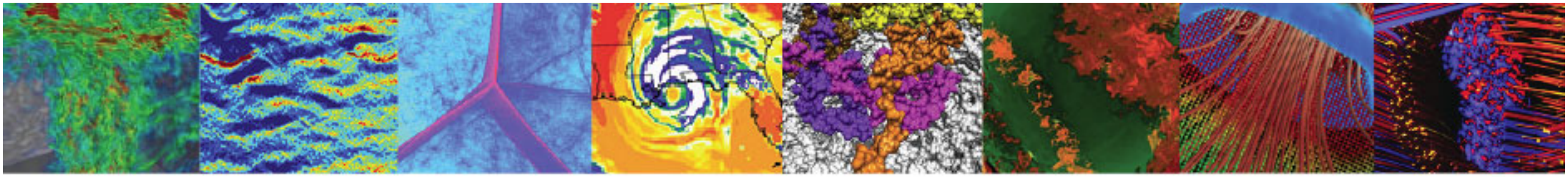


Argonne Training Program on

EXTREME-SCALE COMPUTING

July 31 – August 12, 2016

Adaptive Linear Solvers and Eigensolvers

Jack Dongarra

University of Tennessee
Oak Ridge National Laboratory
University of Manchester

Dense Linear Algebra

- Common Operations

$$Ax = b; \quad \min_x \| Ax - b \|; \quad Ax = \lambda x$$

- A major source of large dense linear systems is problems involving the solution of boundary integral equations.
  - The price one pays for replacing three dimensions with two is that what started as a sparse problem in $O(n^3)$ variables is replaced by a dense problem in $O(n^2)$.
- Dense systems of linear equations are found in numerous other applications, including:
  - airplane wing design;
  - radar cross-section studies;
  - flow around ships and other off-shore constructions;
  - diffusion of solid bodies in a liquid;
  - noise reduction; and
  - diffusion of light through small particles.

Existing Math Software - Dense LA

DIRECT SOLVERS	License	Support	Type		Language			Mode		
			Real	Complex	F77/ F95	C	C++	Shared	Accel.	Dist
Chameleon	CeCILL-C	See authors	X	X		X		X	C	M
DPLASMA	BSD	yes	X	X		X		X	C	M
Eigen	Mozilla	yes	X	X			X	X		
Elemental	New BSD	yes	X	X			X			M
ELPA	LGPL	yes	X	X	F90	X		X		M
FLENS	BSD	yes	X	X			X	X		
hmat-oss	GPL	yes	X	X	X	X	X	X		
LAPACK	BSD	yes	X	X	X	X		X		
LAPACK95	BSD	yes	X	X	X			X		
libflame	New BSD	yes	X	X	X	X		X		
MAGMA	BSD	yes	X	X	X	X		X	C/O/X	
NAPACK	BSD	yes	X		X			X		
PLAPACK	LGPL	yes	X	X	X	X				M
PLASMA	BSD	yes	X	X	X	X		X		
rejtrix	by-nc-sa	yes	X				X	X		
ScaLAPACK	BSD	yes	X	X	X	X				M/P
Trilinos/Pliris	BSD	yes	X	X		X	X			M
ViennaCL	MIT	yes	X				X	X	C/O/X	

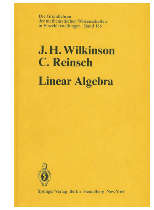
http://www.netlib.org/utk/people/JackDongarra/la-sw.html

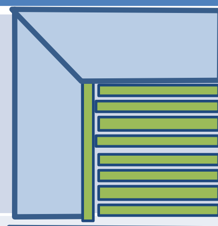
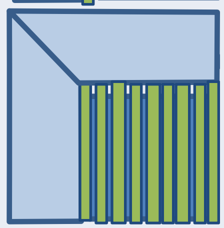
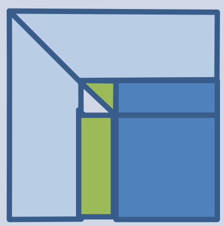
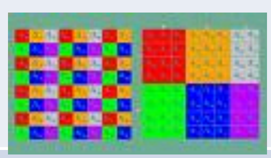
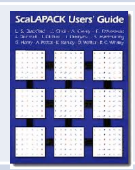
- LINPACK, EISPACK, LAPACK, ScaLAPACK
  - PLASMA, MAGMA

DLA Solvers

- We are interested in developing Dense Linear Algebra Solvers
- Retool LAPACK and ScaLAPACK for multicore and hybrid architectures

40 Years Evolving SW and Alg Tracking Hardware Developments



Software/Algorithms follow hardware evolution in time		
EISPACK (70's) (Translation of Algol)	 	Rely on - Fortran, but row oriented
LINPACK (80's) (Vector operations)	 	Rely on - Level-1 BLAS operations - Column oriented
LAPACK (90's) (Blocking, cache friendly)	 	Rely on - Level-3 BLAS operations
ScaLAPACK (00's) (Distributed Memory)	 	Rely on - PBLAS Mess Passing
PLASMA (10's) New Algorithms (many-core friendly)		Rely on - DAG/scheduler - block data layout - some extra kernels

What do you mean by performance?

- **What is a flop/s?**
  - flop/s is a rate of execution, some number of floating point operations per second.
    - » **Whenever this term is used it will refer to 64 bit floating point operations and the operations will be either addition or multiplication.**

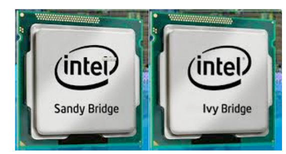
- **What is the theoretical peak performance?**
  - The theoretical peak is based not on an actual performance from a benchmark run, but on a paper computation to determine the theoretical peak rate of execution of floating point operations for the machine.
  - The theoretical peak performance is determined by counting the number of floating-point additions and multiplications (in full precision) that can be completed during a period of time, usually the cycle time of the machine.
  - For example, an Intel Xeon Haswell dual core at 2.3 GHz can complete 16 floating point operations per cycle or a theoretical peak performance of 36.8 GFlop/s per core or 73.6 Gflop/s for the socket.

Peak Performance - Per Core

$$\text{FLOPS} = \text{cores} \times \text{clock} \times \frac{\text{FLOPs}}{\text{cycle}}$$

Floating point operations per cycle per core

- Most of the recent computers have FMA (Fused multiple add): (i.e. $x \leftarrow x + y*z$ in one cycle)
- Intel Xeon earlier models and AMD Opteron have SSE2
  - 2 flops/cycle DP & 4 flops/cycle SP
- Intel Xeon Nehalem ('09) & Westmere ('10) have SSE4
  - 4 flops/cycle DP & 8 flops/cycle SP
- Intel Xeon Sandy Bridge('11) & Ivy Bridge ('12) have AVX
  - 8 flops/cycle DP & 16 flops/cycle SP

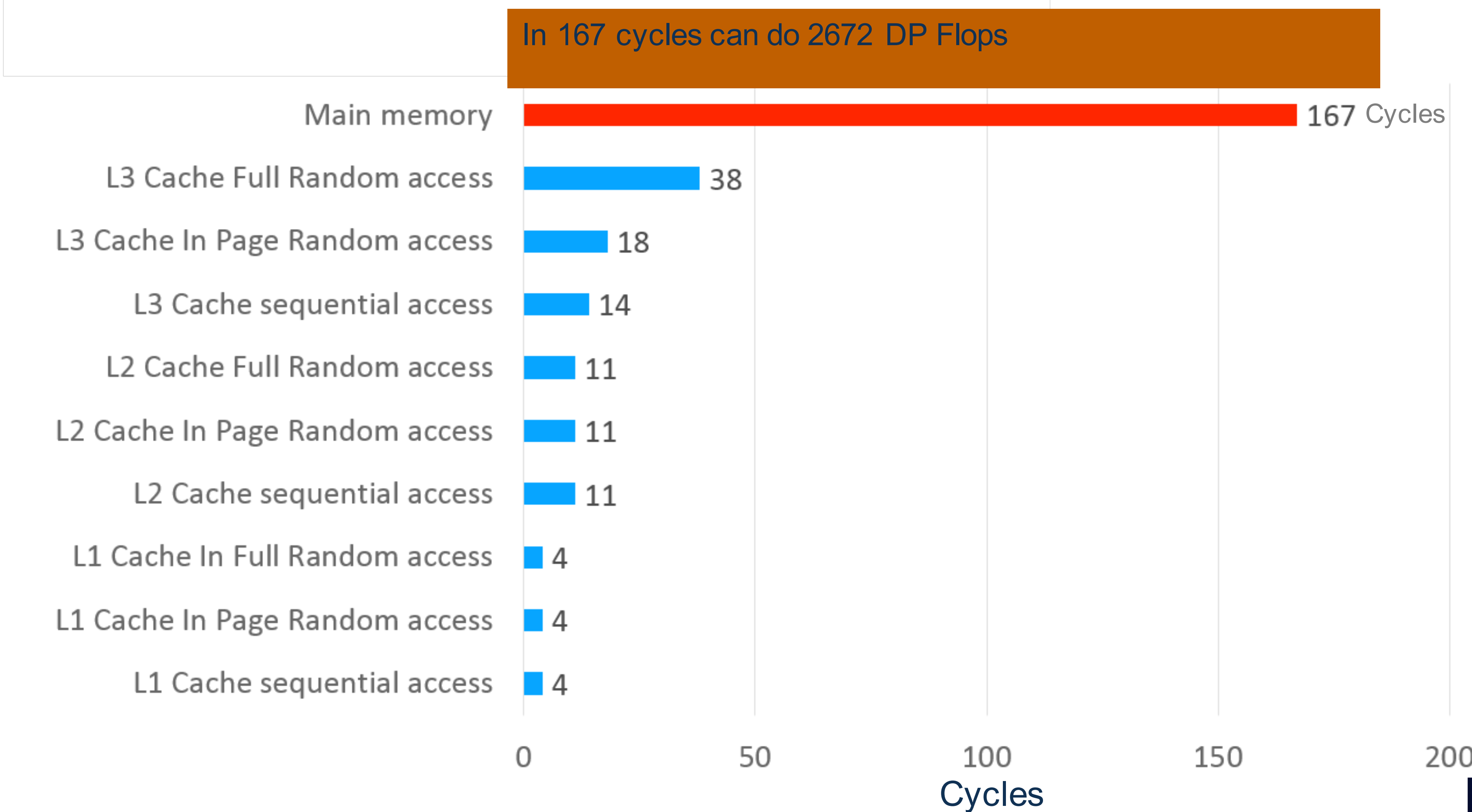


- Intel Xeon Haswell ('13) & (Broadwell ('14)) AVX2
  - 16 flops/cycle DP & 32 flops/cycle SP
  - Xeon Phi (per core) is at 16 flops/cycle DP & 32 flops/cycle SP

We are here

- Intel Xeon Skylake (server) AVX 512
  - 32 flops/cycle DP & 64 flops/cycle SP
  - Knight's Landing

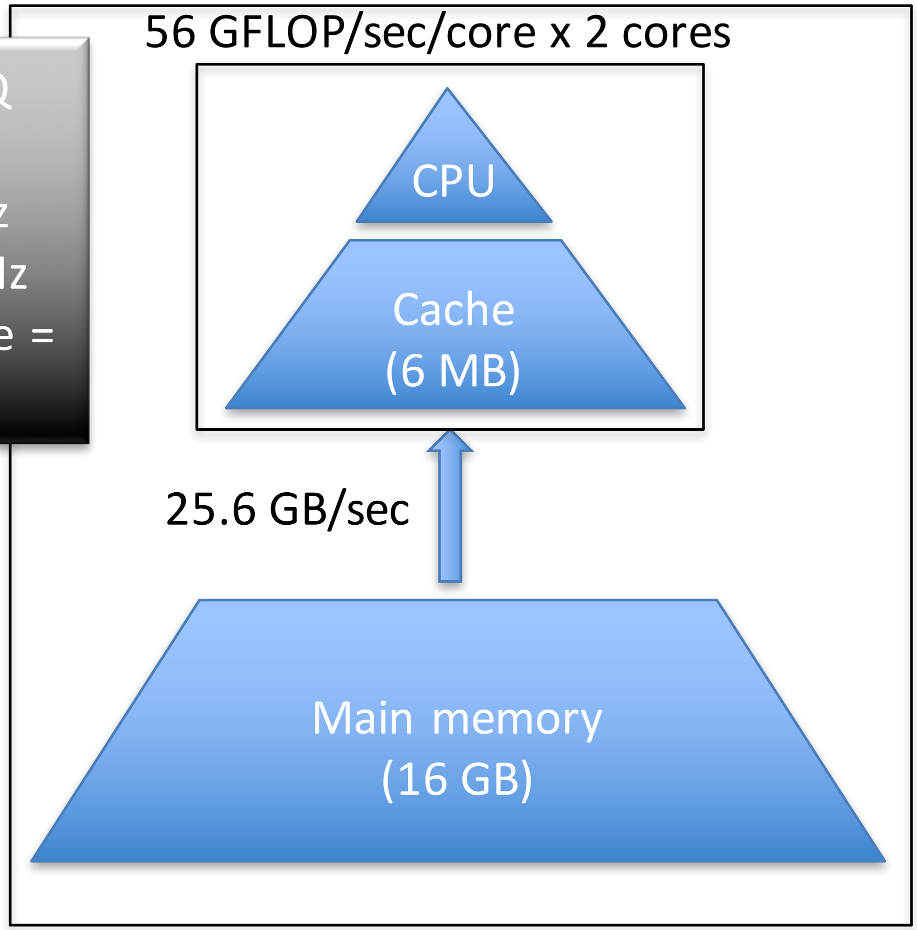
CPU Access Latencies in Clock Cycles



Memory transfer

- One level of memory model on my laptop:

Intel iCore7 4850HQ
Haswell
Cycle time = 2.3 GHz
Turbo Boost = 3.5 GHz
3.5 GHz*16 flops/cycle =
56 Gflop/s per core

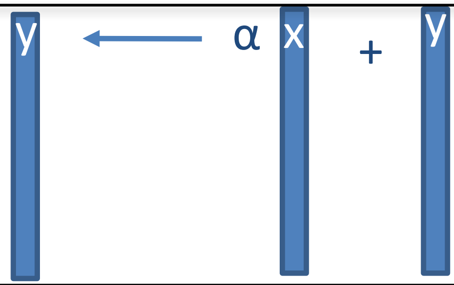


(Omitting latency here.)

