

Argonne Training Program on

# EXTREME-SCALE COMPUTING



July 30 – August 11, 2017

## Adaptive Linear Solvers and Eigensolvers

---

**Jack Dongarra**

University of Tennessee  
Oak Ridge National Laboratory  
University of Manchester

Copy of slides at <http://bit.ly/atpesc-2017-dongarra>



# 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<sub>2</sub>



# Existing Math Software - Dense LA

DIRECT SOLVERS	License	Support	Type		Language			Mode			Dense	Sparse Direct			Sparse Iterative		Sparse Eigenvalue		Last release date
			Real	Complex	F77/ F95	C	C++	Shared	Accel.	Dist		SPD	SI	Gen	SPD	Gen	Sym	Gen	
<a href="#">Chameleon</a>	<a href="#">CeCILL-C</a>	See authors	X	X		X		X	C	M	X								2014-04-15
<a href="#">DPLASMA</a>	<a href="#">BSD</a>	yes	X	X		X		X	C	M	X								2014-04-14
<a href="#">Eigen</a>	<a href="#">Mozilla</a>	yes	X	X			X	X			X	X		X	X	X			2015-01-21
<a href="#">Elemental</a>	<a href="#">New BSD</a>	yes	X	X			X			M	X	X	X	X					2014-11-08
<a href="#">ELPA</a>	<a href="#">LGPL</a>	yes	X	X	F90	X		X		M	X								2015-05-29
<a href="#">FLENS</a>	<a href="#">BSD</a>	yes	X	X			X	X			X								2014-05-11
<a href="#">hmat-oss</a>	<a href="#">GPL</a>	yes	X	X	X	X	X	X			X			X					2015-03-10
<a href="#">LAPACK</a>	<a href="#">BSD</a>	yes	X	X	X	X		X			X								2013-11-26
<a href="#">LAPACK95</a>	<a href="#">BSD</a>	yes	X	X	X			X			X								2000-11-30
<a href="#">libflame</a>	<a href="#">New BSD</a>	yes	X	X	X	X		X			X								2014-03-18
<a href="#">MAGMA</a>	<a href="#">BSD</a>	yes	X	X	X	X		X	C/O/X		X				X	X	X		2015-05-05
<a href="#">NAPACK</a>	<a href="#">BSD</a>	yes	X		X			X			X				X		X		?
<a href="#">PLAPACK</a>	<a href="#">LGPL</a>	yes	X	X	X	X				M	X								2007-06-12
<a href="#">PLASMA</a>	<a href="#">BSD</a>	yes	X	X	X	X		X			X								2015-04-27
<a href="#">rejtrix</a>	<a href="#">by-nc-sa</a>	yes	X				X	X			X				P	P			2013-10-01
<a href="#">ScaLAPACK</a>	<a href="#">BSD</a>	yes	X	X	X	X				M/P	X								2012-05-01
<a href="#">Trilinos/Pliris</a>	<a href="#">BSD</a>	yes	X	X		X	X			M	X								2015-05-07
<a href="#">ViennaCL</a>	<a href="#">MIT</a>	yes	X				X	X	C/O/X		X				X	X	X	X	2014-12-11

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

◆ LINPACK, EISPACK, LAPACK, ScaLAPACK

➤ PLASMA, MAGMA

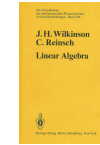


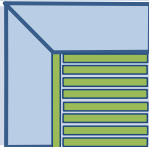

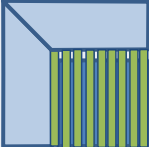



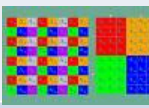


# DLA Solvers

---

- 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 x flop/s?

- x 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.
- Tflop/s refers to trillions ( $10^{12}$ ) of floating point operations per second and
- Pflop/s refers to  $10^{15}$  floating point operations per second.

### ♦ 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 Skylake processor at 2.1 GHz can complete 32 floating point operations per cycle per core or a theoretical peak performance of 67.2 GFlop/s per core or 1.51 Tflop/s for the socket of 24 cores.

# Peak Performance - Per Core

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

## Floating point operations per cycle per core

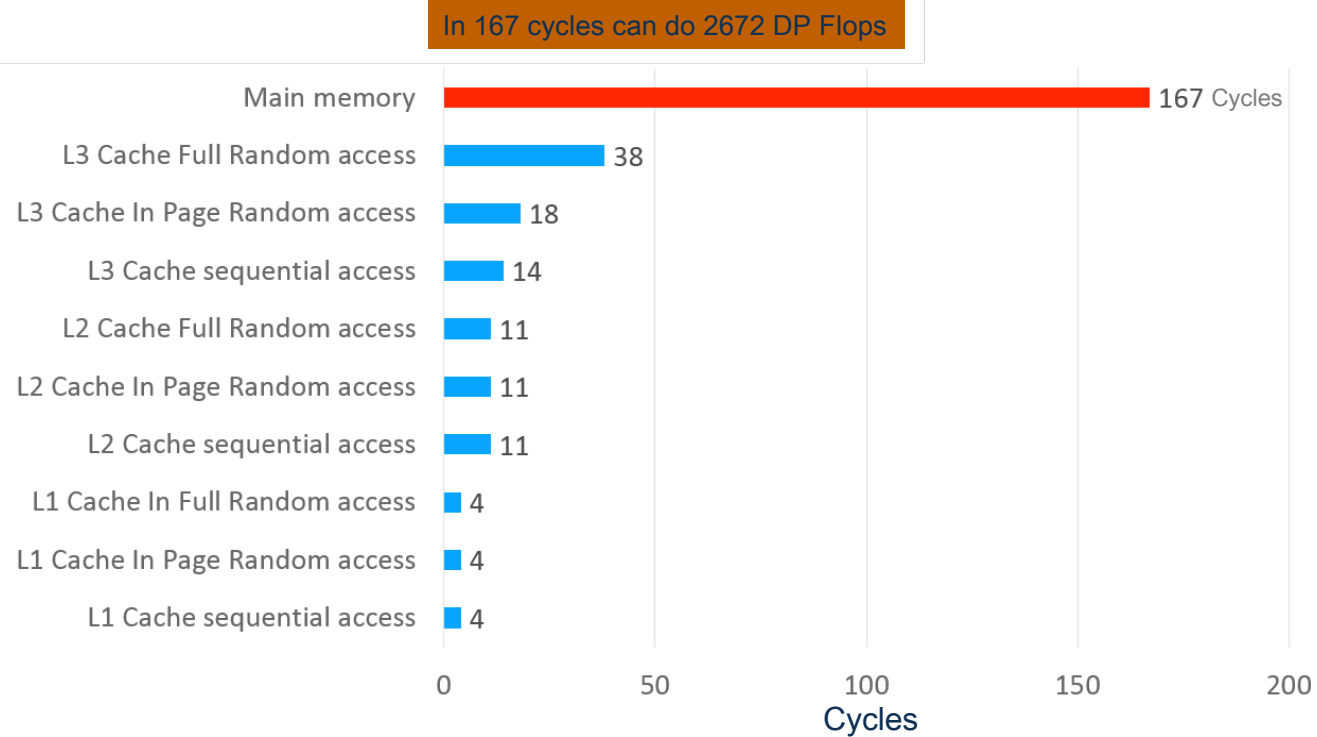
- + Most of the recent computers have FMA (Fused multiple add): (i.e.  $x \leftarrow x + y * z$  in one cycle)
- + Intel Xeon earlier models and AMD Opteron have SSE2
  - + 2 flops/cycle/core DP & 4 flops/cycle/core SP
- + Intel Xeon Nehalem ('09) & Westmere ('10) have SSE4
  - + 4 flops/cycle/core DP & 8 flops/cycle/core SP
- + Intel Xeon Sandy Bridge('11) & Ivy Bridge ('12) have AVX
  - + 8 flops/cycle/core DP & 16 flops/cycle/core SP
- + Intel Xeon Haswell ('13) & (Broadwell ('14)) AVX2
  - + 16 flops/cycle/core DP & 32 flops/cycle/core SP
  - + Xeon Phi (per core) is at 16 flops/cycle DP & 32 flops/cycle SP
- + Intel Xeon Skylake (server) & KNL AVX 512
  - + 32 flops/cycle/core DP & 64 flops/cycle/core SP
  - + Skylake w/24 cores & Knight's Landing w/68 cores



We  
are  
here

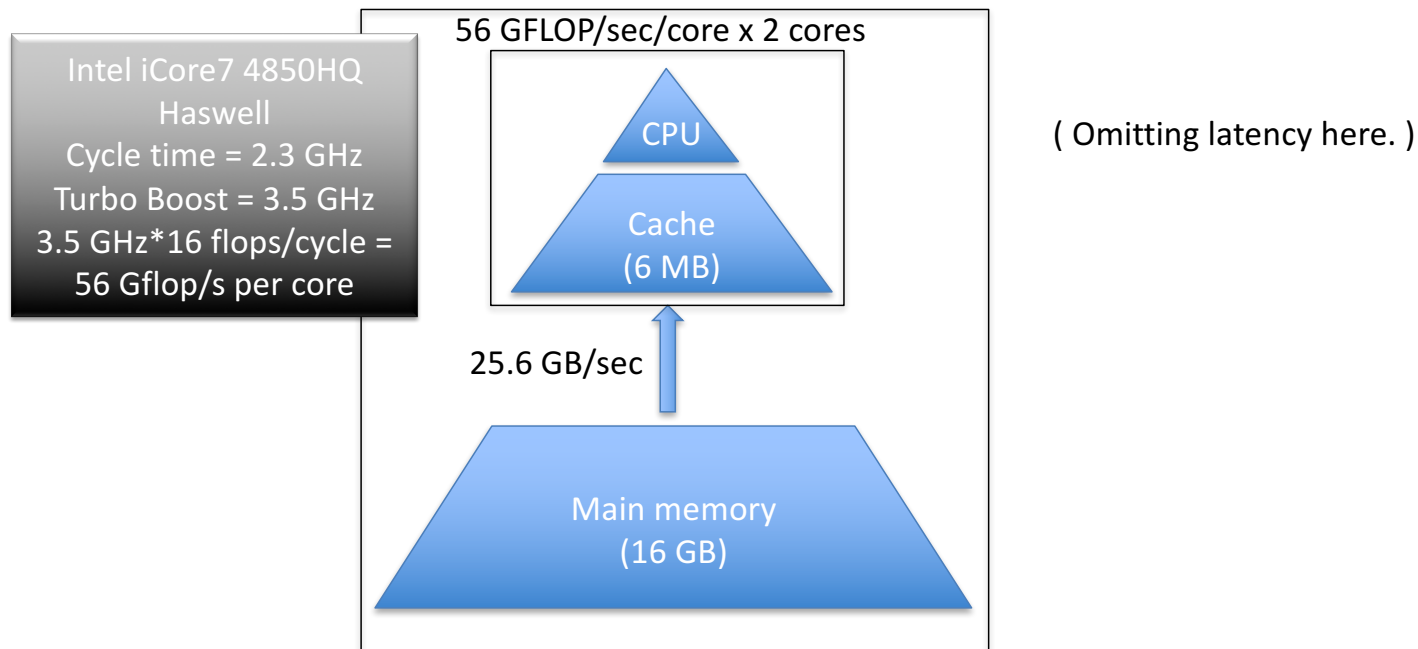


# CPU Access Latencies in Clock Cycles



# Memory transfer

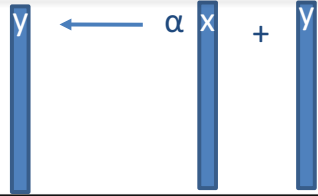
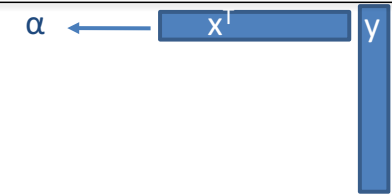
- One level of memory model on my laptop:



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

<p>AXPY:</p> 	<pre>for ( j = 0; j &lt; n; j++)     y[i] += a * x[i];</pre> <p>(without increment)</p>	<p>n MUL n ADD 2n FLOP n FMA</p>
<p>DOT:</p> 	<pre>alpha = 0e+00; for ( j = 0; j &lt; n; j++)     alpha += x[i] * y[i];</pre> <p>(without increment)</p>	<p>n MUL n ADD 2n FLOP n FMA</p>

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:
  - $(375,000 \text{ double}) * (8 \text{ Bytes / double}) = 3 \text{ MBytes}$  per vector
  - ( Two vectors fit in cache (6 MBytes). OK.)

- Time to move the vectors from memory to cache:
  - $(6 \text{ MBytes}) / (25.6 \text{ GBytes/sec}) = \mathbf{0.23 \text{ ms}}$
- Time to perform computation of DOT:
  - $(2n \text{ flops}) / (56 \text{ Gflop/sec}) = \mathbf{0.013 \text{ ms}}$

# Vector Operations

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

$$\text{Performance} = (2 \times 375,000 \text{ flops}) / .23\text{ms} = 3.2 \text{ Gflop/s}$$

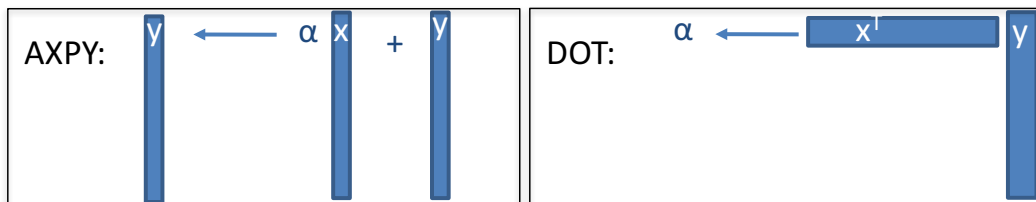
**Performance for DOT  $\leq 3.2$  Gflop/s**

**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 FLOPs

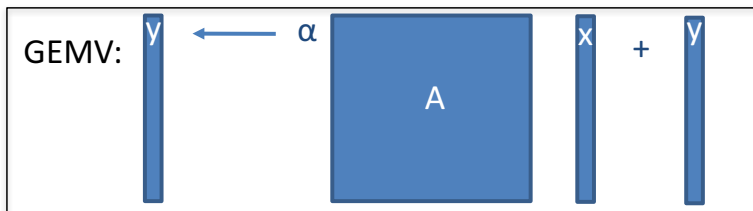
2n memory references

AXPY: 2n READ, n WRITE

DOT: 2n READ

RATIO FLOPs to Memory Ops: 1:1

## Level 2 BLAS Matrix-Vector operations

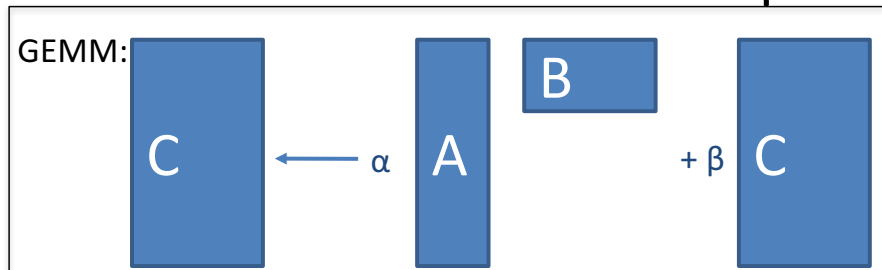


2n<sup>2</sup> FLOPs

n<sup>2</sup> memory references

RATIO FLOPs to Memory Ops: 2:1

## Level 3 BLAS Matrix-Matrix operations



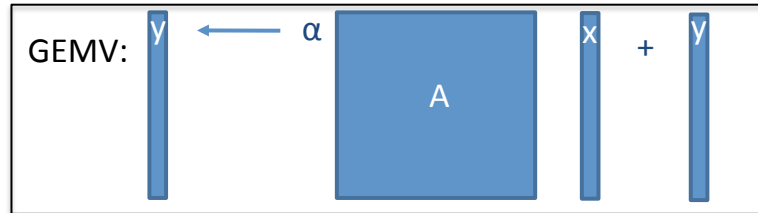
2n<sup>3</sup> FLOPs

3n<sup>2</sup> memory references

3n<sup>2</sup> READ, n<sup>2</sup> WRITE

RATIO FLOPs to Memory Ops: n:2

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



- Data size:
  - $(860^2 + 2 \cdot 860 \text{ double}) \cdot (8 \text{ Bytes / double}) \sim 6 \text{ MBytes}$

