifactual synchronizations in the form of subroutine boundaries**
  - **remove artifactual orderings in the form of pre-scheduled loops**
  - **expose more concurrency**
- **Disadvantages**
  - **pay overhead of managing task graph**
  - **potentially lose some memory locality**

2) Hierarchically low-rank operators

- **Advantages**
  - **shrink memory footprints to live higher on the memory hierarchy**
    - **higher means quick access (↑ arithmetic intensity)**
  - **reduce operation counts**
  - **tune work to accuracy requirements**
    - **e.g., preconditioner versus solver**
- **Disadvantages**
  - **pay cost of compression**
  - **not all operators compress well**

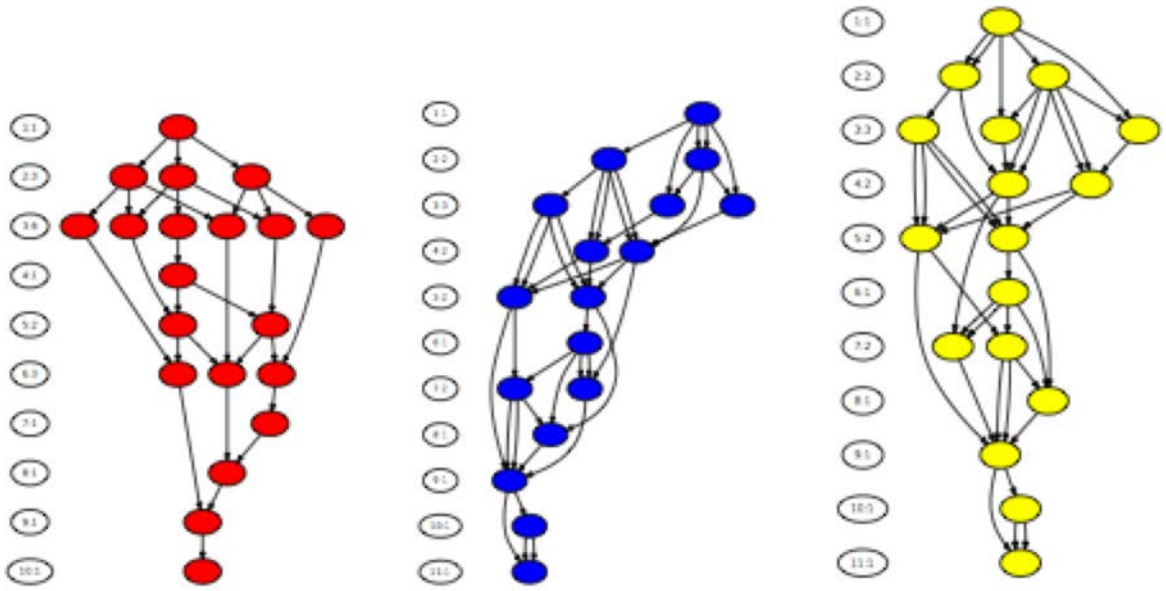
3) Code to the architecture

- **Advantages**
  - **tiling and recursive subdivision create large numbers of small problems that can be marshaled for batched operations on GPUs and MICs**
    - **amortize call overheads**
    - **polyalgorithmic approach based on block size**
  - **non-temporal stores, coalesced memory accesses, double-buffering, etc. reduce sensitivity to memory**
- **Disadvantages**
  - **code is more complex**
  - **code is architecture-specific at the bottom**

1) Reduce over-ordering and synchronization through DAGs, ex.: generalized eigensolver

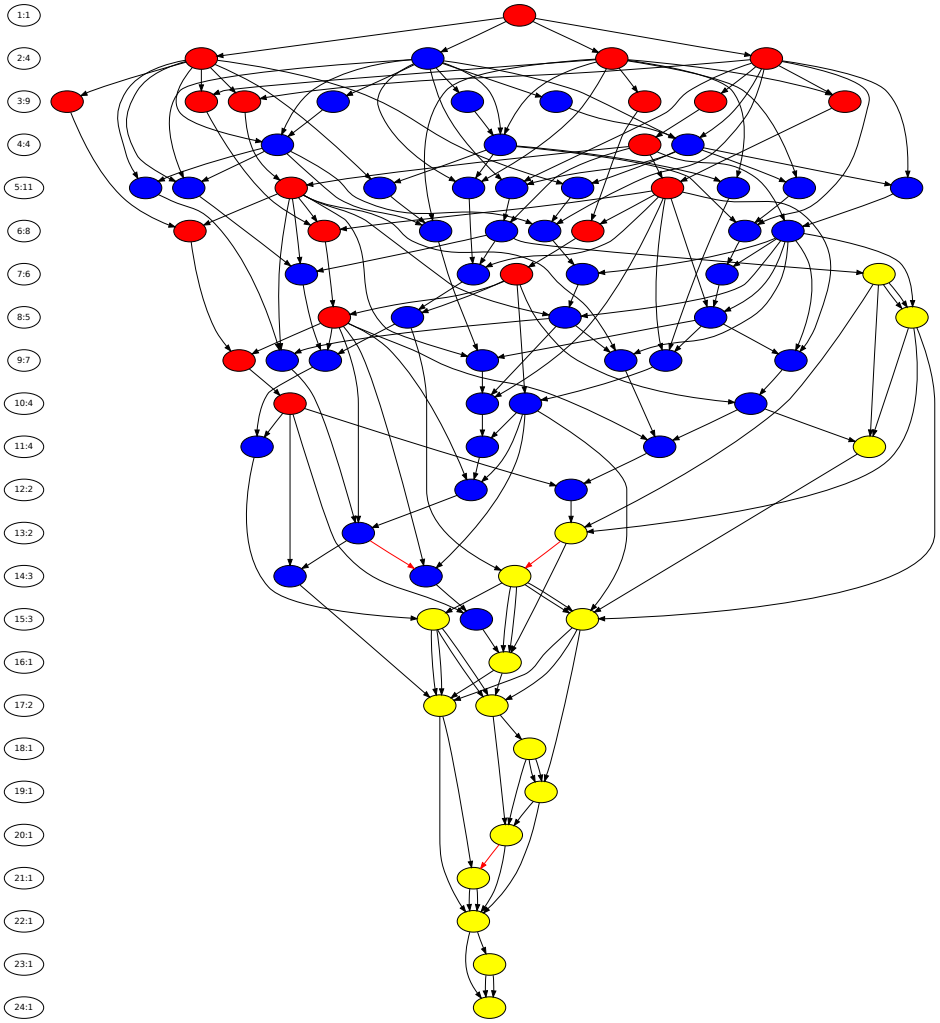
$$Ax = \lambda Bx$$

Operation	Explanation	LAPACK routine name
1. $B = L \times L^T$	Cholesky factorization	POTRF
2. $C = L^{-1} \times A \times L^{-T}$ or HEGST	application of triangular factors	SYGST
3. $T = Q^T \times C \times Q$	tridiagonal reduction	SYEVD or HEEVD
4. $Tx = \lambda x$	QR iteration	STERF



Loop nests and subroutine calls, with their over-orderings, can be replaced with DAGs

- Diagram shows a dataflow ordering of the steps of a 4×4 symmetric generalized eigensolver
- Nodes are tasks, color-coded by type, and edges are data dependencies
- Time is vertically downward
- Wide is good; short is good



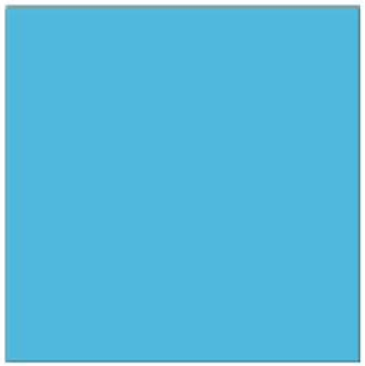
2) Reduce memory footprint and operation complexity with low rank

- **Replace dense blocks with hierarchical representations when they arise during matrix operations**
  - **use high accuracy (high rank, but typically less than full) to build "exact" solvers**
  - **use low accuracy (low rank) to build preconditioners**
- **Tune block structure and rank parameters to variety of hardware configurations**

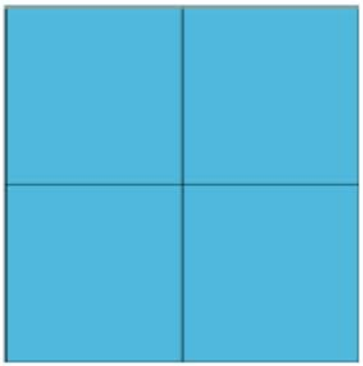
Key tool: hierarchical matrices

- [Hackbusch, 1999] **: off-diagonal blocks of typical differential and integral operators have low effective rank**
- **Similarly: Schur complements of the above, covariance matrices from statistics, Hessians from optimization, etc.**
- **By exploiting low rank, k , memory requirements and operation counts approach optimal in matrix dimension n:**
  - **polynomial in k**
  - **lin-log in n**
  - **constants carry the day**
- **Such hierarchical representations navigate a compromise**
  - **fewer blocks of larger rank ("weak admissibility") or**
  - **more blocks of smaller rank ("strong admissibility")**

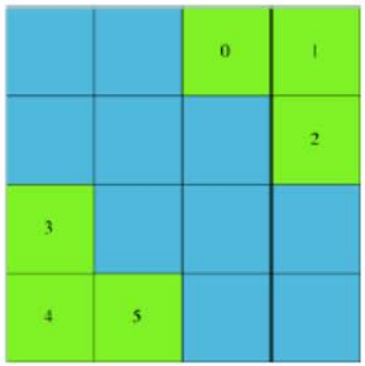
Recursive construction of an $\mathcal{H}$-matrix



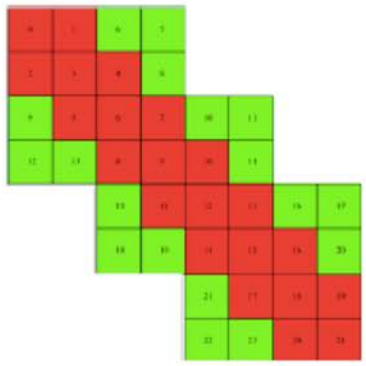
Step 0



Step 1



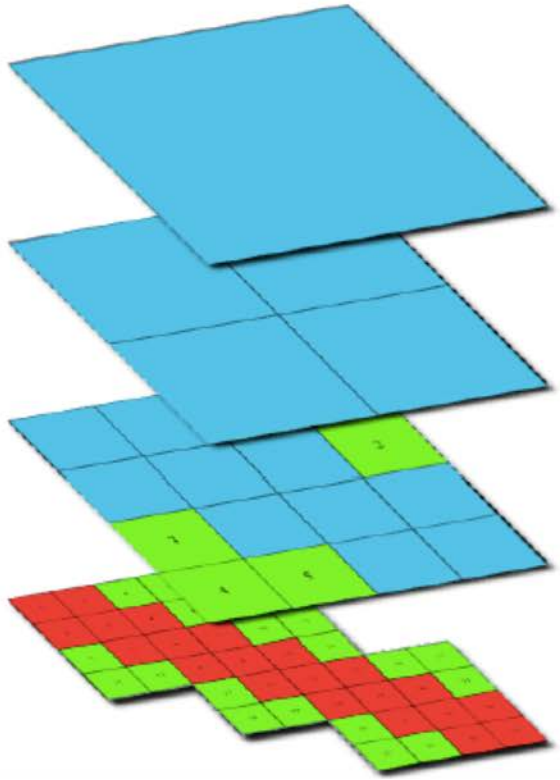
Step 2



Step 3

Specify two parameters:

- **Block size acceptably small to handle densely**
- **Rank acceptably small to represent block**

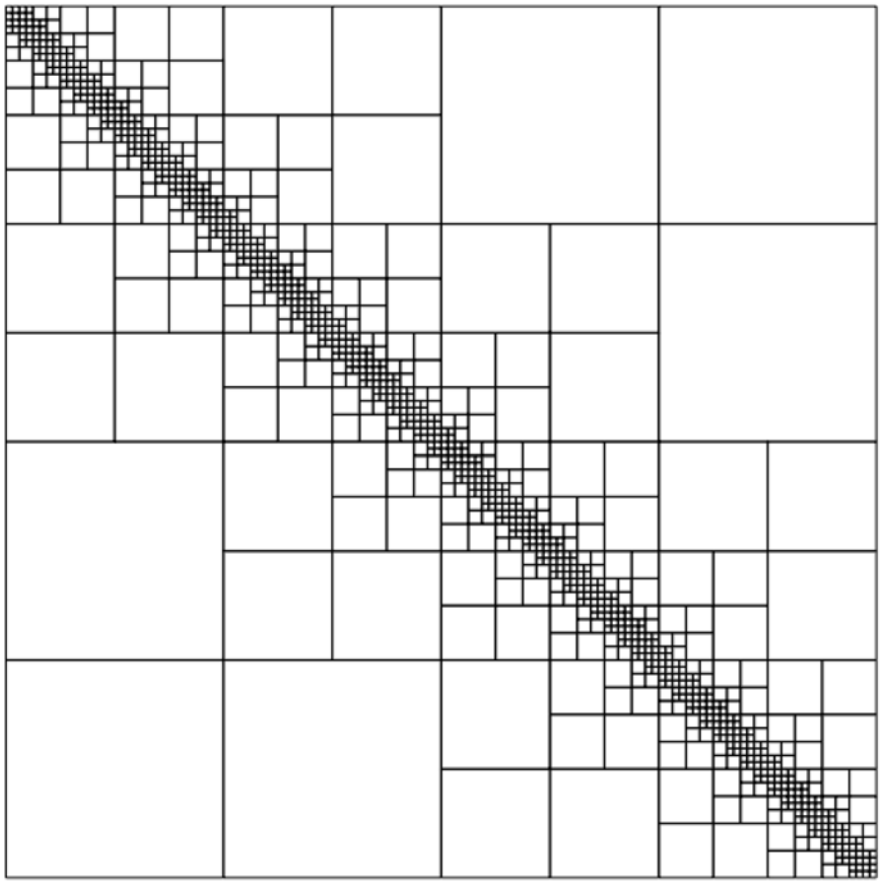


Until each block is acceptably small:

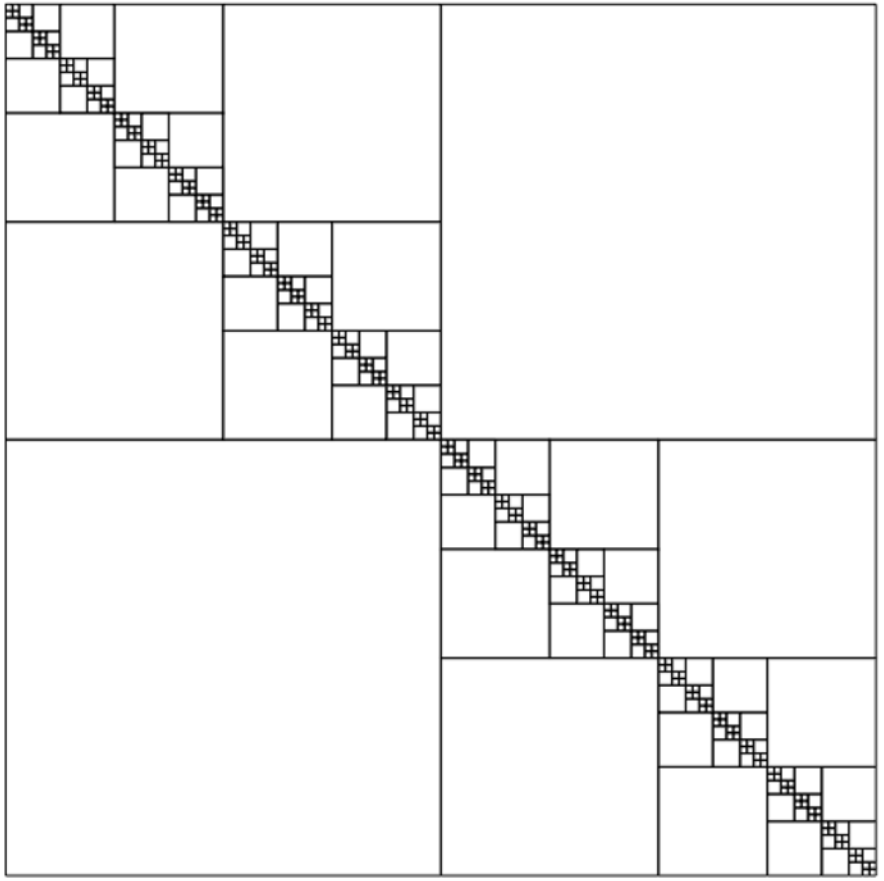
- **Is rank acceptably small?**
- **If not, subdivide block**

Take union of leaf blocks

Tree-like structures of "Standard (strong)" vs. "weak" admissibility



strong admissibility



weak admissibility

after Hackbusch, *et al.*, 2003

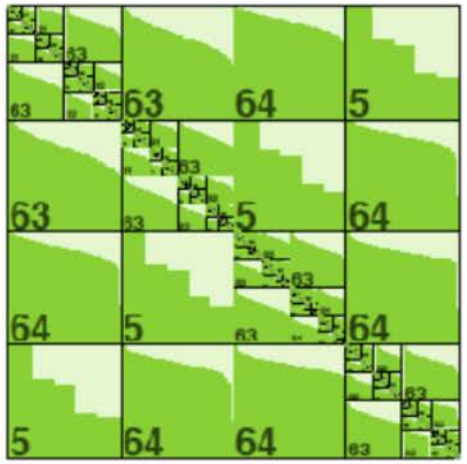
Hierarchically low-rank "renaissance"

