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-2018-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.

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	
Chameleon	CeCILL-C	See authors	X	X		X		X	C	M	X								2014-04-15
DPLASMA	BSD	yes	X	X		X		X	C	M	X								2014-04-14
Eigen	Mozilla	yes	X	X			X	X			X	X		X	X	X			2015-01-21
Elemental	New BSD	yes	X	X			X			M	X	X	X	X					2014-11-08
ELPA	LGPL	yes	X	X	F90	X		X		M	X								2015-05-29
FLENS	BSD	yes	X	X			X	X			X								2014-05-11
hmat-oss	GPL	yes	X	X	X	X	X	X			X			X					2015-03-10
LAPACK	BSD	yes	X	X	X	X		X			X								2013-11-26
LAPACK95	BSD	yes	X	X	X			X			X								2000-11-30
libflame	New BSD	yes	X	X	X	X		X			X								2014-03-18
MAGMA	BSD	yes	X	X	X	X		X	C/O/X		X				X	X	X		2015-05-05
NAPACK	BSD	yes	X		X			X			X				X		X		?
PLAPACK	LGPL	yes	X	X	X	X				M	X								2007-06-12
PLASMA	BSD	yes	X	X	X	X		X			X								2015-04-27
rejtrix	by-nc-sa	yes	X				X	X			X				P	P			2013-10-01
ScaLAPACK	BSD	yes	X	X	X	X				M/P	X								2012-05-01
Trilinos/Pliris	BSD	yes	X	X		X	X			M	X								2015-05-07
ViennaCL	MIT	yes	X				X	X	C/O/X		X				X	X	X	X	2014-12-11

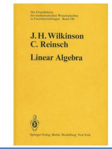
http://www.netlib.org/utk/people/JackDongarra/la-sw.html

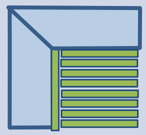
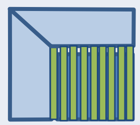
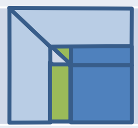
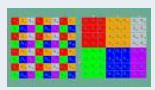
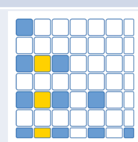
- LINPACK, EISPACK, LAPACK, ScaLAPACK
  - PLASMA, MAGMA

DLA Solvers

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

50 Years Evolving SW and Alg Tracking Hardware Developments



Software/Algorithms follow hardware evolution in time		
EISPACK (1970's) (Translation of Algol)	 	Rely on - Fortran, but row oriented
LINPACK (1980's) (Vector operations	 	Rely on - Level-1 BLAS operations - Column oriented
LAPACK (1990's) (Blocking, cache friendly)	 	Rely on - Level-3 BLAS operations
ScaLAPACK (2000's) (Distributed Memory)	 	Rely on - PBLAS Mess Passing
PLASMA (2010's) New Algorithms (many-core friendly)		Rely on - DAG/scheduler - block data layout - some extra kernels
SLATE (2020's)		Rely on C++ - Tasking DAG scheduling - Tiling, but tiles can come from anywhere - Batched Dispatch

What do you mean by performance?

- **What is a xflop/s?**
  - xflop/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 1.61 Tflop/s for the socket of 24 cores.

Peak Performance - Per Core

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

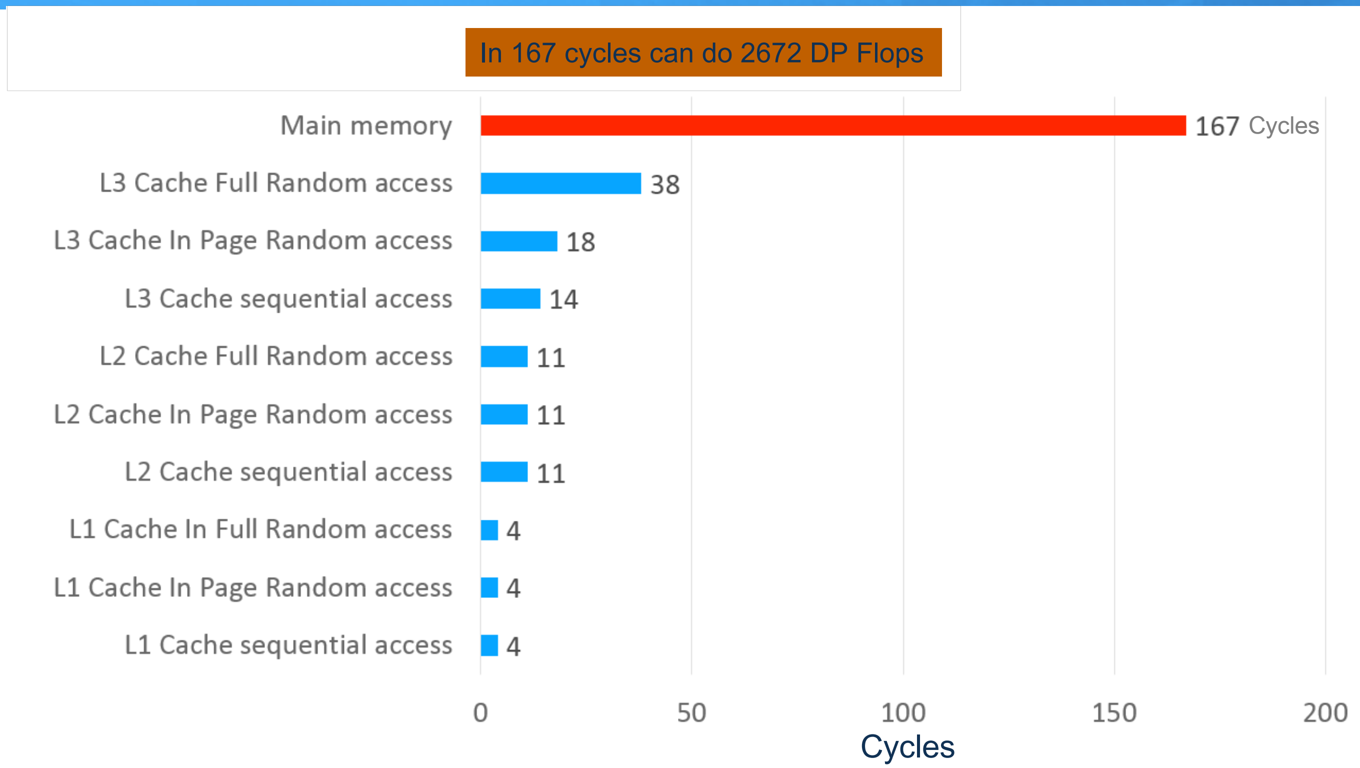
Floating point operations per cycle per core

- Most of the recent computers have FMA (Fused multiple add):
  (i.e. x ←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 (2009) & Westmere (2010) have SSE4**
  - 4 flops/cycle/core DP & 8 flops/cycle/core SP
- **Intel Xeon Sandy Bridge(2011) & Ivy Bridge (2012) have AVX**
  - 8 flops/cycle/core DP & 16 flops/cycle/core SP
- **Intel Xeon Haswell (2013) & Broadwell (2014) 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

We are here

- **Intel Xeon Skylake (server) & KNL AVX 512**
  - 32 flops/cycle/core DP & 64 flops/cycle/core SP
  - Skylake w/24 cores & ~~Xeon Phi (Knight's Landing) w/68 cores~~
- **Intel Xeon Cascade Lake**

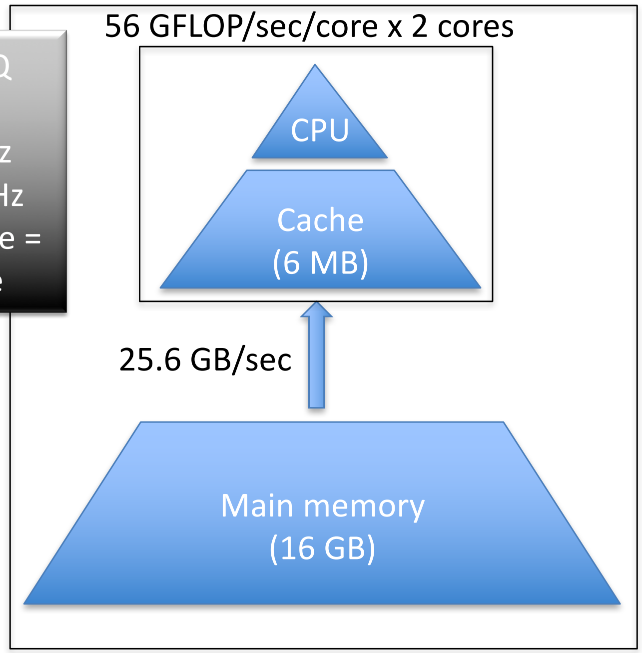
CPU Access Latencies in Clock Cycles



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

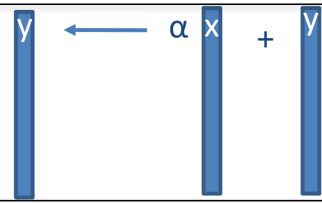


(Omitting latency here.)

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:

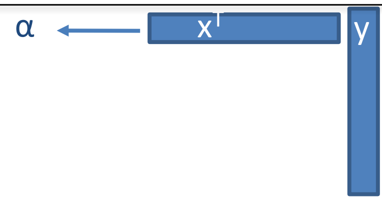



```
for ( j = 0; j < n; j++)
      y[i] += a * x[i];
```