The model IS simplified (see next slide) but it provides an upper bound on performance as well. I.e., we will never go faster than what the model predicts. (And, of course, we can go slower …)

FMA: fused multiply-add

AXPY:

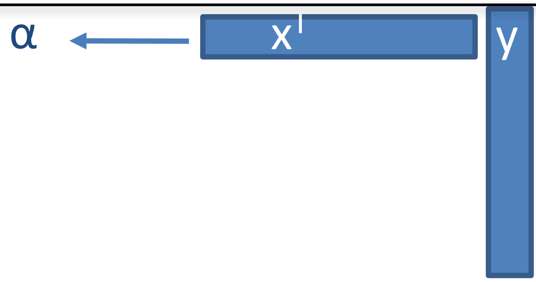



```
for ( j = 0; j < n; j++)
        y[i] += a * x[i];
```

(without increment)

n MUL
n ADD
2n FLOP
n FMA

DOT:




```
alpha = 0e+00;
for ( j = 0; j < n; j++)
        alpha += x[i] * y[i];
```

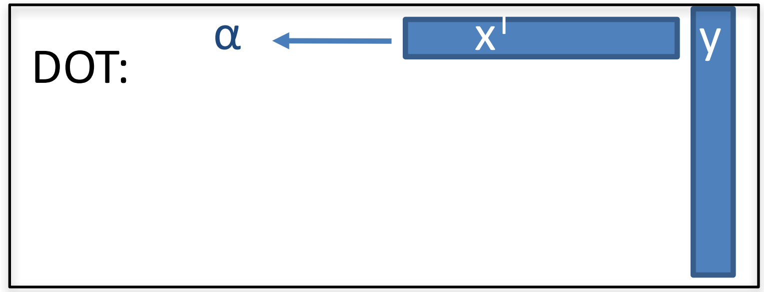
(without increment)

n MUL
n ADD
2n FLOP
n FMA

Note: It is reasonable to expect the one loop codes shown here to perform as well as their Level 1 BLAS counterpart (on multicore with an OpenMP pragma for example).

The true gain these days with using the BLAS is (1) Level 3 BLAS, and (2) portability.

- Take two double precision vectors x and y of size n=375,000.



- Data size:
  - (375,000 double) * (8 Bytes / double) = 3 MBytes per vector

  (Two vectors fit in cache (6 MBytes). OK.)

- Time to move the vectors from memory to cache:
  - (6 MBytes) / (25.6 GBytes/sec) = **0.23 ms**

- Time to perform computation of DOT:
  - (2n flop) / (56 Gflop/sec) = **0.01 ms**

Vector Operations

$$\text{total\_time} \geq \max(\text{time\_comm}, \text{time\_comp})$$
$$= \max(0.23\text{ms}, 0.01\text{ms}) = 0.23\text{ms}$$

Performance = (2 x 375,000 flops)/.23ms = 3.2 Gflop/s

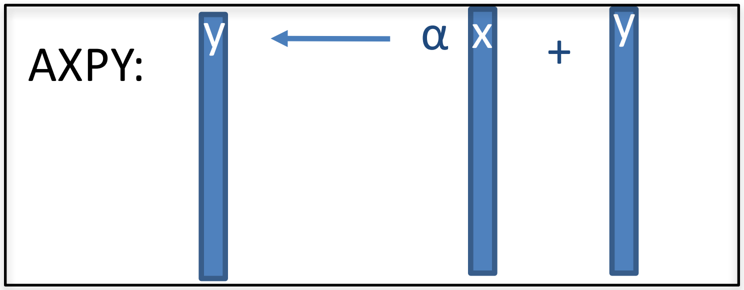
Performance for DOT ≤ 3.2 Gflop/s

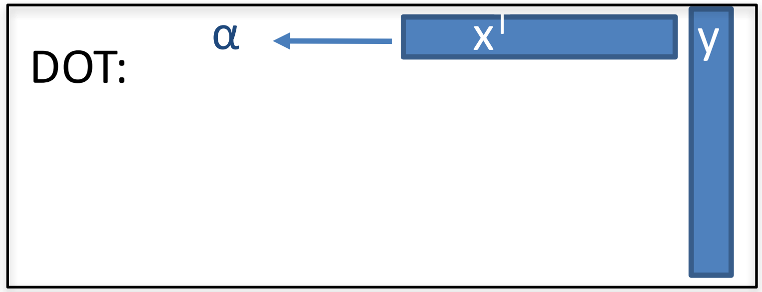
Peak is 56 Gflop/s

We say that the operation is communication bounded. No reuse of data.

Level 1, 2 and 3 BLAS

Level 1 BLAS Matrix-Vector operations

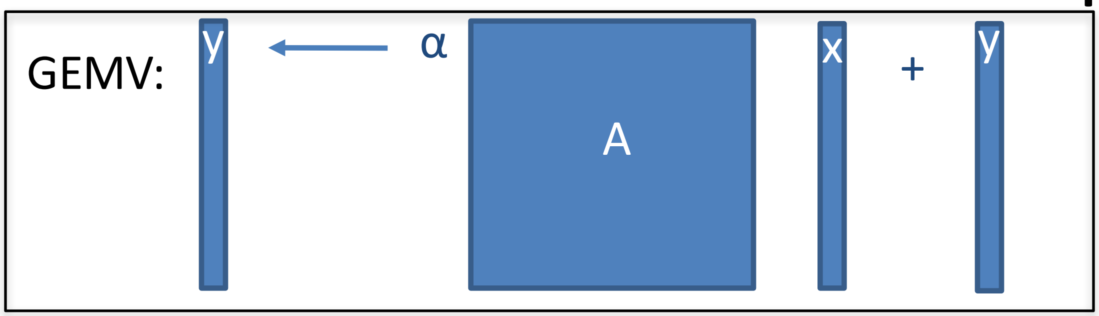




2n FLOP
2n memory reference
AXPY: 2n READ, n WRITE
DOT: 2n READ

RATIO: 1

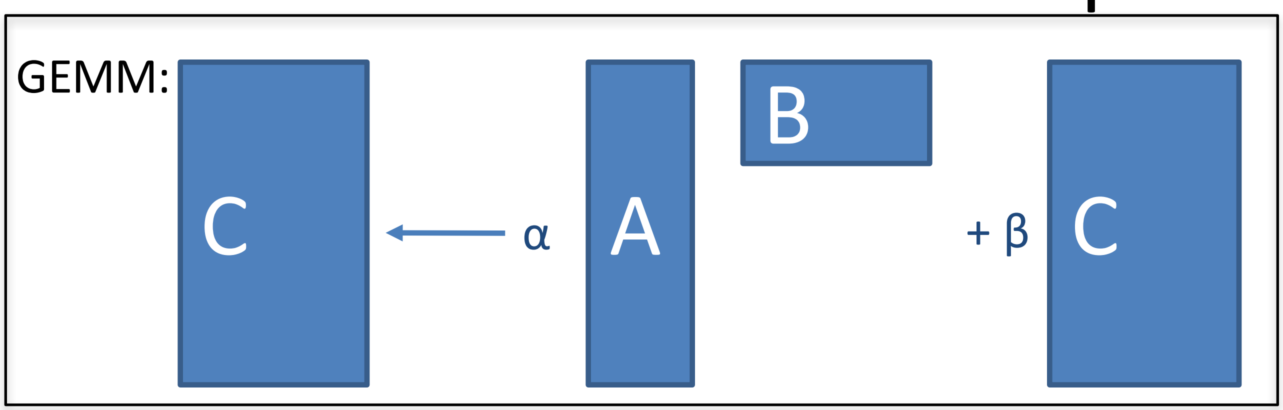
Level 2 BLAS Matrix-Vector operations



$2n^2$ FLOP
n^2 memory references

RATIO: 2

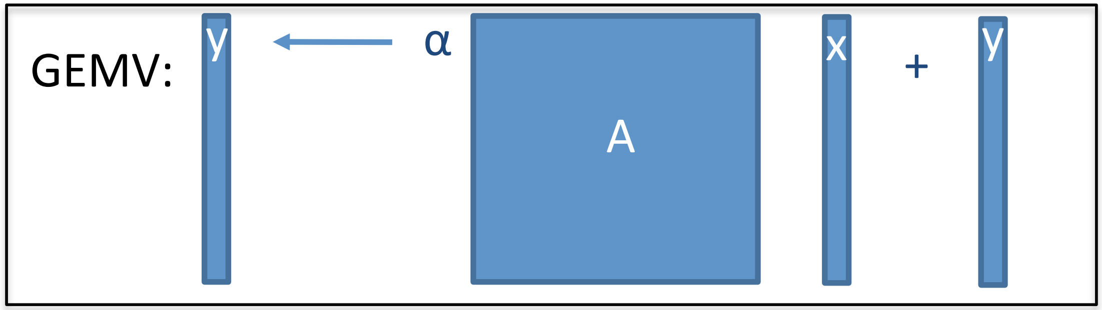
Level 3 BLAS Matrix-Matrix operations



$2n^3$ FLOP
$3n^2$ memory references
$3n^2$ READ, n^2 WRITE

RATIO: 2/3 n

- Double precision matrix A and vectors x and y of size n=860.



- Data size:
  - (860^2 + 2*860 double) * (8 Bytes / double) ~ 6 MBytes

Matrix and two vectors fit in cache (6 MBytes).

- Time to move the data from memory to cache:
  - (6 MBytes) / (25.6 GBytes/sec) = **0.23 ms**

- Time to perform computation of DOT:
  - ($2n^2$ flop) / (56 Gflop/sec) = **0.26 ms**

Matrix - Vector Operations

total_time ≥ max (time_comm , time_comp)

= max (0.23ms , 0.26ms) = 0.26ms

Performance = $(2 \times 860^2$ flops$)/.26$ms = 5.7 Gflop/s

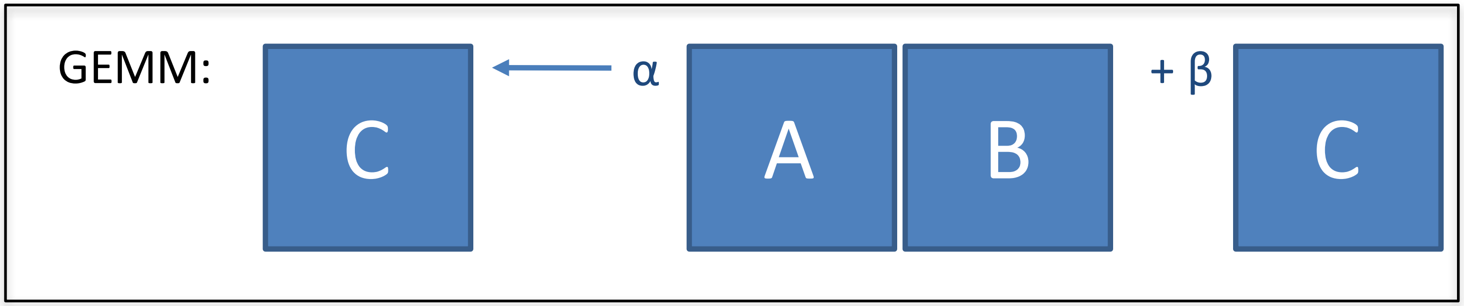
Performance for GEMV ≤ 5.7 Gflop/s

Performance for DOT ≤ 3.2 Gflop/s

Peak is 56 Gflop/s

We say that the operation is communication bounded. Very little reuse of data.

- Take two double precision vectors x and y of size n=500.



- Data size:
  - (500^2 double) * (8 Bytes / double) = 2 MBytes per matrix

  (Three matrices fit in cache (6 MBytes). OK.)

- Time to move the matrices in cache:
  - (6 MBytes) / (25.6 GBytes/sec) = **0.23 ms**

- Time to perform computation in GEMM:
  - ($2n^3$ flop) / (56 Gflop/sec) = **4.46 ms**

Matrix Matrix Operations

total_time ≥ max (time_comm , time_comp)

= max(0.23ms , 4.46ms) = 4.46ms

For this example, communication time is less than 6% of the computation time.

Performance = (2 x 500 ^{3} flops)/4.69ms = 53.3 Gflop/s

There is a lots of data reuse in a GEMM; 2/3n per data element. Has good temporal locality.

If we assume total_time ≈ time_comm +time_comp, we get

Performance for GEMM ≈ 53.3 Gflop/sec

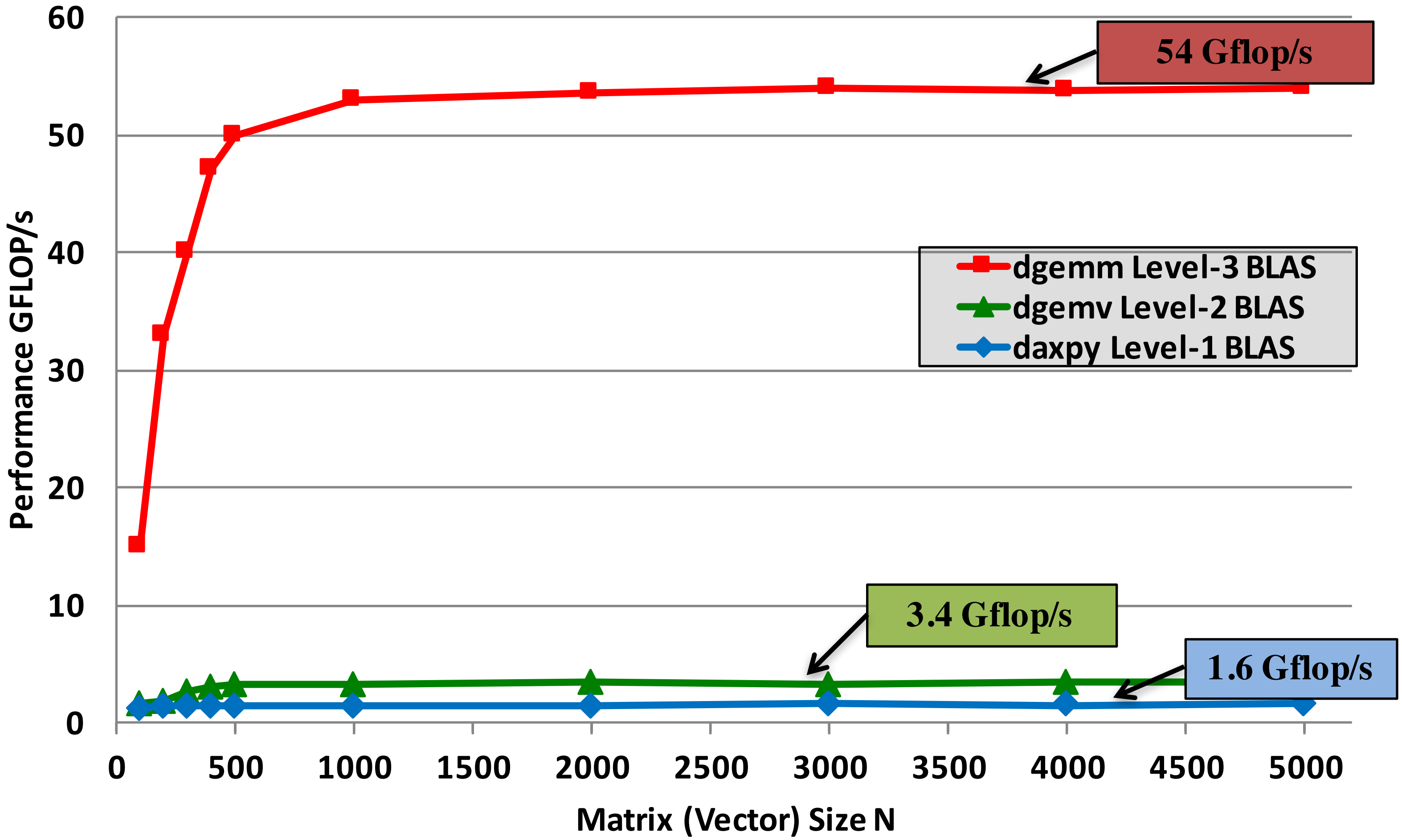
Performance for DOT ≤ 3.2 Gflop/s

Performance for GEMV ≤ 5.7 Gflop/s

(Out of 56 Gflop/sec possible, so that would be 95% peak performance efficiency.)