Replace dense linear algebra

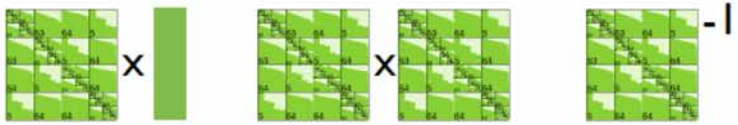
Compute : $\mathcal{O}(N^3) \longrightarrow \mathcal{O}(k^a N \log^b N)$

Memory : $\mathcal{O}(N^2) \longrightarrow \mathcal{O}(kN)$

Hierarchical off-diagonal blocks
Approximated with rank k
a and b are small constants



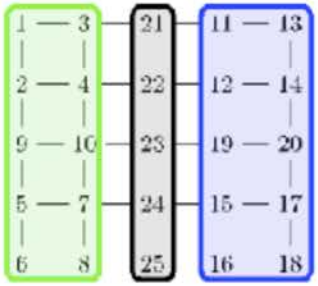
Augment sparse linear algebra

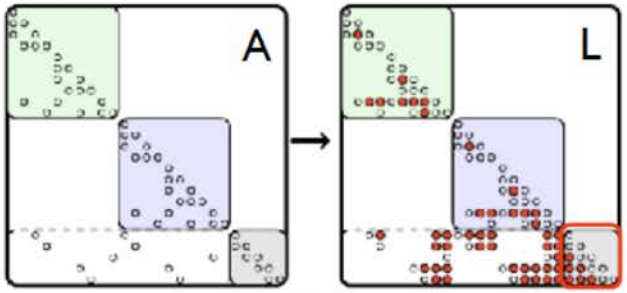


Sparse direct solvers

Schur complement (frontal matrix) is dense but numerically low-rank

Nested dissection





Schur complement

Iterative solvers

Use small k to precondition

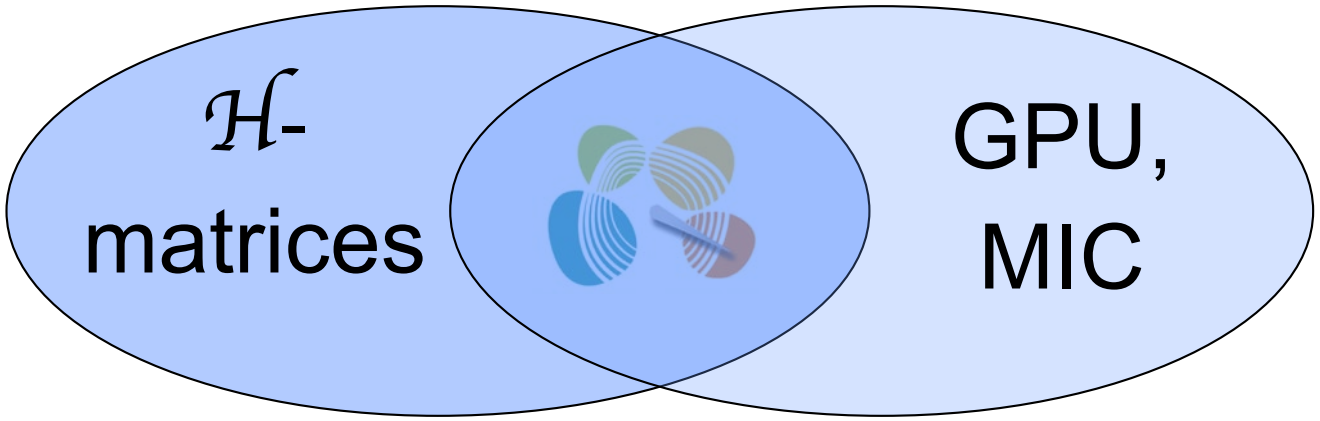
Less sensitive to matrix condition than multigrid

c/o Rio Yokota (Tokyo Tech/KAUST)

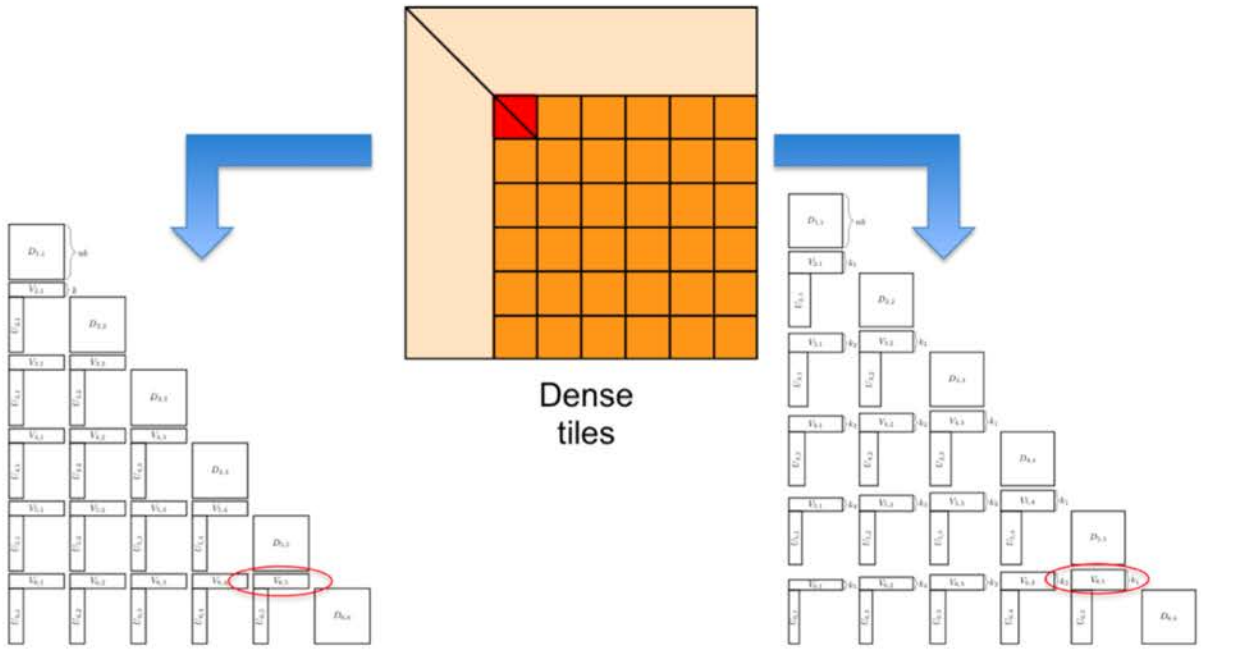
Hierarchical algorithms and extreme scale

Must address the *tension* between

- **highly uniform vector, matrix, and general SIMT operations**
- **hierarchical algorithms with tree-like data structures and scale recurrence**



Tile Low Rank (TLR) is a compromise between optimality and complexity



T. Mary, PhD Dissertation, Block Low-Rank multifrontal solvers: complexity, performance, and scalability, 2017.

C. Weisberger, PhD Dissertation, Improving multifrontal solvers by means of algebraic Block Low-Rank representations, 2013.

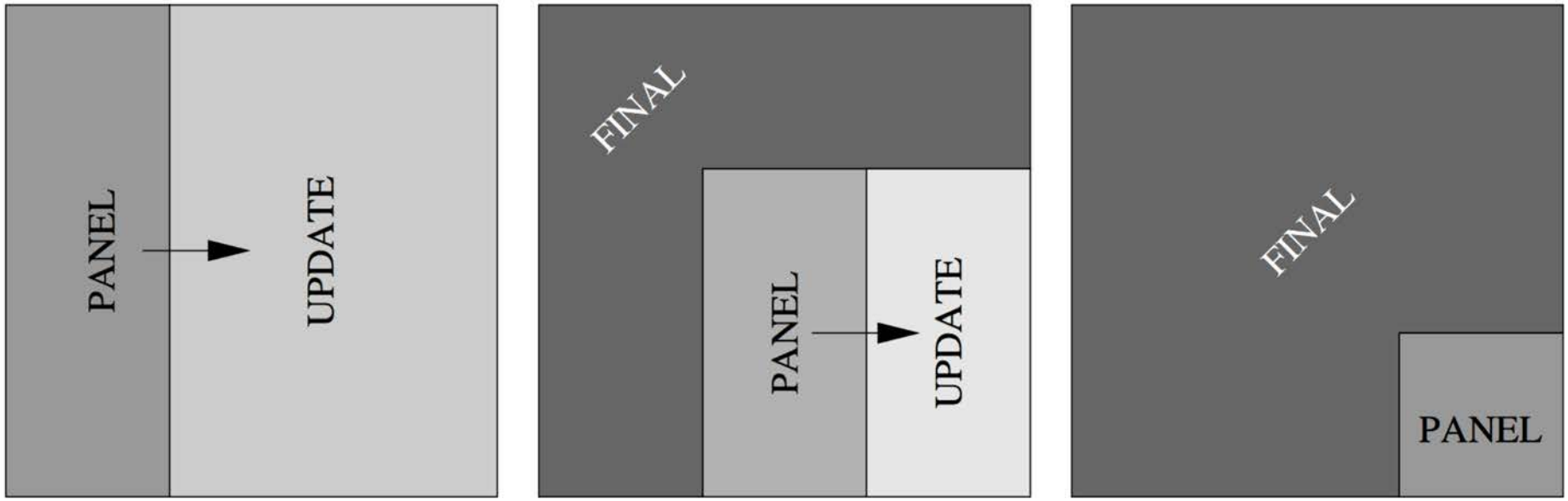
Example: Cholesky

The Cholesky factorization of an $N \times N$ real symmetric, positive-definite matrix A has the form

$$A = LL^T,$$

where L is an $N \times N$ real lower triangular matrix with positive diagonal elements.

Panel algorithm (LAPACK's POTRF)



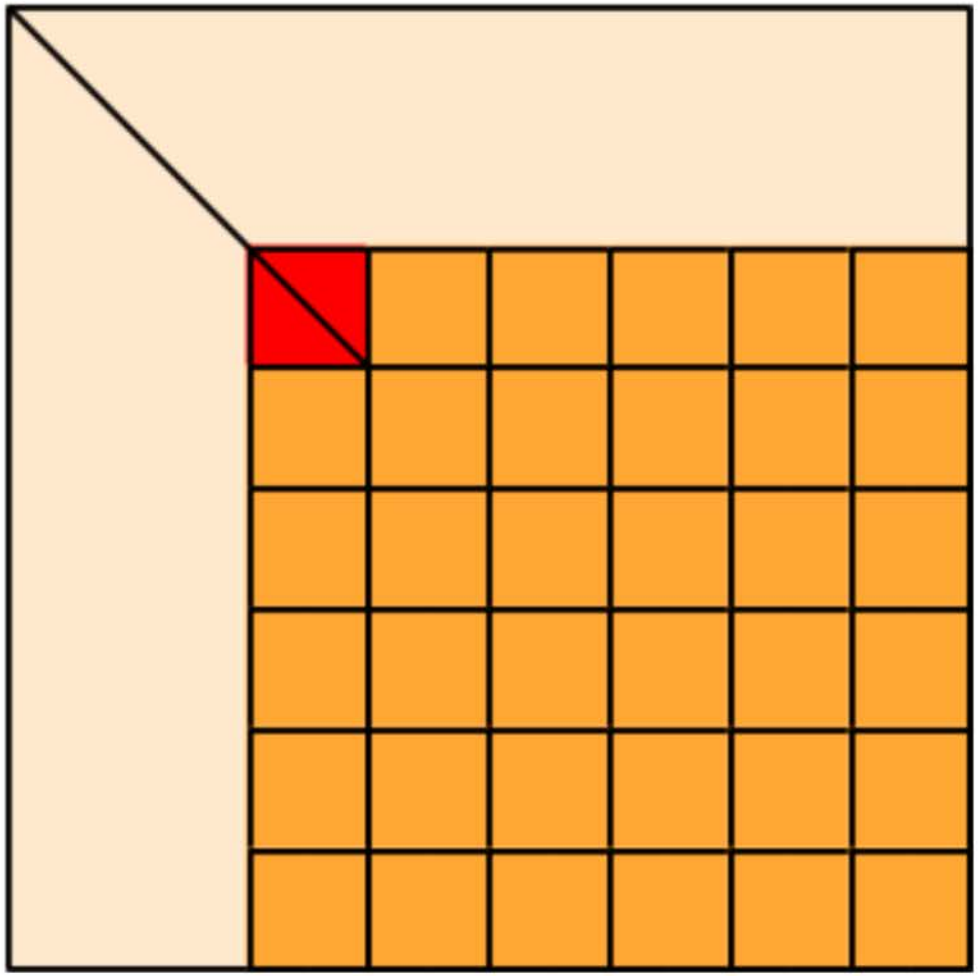
(a) First step. (b) Second step. (c) Third step.

Left- and right-looking variants, column-blocked for BLAS3

Decreasing concurrency

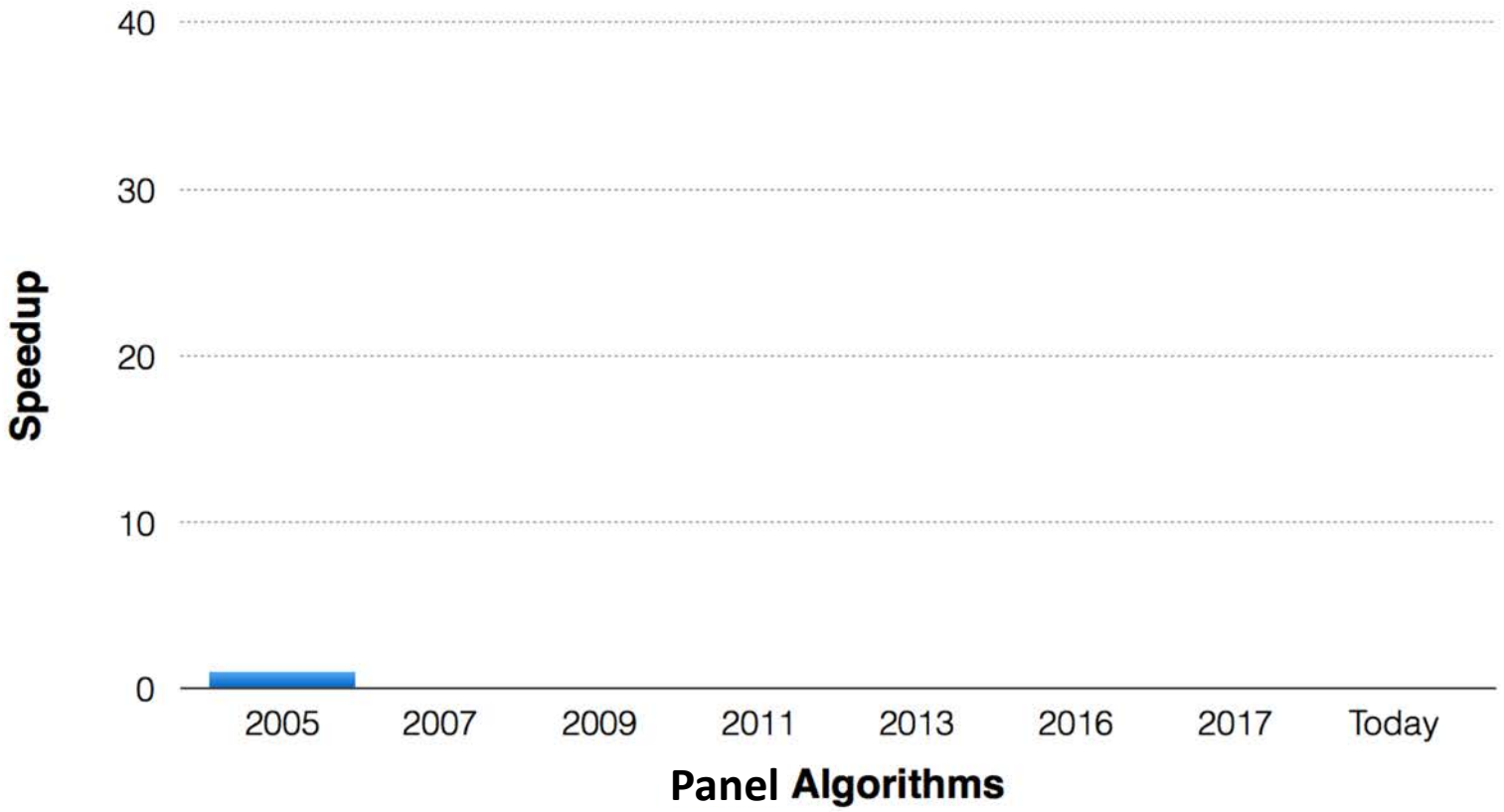
Artifactual over-ordering

Tile algorithm
(PLASMA, MAGMA, Chameleon)