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

- Time to move the data from memory to cache:
  - $(6 \text{ MBytes}) / (25.6 \text{ GBytes/sec}) = \mathbf{0.23 \text{ ms}}$
- Time to perform computation of GEMV:
  - $(2n^2 \text{ flops}) / (56 \text{ Gflop/sec}) = \mathbf{0.026 \text{ ms}}$

# Matrix - Vector Operations

$$\begin{aligned}\text{total\_time} &\geq \max ( \text{time\_comm} , \text{time\_comp} ) \\ &= \max ( 0.23\text{ms} , 0.026\text{ms} ) = 0.23\text{ms}\end{aligned}$$

$$\text{Performance} = (2 \times 860^2 \text{ flops}) / .23\text{ms} = 6.4 \text{ Gflop/s}$$

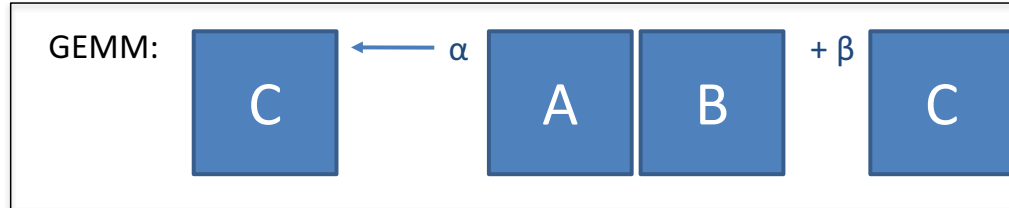
**Performance for GEMV  $\leq 6.4 \text{ Gflop/s}$**

Performance for DOT  $\leq 3.2 \text{ Gflop/s}$

**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:
  - $(500^2 \text{ double}) * (8 \text{ Bytes / double}) = 2 \text{ MBytes per matrix}$
  - ( Three matrices fit in cache (6 MBytes). OK.)
- Time to move the matrices in cache:
  - $(6 \text{ MBytes}) / (25.6 \text{ GBytes/sec}) = \mathbf{0.23 \text{ ms}}$
- Time to perform computation in GEMM:
  - $(2n^3 \text{ flops}) / (56 \text{ Gflop/sec}) = \mathbf{4.5 \text{ ms}}$



# Matrix Matrix Operations

$$\begin{aligned}\text{total\_time} &\geq \max(\text{time\_comm}, \text{time\_comp}) \\ &= \max(0.23\text{ms}, 4.46\text{ms}) = 4.46\text{ms}\end{aligned}$$

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

$$\text{Performance} = (2 \times 500^3 \text{ flops}) / 4.5\text{ms} = 55.5 \text{ Gflop/s}$$

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

If we assume  $\text{total\_time} \approx \text{time\_comm} + \text{time\_comp}$ , we get

**Performance for GEMM  $\approx 55.5 \text{ Gflop/sec}$**

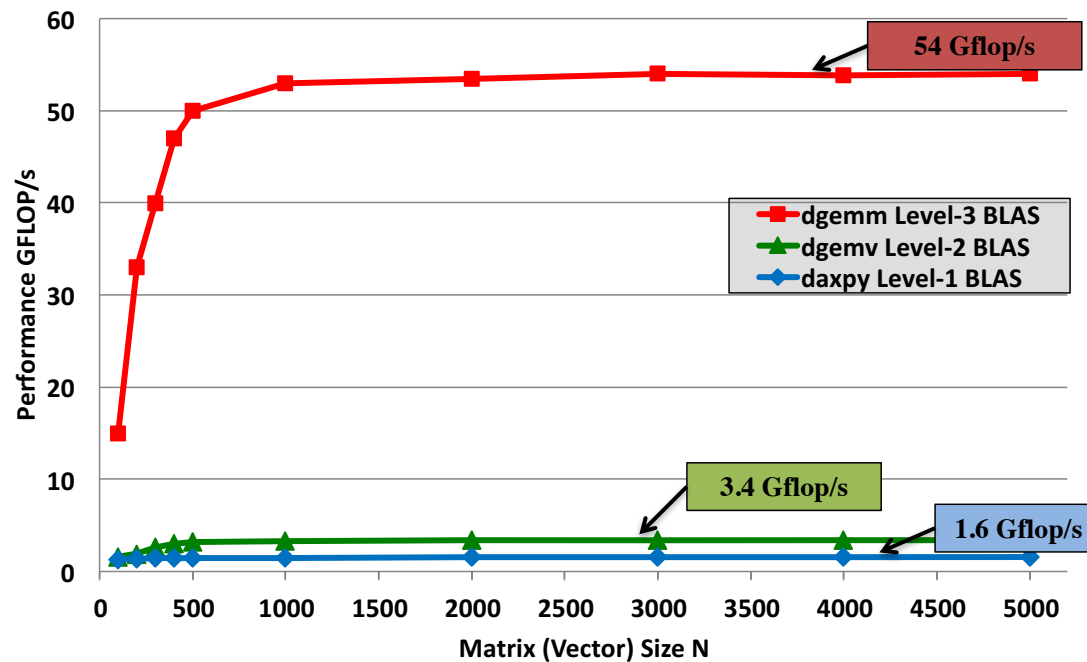
Performance for DOT  $\leq 3.2 \text{ Gflop/s}$

Performance for GEMV  $\leq 6.4 \text{ Gflop/s}$

(Out of 56 Gflop/sec possible, so that would be 99% 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

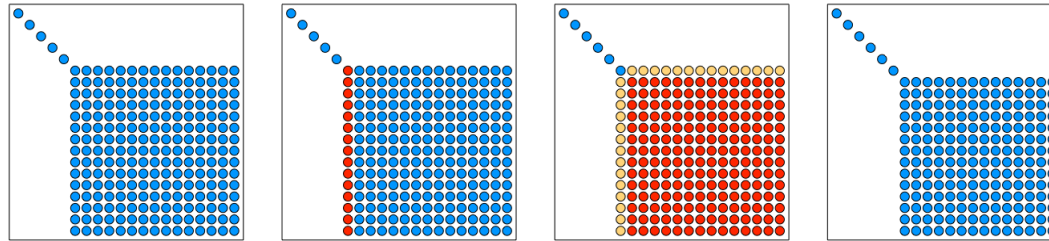
- Reuse based on matrices that fit into cache.
- What if you have matrices bigger than cache?

# Issues

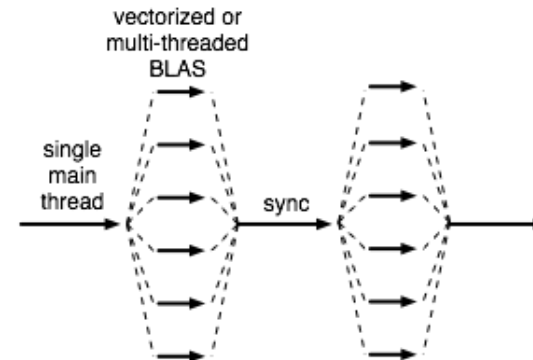
- Reuse based on matrices that fit into cache.
- What if you have matrices bigger than cache?
- Break matrices into blocks or tiles that will fit.

The diagram illustrates a matrix operation involving three 3x3 grids of elements. The first grid on the left contains elements  $C_{11}, C_{12}, C_{13}$  in the first row,  $C_{21}, C_{22}, C_{23}$  in the second row, and  $C_{31}, C_{32}, C_{33}$  in the third row. An arrow points from this grid to the second grid, which is identical to the first. To the right of the second grid is a plus sign followed by the Greek letter  $\alpha$ . The third grid contains elements  $A_{11}, A_{12}, A_{13}$  in the first row,  $A_{21}, A_{22}, A_{23}$  in the second row, and  $A_{31}, A_{32}, A_{33}$  in the third row. To the right of the third grid is a multiplication sign  $*$ . The fourth grid contains elements  $B_{11}, B_{12}, B_{13}$  in the first row,  $B_{21}, B_{22}, B_{23}$  in the second row, and  $B_{31}, B_{32}, B_{33}$  in the third row. The overall expression is: 
$$\begin{bmatrix} C_{11} & C_{12} & C_{13} \\ C_{21} & C_{22} & C_{23} \\ C_{31} & C_{32} & C_{33} \end{bmatrix} \leftarrow \beta \begin{bmatrix} C_{11} & C_{12} & C_{13} \\ C_{21} & C_{22} & C_{23} \\ C_{31} & C_{32} & C_{33} \end{bmatrix} + \alpha \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} * \begin{bmatrix} B_{11} & B_{12} & B_{13} \\ B_{21} & B_{22} & B_{23} \\ B_{31} & B_{32} & B_{33} \end{bmatrix}$$

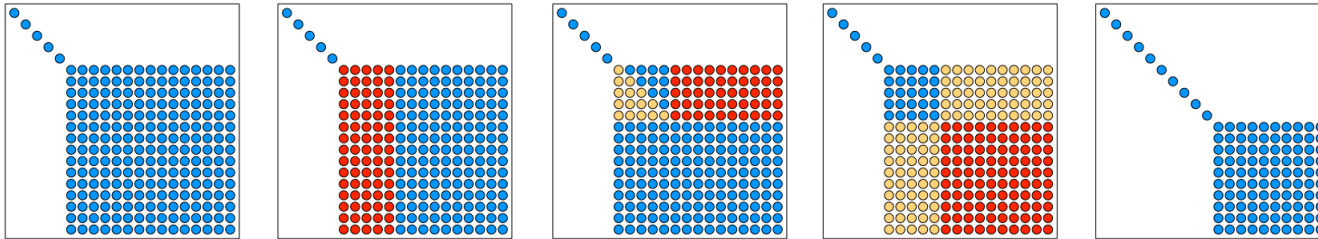
## LU Factorization in LINPACK (1970's)



- Factor one column at a time
  - `i_amax` and `_scal`
- Update each column of trailing matrix, one column at a time
  - `_axpy`
- Level 1 BLAS
- Bulk synchronous
  - Single main thread
  - Parallel work in BLAS
  - “Fork-and-join” model



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

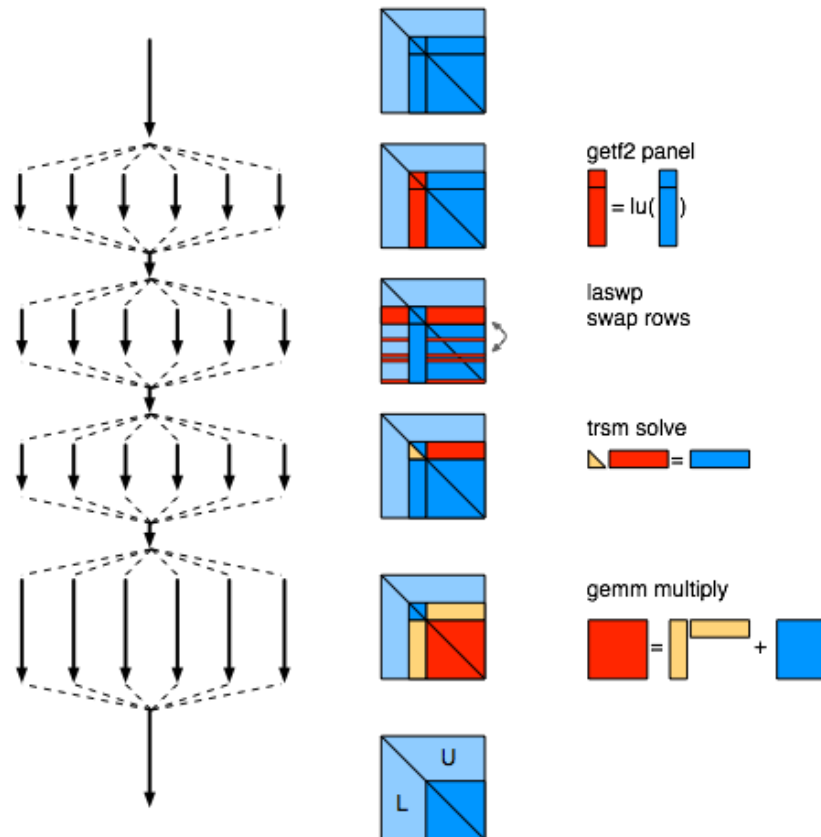


- Factor panel of  $n_b$  columns
  - getf2, unblocked BLAS-2 code
- Level 3 BLAS update block-row of U
  - trsm
- Level 3 BLAS update trailing matrix
  - gemm
  - Aimed at machines with cache hierarchy
- Bulk synchronous

# Parallelism in LAPACK



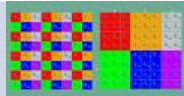
## ♦ Most flops in gemm update

- $\frac{2}{3} n^3$  term
- Easily parallelized using multi-threaded BLAS
- Done in any reasonable software
- Other operations lower order
  - Potentially expensive if not parallelized





# 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

## 2D Block Cyclic Layout

Matrix point of view	Processor point of view																																																																																																																																																																											
<table><tr><td>0</td><td>2</td><td>4</td><td>0</td><td>2</td><td>4</td><td>0</td><td>2</td><td>4</td></tr><tr><td>1</td><td>3</td><td>5</td><td>1</td><td>3</td><td>5</td><td>1</td><td>3</td><td>5</td></tr><tr><td>0</td><td>2</td><td>4</td><td>0</td><td>2</td><td>4</td><td>0</td><td>2</td><td>4</td></tr><tr><td>1</td><td>3</td><td>5</td><td>1</td><td>3</td><td>5</td><td>1</td><td>3</td><td>5</td></tr><tr><td>0</td><td>2</td><td>4</td><td>0</td><td>2</td><td>4</td><td>0</td><td>2</td><td>4</td></tr><tr><td>1</td><td>3</td><td>5</td><td>1</td><td>3</td><td>5</td><td>1</td><td>3</td><td>5</td></tr><tr><td>0</td><td>2</td><td>4</td><td>0</td><td>2</td><td>4</td><td>0</td><td>2</td><td>4</td></tr><tr><td>1</td><td>3</td><td>5</td><td>1</td><td>3</td><td>5</td><td>1</td><td>3</td><td>5</td></tr><tr><td>0</td><td>2</td><td>4</td><td>0</td><td>2</td><td>4</td><td>0</td><td>2</td><td>4</td></tr><tr><td>1</td><td>3</td><td>5</td><td>1</td><td>3</td><td>5</td><td>1</td><td>3</td><td>5</td></tr></table>	0	2	4	0	2	4	0	2	4	1	3	5	1	3	5	1	3	5	0	2	4	0	2	4	0	2	4	1	3	5	1	3	5	1	3	5	0	2	4	0	2	4	0	2	4	1	3	5	1	3	5	1	3	5	0	2	4	0	2	4	0	2	4	1	3	5	1	3	5	1	3	5	0	2	4	0	2	4	0	2	4	1	3	5	1	3	5	1	3	5	<table><tr><td>0</td><td>0</td><td>0</td><td>2</td><td>2</td><td>2</td><td>4</td><td>4</td><td>4</td></tr><tr><td>0</td><td>0</td><td>0</td><td>2</td><td>2</td><td>2</td><td>4</td><td>4</td><td>4</td></tr><tr><td>0</td><td>0</td><td>0</td><td>2</td><td>2</td><td>2</td><td>4</td><td>4</td><td>4</td></tr><tr><td>0</td><td>0</td><td>0</td><td>2</td><td>2</td><td>2</td><td>4</td><td>4</td><td>4</td></tr><tr><td>0</td><td>0</td><td>0</td><td>2</td><td>2</td><td>2</td><td>4</td><td>4</td><td>4</td></tr><tr><td>1</td><td>1</td><td>1</td><td>3</td><td>3</td><td>3</td><td>5</td><td>5</td><td>5</td></tr><tr><td>1</td><td>1</td><td>1</td><td>3</td><td>3</td><td>3</td><td>5</td><td>5</td><td>5</td></tr><tr><td>1</td><td>1</td><td>1</td><td>3</td><td>3</td><td>3</td><td>5</td><td>5</td><td>5</td></tr><tr><td>1</td><td>1</td><td>1</td><td>3</td><td>3</td><td>3</td><td>5</td><td>5</td><td>5</td></tr></table>	0	0	0	2	2	2	4	4	4	0	0	0	2	2	2	4	4	4	0	0	0	2	2	2	4	4	4	0	0	0	2	2	2	4	4	4	0	0	0	2	2	2	4	4	4	1	1	1	3	3	3	5	5	5	1	1	1	3	3	3	5	5	5	1	1	1	3	3	3	5	5	5	1	1	1	3	3	3	5	5	5
0	2	4	0	2	4	0	2	4																																																																																																																																																																				
1	3	5	1	3	5	1	3	5																																																																																																																																																																				
0	2	4	0	2	4	0	2	4																																																																																																																																																																				
1	3	5	1	3	5	1	3	5																																																																																																																																																																				
0	2	4	0	2	4	0	2	4																																																																																																																																																																				
1	3	5	1	3	5	1	3	5																																																																																																																																																																				
0	2	4	0	2	4	0	2	4																																																																																																																																																																				
1	3	5	1	3	5	1	3	5																																																																																																																																																																				
0	2	4	0	2	4	0	2	4																																																																																																																																																																				
1	3	5	1	3	5	1	3	5																																																																																																																																																																				
0	0	0	2	2	2	4	4	4																																																																																																																																																																				
0	0	0	2	2	2	4	4	4																																																																																																																																																																				
0	0	0	2	2	2	4	4	4																																																																																																																																																																				
0	0	0	2	2	2	4	4	4																																																																																																																																																																				
0	0	0	2	2	2	4	4	4																																																																																																																																																																				
1	1	1	3	3	3	5	5	5																																																																																																																																																																				
1	1	1	3	3	3	5	5	5																																																																																																																																																																				
1	1	1	3	3	3	5	5	5																																																																																																																																																																				
1	1	1	3	3	3	5	5	5																																																																																																																																																																				

