

University of Arkansas, Fayetteville

ScholarWorks@UARK

Mathematical Sciences Spring Lecture Series

Mathematical Sciences

4-6-2021

Lecture 11: The Road to Exascale and Legacy Software for Dense Linear Algebra

Jack Dongarra
University of Tennessee

Follow this and additional works at: <https://scholarworks.uark.edu/mascsls>



Part of the [Algebra Commons](#), [Computer and Systems Architecture Commons](#), [Databases and Information Systems Commons](#), [Data Storage Systems Commons](#), [Digital Communications and Networking Commons](#), [Numerical Analysis and Computation Commons](#), [Numerical Analysis and Scientific Computing Commons](#), [Ordinary Differential Equations and Applied Dynamics Commons](#), and the [Programming Languages and Compilers Commons](#)

Citation

Dongarra, J. (2021). Lecture 11: The Road to Exascale and Legacy Software for Dense Linear Algebra. *Mathematical Sciences Spring Lecture Series*. Retrieved from <https://scholarworks.uark.edu/mascsls/10>

This Video is brought to you for free and open access by the Mathematical Sciences at ScholarWorks@UARK. It has been accepted for inclusion in Mathematical Sciences Spring Lecture Series by an authorized administrator of ScholarWorks@UARK. For more information, please contact ccmiddle@uark.edu.

46th University of Arkansas Spring Lecture Series
Virtual Conference

Scalable Solvers: Universals and Innovations

The Road to Exascale and Legacy Software for Dense Linear Algebra

Jack Dongarra

University of Tennessee
Oak Ridge National Laboratory
University of Manchester

Copy of slides at <http://bit.ly/dongarra-arkansas-042021>

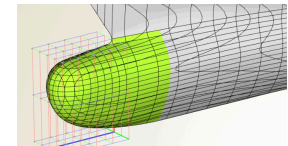
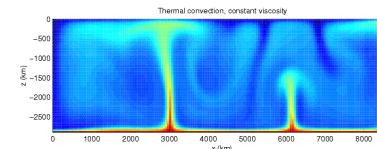
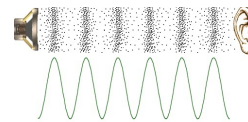
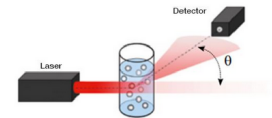
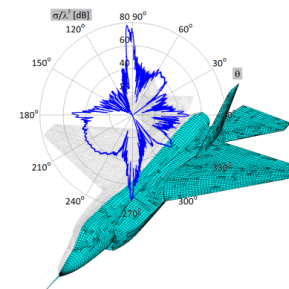
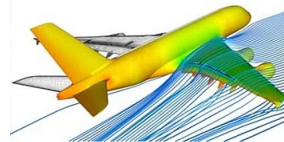
Dense Linear Algebra

- 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	yes	X	X		X		X	C	M	X								2018-09-15
DPLASMA	BSD	yes	X	X		X		X	C	M	X								2014-04-14
Eigen	MPL2	yes	X	X			X	X			X	X		X	X				2018-07-23
Elemental	New BSD	yes	X	X			X			M	X	X	X	X					2017-02-06
ELPA	LGPL	yes	X	X	F90	X		X		M	X								2018-06-01
FLENS	BSD	yes	X	X			X	X			X								2014-05-11
LAPACK	BSD	yes	X	X	X	X		X			X								2017-11-12
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		2018-06-25
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								2018-09-04
ScaLAPACK	BSD	yes	X	X	X	X				M/P	X								2018-08-20
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	2016-01-20

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

◆ LINPACK, EISPACK, LAPACK, ScaLAPACK

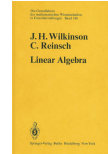
➤ PLASMA, MAGMA



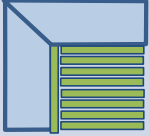








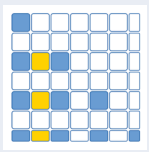
DLA Solvers

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

Over the Past 50 Years Evolving SW and Alg Tracking Hardware Developments



Software/Algorithms follow hardware evolution in time

<p>EISPACK (1970's) (Translation of Algol to F66)</p>			<p>Rely on - Fortran, but row oriented</p>
<p>LINPACK (1980's) (Vector operations)</p>			<p>Rely on - Level-1 BLAS operations - Column oriented</p>
<p>LAPACK (1990's) (Blocking, cache friendly)</p>			<p>Rely on - Level-3 BLAS operations</p>
<p>ScaLAPACK (2000's) (Distributed Memory)</p>			<p>Rely on - PBLAS Mess Passing</p>
<p>PLASMA / MAGMA (2010's) (Many-core friendly & GPUs)</p>			<p>Rely on - DAG/scheduler - block data layout</p>
<p>SLATE (2020's) (DM and Heterogeneous arch)</p>			<p>Rely on C++ - Tasking DAG scheduling - Tiling, but tiles can come from anywhere - Heterogeneous HW, Batched dispatch</p>

What do we mean by performance?

- ◆ **What is the unit: floating point operations per second (flop/s)?**
 - flop/s is a rate of execution, some number of floating point operations per second.
 - Whenever this term is used it will refer to 64 bit floating point operations and the operations will be either addition or multiplication.
 - Tflop/s refers to trillions (10^{12}) of floating point operations per second
 - Pflop/s refers to 10^{15} floating point operations per second.
 - Eflop/s is 10^{18} 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 64-bit precision) that can be completed during a period of time, usually the cycle time of the machine.

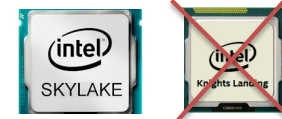
 - For example, an Intel Skylake core at 2.1 GHz can complete 32 floating point operations per cycle or a theoretical peak performance per core of:
 $32 \text{ fl.pt. ops / cycle} * 2.1 \text{ G-cycles / second} = 67.2 \text{ Gflop/s}$
 - With 24 cores per socket: $24 * 67.2 \text{ Gflop/s}$ or 1.61 Tflop/s for the socket.

Peak Performance - Per Core

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

Floating point operations per cycle per core

- Most of the recent computers have FMA (Fused multiple add):
(i.e. $x \leftarrow x + y * z$ in one cycle)
- Intel Xeon earlier models and AMD Opteron have SSE2
 - 2 flops/cycle/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 (vector instructions)
 - 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
- 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, Kaby Lake, Coffee Lake, Ice Lake...
 - 32 flops/cycle/core DP & 64 flops/cycle/core SP
- Next Gen Sapphire Rapids. with AMX (matrix instructions)



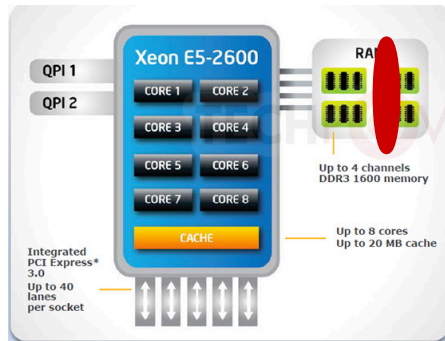
We are here





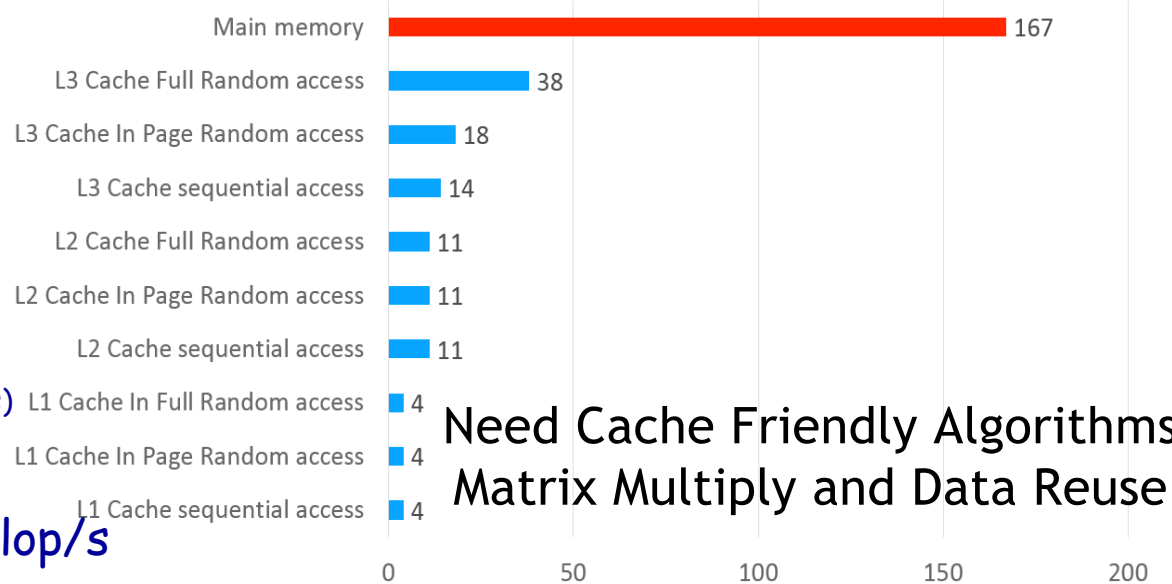
Commodity Processors ...

