Factorization of Hierarchical Low-rank Matrices with Nested Basis

Qianxiang Ma

Rio Yokota Lab

School of Computing

Tokyo Institute of Technology

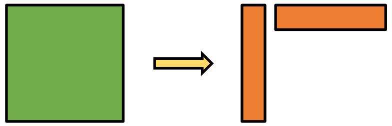
Presentation Outline

- Problem Review
  - Dense solvers and LU factorization
  - Hierarchical Low-rank Matrices
- Runtime system for batched H-LU factorization on GPU and its results
- Nested basis and H^2-Matrices and its results
- Conclusion

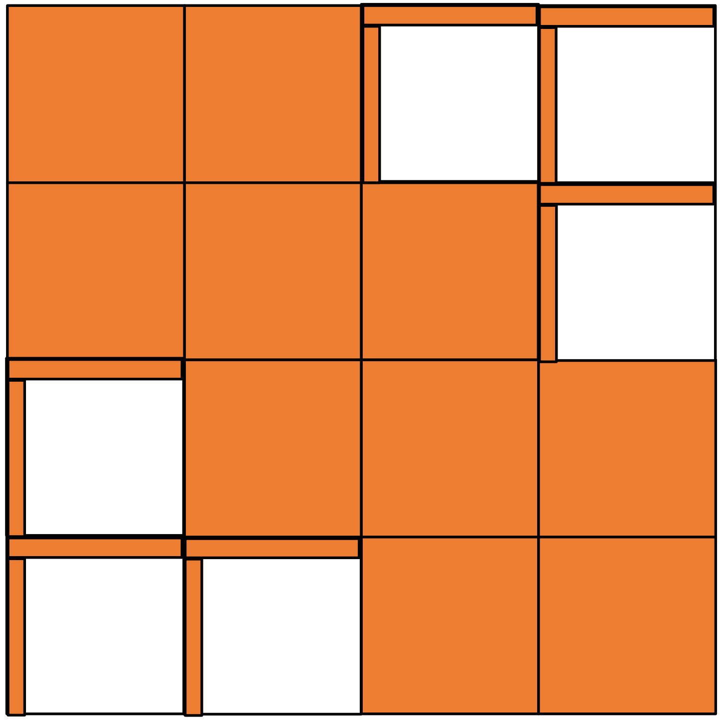
Problem Review

Block Low Rank Matrices

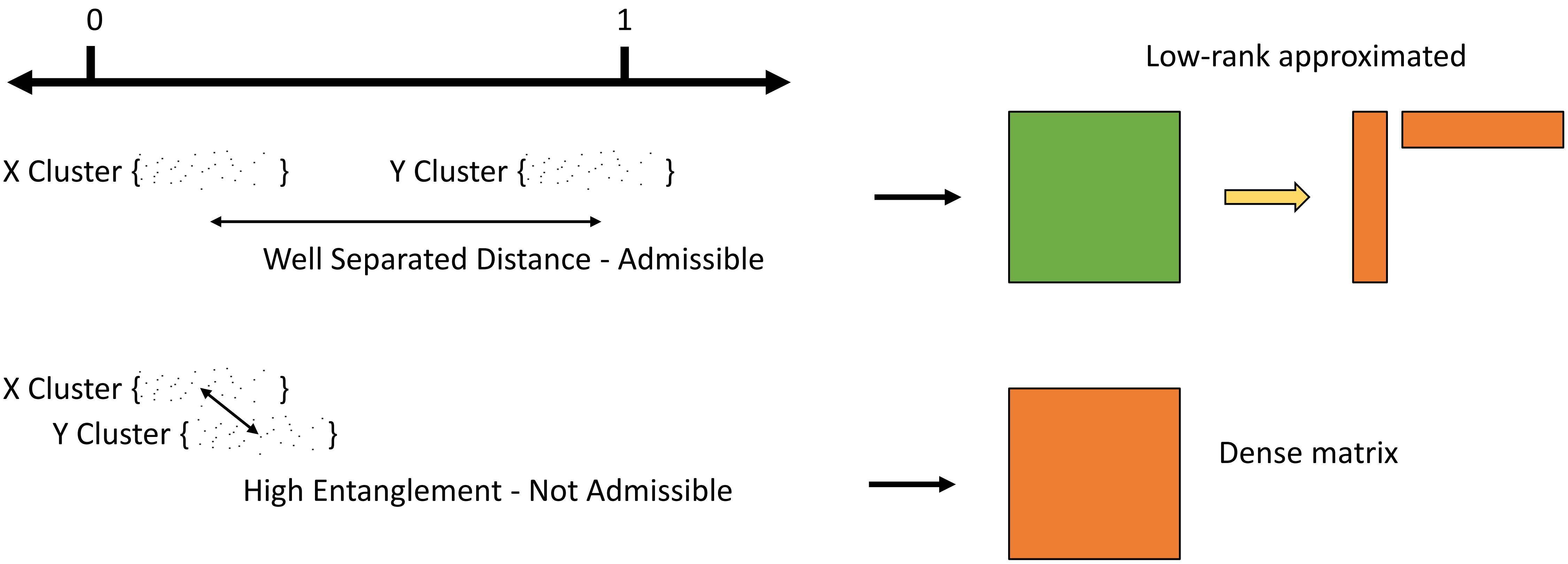
- Low-rank approximation:
  - Represent blocks as products of matrices of smaller ranks

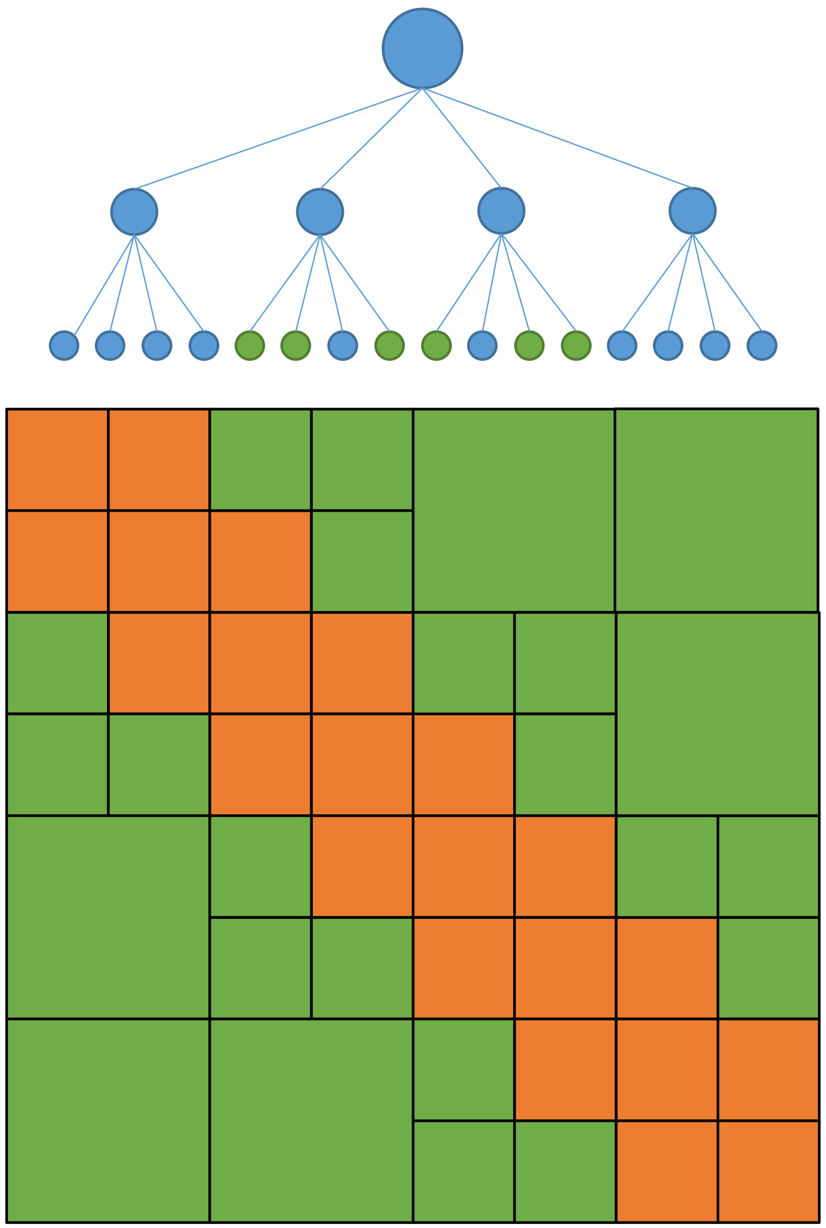


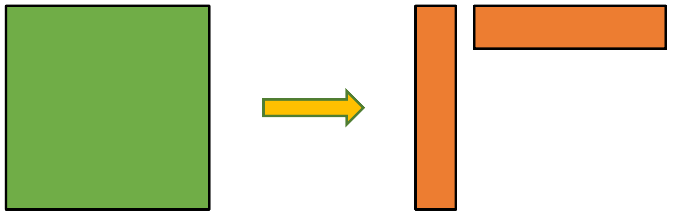
- Kernel matrices:
  - Full in entries / not sparse
  - Higher ranks closer to the diagonal
  - Blocks further away have lower ranks
  - Green's function, radial basis functions etc.



Admissibility Condition







Low-rank blocks in Hierarchical Matrix:

$$A = U \times S \times V^T$$

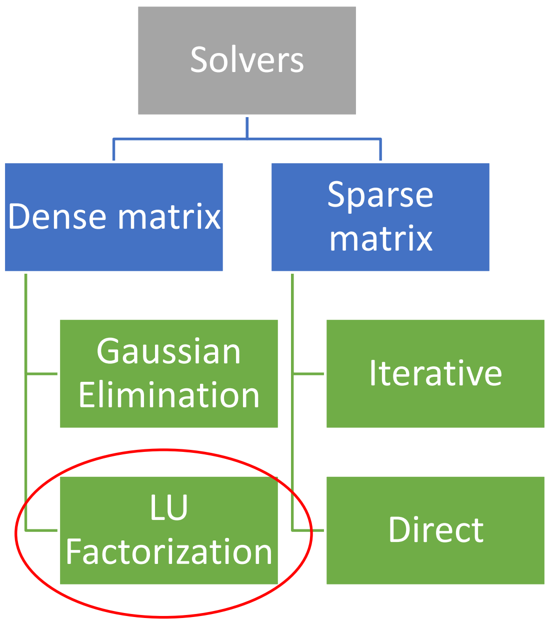
or:

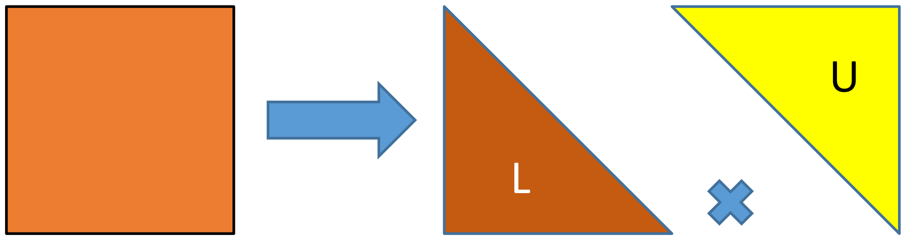
$$A = U \times V^T$$

Solving Linear Systems - LU factorization

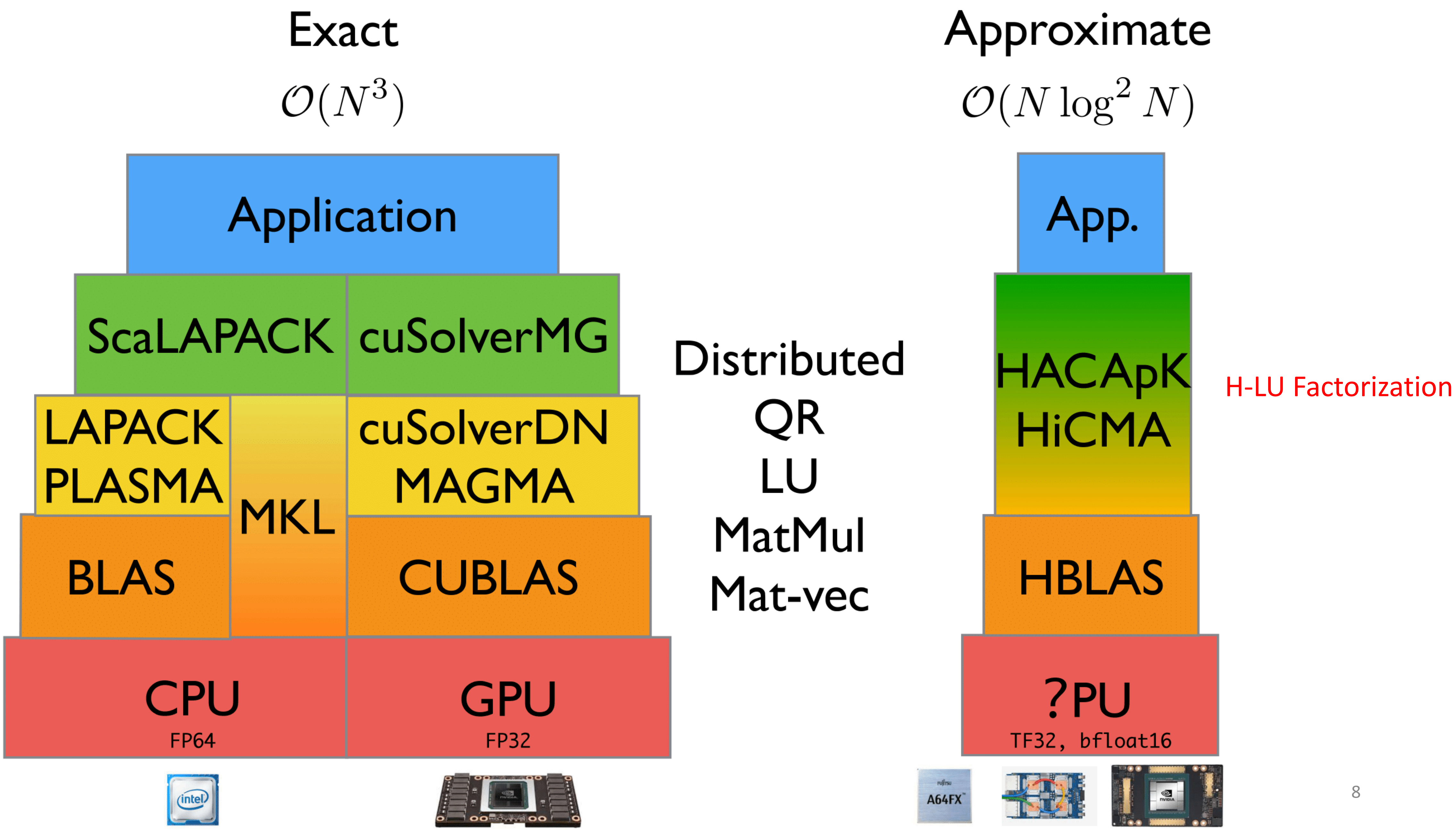
- Expressed as: $Ax = b$
- $A = LU$: solve $Ly = b$ then $Ux = y$