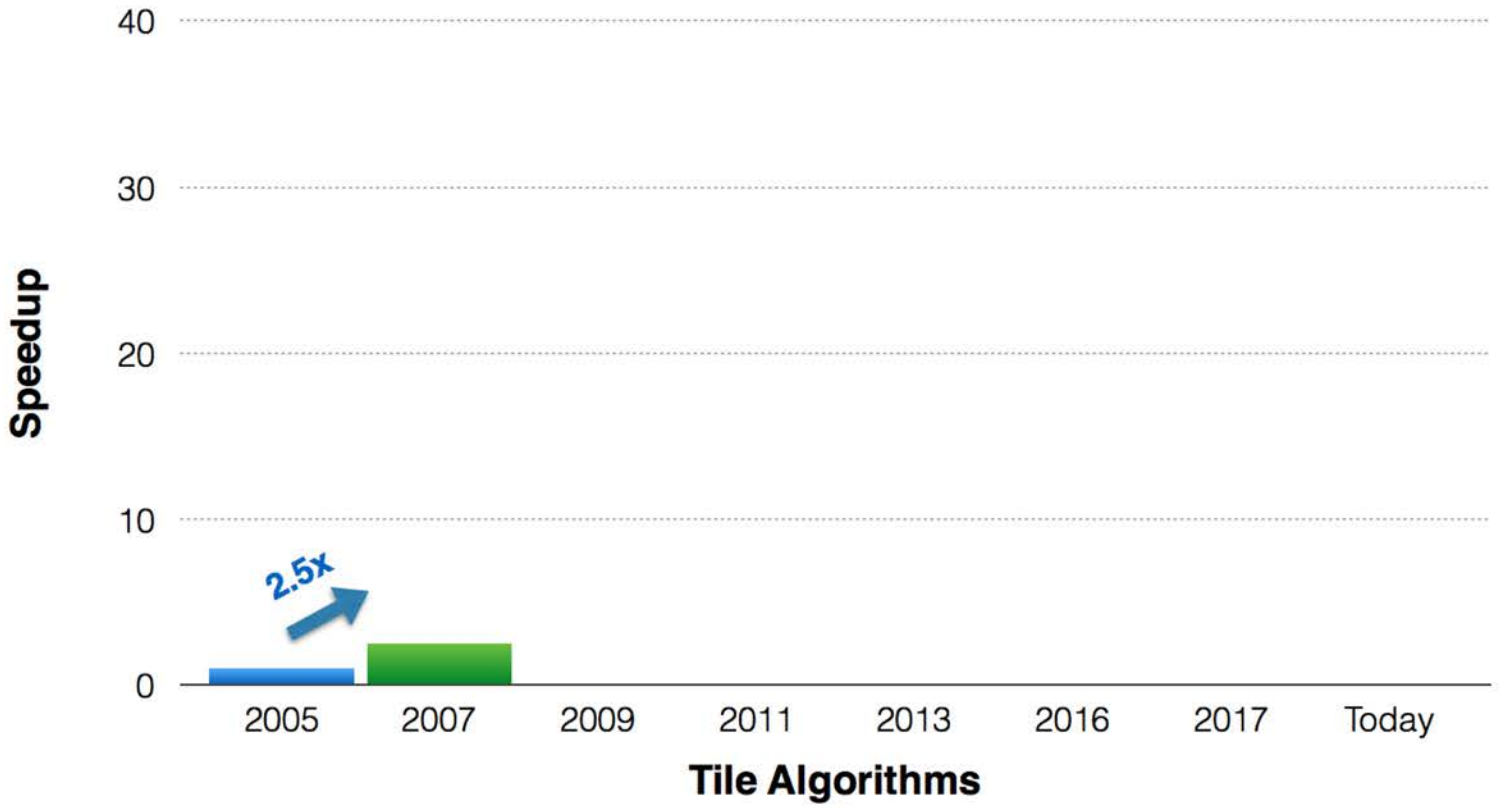


Implemented with DAG-based scheduling

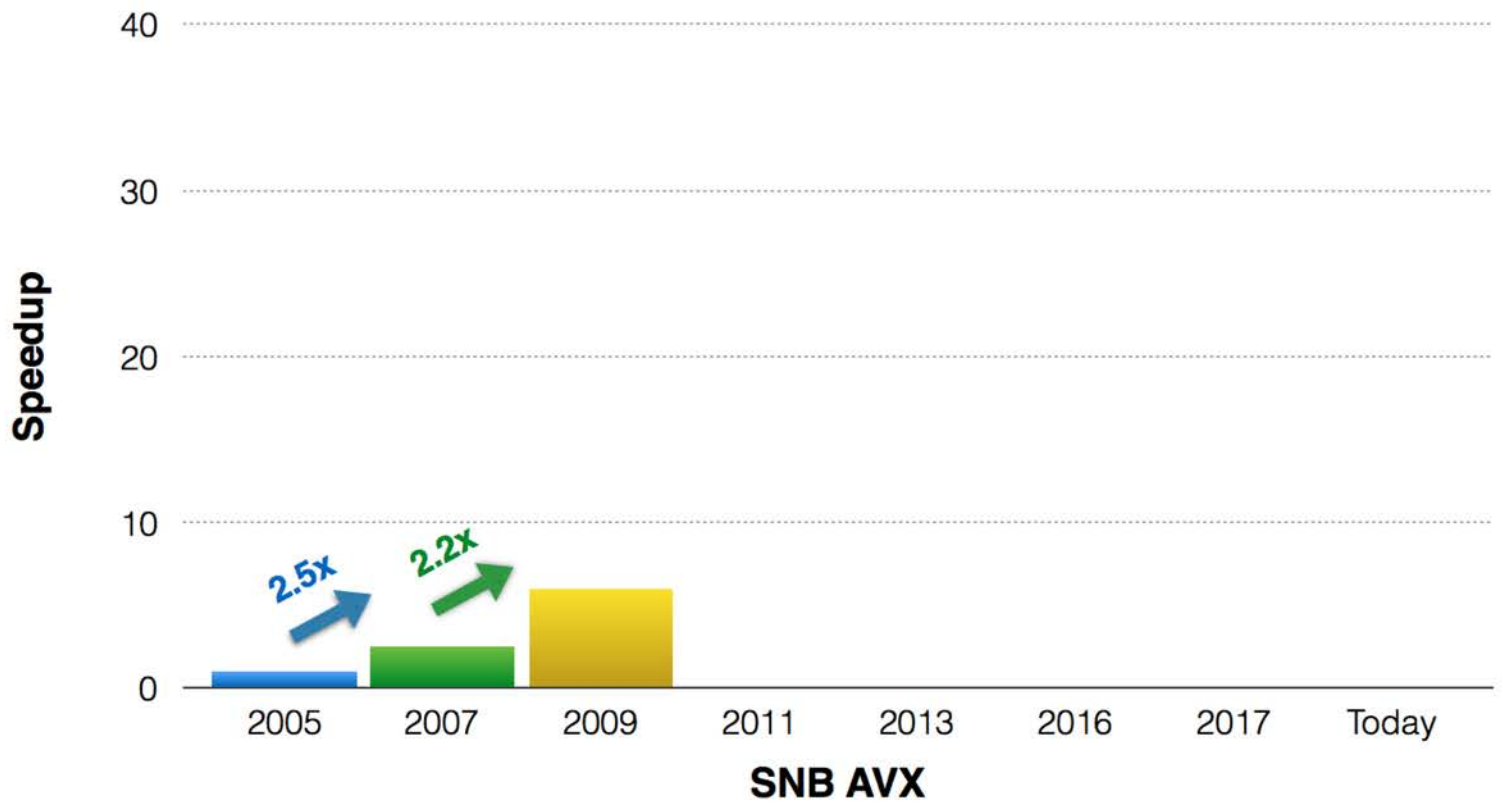
Performance evolution of dense Cholesky



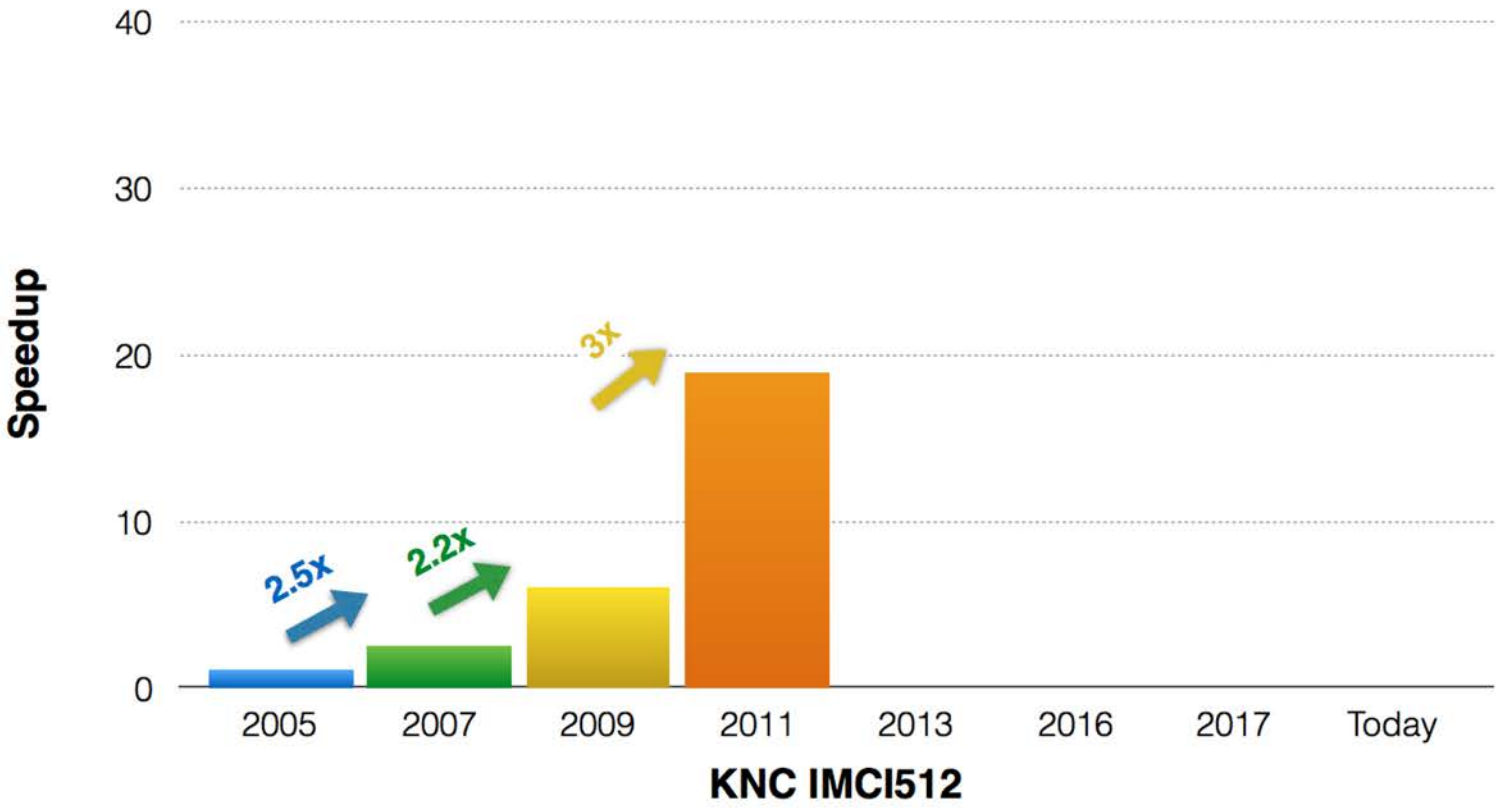
Performance evolution of dense Cholesky



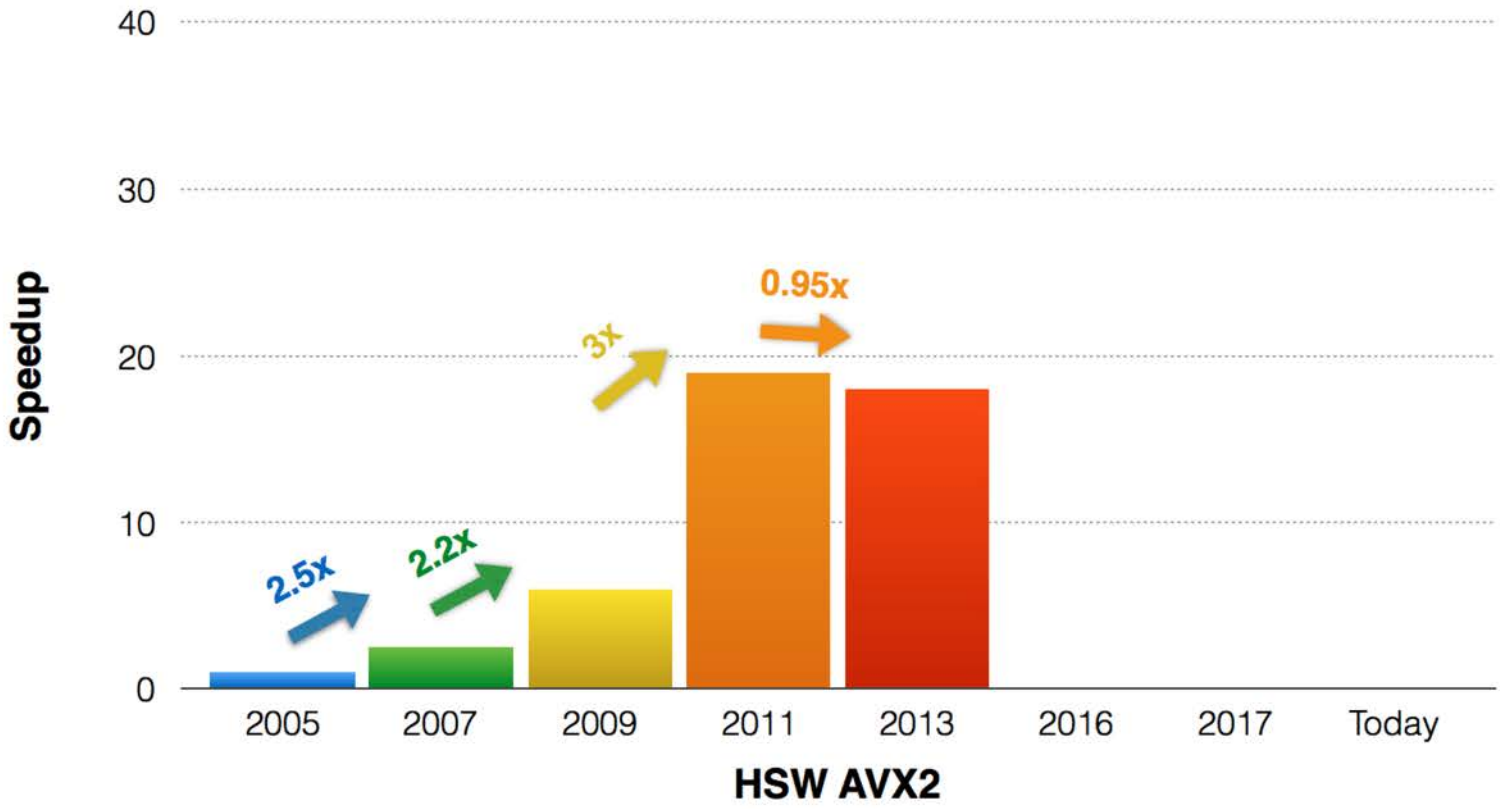
Performance evolution of dense Cholesky



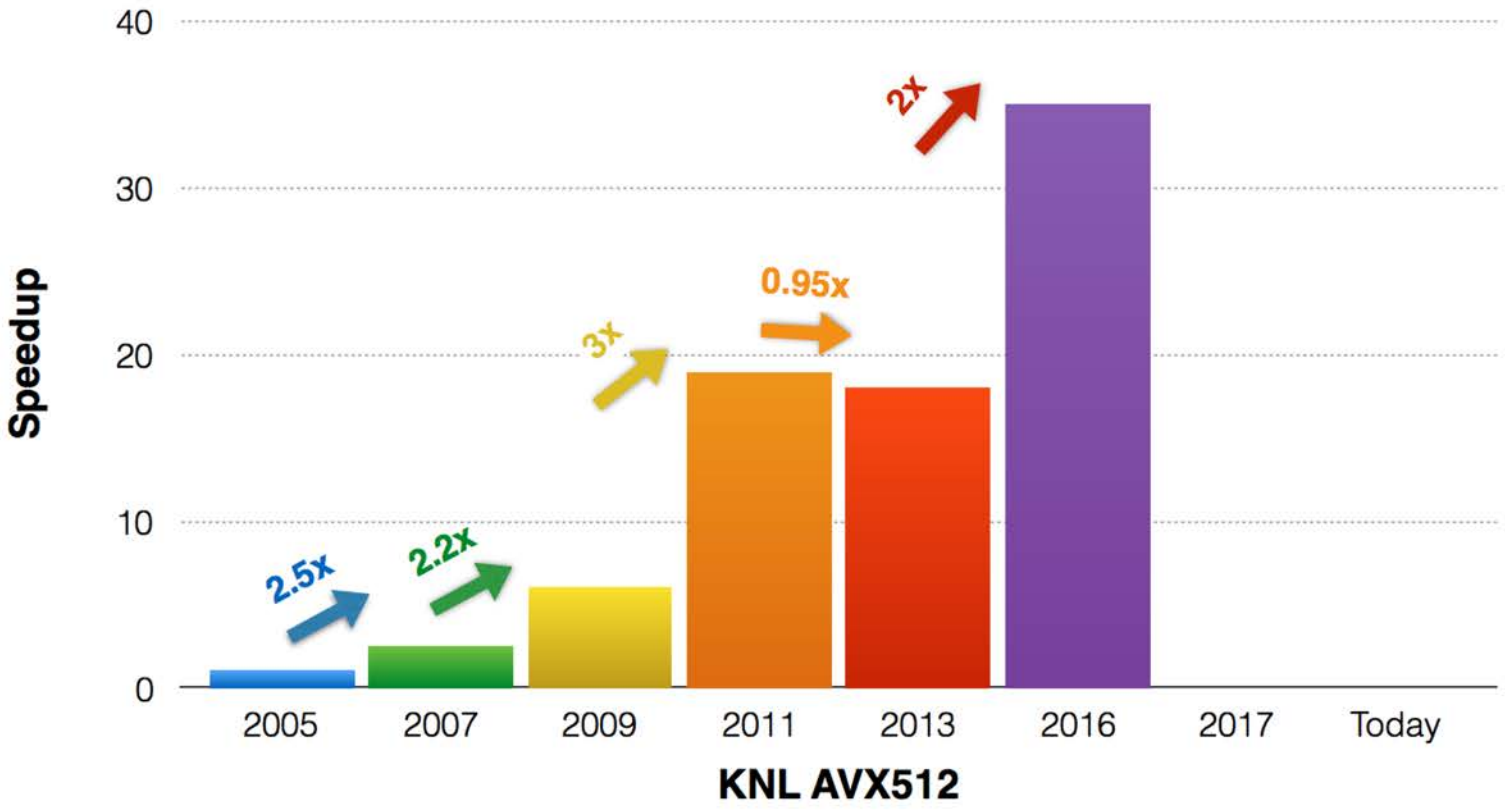
Performance evolution of dense Cholesky



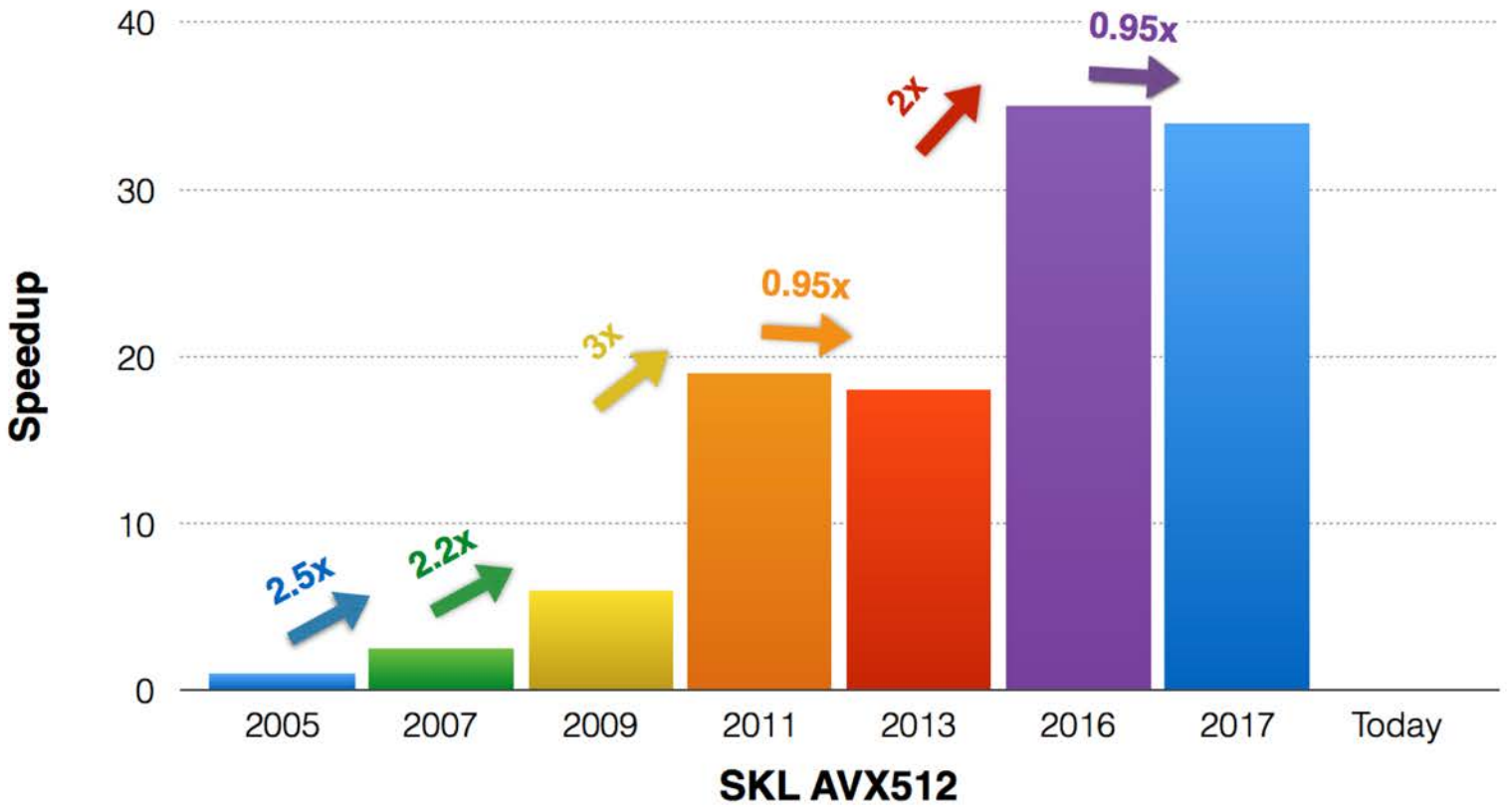