Over provisioned for floating point operations
Today it's all about data movement



Memory Access Latencies in Clock Cycles
167 cycles to move a word from memory to a register

In 167 cycles single core: 5344 DP Flops, socket: >40K Flops



Each Core: 32 Flops per core / cycle
With 2.6 GHz

$(32 \text{ flops/cycle} * 2.6 \text{ Gcycles/sec} = 83.2 \text{ Gflop/s})$

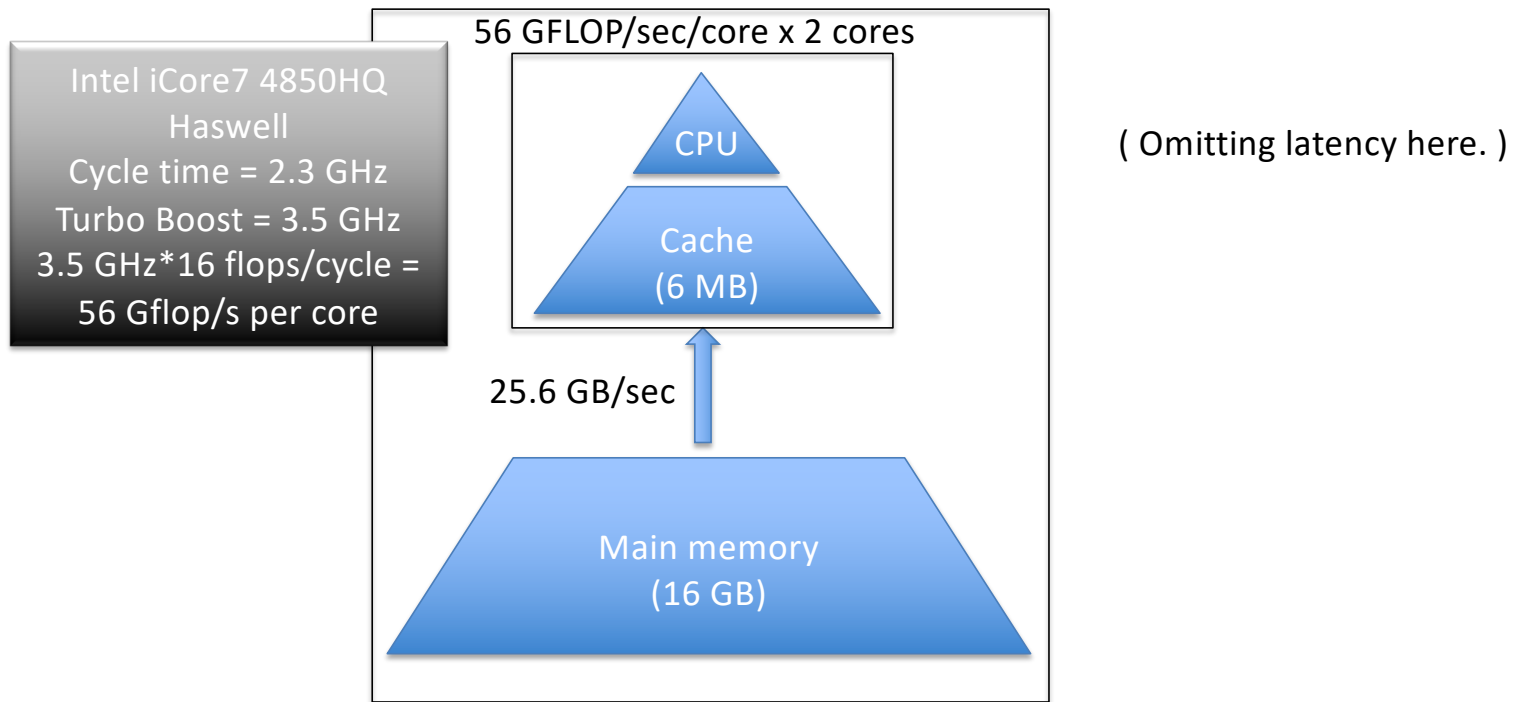
Each Core Peak DP 83.2 Gflop/s

Each Socket (8 cores) Peak 665.6 Gflop/s

Need Cache Friendly Algorithms
Matrix Multiply and Data Reuse

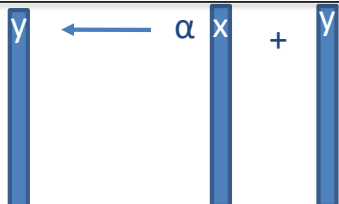
Memory transfer

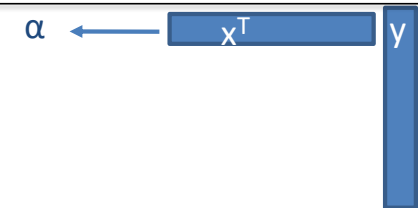
- One level of memory model on my laptop:



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

AXPY:  `for (j = 0; j < n; j++)
y[i] += a * x[i];` **n MUL**
n ADD
2n FLOP
n FMA
(without increment)

DOT:  `alpha = 0e+00;
for (j = 0; j < n; j++)
alpha += x[i] * y[i];` **n MUL**
n ADD
2n FLOP
n FMA
(without increment)

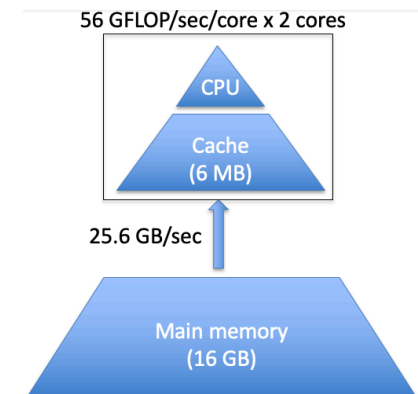
Note: It is reasonable to expect the one loop codes shown here to perform as well as their Level 1 BLAS counterpart (on multicore with an OpenMP pragma for example).

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



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

- 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

$$\begin{aligned} \text{total_time} &\geq \max (\text{time_comm} , \text{time_comp}) \\ &= \max (0.23\text{ms} , 0.01\text{ms}) = 0.23\text{ms} \end{aligned}$$