(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];
```

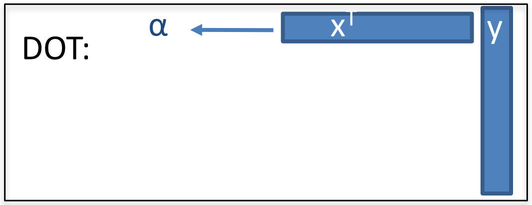
(without increment)

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 flops) / (56 Gflop/sec) = **0.013 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

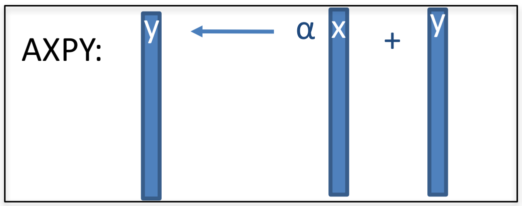
Performance for DOT ≤ 3.2 Gflop/s

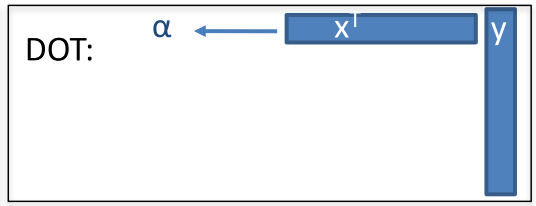
Peak is 56 Gflop/s

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

Level 1, 2 and 3 BLAS

Level 1 BLAS Matrix-Vector operations

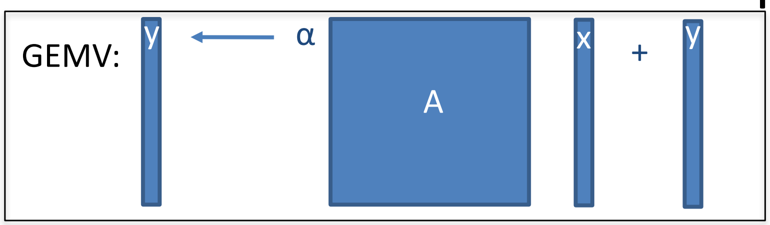




2n FLOPs
2n memory references
AXPY: 2n READ, n WRITE
DOT: 2n READ

RATIO Fl Pt Ops to Memory Ops: 1:1

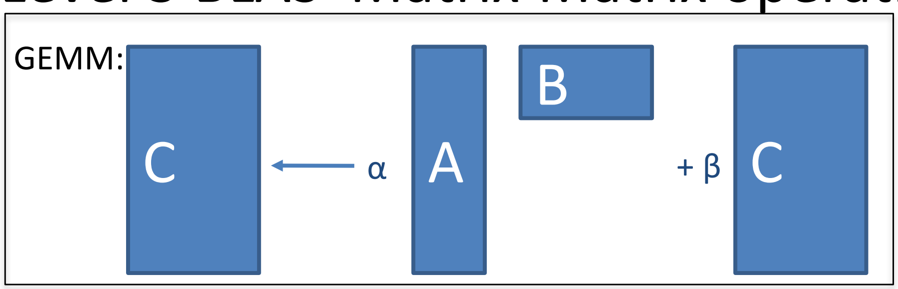
Level 2 BLAS Matrix-Vector operations



$2n^2$ FLOPs
n^2 memory references

RATIO Fl Pt Ops to Memory Ops: 2:1

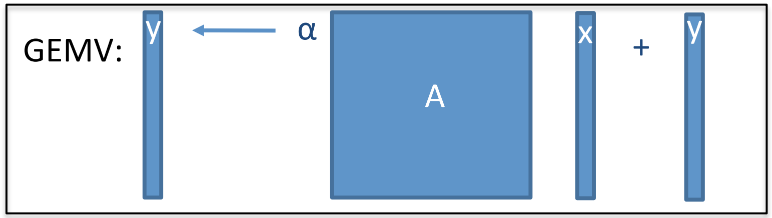
Level 3 BLAS Matrix-Matrix operations



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

RATIO Fl Pt Ops to Memory Ops: n:2

- 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 GEMV:
  – ($2n^2$ flops) / (56 Gflop/sec) = **0.026 ms**

Matrix - Vector Operations

total_time $\geq$ max (time_comm , time_comp)

= max (0.23ms , 0.026ms) = 0.23ms

Performance = (2 x 860^2 flops)/.23ms = 6.4 Gflop/s

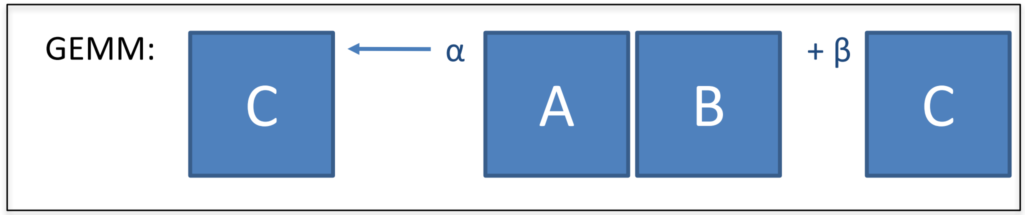
Performance for GEMV $\leq$ 6.4 Gflop/s

Performance for DOT $\leq$ 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:
  - ($2n^3$ flops) / (56 Gflop/sec) = **4.5 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)/4.5ms = 55.5 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 ≈ 55.5 Gflop/sec

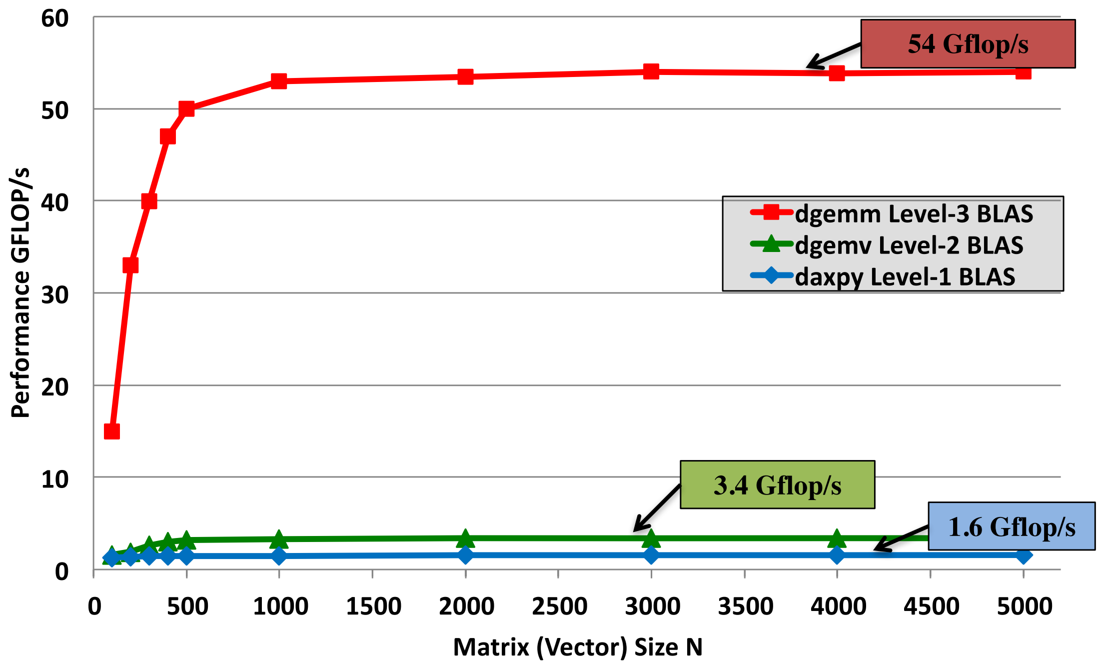
Performance for DOT ≤ 3.2 Gflop/s

Performance for GEMV ≤ 6.4 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?

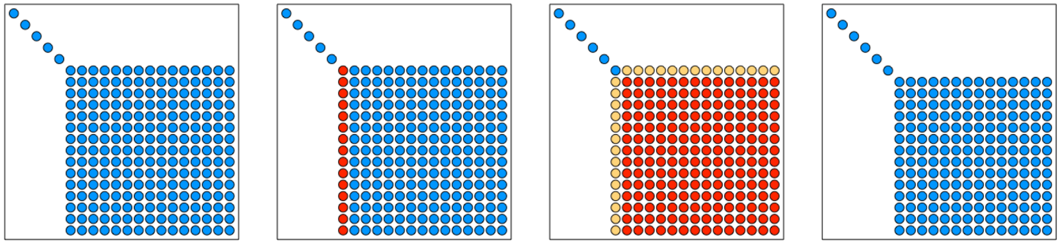
Issues

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

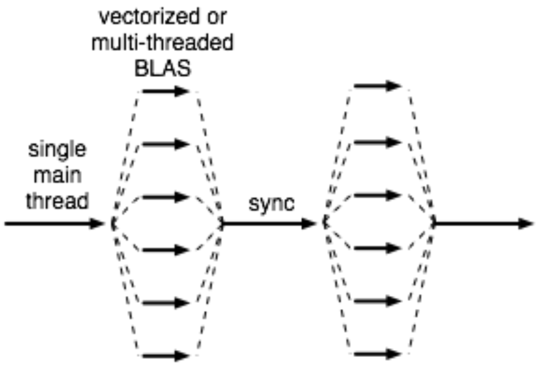
- Break matrices into blocks or tiles that will fit.

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

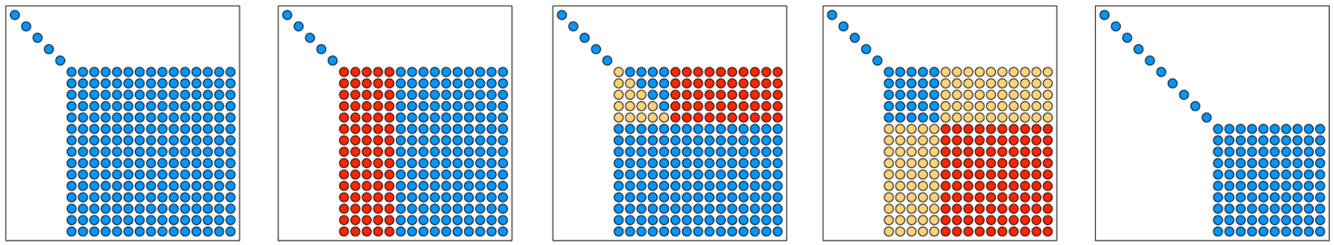
LU Factorization in LINPACK (1970's)



- 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



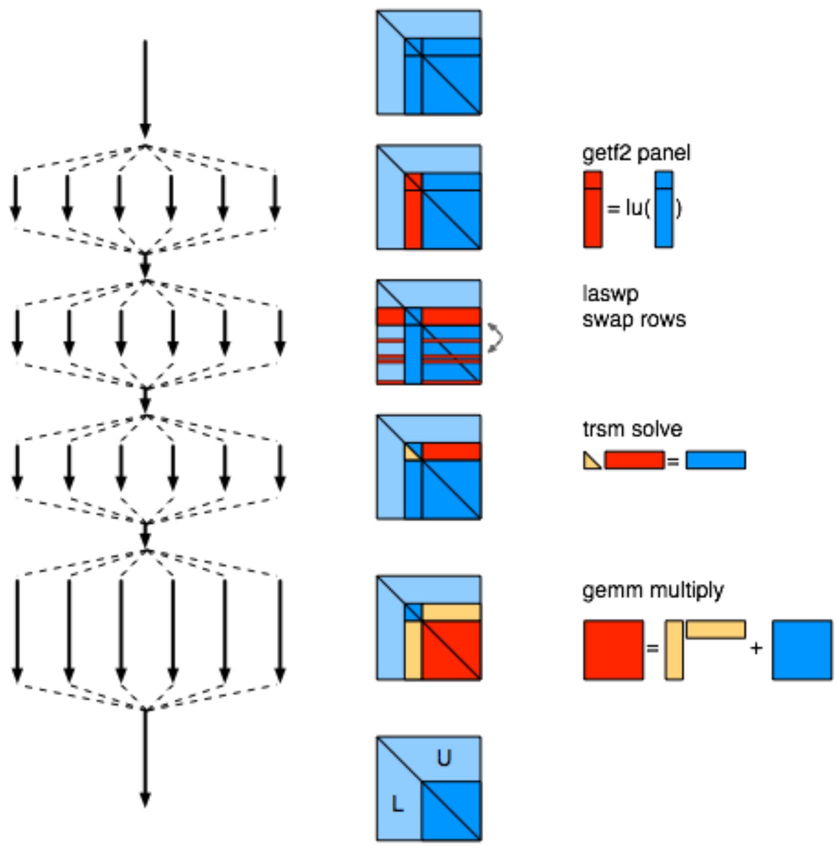
The Standard LU Factorization LAPACK
1980's HPC of the Day: Cache Based SMP



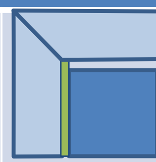
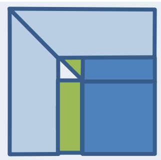
- Factor panel of n_b columns
  - getf2, unblocked BLAS-2 code
- Level 3 BLAS update block-row of U
  - trsm
- Level 3 BLAS update trailing matrix
  - gemm
  - Aimed at machines with cache hierarchy
- Bulk synchronous

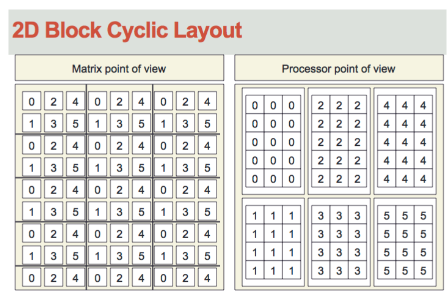
Parallelism in LAPACK

- **Most flops in gemm update**
  - 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



ScaLAPACK

Scalable Linear Algebra PACKage

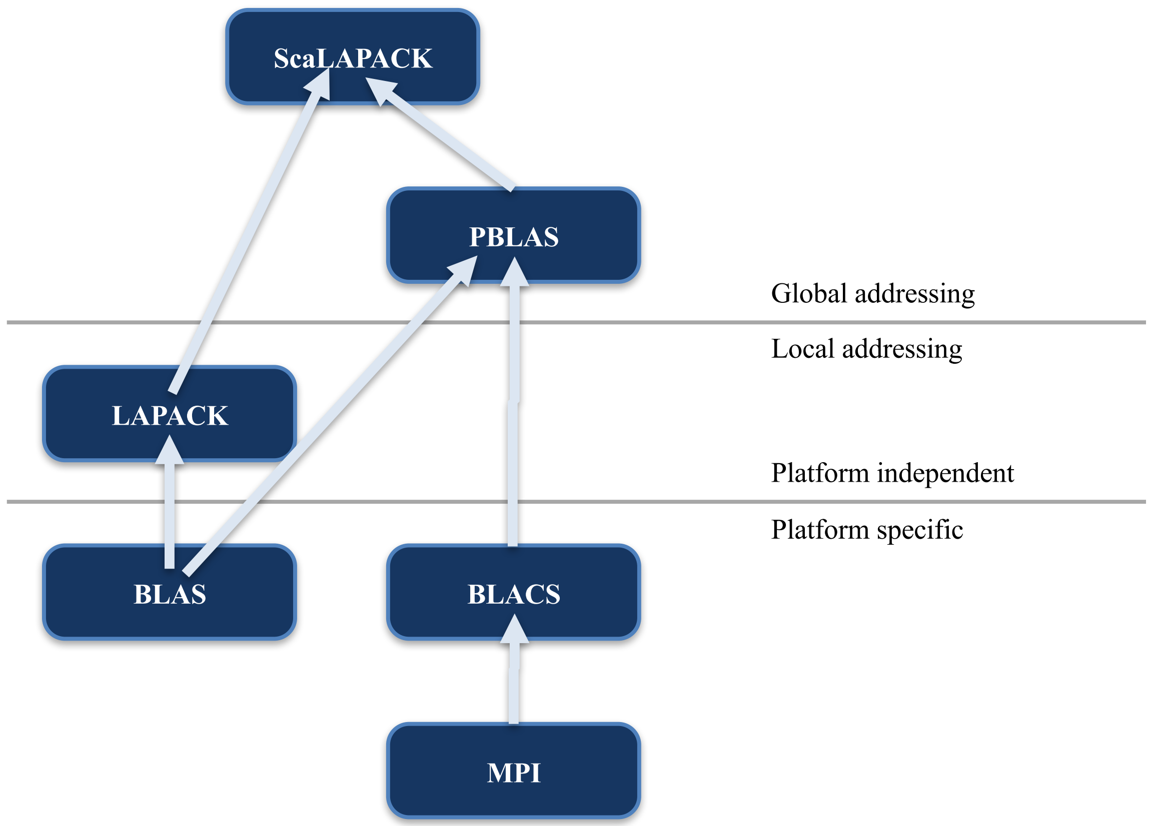
- Distributed memory
- Message Passing
  - Clusters of SMPs
  - Supercomputers
- Dense linear algebra
- Modules
  - PBLAS: Parallel BLAS
  - BLACS: Basic Linear Algebra Communication Subprograms

PBLAS

- Similar to BLAS in functionality and naming
- Built on BLAS and BLACS
- Provide global view of matrix

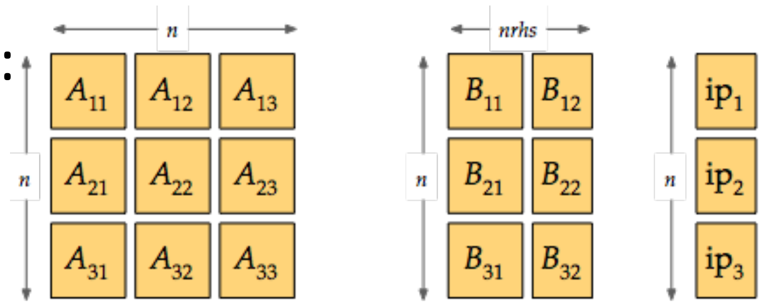
- LAPACK: `dge___( m, n, A(ia, ja), lda, ... )`
  - Submatrix offsets implicit in pointer