# ScaLAPACK

Scalable Linear Algebra PACKage



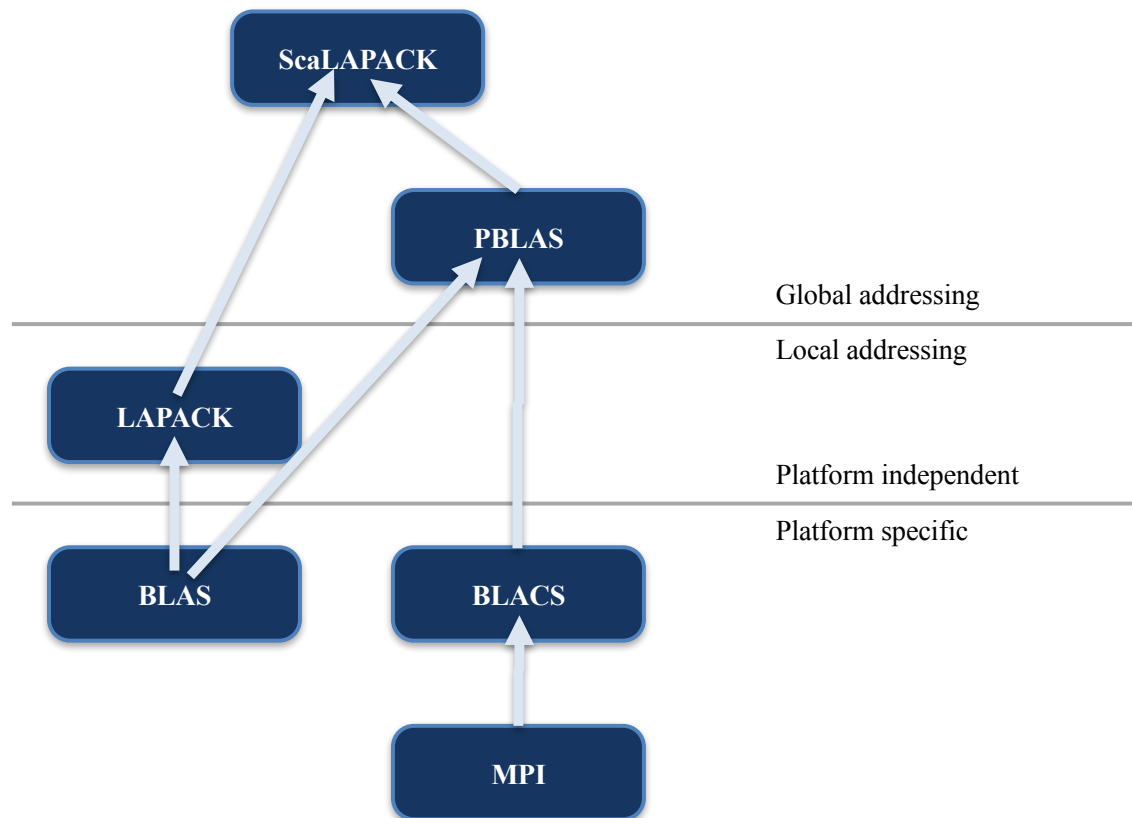
- Distributed memory
- Message Passing
  - Clusters of SMPs
  - Supercomputers
- Dense linear algebra
- Modules
  - PBLAS: Parallel BLAS
  - BLACS: Basic Linear Algebra Communication Subprograms

## PBLAS

- Similar to BLAS in functionality and naming
- Built on BLAS and BLACS
- Provide global view of matrix
- LAPACK: `dge____( m, n, A(ia, ja), lda, ... )`
  - Submatrix offsets implicit in pointer
- ScaLAPACK: `pdge____( m, n, A, ia, ja, descA, ...`
  - Pass submatrix offsets and matrix descriptor

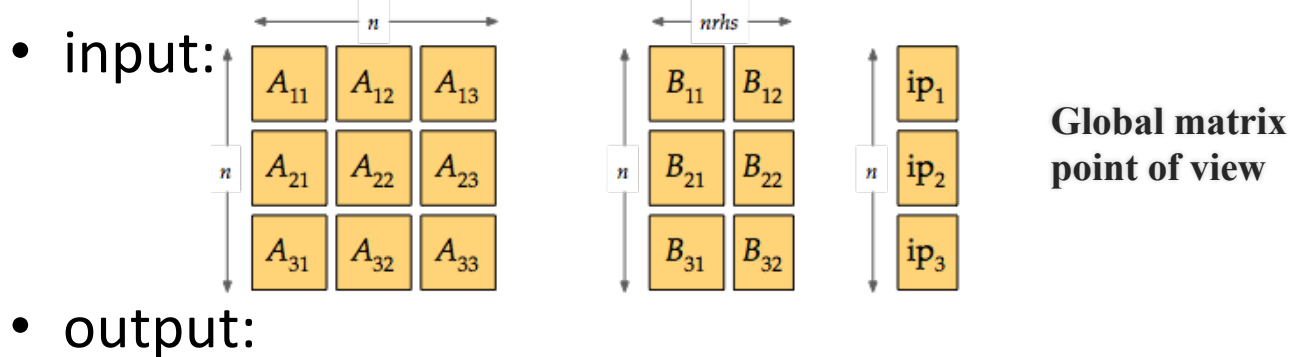


# ScaLAPACK structure

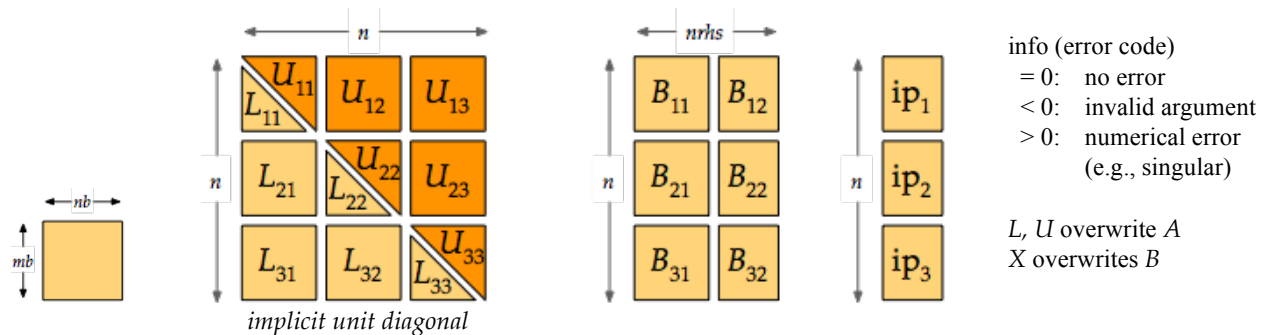


# ScaLAPACK routine, solve $AX = B$

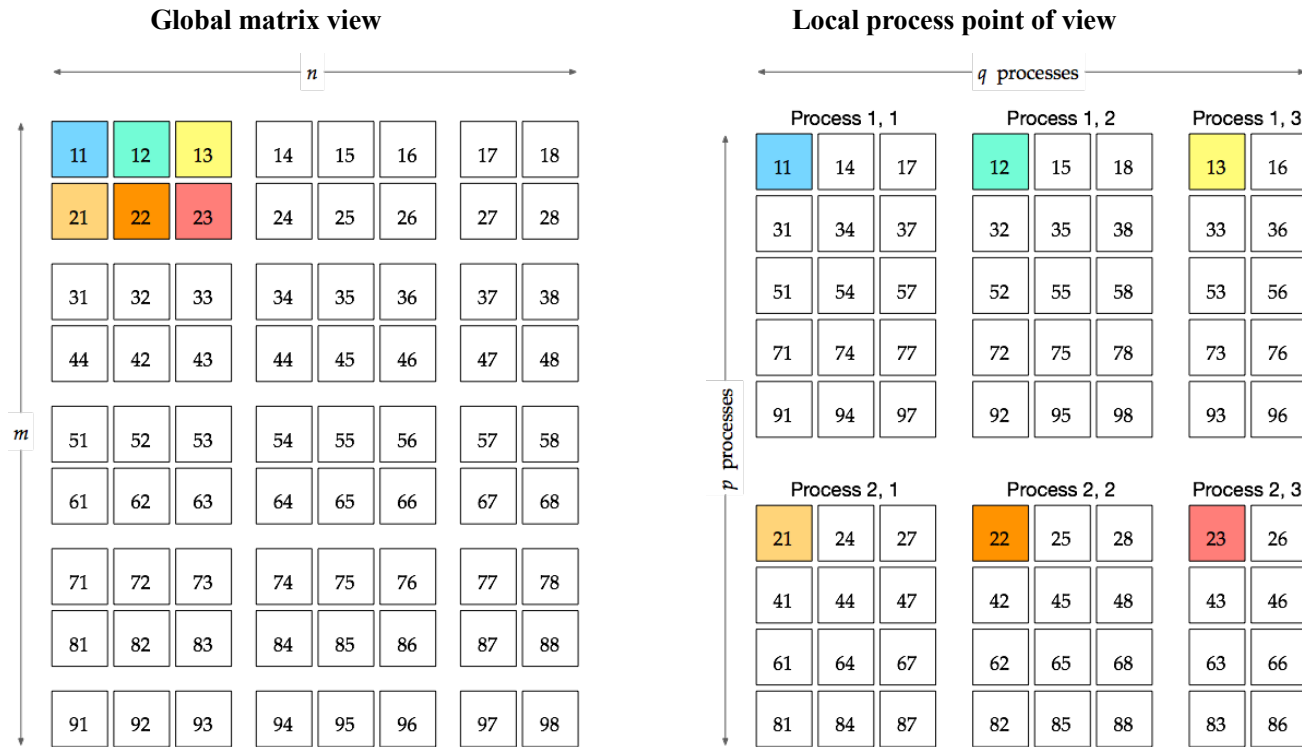
- LAPACK: `dgesv(n, nrhs, A, lda, ipiv, B, ldb, info)`
- ScaLAPACK: `pdgesv(n, nrhs, A, ia, ja, descA, ipiv, B, ib, jb, descB, info)`



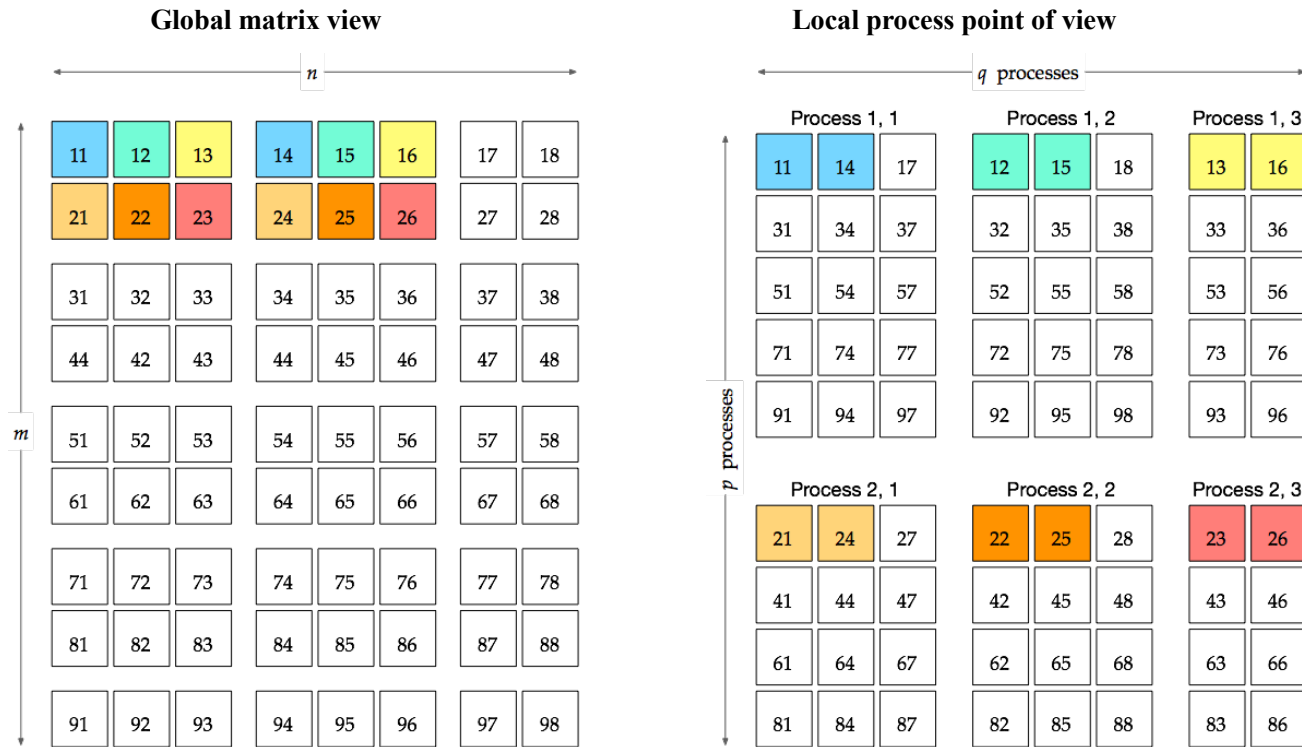
- output:



# 2D block-cyclic layout $m \times n$ matrix $p \times q$ process grid

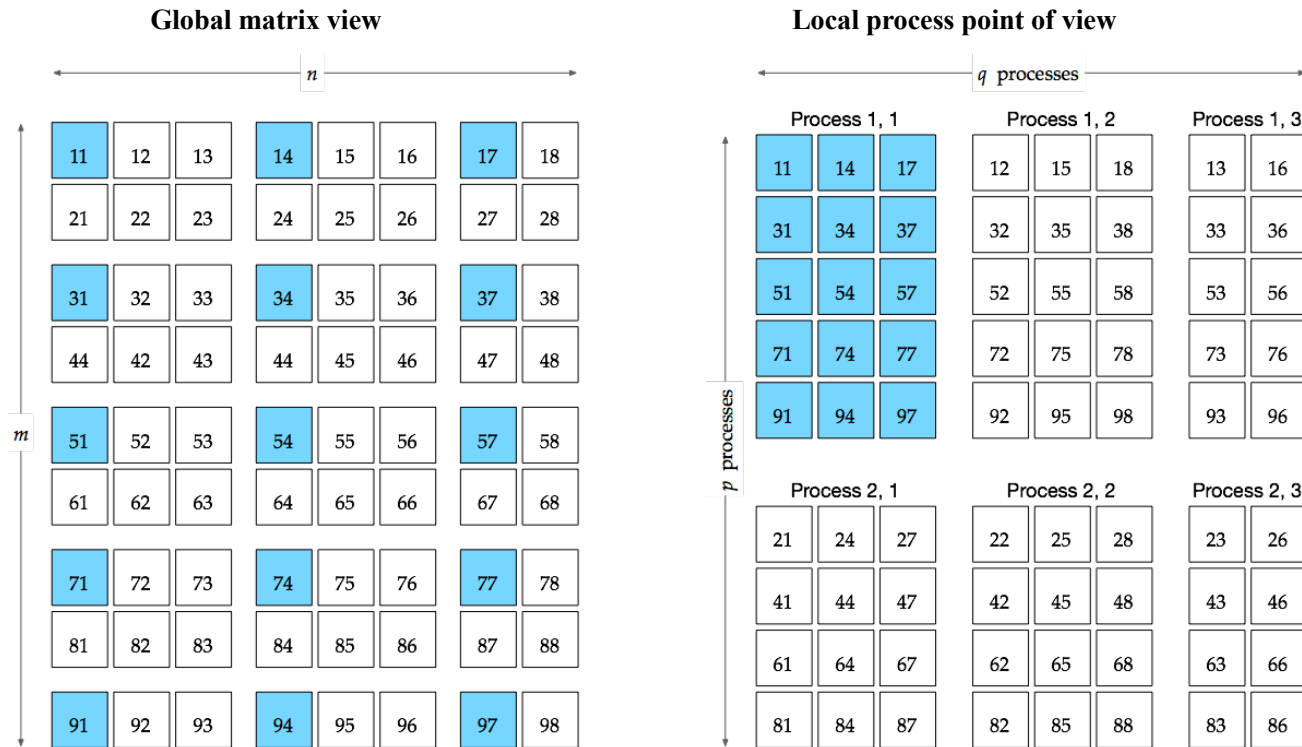


# 2D block-cyclic layout $m \times n$ matrix $p \times q$ process grid

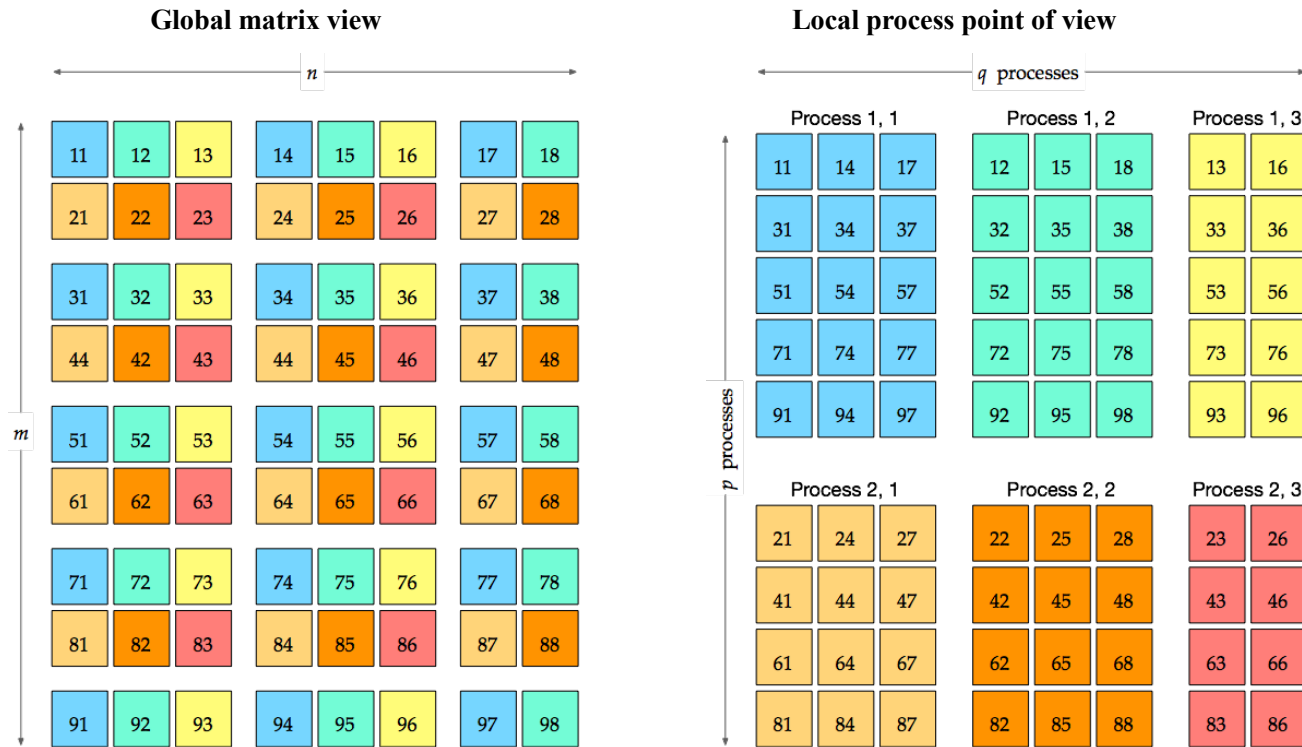




# 2D block-cyclic layout $m \times n$ matrix $p \times q$ process grid

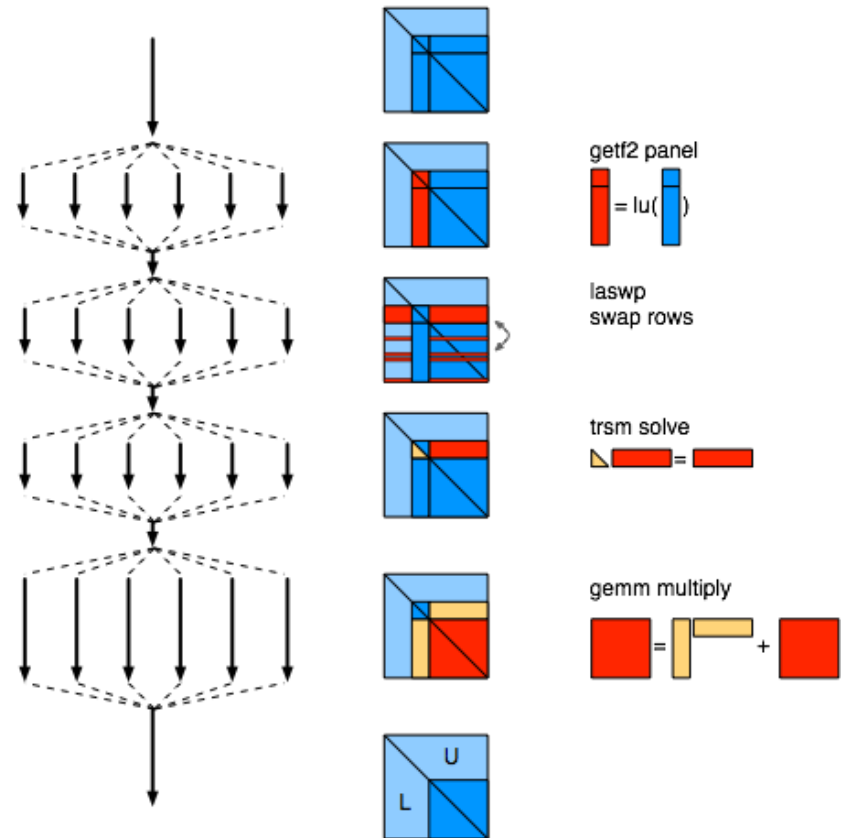


# 2D block-cyclic layout $m \times n$ matrix $p \times q$ process grid

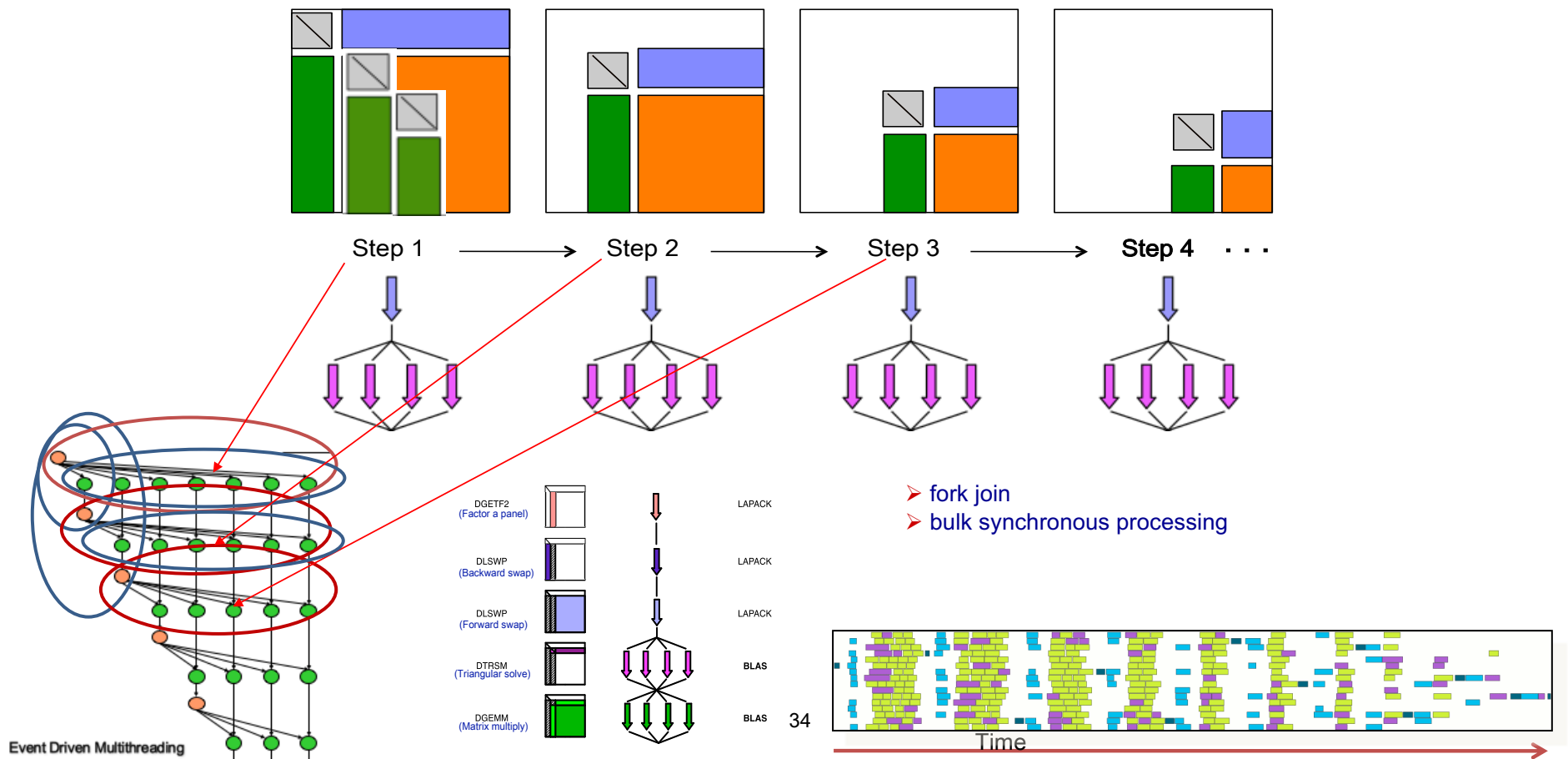


# Parallelism in ScaLAPACK

- Similar to LAPACK
- Bulk-synchronous
- Most flops in gemm update
  - $\frac{2}{3} n^3$  term
  - Can use **sequential BLAS**,  
 $p \times q = \# \text{ cores}$   
 $= \# \text{ MPI processes,}$   
 $\text{num\_threads} = 1$
  - Or **multi-threaded BLAS**,  
 $p \times q = \# \text{ nodes}$   
 $= \# \text{ MPI processes,}$   
 $\text{num\_threads} = \# \text{ cores/node}$

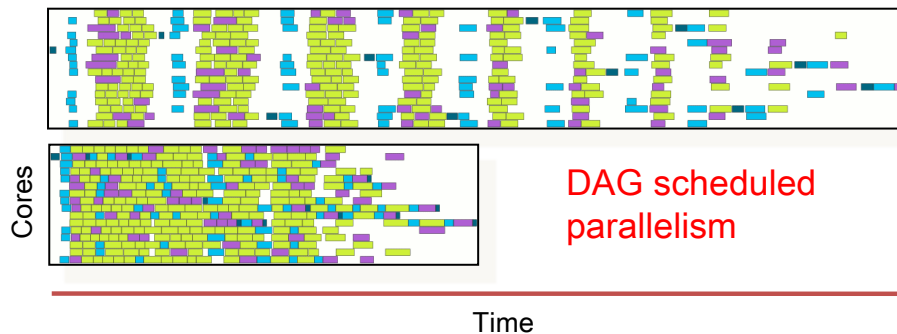
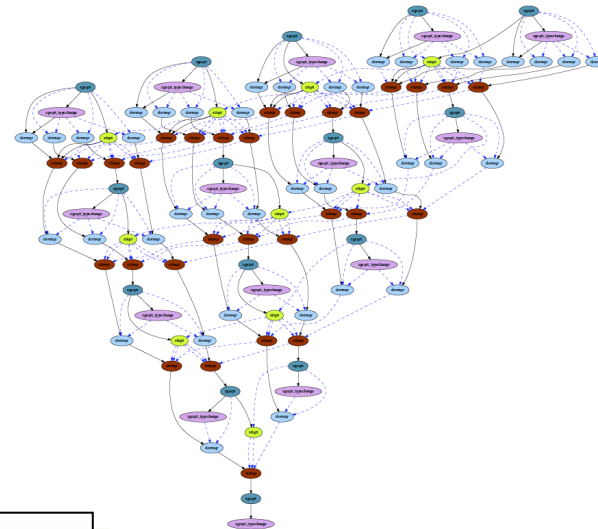


# Synchronization (in LAPACK)



# Dataflow Based Design

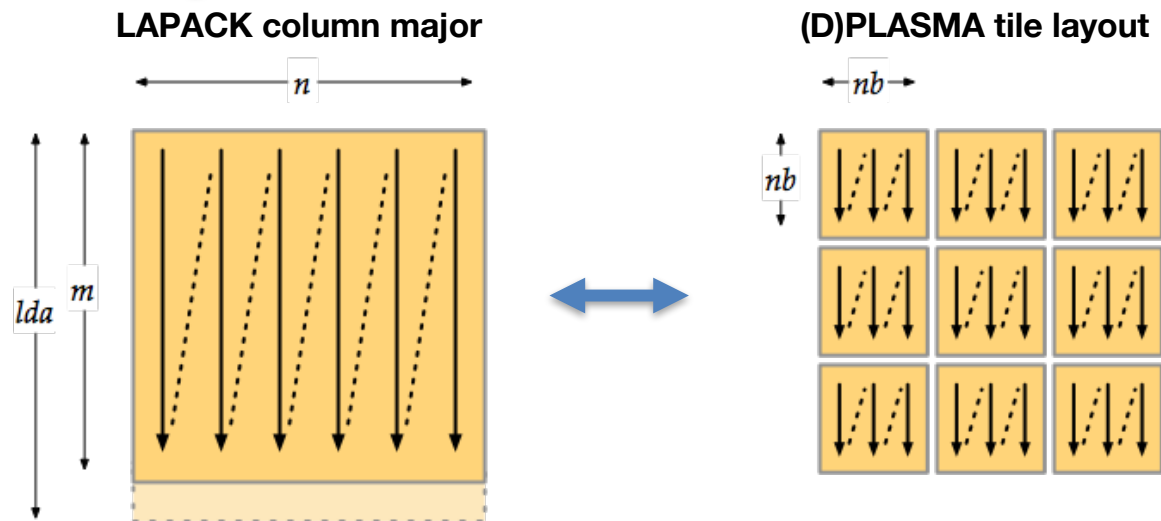
- **Objectives**
  - High utilization of each core
  - Scaling to large number of cores
  - Synchronization reducing algorithms
- **Methodology**
  - Dynamic DAG scheduling
  - Explicit parallelism
  - Implicit communication
  - Fine granularity / block data layout
- **Arbitrary DAG with dynamic scheduling**



DAG scheduled  
parallelism

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

# Tile matrix layout

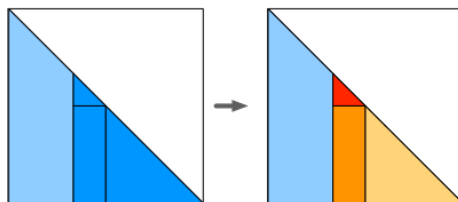


- Tiled layout

- Each tile is contiguous (column major)
- Enables dataflow scheduling
- Cache and TLB efficient (reduces conflict misses and false sharing)
- MPI messaging efficiency (zero-copy communication)
- In-place, parallel layout translation

# Tile algorithms: Cholesky

LAPACK Algorithm (right looking)