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

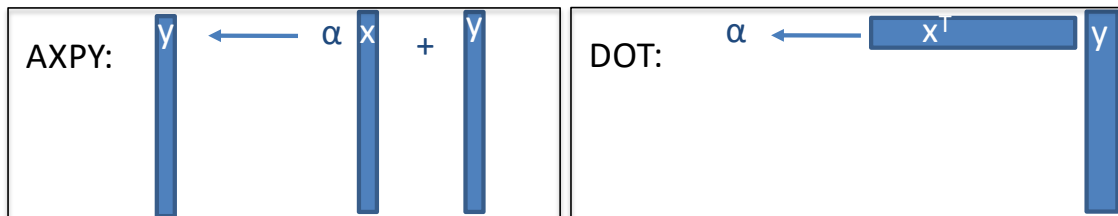
Performance for DOT \leq 3.2 Gflop/s

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

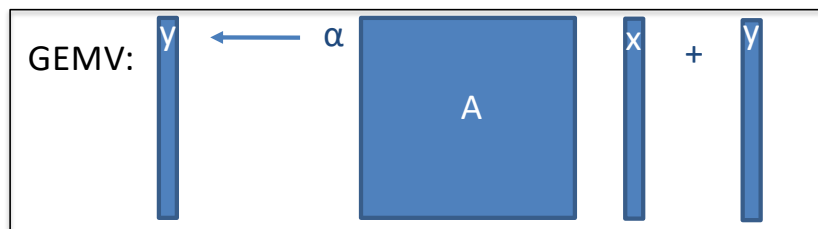
2n memory references

AXPY: 2n READ, n WRITE

DOT: 2n READ

RATIO FLOPs to Memory Ops: 1:1

Level 2 BLAS Matrix-Vector operations

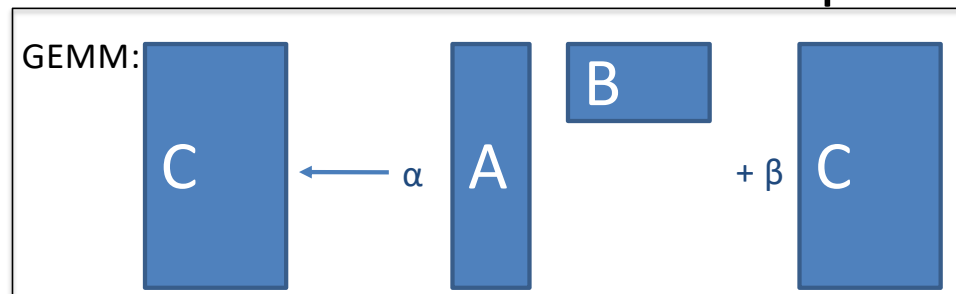


2n² FLOPs

n² memory references

RATIO FLOPs to Memory Ops: 2:1

Level 3 BLAS Matrix-Matrix operations



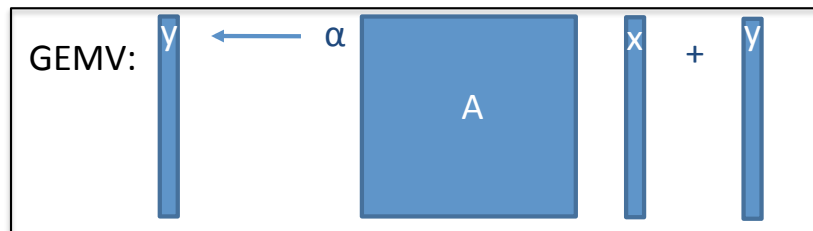
2n³ FLOPs

3n² memory references

3n² READ, n² WRITE

RATIO FLOPs to Memory Ops: n:2

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



- Data size:

– $(860^2 + 2*860 \text{ double}) * (8 \text{ Bytes / double}) \sim 6 \text{ MBytes}$
 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

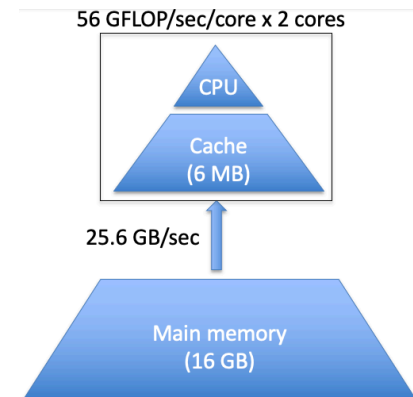
$$\begin{aligned} \text{total_time} &\geq \max (\text{time_comm} , \text{time_comp}) \\ &= \max (0.23\text{ms} , 0.026\text{ms}) = 0.23\text{ms} \end{aligned}$$

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

Performance for GEMV ≤ 6.4 Gflop/s

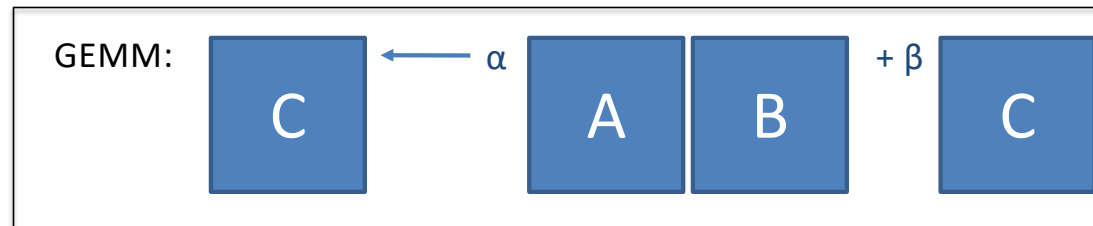
Performance for DOT ≤ 3.2 Gflop/s

Peak is 56 Gflop/s



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

- Take two double precision matrices A and B 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**

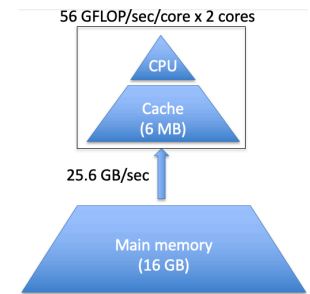
Matrix Matrix Operations

$$\begin{aligned} \text{total_time} &\geq \max(\text{time_comm}, \text{time_comp}) \\ &= \max(0.23\text{ms}, 4.46\text{ms}) = 4.46\text{ms} \end{aligned}$$

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 ≈ 55.5 Gflop/sec

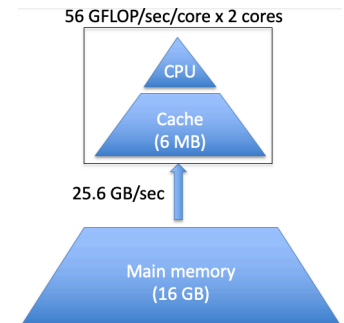
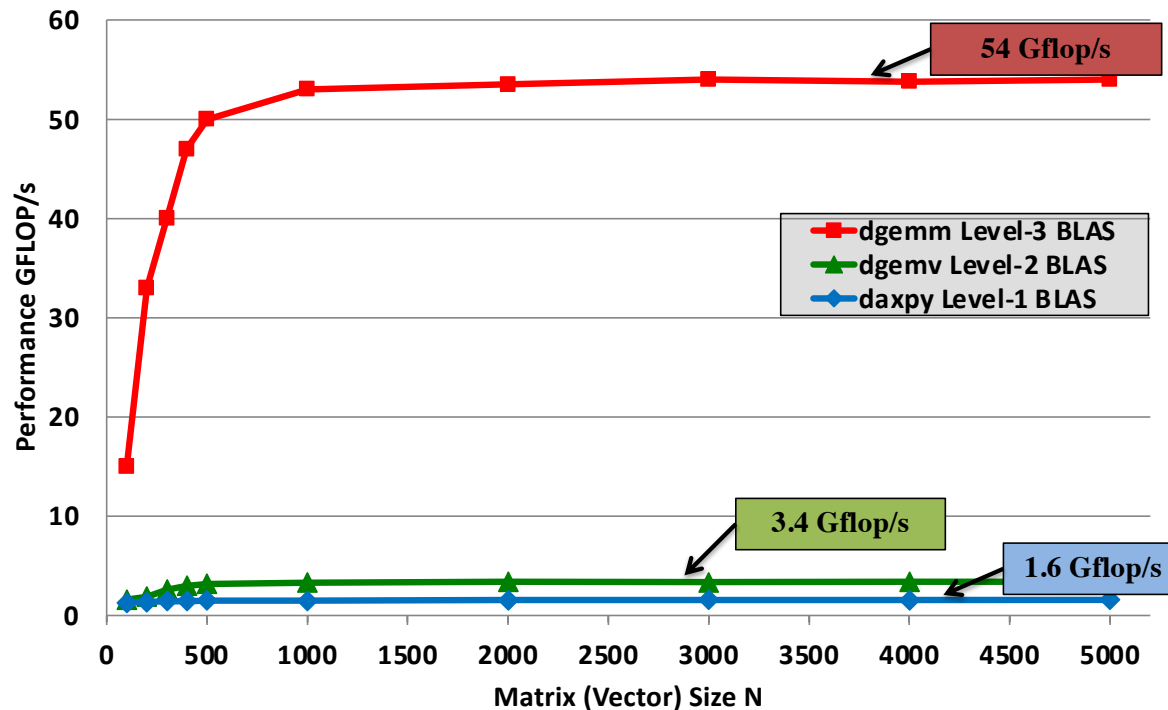
Performance for DOT ≤ 3.2 Gflop/s