Level 1, 2 and 3 BLAS

1 core Intel Haswell i7-4850HQ, 2.3 GHz (Turbo Boost at 3.5 GHz);
Peak = 56 Gflop/s



1 core Intel Haswell i7-4850HQ, 2.3 GHz, Memory: DDR3L-1600MHz
6 MB shared L3 cache, and each core has a private 256 KB L2 and 64 KB L1.
The theoretical peak per core double precision is 56 Gflop/s per core.
Compiled with gcc and using Veclib

Issues

- Reuse based on matrices that fit into cache.
- What if you have matrices bigger than cache?

Issues

- Reuse based on matrices that fit into cache.
- What if you have matrices bigger than cache?

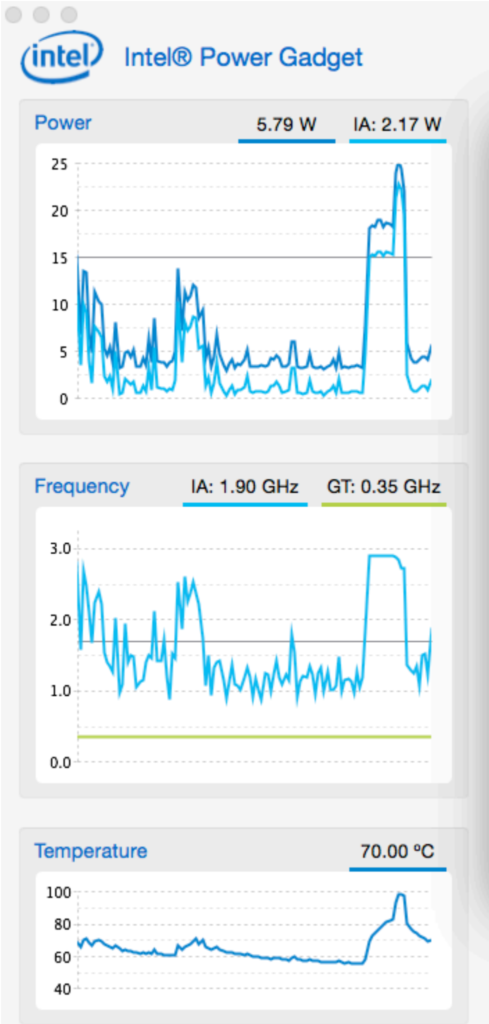
- Break matrices into blocks or tiles that will fit.

$$\begin{bmatrix} C_{11} & C_{12} & C_{13} \\ C_{21} & C_{22} & C_{23} \\ C_{31} & C_{32} & C_{33} \end{bmatrix} \longleftarrow \beta \begin{bmatrix} C_{11} & C_{12} & C_{13} \\ C_{21} & C_{22} & C_{23} \\ C_{31} & C_{32} & C_{33} \end{bmatrix} + \alpha \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} * \begin{bmatrix} B_{11} & B_{12} & B_{13} \\ B_{21} & B_{22} & B_{23} \\ B_{31} & B_{32} & B_{33} \end{bmatrix}$$

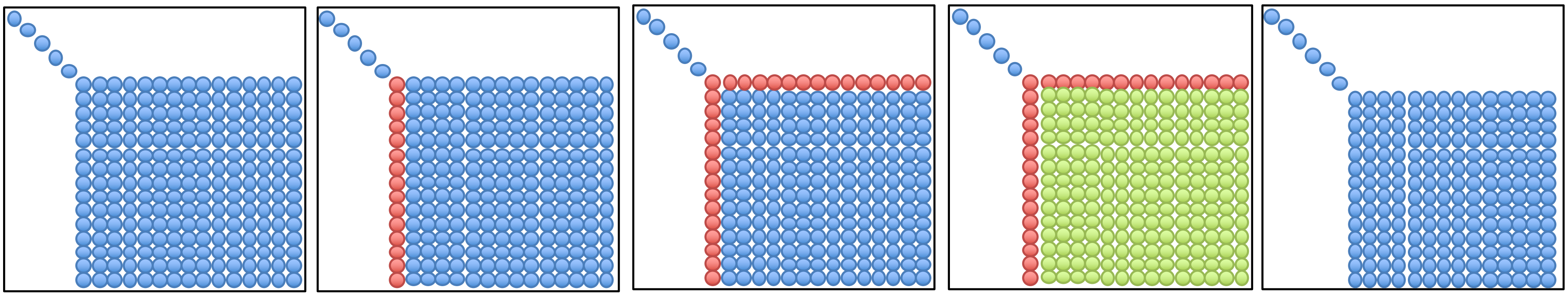
By the way
Performance for your laptop

- If you are interested in running the Linpack Benchmark on your system see:https://software.intel.com/en-us/node/157667?wapkw=mkl+linpack
- Also Intel has a power meter, see:

  https://software.intel.com/en-us/articles/intel-power-gadget-20



The Standard LU Factorization LINPACK 1970's HPC of the Day: Vector Architecture



Factor column with Level 1 BLAS

Divide by Pivot row

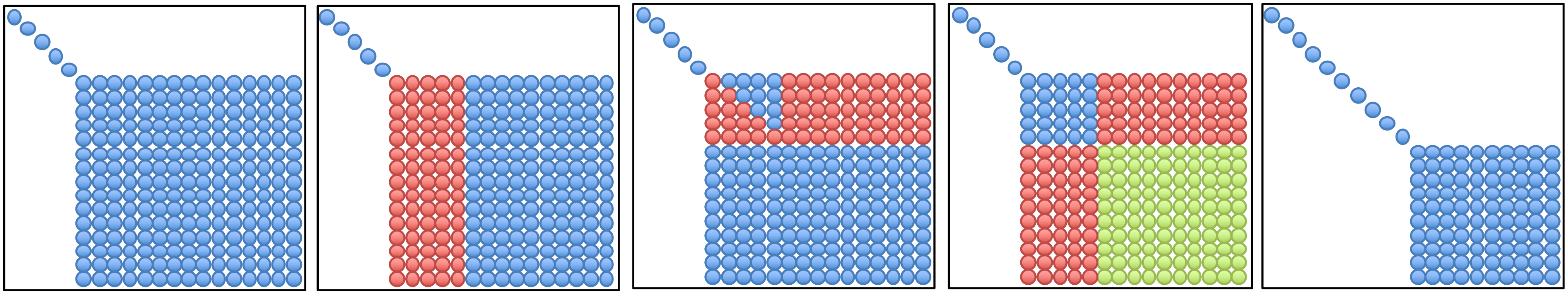
Schur complement update (Rank 1 update)

Next Step

Main points

- Factorization column (zero) mostly sequential due to memory bottleneck
- Level 1 BLAS
- Divide pivot row has little parallelism
- Rank -1 Schur complement update is the only easy parallelize task
- Partial pivoting complicates things even further
- Bulk synchronous parallelism (fork-join)
  - Load imbalance
  - Non-trivial Amdahl fraction in the panel
  - Potential workaround (look-ahead) has complicated implementation

The Standard LU Factorization LAPACK 1980's HPC of the Day: Cache Based SMP



Factor panel with Level 1,2 BLAS

Triangular update

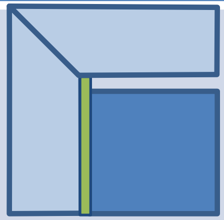
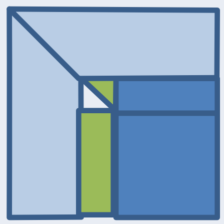
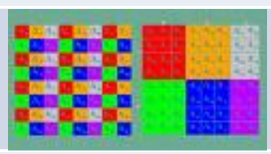
Schur complement update

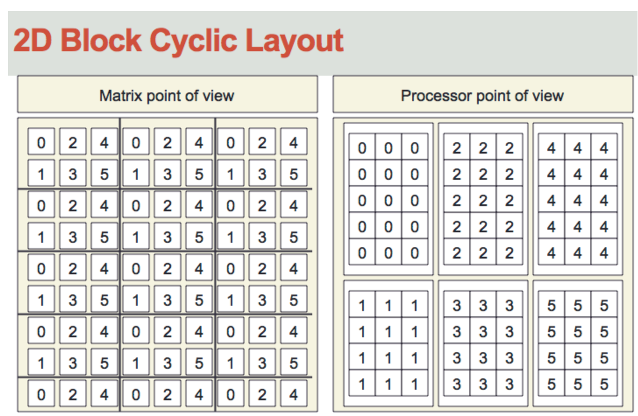
Next Step

Main points

- Panel factorization mostly sequential due to memory bottleneck
- Triangular solve has little parallelism
- Schur complement update is the only easy parallelize task
- Partial pivoting complicates things even further
- Bulk synchronous parallelism (fork-join)
    - Load imbalance
    - Non-trivial Amdahl fraction in the panel
    - Potential workaround (look-ahead) has complicated implementation

Last Generations of DLA Software

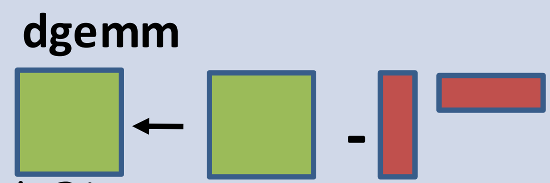
Software/Algorithms follow hardware evolution in time		
LINPACK (70's) (Vector operations)		Rely on - Level-1 BLAS operations
LAPACK (80's) (Blocking, cache friendly)		Rely on - Level-3 BLAS operations
ScaLAPACK (90's) (Distributed Memory)		Rely on - PBLAS Mess Passing

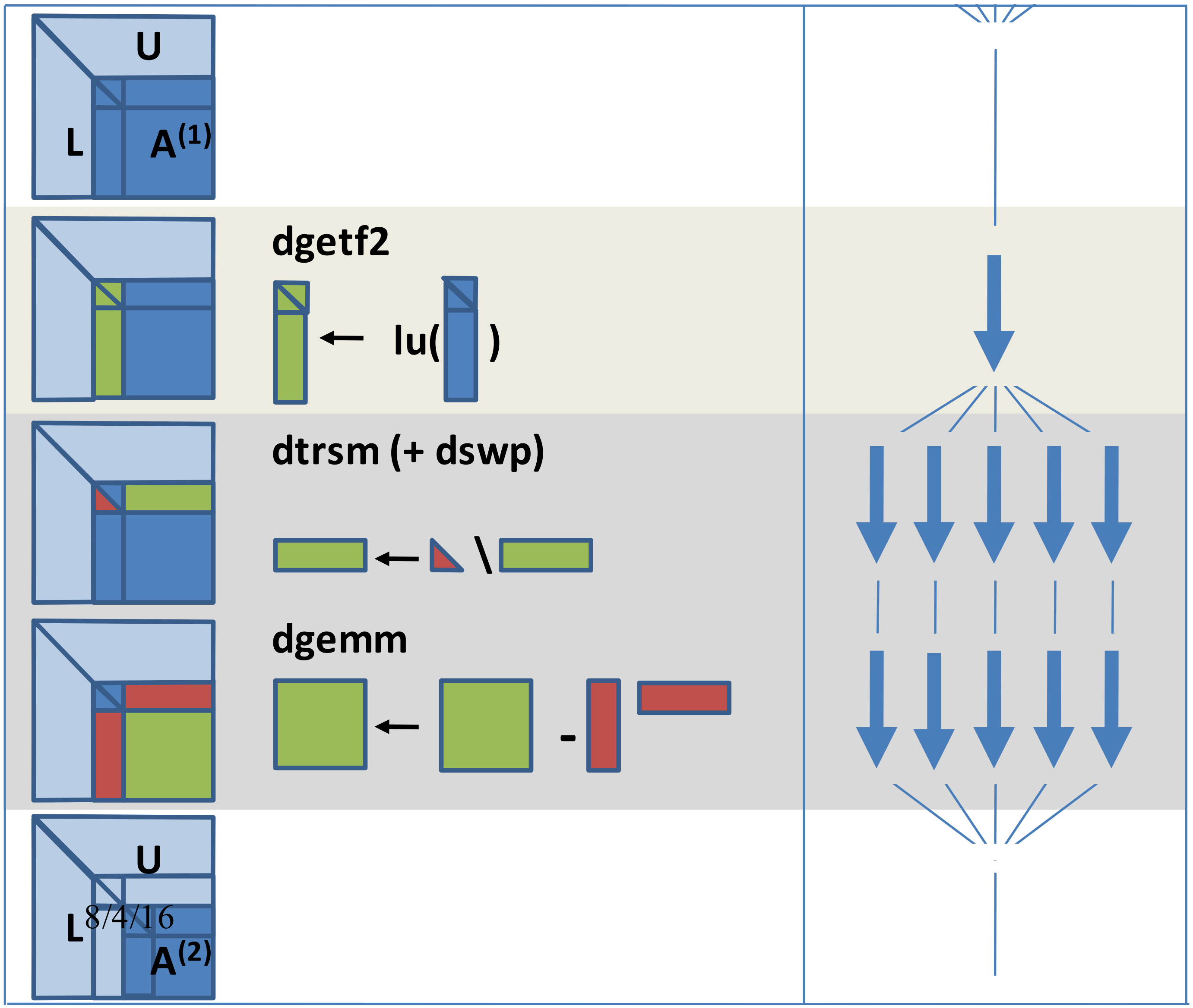


Parallelization of LU and QR.