Performance evolution of dense Cholesky



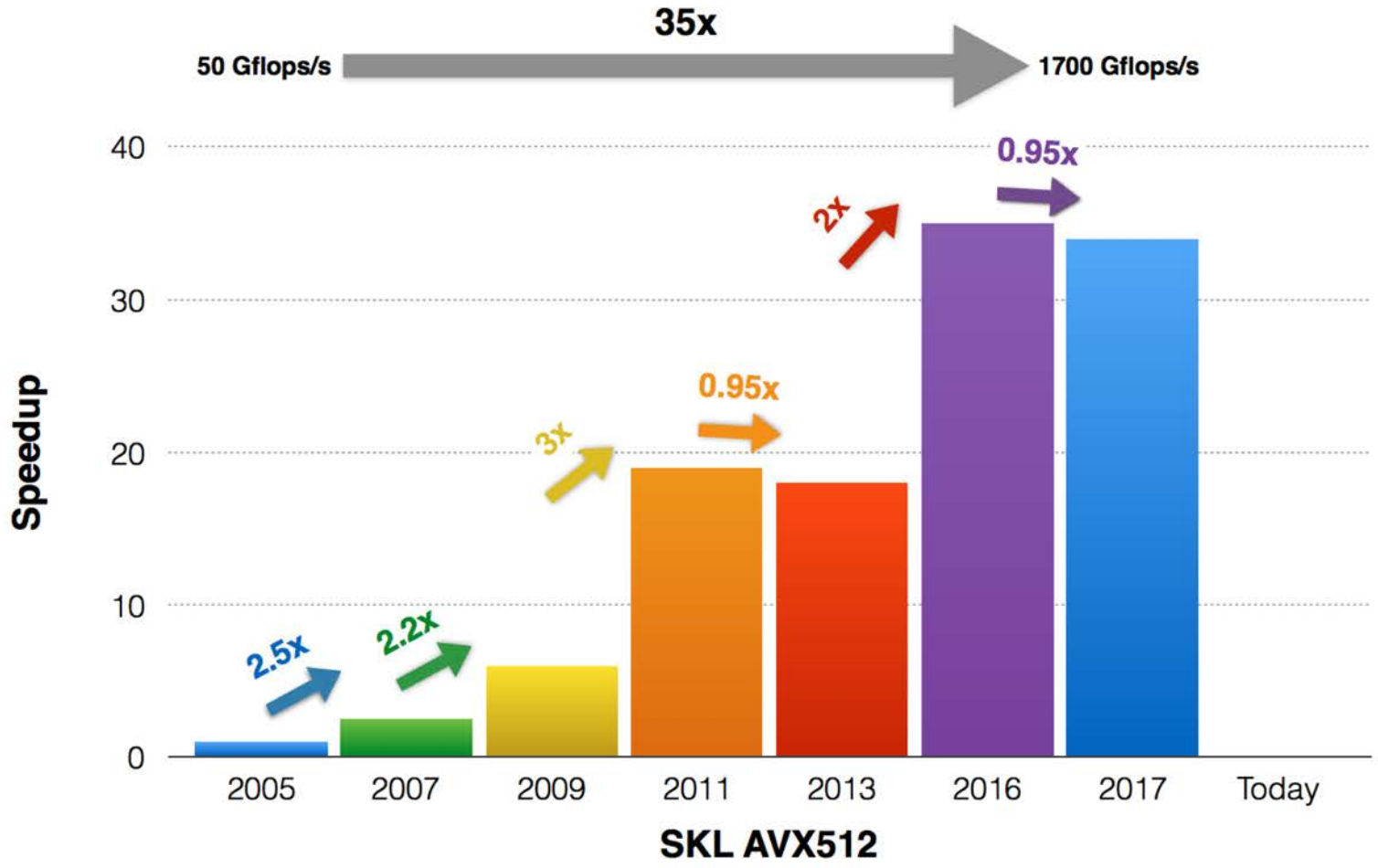
Performance evolution of dense Cholesky



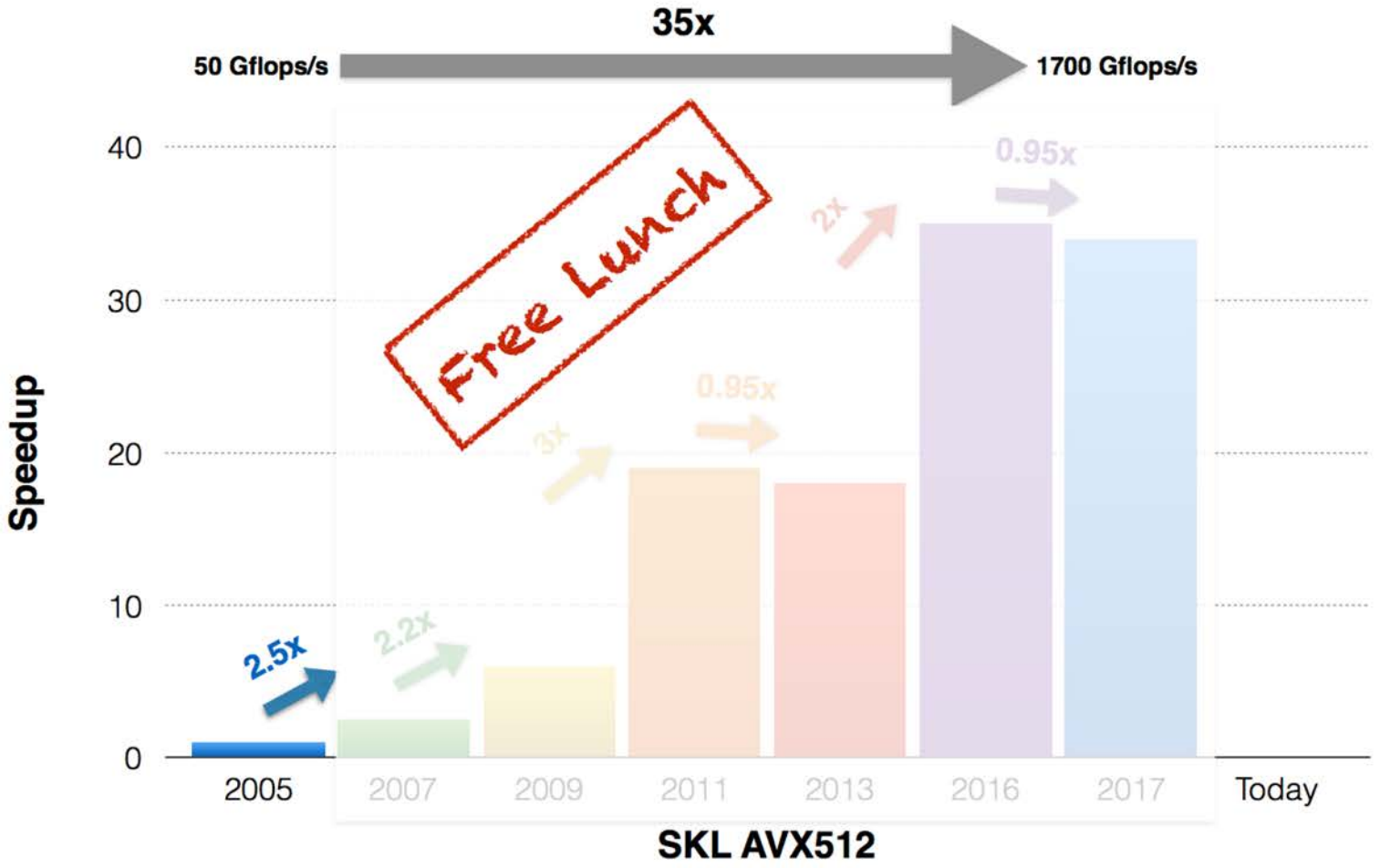
Performance evolution of dense Cholesky



Performance evolution of dense Cholesky

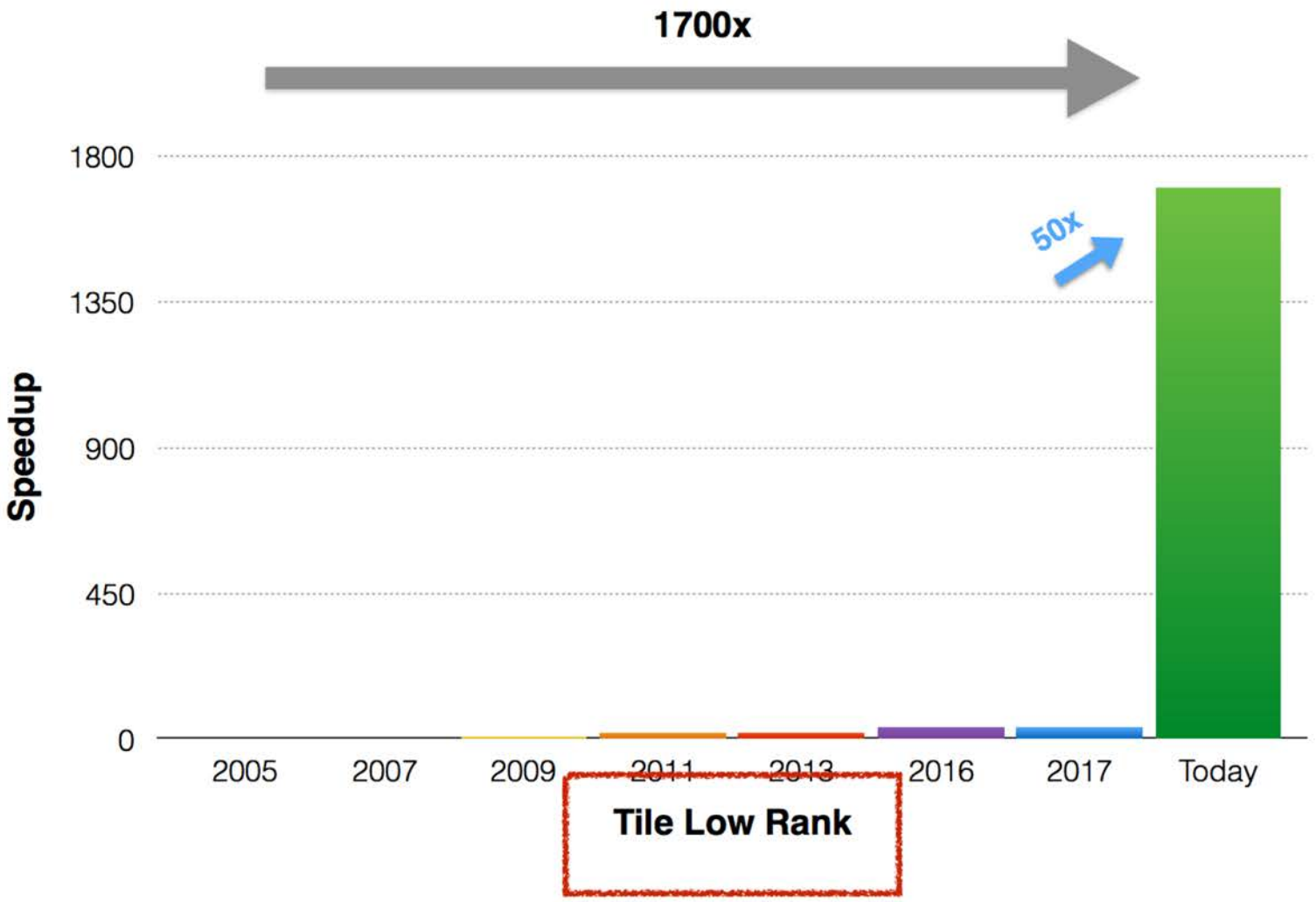


Performance evolution of dense Cholesky



(first factor based on tiling;
successive factors, 2007-2017, based on Top500 hardware generations)