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

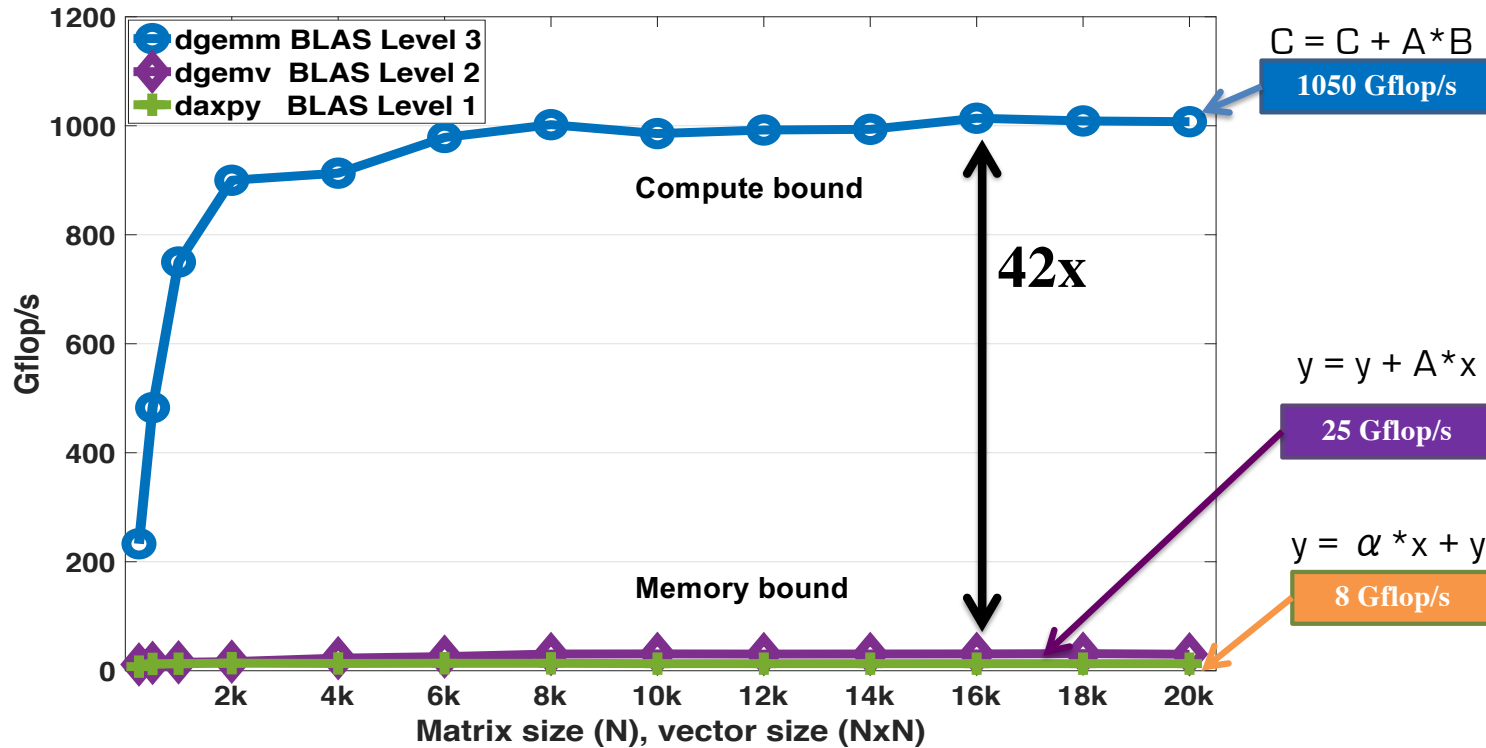
1 core Intel Haswell i7-4850HQ, 2.3 GHz (Turbo Boost at 3.5 GHz);
Peak = 56 Gflop/s



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

Level 1, 2 and 3 BLAS

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



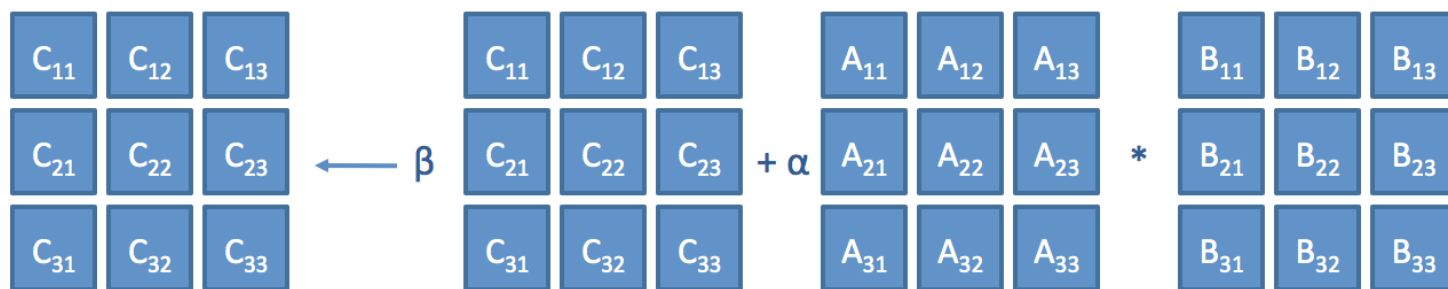
18 cores Intel Xeon Gold 6140, 2.3 GHz (Skylake)
The theoretical peak double precision is 1325 Gflop/s
Compiled with icc and using Intel MKL 2018

Issues

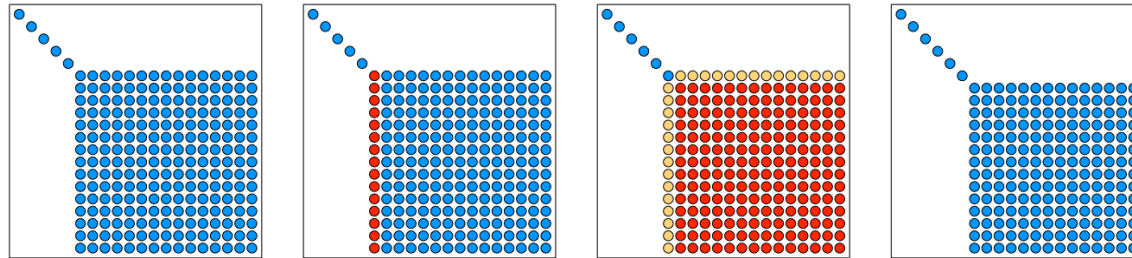
- Reuse based on matrices that fit into cache.
- What if you have matrices bigger than cache?

Issues

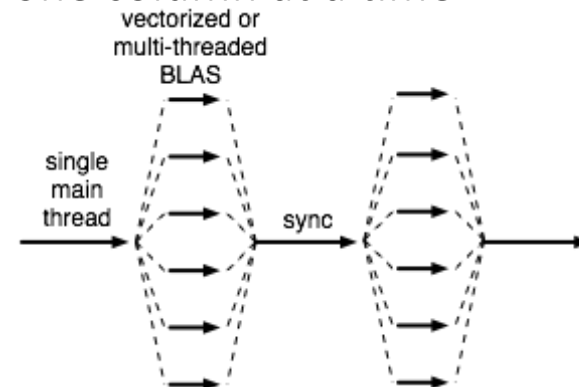
- Reuse based on matrices that fit into cache.
- What if you have matrices bigger than cache?
- Break matrices into blocks or tiles that will fit.



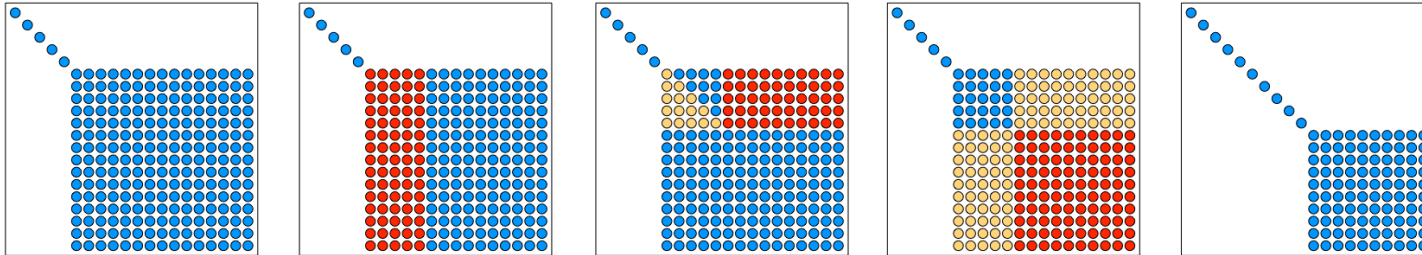
LU Factorization in LINPACK (1970's)



- 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 nb 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