- ScaLAPACK: `pdge___( m, n, A, ia, ja, descA, ...`
  - Pass submatrix offsets and matrix descriptor

ScaLAPACK structure



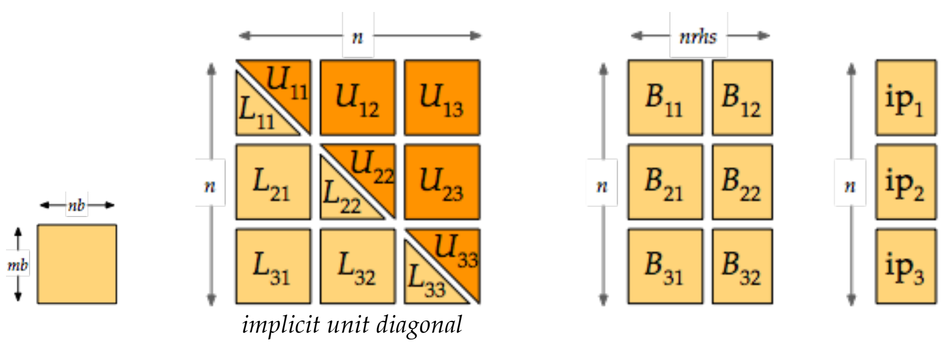
ScaLAPACK routine, solve $AX = B$

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



Global matrix point of view

- output:

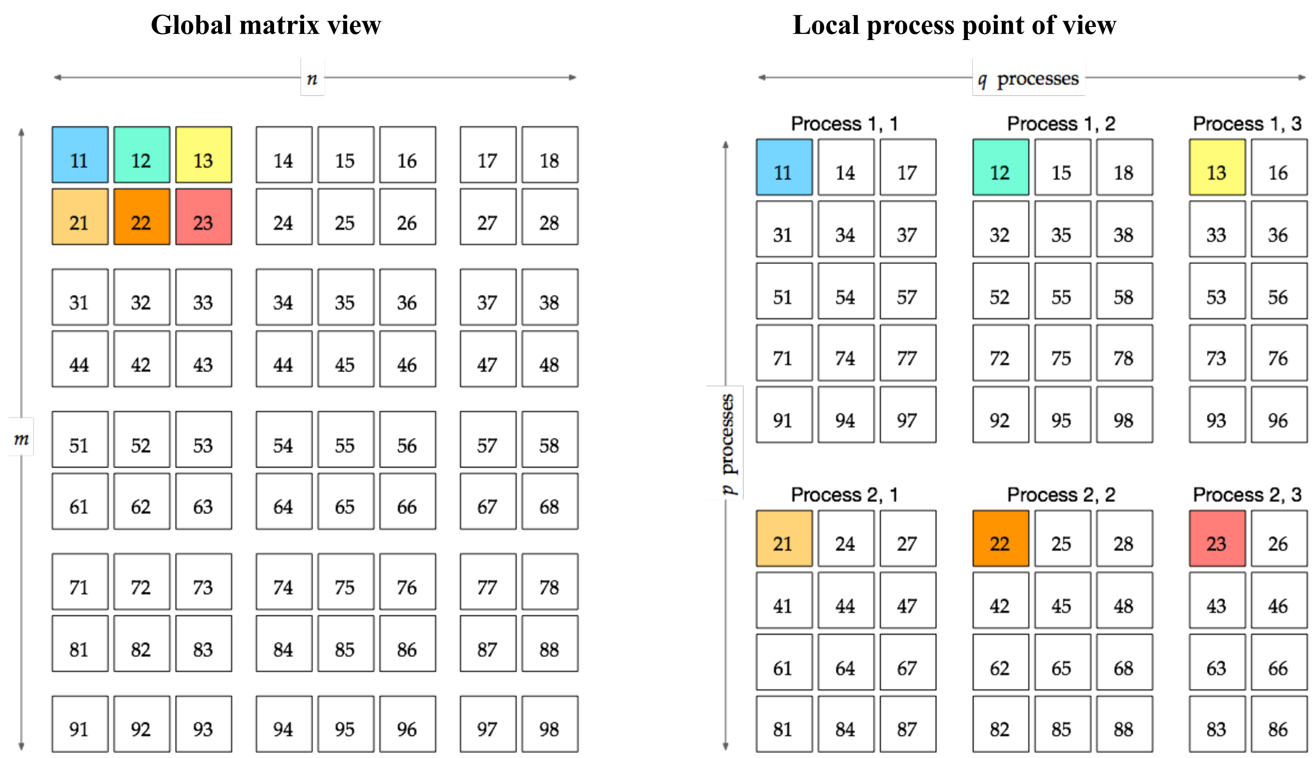


info (error code)
= 0: no error
< 0: invalid argument
> 0: numerical error (e.g., singular)

L, *U* overwrite *A*
X overwrites *B*

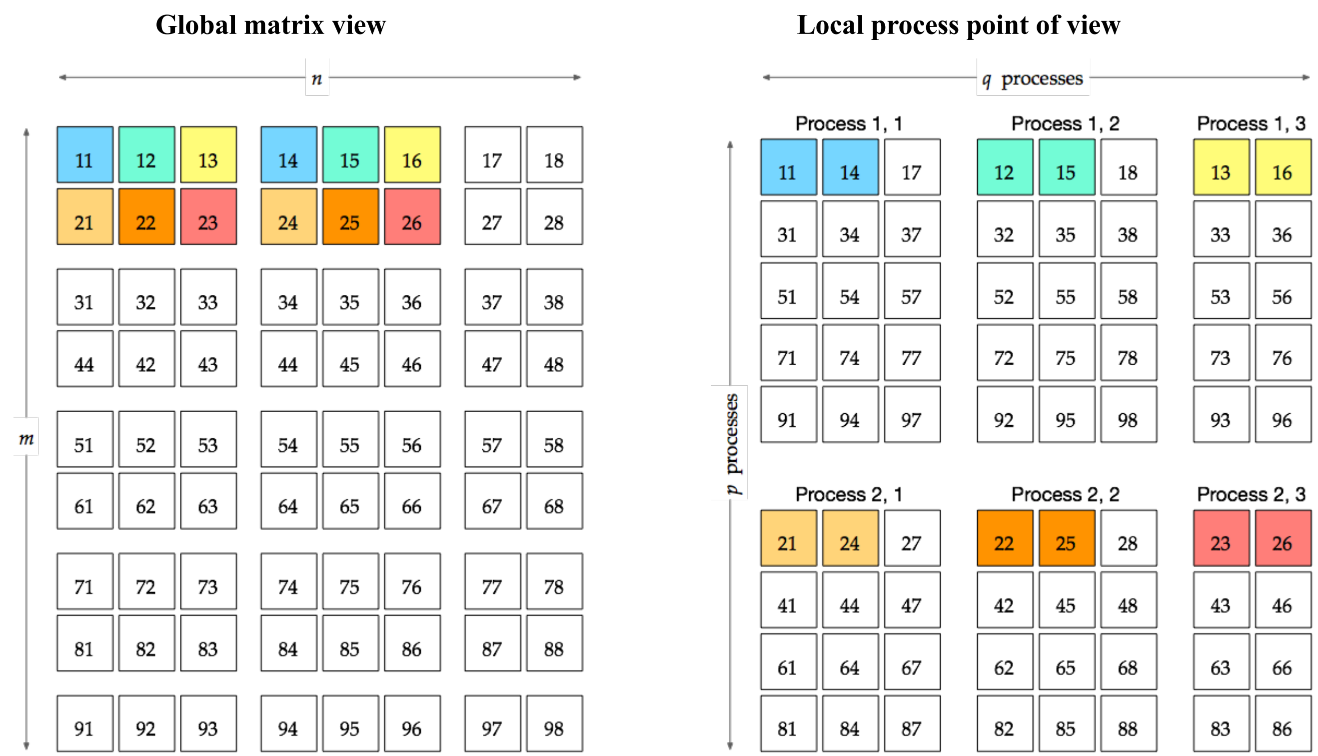
2D block-cyclic layout

$m \times n$ matrix
$p \times q$ process grid



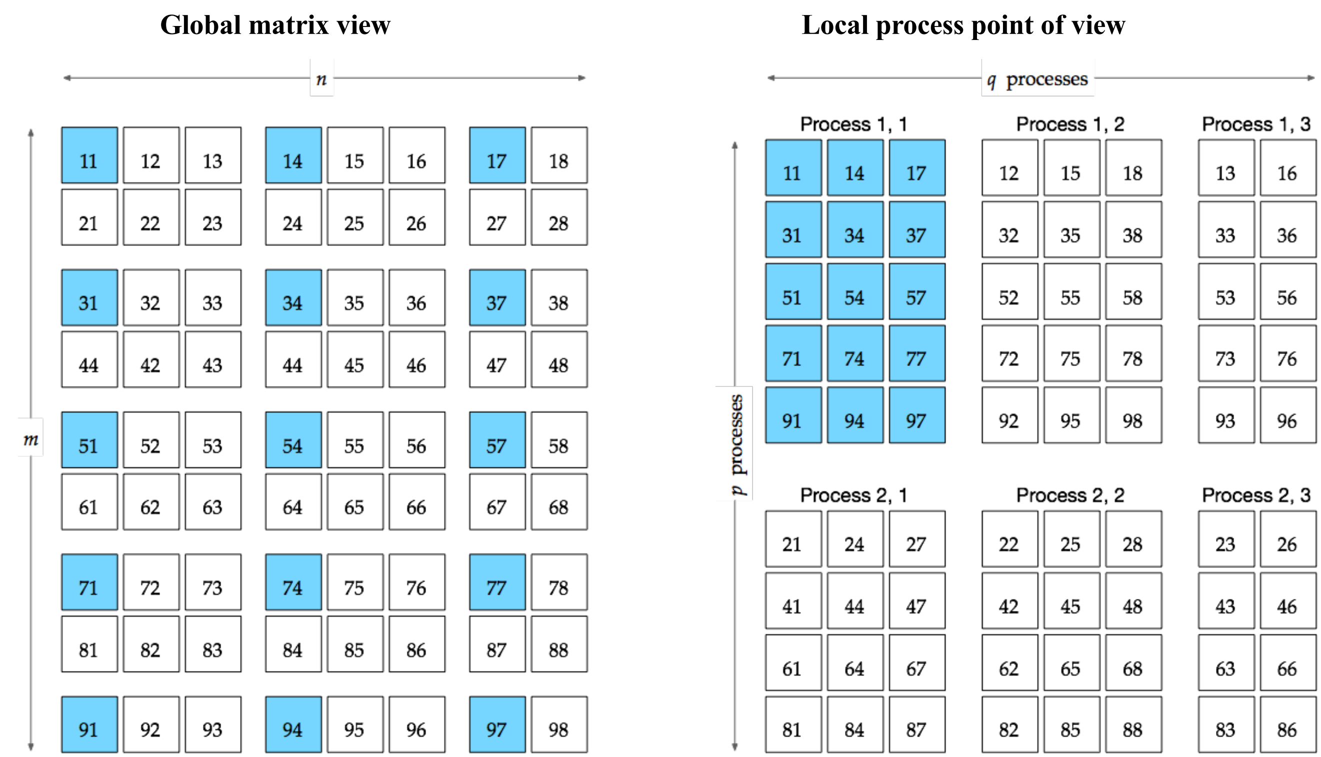
2D block-cyclic layout

$m \times n$ matrix
$p \times q$ process grid



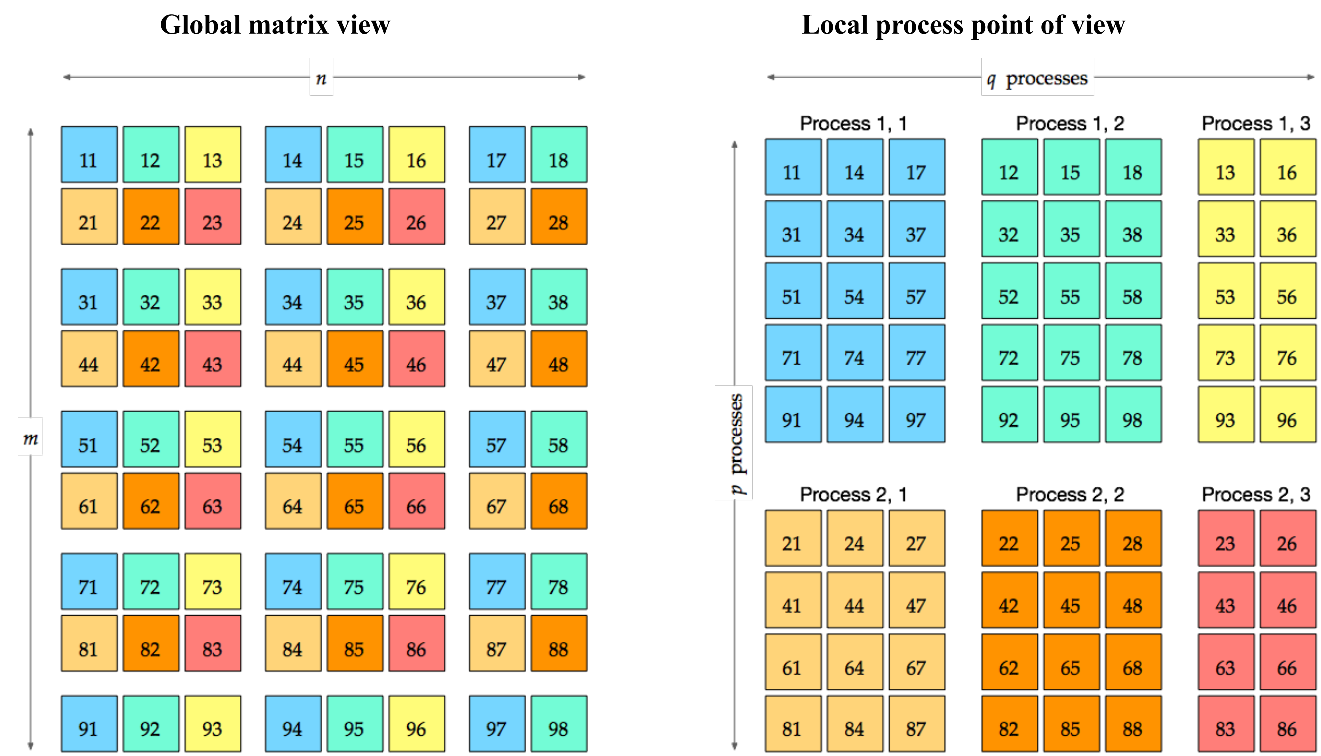
2D block-cyclic layout

$m \times n$ matrix
$p \times q$ process grid



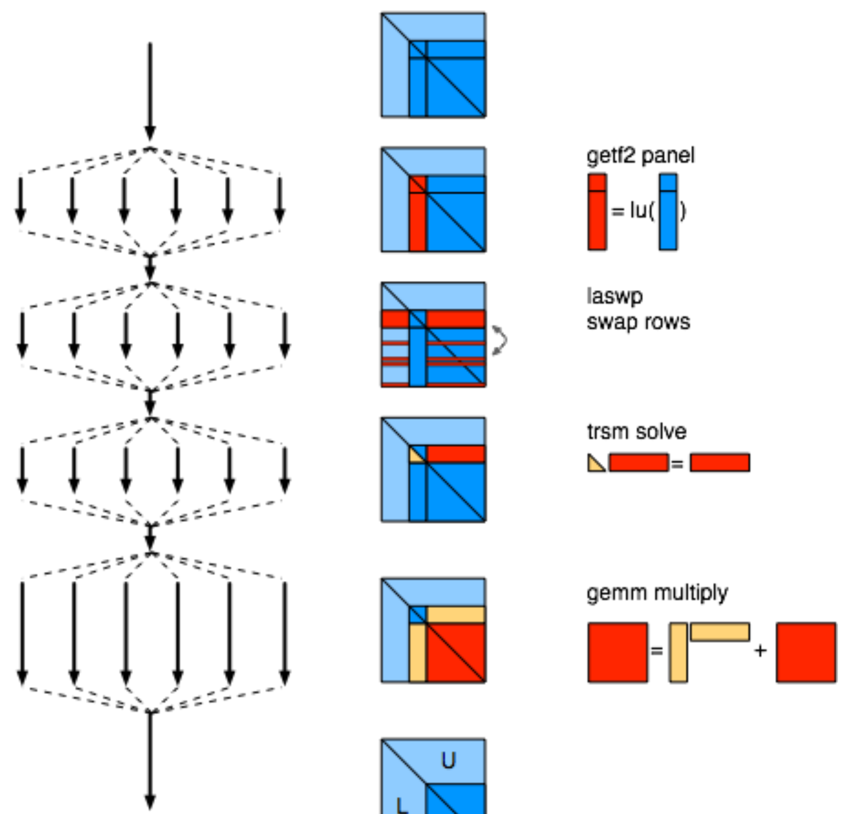
2D block-cyclic layout

$m \times n$ matrix
$p \times q$ process grid

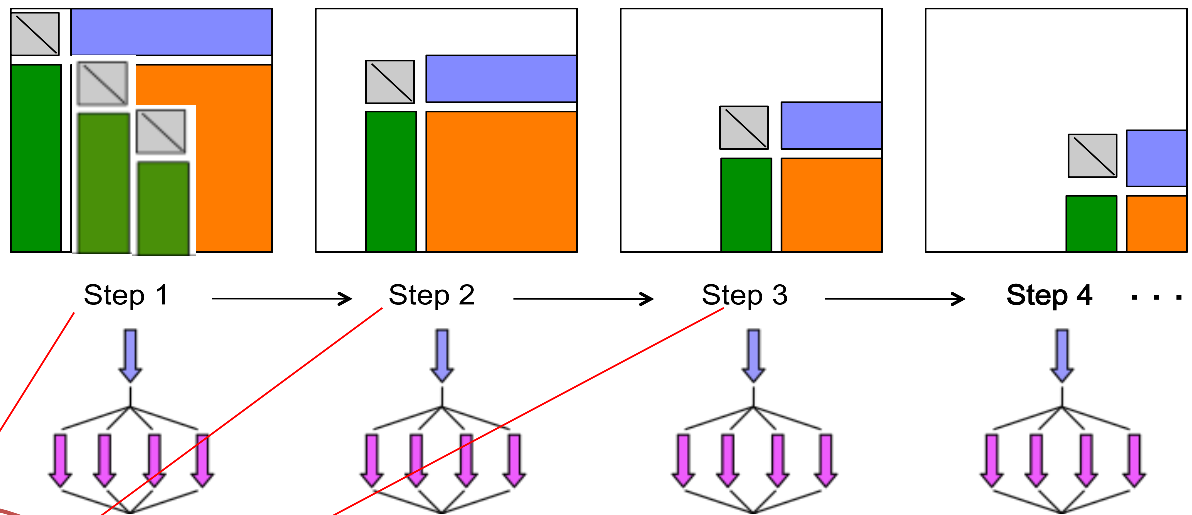


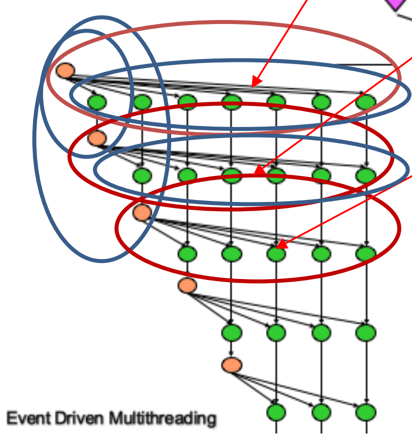
Parallelism in ScaLAPACK

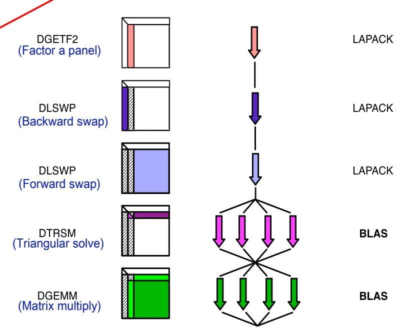
- Similar to LAPACK
- Bulk-synchronous
- Most flops in gemm update
  - 2/3 n^3 term
  - Can use **sequential BLAS**,
    p x q = # cores
    = # MPI processes,
    num_threads = 1
  - Or **multi-threaded BLAS**,
    p x q = # nodes
    = # MPI processes,
    num_threads = # cores/node



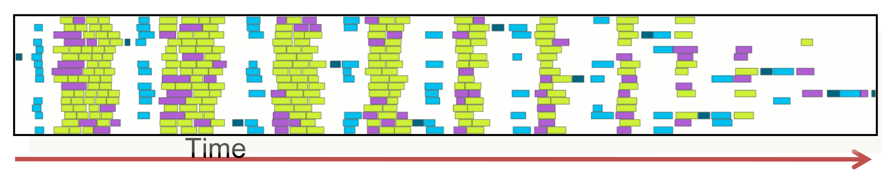
Synchronization (in LAPACK)





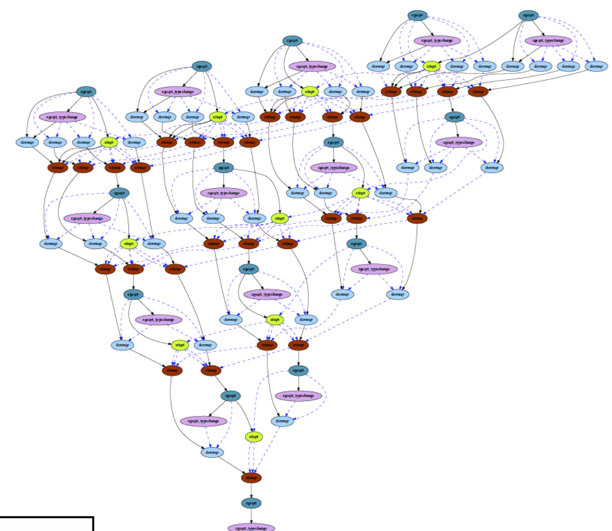


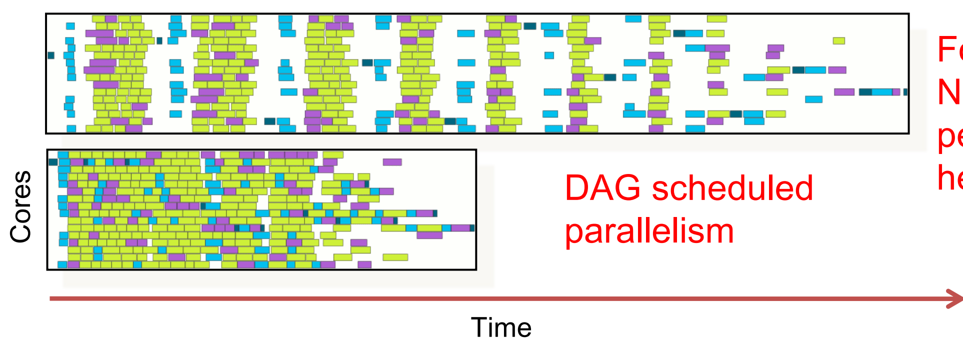
- ➢ fork join
- ➢ bulk synchronous processing



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

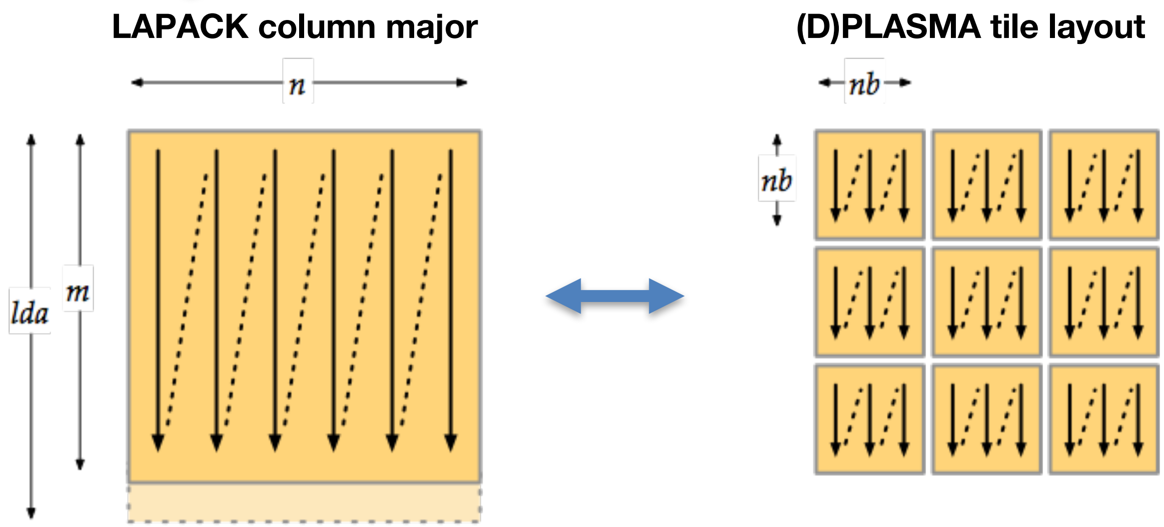




Fork-join parallelism
Notice the synchronization penalty in the presence of heterogeneity.

DAG scheduled parallelism

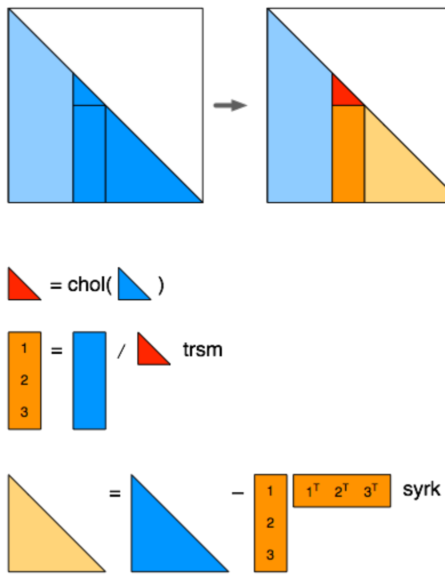
Tile matrix layout



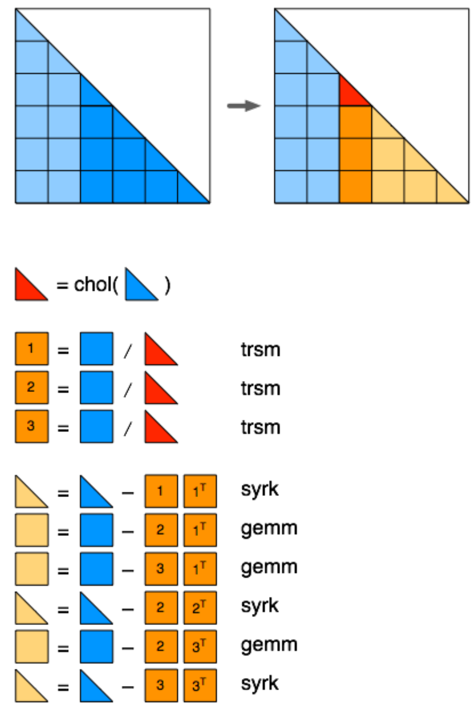
- Tiled layout
  - Each tile is contiguous (column major)
  - Enables dataflow scheduling
  - Cache and TLB efficient (reduces conflict misses and false sharing)
  - MPI messaging efficiency (zero-copy communication)
  - In-place, parallel layout translation

Tile algorithms: Cholesky

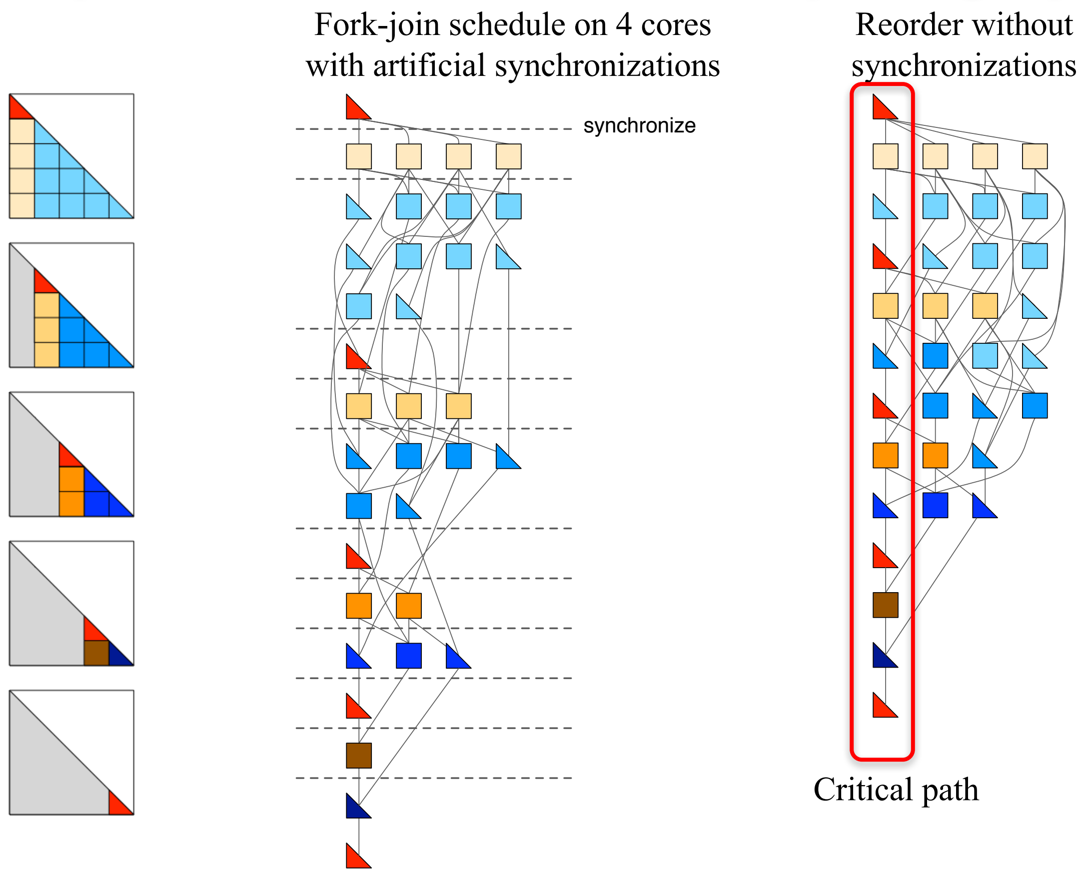
LAPACK Algorithm (right looking)



Tile Algorithm

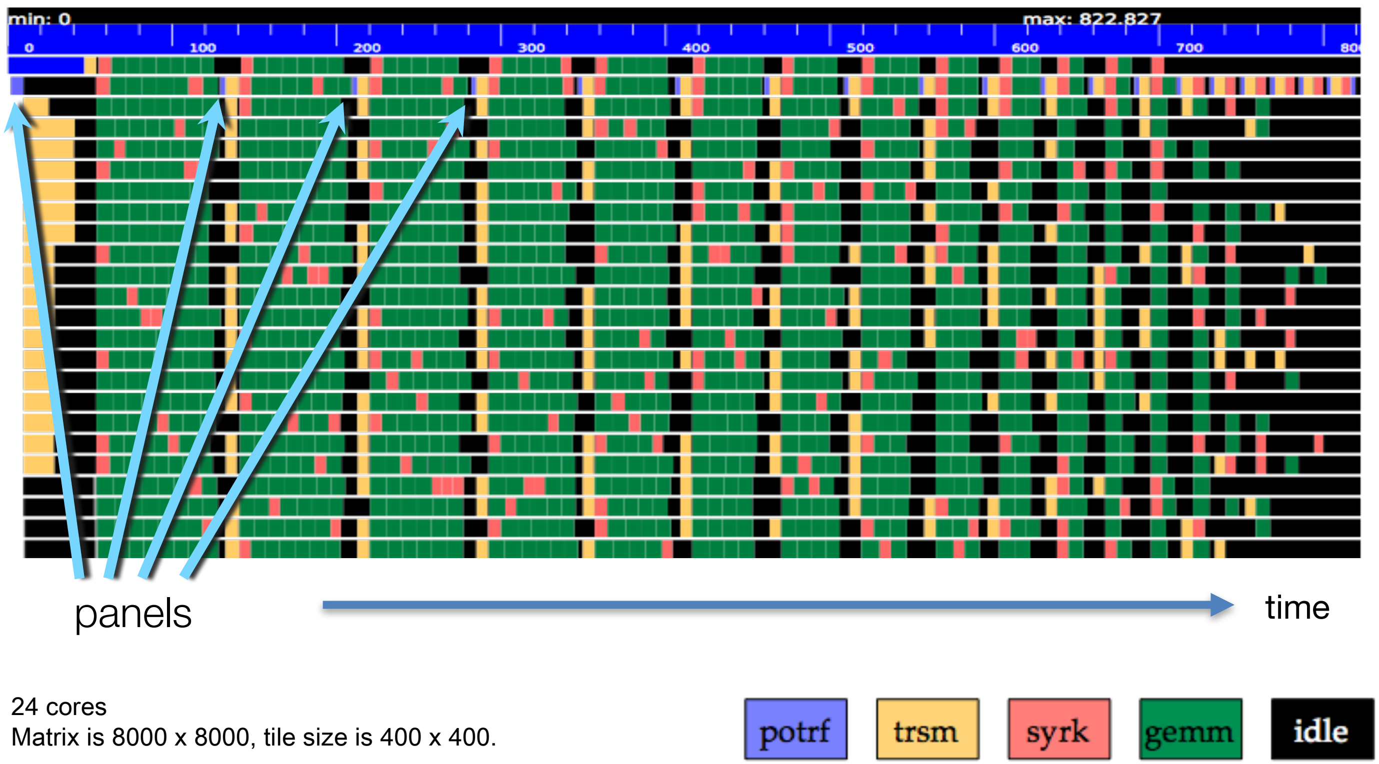


Track dependencies — Directed acyclic graph (DAG)



Execution trace

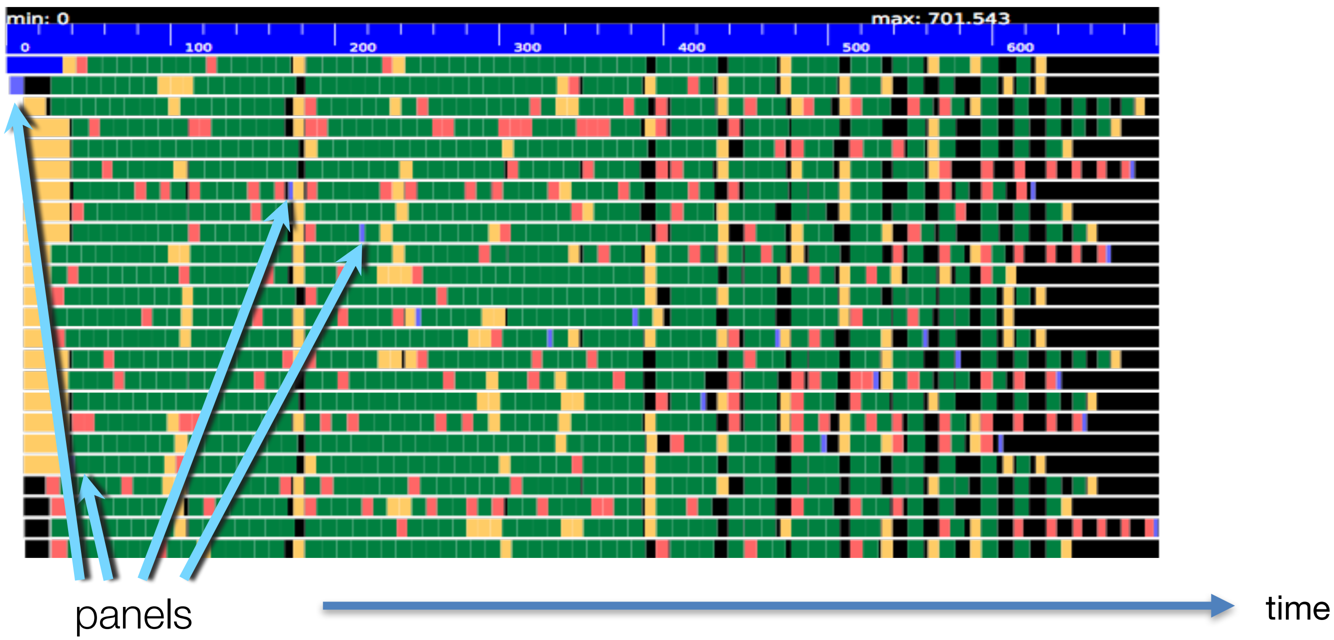
- LAPACK-style fork-join leave cores idle



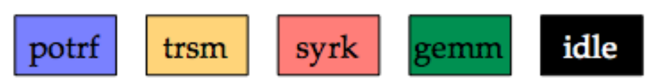
24 cores
Matrix is 8000 x 8000, tile size is 400 x 400.

Execution trace

- PLASMA squeezes out idle time

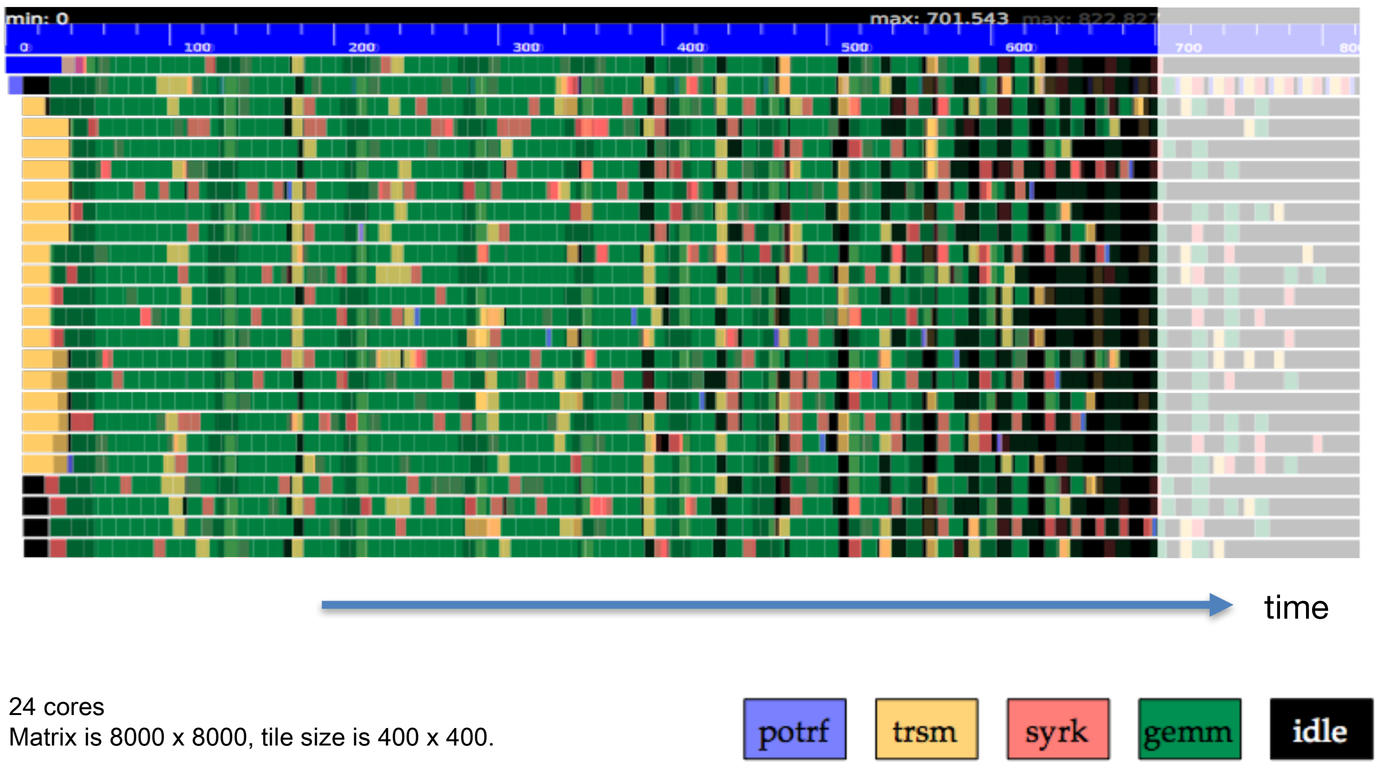


24 cores
Matrix is 8000 x 8000, tile size is 400 x 400.



Execution trace

- PLASMA squeezes out idle time

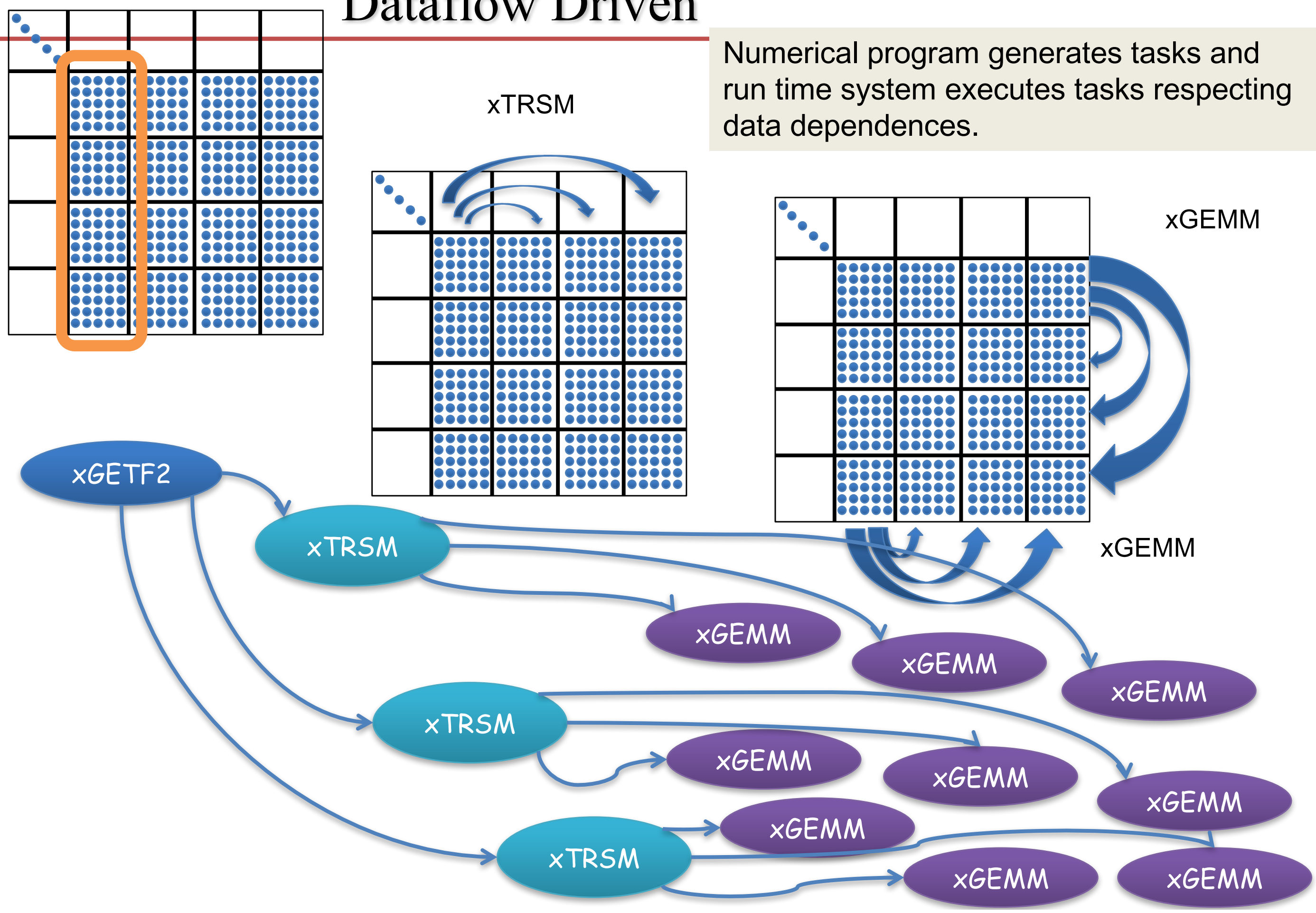


24 cores
Matrix is 8000 x 8000, tile size is 400 x 400.

PLASMA Factorization

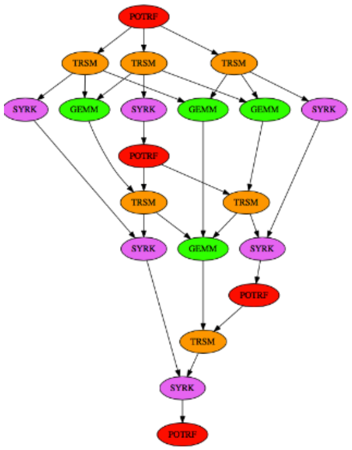
Dataflow Driven

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

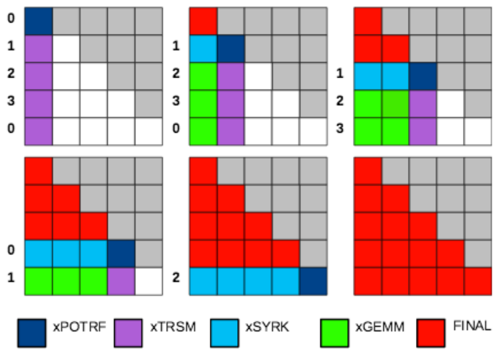


OpenMP tasking

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



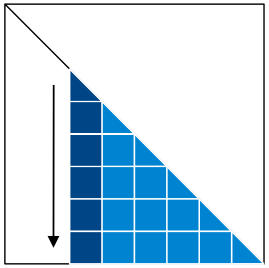
Tiled Cholesky Decomposition



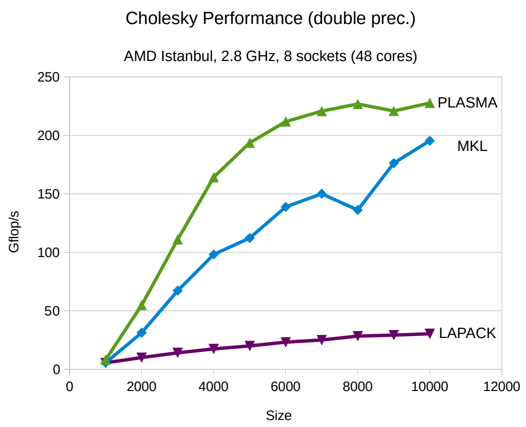