- Direct / Dense solver
- Compares to:
  - Inexact iterative solvers
  - Sparse solvers



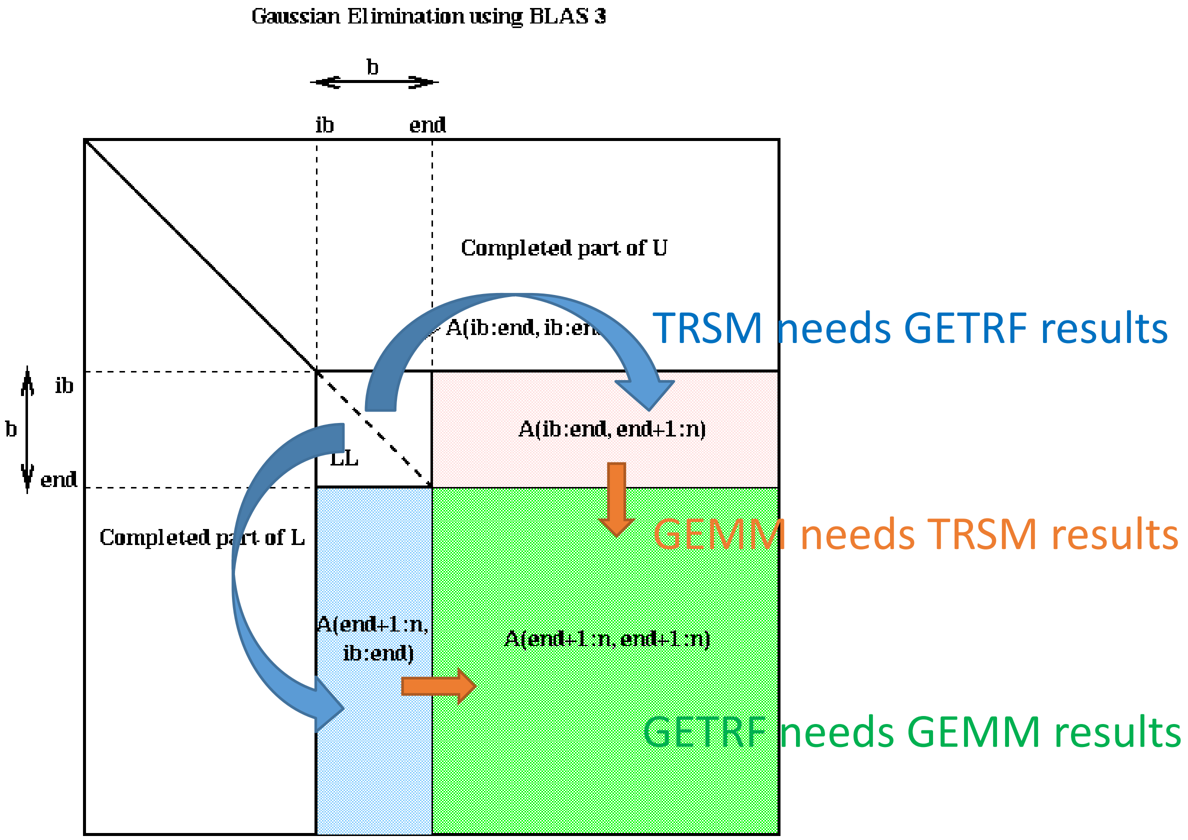


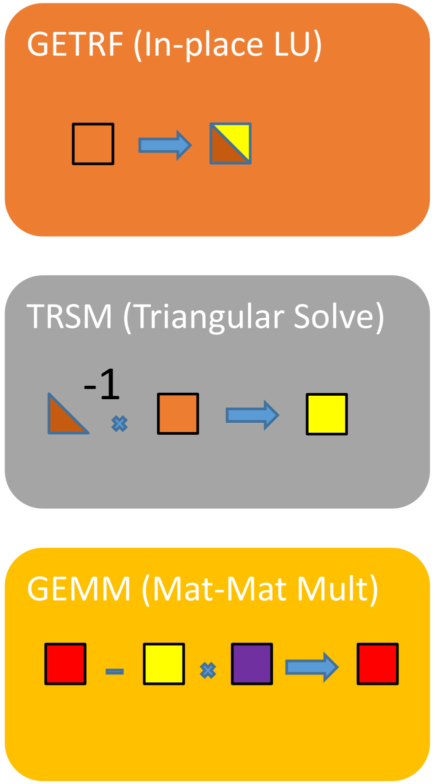
Replacing Exact Linear Algebra with Low-Rank



Related Work

- James Demmel et al., Block LU Factorization. 1995
  - Dense exact solver
  - $O(n^3)$ complexity
- Ronald Kriemann, H-LU factorization on many-core systems. 2013
  - H-solver based on parallelized Hierarchical LU factorization
  - CPU-only execution
- Jack Dongarra et al., MAGMA. 2014
  - Task-based dense exact solver on GPU
  - $O(n^3)$ complexity
- Kadir Akbudak et al. Tile low-rank Cholesky Factorization. 2017
  - Block Low-rank solver parallelized in OpenMP
  - CPU-only execution and $O(n^2)$ complexity





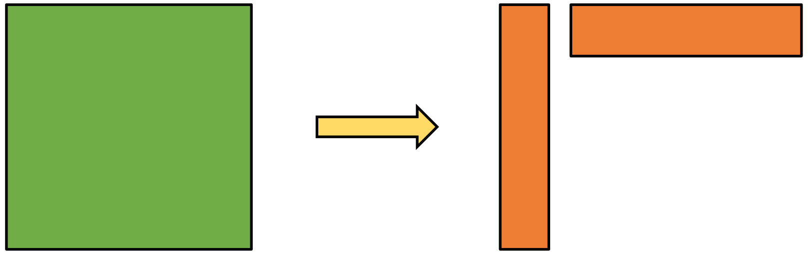
$A = L * U$

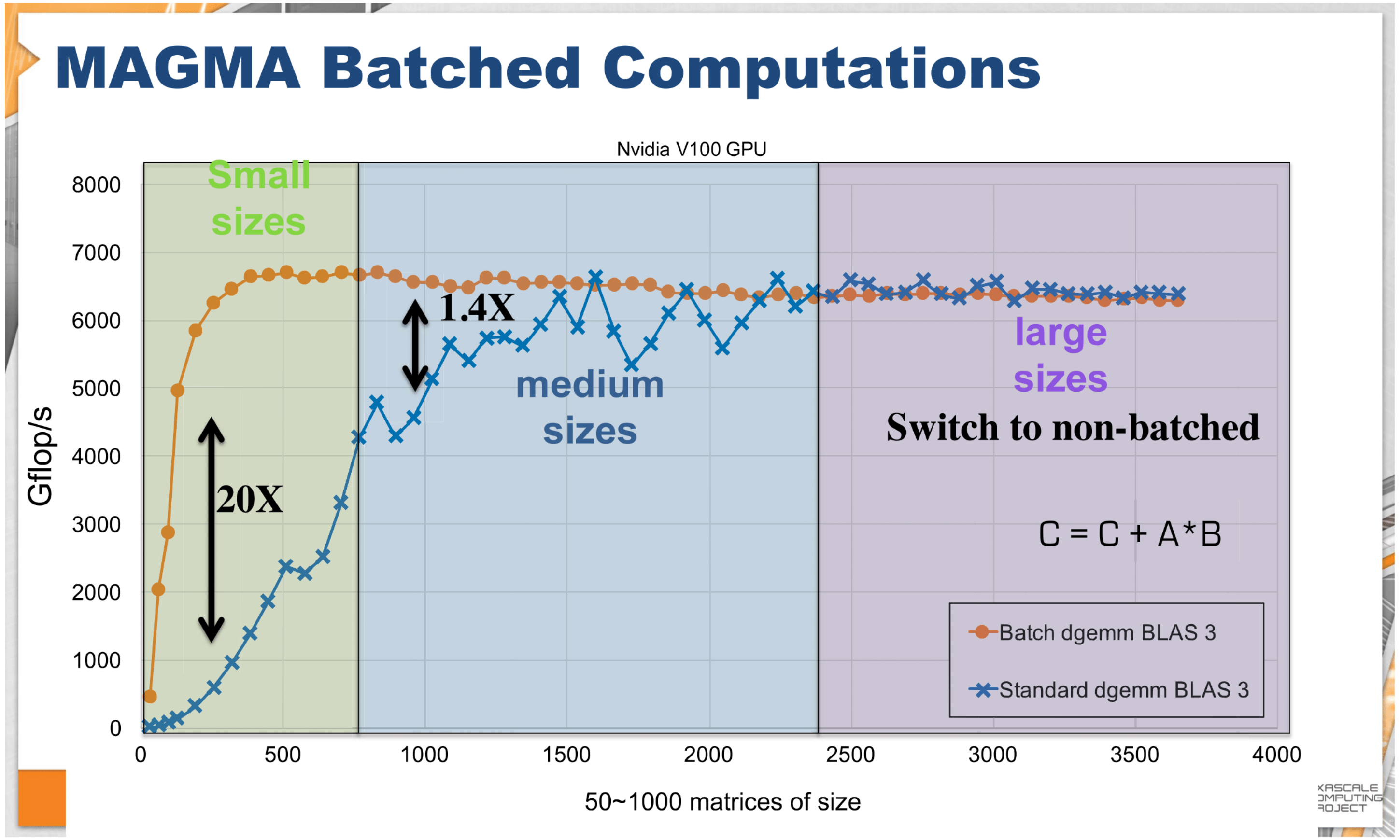
$A = L^{-1} * B$

$A = A + B * C$

https://people.eecs.berkeley.edu/~demmel/cs267/lecture12/lecture12.html

Problem 1:
Data dependency in block & hierarchical LU factorization



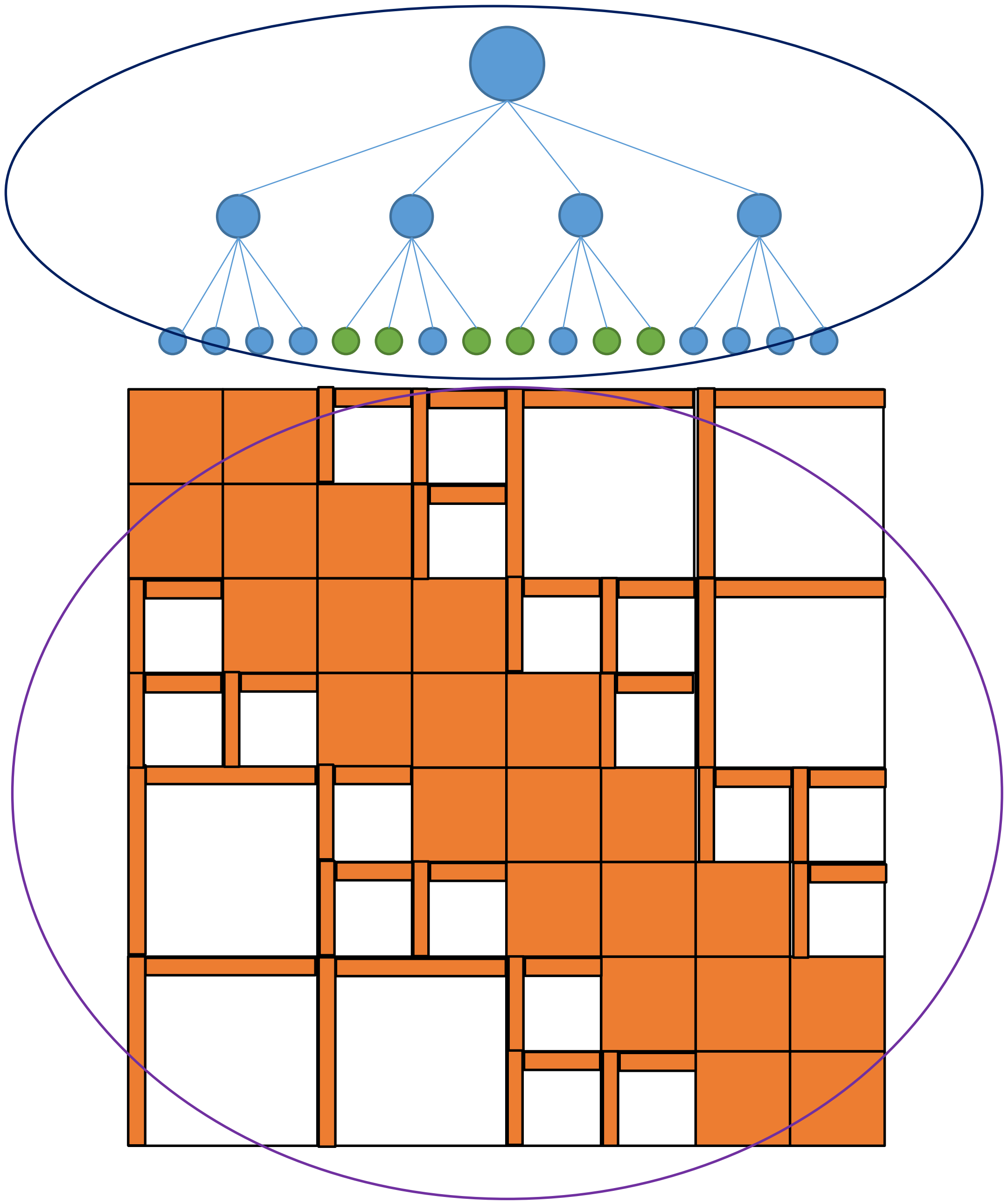


http://icl.utk.edu/projectsfiles/magma/tutorial/ecp2018-magma-tutorial.pdf

Problem 2:

LR blocks can exhibit large kernel setup overheads, if they are not batched.