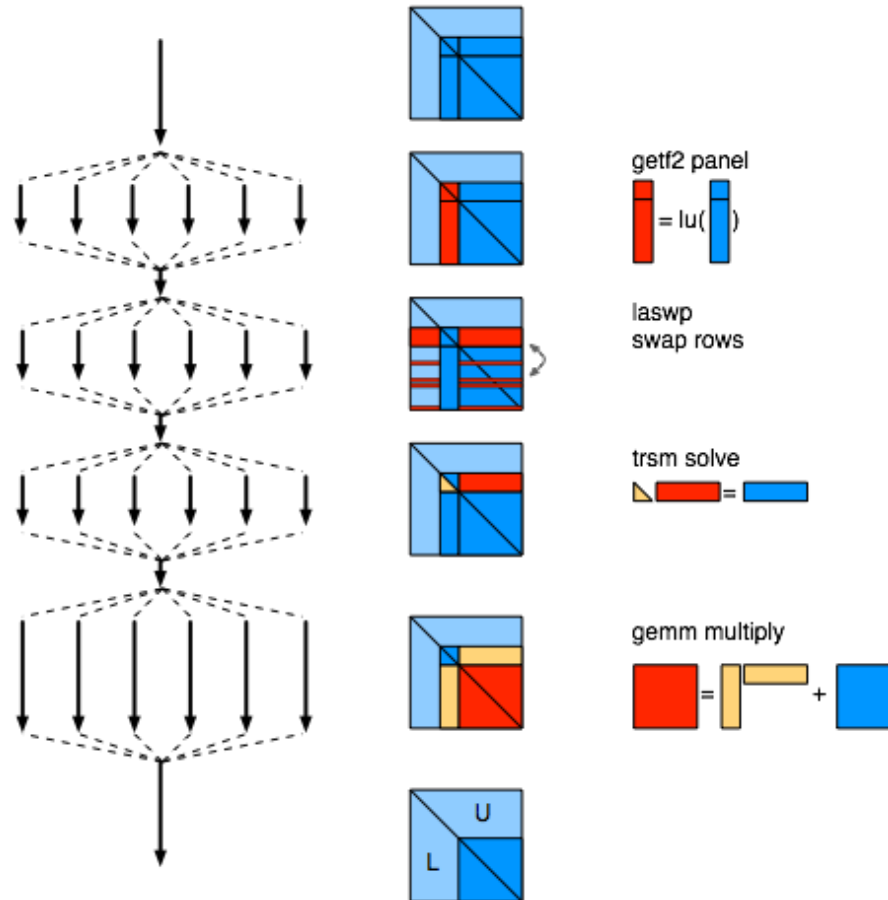
Parallelism in LAPACK

◆ **Most flops in gemm update**

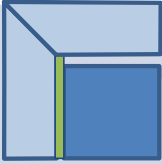


- $\frac{2}{3} n^3$ term
- Easily parallelized using multi-threaded BLAS
- Done in any reasonable software

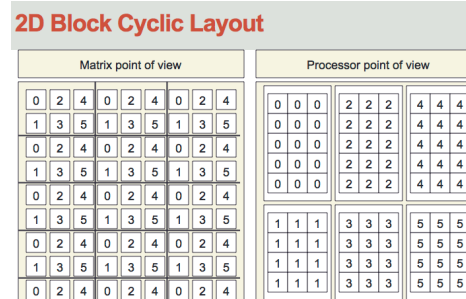
• **Other operations lower order**

- 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



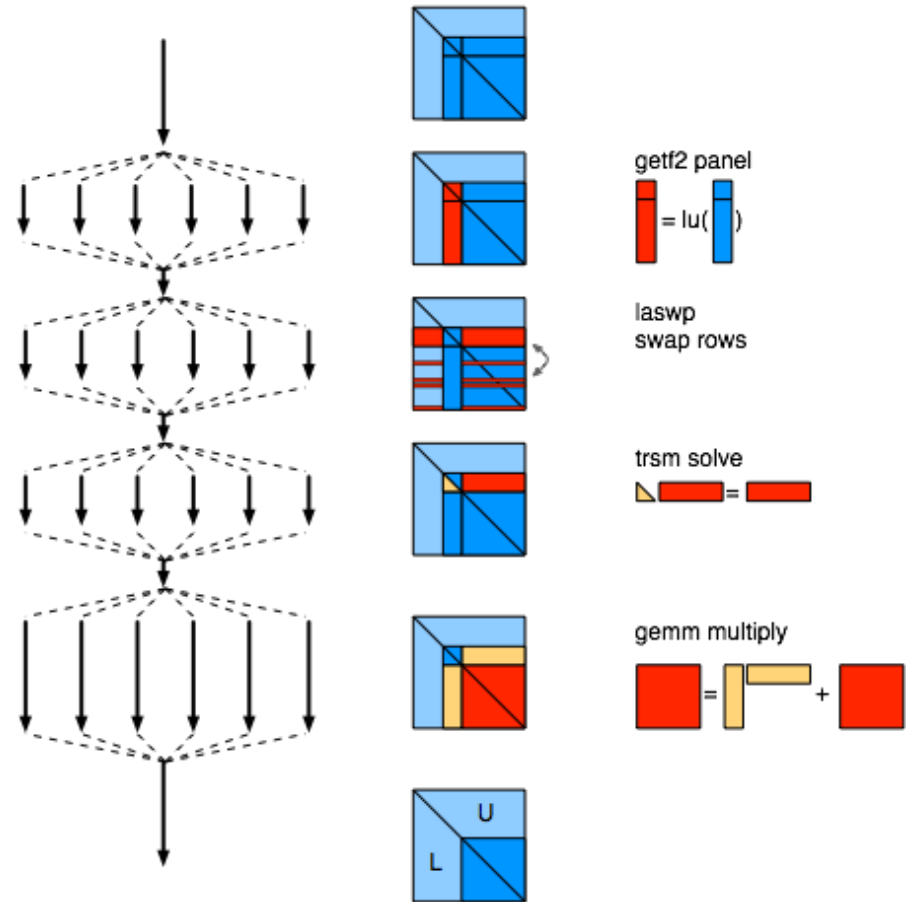
ScaLAPACK

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

Parallelism in ScaLAPACK

- Similar to LAPACK
- Bulk-synchronous processing
 - separate message passing & compute
- 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**,
 $p \times q = \# \text{ nodes}$
 $= \# \text{ MPI processes,}$
 $\text{num_threads} = \# \text{ cores/node}$



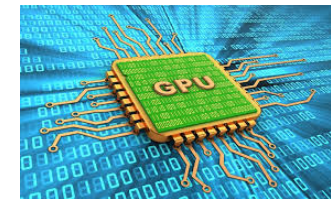
Today's HPC Environment for Numerical Libraries

- Highly parallel
 - Distributed memory
 - MPI + Open-MP programming model

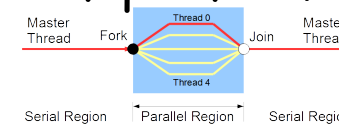


ORNL Summit, 200 Pflop/s, 4608 nodes
(node= 2-Power9 chips + 6-Nvidia GPUs)
 2.3×10^6 Cores

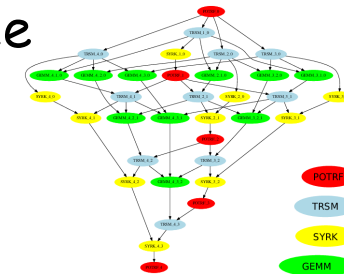
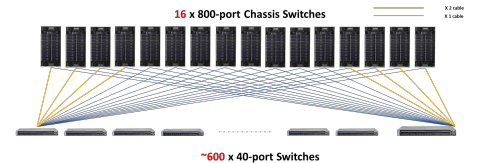
- Heterogeneous
 - Commodity processors + GPU accelerators



- Simple loop level parallelism too limiting in terms of performance



- Communication between parts very expensive compared to floating point ops



- Comparison of operation counts may not reflect time to solution



- Floating point hardware at 64, 32, and 16 bit levels

Type	Size	Range	$u = 2^{-l}$
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	$10^{\pm 4932}$	$2^{-113} \approx 9.6 \times 10^{-35}$

Yesterday's HPC

ScaLAPACK

- First released Feb 1995, 25 years old
- Lacks dynamic scheduling, look-ahead panels, communication avoiding algorithms, ...
- Can't be adequately retrofitted for accelerators
- Written in archaic language (Fortran 77)

SGI Origin 2000 (ASCI Blue Mountain, 1998)