```
#pragma omp parallel
#pragma omp master
{  CHOLESKY( A );   }
CHOLESKY( A ) {
     for (k = 0; k < M; k++) {
          #pragma omp task depend(inout:A(k,k)[0:tilesize]
          {  POTRF( A(k,k) );   }
          for (m = k+1; m < M; m++) {
               #pragma omp task \
                    depend(in:A(k,k)[0:tilesize])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

Algorithms
Cholesky

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

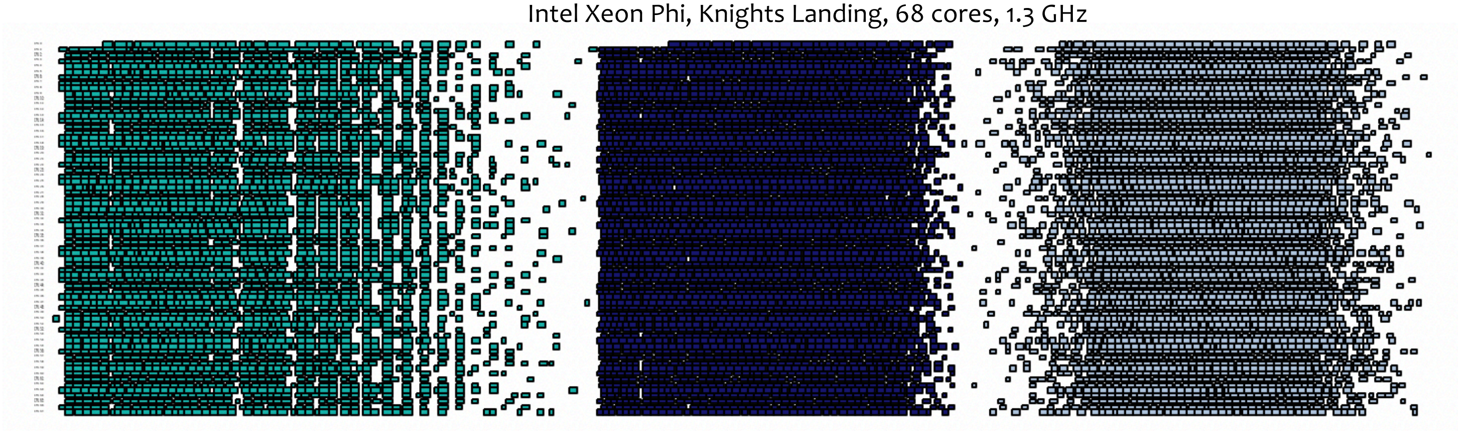


- **Algorithm**
  - equivalent to LAPACK
- **Numerics**
  - same as LAPACK
- **Performance**
  - comparable to vendor on few cores
  - much better than vendor on many cores

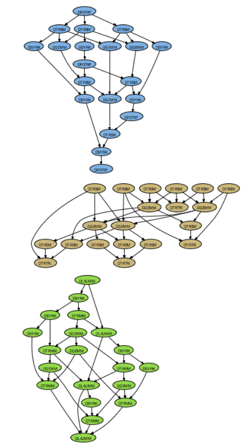


PLASMA – Inverse of the Variance-Covariance Matrix

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

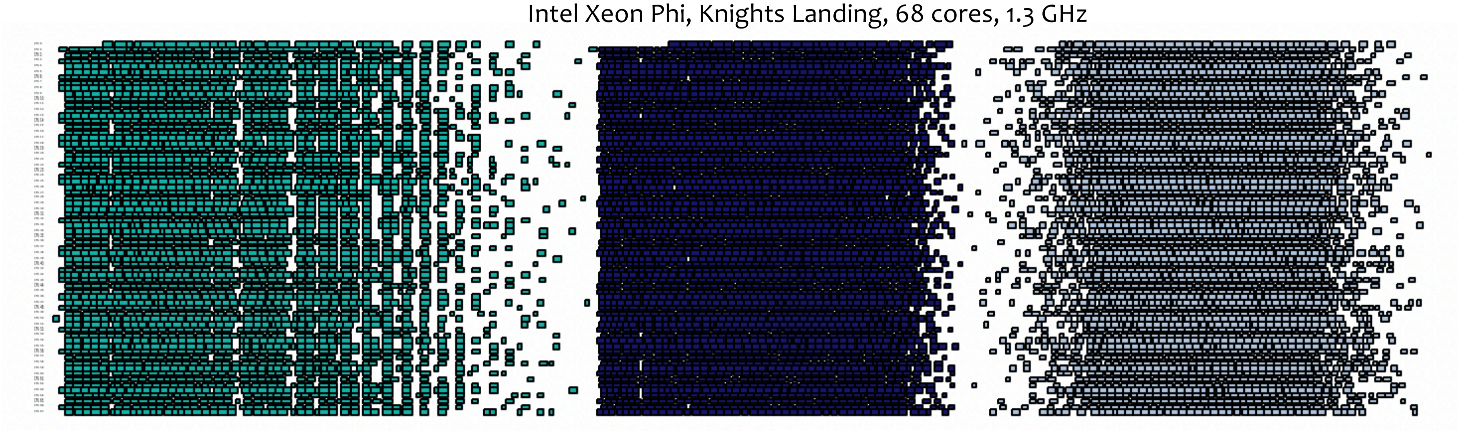


Assume a t by t matrix tiling then Cholesky Factorization alone: 3t-2 Total: 25(7t-3)

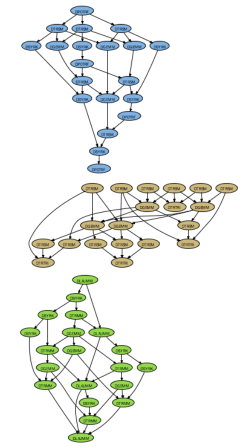
sync:
770 Gflop/s

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



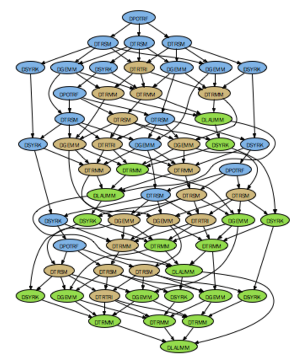
Factor matrix $A = LL^T$ | Compute inverse of factor L | Computer $A^{-1} = L^{-T}L^{-1}$



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

sync:
770 Gflop/s

async:
1001 Gflop/s



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-pf-the-art algorithms

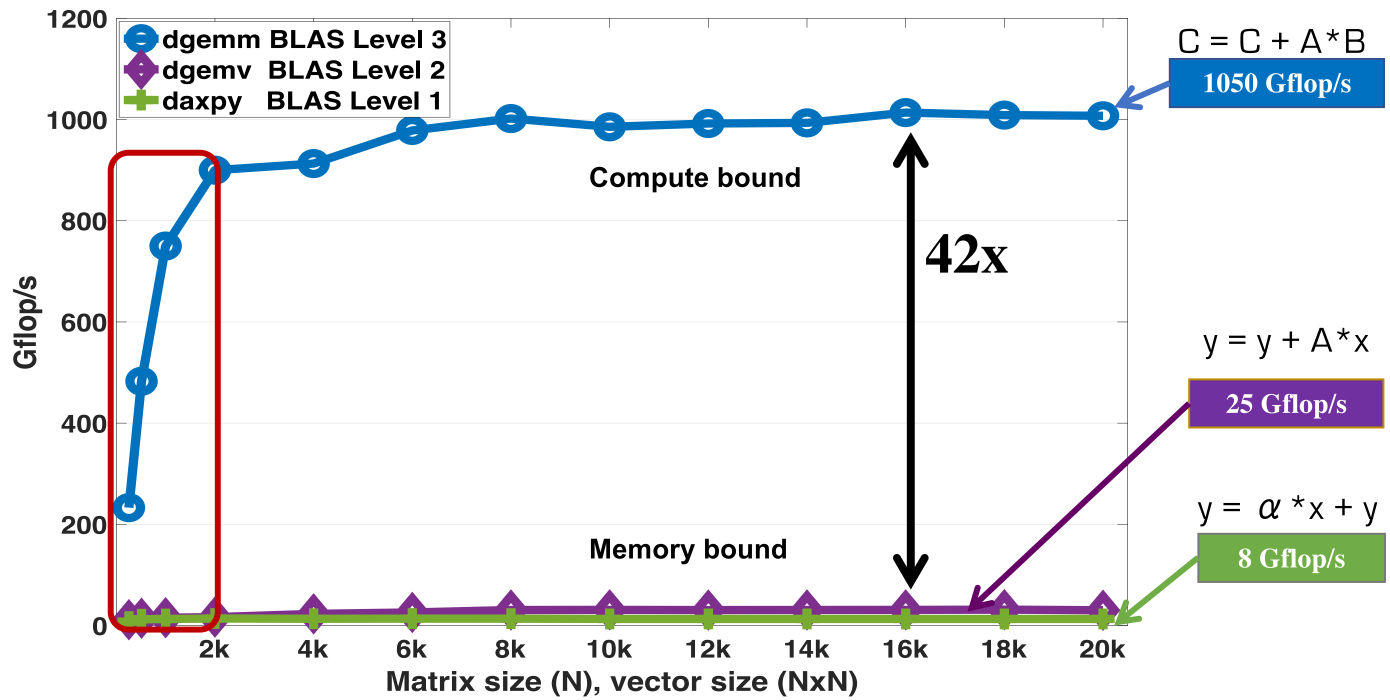
2009 2011 2014 2017

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

18 cores **Intel Xeon Gold 6140 (Skylake)**, 2.3 GHz, Peak DP = 1325 Gflop/s

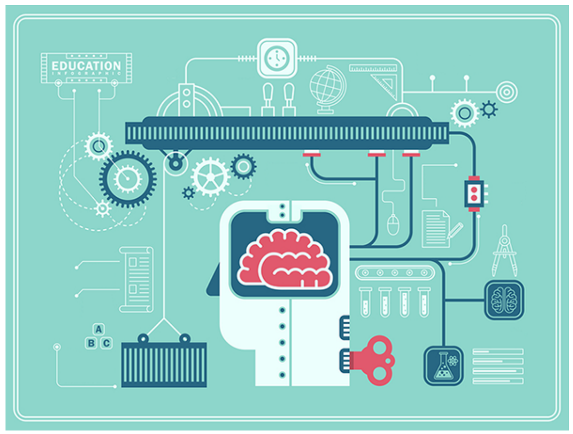


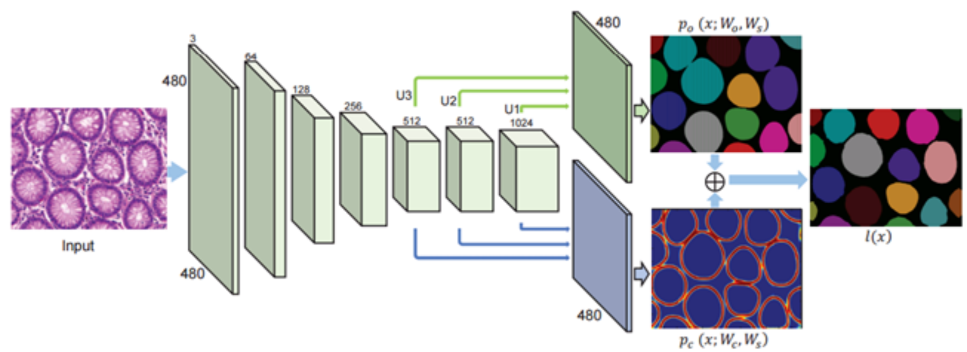
18 cores Intel Xeon Gold 6140, 2.3 GHz (Skylake)
The theoretical peak double precision is 1325 Gflop/s
Compiled with icc and using Intel MKL 2018

Machine Learning in Computational Science

Many fields are beginning to adopt machine learning to augment modeling and simulation methods

- Climate
- Biology
- Drug Design
- Epidemology
- Materials
- Cosmology
- High-Energy Physics



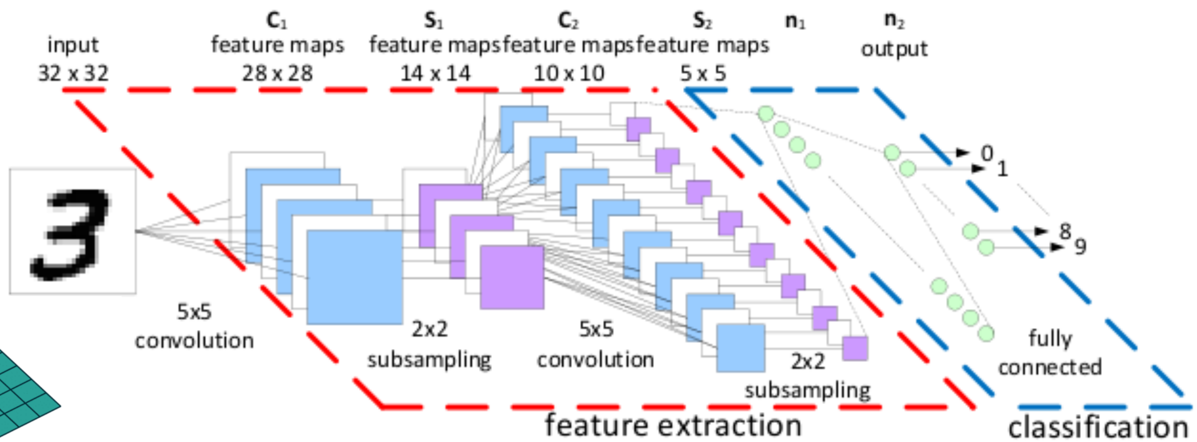


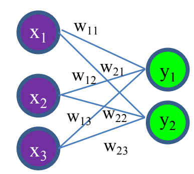
Deep Learning Needs Small Matrix Operations

Matrix Multiply is the time consuming part.