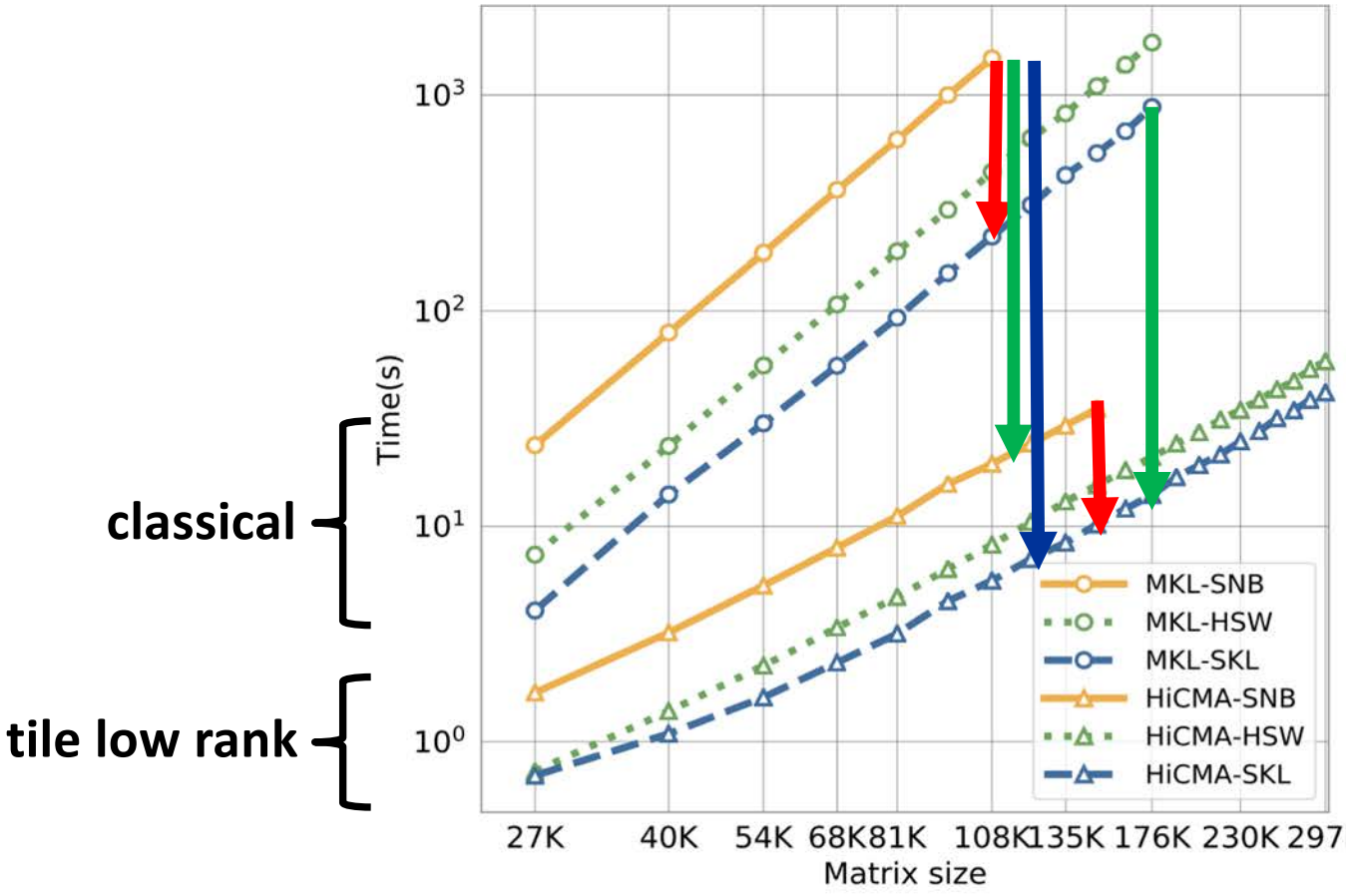
Performance evolution of dense Cholesky



HiCMA vs. Intel MKL on shared memory

Geospatial statistics (Gaussian kernel) to accuracy 1.0e-8

- Three generations of Intel manycore (Sandy Bridge, Haswell, Skylake)
- Two generations of linear algebra (classical dense and tile low rank)



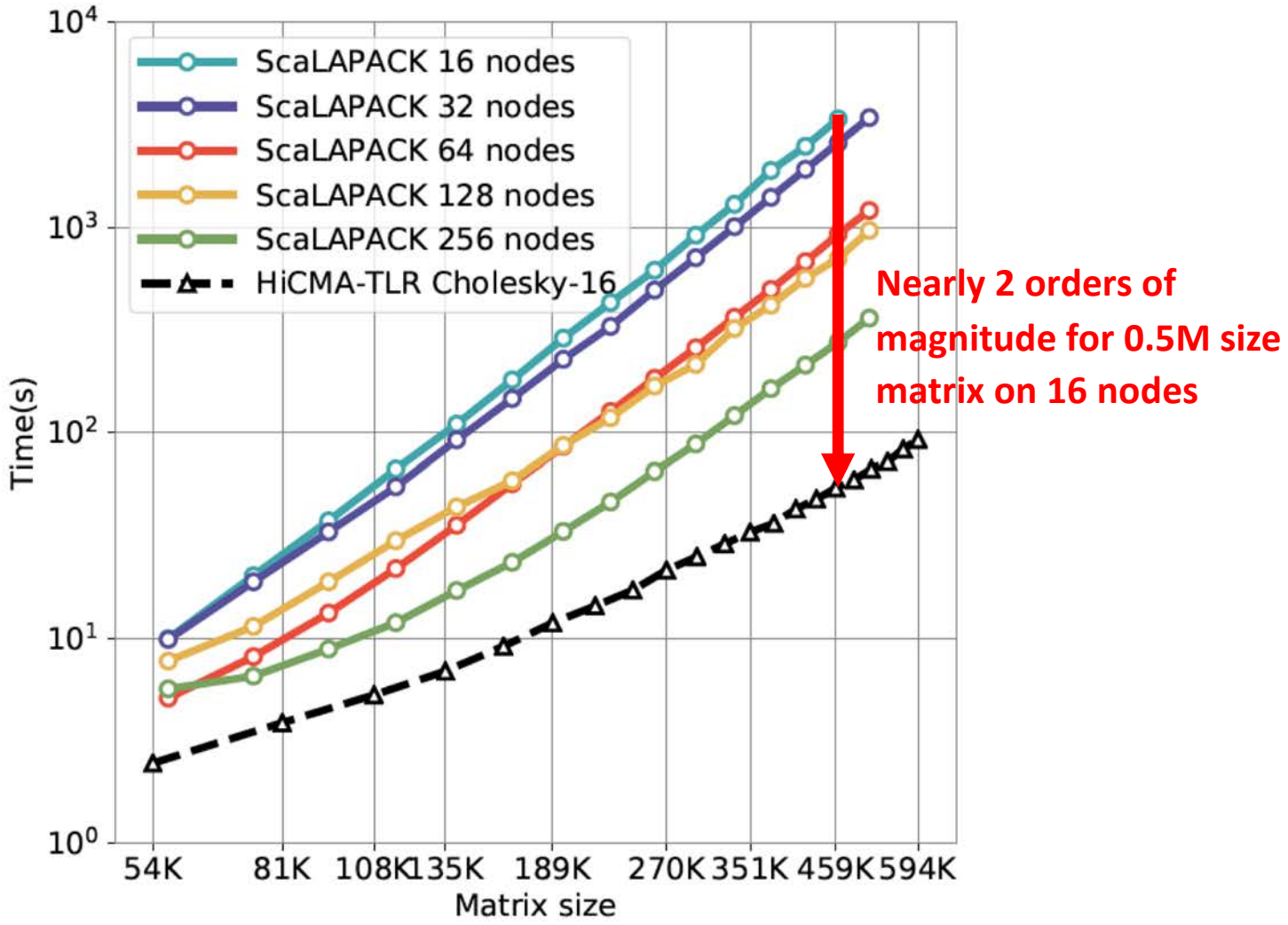
Red arrows: speedups from hardware, same algorithm

Green arrows: speedups from algorithm, same hardware

Blue arrow: From both

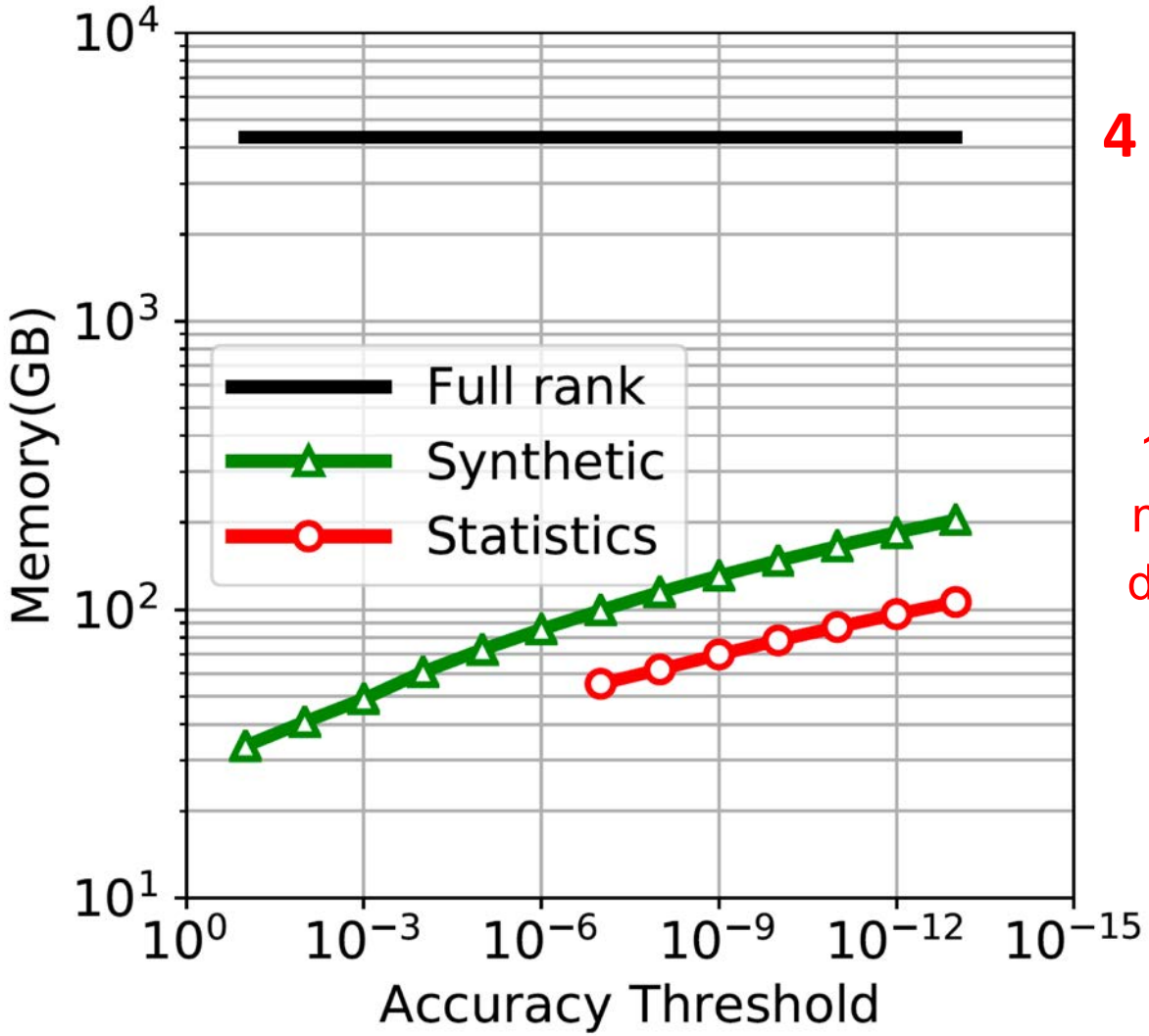
Akbudak, Ltaief, Mikhalev, Charara, & Keyes, Exploiting Data Sparsity for Large-scale Matrix Computations, Europar 2018.

HiCMA vs. ScaLAPACK on distributed memory



K. Akbudak, H. Ltaief, A. Mikhalev, A. Charara, and D. E. Keyes, Exploiting Data Sparsity for Large-Scale Matrix Computations, EuroPar 2018

Memory footprint for DP matrix of size 1M



4 TB

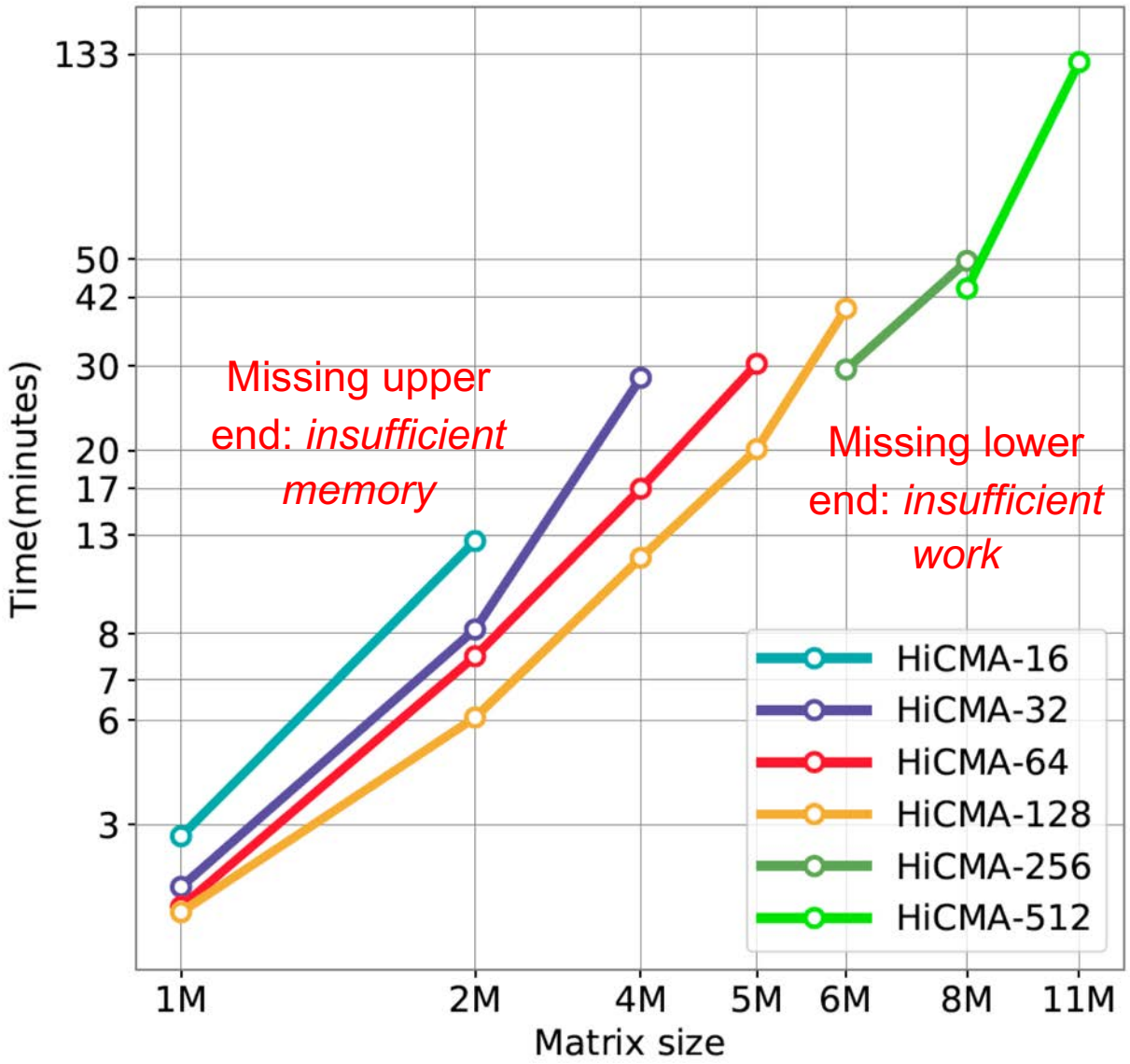
1 to 2 orders of magnitude less, depending upon accuracy

K. Akbudak, H. Ltaief, A. Mikhalev, A. Charara, and D. E. Keyes, Exploiting Data Sparsity for Large-Scale Matrix Computations, EuroPar 2018

HiCMA on distributed memory

Geospatial statistics (Gaussian kernel) to accuracy 1.0e-8

- Cray XC40 (“Shaheen”, 32 Haswell cores per node)
- Range of problem sizes and core count



Akbudak, Ltaief, Mikhalev, Charara, & Keyes, Exploiting Data Sparsity for Large-scale Matrix Computations, Europar 2018.