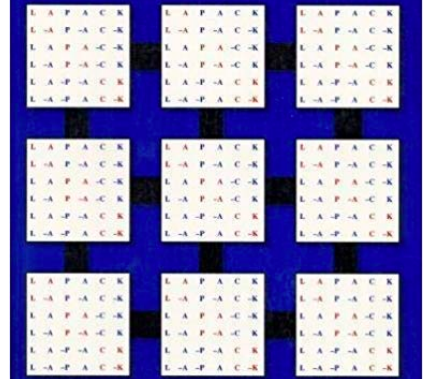
- 6,144 MIPS R10000
- 3 Tflop/s

ASCI Blue Mountain



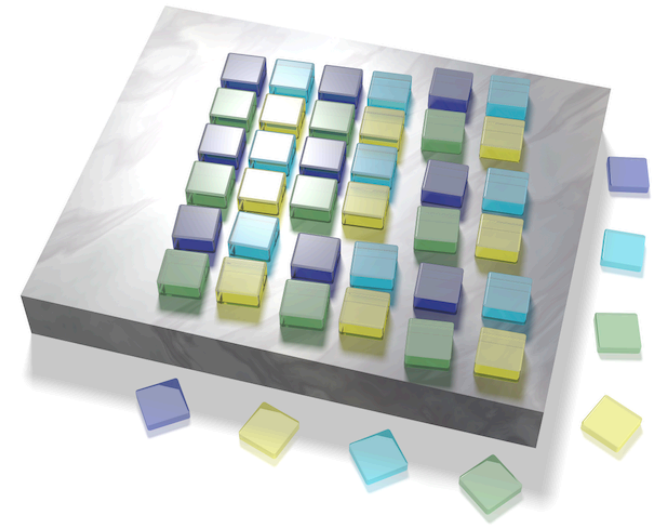
ScaLAPACK Users' Guide

L. S. Blackford · J. Choi · A. Cleary · E. D'Azevedo
J. Demmel · I. Dhillon · J. Dongarra · S. Hammarling
G. Henry · A. Petitet · K. Stanley · D. Walker · R. C. Whaley



SLATE: Software for Linear Algebra Targeting Exascale

- Distributed, GPU-accelerated, dense linear algebra library
 - Target large HPC machines
 - BLAS: matrix multiply ($C = AB$), etc.
 - Linear systems ($Ax = b$)
 - LU, Cholesky, symmetric indefinite
 - Least squares ($Ax \approx b$)
 - QR, LQ
 - Eigenvalue ($Ax = \lambda x$)
 - SVD ($A = U\Sigma V^H$)
- Modern replacement for ScaLAPACK
 - Explicit multi-threading (OpenMP)
 - C++



Coverage

Basic linear algebra ($C = AB, \dots$)

	ScaLAPACK	SLATE
Level 1 PBLAS	✓	✗ (use Level 3)
Level 2 PBLAS	✓	✗ (use Level 3)
Level 3 PBLAS	✓	✓
Matrix norms	✓	✓
Test matrix generation	✓	✓ (new)

Linear systems ($Ax = b$)

	ScaLAPACK	SLATE
LU (partial pivoting)	✓	✓
LU, band (pp)	✓	✓
LU (non-pivoting)	✗	✓ (new)
Cholesky	✓	✓
Cholesky, band	✓	✓ (new)
Symmetric Indefinite (Aasen)	✗	✓ (CPU only)
Mixed precision	✗	✓
Inverses (LU, Cholesky)	✓	✓

Least squares ($Ax \cong b$)

	ScaLAPACK	SLATE
QR	✓	✓
LQ	✓	✓ (new)
Least squares solver	✓	✓

SVD, eigenvalues ($A = U\Sigma V^H, Ax = \lambda x$)

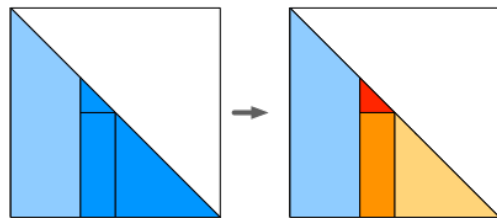
	ScaLAPACK	SLATE
SVD	✓	✓ values (new)
Symmetric eigenvalues	✓	✓ values (new)
Generalized symmetric eig.	✓	✓ values (new)
Polar decomposition (QDWH)	✗	✓ (new)
Non-symmetric eigenvalues	✗ pieces	✗ (2021–2022)

All SLATE routines listed are GPU-accelerated, except symmetric indefinite

(new) since Sep 2019 review

Tile Algorithms: Matrix Decomposition

LAPACK Algorithm (right looking)

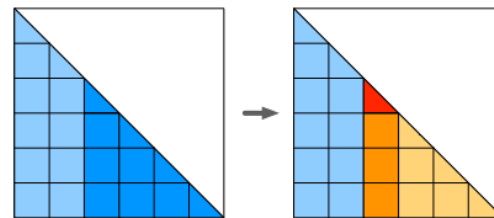


$$\text{Red Triangle} = \text{chol}(\text{Blue Triangle})$$

$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} \text{Blue Triangle} \\ \text{Blue Triangle} \\ \text{Blue Triangle} \end{bmatrix} / \text{Red Triangle} \quad \text{trsm}$$

$$\text{Yellow Triangle} = \begin{bmatrix} \text{Blue Triangle} \\ \text{Blue Triangle} \\ \text{Blue Triangle} \end{bmatrix} - \begin{bmatrix} 1 & 1^T & 2^T & 3^T \\ 2 & & & \\ 3 & & & \end{bmatrix} \quad \text{syrk}$$

SLATE: Tile Algorithm

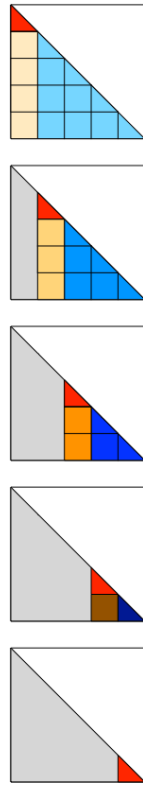


$$\text{Red Triangle} = \text{chol}(\text{Blue Triangle})$$

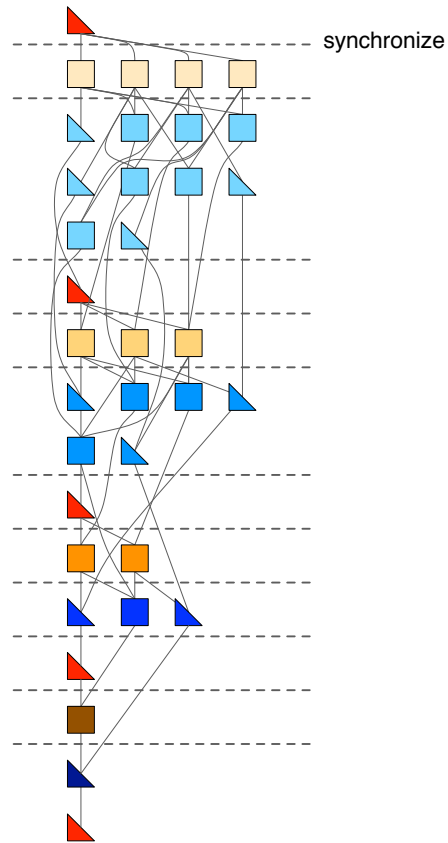
$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} \text{Blue Triangle} \\ \text{Blue Triangle} \\ \text{Blue Triangle} \end{bmatrix} / \text{Red Triangle} \quad \text{trsm}$$

$$\begin{bmatrix} \text{Yellow Triangle} \\ \text{Yellow Triangle} \\ \text{Yellow Triangle} \end{bmatrix} = \begin{bmatrix} \text{Blue Triangle} \\ \text{Blue Triangle} \\ \text{Blue Triangle} \end{bmatrix} - \begin{bmatrix} 1 & 1^T \\ 2 & 2^T \\ 3 & 3^T \end{bmatrix} \quad \text{syrk}$$

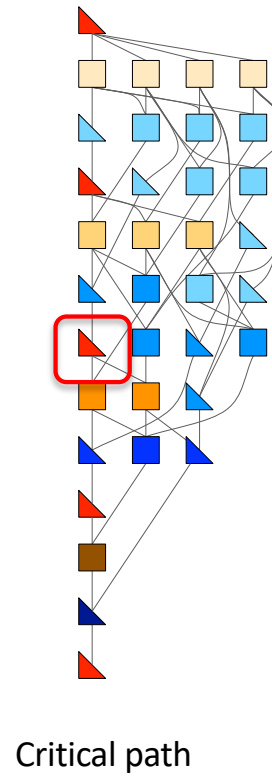
Track dependencies — Directed acyclic graph (DAG)