Runtime System Design



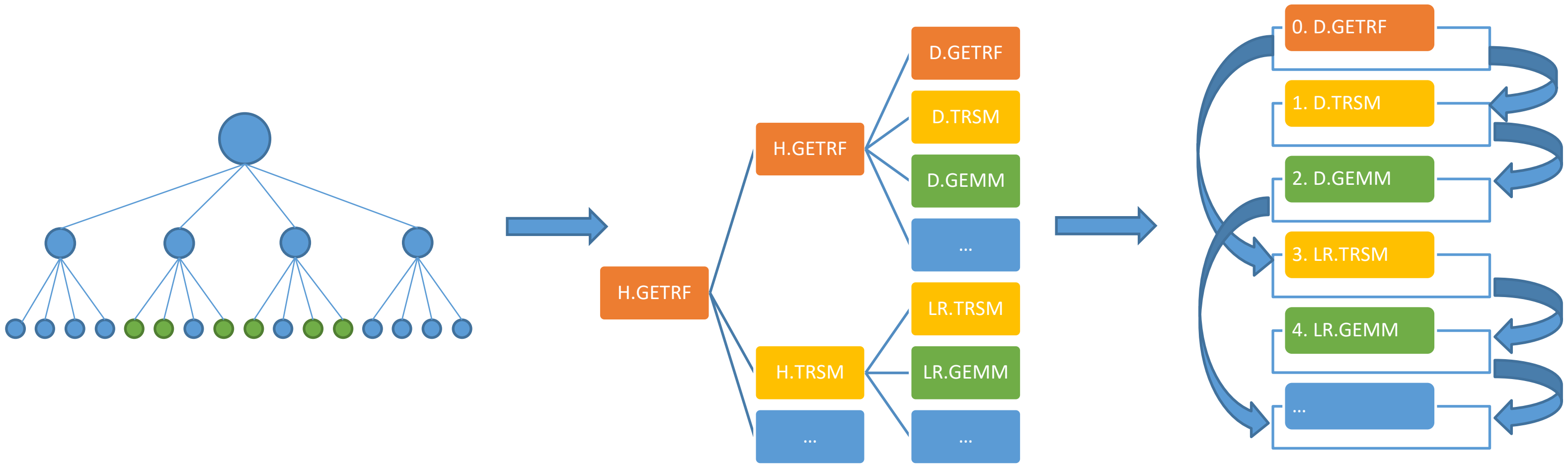
CPU:

1. Processes H-matrix tree structure.

2. Batches tasks together.

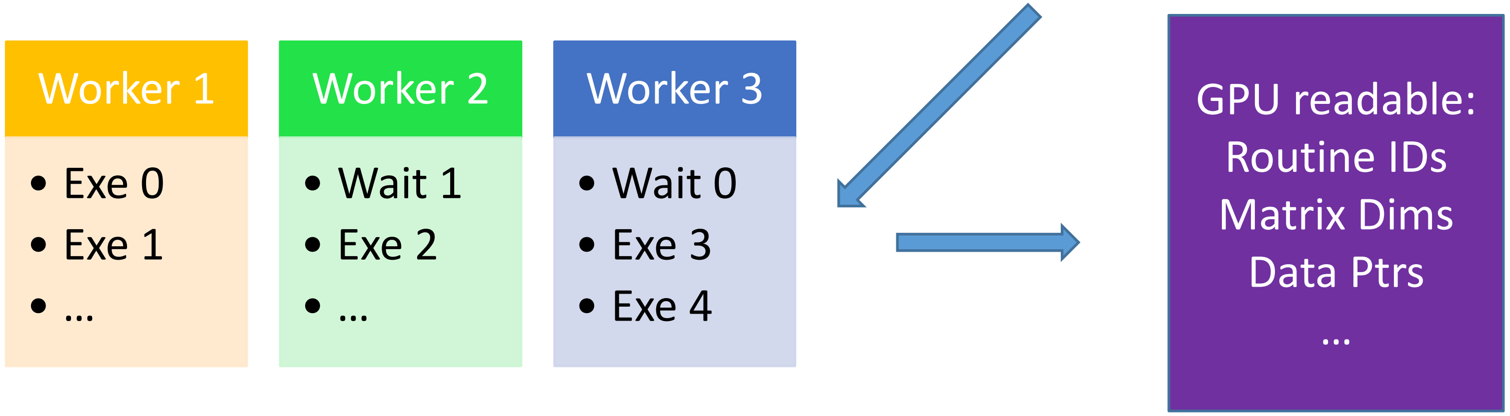
GPU:

3. Processes the matrix block by block.



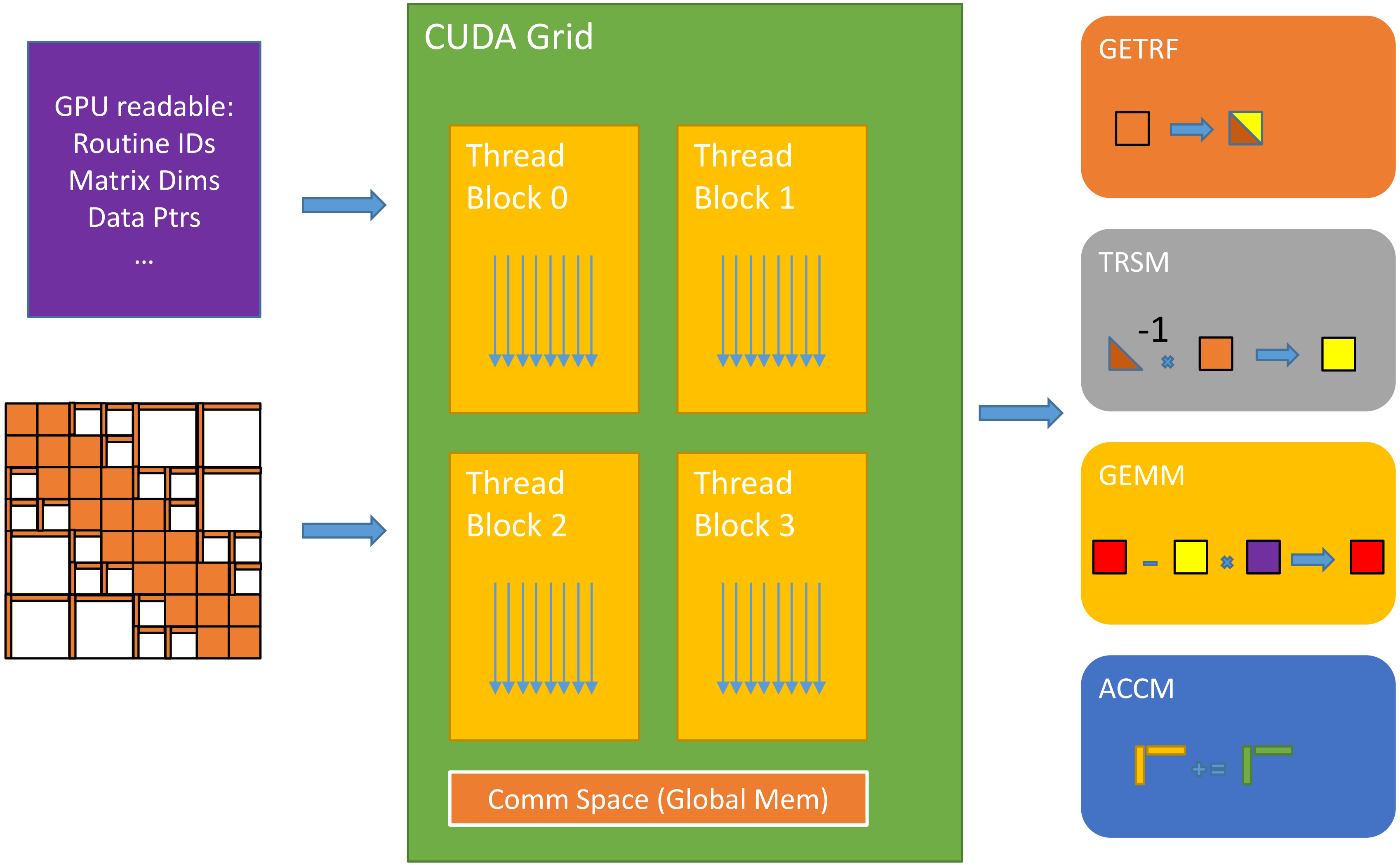
1. Task tree generation

2. Tree Flattening & DAG generation



3. Scheduling the Tasks

4. GPU Inst generation

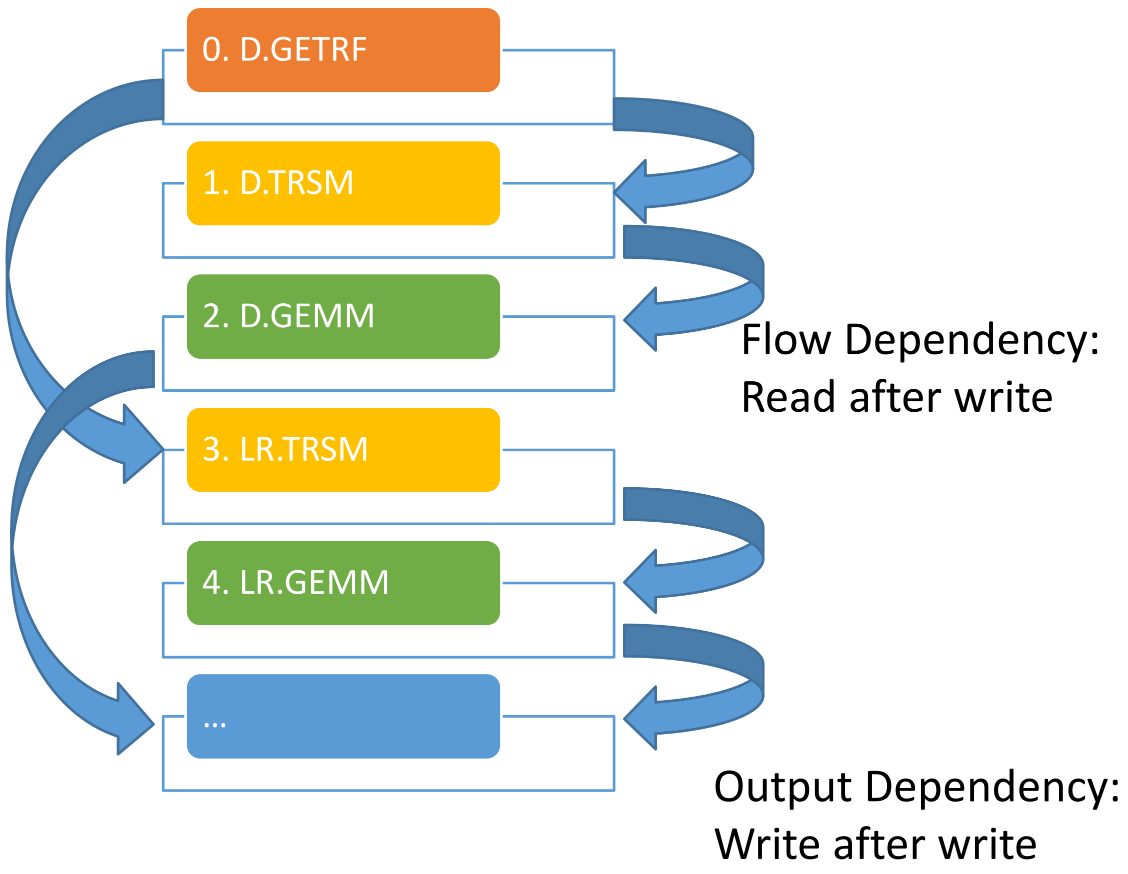


Low-rank
Accumulation
LR += LR

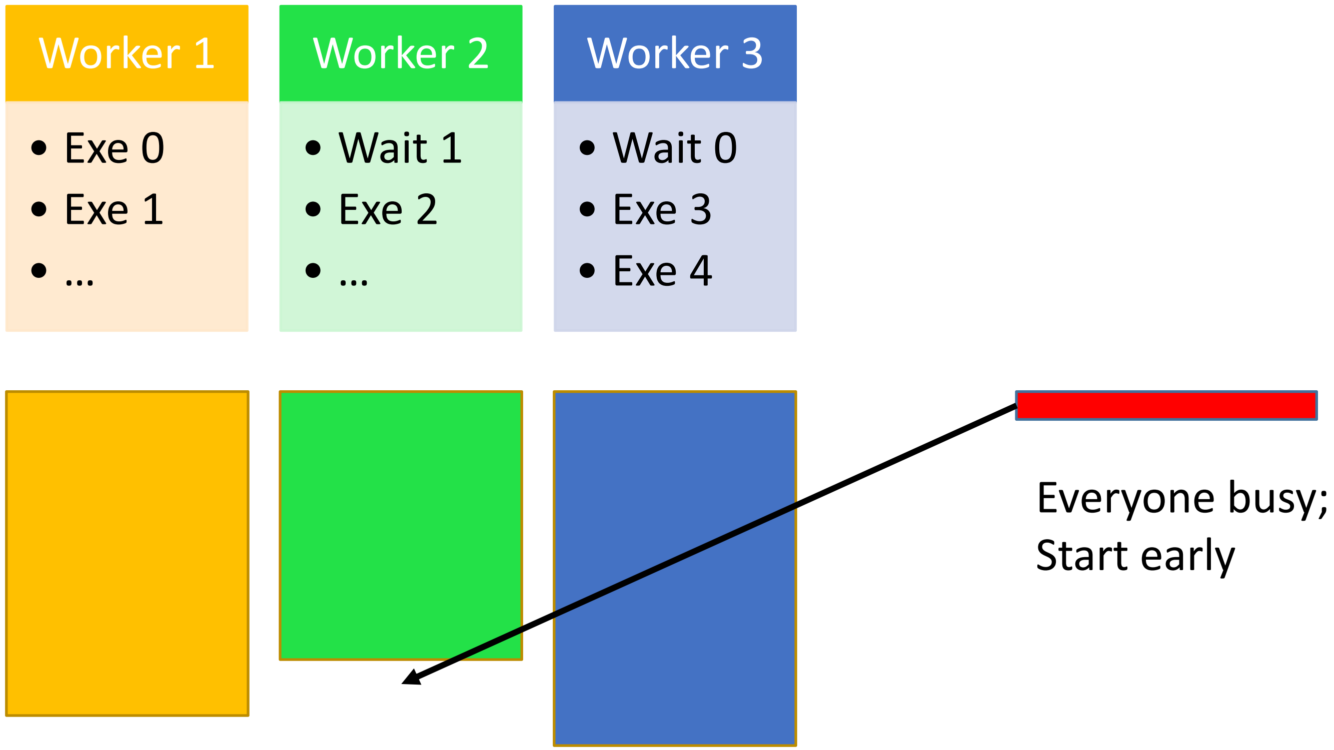
1. Dependency Checker

- Naive:
- Check between every two instructions

- More optimized:
- Inherits dependency relations from parent operation

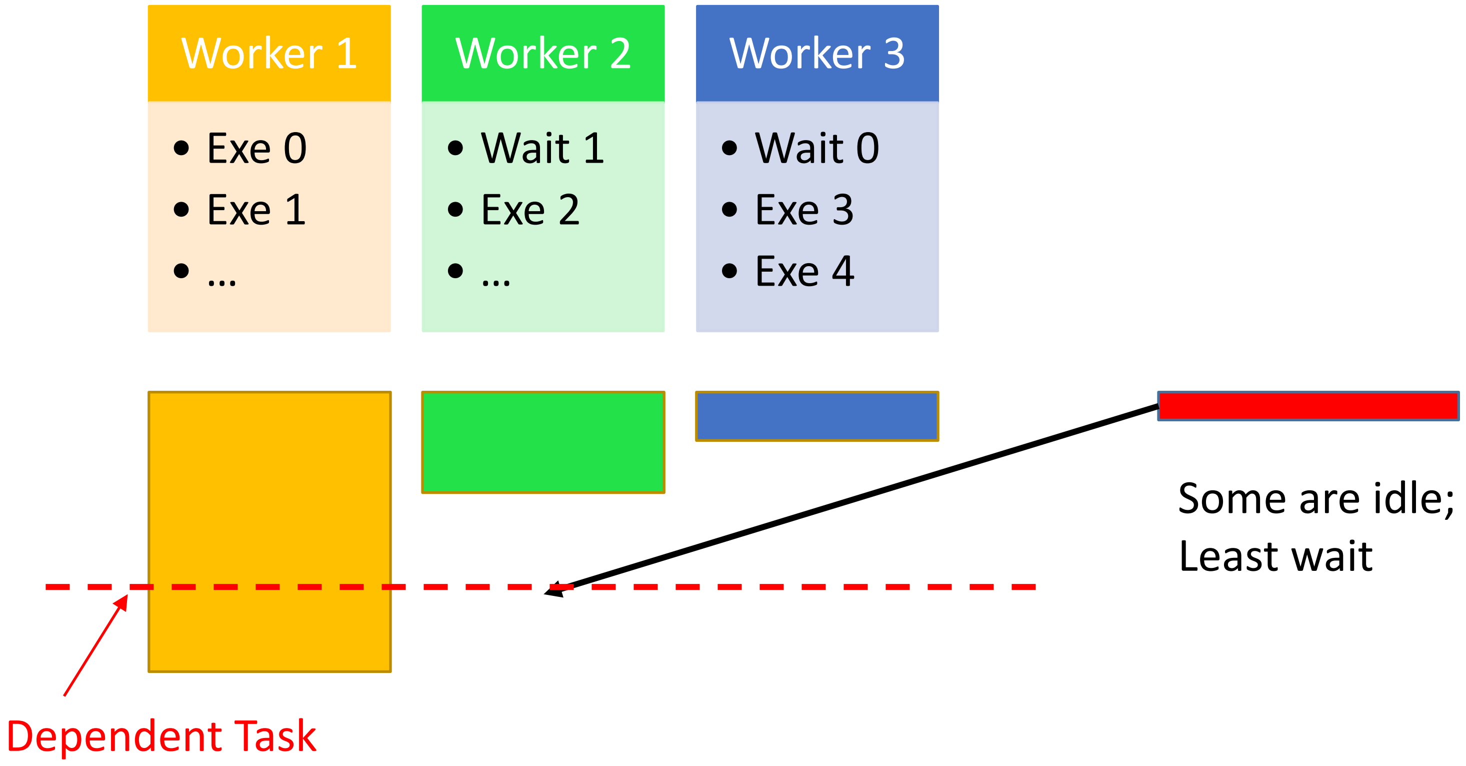


2. Scheduler



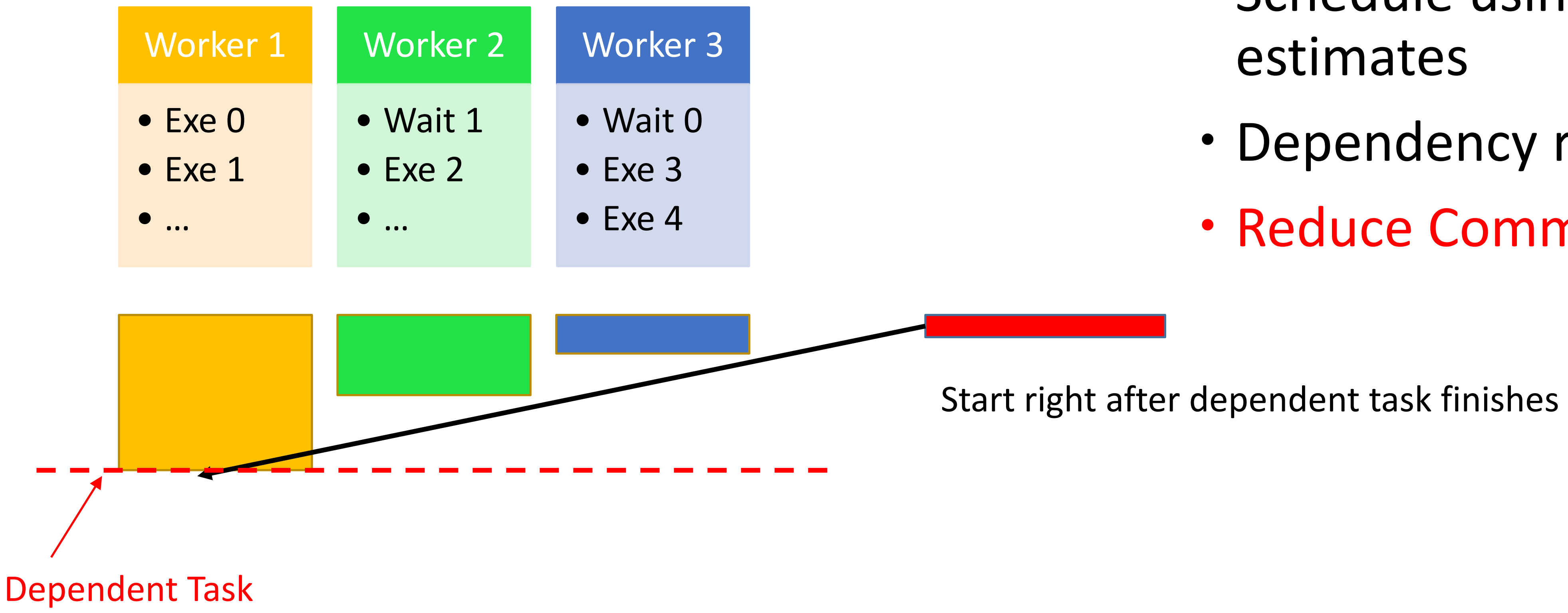
- 3 Heuristics
- Schedule using FLOPs estimates