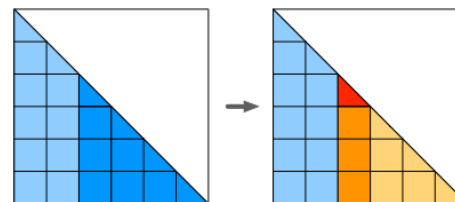


$$\text{red triangle} = \text{chol}(\text{blue triangle})$$

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} \text{blue} \\ \text{blue} \\ \text{blue} \end{bmatrix} / \text{red triangle} \quad \text{trsm}$$

$$\text{yellow triangle} = \text{blue triangle} - \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \begin{bmatrix} 1^T & 2^T & 3^T \end{bmatrix} \quad \text{syrk}$$

Tile Algorithm

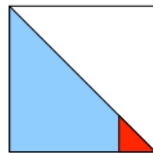
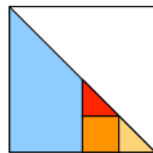
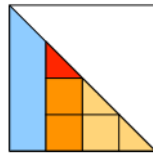
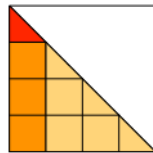


$$\text{red triangle} = \text{chol}(\text{blue triangle})$$

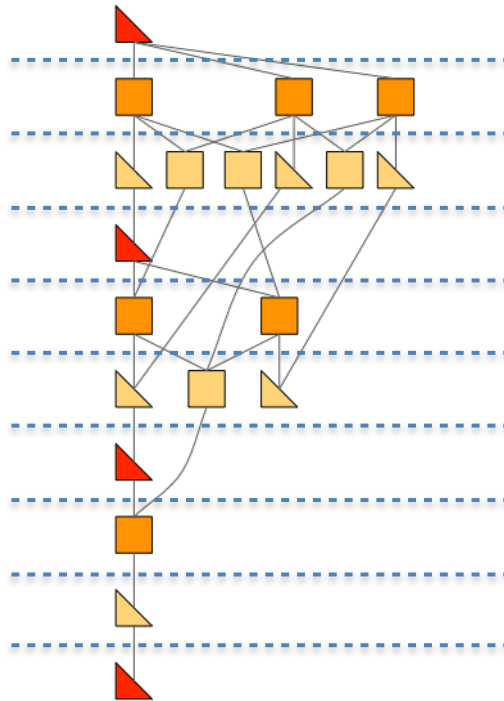
$$\begin{aligned} 1 &= \text{blue} / \text{red triangle} && \text{trsm} \\ 2 &= \text{blue} / \text{red triangle} && \text{trsm} \\ 3 &= \text{blue} / \text{red triangle} && \text{trsm} \end{aligned}$$

$$\begin{aligned} \text{yellow triangle} &= \text{blue triangle} - \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \begin{bmatrix} 1^T & 2^T & 3^T \end{bmatrix} && \text{syrk} \\ \text{orange triangle} &= \text{blue triangle} - \begin{bmatrix} 2 \\ 3 \end{bmatrix} \begin{bmatrix} 1^T \end{bmatrix} && \text{gemm} \\ \text{orange triangle} &= \text{blue triangle} - \begin{bmatrix} 3 \end{bmatrix} \begin{bmatrix} 1^T \end{bmatrix} && \text{gemm} \\ \text{yellow triangle} &= \text{blue triangle} - \begin{bmatrix} 2 \\ 3 \end{bmatrix} \begin{bmatrix} 2^T \end{bmatrix} && \text{syrk} \\ \text{orange triangle} &= \text{blue triangle} - \begin{bmatrix} 2 \\ 3 \end{bmatrix} \begin{bmatrix} 3^T \end{bmatrix} && \text{gemm} \\ \text{yellow triangle} &= \text{blue triangle} - \begin{bmatrix} 3 \end{bmatrix} \begin{bmatrix} 3^T \end{bmatrix} && \text{syrk} \end{aligned}$$

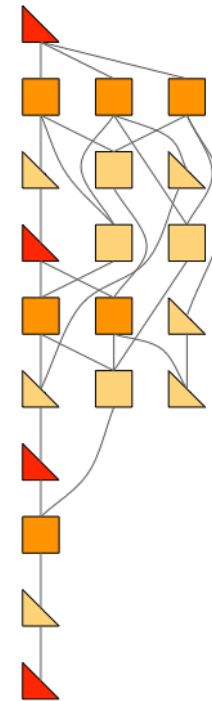
# Track dependencies — Directed acyclic graph (DAG)



Classical fork-join schedule  
with loop synchronizations



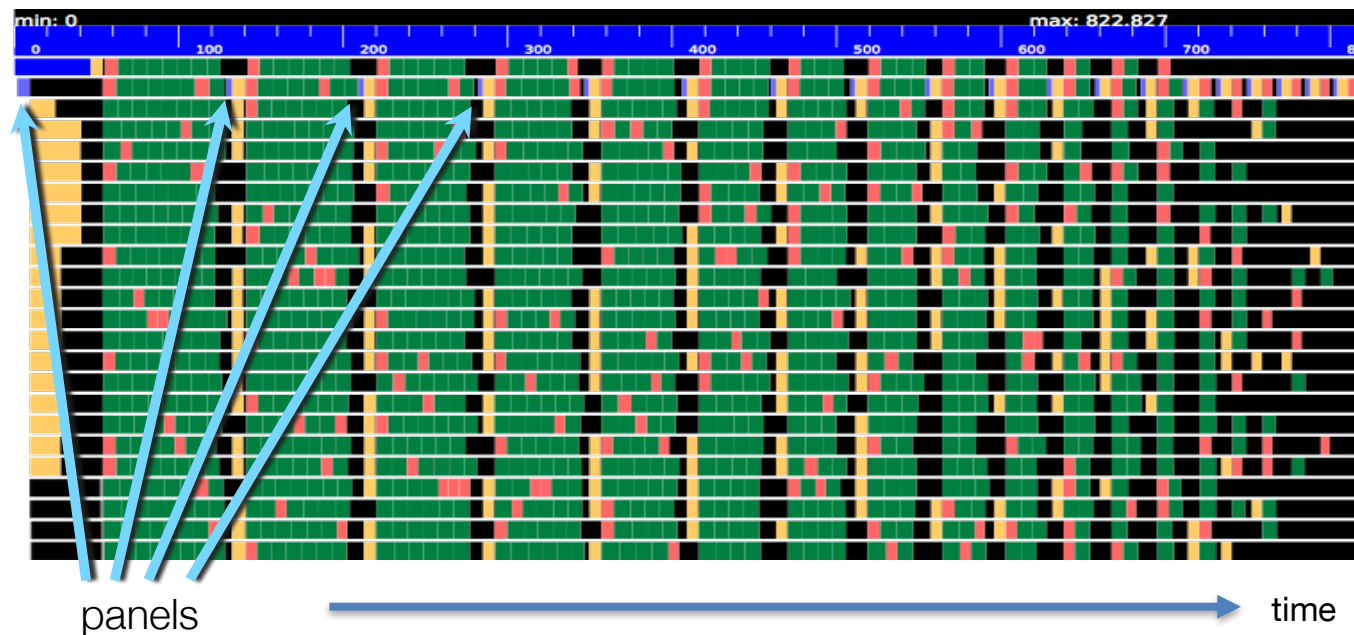
Reordered for 3 cores,  
without synchronizations





# Execution trace

- LAPACK-style fork-join leave cores idle

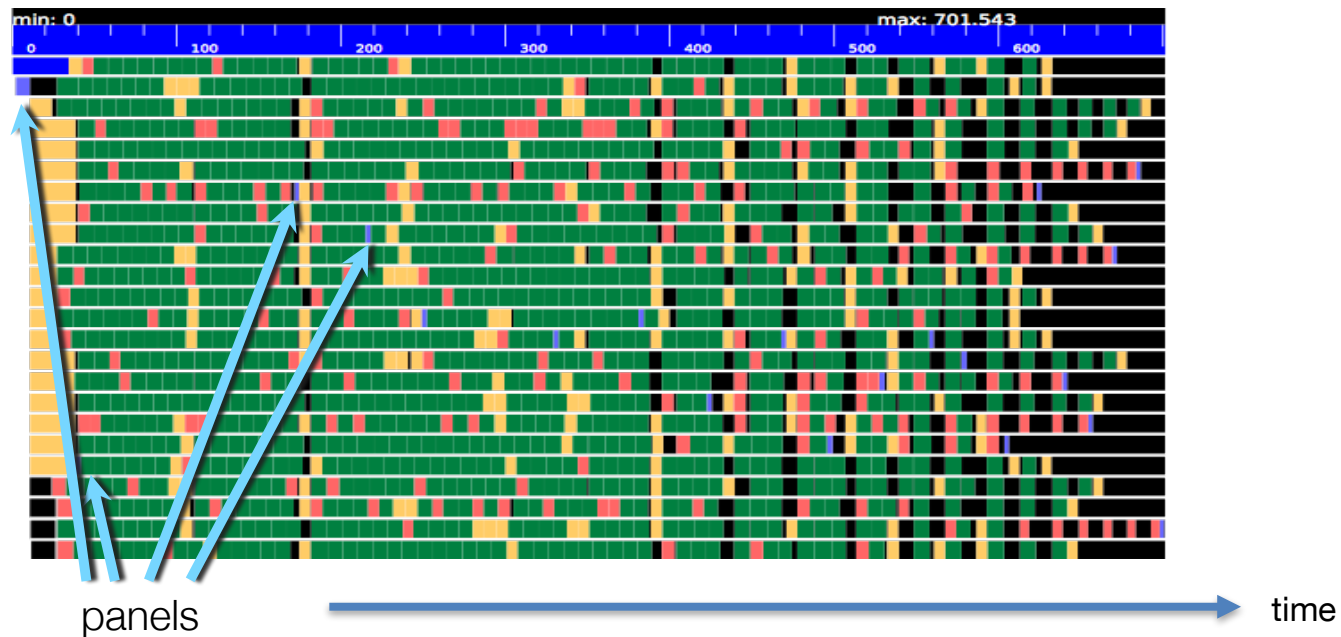


24 cores  
Matrix is 8000 x 8000, tile size is 400 x 400.

potrf trsm syrk gemm idle

# Execution trace

- PLASMA squeezes out idle time

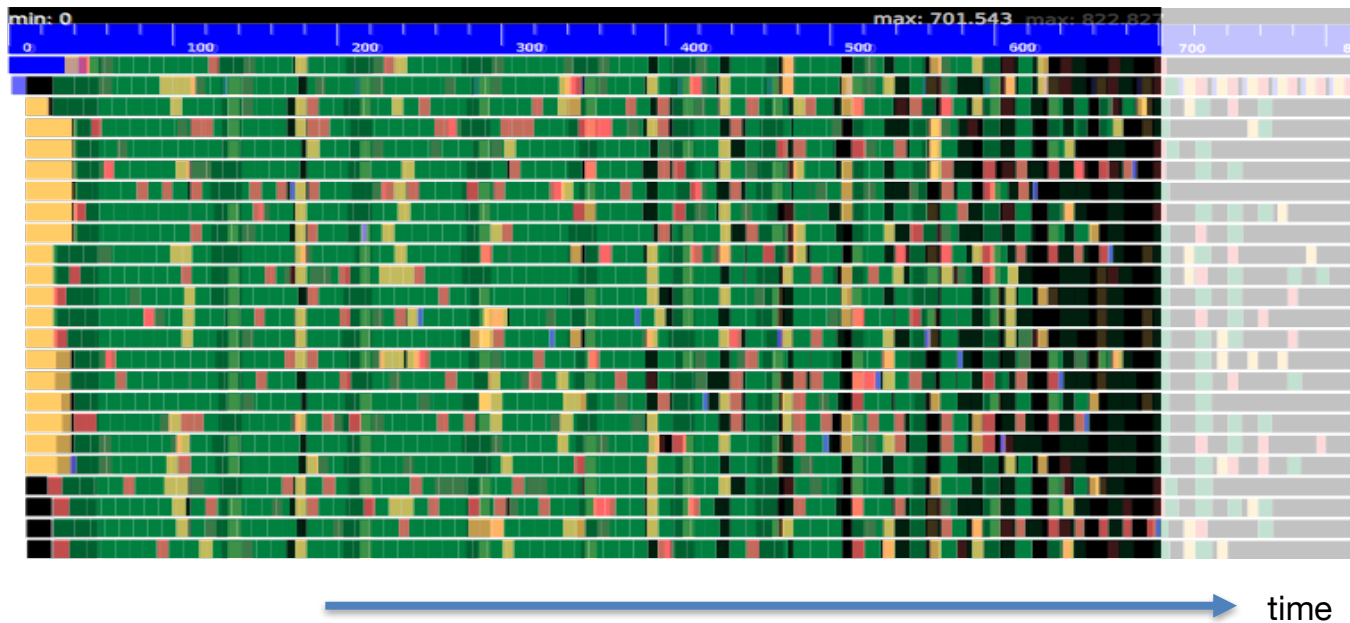


24 cores  
Matrix is 8000 x 8000, tile size is 400 x 400.

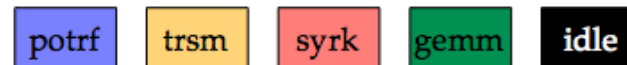
potrf trsm syrk gemm idle

# Execution trace

- PLASMA squeezes out idle time



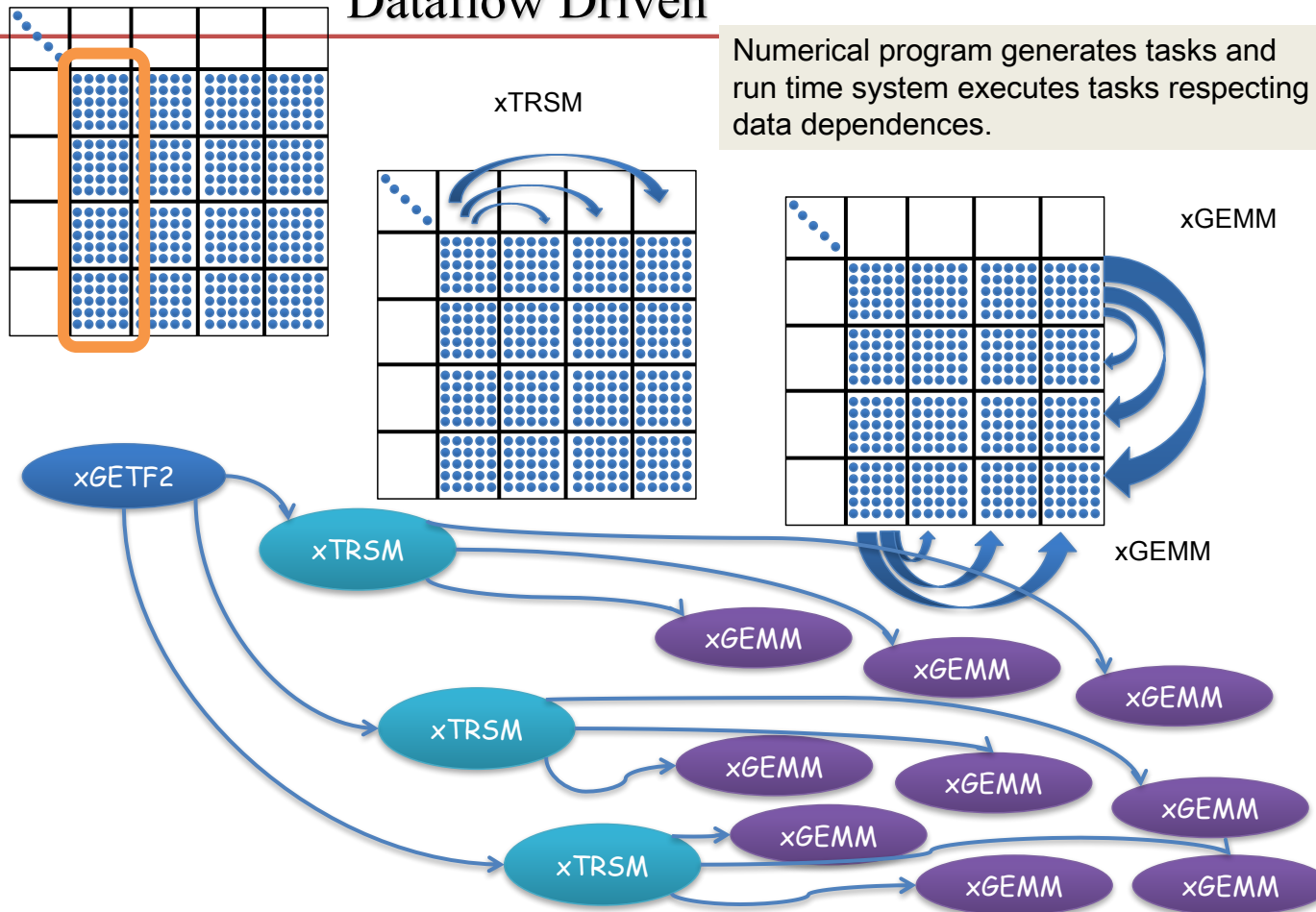
24 cores  
Matrix is 8000 x 8000, tile size is 400 x 400.