Fork-join schedule on 4 cores
with artificial synchronizations

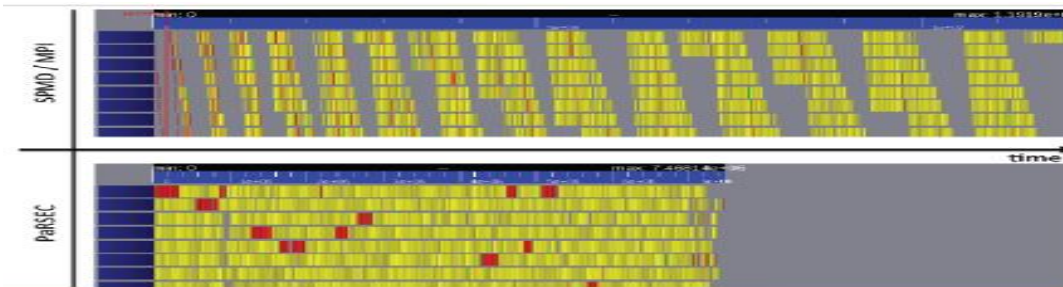
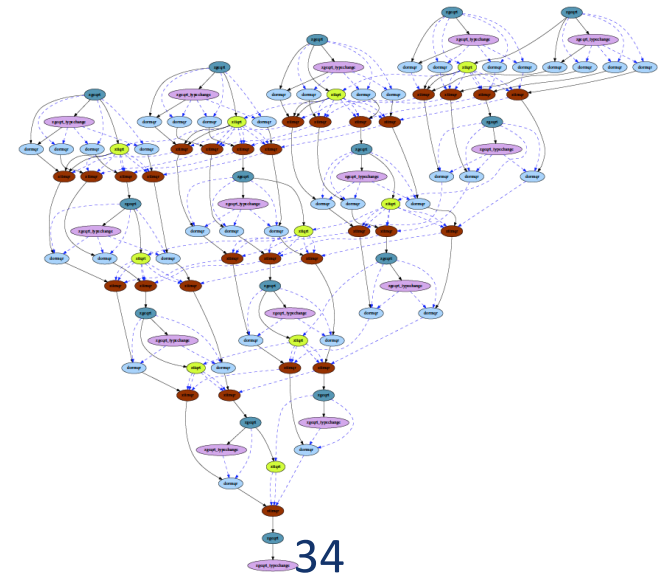
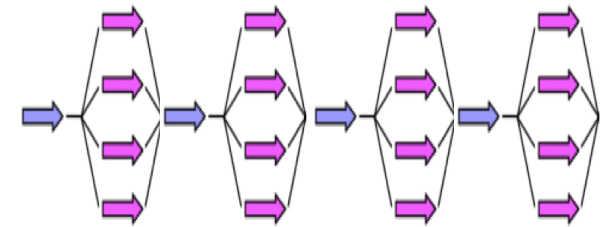
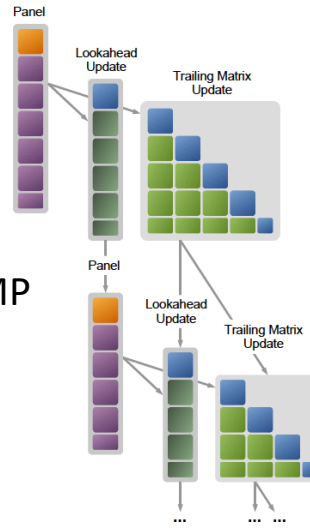


Reorder without
synchronizations



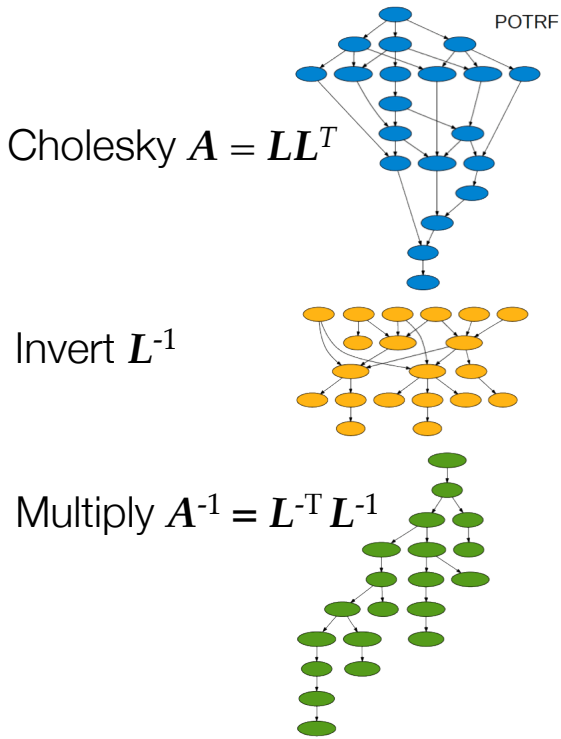
Dataflow Based Design

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

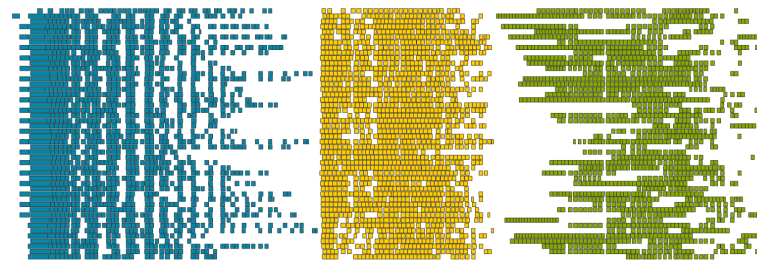


Cholesky; 45% improvement

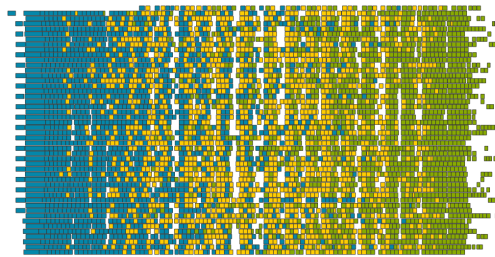
Merging DAGs



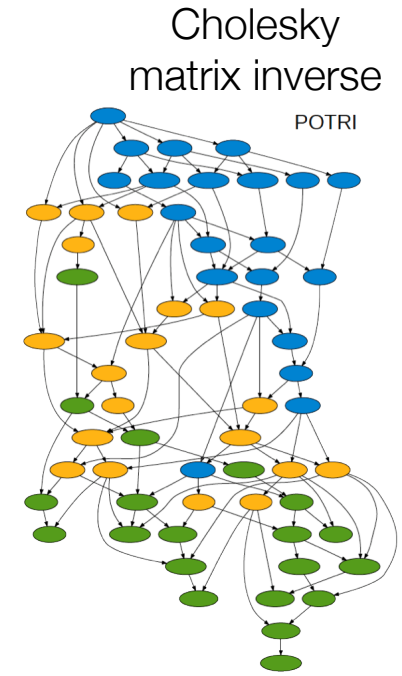
48 cores, matrix is 4000 x 4000, tile size is 200 x 200.



time →



time →



Total: $18(3t+6)$

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

Accelerator platforms

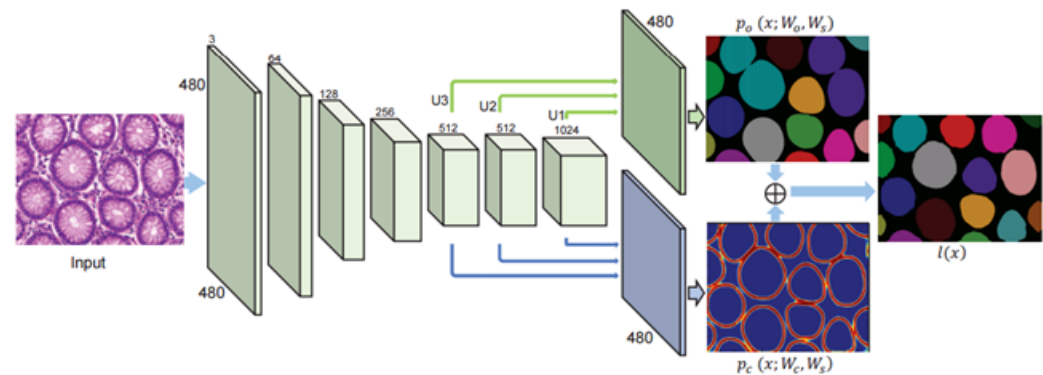
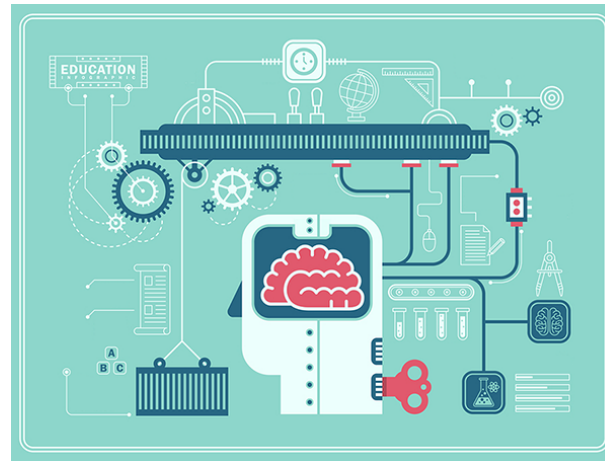
- Initial version with NVIDIA CUDA
- Port to AMD and Intel in progress
- Use BLAS++ as abstraction layer
 - cuBLAS backend (done)
 - hip/rocBLAS backend (done)
 - oneAPI backend (in progress)
- Few CUDA kernels are memory bound (batched add tiles, scale tiles, norms of tiles)
 - Port to HIP using hipify (prototype done)
 - Port to DPC++ in progress
 - Alternatively, port to OpenMP offload