Parallelize the update:

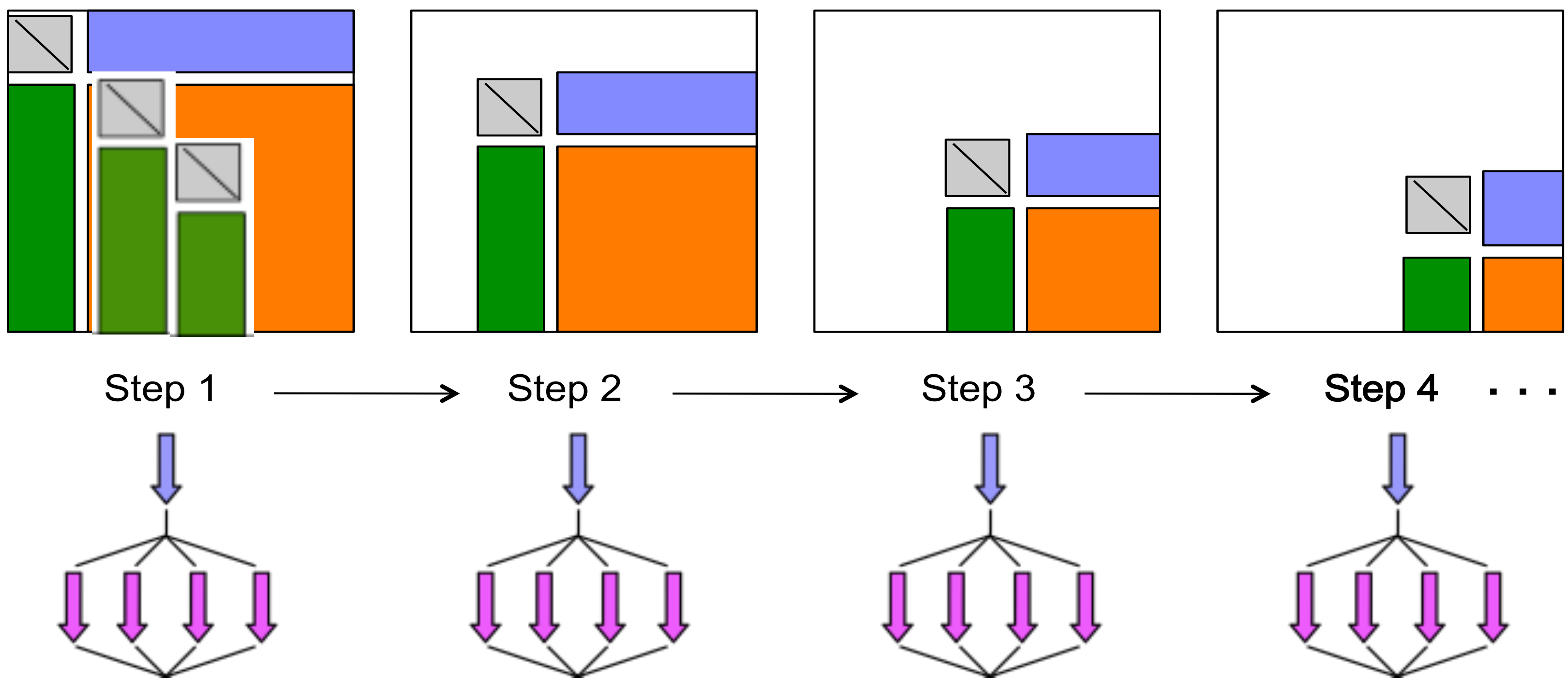
- Easy and done in any reasonable software.
- This is the $2/3n^3$ term in the FLOPs count.
- Can be done efficiently with LAPACK+multithreaded BLAS

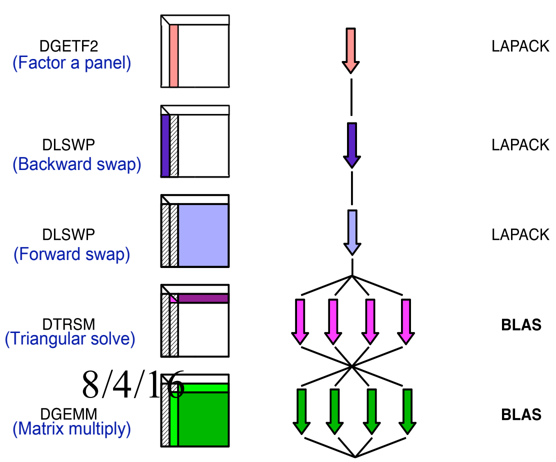




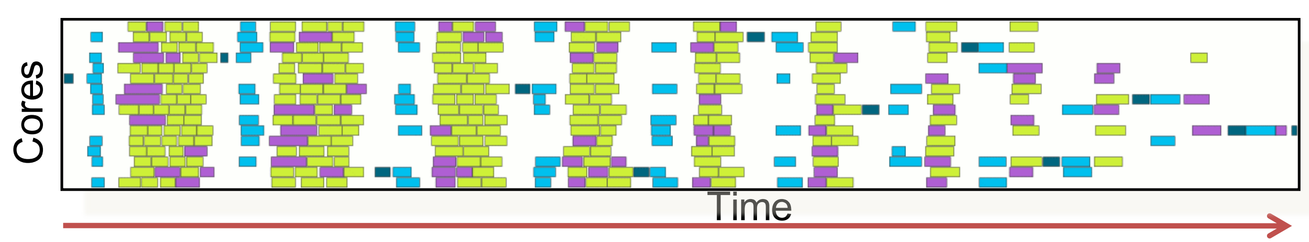
Fork - Join parallelism
Bulk Sync Processing

Synchronization (in LAPACK LU)





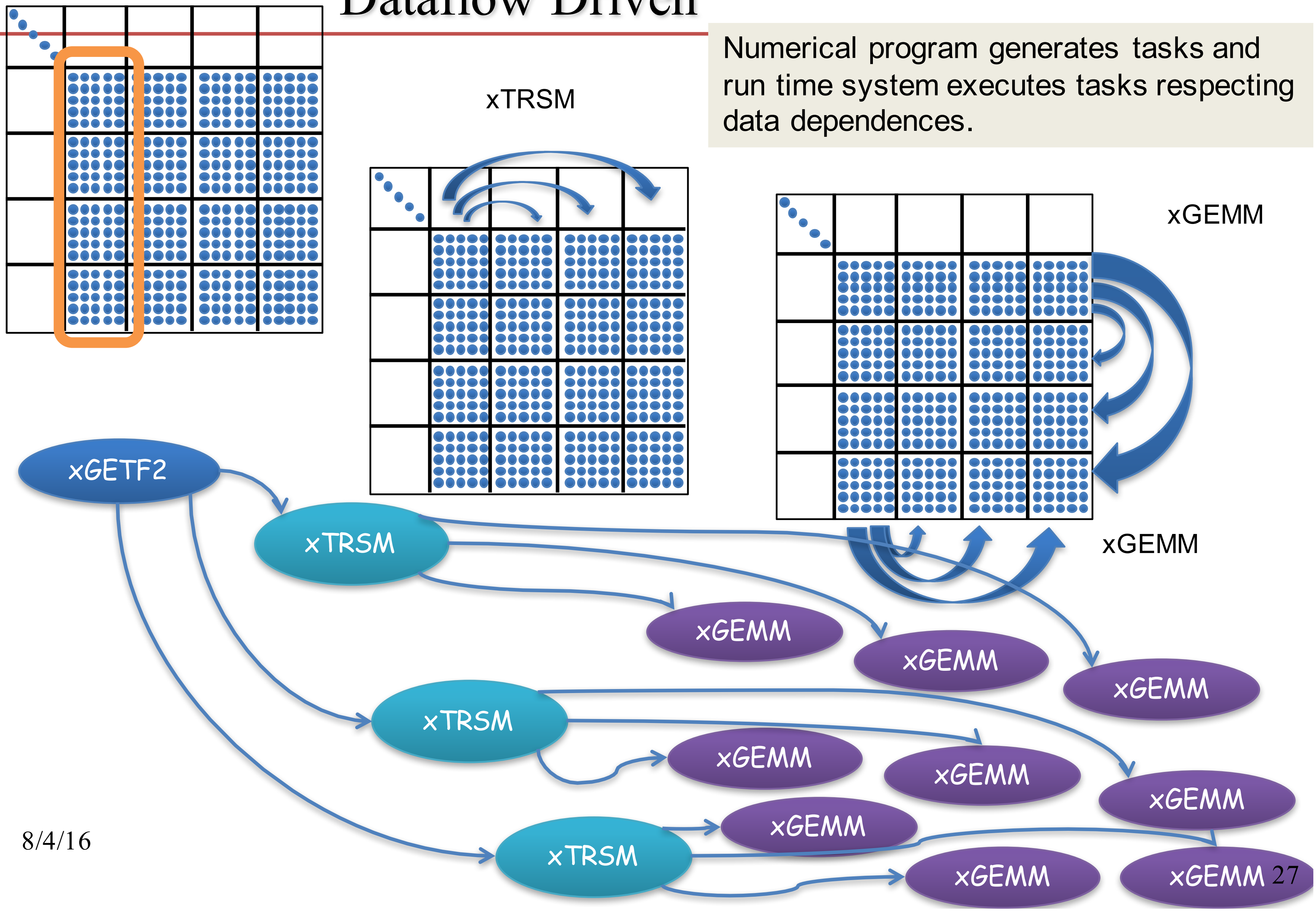
- fork join
- bulk synchronous processing



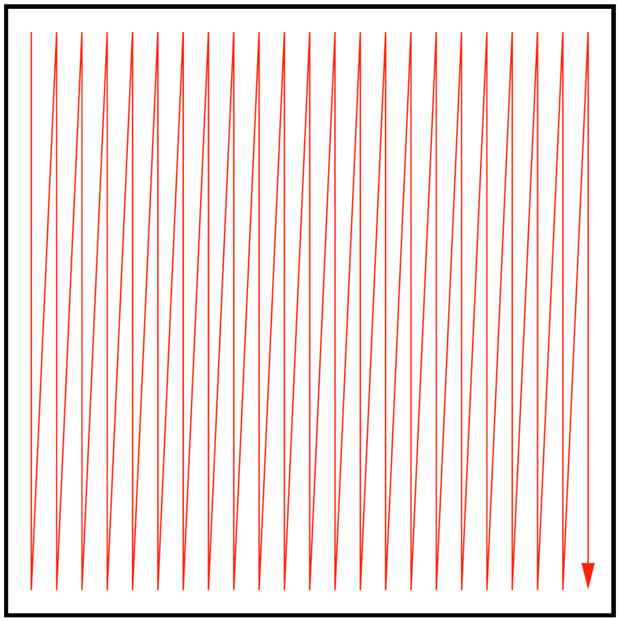
PLASMA LU Factorization

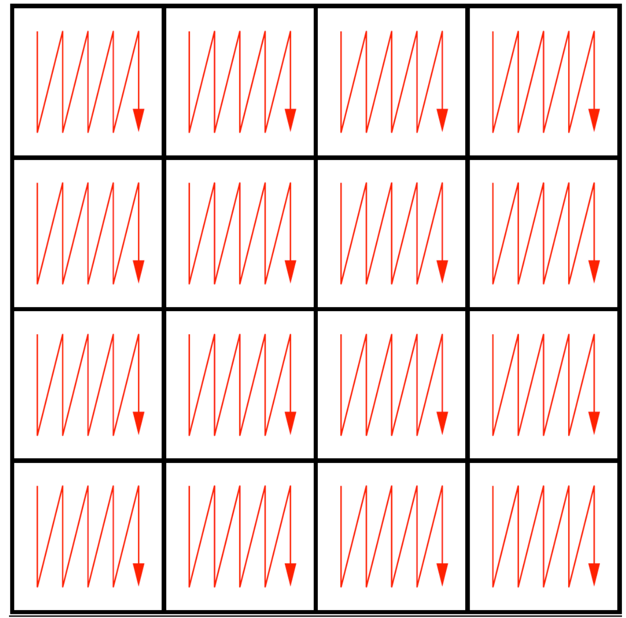
Dataflow Driven

Numerical program generates tasks and run time system executes tasks respecting data dependences.



Data Layout is Critical

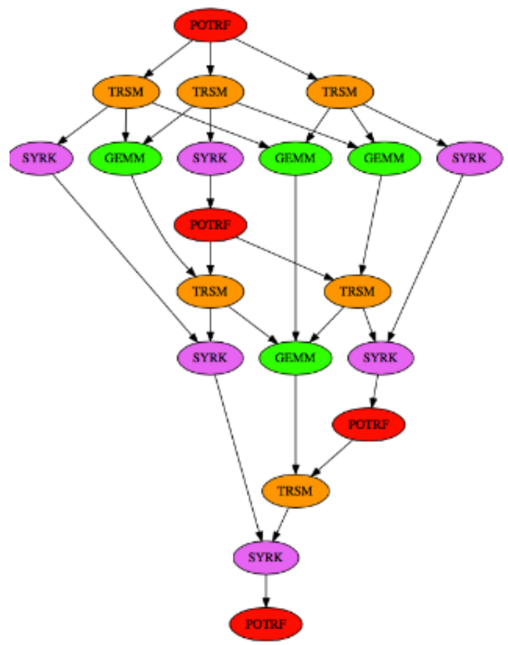




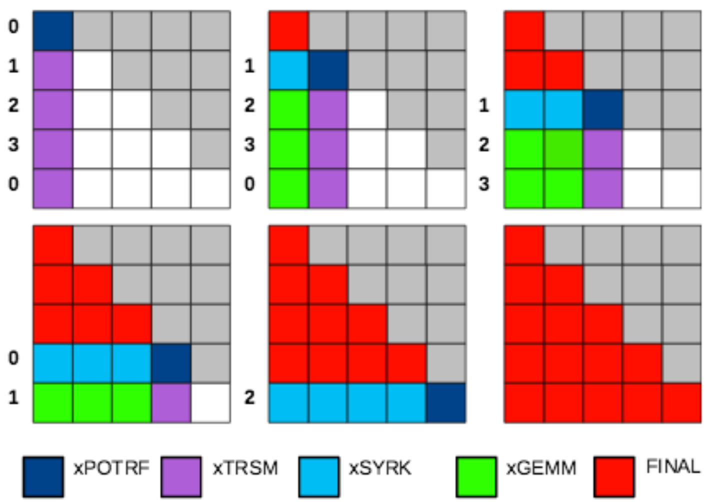
- Tile data layout where each data tile is contiguous in memory
- Decomposed into several fine-grained tasks, which better fit the memory of the small core caches

OpenMP tasking

- Added with OpenMP 3.0 (2009)
- Allows parallelization of irregular problems
- OpenMP 4.0 (2013) - Tasks can have dependencies
  - DAGs



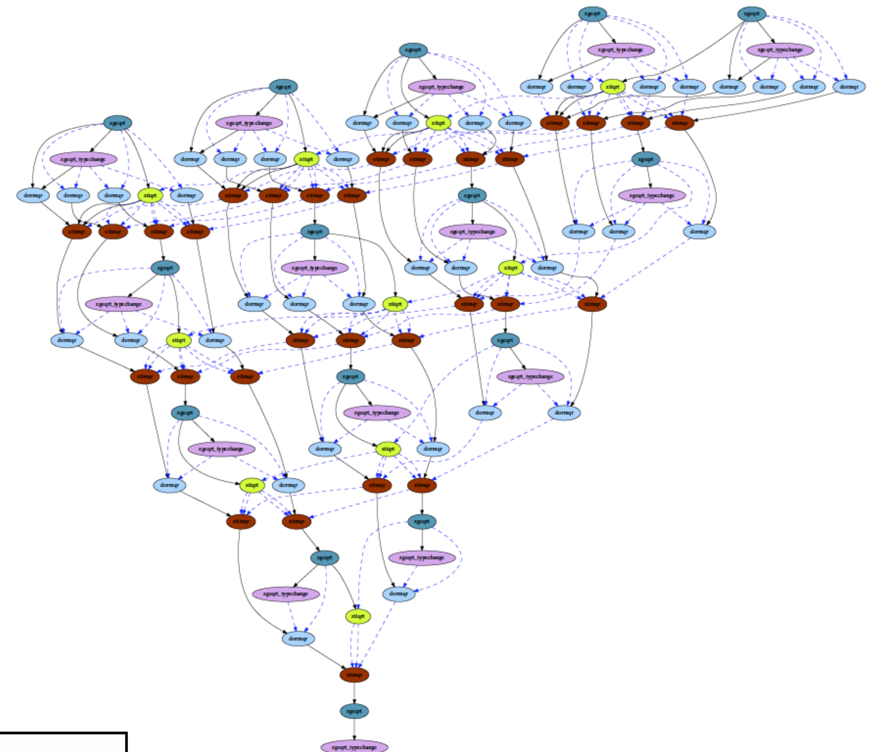
Tiled Cholesky Decomposition

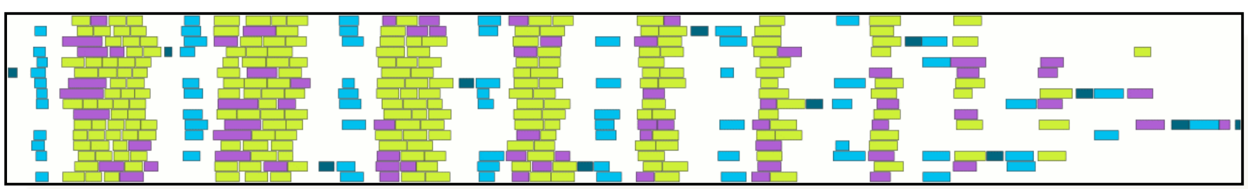



```
#pragma omp parallel
#pragma omp master
{  CHOLESKY( A );   }
CHOLESKY( A ) {
    for (k = 0; k < M; k++) {
        #pragma omp task depend(inout:A(k,k)[0:tilesize]
        {  POTRF( A(k,k) );   }
        for (m = k+1; m < M; m++) {
            #pragma omp task \
                depend(in:A(k,k)[0:tilesize]) \
                depend(inout:A(m,k)[0:tilesize])
            { TRSM( A(k,k), A(m,k) ); }
        }
        for (m = k+1; m < M; m++) {
            #pragma omp task \
                depend(in:A(m,k)[0:tilesize]) \
                depend(inout:A(m,m)[0:tilesize])
            { SYRK( A(m,k), A(m,m) ); }
            for (n = k+1; n < m; n++) {
                #pragma omp task \
                    depend(in:A(m,k)[0:tilesize], \
                              A(n,k)[0:tilesize]) \
                    depend(inout:A(m,n)[0:tilesize])
                {  GEMM( A(m,k), A(n,k), A(m,n) ); }
            }
        }
    }
}
```