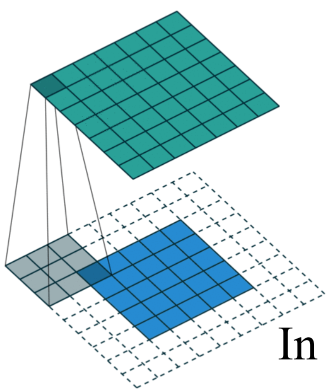
Convolution Layers and Fully Connected Layers require matrix multiply

There are many GEMM's of small matrices, perfectly parallel, can get by with 16-bit floating point





$$\begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$



Convolution Step
In this case 3x3 GEMM

Fully Connected
Classification

Examples

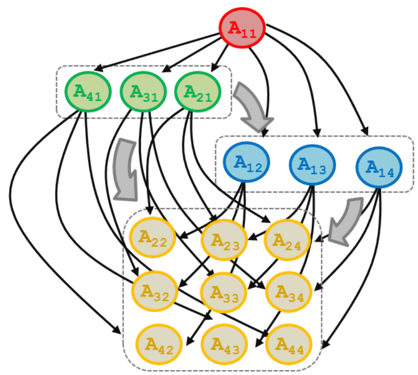
Need of **Batched** routines for **Numerical LA**

[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

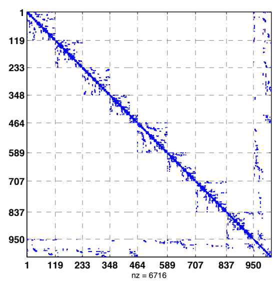
$$\begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix}$$

DAG-based factorization

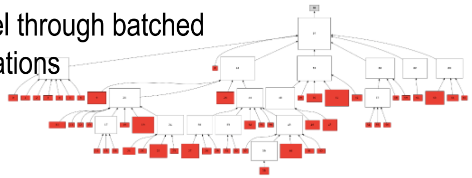


To capture main LA patterns needed in a numerical library for Batched LA

- **LU, QR,** or **Cholesky** on small diagonal matrices
- **TRSM**s, **QR**s, or **LU**s
- **TRSM**s, **TRMM**s
- **Updates (Schur complement) GEMM**s, **SYRK**s, **TRMM**s



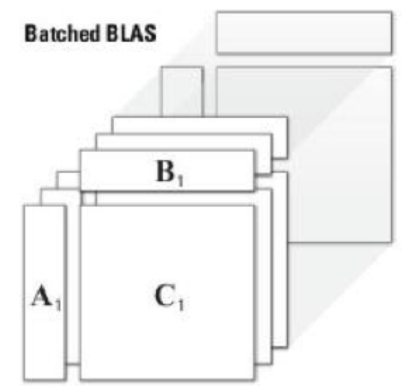
- Example matrix from Quantum chromodynamics
- Reordered and ready for sparse direct multifrontal solver
- Diagonal blocks can be handled in parallel through batched LU, QR, or Cholesky factorizations

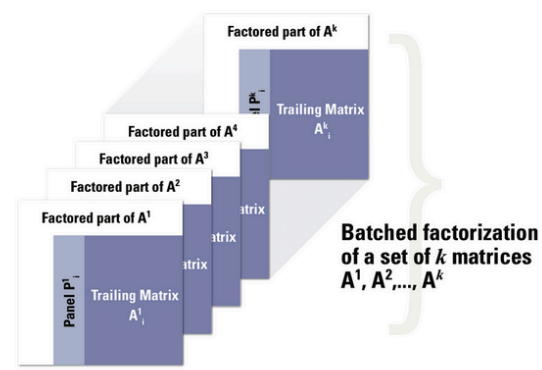


Standard for Batched Computations

- Define standard API for batched BLAS and LAPACK in collaboration with Intel/Nvidia/ECP/other users
- Fixed size most of BLAS and LAPACK released
- Variable size most of BLAS released
- Variable size LAPACK in the branch
- Native GPU algorithms (Cholesky, LU, QR) in the branch
- Tiled algorithm using batched routines on tile or LAPACK data layout in the branch

- Framework for Deep Neural Network kernels
- CPU, KNL and GPU routines
- FP16 routines in progress



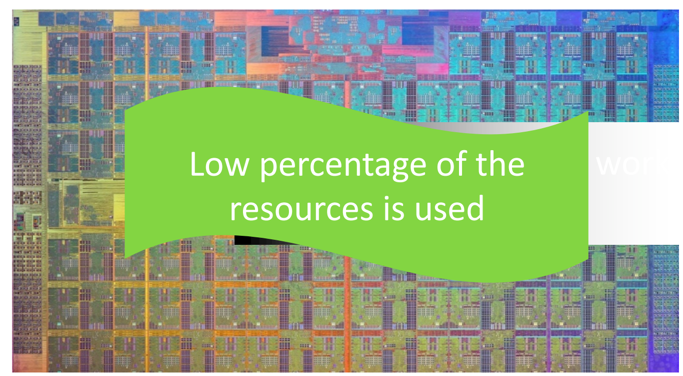


Batched Computations

1. Non-batched computation

- **loop over the matrices one by one** and compute using multithread (note that, since matrices are of small sizes there is not enough work for all the cores). So we expect low performance as well as threads contention might also affect the performance

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



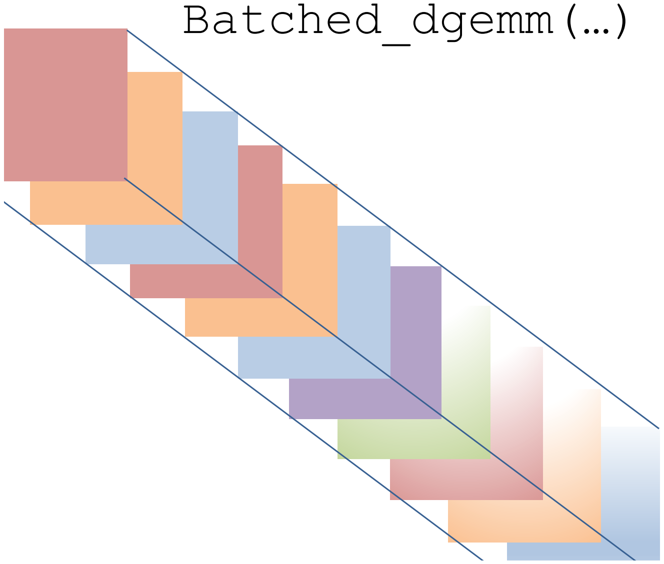
Batched Computations

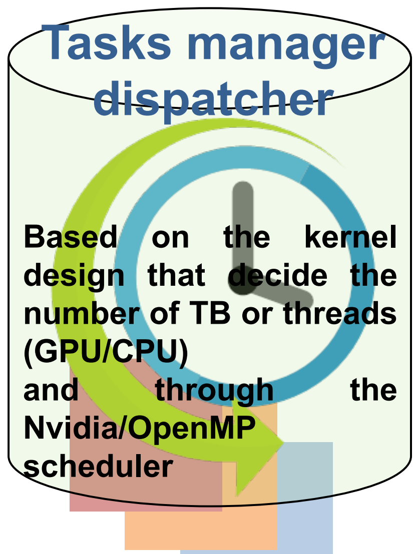
1. Batched computation

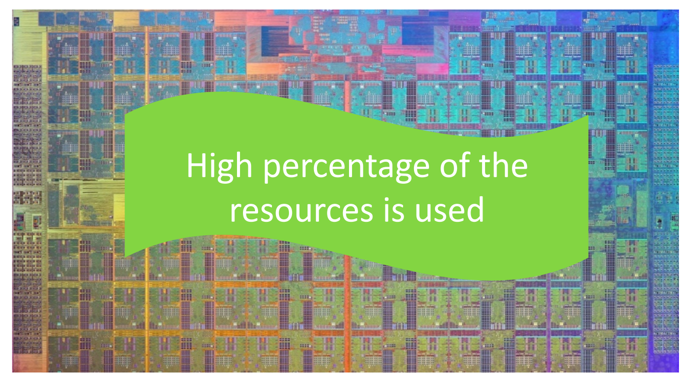
Distribute all the matrices over the available resources by assigning a matrix to each group of core/TB to operate on it independently

- For very small matrices, assign a matrix/core (CPU) or per TB for GPU
- For medium size a matrix go to a team of cores (CPU) or many TB's (GPU)
- For large size switch to multithreads classical 1 matrix per round.

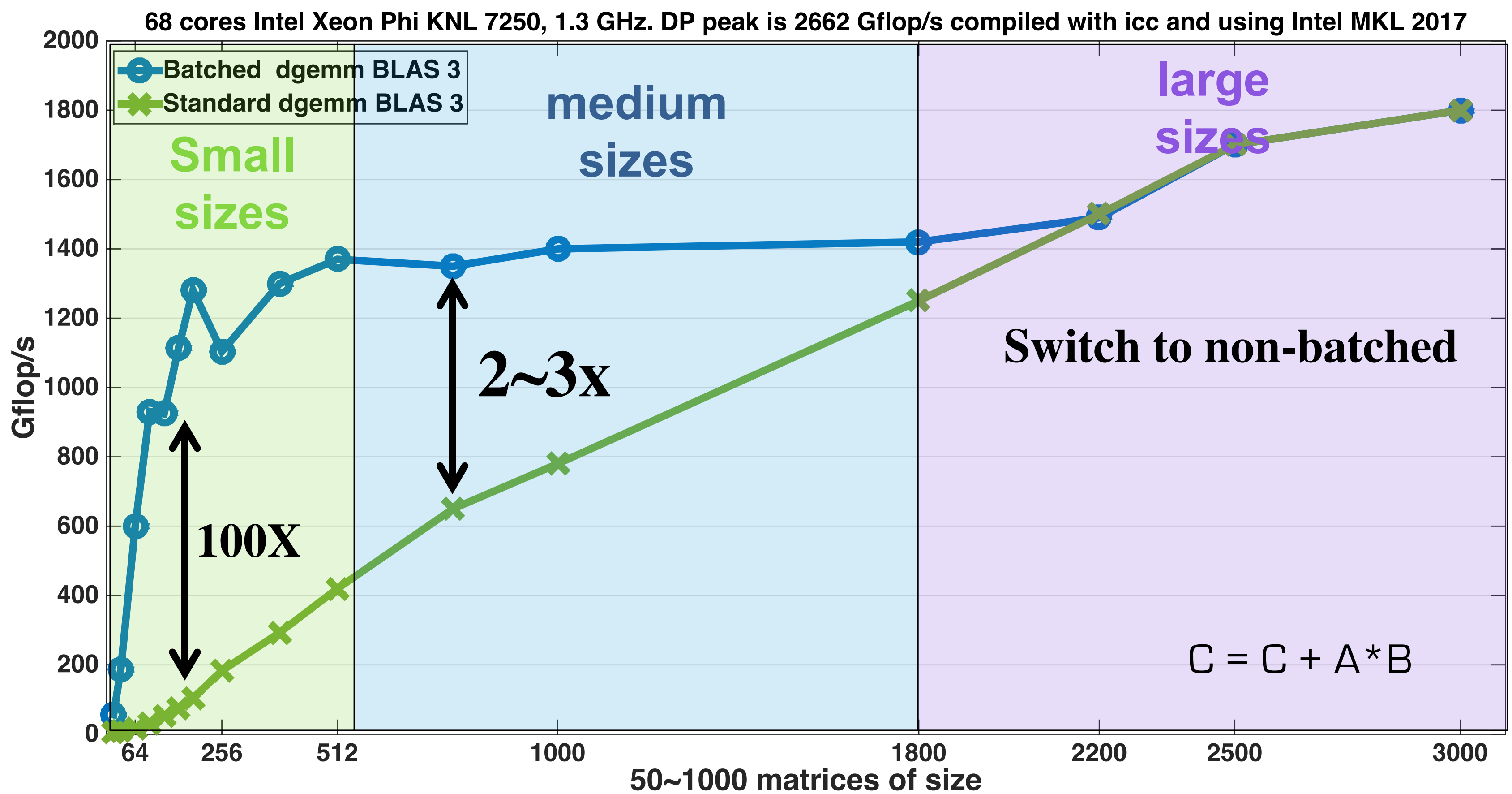
`Batched_dgemm(…)`



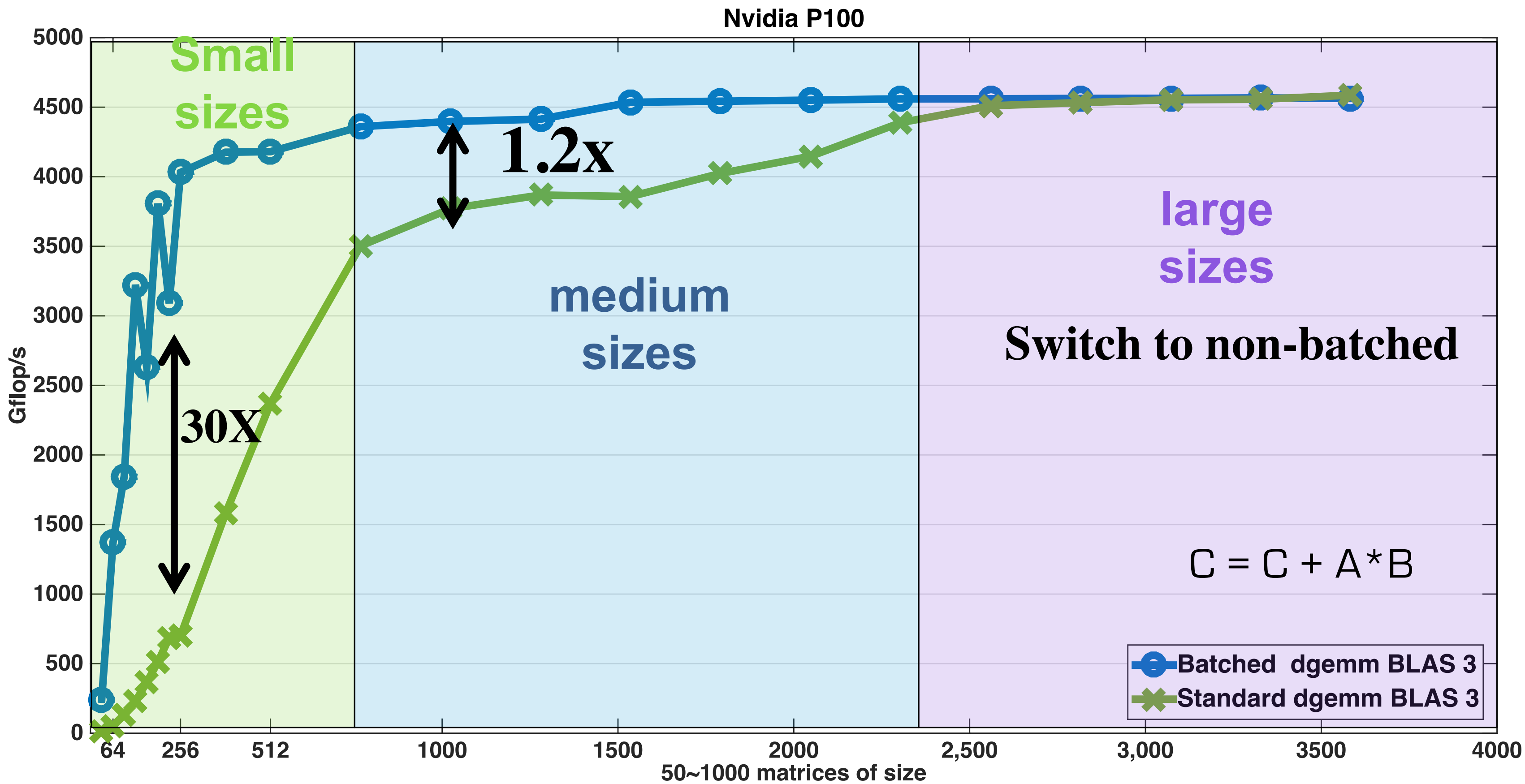




Batched Computations: How do we design and optimize

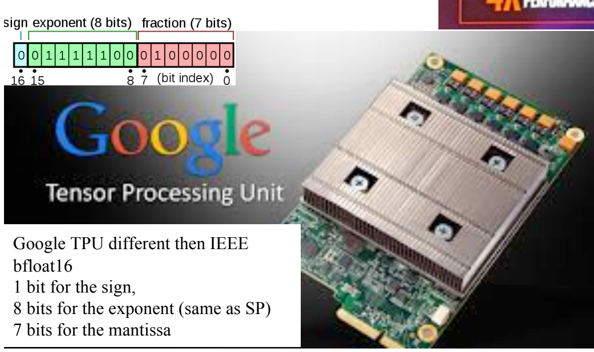


Batched Computations: How do we design and optimize



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

A lot of interest driven by "machine learning"



Google TPU different then IEEE
bfloat16
1 bit for the sign,
8 bits for the exponent (same as SP)
7 bits for the mantissa

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	?

GPU PERFORMANCE COMPARISON

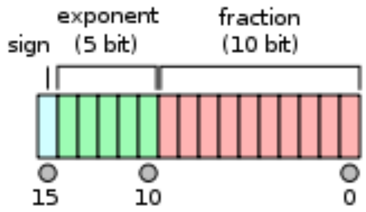
	P100	V100	Ratio
DL Training FP16	10 TFLOPS	120 TFLOPS	12x
DL Inferencing FP16	21 TFLOPS	120 TFLOPS	6x
FP64/FP32	5/10 TFLOPS	7.5/15 TFLOPS	1.5x
HBM2 Bandwidth	720 GB/s	900 GB/s	1.2x
STREAM Triad Perf	557 GB/s	855 GB/s	1.5x
NVLink Bandwidth	160 GB/s	300 GB/s	1.9x
L2 Cache	4 MB	6 MB	1.5x
L1 Caches	1.3 MB	10 MB	7.7x

Mixed Precision

- **Today many precisions to deal with**

Type	Size	Range	$u = 2^{-t}$
half	16 bits	$10^{\pm 5}$	$2^{-11} \approx 4.9 \times 10^{-4}$
single	32 bits	$10^{\pm 38}$	$2^{-24} \approx 6.0 \times 10^{-8}$
double	64 bits	$10^{\pm 308}$	$2^{-53} \approx 1.1 \times 10^{-16}$
quadruple	128 bits	$10^{\pm 4932}$	$2^{-113} \approx 9.6 \times 10^{-35}$

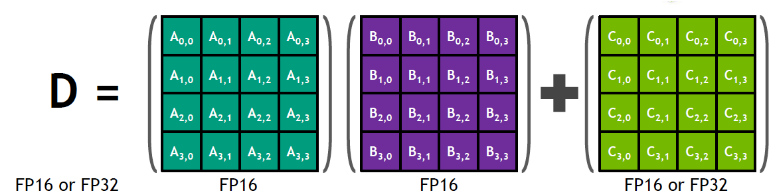
- **Note the number range with half precision (16 bit fl.pt.)**