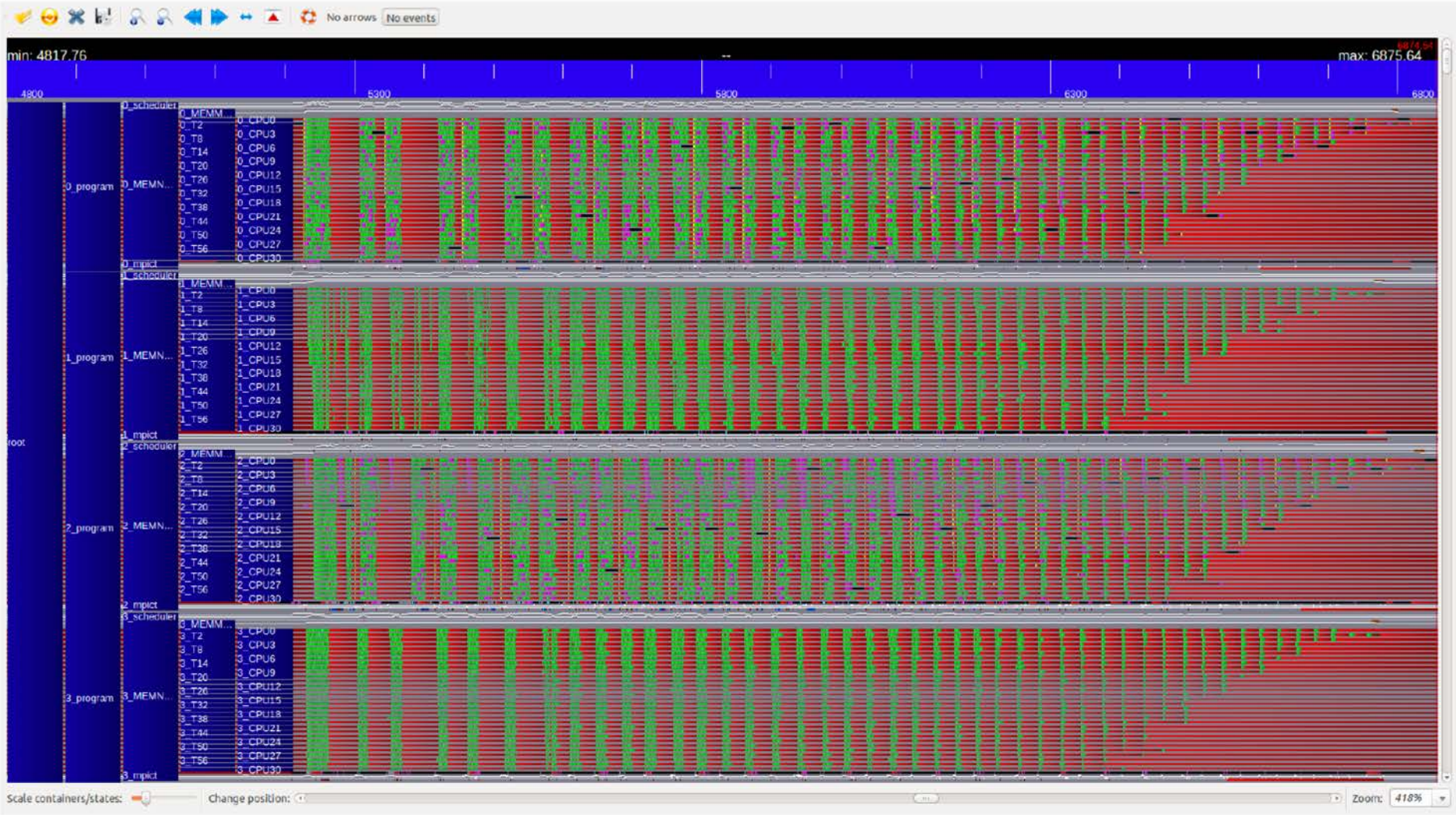
Execution traces

Chameleon: Dense DPOTRF time **18.1s**
4 nodes of Shaheen with a matrix size of 54K

Akbudak, Ltaief, Mikhalev, Charara, & Keyes, Exploiting Data Sparsity for Large-scale Matrix Computations, Europar 2018.

Execution traces

HiCMA: TLR DPOTRF time **1.8s** (10X faster)
4 nodes of Shaheen with a matrix size of 54K



Akbudak, Ltaief, Mikhalev, Charara, & Keyes, Exploiting Data Sparsity for Large-scale Matrix Computations, Europar 2018.

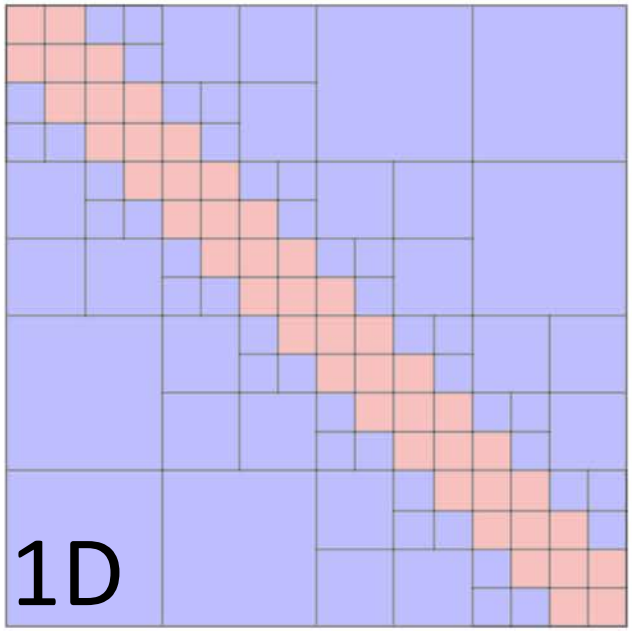
So far, just Tile Low Rank…

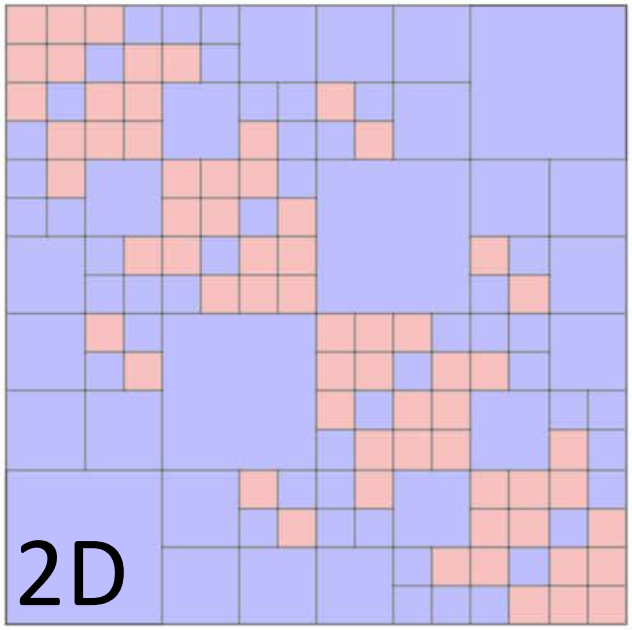
Now: general $\mathcal{H}$-matrices

- **First advance: adopt $\mathcal{H}^2$ matrix structure**
- **Second advance: use randomized SVD (Halko, Martinsson & Tropp, 2009) to form the low-rank blocks at the leaves**
  - **an easy, flop-intensive GEMM-based flat algorithm**
- **Third advance: implement using "batches" on GPUs/multi-GPUs**

$\mathcal{H}^2$ hierarchical matrix representation

- general blocking structure

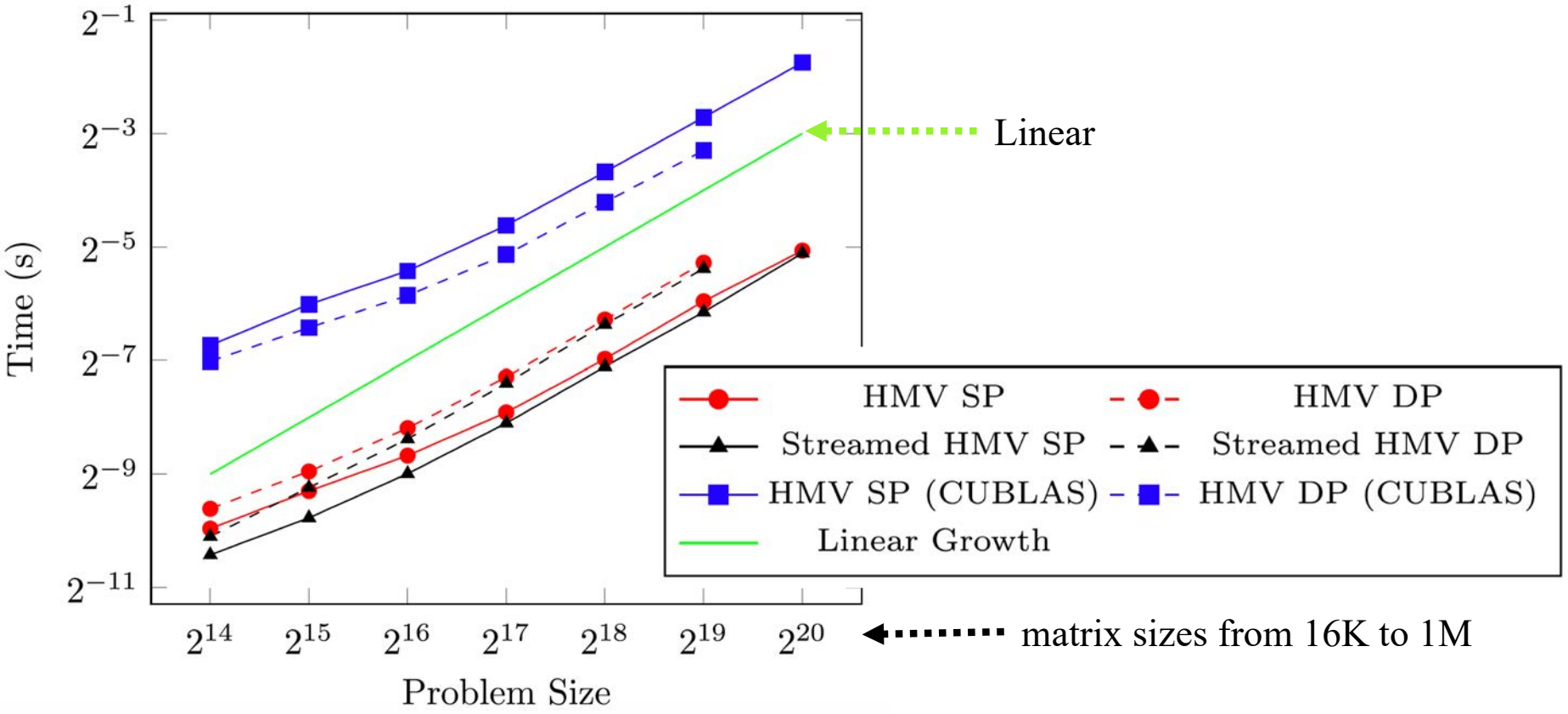




- nested bases
  - $A_{ij}^l = U_i^l \, S_{ij}^l \, {V_j^l}^T$
  - $U_i^{l-1} = \begin{bmatrix} U_{i_1}^l & \\ & U_{i_2}^l \end{bmatrix} \begin{bmatrix} E_{i_1}^l \\ E_{i_2}^l \end{bmatrix}$
  - reduces memory footprint to $O(n)$ from $O(n \log n)$

Boukaram, Turkiyyah & Keyes, Hierarchical Matrix Operations on GPUs: Matrix-Vector Multiplication and Compression, ACM TOMS 45:3 (2019)

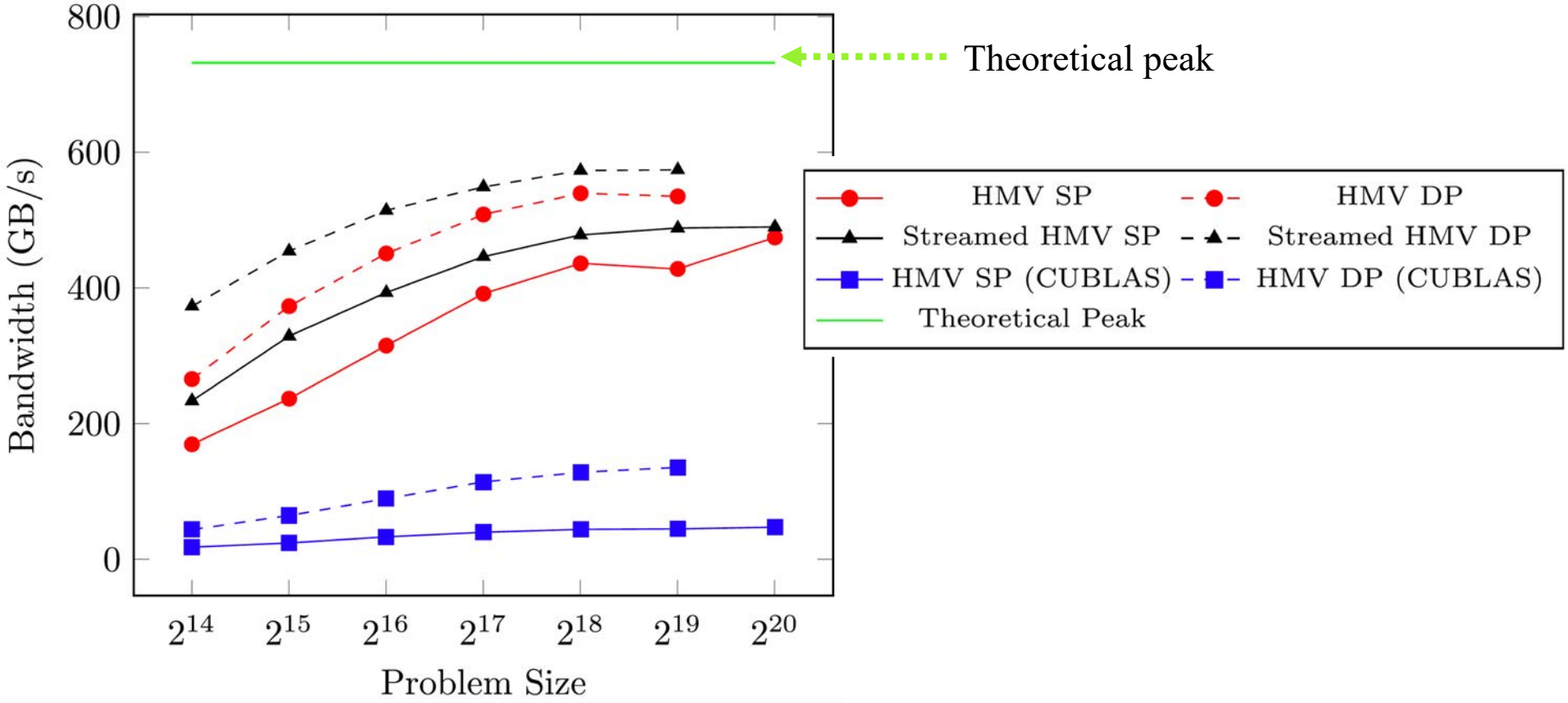
Hierarchical MatVec execution time



- 3D covariance matrices from spatial statistics
- running on P100 GPU
- accuracy 10^{-3} computed as $\|Ax - A^{\mathcal{H}}x\|/\|Ax\|$
- leaf size $m = 64$

Boukaram, Turkiyyah & Keyes, Hierarchical Matrix Operations on GPUs: Matrix-Vector Multiplication and Compression, ACM TOMS 45:3 (2019)