# PLASMA Factorization

## Dataflow Driven

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





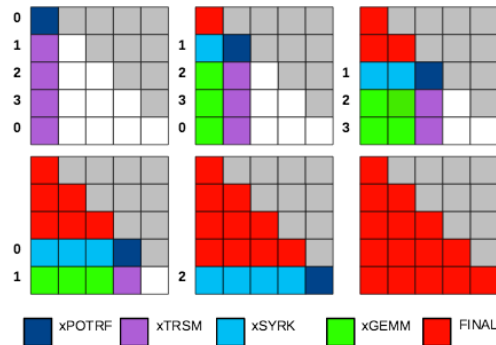
- 
- ```

graph TD
    POTRF((POTRF)) --> TRSM1((TRSM))
    POTRF --> TRSM2((TRSM))
    POTRF --> TRSM3((TRSM))
    TRSM1 --> SYRK1((SYRK))
    TRSM1 --> GEMM1((GEMM))
    TRSM2 --> SYRK2((SYRK))
    TRSM2 --> GEMM2((GEMM))
    TRSM3 --> SYRK3((SYRK))
    TRSM3 --> GEMM3((GEMM))
    SYRK1 --> POTRF2((POTRF2))
    SYRK1 --> SYRK4((SYRK))
    GEMM1 --> POTRF2
    GEMM1 --> GEMM4((GEMM))
    SYRK2 --> POTRF2
    SYRK2 --> SYRK5((SYRK))
    GEMM2 --> POTRF2
    GEMM2 --> GEMM4
    SYRK3 --> POTRF3((POTRF3))
    SYRK3 --> SYRK6((SYRK))
    GEMM3 --> POTRF3
    GEMM3 --> GEMM4
    SYRK4 --> TRSM4((TRSM))
    SYRK5 --> TRSM4
    GEMM4 --> TRSM4
    SYRK6 --> SYRK7((SYRK))
    TRSM4 --> SYRK7
    POTRF2 --> POTRF3
    POTRF3 --> POTRF4((POTRF))

```



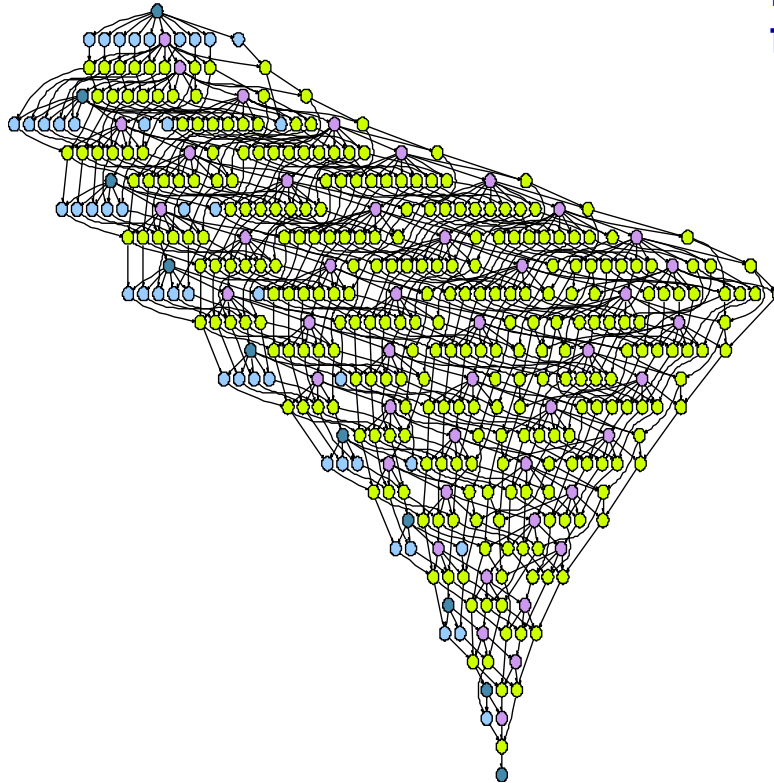
# Tiled Cholesky Decomposition



```
#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) ); }
            }
        }
    }
}
```

# PLASMA Local Scheduling

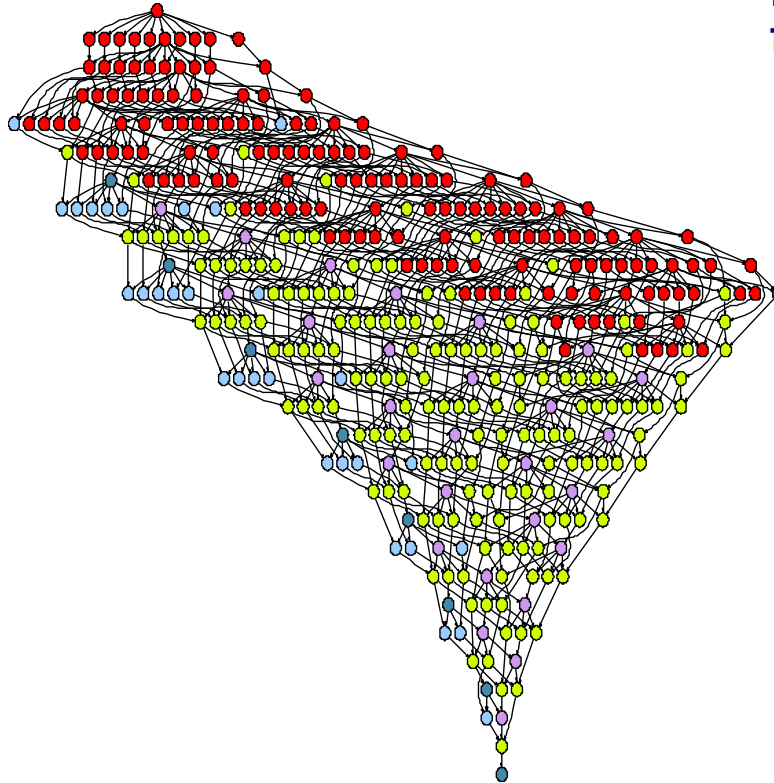
Dynamic Scheduling: Sliding Window



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

# PLASMA Local Scheduling

Dynamic Scheduling: Sliding Window

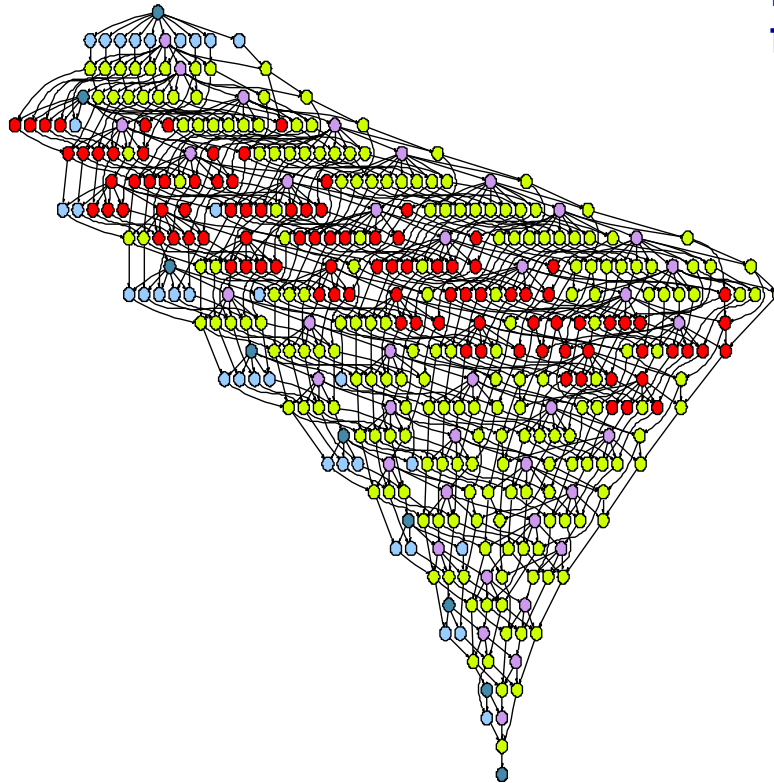


- DAGs get very big, very fast
  - So windows of active tasks are used; this means no global critical path
  - Matrix of  $NB \times NB$  tiles;  $NB^3$  operation
    - $NB=100$  gives 1 million tasks



# PLASMA Local Scheduling

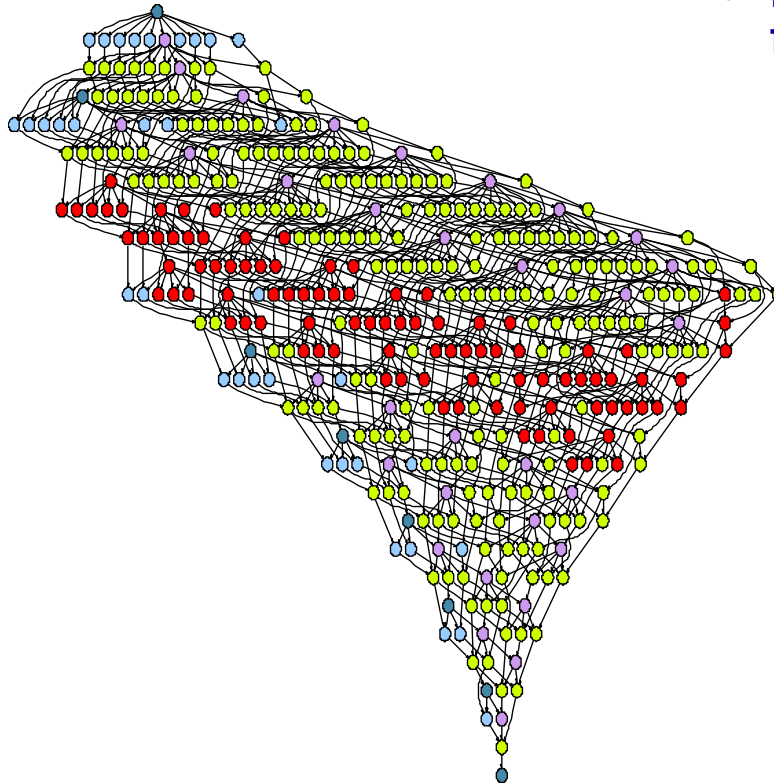
Dynamic Scheduling: Sliding Window



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

# PLASMA Local Scheduling

Dynamic Scheduling: Sliding Window

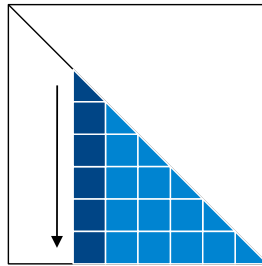


- DAGs get very big, very fast
  - So windows of active tasks are used; this means no global critical path
  - Matrix of  $NB \times NB$  tiles;  $NB^3$  operation
    - $NB=100$  gives 1 million tasks

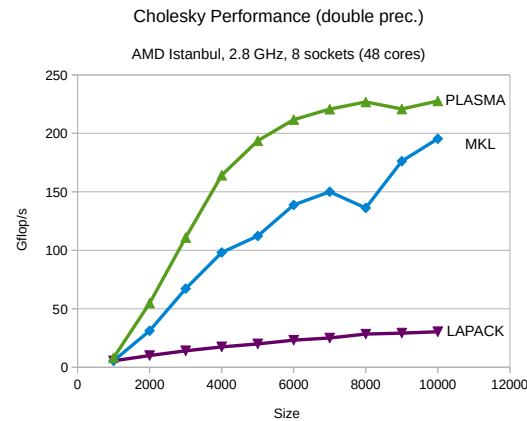
# Algorithms

Cholesky

PLASMA\_<sub>[scdz]</sub>potrf[\_Tile][\_Async]()



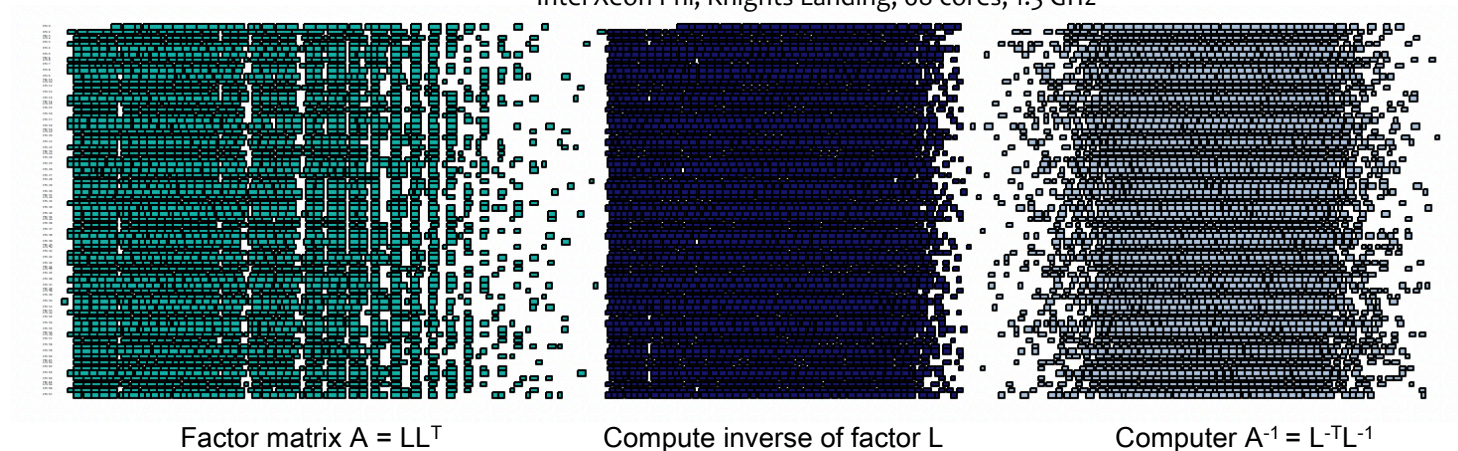
- **Algorithm**
  - equivalent to LAPACK
- **Numerics**
  - same as LAPACK
- **Performance**
  - comparable to vendor on few cores
  - much better than vendor on many cores



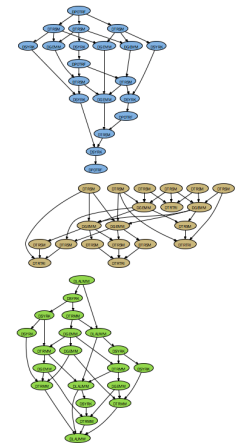


# PLASMA – Inverse of the Variance-Covariance Matrix

Cholesky inversion using OpenMP  
tiles of size 288 x 288, (7200 x 7200)  
Intel Xeon Phi, Knights Landing, 68 cores, 1.3 GHz



sync:  
770 Gflop/s

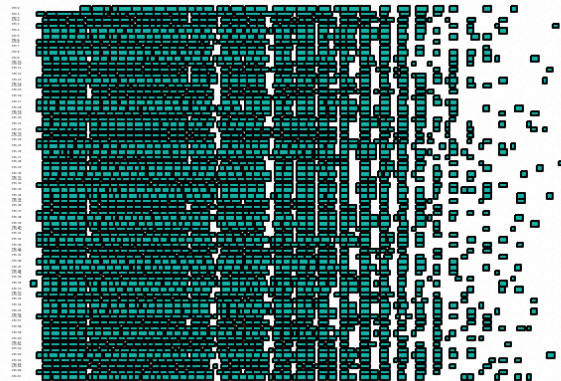


Assume a  $t$  by  $t$  matrix  
tiling then Cholesky  
Factorization alone:  $3t-2$   
Total:  $25(7t-3)$