2. Scheduler



- 3 Heuristics
- Schedule using FLOPs estimates
- Dependency relations

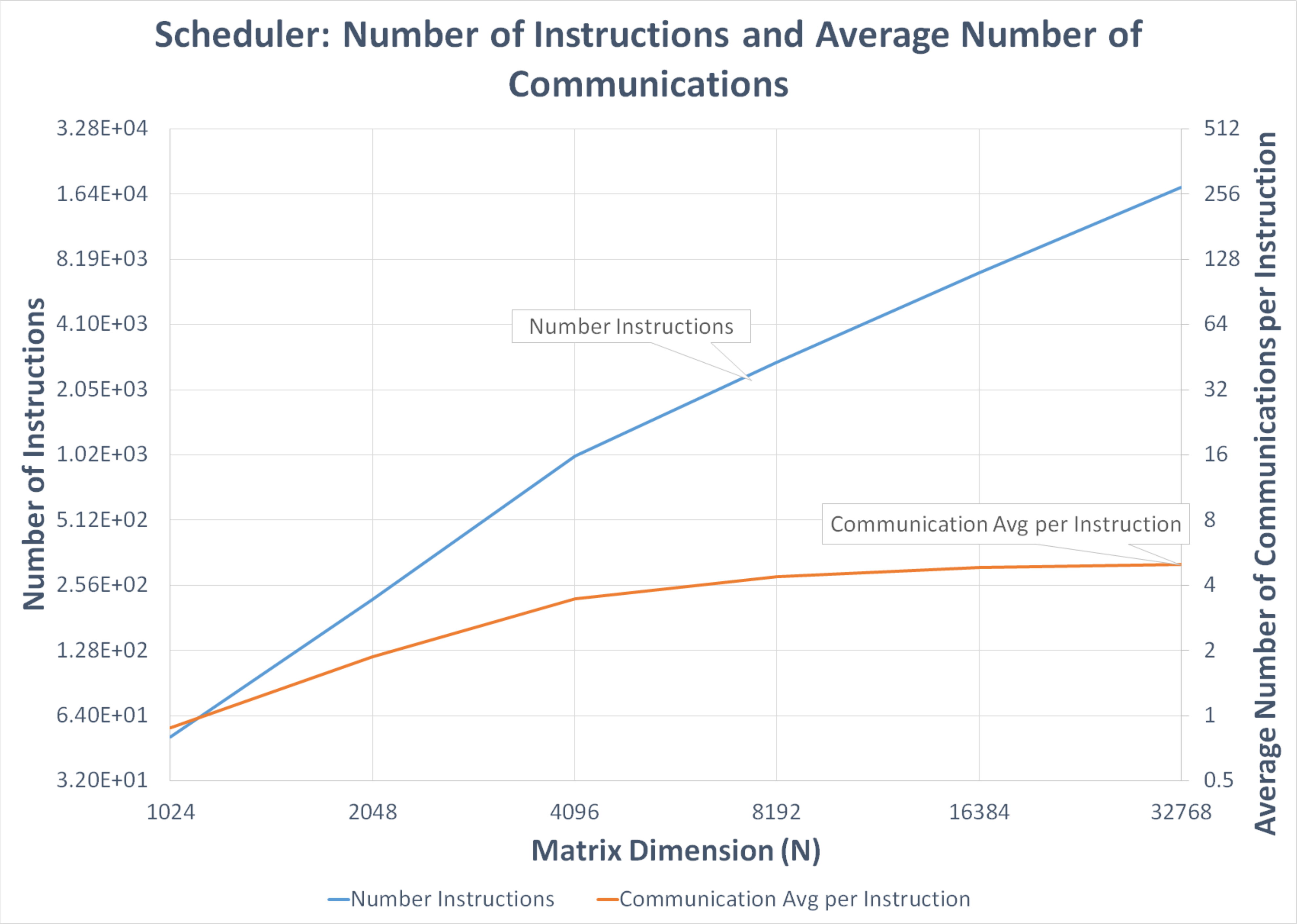
2. Scheduler

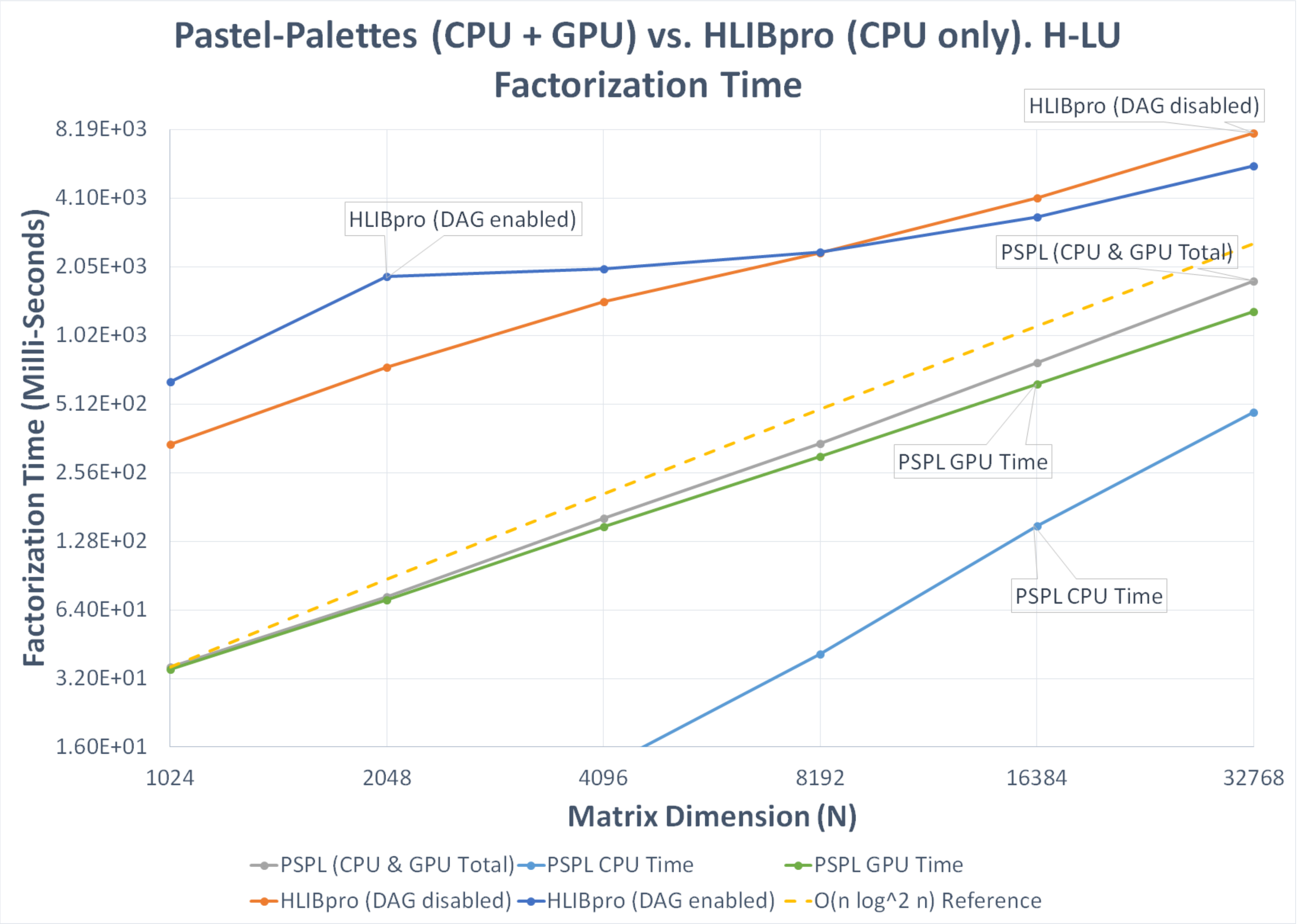


- 3 Heuristics
- Schedule using FLOPs estimates
- Dependency relations
- Reduce Communications

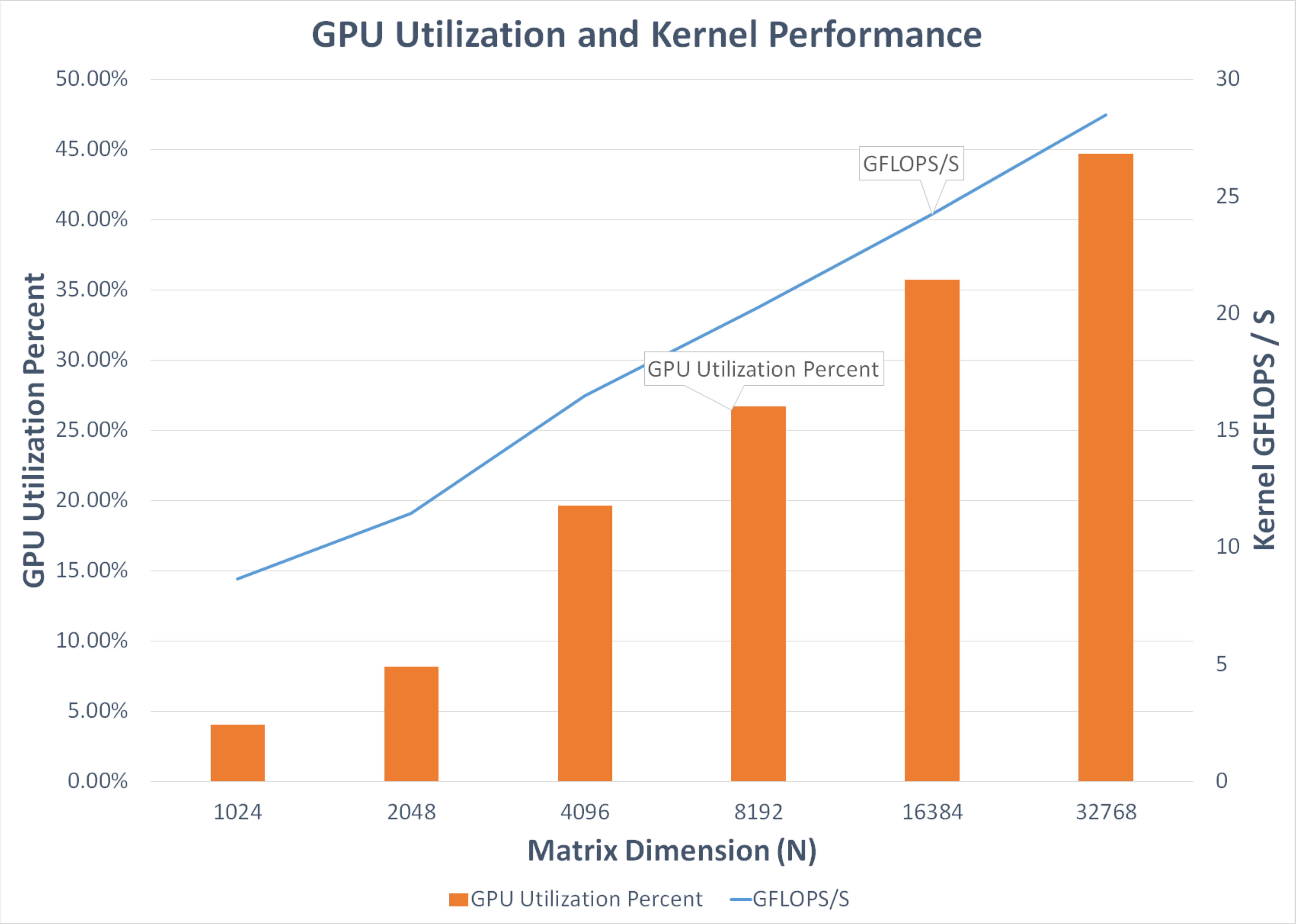
Experiments and Results

- We are most interested in
  1. Scheduler performance: load balancing and communications
  2. Kernel performance: time required to finish H-LU on GPU
  3. Runtime setup cost: additional work done other than factorization
- Experiment setup:
  - CPU: Intel Core-i9 9900k
  - GPU: NVIDIA GeForce RTX2080Ti
  - Reference library: HLIBpro that runs on Intel Core i9-9900k


Scheduler: Number of Instructions and Average Number of Communications
3.28E+04
1.64E+04
8.19E+03
4.10E+03
2.05E+03
1.02E+03
5.12E+02
2.56E+02
1.28E+02
6.40E+01
3.20E+01
Number of Instructions
512
256
128
64
32
16
8
4
2
1
0.5
Average Number of Communications per Instruction
1024
2048
4096
8192
16384
32768
Matrix Dimension (N)
Number Instructions
Communication Avg per Instruction
Number Instructions
Communication Avg per Instruction

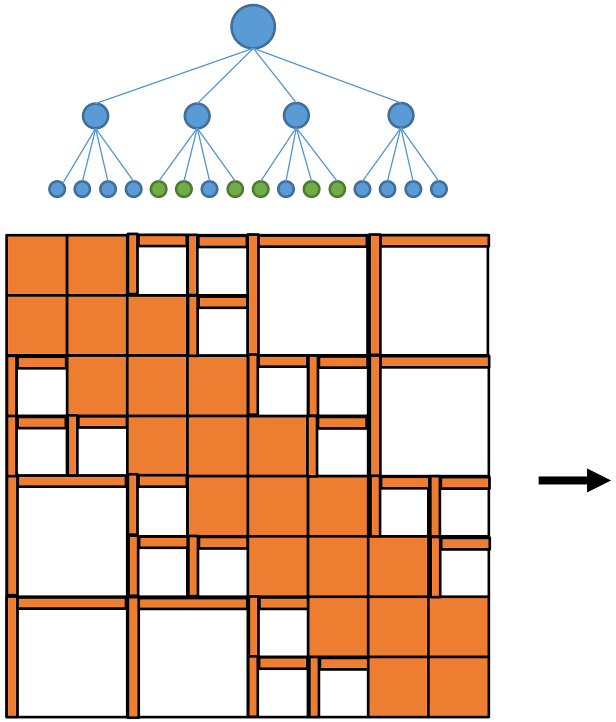


Pastel-Palettes (CPU + GPU) vs. HLIBpro (CPU only). H-LU Factorization Time
HLIBpro (DAG disabled)
HLIBpro (DAG enabled)
PSPL (CPU & GPU Total)
PSPL GPU Time
PSPL CPU Time
Factorization Time (Milli-Seconds)
8.19E+03
4.10E+03
2.05E+03
1.02E+03
5.12E+02
2.56E+02
1.28E+02
6.40E+01
3.20E+01
1.60E+01
1024
2048
4096
8192
16384
32768
Matrix Dimension (N)
PSPL (CPU & GPU Total)
PSPL CPU Time
PSPL GPU Time
HLIBpro (DAG disabled)
HLIBpro (DAG enabled)
O(n log^2 n) Reference

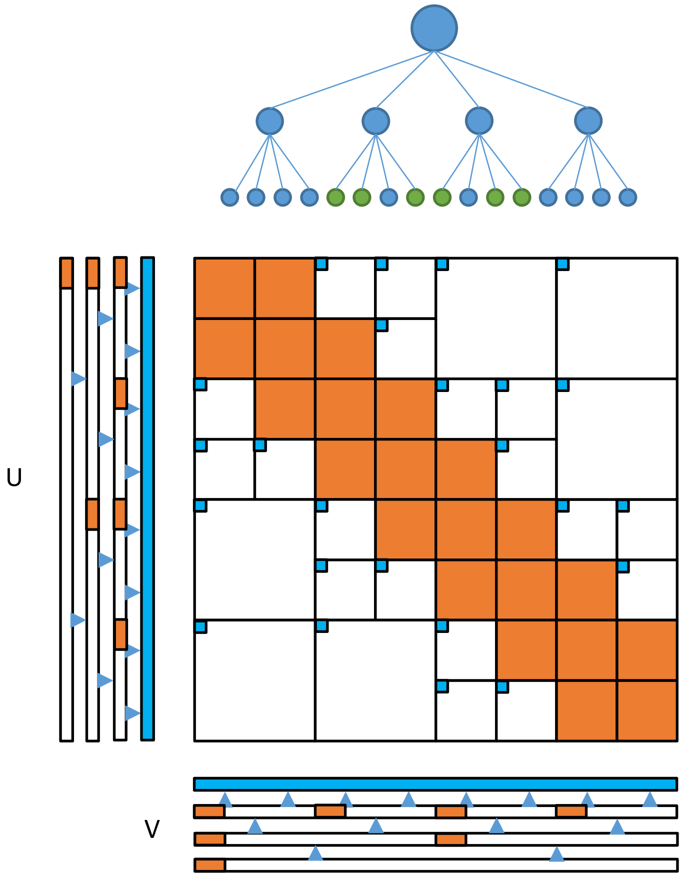

GPU Utilization and Kernel Performance
50.00%
45.00%
40.00%
35.00%
30.00%
25.00%
20.00%
15.00%
10.00%
5.00%
0.00%
GPU Utilization Percent
30
25
20
15
10
5
0
Kernel GFLOPS / S
GFLOPS/S
GPU Utilization Percent
1024
2048
4096
8192
16384
32768
Matrix Dimension (N)
GPU Utilization Percent
GFLOPS/S

Nested basis and H^2-Matrices

H2-Matrix



H-Matrix



H2-Matrix

Shared Basis:

$$A = U_i \times S \times V_j$$

Both U and V are from shared entries outside the matrix

Nested Basis:

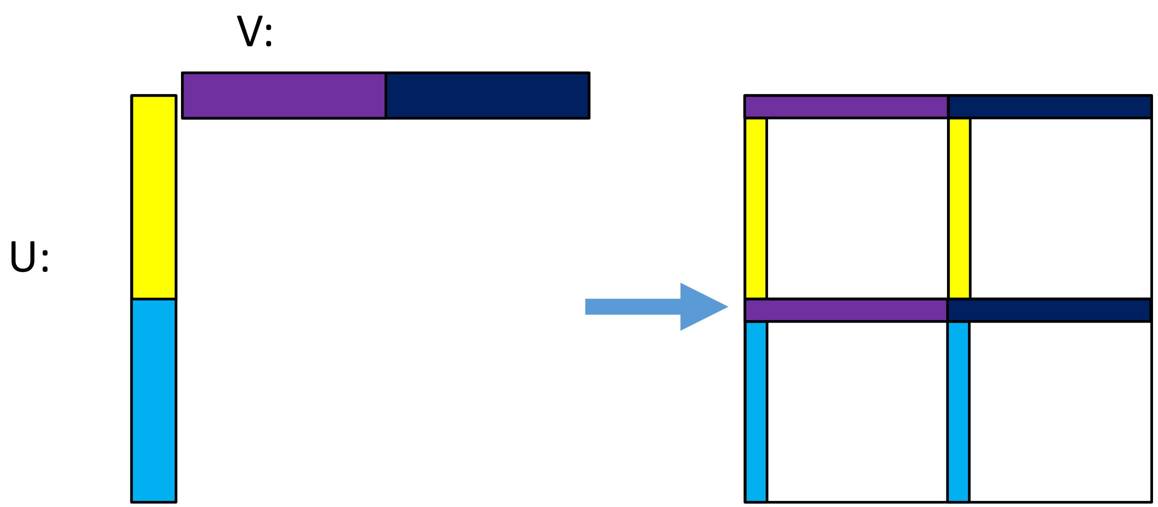
Connection between different layers in bases