Nvidia Volta peak rates

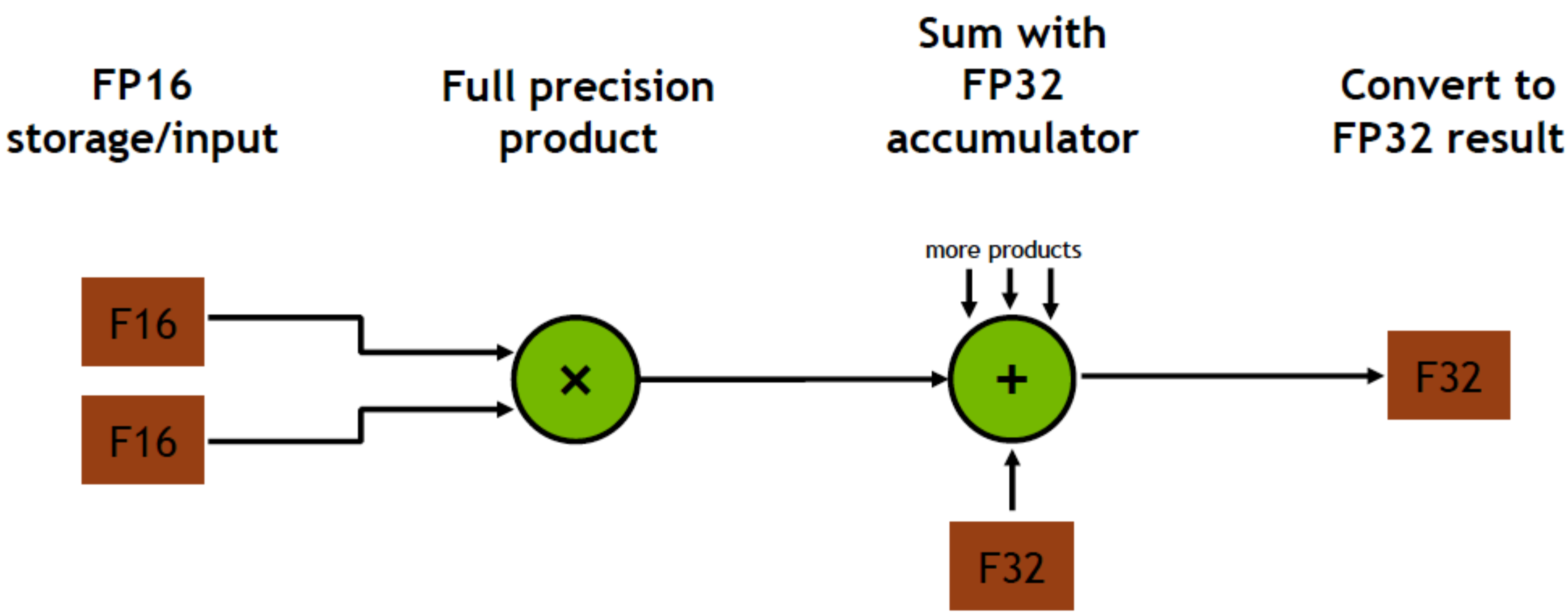
- 64 bit floating point (FMA): 7.5 Tflop/s
- 32 bit floating point (FMA): 15 Tflop/s
- 16 bit floating point (FMA): 30 Tflop/s
- 16 bit floating point with Tensor core: 120 Tflop/s

Mixed Precision Matrix Multiply
4x4 Matrices



D = AB + C

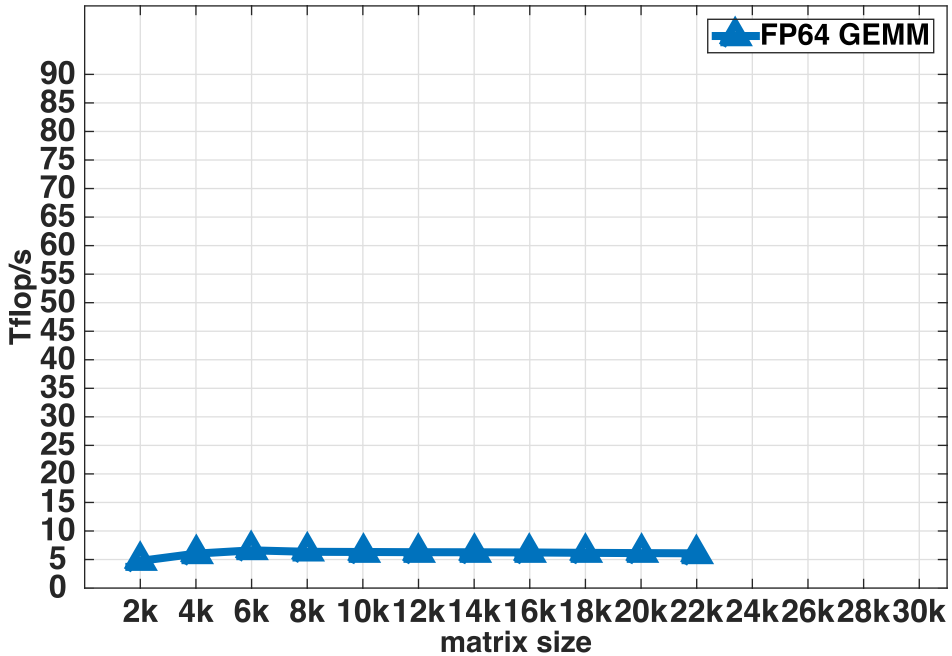
VOLTA TENSOR OPERATION



Also supports FP16 accumulator mode for inferencing

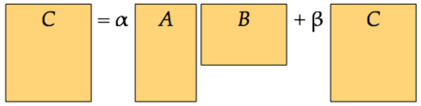
Leveraging Half Precision in HPC on V100

Study of the Matrix Matrix multiplication kernel on Nvidia V100



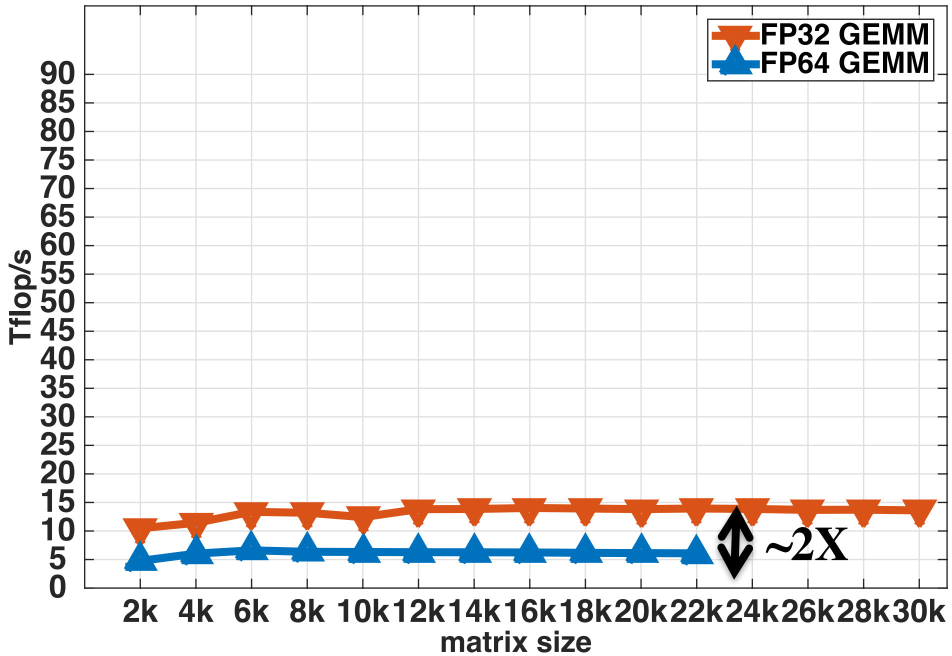
- dgemm achieve about 6.4 Tflop/s

Matrix matrix multiplication GEMM



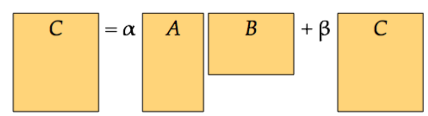
Leveraging Half Precision in HPC on V100

Study of the Matrix Matrix multiplication kernel on Nvidia V100



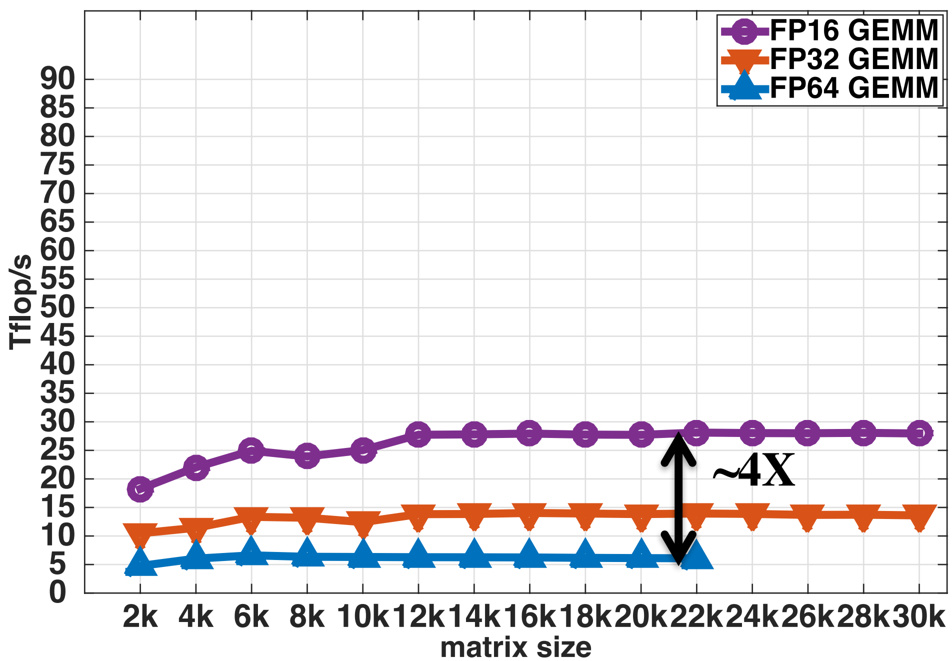
- dgemm achieve about 6.4 Tflop/s
- sgemm achieve about 14 Tflop/s

Matrix matrix multiplication GEMM



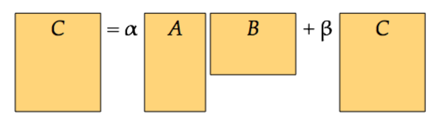
Leveraging Half Precision in HPC on V100

Study of the Matrix Matrix multiplication kernel on Nvidia V100



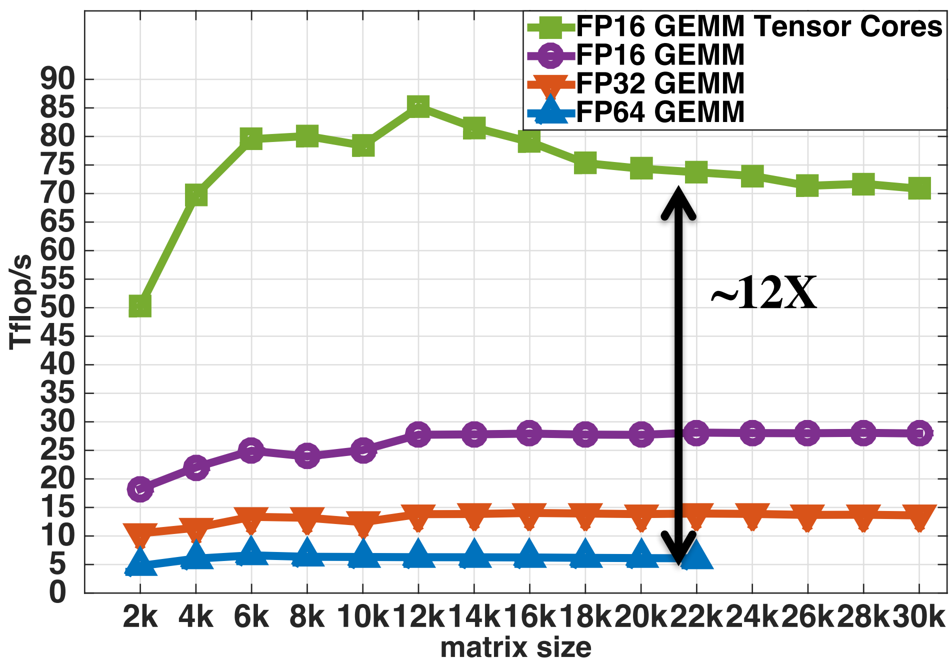
dgemm achieve about 6.4 Tflop/s
sgemm achieve about 14 Tflop/s
hgemm achieve about 27 Tflop/s

Matrix matrix multiplication GEMM



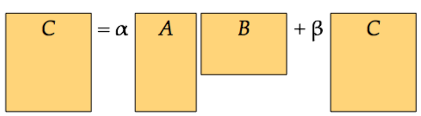
Leveraging Half Precision in HPC on V100

Study of the Matrix Matrix multiplication kernel on Nvidia V100



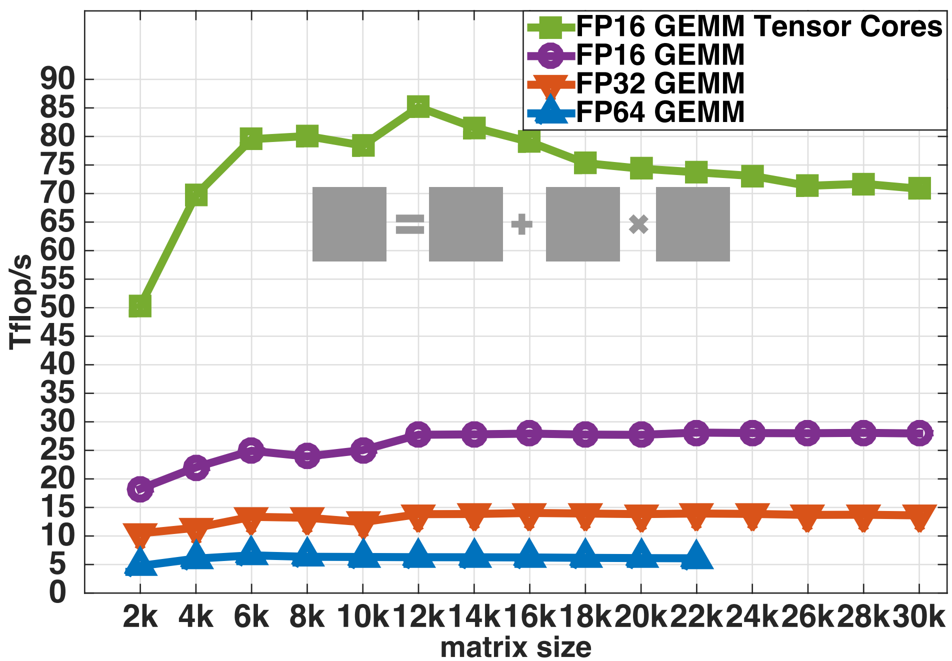
dgemm achieve about 6.4 Tflop/s
sgemm achieve about 14 Tflop/s
hgemm achieve about 27 Tflop/s
Tensor cores gemm reach about 85 Tflop/s

Matrix matrix multiplication GEMM



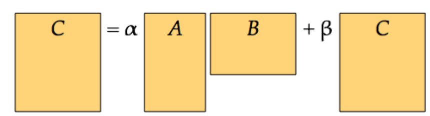
Leveraging Half Precision in HPC on V100

Study of the Matrix Matrix multiplication kernel on Nvidia V100



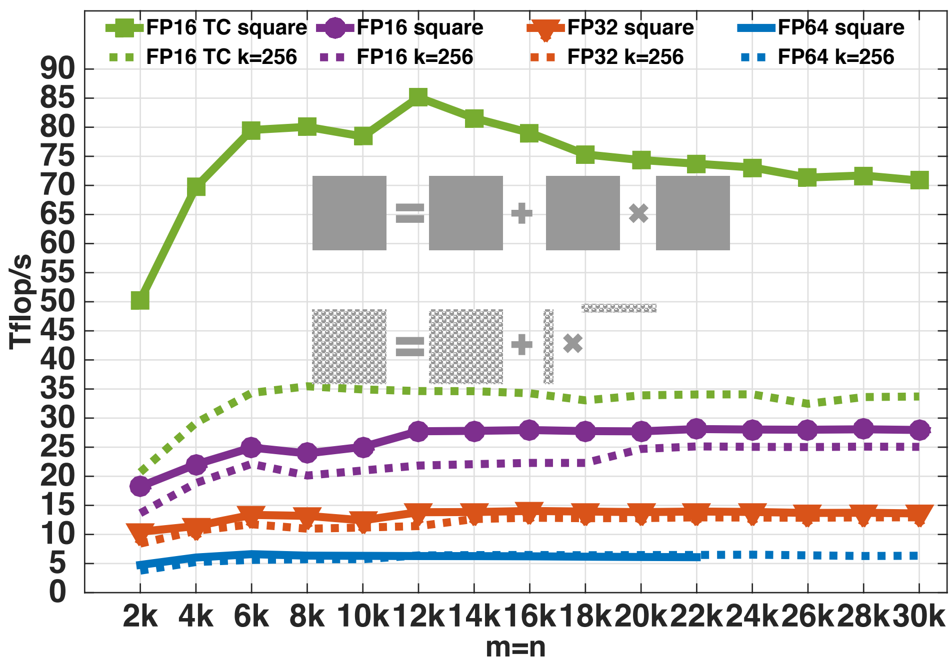
dgemm achieve about 6.4 Tflop/s
sgemm achieve about 14 Tflop/s
hgemm achieve about 27 Tflop/s
Tensor cores gemm reach about 85 Tflop/s

Matrix matrix multiplication GEMM



Leveraging Half Precision in HPC on V100

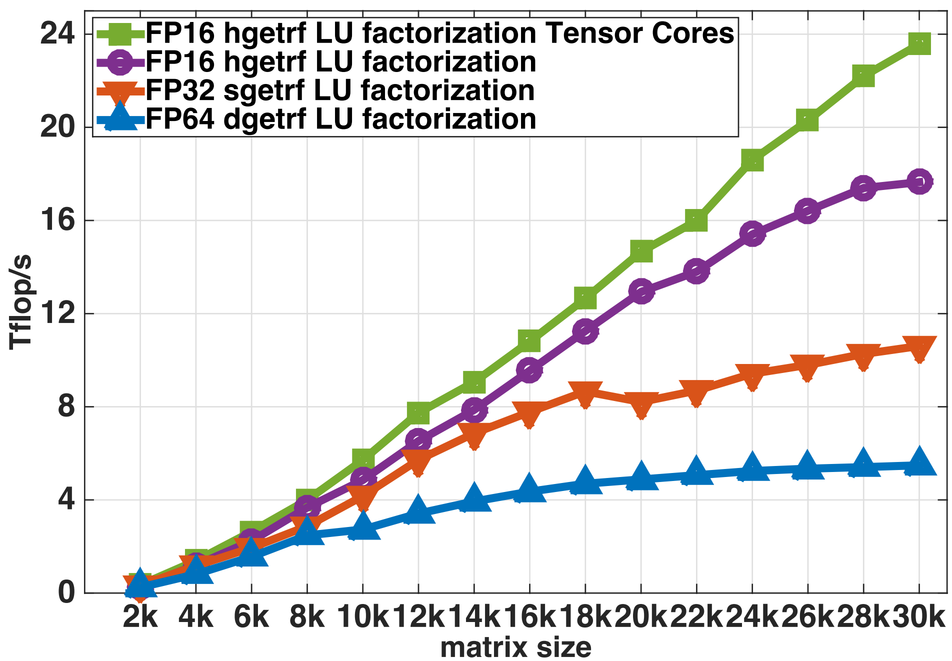
Study of the rank k update used by the LU factorization algorithm on Nvidia V100



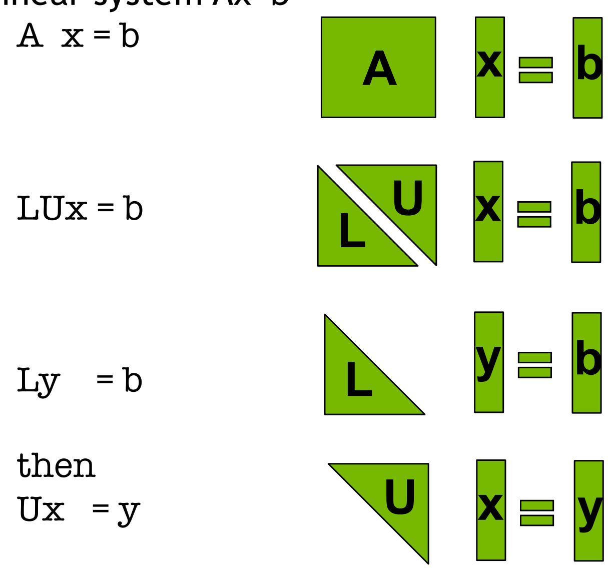
- In LU factorization need matrix multiple but operations is a rank-k update computing the Schur complement

Leveraging Half Precision in HPC on V100

Study of the LU factorization algorithm on Nvidia V100



- LU factorization is used to solve a linear system Ax=b



Leveraging half precision for HPC

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.

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

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

Leveraging Half Precision in HPC on V100

Use Mixed Precision algorithms

- ➢Achieve higher performance → faster time to solution
- ➢Reduce power consumption reduce power consumption by decreasing the execution time → **Energy Savings !!!**

Reference:

A. Haidar, P. Wu, S. Tomov, J. Dongarra,
Investigating Half Precision Arithmetic to Accelerate Dense Linear System Solvers,
SC-17, ScalA17: 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, ACM, Denver, Colorado, November 12-17, 2017.

Leveraging Half Precision in HPC on V100

Idea: use low precision to compute the expensive flops (LU $O(n^3)$) and then iteratively refine the solution in order to achieve the FP64 arithmetic

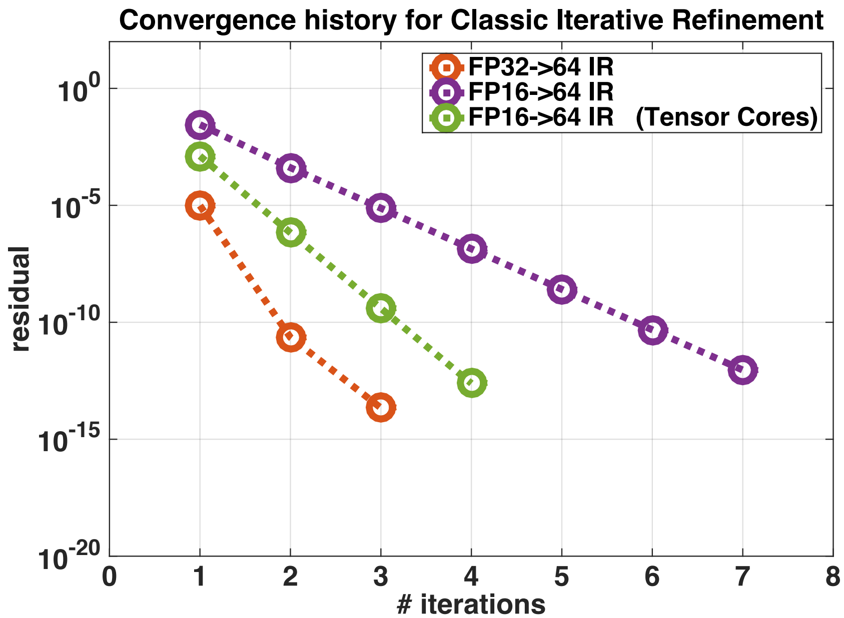
Iterative refinement for dense systems, $Ax = b$, can work this way.