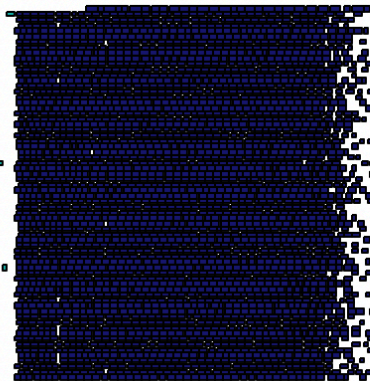


# PLASMA – Inverse of the Variance-Covariance Matrix

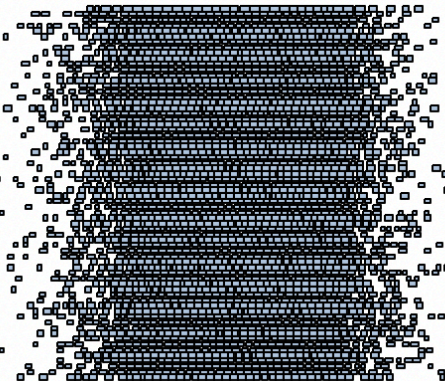
Cholesky inversion using OpenMP  
 tiles of size 288 x 288, (7200 x 7200)  
 Intel Xeon Phi, Knights Landing, 68 cores, 1.3 GHz



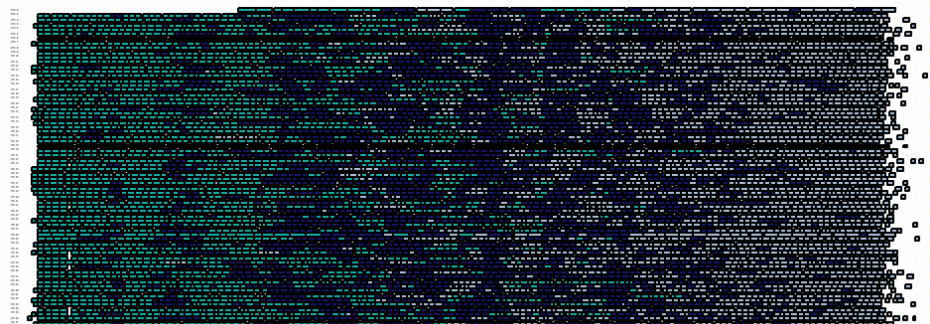
Factor matrix  $A = LL^T$



Compute inverse of factor L

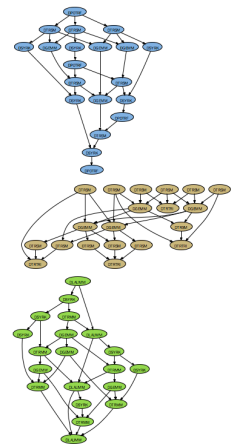


Computer  $A^{-1} = L^{-T} L^{-1}$

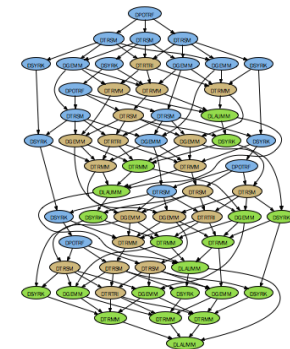


sync:  
770 Gflop/s

async:  
1001 Gflop/s



Assume a  $t$  by  $t$  matrix  
 tiling then Cholesky  
 Factorization alone:  $3t-2$   
 Total:  $25(7t-3)$



Total:  $18(3t+6)$

# Emerging software solutions

## • PLASMA

- Tile layout & algorithms
- Dynamic scheduling — OpenMP 4

## • MAGMA

- Hybrid multicore + accelerator (GPU, Xeon Phi)
- Block algorithms (LAPACK style)
- Standard layout/Static scheduling

## • DPLASMA — PaRSEC

- Distributed
- Tile layout & algorithms
- Dynamic scheduling — parameterized task graph

## • SLATE – DOE ECP Project

- DPLAMA Hybrid
- C++
- Update to state-of-the-art algorithms





## API for Batching BLAS Operations

---

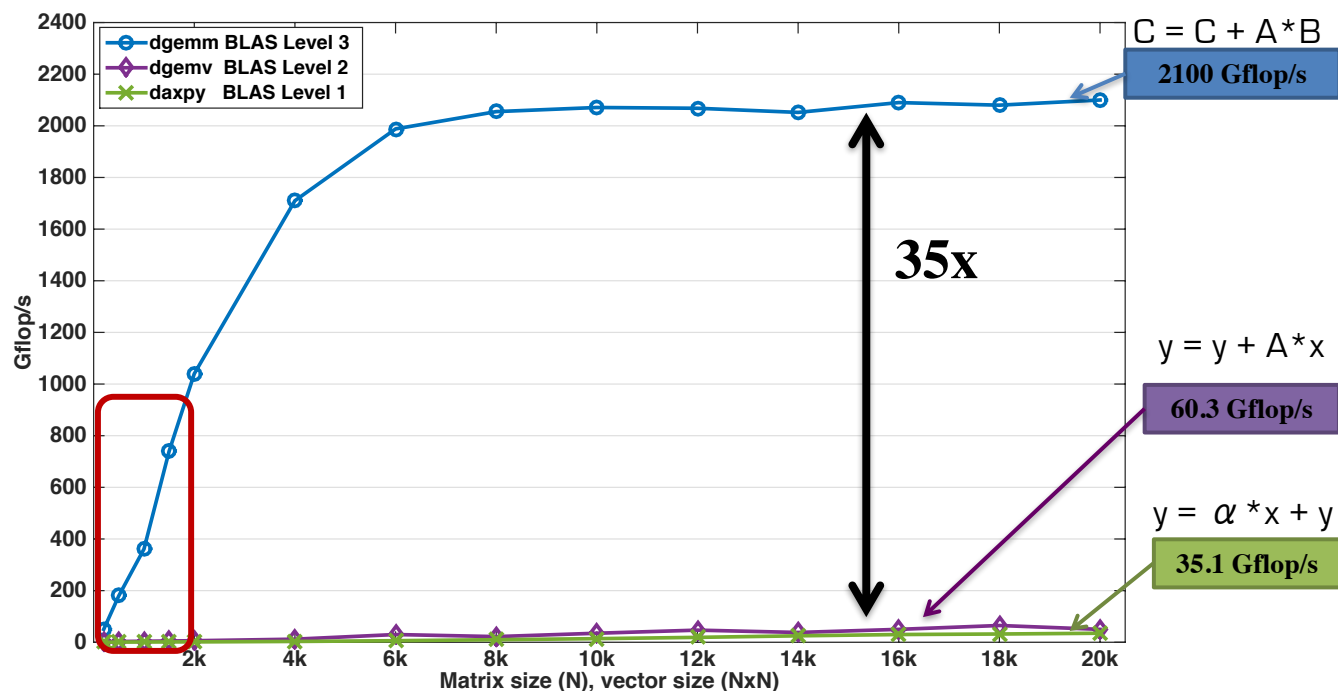
- We are proposing, as a community standard, an API for Batched Basic Linear Algebra Operations
- The focus is on multiple independent BLAS operations
  - Think “small” matrices ( $n < 500$ ) that are operated on in a single routine.
- Goal to be more efficient and portable for multi/manycore & accelerator systems.
- We can show 2x speedup and 3x better energy efficiency.



## Level 1, 2 and 3 BLAS



68 cores Intel Xeon Phi KNL, 1.3 GHz, Peak DP = 2662 Gflop/s



68 cores Intel Xeon Phi KNL, 1.3 GHz  
The theoretical peak double precision is 2662 Gflop/s  
Compiled with icc and using Intel MKL 2017b1 20160506

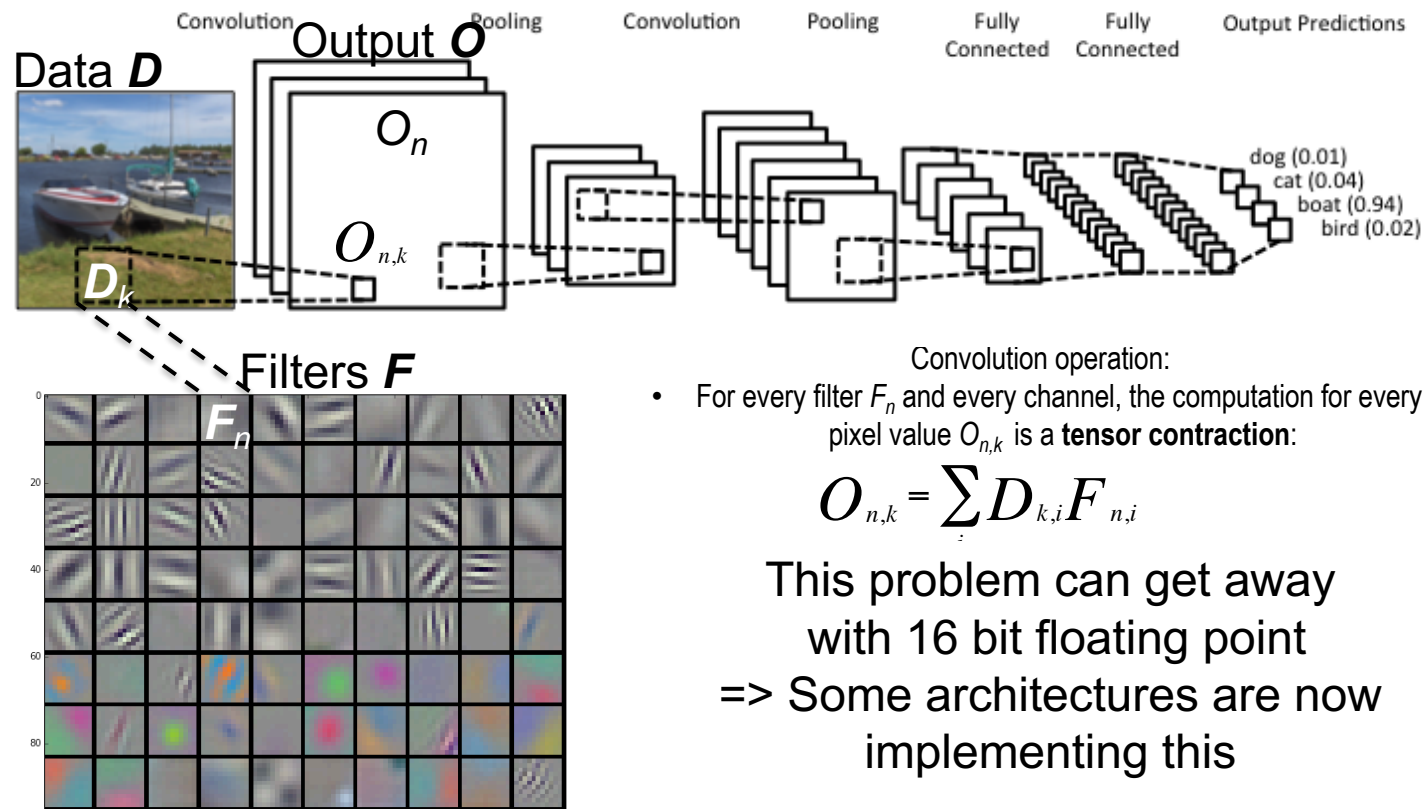


# Machine Learning

## Need of Batched and/or Tensor contraction routines in machine learning –

e.g., Convolutional Neural Networks (CNNs) used in computer vision

Key computation is convolution of Filter  $F_i$  (feature detector) and input image  $D$  (data):

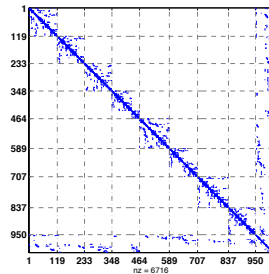
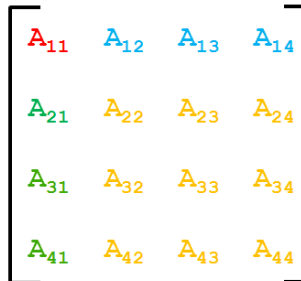


# Examples

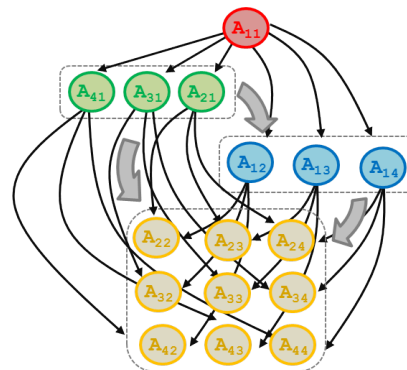
## Need of **Batched** routines for Numerical LA

[ e.g., sparse direct multifrontal methods, preconditioners for sparse iterative methods, tiled algorithms in dense linear algebra, etc.; ]  
 [ collaboration with Tim Davis at al., Texas A&M University]

### Sparse / Dense Matrix System



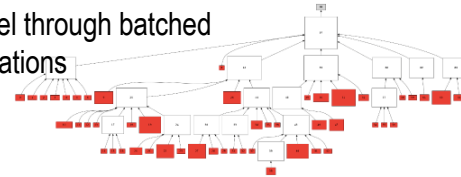
### DAG-based factorization



### To capture main LA patterns needed in a numerical library for **Batched LA**

- LU, QR, or Cholesky on small diagonal matrices
- TRSMs, QRs, or LUs
- TRSMs, TRMMs
- Updates (Schur complement) GEMMs, SYRKs, TRMMs

- Example matrix from Quantum chromodynamics
- Reordered and ready for sparse direct multifrontal solver
- Diagonal blocks can be handled in parallel through batched LU, QR, or Cholesky factorizations

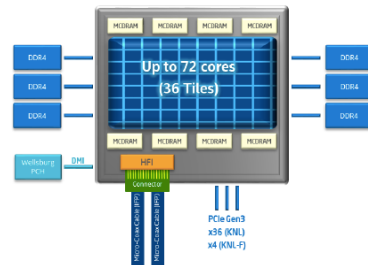
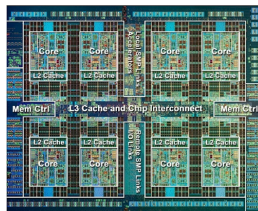


# MAGMA Batched Computations CPU

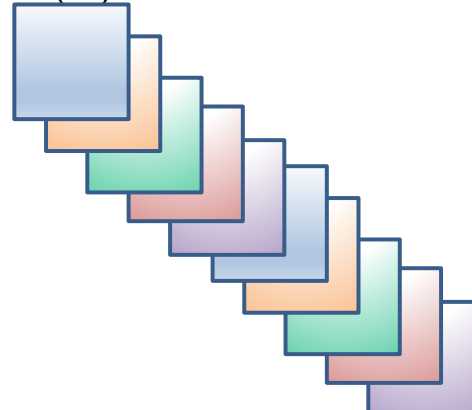
## 1. Non-batched computation

loop over the matrices one by one and compute either:

- One call for each matrix.
- Sequentially wasting all the other cores, and attaining very poor performance
- Or using multithread (note that for small matrices there is not enough work for all cores so expect low efficiency as well as threads contention can affect the performance)



```
for (i=0; i<batchcount; i++)  
  dgemm(...)
```

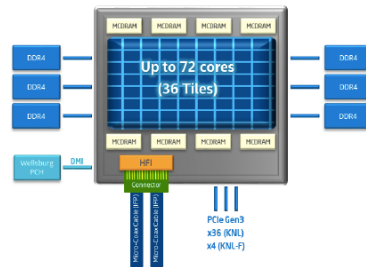
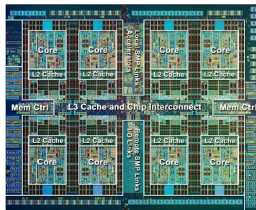


# MAGMA Batched Computations CPU

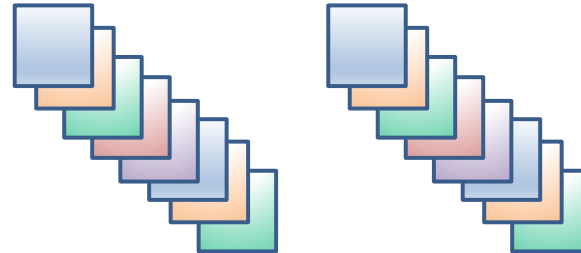
## 2. Batched computation

loop over the matrices and assign a matrix to each core working on it sequentially and independently

- Since matrices are very small, all the  $n\_cores$  matrices will fit into L2 cache thus we do not increase L2 cache misses while performing in parallel  $n\_cores$  computations reaching the best of each core