H2-Matrix Construction

1. Cluster basis construction - directly from the entries, 4 steps
    1. Accumulation (Top-down recursion on hierarchy)
    2. Propagation (Top-down recursion on basis)
    3. Orthogonalization (Top-down recursion on basis)
    4. Translation (Top-down recursion on basis)
2. Cluster basis constructed using admissibility condition

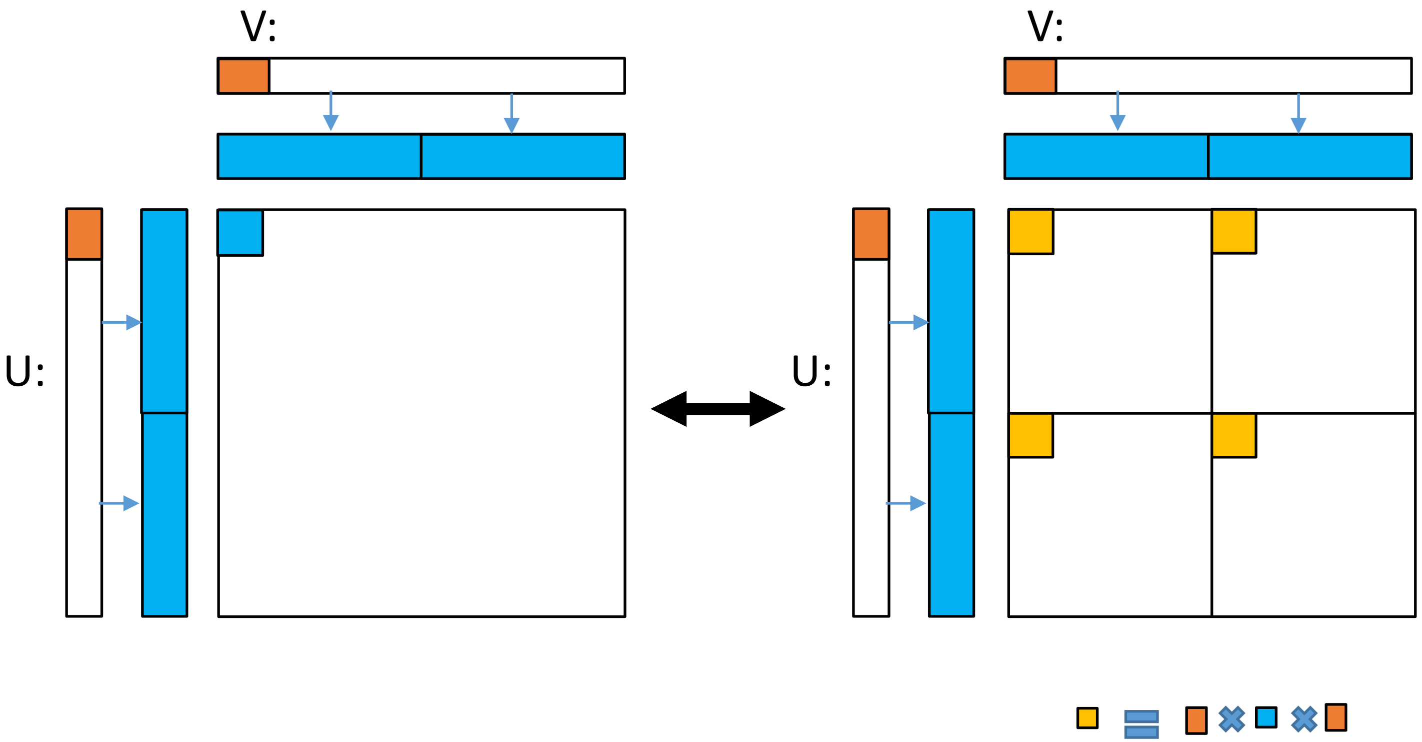
    (More error but generally faster)

Split



Slice the U and V portion to use in the hierarchical matrix

Split in nested basis context

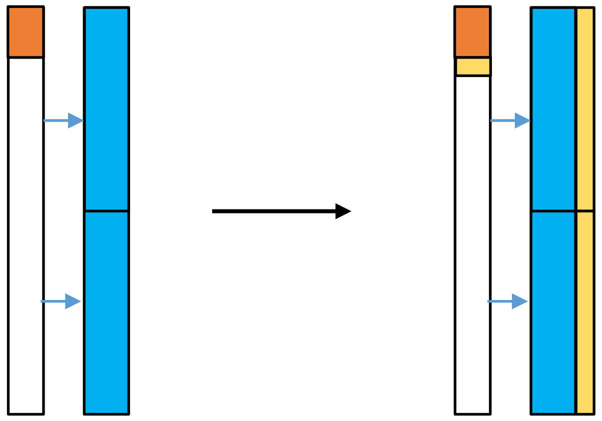


Slice the U and V portion to use in the hierarchical matrix

Multiplied by translation matrices, sliced parts still shares the basis

Nested basis update

- In order to reflect the accumulated results from other clusters, while keeping the blocks low-rank
- Appending columns to prevent side-effects
- Rows needs to be appended: Input matrix projected off the basis
- $M - Basis \times Basis^T \times M = U \times S \times V$

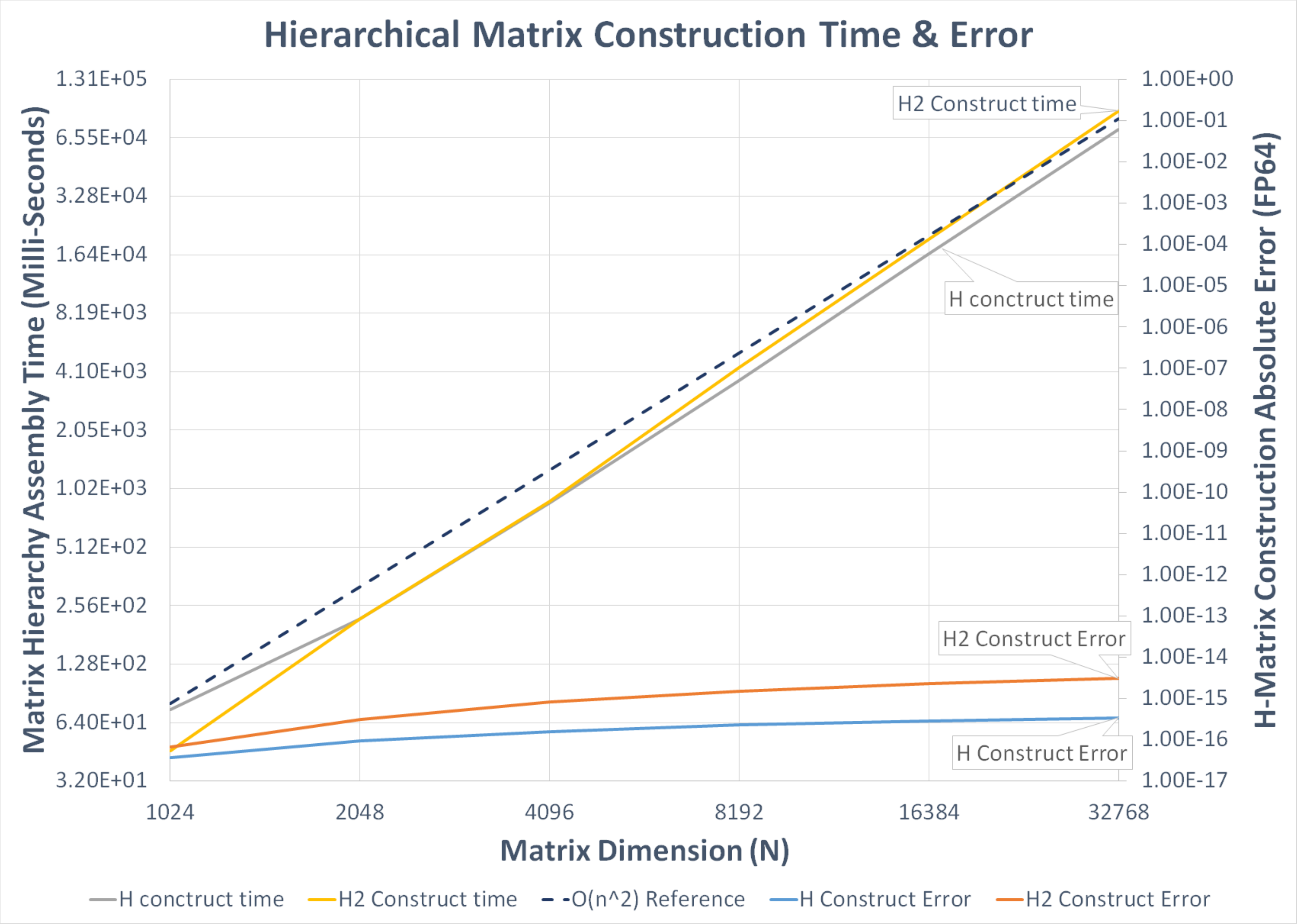


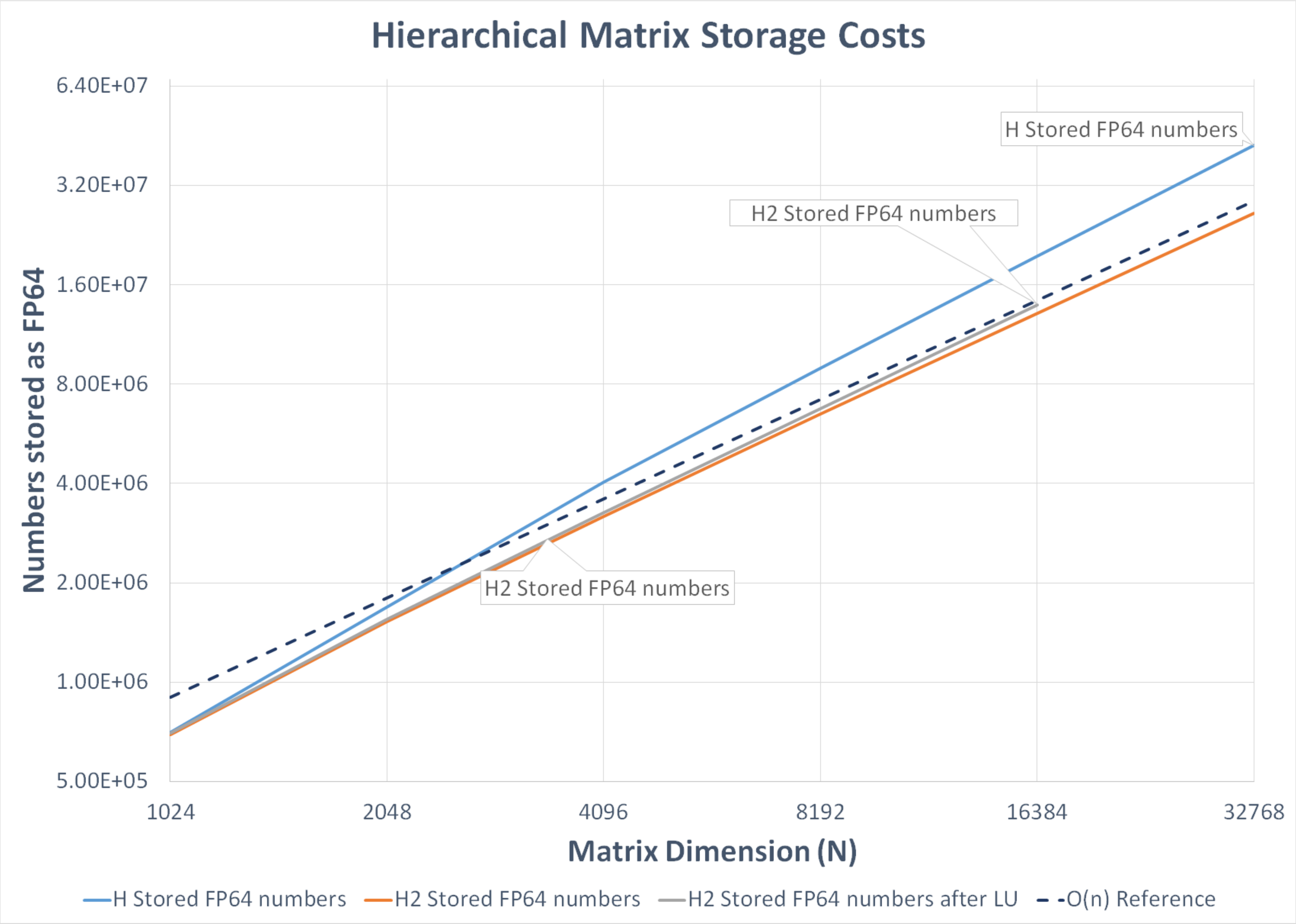
Enlarged S uses the original entries + appended rows