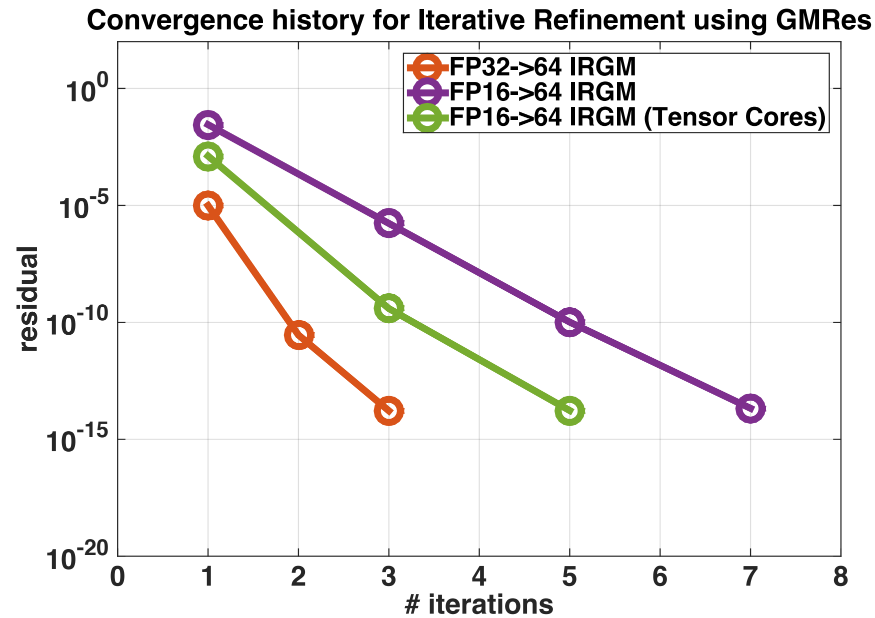
L U = lu(A)		lower precision	$O(n^3)$
x = U\(L\b)		lower precision	$O(n^2)$
r = b – Ax		FP64 precision	$O(n^2)$
WHILE || r || not small enough			
1. find a correction "z" to adjust x that satisfy Az=r solving Az=r could be done by either:			
➢ z = U\(L\r)	Classical Iterative Refinement	lower precision	$O(n^2)$
➢ GMRes preconditioned by the LU to solve Az=r	Iterative Refinement using GMRes	lower precision	$O(n^2)$
2. x = x + z		FP64 precision	$O(n^1)$
3. r = b – Ax		FP64 precision	$O(n^2)$
END			

Higham and Carson showed can solve the inner problem with iterative method and not infect the solution.

- 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.
- Need the original matrix to compute residual (r) and matrix cannot be too badly conditioned

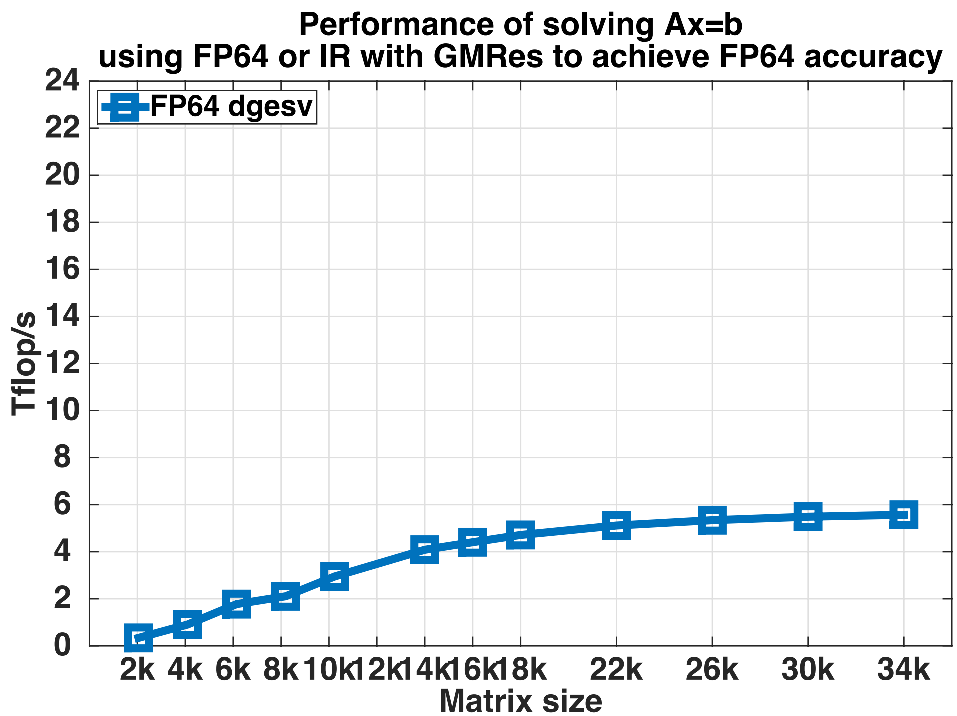
Leveraging Half Precision in HPC on V100





Matrix of size 10240 generated with positive λ and arithmetic distribution of its singular values $\sigma_i = 1 - (\frac{i-1}{n-1})(1 - \frac{1}{cond})$ and where its condition number is equal to 10^2.

Leveraging Half Precision in HPC on V100

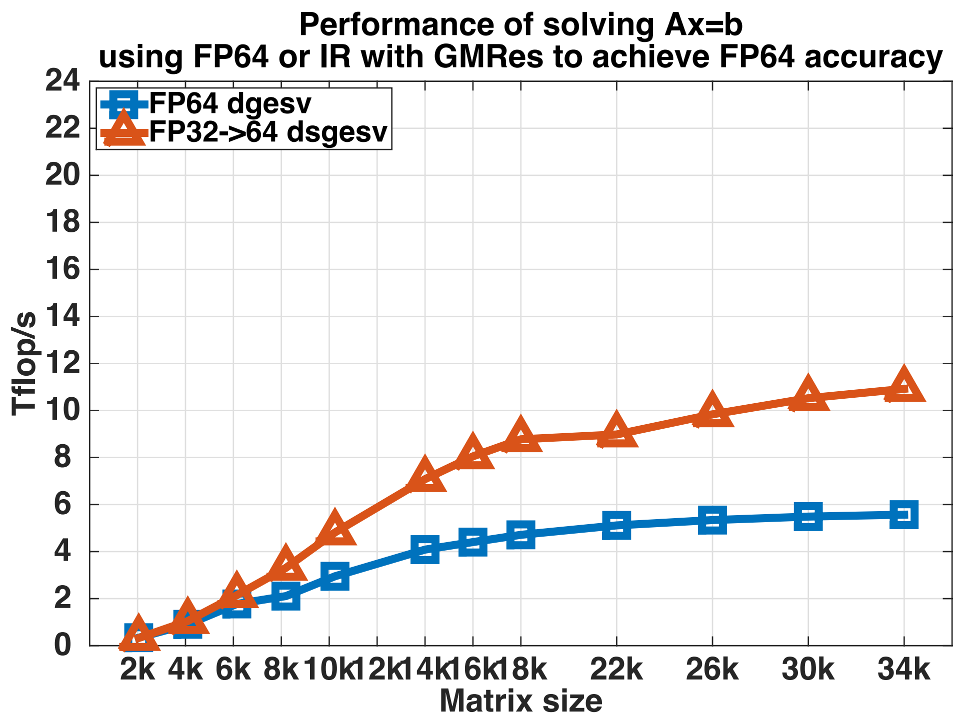


Flops = $2n^3/(3 \text{ time})$
meaning twice higher is twice faster

- solving Ax = b using **FP64 LU**

Matrices generated with positive λ and arithmetic distribution of its singular values $\sigma_i = 1 - (\frac{i-1}{n-1})(1 - \frac{1}{cond})$ and where its condition number is equal to 10^2.

Leveraging Half Precision in HPC on V100

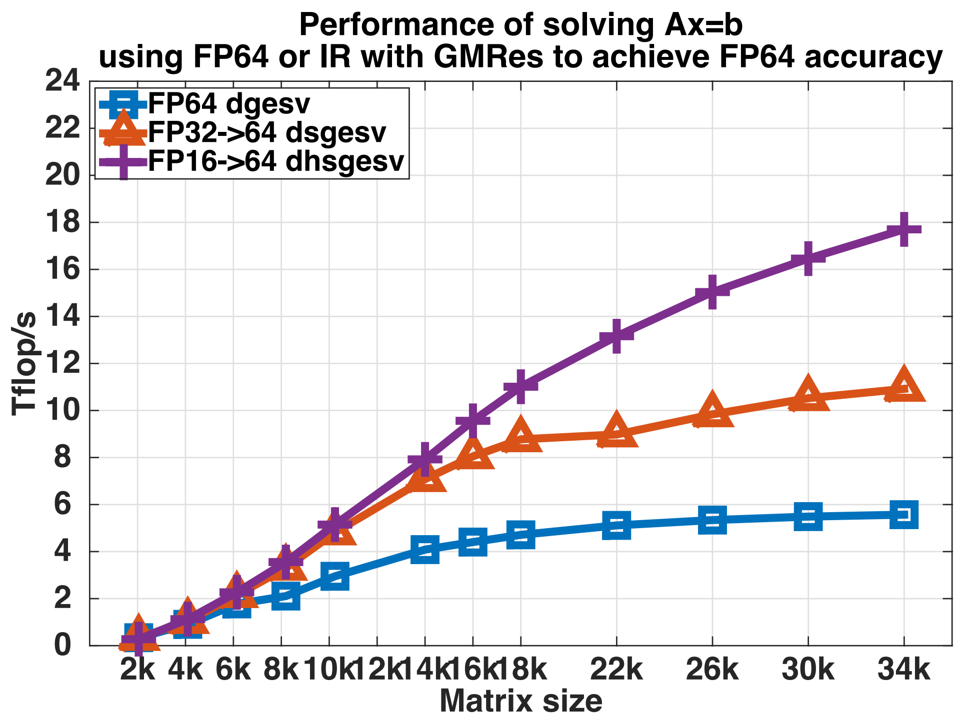


Flops = $2n^3/(3 \text{ time})$
meaning twice higher is twice faster

- solving Ax = b using **FP64 LU**
- solving Ax = b using **FP32 LU** and iterative refinement to achieve FP64 accuracy

Matrices generated with positive λ and arithmetic distribution of its singular values $\sigma_i = 1 - (\frac{i-1}{n-1})(1 - \frac{1}{cond})$ and where its condition number is equal to 10^2.

Leveraging Half Precision in HPC on V100



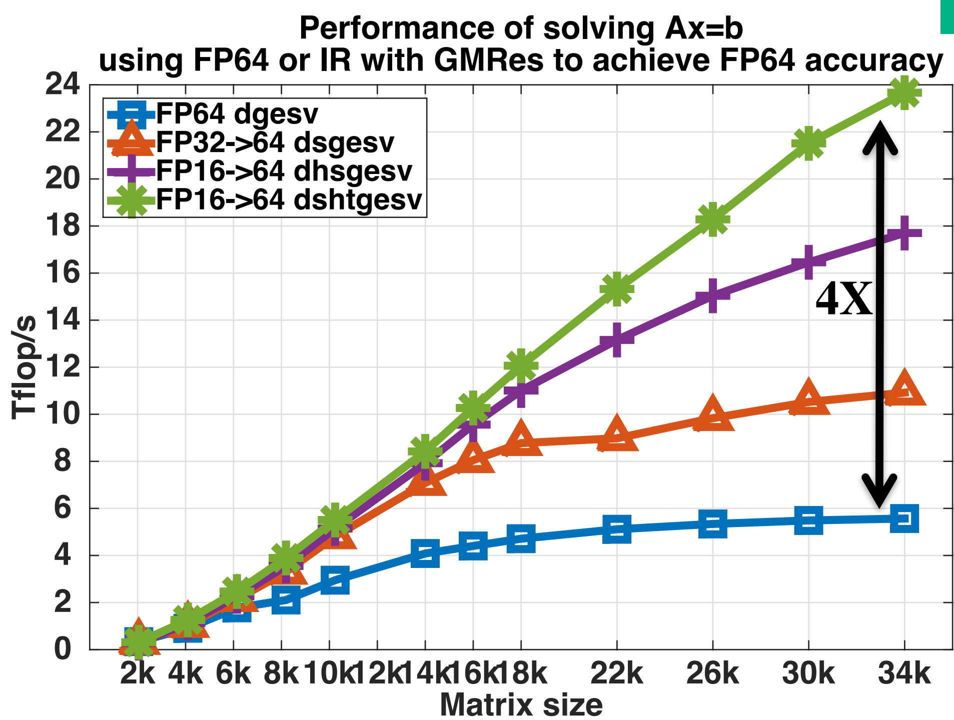
Flops = $2n^3/(3\ \text{time})$
meaning twice higher is twice faster

- solving Ax = b using **FP64 LU**
- solving Ax = b using **FP32 LU** and iterative refinement to achieve FP64 accuracy
- solving Ax = b using **FP16 LU** and iterative refinement to achieve FP64 accuracy

Matrices generated with positive λ and arithmetic distribution of its singular values $\sigma_i = 1 - (\frac{i-1}{n-1})(1 - \frac{1}{cond})$ and where its condition number is equal to 10^2.

Leveraging Half Precision in HPC on V100

Showing Tflop/s but really time to solution

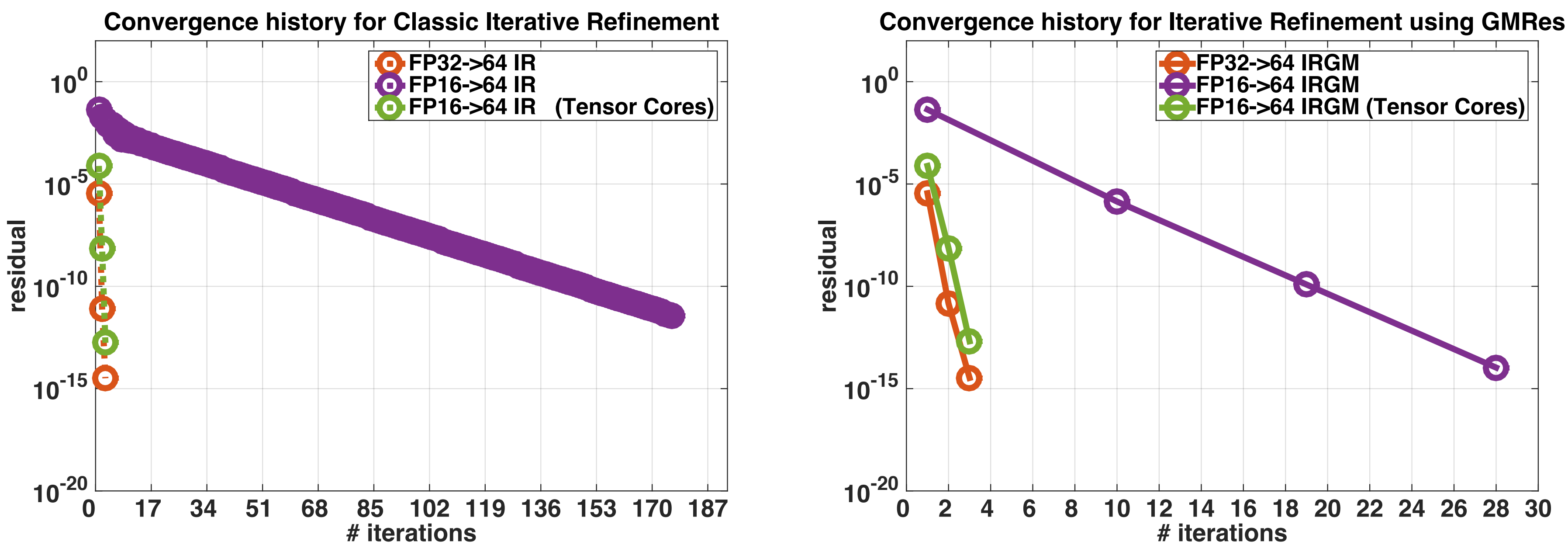


Flops = $2n^3/(3\ \text{time})$
meaning twice higher is twice faster

- solving Ax = b using **FP64 LU**
- solving Ax = b using **FP32 LU** and iterative refinement to achieve FP64 accuracy
- solving Ax = b using **FP16 LU** and iterative refinement to achieve FP64 accuracy
- solving Ax = b using **FP16 Tensor Cores LU** and iterative refinement to achieve FP64 accuracy

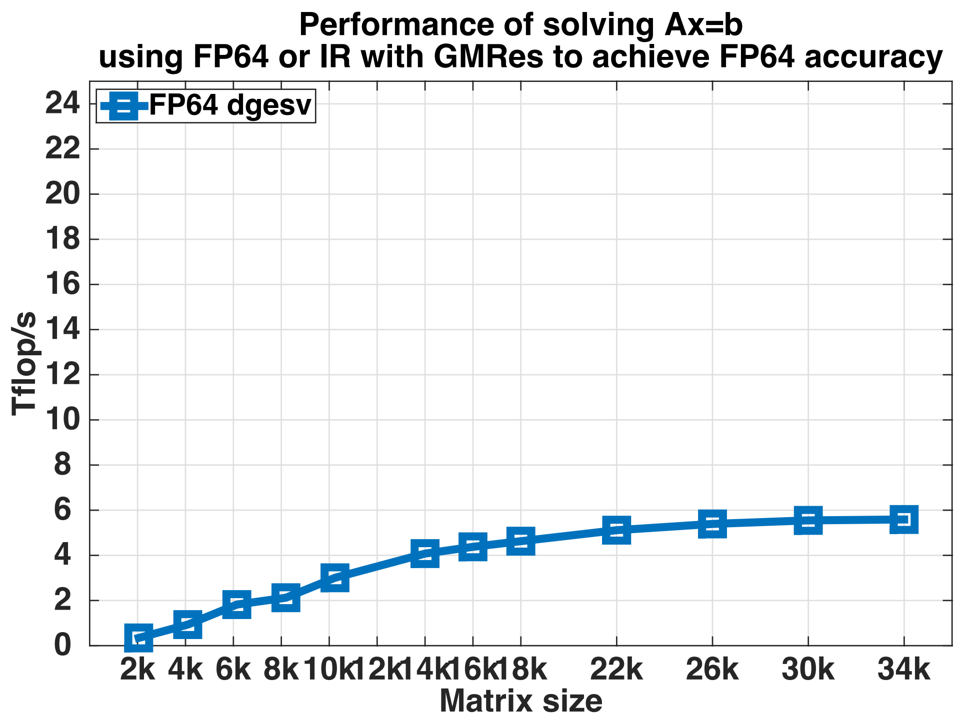
Matrices generated with positive λ and arithmetic distribution of its singular values $\sigma_i = 1 - (\frac{i-1}{n-1})(1 - \frac{1}{cond})$ and where its condition number is equal to 10^2.

Leveraging Half Precision in HPC on V100



Matrix of size 10240 generated with positive λ and clustered singular values, $\sigma_i = (1, \cdots, 1, \frac{1}{cond})$ and where its condition number is equal to 10^2.

Leveraging Half Precision in HPC on V100

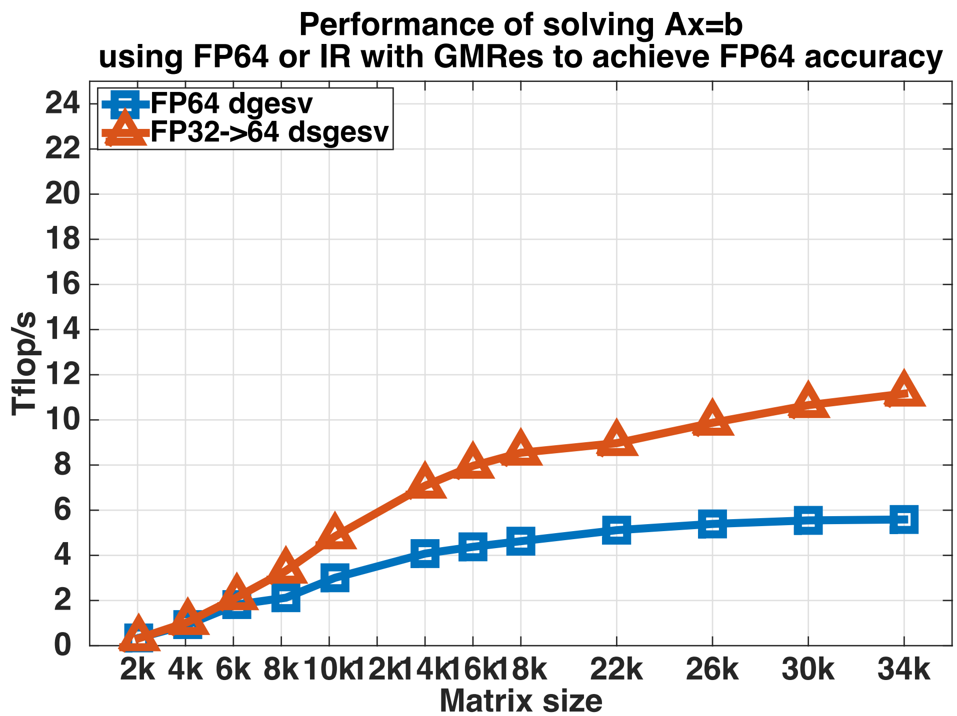


Flops = $2n^3/(3\ \text{time})$
meaning twice higher is twice faster

- solving Ax = b using **FP64 LU**

Matrices generated with positive λ and clustered distribution of its singular values σ_i=$(1, \cdots, 1, \frac{1}{cond})$ and where its condition number is equal to 10^2.