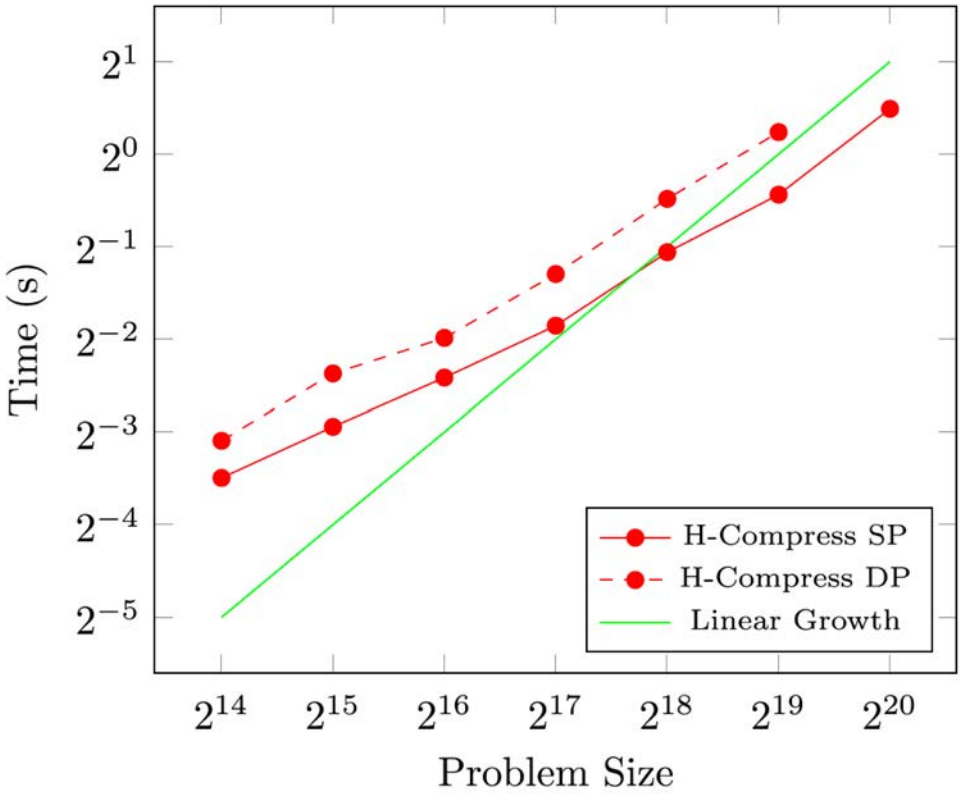
Hierarchical MatVec bandwidth

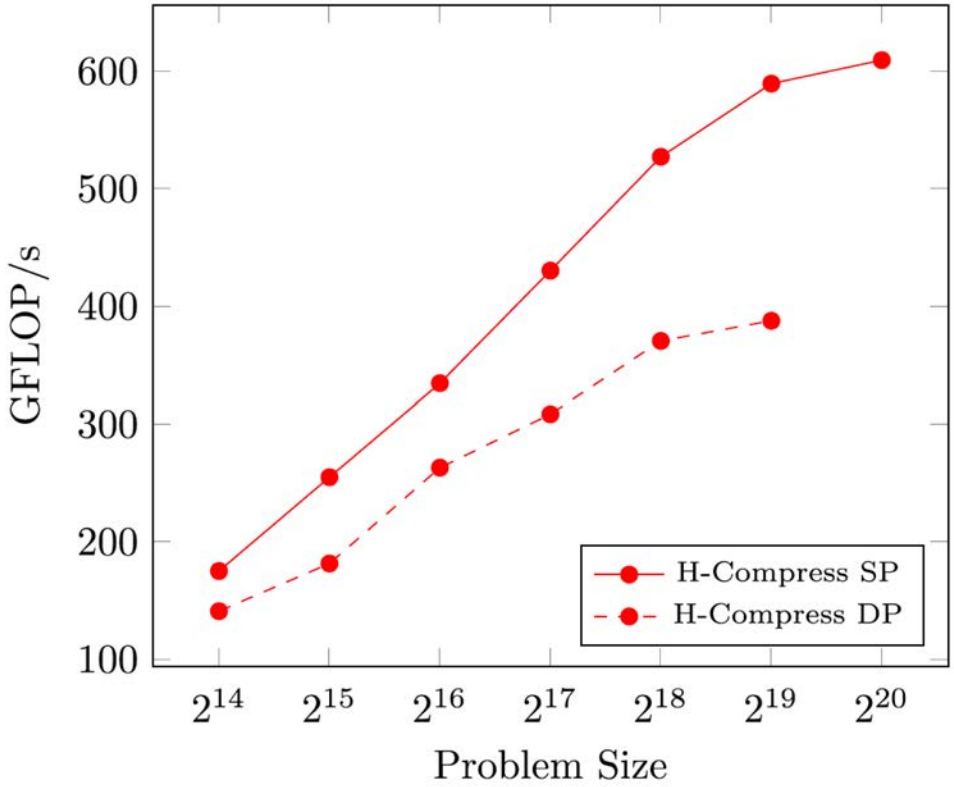


- 3D covariance matrices from spatial statistics
- running on P100 GPU
- accuracy 10^{-3} computed as $\|Ax - A^{\mathcal{H}}x\|/\|Ax\|$
- leaf size $m = 64$

Boukaram, Turkiyyah & Keyes, Hierarchical Matrix Operations on GPUs: Matrix-Vector Multiplication and Compression, ACM TOMS 45:3 (2019)

Hierarchical compression





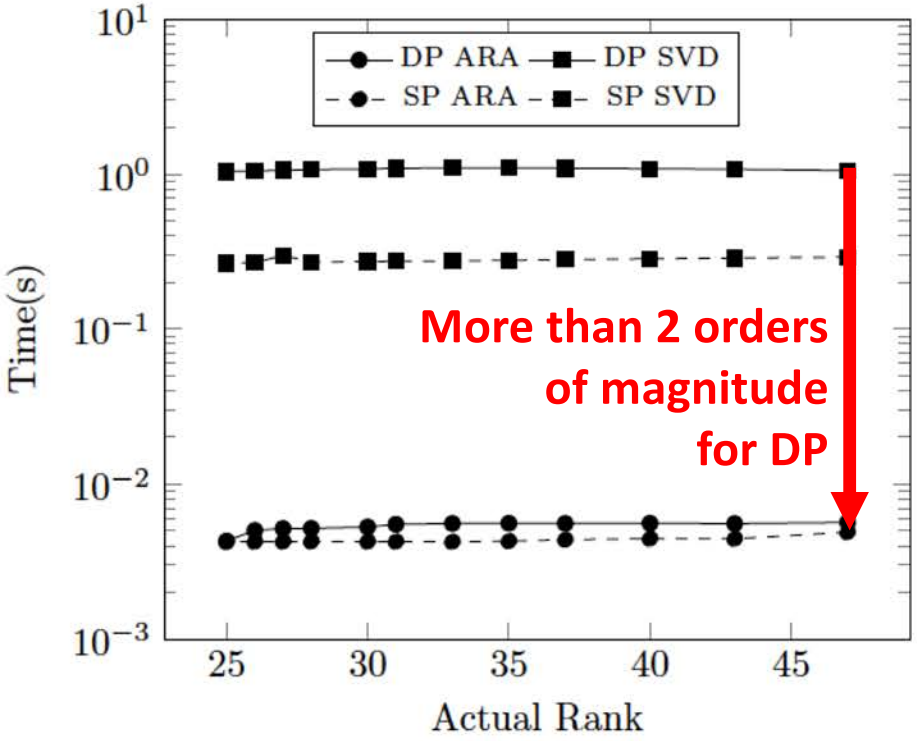
- 3D covariance matrices from spatial statistics
- running on P100 GPU
- accuracy 10^{-3} computed as $\left\|Ax - A^{\mathcal{H}}x\right\| / \|Ax\|$
- leaf size $m = 64$

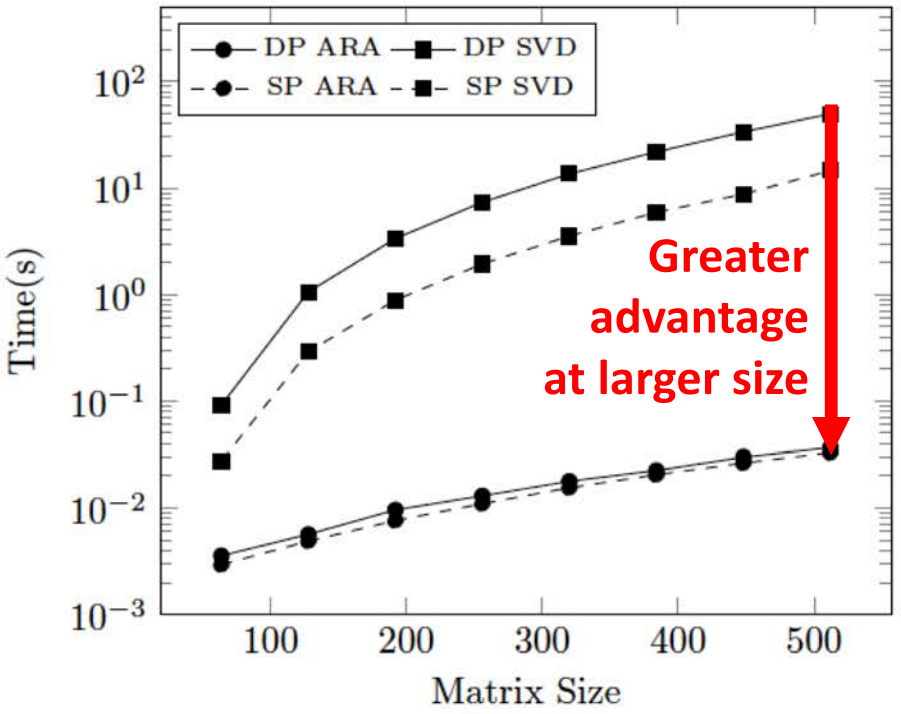
Boukaram, Turkiyyah & Keyes, Hierarchical Matrix Operations on GPUs: Matrix-Vector Multiplication and Compression, ACM TOMS 45:3 (2019)

Adaptive Randomized Approximations (ARA)

- allow fast construction of low rank approximations
- rely on sampling the matrix through mat-vec operations
  - can be done on multiple vectors simultaneously
  - for increased arithmetic intensity
- applicable to dense matrices and, with hierarchical extensions, to $\mathcal{H}$ matrices
- particularly effective on GPUs
  - can leverage high-performing GEMM routines

Comparison with Jacobi (Givens) SVD

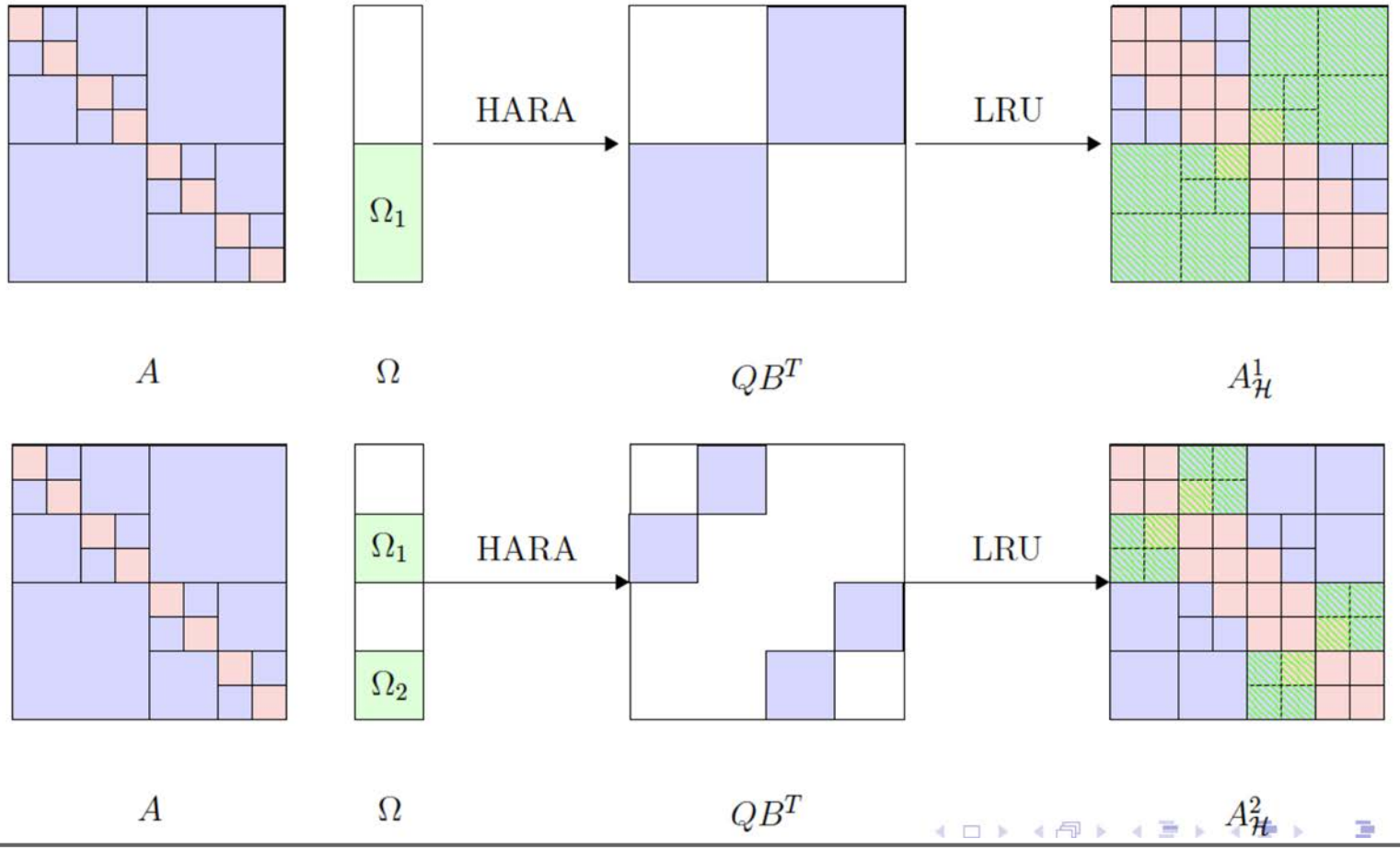




- batch of 1000 matrices in single and double precision
- varying rank for fixed size (128)
- varying size for fixed rank (47)

Hierarchical Adaptive Randomized Approximation (HARA)

- extends the basic idea to hierarchical matrices
- samples blocks of the matrix and accumulates the local low rank updates into an $\mathcal{H}$-matrix that is recompressed

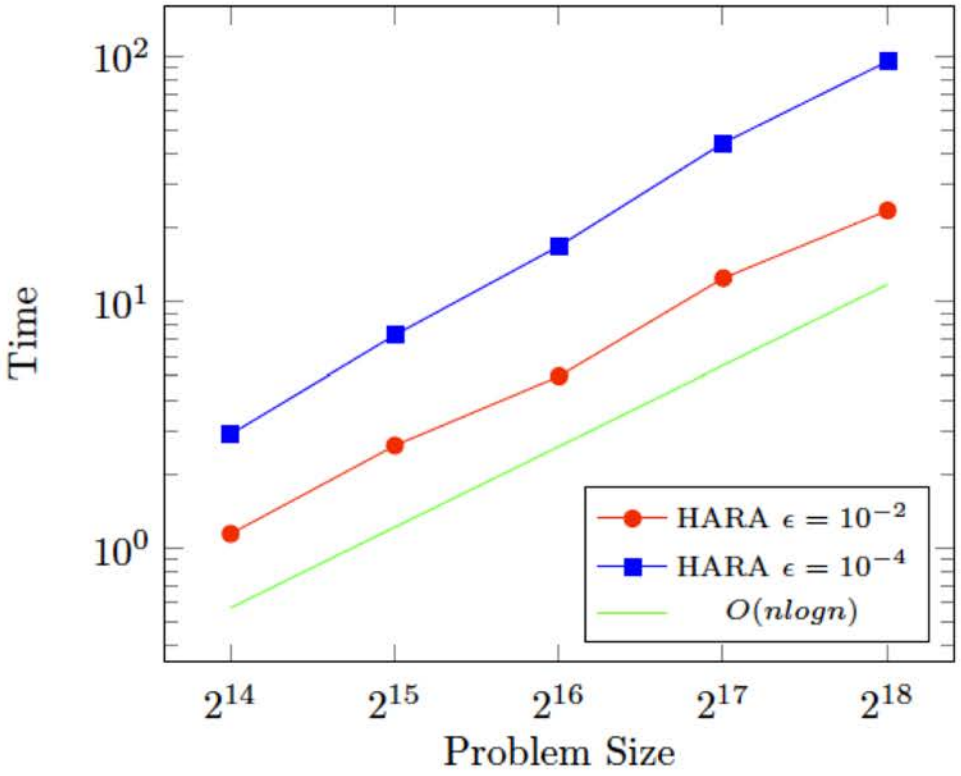


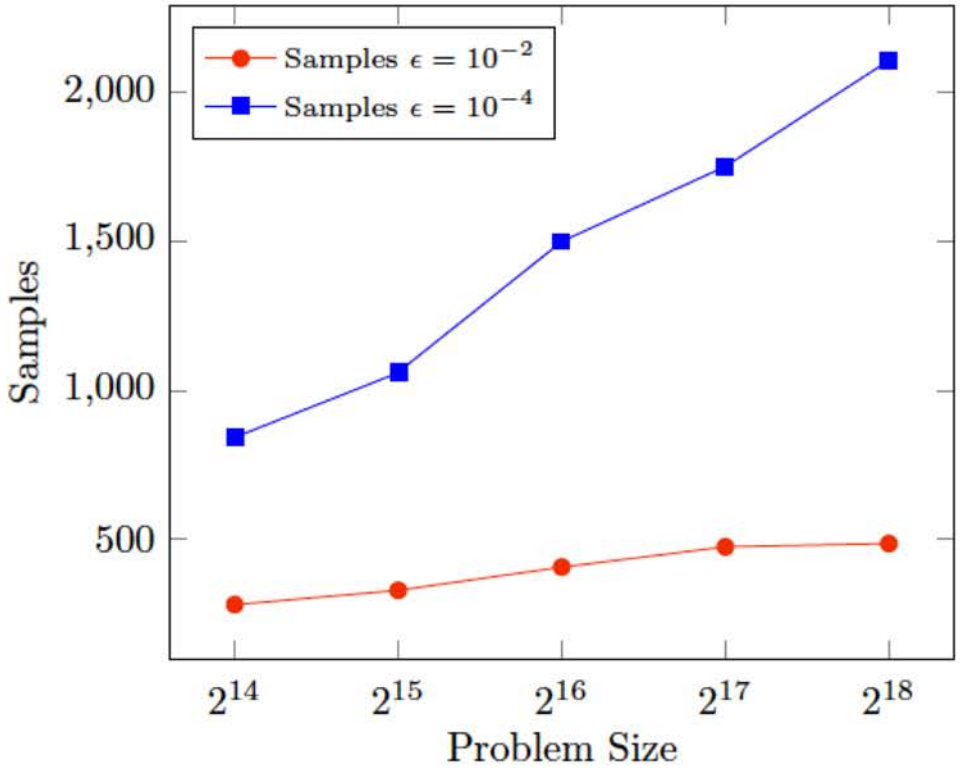
Boukaram, Turkiyyah & Keyes, Randomized GPU Algorithms for the Construction of Hierarchical Matrices from MatVec Operations, submitted to SISC (2019)