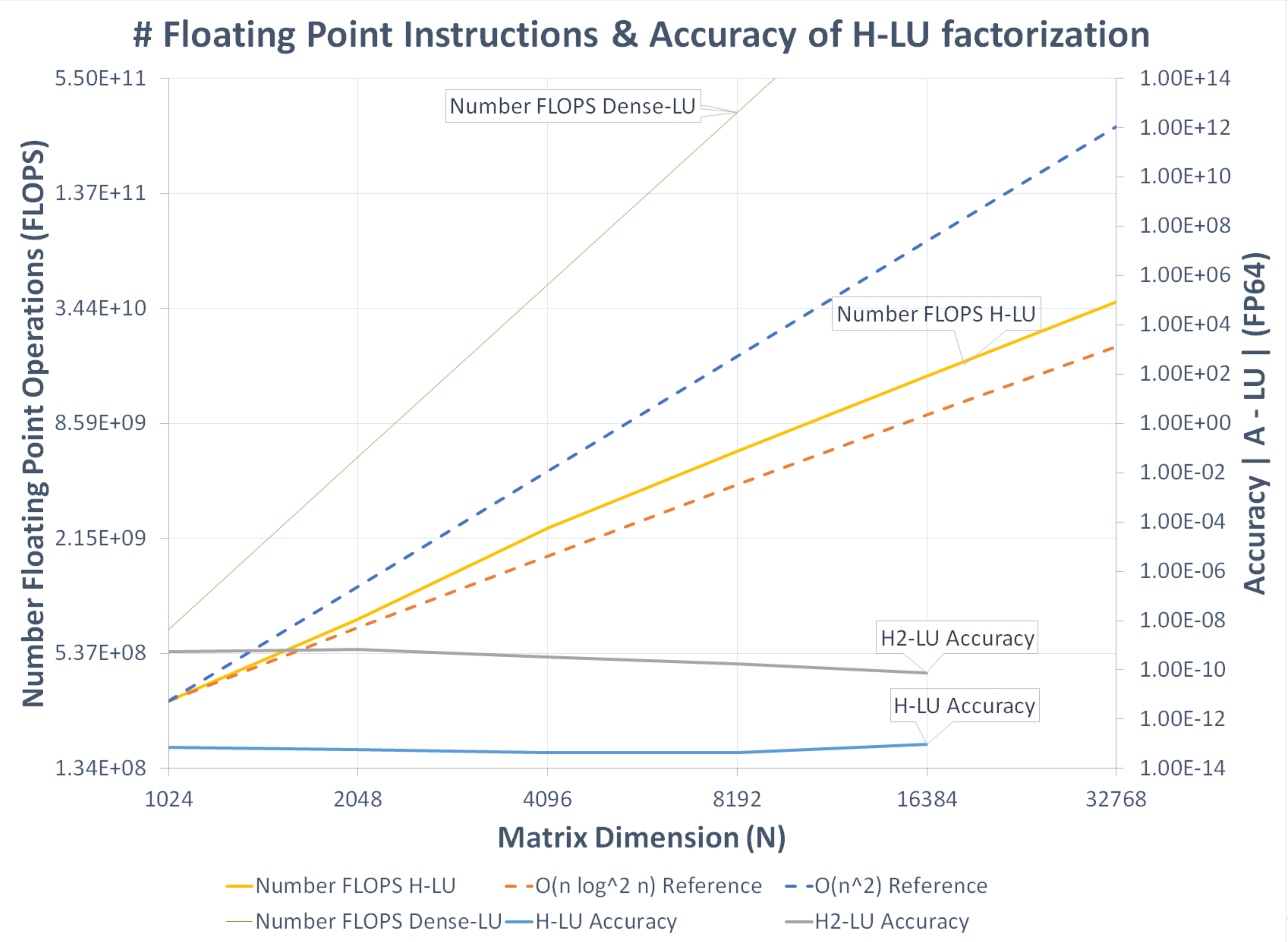
Other S uses only the original entries, so no contents changed

Experiments and Results

- We are most interested in:
  - Construction of nested basis: comparing with H-matrix and its algorithmic complexity
  - Storage cost difference between H2-matrix and H-matrix
  - Accuracy of H2-LU factorization and H-LU factorization, with respect to larger matrix dimensions
- Experiment setup:
  - CPU: Intel Core-i9 9900k
  - Reference: H-LU factorized results from GPU


Hierarchical Matrix Construction Time & Error
Matrix Hierarchy Assembly Time (Milli-Seconds)
1.31E+05
6.55E+04
3.28E+04
1.64E+04
8.19E+03
4.10E+03
2.05E+03
1.02E+03
5.12E+02
2.56E+02
1.28E+02
6.40E+01
3.20E+01
H-Matrix Construction Absolute Error (FP64)
1.00E+00
1.00E-01
1.00E-02
1.00E-03
1.00E-04
1.00E-05
1.00E-06
1.00E-07
1.00E-08
1.00E-09
1.00E-10
1.00E-11
1.00E-12
1.00E-13
1.00E-14
1.00E-15
1.00E-16
1.00E-17
1024
2048
4096
8192
16384
32768
Matrix Dimension (N)
H2 Construct time
H conctruct time
H2 Construct Error
H Construct Error
H conctruct time
H2 Construct time
O(n^2) Reference
H Construct Error
H2 Construct Error


Hierarchical Matrix Storage Costs
Numbers stored as FP64
6.40E+07
3.20E+07
1.60E+07
8.00E+06
4.00E+06
2.00E+06
1.00E+06
5.00E+05
1024
2048
4096
8192
16384
32768
Matrix Dimension (N)
H Stored FP64 numbers
H2 Stored FP64 numbers
H2 Stored FP64 numbers
H Stored FP64 numbers
H2 Stored FP64 numbers
H2 Stored FP64 numbers after LU
O(n) Reference



Floating Point Instructions & Accuracy of H-LU factorization
Number Floating Point Operations (FLOPS)
5.50E+11
1.37E+11
3.44E+10
8.59E+09
2.15E+09
5.37E+08
1.34E+08
Accuracy | A - LU | (FP64)
1.00E+14
1.00E+12
1.00E+10
1.00E+08
1.00E+06
1.00E+04
1.00E+02
1.00E+00
1.00E-02
1.00E-04
1.00E-06
1.00E-08
1.00E-10
1.00E-12
1.00E-14
Number FLOPS Dense-LU
Number FLOPS H-LU
H2-LU Accuracy
H-LU Accuracy
1024
2048
4096
8192
16384
32768
Matrix Dimension (N)
Number FLOPS H-LU
O(n log^2 n) Reference
O(n^2) Reference
Number FLOPS Dense-LU
H-LU Accuracy
H2-LU Accuracy

Conclusion and Future Work

Conclusion

- 1. Runtime system for hierarchical LU factorization
    - Achieved 4x speed up without too much effort in optimization when comparing with HLIBpro
    - Less runtime setup overhead exposed comparing with the tasked based runtime system that HLIBpro is using

- 2. Implementation of hierarchical LU factorization with nested basis
    - $O(n^2)$ construction and $O(n)$ storage cost
    - Produced factorization results that is as accurate as dense and H-LU

Future Work

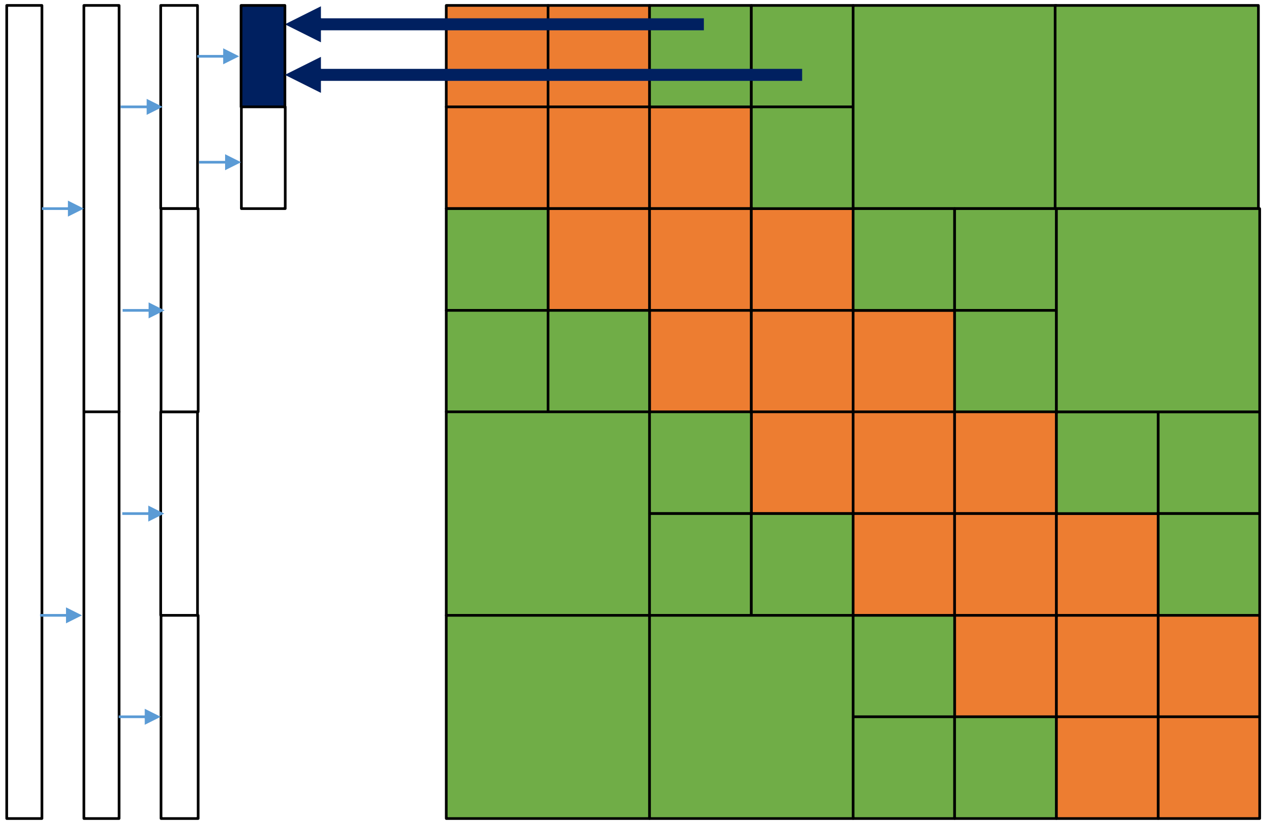
- 1. Aims on both performance and accuracy controlling
- 2. Runtime extendibility to nested basis factorization

Hierarchical LU Factorization	CPU	GPU
Non-nested (H-matrix)	HLIBpro	○
Nested (H2-matrix)	○	In the future

Thank you!