Leveraging Half Precision in HPC on V100

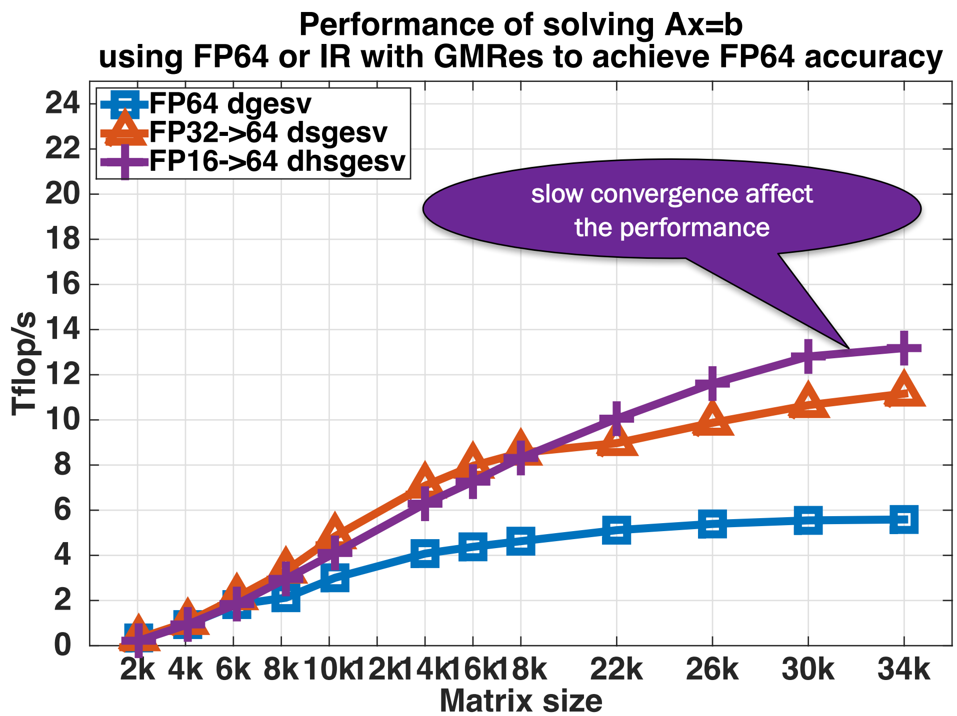


Flops = $2n^3/(3\ \text{time})$
meaning twice higher is twice faster

- solving Ax = b using **FP64 LU**
- solving Ax = b using **FP32 LU** and iterative refinement to achieve FP64 accuracy

Matrices generated with positive λ and clustered distribution of its singular values $\sigma_i = (1, \cdots, 1, \frac{1}{cond})$ and where its condition number is equal to 10^2.

Leveraging Half Precision in HPC on V100

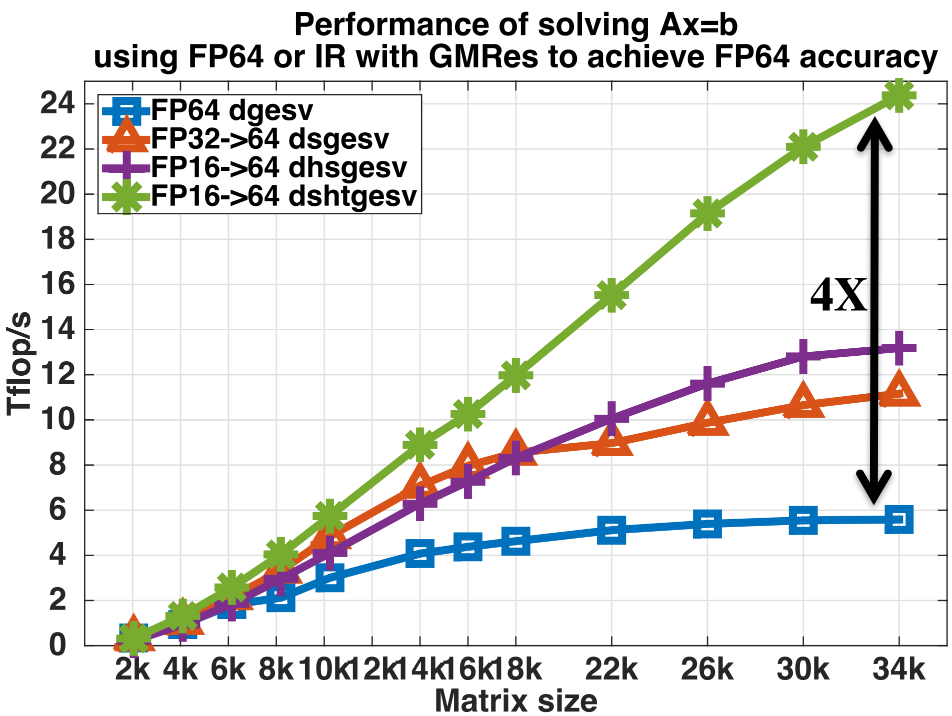


Flops = $2n^3/(3 \text{ time})$
meaning twice higher is twice faster

- solving Ax = b using **FP64 LU**
- solving Ax = b using **FP32 LU** and iterative refinement to achieve FP64 accuracy
- solving Ax = b using **FP16 LU** and iterative refinement to achieve FP64 accuracy

Matrices generated with positive λ and clustered distribution of its singular values σ_i=$(1, \cdots, 1, \frac{1}{cond})$ and where its condition number is equal to 10^2.

Leveraging Half Precision in HPC on V100

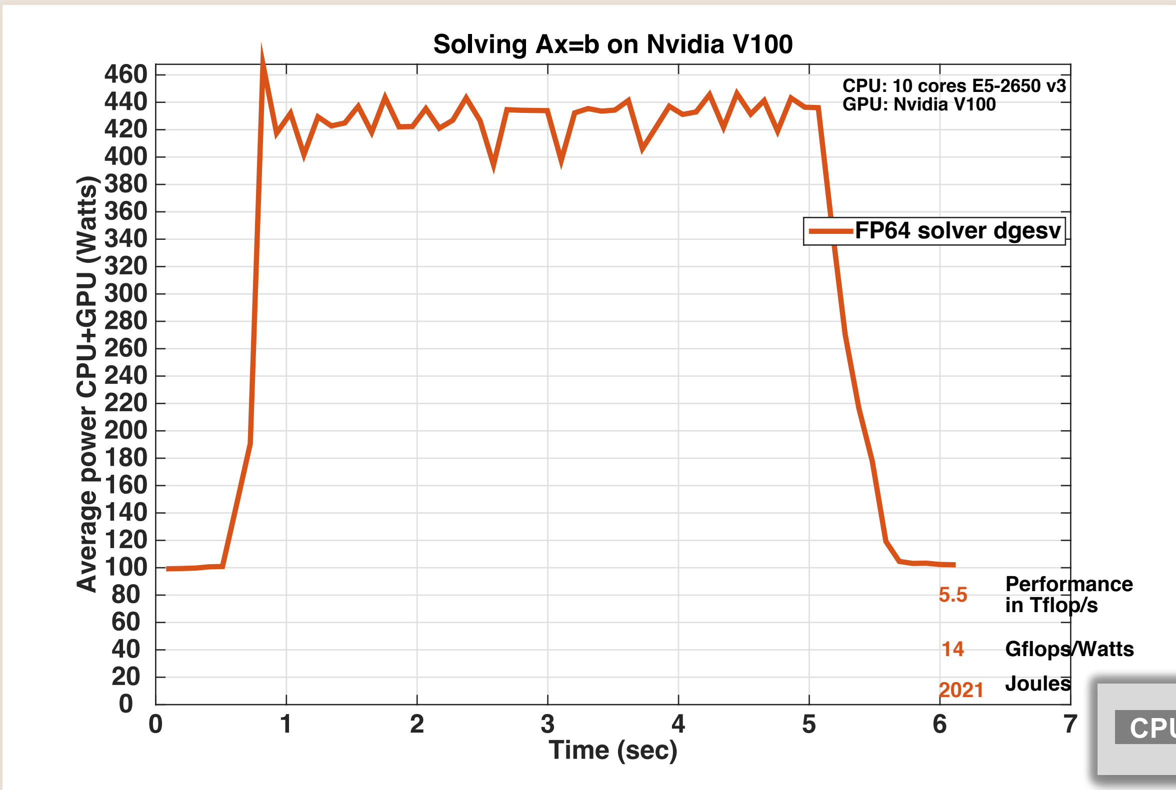


Flops = $2n^3/(3\text{ time})$
meaning twice higher is twice faster

- solving Ax = b using **FP64 LU**
- solving Ax = b using **FP32 LU** and iterative refinement to achieve FP64 accuracy
- solving Ax = b using **FP16 LU** and iterative refinement to achieve FP64 accuracy
- solving Ax = b using **FP16 Tensor Cores LU** and iterative refinement to achieve FP64 accuracy

Matrices generated with positive λ and clustered distribution of its singular values $\sigma_i = (1, \cdots, 1, \frac{1}{cond})$ and where its condition number is equal to 10^2.

Leveraging Half Precision in HPC
Power awareness



- Power consumption of the FP64 algorithm to solve Ax=b for a matrix of size 34K, it achieve 5.5 Tflop/s and requires about 2021 joules providing about 14 Gflops/Watts.

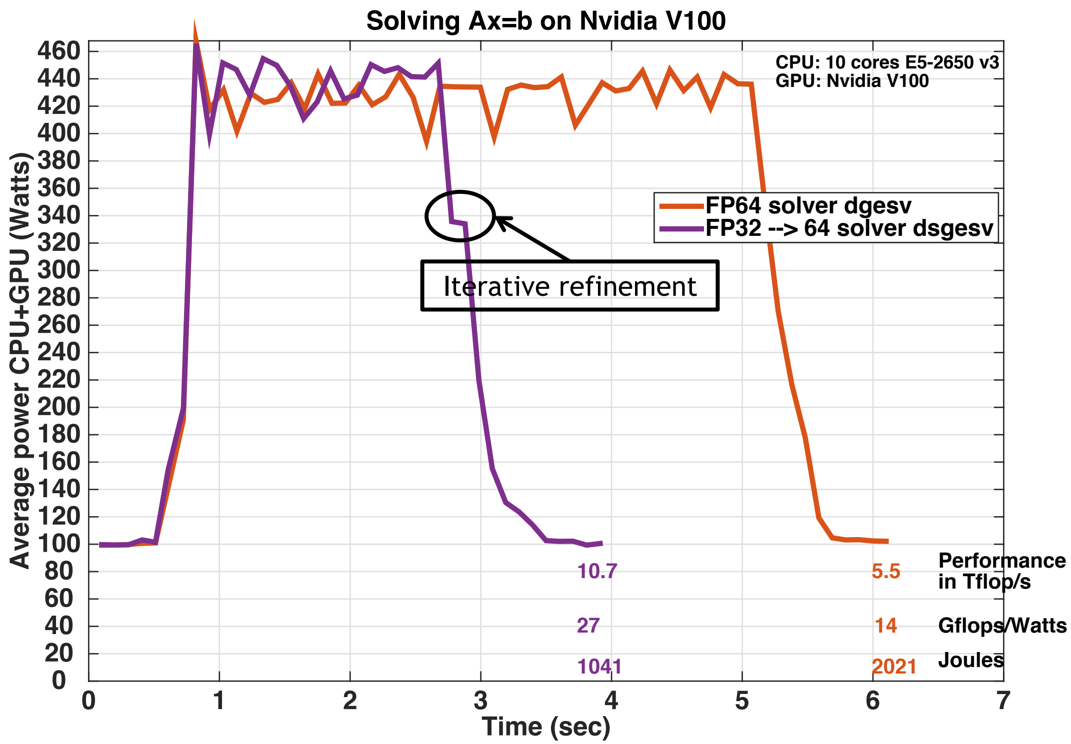
Power is for GPU + CPU + DRAM

CPU	Intel Xeon E5-2650 v3 (Haswell) 2x10 cores @ 2.30 GHz	V100	NVIDIA Volta GPU 80 MP x 64 @ 1.38 GHz

Problem generated with an arithmetic distribution of the singular values $\sigma_i = 1 - (\frac{i-1}{n-1})(1 - \frac{1}{cond})$ and positive eigenvalues.

Leveraging Half Precision in HPC
Power awareness

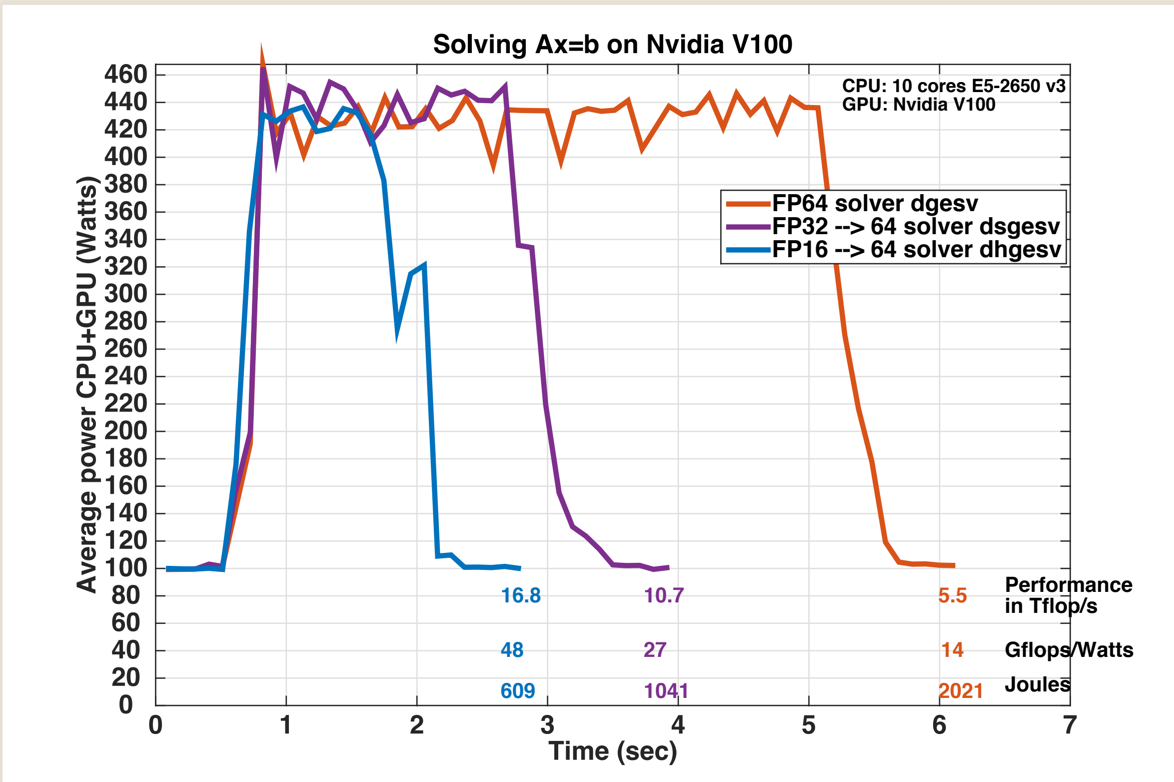
Mixed precision techniques can provide a large gain in energy efficiency



- Power consumption of the FP64 algorithm to solve Ax=b for a matrix of size 34K, it achieve 5.5 Tflop/s and requires about 2021 joules providing about 14 Gflops/Watts.
- Power consumption of the mixed precision FP32→64 algorithm to solve Ax=b for a matrix of size 34K, it achieve 10.7 Tflop/s and requires about 1041 joules providing about 30 Gflops/Watts.

Problem generated with an arithmetic distribution of the singular values $\sigma_i = 1 - (\frac{i-1}{n-1})(1 - \frac{1}{cond})$ and positive eigenvalues.

Leveraging Half Precision in HPC
Power awareness



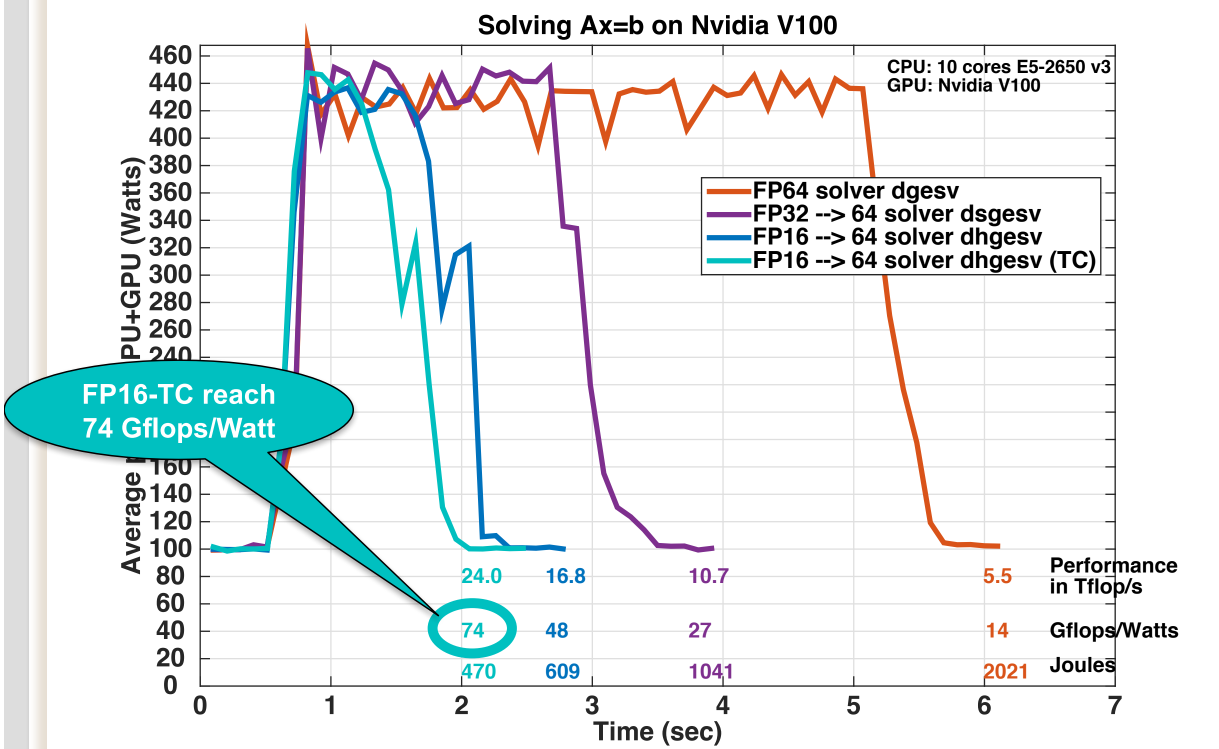
Mixed precision techniques can provide a large gain in energy efficiency

- Power consumption of the FP64 algorithm to solve Ax=b for a matrix of size 34K, it achieve 5.5 Tflop/s and requires about 2021 joules providing about 14 Gflops/Watts.
- Power consumption of the mixed precision FP32→64 algorithm to solve Ax=b for a matrix of size 34K, it achieve 10.7 Tflop/s and requires about 1041 joules providing about 30 Gflops/Watts.
- Power consumption of the mixed precision FP16→64 algorithm to solve Ax=b for a matrix of size 34K, it achieve 16.8 Tflop/s and requires about 609 joules providing about 48 Gflops/Watts.

Problem generated with an arithmetic distribution of the singular values $\sigma_i = 1 - (\frac{i-1}{n-1})(1 - \frac{1}{cond})$ and positive eigenvalues.

Leveraging Half Precision in HPC
Power awareness

Mixed precision techniques can provide a large gain in energy efficiency



- Power consumption of the FP64 algorithm to solve Ax=b for a matrix of size 34K, it achieve 5.5 Tflop/s and requires about 2021 joules providing about 14 Gflops/Watts.
- Power consumption of the mixed precision FP32→64 algorithm to solve Ax=b for a matrix of size 34K, it achieve 10.7 Tflop/s and requires about 1041 joules providing about 30 Gflops/Watts.
- Power consumption of the mixed precision FP16→64 algorithm to solve Ax=b for a matrix of size 34K, it achieve 16.8 Tflop/s and requires about 609 joules providing about 48 Gflops/Watts.
- Power consumption of the mixed precision FP16→64 TC algorithm using Tensor Cores to solve Ax=b for a matrix of size 34K, it achieve 24 Tflop/s and requires about 470 joules providing about 74 Gflops/Watts.

Problem generated with an arithmetic distribution of the singular values $\sigma_i = 1 - (\frac{i-1}{n-1})(1 - \frac{1}{cond})$ and positive eigenvalues.

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

MAGMA PLASMA