Performance of HARA on GPU

Matrix sizes from 16K to 256K; accuracies 10^{-2} to 10^{-4}

Time and no. samples to compress





- spatial covariance matrix reconstructed from HGEMV products

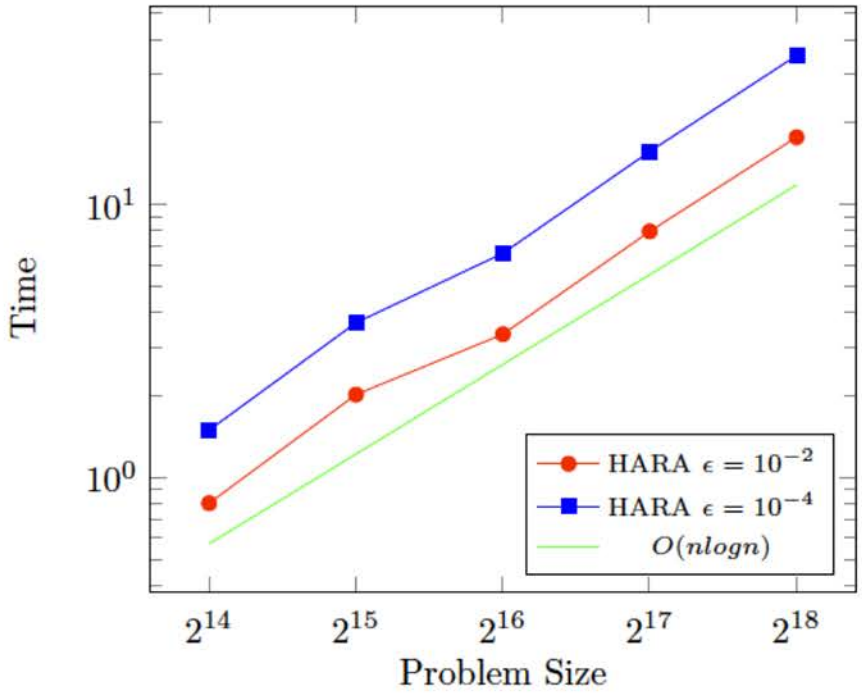
---

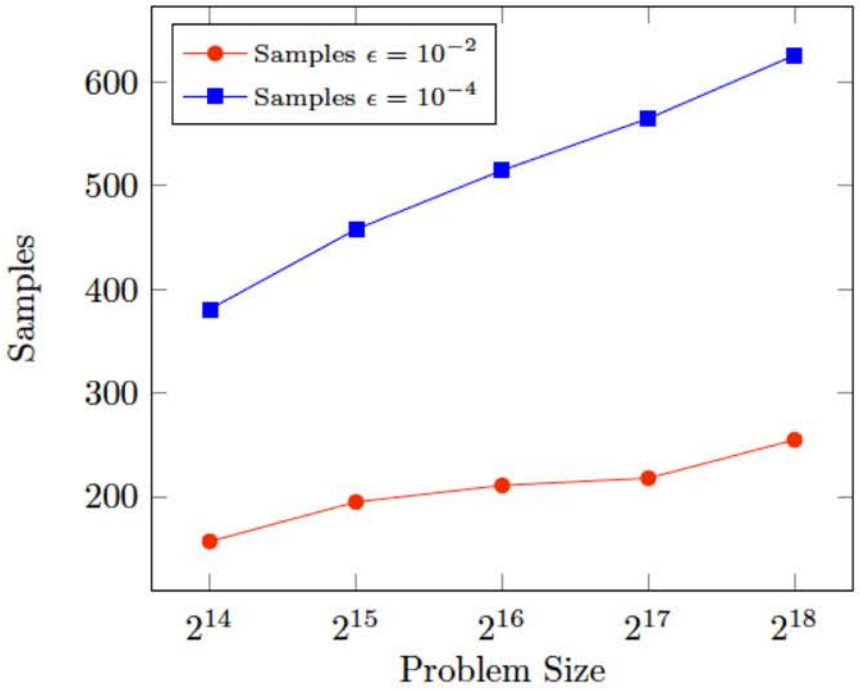
Boukaram, Turkiyyah & Keyes, Randomized GPU Algorithms for the Construction of Hierarchical Matrices from MatVec Operations, submitted to SISC (2019)

$\mathcal{H}$ matrix-$\mathcal{H}$ matrix multiplication

- can be cast as the problem of constructing an $\mathcal{H}$-matrix from matvec operations
- we can do HGEMV operations efficiently on GPUs
  - HGEMV on multiple vectors is even more efficient
- HARA construction of product also performed efficiently on the GPU

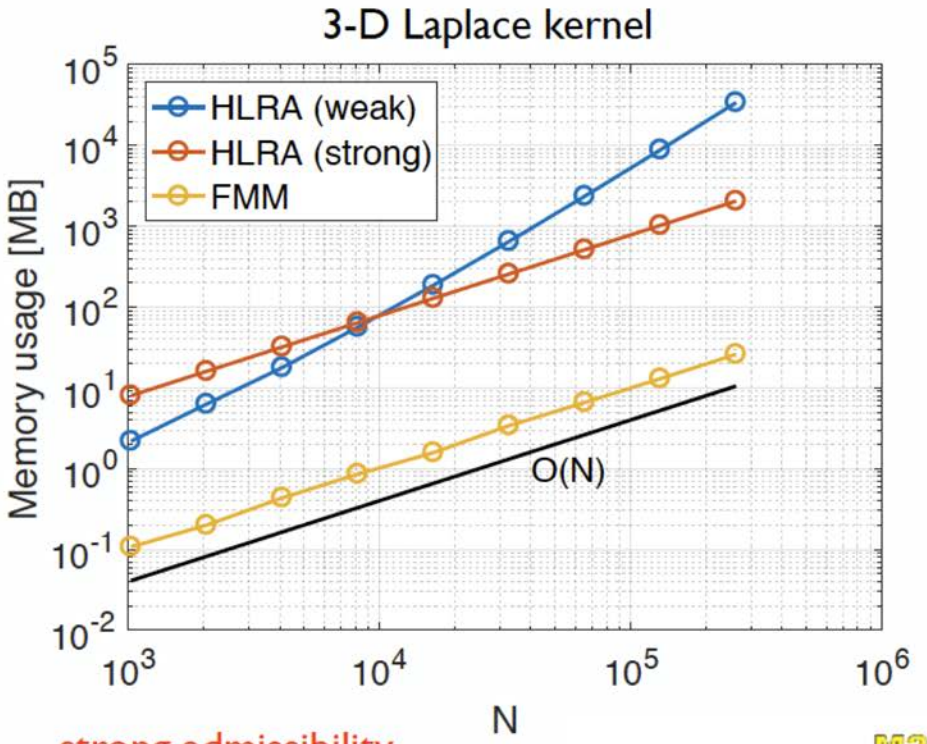
Fast matvecs ⇒ fast approx inversions with Newton-Schulz

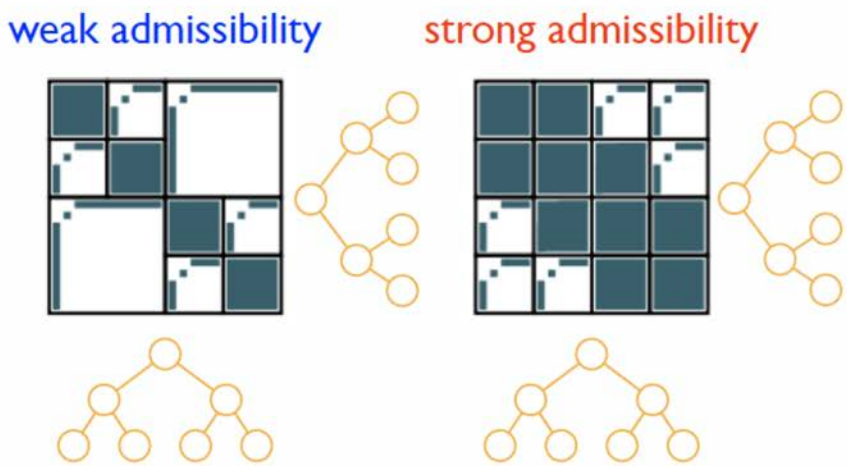


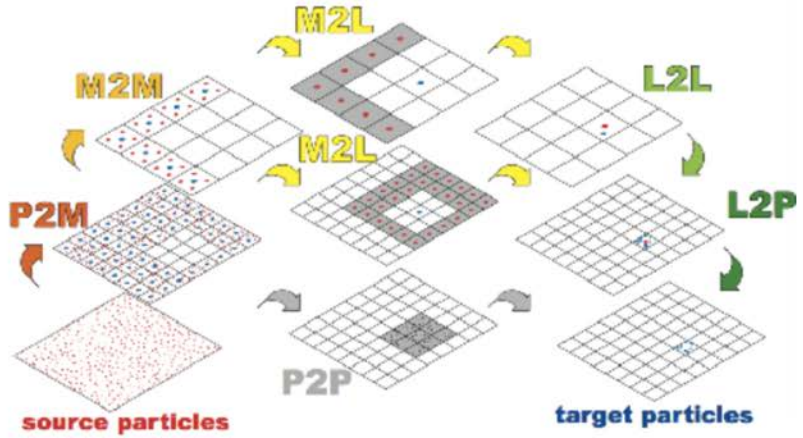


Boukaram, Turkiyyah & Keyes, Randomized GPU Algorithms for the Construction of Hierarchical Matrices from MatVec Operations, submitted to SISC (2019)

Memory complexity of FMM vs $\mathcal{H}$ (HLRA)





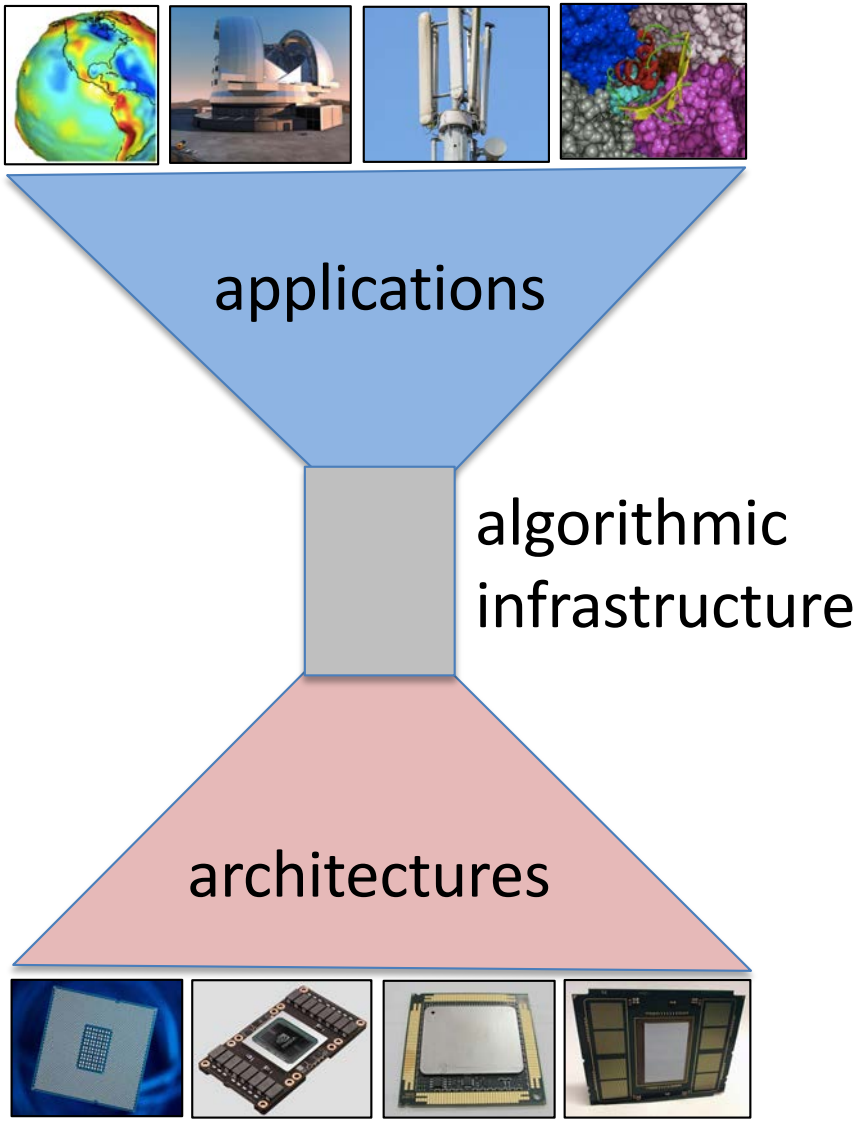


c/o Rio Yokota (Tokyo Tech/KAUST)

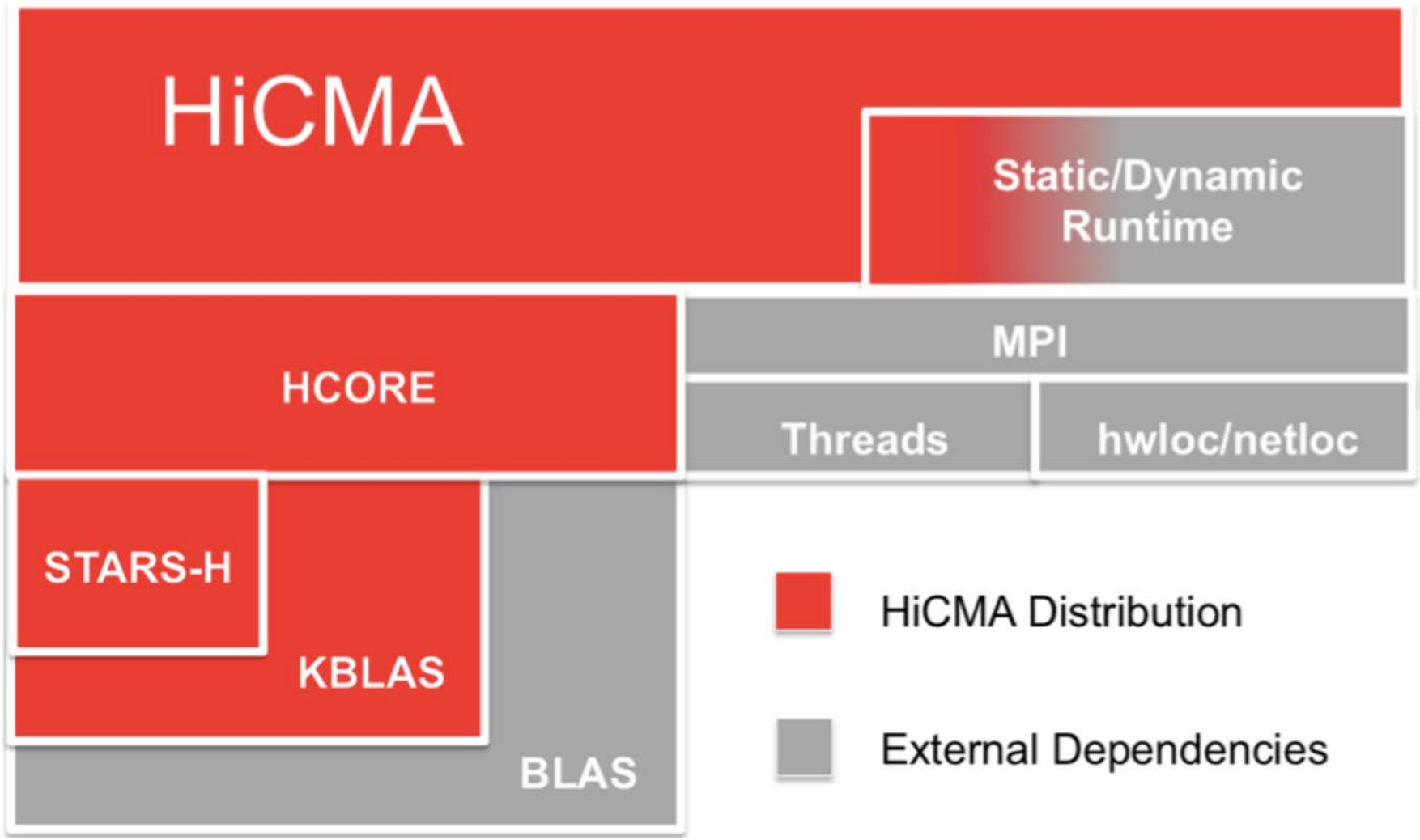
Conclusions

- **Plenty of ideas exist to adapt or substitute for favorite solvers with methods that have:**
  - **reduced synchrony (in frequency and/or span)**
  - **higher residence on the memory hierarchy**
  - **greater SIMT/SIMD-style shared-memory concurrency**
- **Programming models and runtimes may have to be stretched to accommodate**
- **Everything should be on the table for trades, beyond disciplinary thresholds ➔ "co-design"**

“Hourglass” model for algorithms (borrowed from internet protocols)



Hierarchical Computations on Manycore Architectures: HiCMA*



* appearing incrementally at https://github.com/ecrc

Acknowledgments

This talk was made possible through baseline KAUST support for ECRC research scientists and our vendor-sponsored efforts on hosting FMM and $\mathcal{H}$-matrix methods on MIC and GPU architectures


Thank you!
جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology
شكرا
david.keyes@kaust.edu.sa