Dataflow Based Design

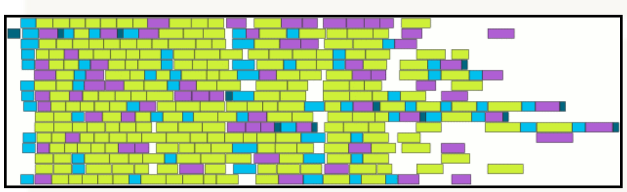
- **Objectives**
  - High utilization of each core
  - Scaling to large number of cores
  - Synchronization reducing algorithms
- **Methodology**
  - Dynamic DAG scheduling
  - Explicit parallelism
  - Implicit communication
  - Fine granularity / block data layout
- **Arbitrary DAG with dynamic scheduling**





Fork-join parallelism
Notice the synchronization penalty in the presence of heterogeneity.

Cores

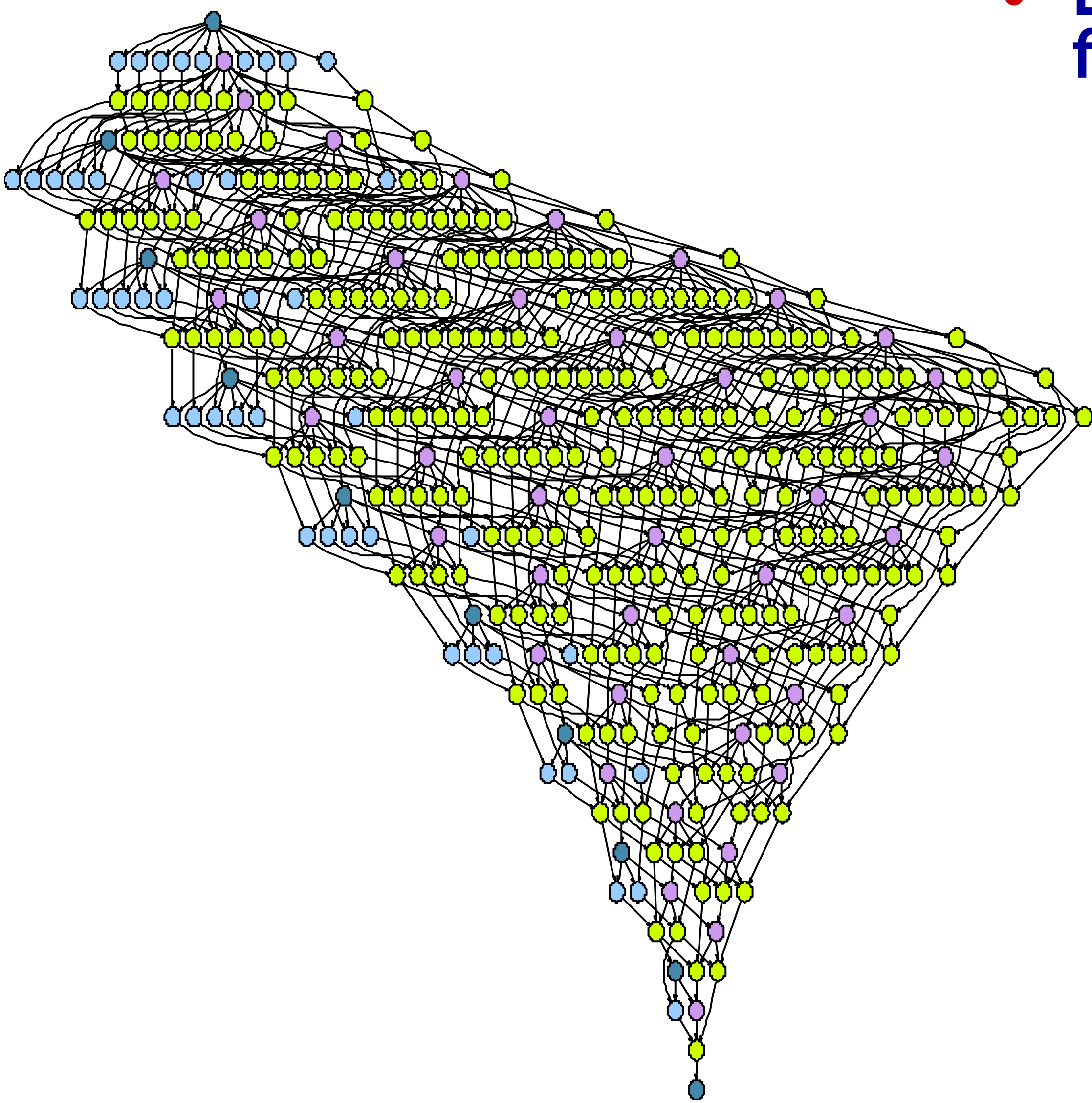


DAG scheduled parallelism

Time

PLASMA Local Scheduling

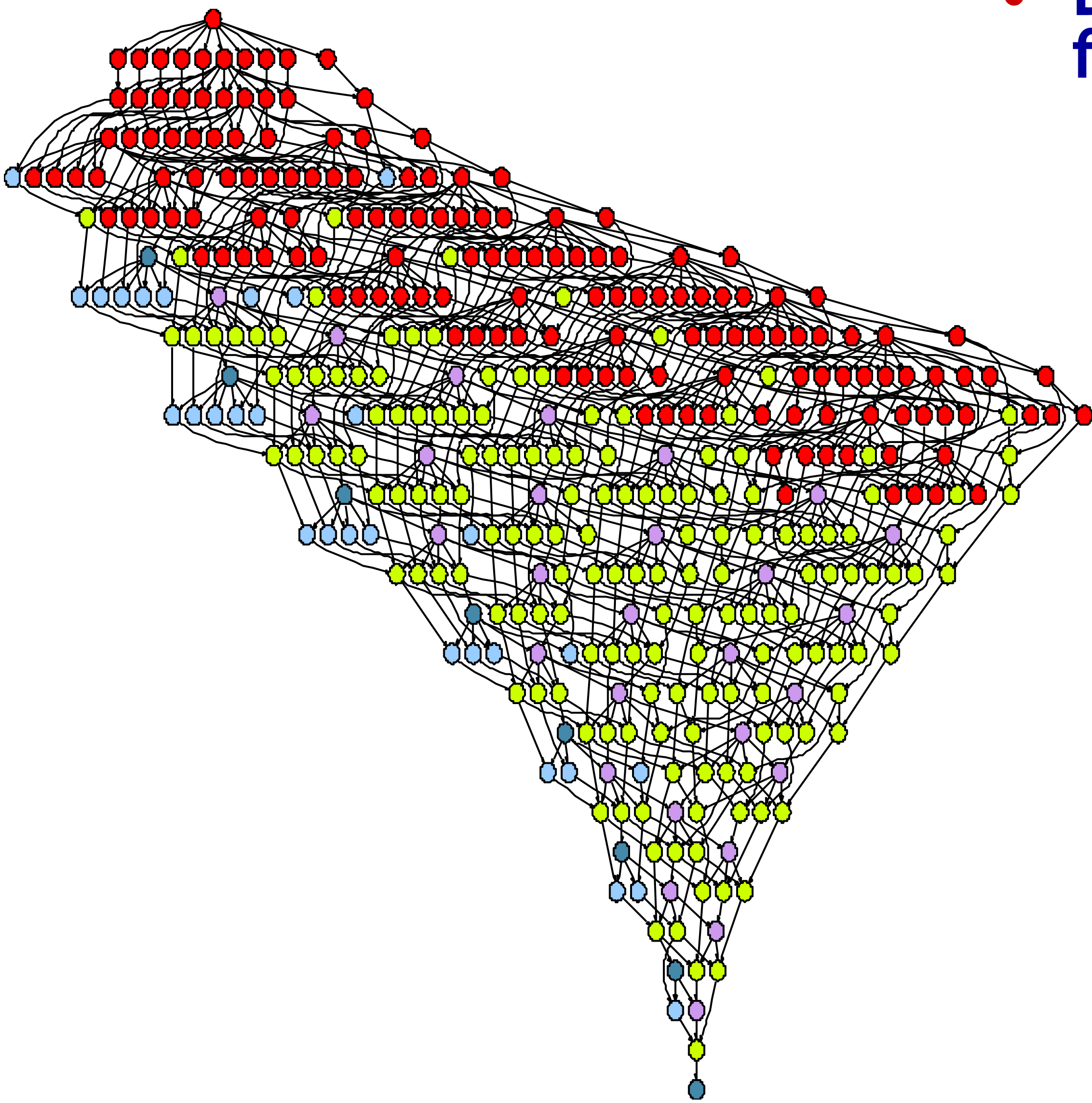
Dynamic Scheduling: Sliding Window



- **DAGs get very big, very fast**
  - **So windows of active tasks are used; this means no global critical path**
  - **Matrix of NBxNB tiles; NB^3 operation**
    - **NB=100 gives 1 million tasks**

PLASMA Local Scheduling

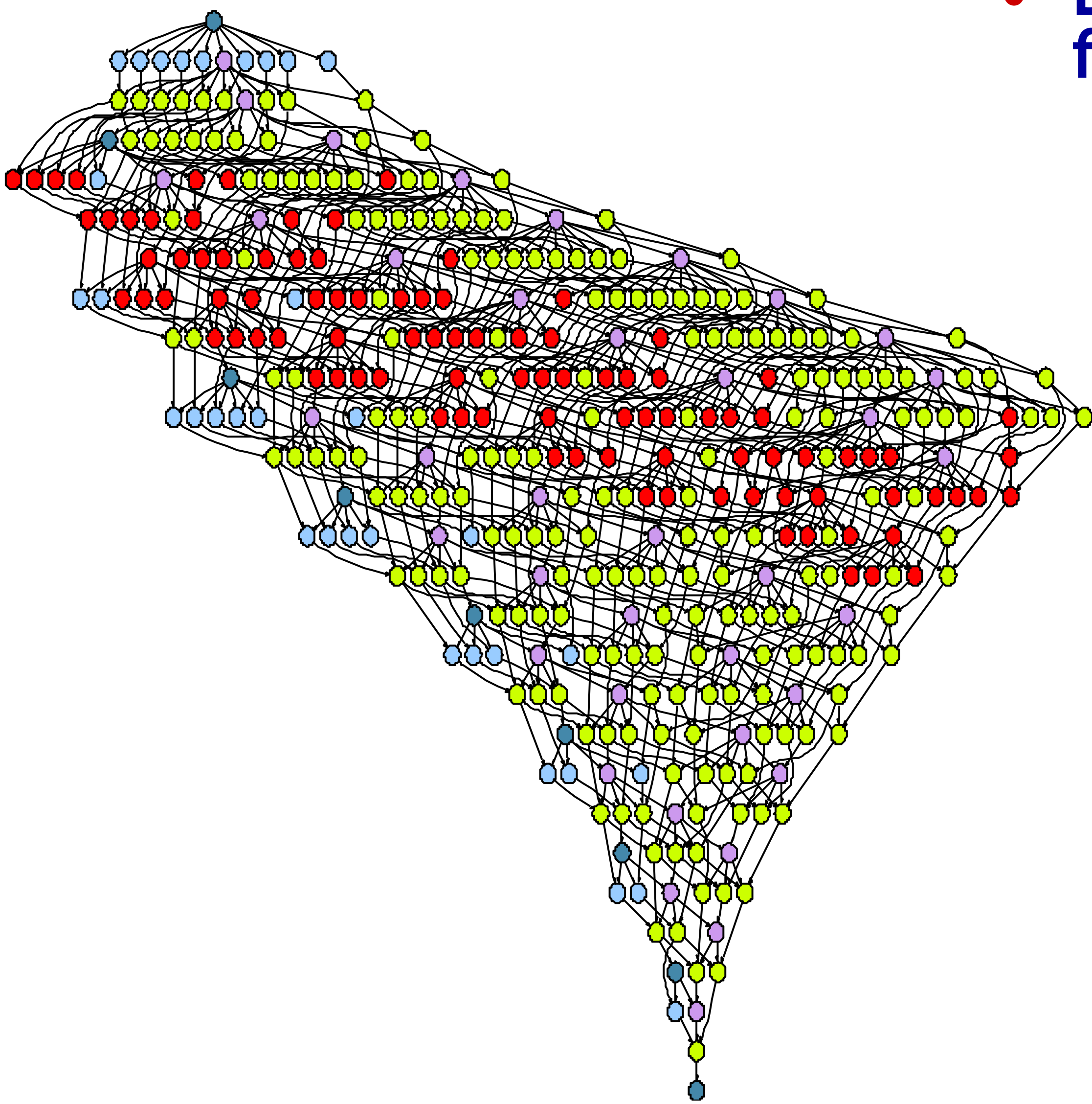
Dynamic Scheduling: Sliding Window



- **DAGs get very big, very fast**
  - So windows of active tasks are used; this means no global critical path
  - Matrix of NBxNB tiles; NB^3 operation
    - NB=100 gives 1 million tasks