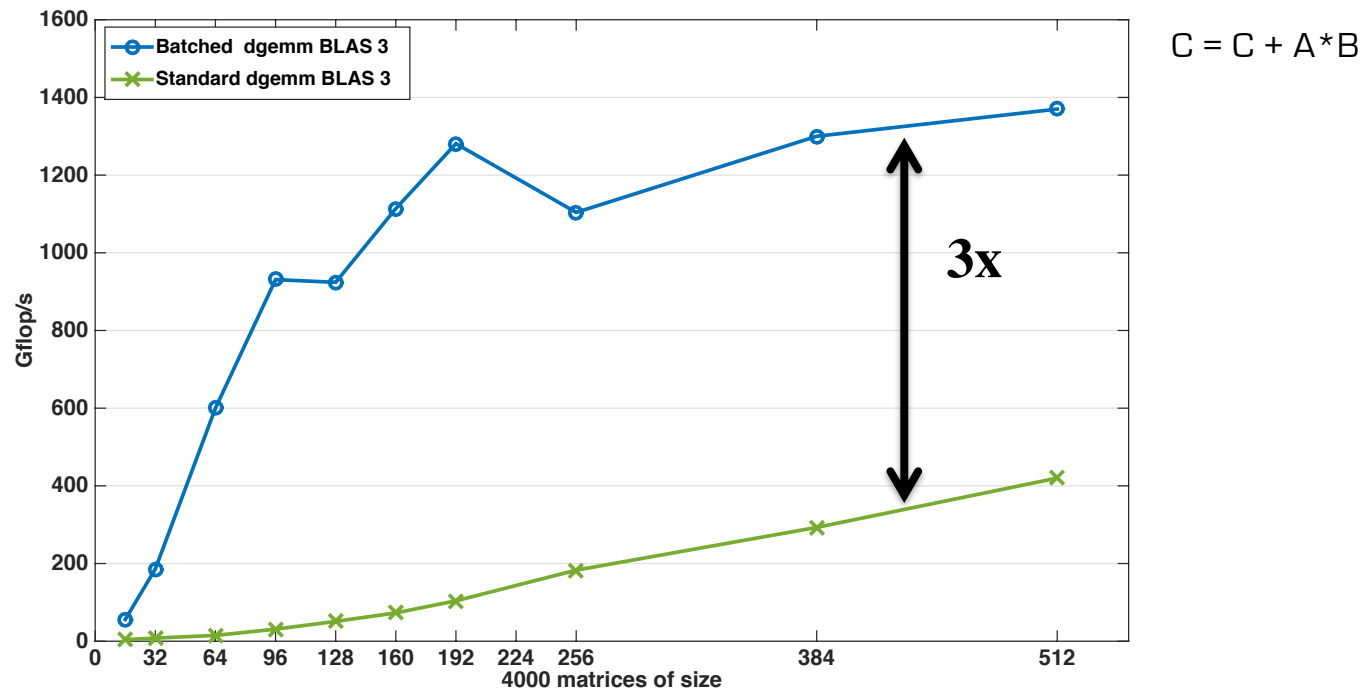
```
for (i=cpu_id; i<batchcount; i+=n_cpu)
    batched_dgemm(...)
```





## Level 1, 2 and 3 BLAS

68 cores Intel Xeon Phi KNL, 1.3 GHz, Peak DP = 2662 Gflop/s



68 cores Intel Xeon Phi KNL, 1.3 GHz  
The theoretical peak double precision is 2662 Gflop/s  
Compiled with icc and using Intel MKL 2017b1 20160506

# 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;  $\Delta x$  rather than  $x$ .

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



## 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.

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

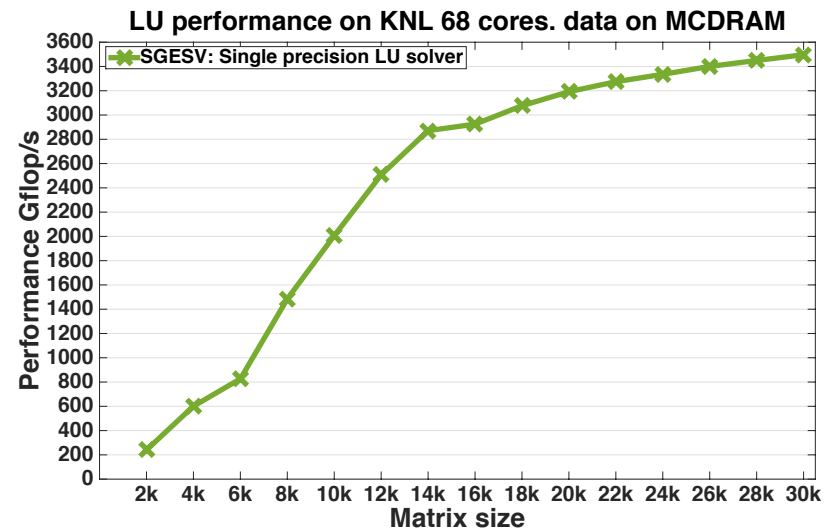
- Wilkinson, Moler, Stewart, & Higham provide error bound for SP fl pt results when using DP fl pt.
- It can be shown that using this approach we can compute the solution to 64-bit floating point precision.

- 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;  $O(10^8)$



# Power-awareness in Algorithms

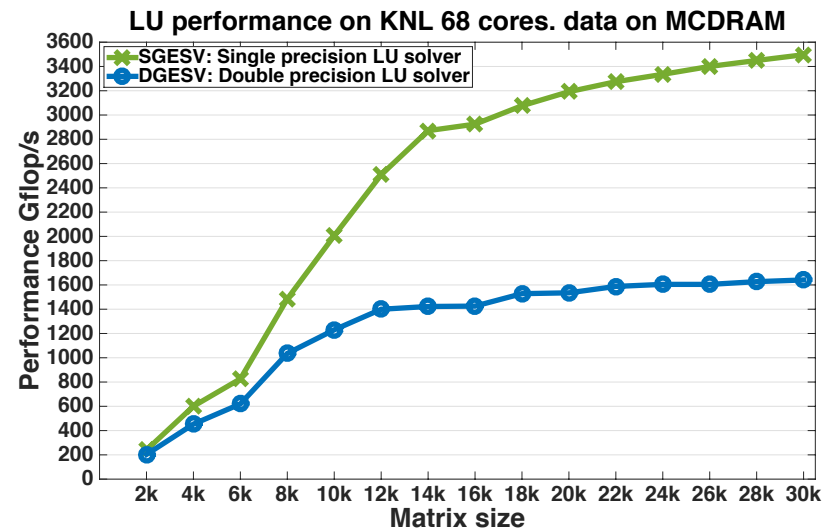
Iterative refinement to solve  $Ax=b$  getting a solution in double precision arithmetic





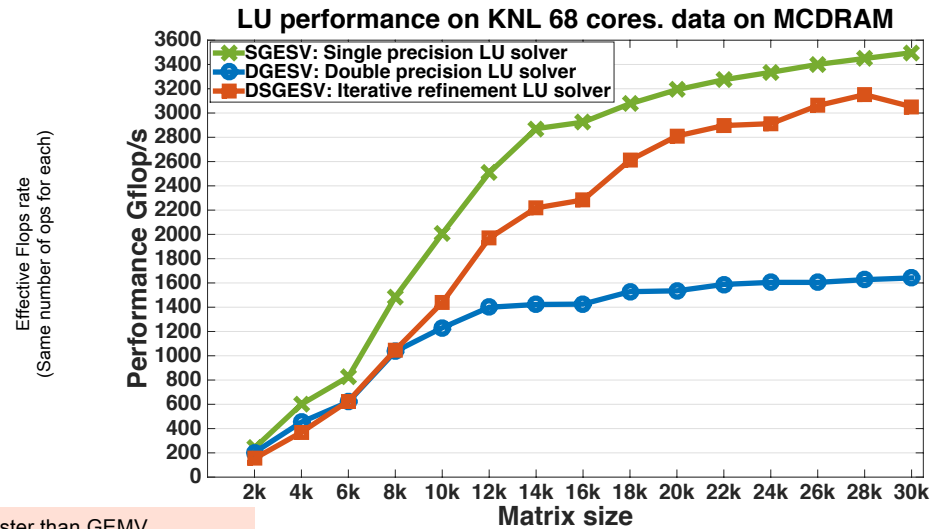
# Power-awareness in Algorithms

Iterative refinement to solve  $Ax=b$  getting a solution in double precision arithmetic



# Power-awareness in Algorithms

Iterative refinement to solve  $Ax=b$  getting a solution in double precision arithmetic

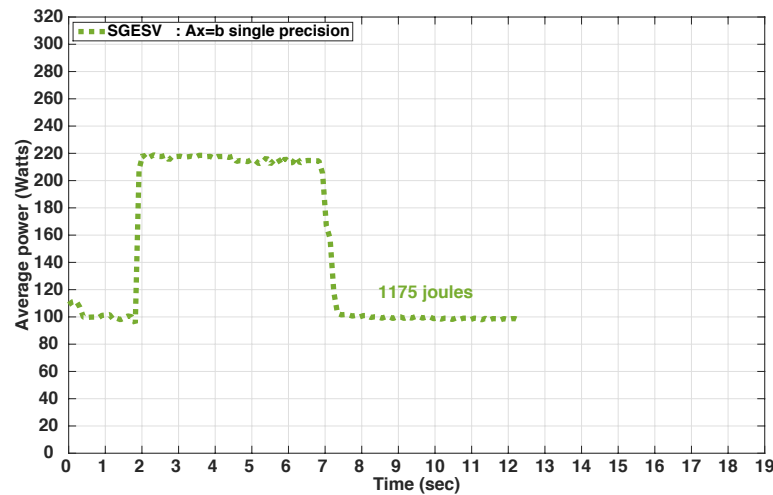


GEMM 30X faster than GEMV  
GEMM 50X faster than TRSV



# Power-awareness in Algorithms

Iterative refinement to solve  $Ax=b$  getting a solution in double precision arithmetic

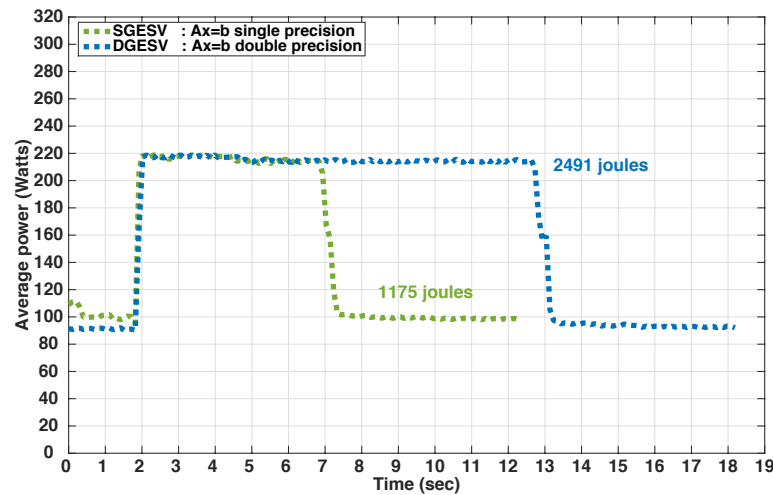


- Power consumption of SP, DP, and mixed precision algorithm to solve  $Ax=b$  for a matrix of size 30K on KNL.



# Power-awareness in Algorithms

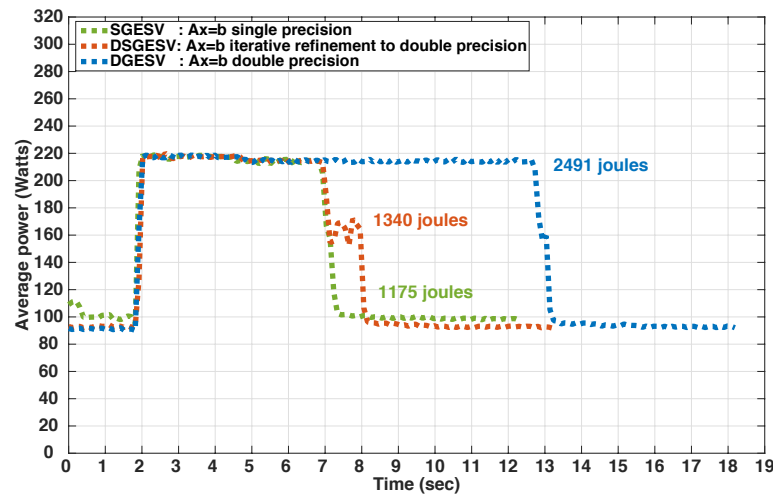
Iterative refinement to solve  $Ax=b$  getting a solution in double precision arithmetic



- Power consumption of SP, DP, and mixed precision algorithm to solve  $Ax=b$  for a matrix of size 30K on KNL.

# Power-awareness in Algorithms

Iterative refinement to solve  $Ax=b$  getting a solution in double precision arithmetic

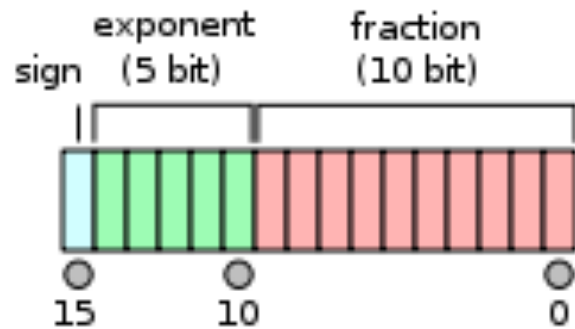


- Power consumption of SP, DP, and mixed precision algorithm to solve  $Ax=b$  for a matrix of size 30K on KNL.
- Algorithmic advancements such as mixed precision techniques can also provide a large gain in energy and power consumption. We can reduce the energy by about the half.



# IEEE 754 Half Precision (16-bit) Floating Pt Standard

A lot of interest driven by “machine learning”



Range =  $10^{\pm 5}$



UP TO 400GB DIRECT MEMORY ACCESS VS 16GB WITH A GPU

NEAR LINEAR SCALING REDUCTION IN TIME TO TRAIN 31X WHEN SCALING TO 32 NODES

KNIGHTS MILL Next Gen Xeon Phi AVAILABLE 2017 4X DEEP LEARNING PERFORMANCE

INTEL XEON PHI RESULTS NOV'16 TOP500 LIST +45 PELOPS 80% NEW SYSTEM ACCELERATION FLOPS

70 / 57

| AMD Radeon Instinct     |              |               |                                       |
|-------------------------|--------------|---------------|---------------------------------------|
|                         | Instinct MI6 | Instinct MI8  | Instinct MI25                         |
| Memory Type             | 16GB GDDR5   | 4GB HBM       | "High Bandwidth Cache and Controller" |
| Memory Bandwidth        | 224GB/sec    | 512GB/sec     | ?                                     |
| Single Precision (FP32) | 5.7 TFLOPS   | 8.2 TFLOPS    | 12.5 TFLOPS                           |
| Half Precision (FP16)   | 5.7 TFLOPS   | 8.2 TFLOPS    | 25 TFLOPS                             |
| TDP                     | <150W        | <175W         | <300W                                 |
| Cooling                 | Passive      | Passive (SFF) | Passive                               |
| GPU                     | Polaris 10   | Fiji          | Vega                                  |
| Manufacturing Process   | GloFo 14nm   | TSMC 28nm     | ?                                     |

| Tesla Product             | Tesla K40      | Tesla M40       | Tesla P100     | Tesla V100    |
|---------------------------|----------------|-----------------|----------------|---------------|
| GPU                       | GK110 (Kepler) | GM200 (Maxwell) | GP100 (Pascal) | GV100 (Volta) |
| SIMs                      | 15             | 24              | 56             | 80            |
| TPCs                      | 15             | 24              | 28             | 40            |
| FP32 Cores / SM           | 192            | 128             | 64             | 64            |
| FP32 Cores / GPU          | 2880           | 3072            | 3584           | 5120          |
| FP64 Cores / SM           | 64             | 4               | 32             | 32            |
| FP64 Cores / GPU          | 960            | 96              | 1792           | 2560          |
| Tensor Cores / SM         | NA             | NA              | NA             | 8             |
| Tensor Cores / GPU        | NA             | NA              | NA             | 640           |
| GPU Boost Clock           | 810/875 MHz    | 1114 MHz        | 1480 MHz       | 1455 MHz      |
| Peak FP32 TFLOP/s*        | 5.04           | 6.8             | 10.6           | 15            |
| Peak FP64 TFLOP/s*        | 1.68           | 2.1             | 5.3            | 7.5           |
| Peak Tensor Core TFLOP/s* | NA             | NA              | NA             | 120           |







# 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
  - Now we have 16 bit floating point as well
- **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