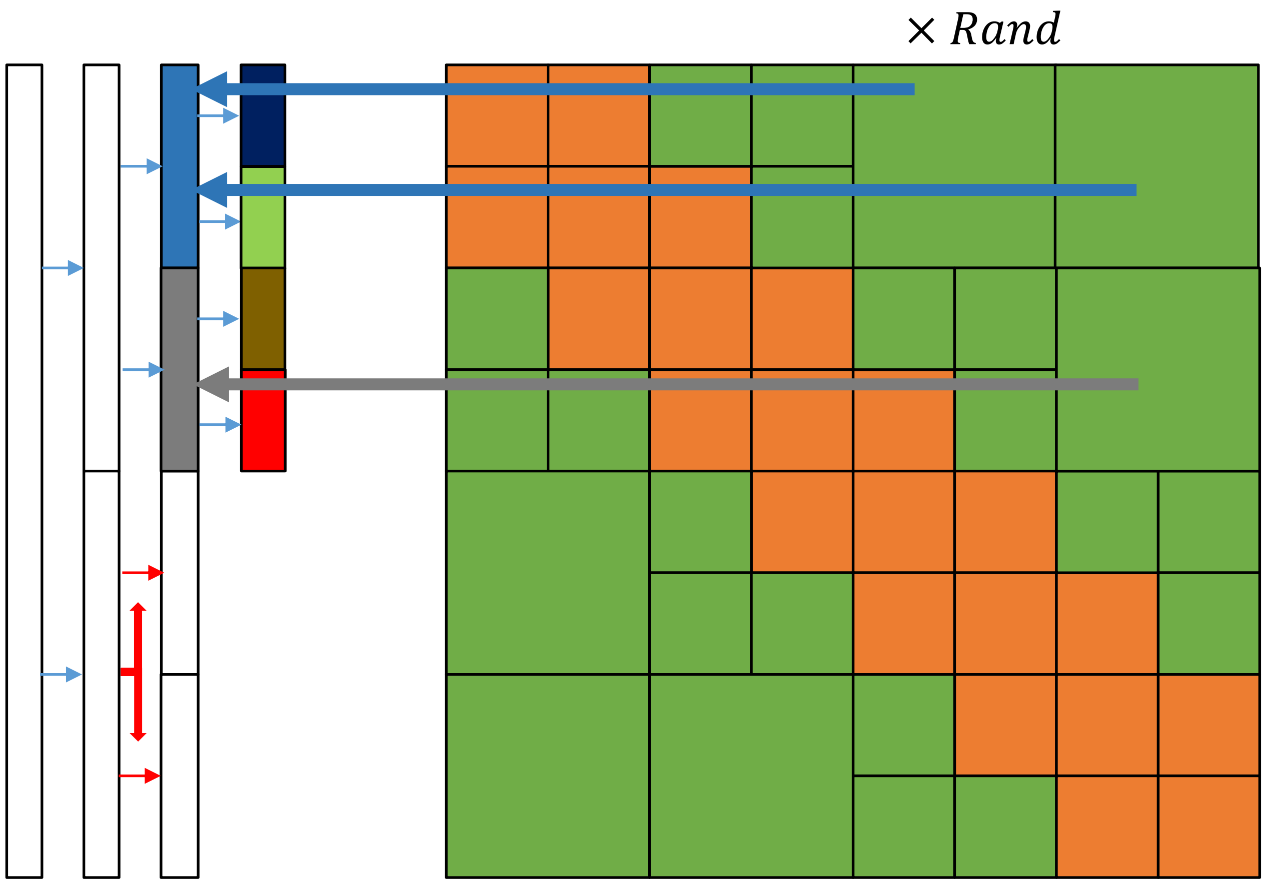
Any question is welcomed

Accumulation



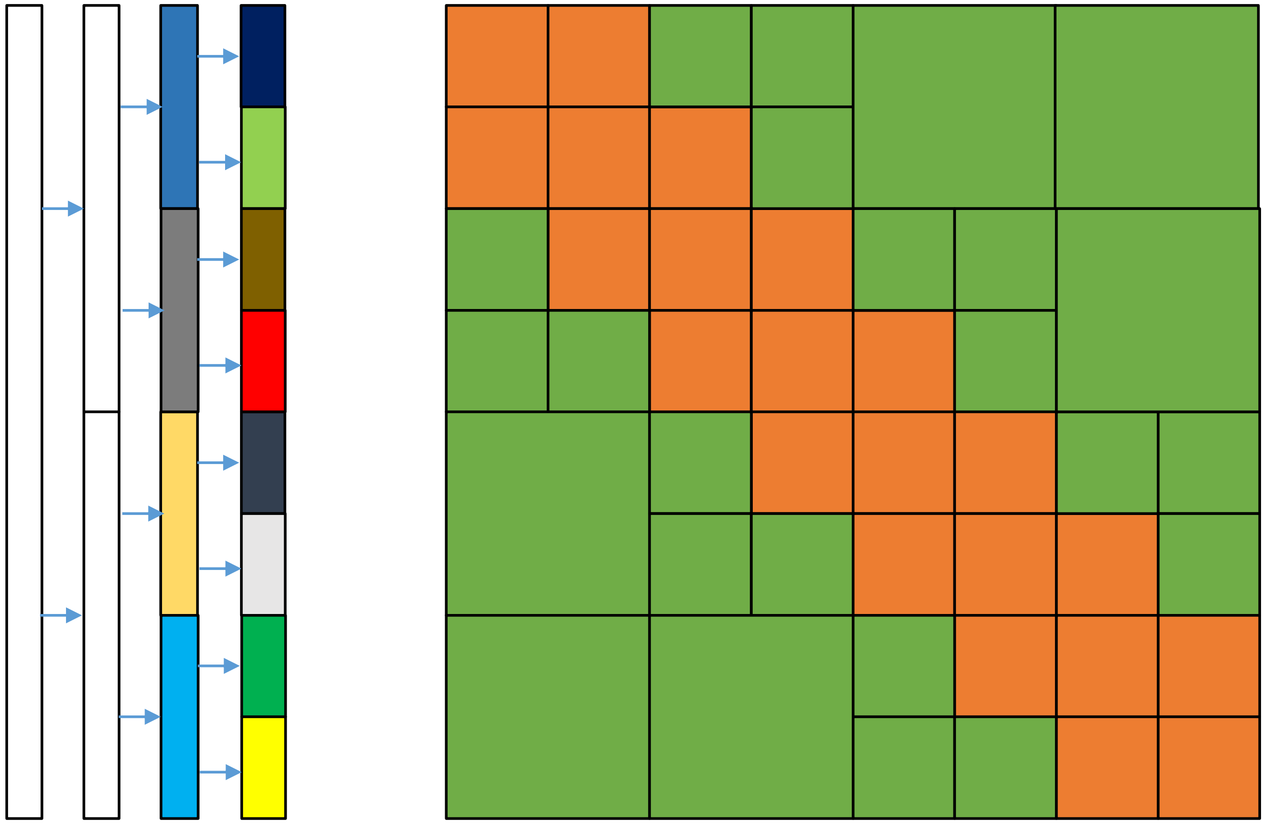
- Admissible blocks contribute to their corresponding row/col
- Init basis with all 0s, fixed dim & rank
- Only accumulate admissible block on current recursion level
- Basis += d x rand_mat(d.n, rank);
- Some might be accumulated more than once
- And some others might never have been accumulated

Accumulation

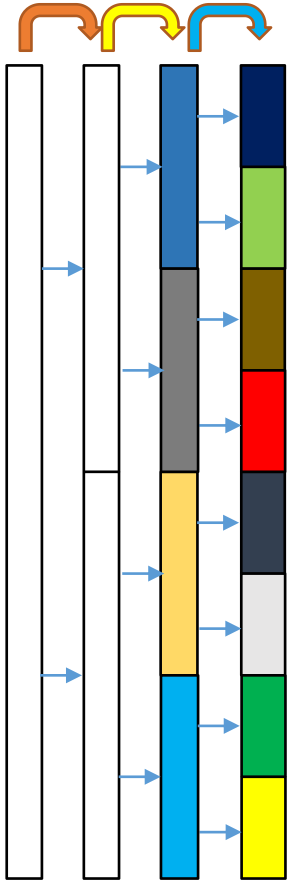


- Recursion:
- If the node on basis tree has no children, generate empty basis for it (Red Arrow)
- Accumulation on lower level has no effect on the upper level and vice versa (Blue Arrow and Gray Arrow)

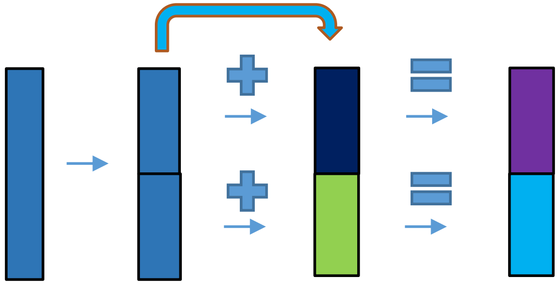
Accumulation (finished)



Propagation



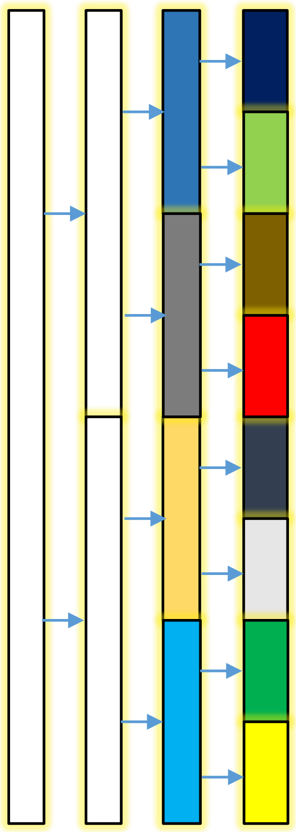
- A non-leaf node with n children:
- Slice the basis into n parts b0 - bn-1, where the row dimension matches its corresponding children.
- For children i:
- Basis += bi;
- Or
- Basis += bi x rand_mat(rank, rank);   // More randomization



So that:

1. No empty low-level bases
2. Lower-level bases also takes higher-level admis blocks into calculation

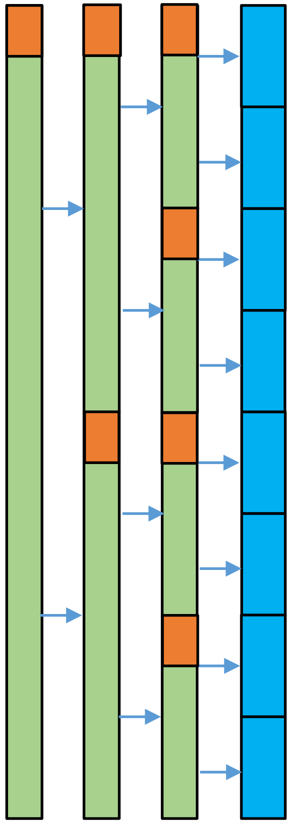
Orthogonalization



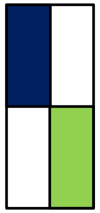
- Basis gets updated with its orthogonalized version:
- Option 1: Interpolative decomposition (with randomization already completed)
- Basis = Basis.qr().getQ();
- Option 2: RSVD (with randomization already completed)
- Y = Basis.qr().getQ();
- U x S x VT = svd(YT x Basis);
- Basis = Y x U;

- RSVD is more costly but sorts the basis according to their singular values
- Might be better for adaptive-rank H2 construction

Translation

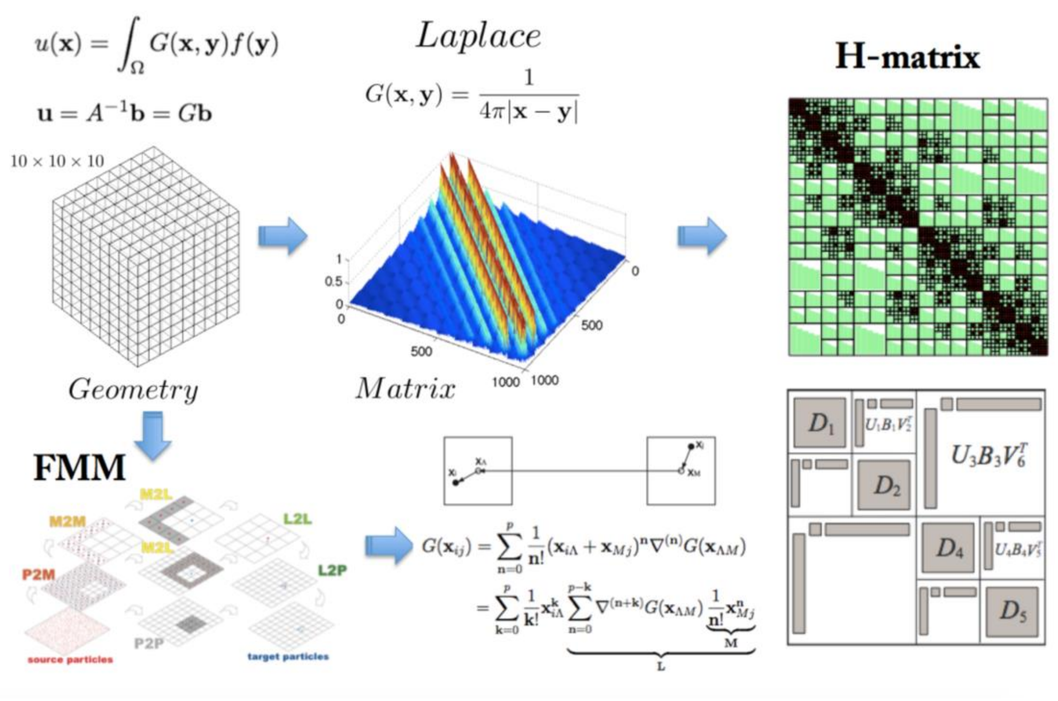


- Use translation matrices at non-leaf levels for more efficient storage
- basis = lower x Trans;
- Which is also:
- Trans = lowerT x basis;
- Lower is aligning the children bases diagonally:



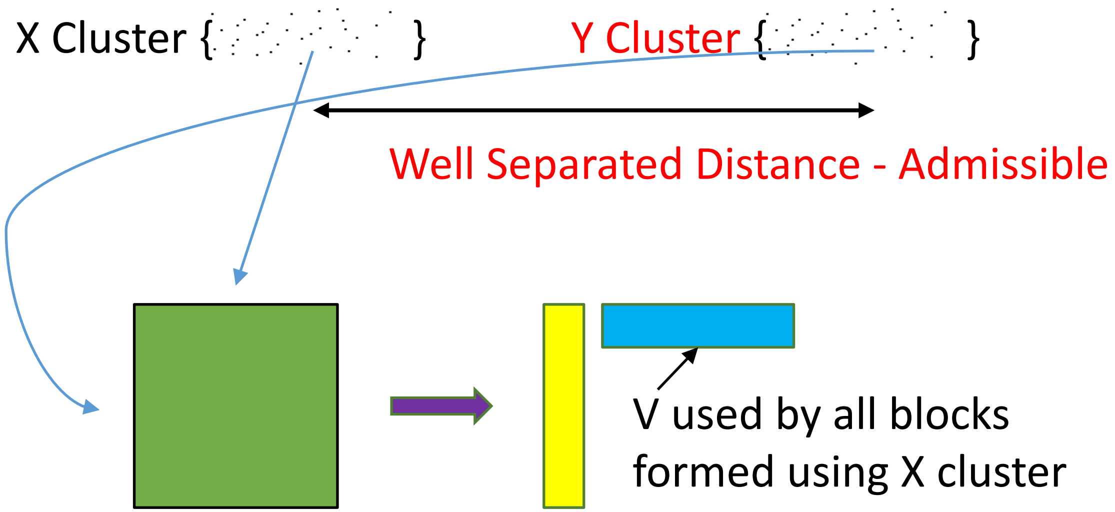
- H2 hierarchy assembly should happen before translation for more efficient calculation

Cluster basis from admissibility condition



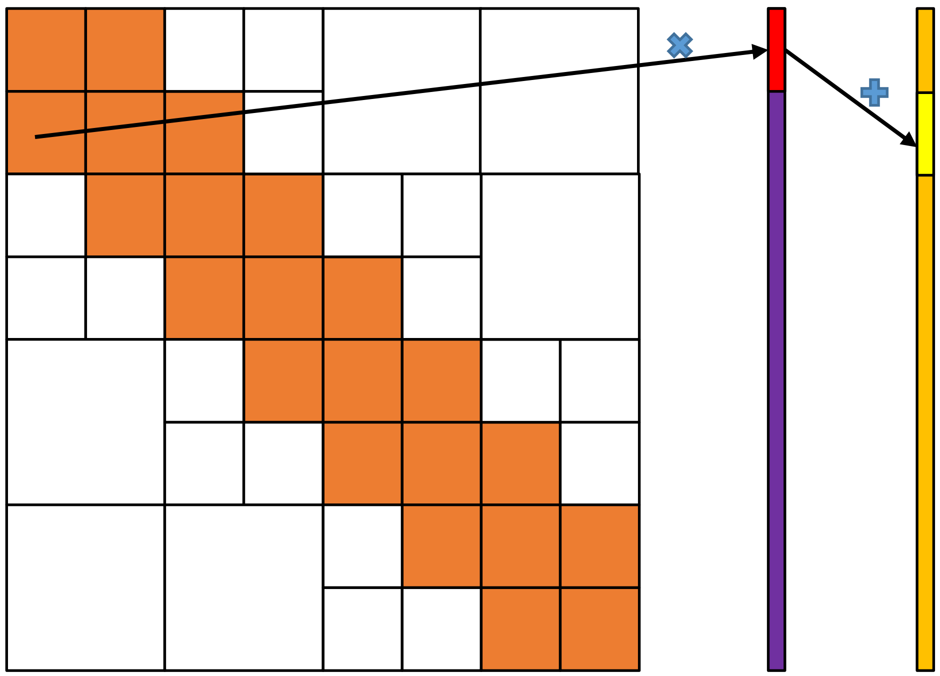
Rio Yokota - Introduction of FMM lecture

- Only one cluster determines values in U/V
- Use a "representative" block to obtain U/V



Matrix Vector multiplication with nested basis

- Two accumulator vectors: admissible and non-admissible
- Non-admissible: same as H-matrix



Matrix Vector multiplication with nested basis

- Two accumulator vectors: admissible and non-admissible
- Admissible: Approximates all admissible blocks