PLASMA Local Scheduling

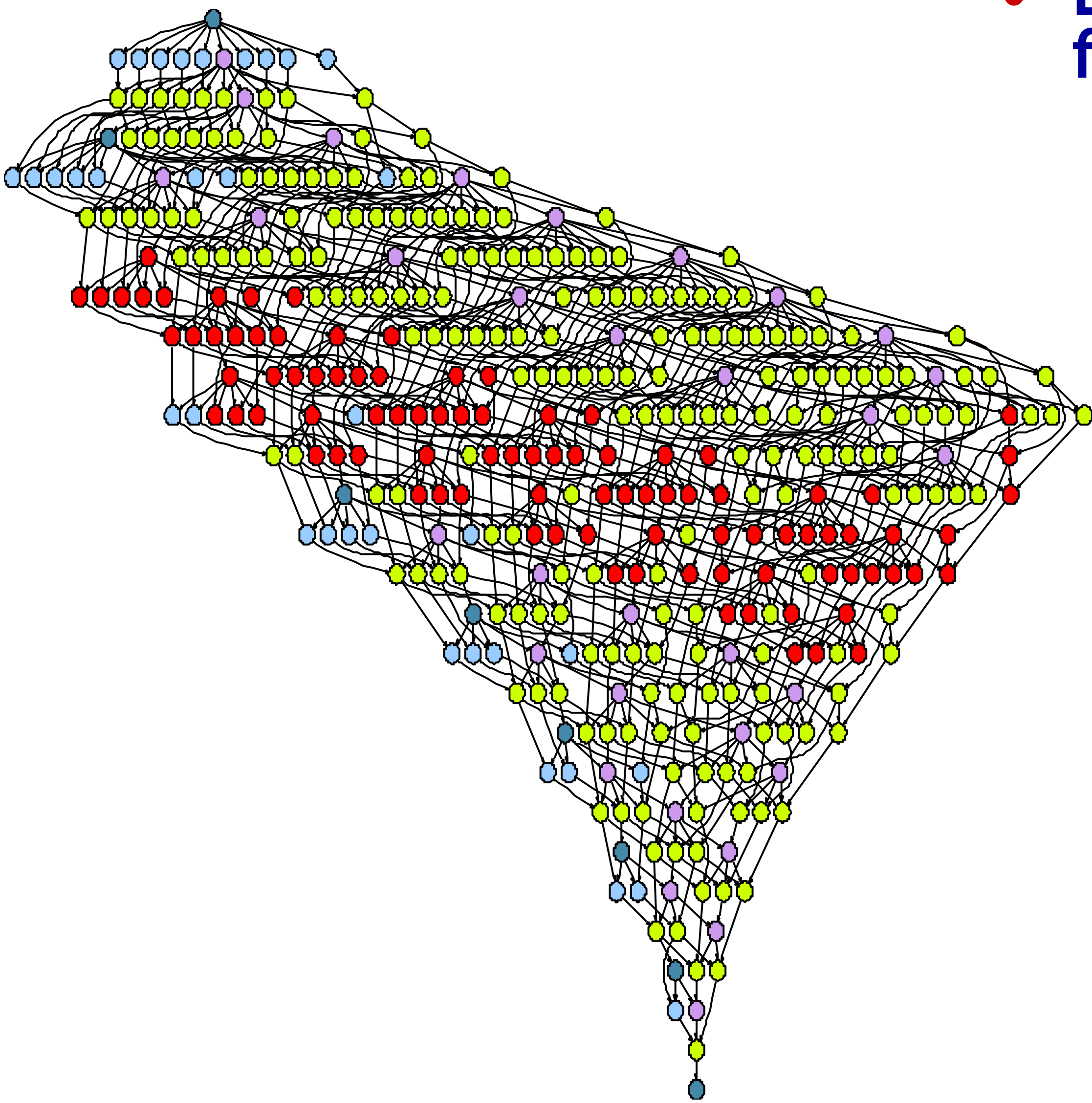
Dynamic Scheduling: Sliding Window



- **DAGs get very big, very fast**
  - **So windows of active tasks are used; this means no global critical path**
  - **Matrix of NBxNB tiles; NB^3 operation**
    - **NB=100 gives 1 million tasks**

PLASMA Local Scheduling

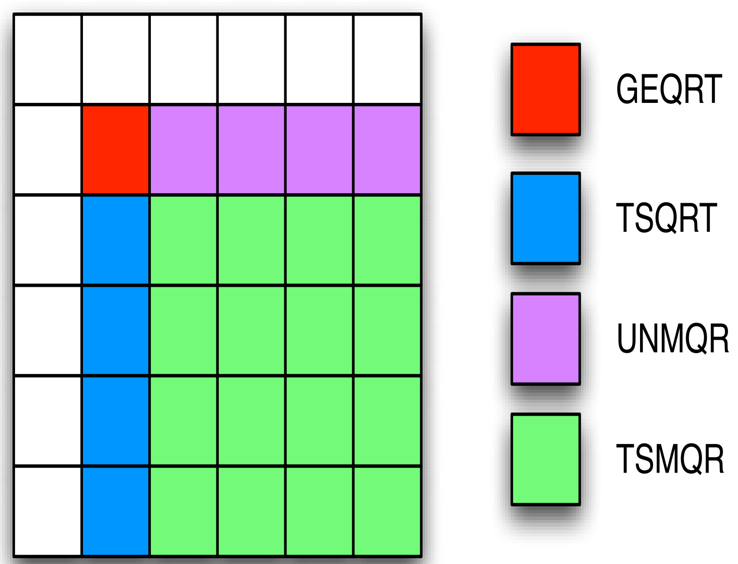
Dynamic Scheduling: Sliding Window



- **DAGs get very big, very fast**
  - **So windows of active tasks are used; this means no global critical path**
  - **Matrix of NBxNB tiles; NB^3 operation**
    - **NB=100 gives 1 million tasks**

Example: QR Factorization

```
FOR k = 0 .. SIZE - 1

    A[k][k], T[k][k]  <-  GEQRT( A[k][k] )

    FOR m = k+1 .. SIZE - 1

        A[k][k]|Up, A[m][k], T[m][k]  <-
            TSQRT( A[k][k]|Up, A[m][k], T[m][k] )

    FOR n = k+1 .. SIZE - 1

        A[k][n] <- UNMQR( A[k][k]|Low, T[k][k], A[k][n] )

        FOR m = k+1 .. SIZE - 1

            A[k][n], A[m][n] <-
                TSMQR( A[m][k], T[m][k], A[k][n], A[m][n] )
```



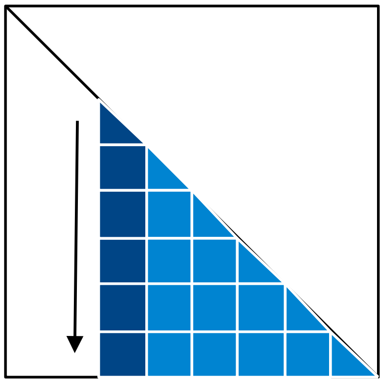
Input Format – Quark (PLASMA)

```
for (k = 0; k < A.mt; k++) {
 Insert_Task( zgeqrt,  A[k][k], INOUT,
                  T[k][k], OUTPUT);
 for (m = k+1; m < A.mt; m++) {
  Insert_Task( ztsqrt,  A[k][k], INOUT | REGION_D|REGION_U,
                  A[m][k], INOUT | LOCALITY,
                  T[m][k], OUTPUT);
 }
 for (n = k+1; n < A.nt; n++) {
  Insert_Task( zunmqr,  A[k][k], INPUT | REGION_L,
                  T[k][k], INPUT,
                  A[k][m], INOUT);
  for (m = k+1; m < A.mt; m++) {
   Insert_Task( ztsmqr, A[k][n], INOUT,
                  A[m][n], INOUT | LOCALITY,
                  A[m][k], INPUT,
                  T[m][k], INPUT);
  }
 }
}
```

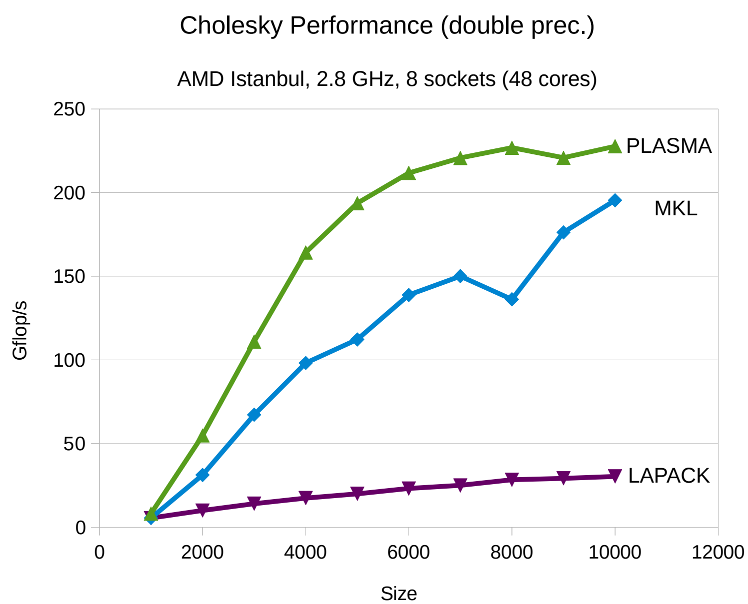
- Sequential C code
- Annotated through QUARK-specific syntax
  - Insert_Task
  - INOUT, OUTPUT, INPUT
  - REGION_L, REGION_U, REGION_D, …
  - LOCALITY
- Executes thru the QUARK RT to run on multicore SMPs

Algorithms
Cholesky

PLASMA_[scdz]**potrf**[_Tile][_Async]()



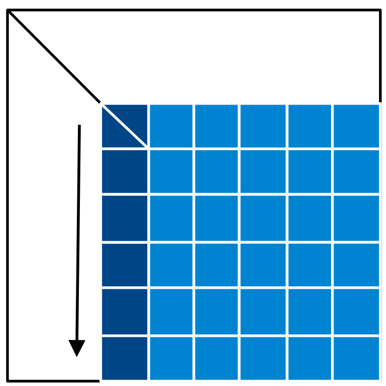
- **Algorithm**
  - equivalent to LAPACK
- **Numerics**
  - same as LAPACK
- **Performance**
  - comparable to vendor on few cores
  - much better than vendor on many cores



Algorithms

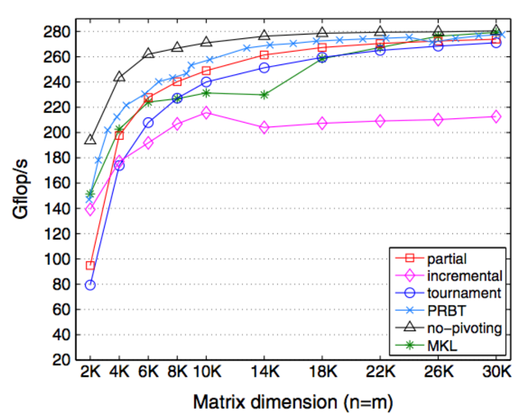
LU

PLASMA_[scdz]**getrf**[_Tile][_Async]()

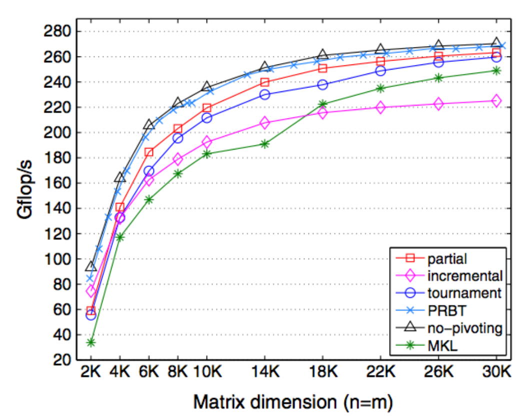


- **<u>Algorithm</u>**
  - equivalent to LAPACK
  - same pivot vector
  - same L and U factors
  - same forward substitution procedure
- **<u>Numerics</u>**
  - same as LAPACK
- **<u>Performance</u>**
  - comparable to vendor on few cores
  - much better than vendor on many cores

16 Sandy Bridge cores



Factorization alone, using16 cores

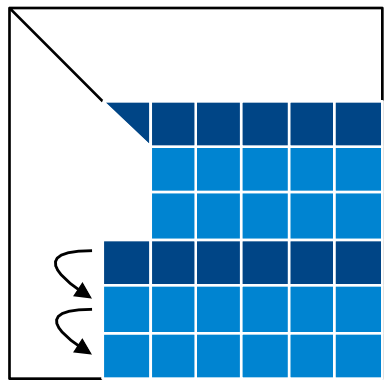


Factorization and solve with iterative refinement, using 16 cores

Algorithms

incremental QR Factorization

PLASMA_[scdz]**geqrt**[_Tile][_Async]()



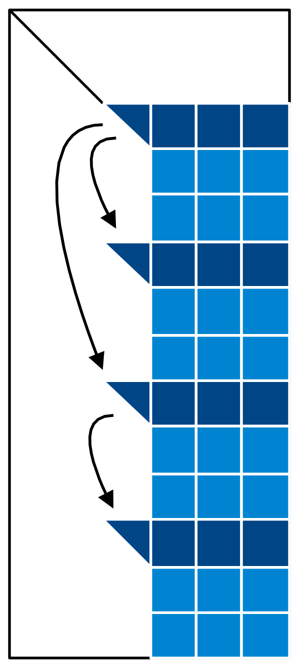
- **Algorithm**
  - the same R factor as LAPACK (absolute values)
  - different set of Householder reflectors
  - different Q matrix
  - different Q generation / application procedure
- **Numerics**
  - same as LAPACK
- **Performance**
  - comparable to vendor on few cores
  - much better than vendor on many cores

Algorithms

incremental QR Factorization (Communication Avoiding)

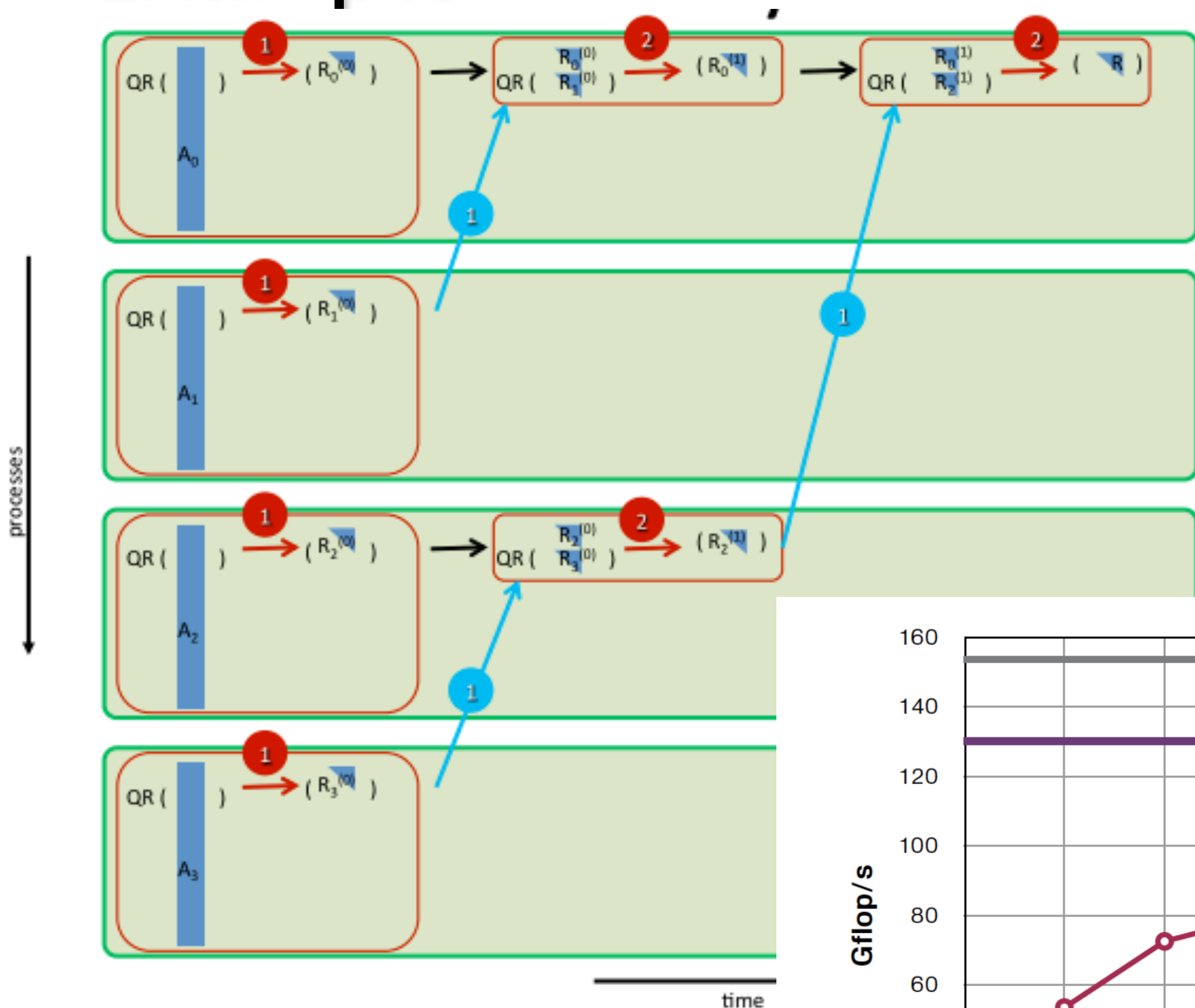
PLASMA_[scdz]**geqrt**[_Tile][_Async]()

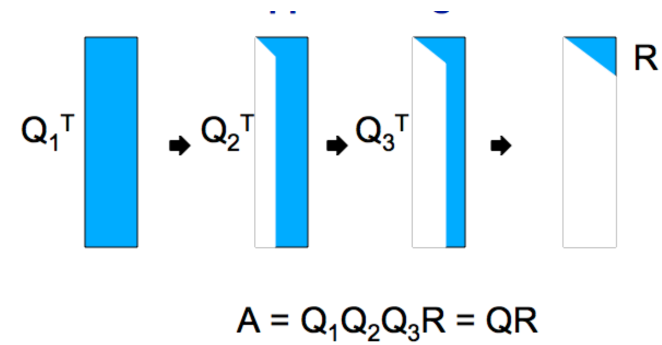
```
PLASMA_Set(
   PLASMA_HOUSEHOLDER_MODE,
   PLASMA_TREE_HOUSEHOLDER);
```



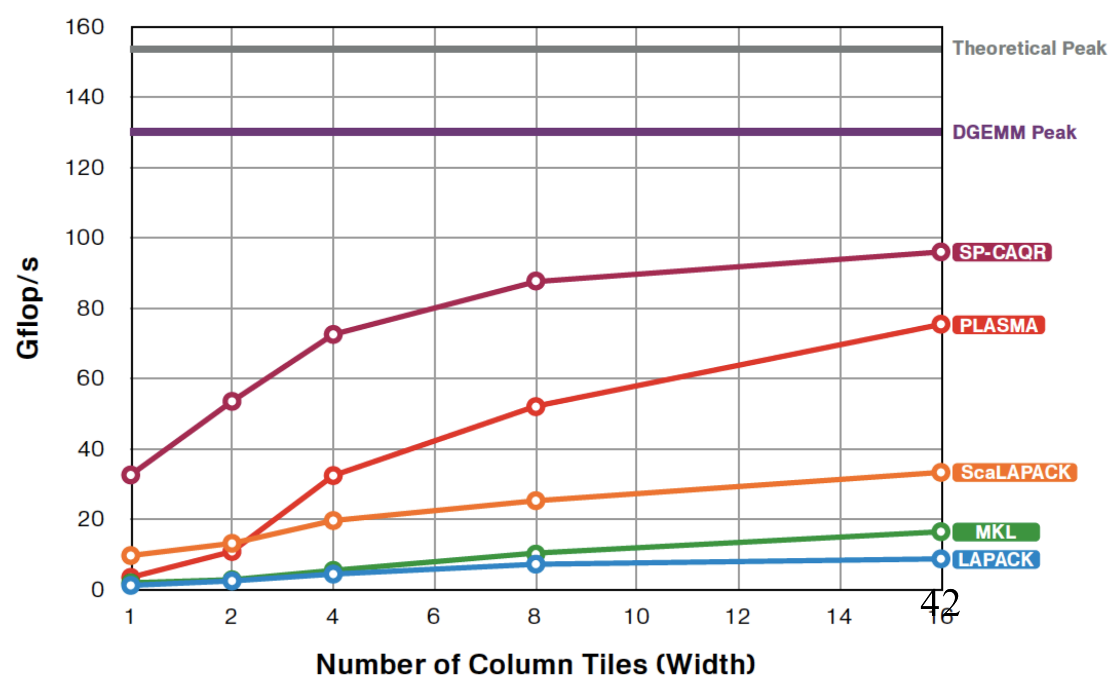
- **Algorithm**
  - the same R factor as LAPACK (absolute values)
  - different set of Householder reflectors
  - different Q matrix
  - different Q generation / application procedure
- **Numerics**
  - same as LAPACK
- **Performance**
  - absolutely superior for tall matrices

Communication Avoiding QR Example





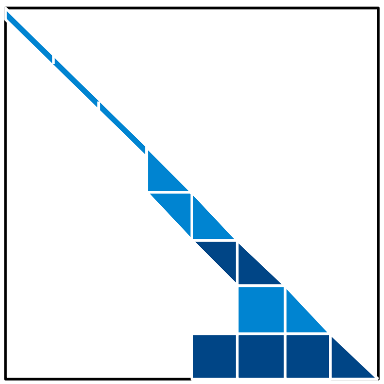
Quad-socket, quad-core machine Intel Xeon EMT64 E7340 at 2.39 GHz.
Theoretical peak is 153.2 Gflop/s with 16 cores.
Matrix size 51200 by 3200



Algorithms

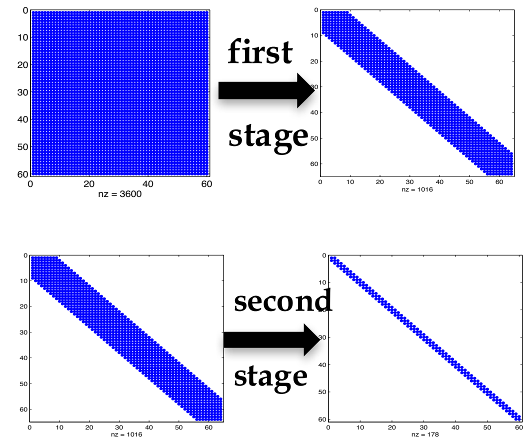
three-stage symmetric EVP

PLASMA_[scdz]**syev**[_Tile][_Async]()

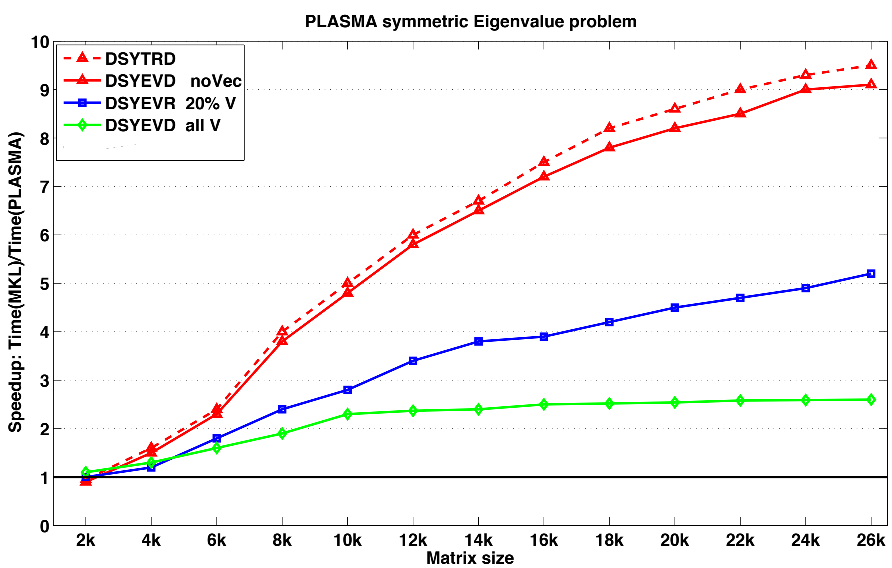


- **Algorithm**
  - two-stage tridiagonal reduction + QR Algorithm
  - fast eigenvalues, slower eigenvectors

  (possibility to calculate a subset)



- **Numerics**
  - same as LAPACK
- **Performance**
  - comparable to MKL for very small problems
  - absolutely superior for larger problems

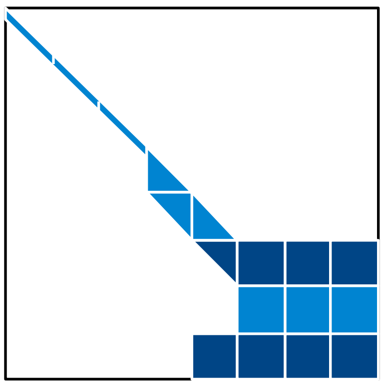


16 cores of Intel Sandy Bridge

Algorithms

three-stage SVD

PLASMA_[scdz]**gesvd**[_Tile][_Async]()

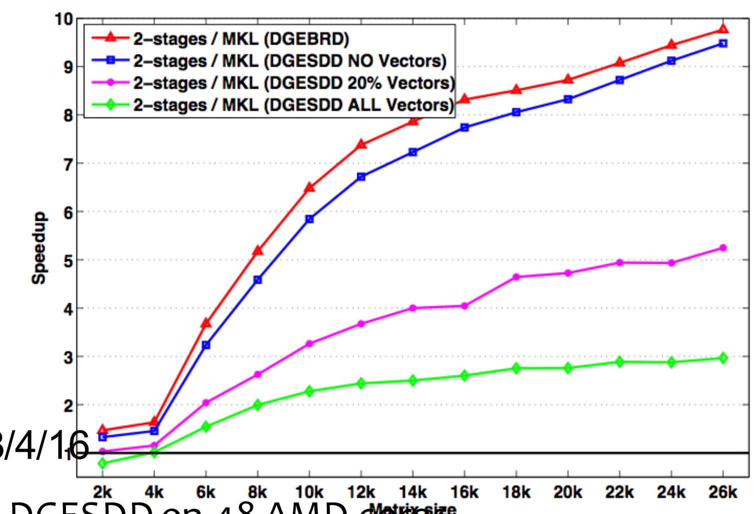


- <u>**Algorithm**</u>
  - two-stage bidiagonal reduction + QR iteration
  - fast singular values, slower singular vectors

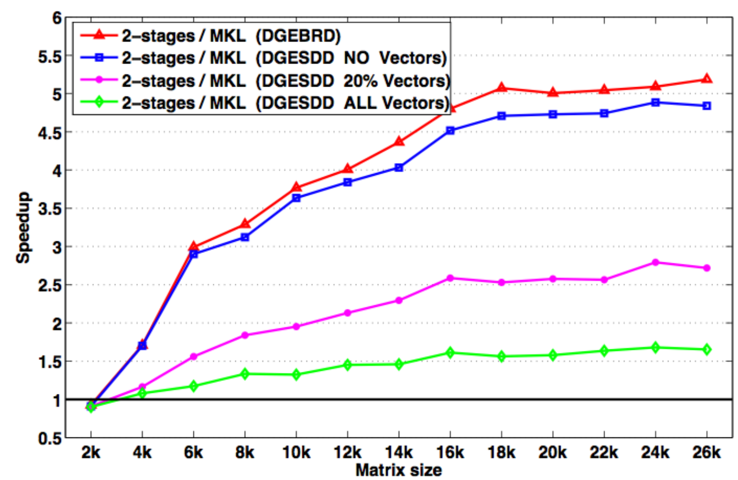
  (possibility of calculating a subset)

- <u>**Numerics**</u>
  - same as LAPACK

- <u>**Performance**</u>
  - comparable with MKL for very small problems
  - absolutely superior for larger problems

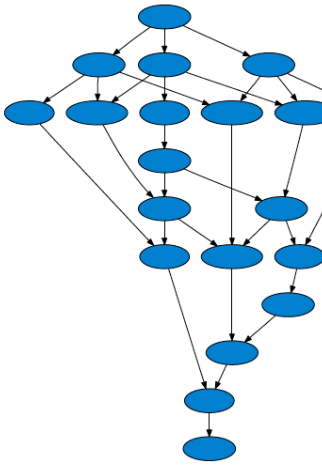


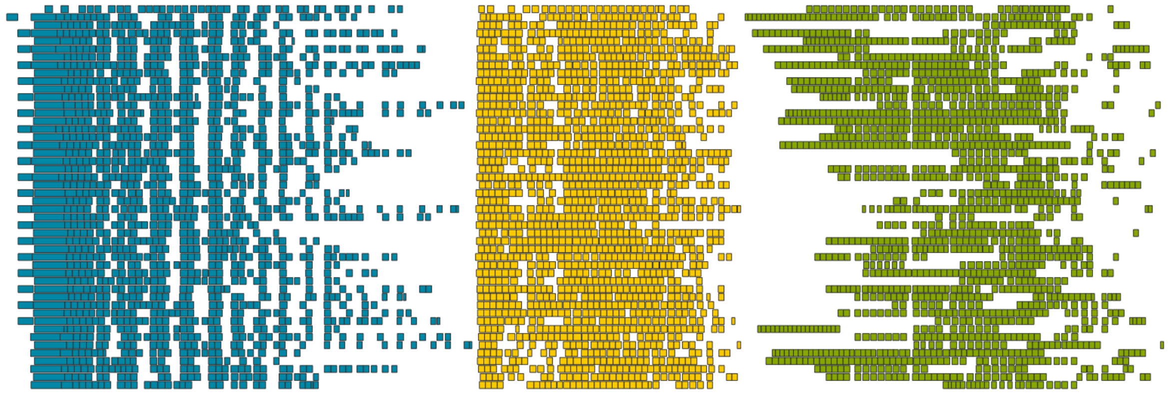
DGESDD on 48 AMD cores

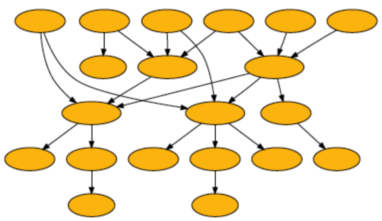


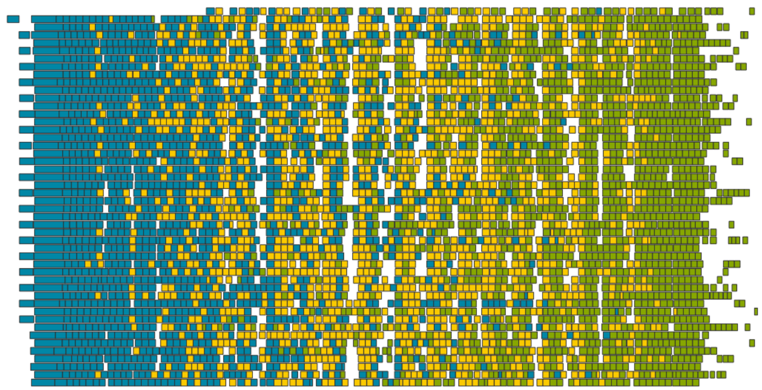
DGESDD on 16 Sandy Bridge cores

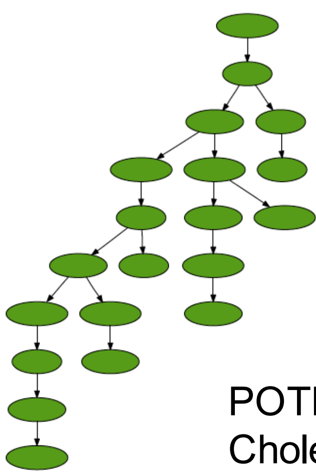
Pipelining: Cholesky Inversion
3 Steps: Factor, Invert L, Multiply L's

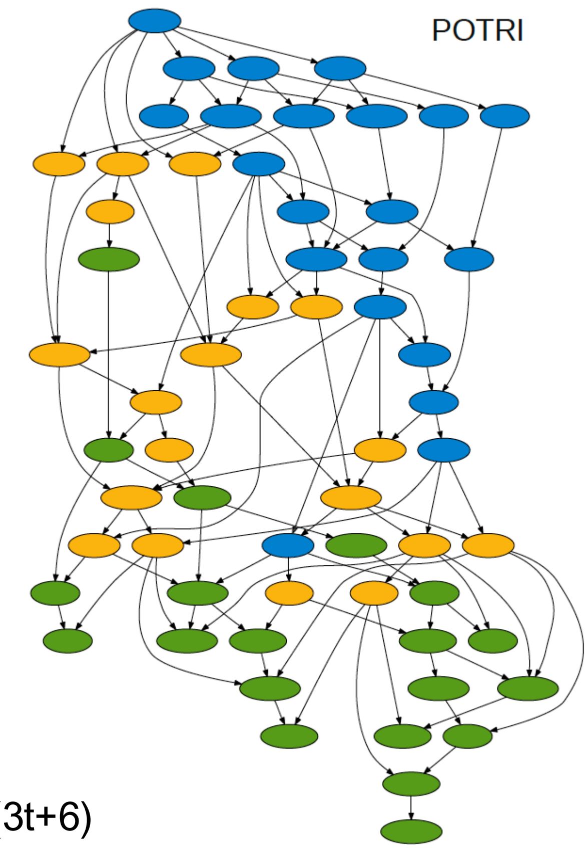












48 cores
POTRF, TRTRI and LAUUM.
The matrix is 4000 x 4000,tile size is 200 x 200,

POTRF+TRTRI+LAUUM: 25 (7t-3)
Cholesky Factorization alone: 3t-2

Pipelined: 18 (3t+6)

Random Butterfly Pivoting (RBP)

- To solve $Ax = b$:
  - Compute $A_r = U^T A V$, with U and V random matrices
  - Factorize A_r without pivoting (GENP)
  - Solve $A_r\, y = U^T b$ and then Solve $x = Vy$
- U and V are Recursive Butterfly Matrices
  - Randomization is cheap ($O(n^2)$ operations)
  - GENP is fast ("Cholesky" speed, take advantage of the GPU)
  - Accuracy is in practice similar to GEPP (with iterative refinement), but…

Think of this as a preconditioner step.

Goal: Transform A into a matrix that would be sufficiently "random" so that, with a probability close to 1, pivoting is not needed.

A **butterfly matrix** is defined as any *n*-by-*n* matrix of the form:

$$B = \frac{1}{\sqrt{2}} \begin{pmatrix} R & S \\ R & -S \end{pmatrix}$$

where R and S are random diagonal matrices.

$B =$ 

PLASMA RBT execution trace

This image cannot currently be displayed.

- with n=2000, nb=250 on 12-core AMD Opteron -

- Partial randomization (i.e. gray) is inexpensive.
- Factorization without pivoting is scalable without synchronizations.

Mixed Precision Methods

- Mixed precision, use the lowest precision required to achieve a given accuracy outcome
  - Improves runtime, reduce power consumption, lower data movement
  - Reformulate to find correction to solution, rather than solution; Δx rather than x.

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

$$x_{i+1} - x_i = -\frac{f(x_i)}{f'(x_i)}$$

48

Idea Goes Something Like This…

- Exploit 32 bit floating point as much as possible.
  - Especially for the bulk of the computation
- Correct or update the solution with selective use of 64 bit floating point to provide a refined results
- Intuitively:
  - Compute a 32 bit result,
  - Calculate a correction to 32 bit result using selected higher precision and,
  - Perform the update of the 32 bit results with the correction using high precision.

Mixed-Precision Iterative Refinement

- Iterative refinement for dense systems, $Ax = b$, can work this way.

L U = lu(A)	$O(n^3)$
x = L\(U\b)	$O(n^2)$
r = b – Ax	$O(n^2)$
WHILE || r || not small enough	
z = L\(U\r)	$O(n^2)$
x = x + z	$O(n^1)$
r = b – Ax	$O(n^2)$
END	

  - Wilkinson, Moler, Stewart, & Higham provide error bound for SP fl pt results when using DP fl pt.

Mixed-Precision Iterative Refinement

- Iterative refinement for dense systems, $Ax = b$, can work this way.

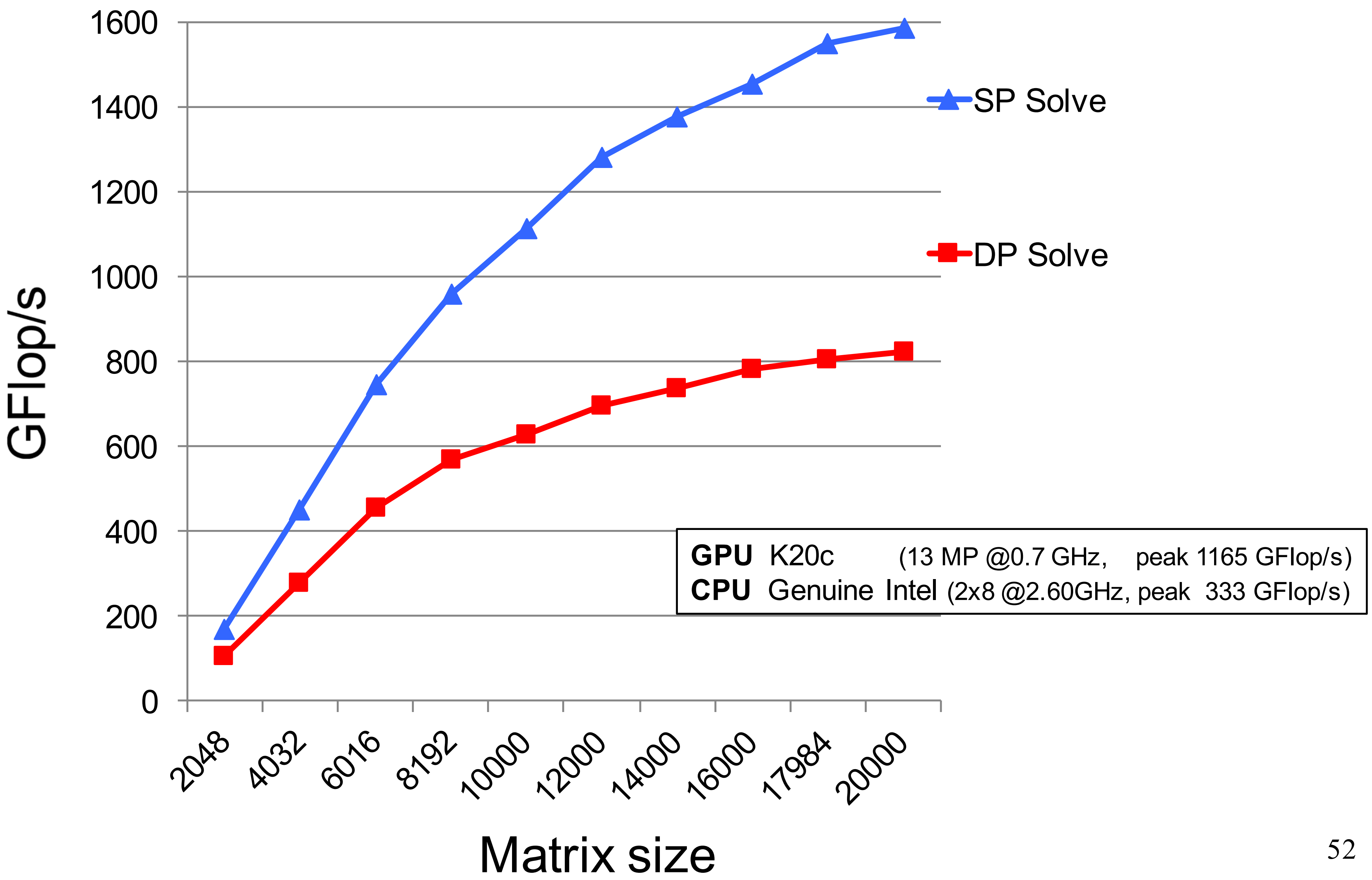
L U = lu(A)	SINGLE	$O(n^3)$
x = L\(U\b)	SINGLE	$O(n^2)$
r = b – Ax	DOUBLE	$O(n^2)$
WHILE || r || not small enough		
z = L\(U\r)	SINGLE	$O(n^2)$
x = x + z	DOUBLE	$O(n^1)$
r = b – Ax	DOUBLE	$O(n^2)$
END		

- Wilkinson, Moler, Stewart, & Higham provide error bound for SP fl pt results when using DP fl pt.
- It can be shown that using this approach we can compute the solution to 64-bit floating point precision.

- Requires extra storage, total is 1.5 times normal;
- $O(n^3)$ work is done in lower precision
- $O(n^2)$ work is done in high precision
- Problems if the matrix is ill-conditioned in sp; $O(10^8)$

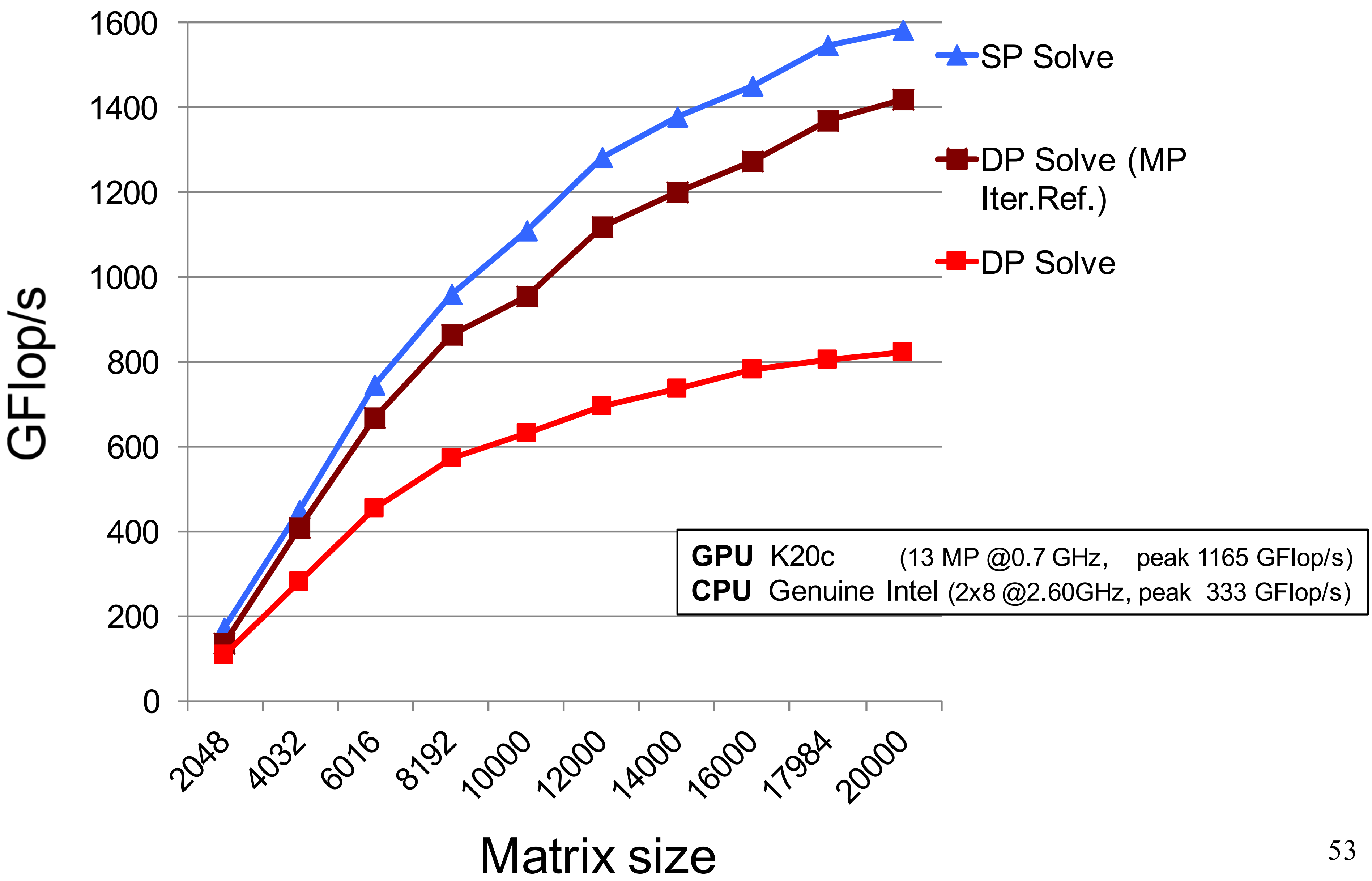
Mixed precision iterative refinement

Solving general dense linear systems using mixed precision iterative refinement



Mixed precision iterative refinement

Solving general dense linear systems using mixed precision iterative refinement



Critical Issues at Peta & Exascale for Algorithm and Software Design

- Synchronization-reducing algorithms
  - Break Fork-Join model
- Communication-reducing algorithms
  - Use methods which have lower bound on communication
- Mixed precision methods
  - 2x speed of ops and 2x speed for data movement
- Autotuning
  - Today's machines are too complicated, build "smarts" into software to adapt to the hardware
- Fault resilient algorithms
  - Implement algorithms that can recover from failures/bit flips
- Reproducibility of results
  - Today we can't guarantee this. We understand the issues, but some of our "colleagues" have a hard time with this.

Collaborators / Software / Support

- **PLASMA**
  http://icl.cs.utk.edu/plasma/
- **MAGMA**
  http://icl.cs.utk.edu/magma/
- **Quark (RT for Shared Memory)**
- **http://icl.cs.utk.edu/quark/**
- **PaRSEC(**Parallel Runtime Scheduling and Execution Control**)**
- **http://icl.cs.utk.edu/parsec/**

- Collaborating partners
  University of Tennessee, Knoxville
  University of California, Berkeley
  University of Colorado, Denver

MAGMA **PLASMA**