Machine Learning in Computational Science

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

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

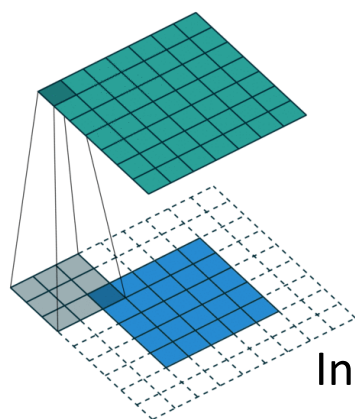
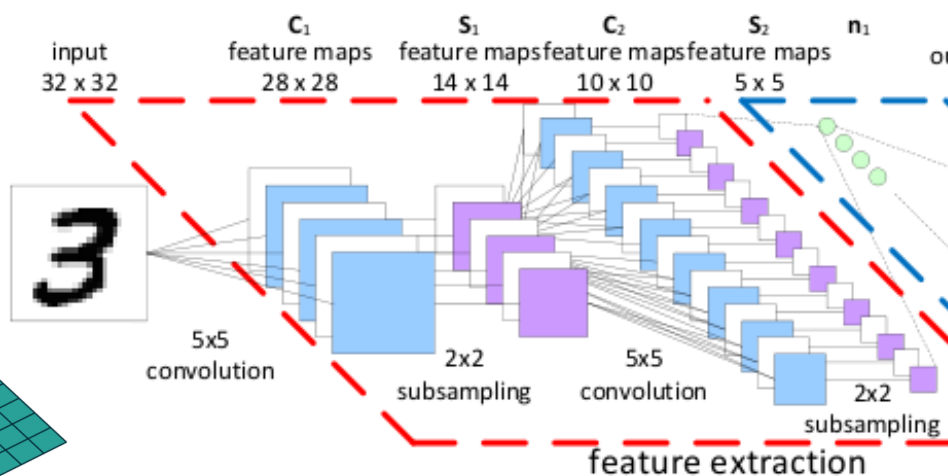


Deep Learning Needs Small Matrix Operations

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 b



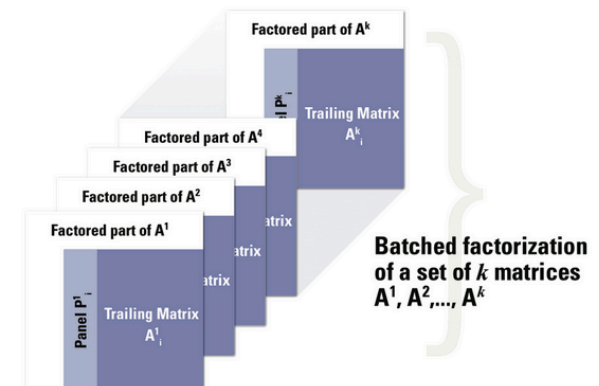
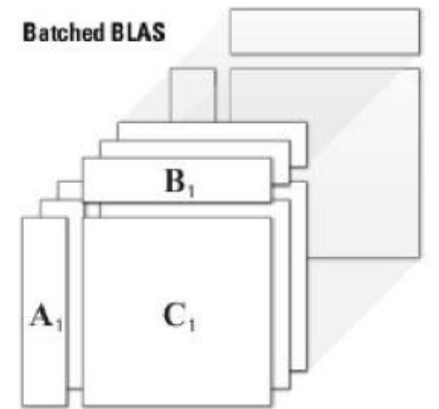
Convolution Step
In this case 3x3 GEMM



Fully Connected
Classification

Standard for Batched Computations

- Define standard API for batched BLAS and LAPACK in collaboration with Intel/Nvidia/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

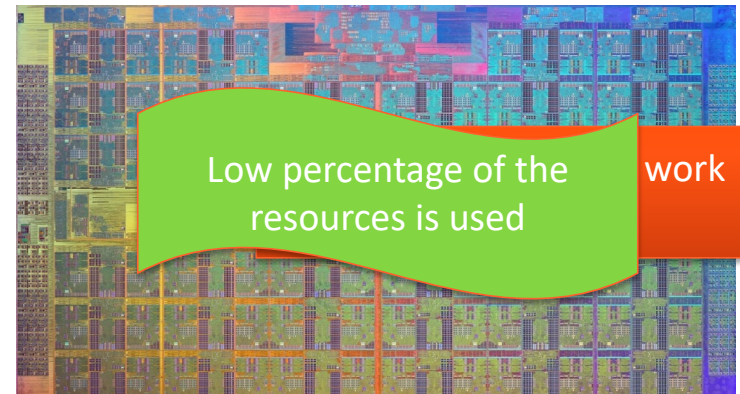
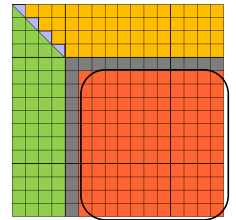
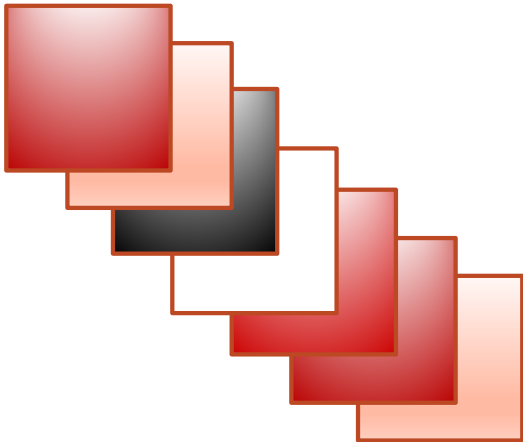


Batched Computations

- **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 (...)
```

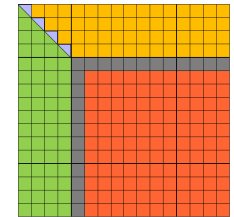


Batched Computations

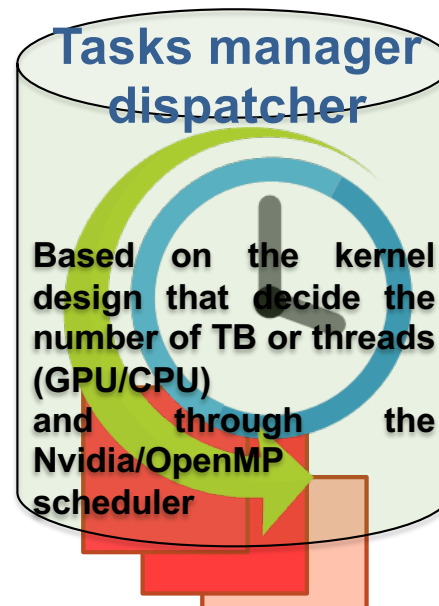
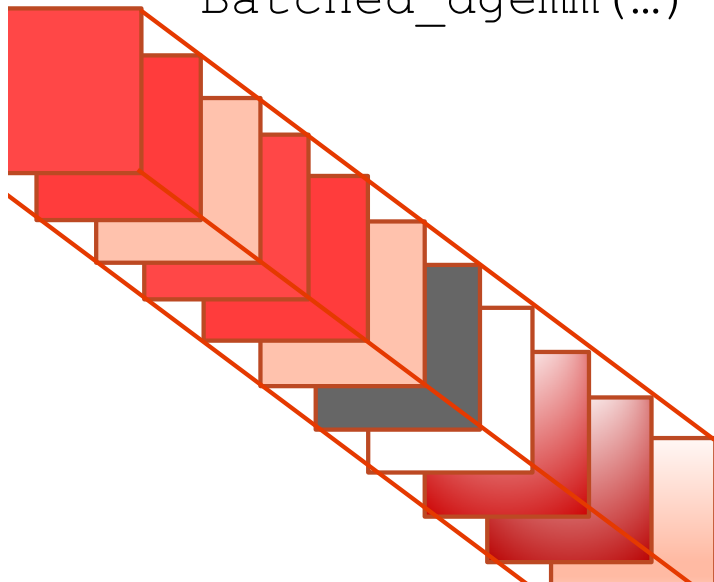
- **Batched computation**

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

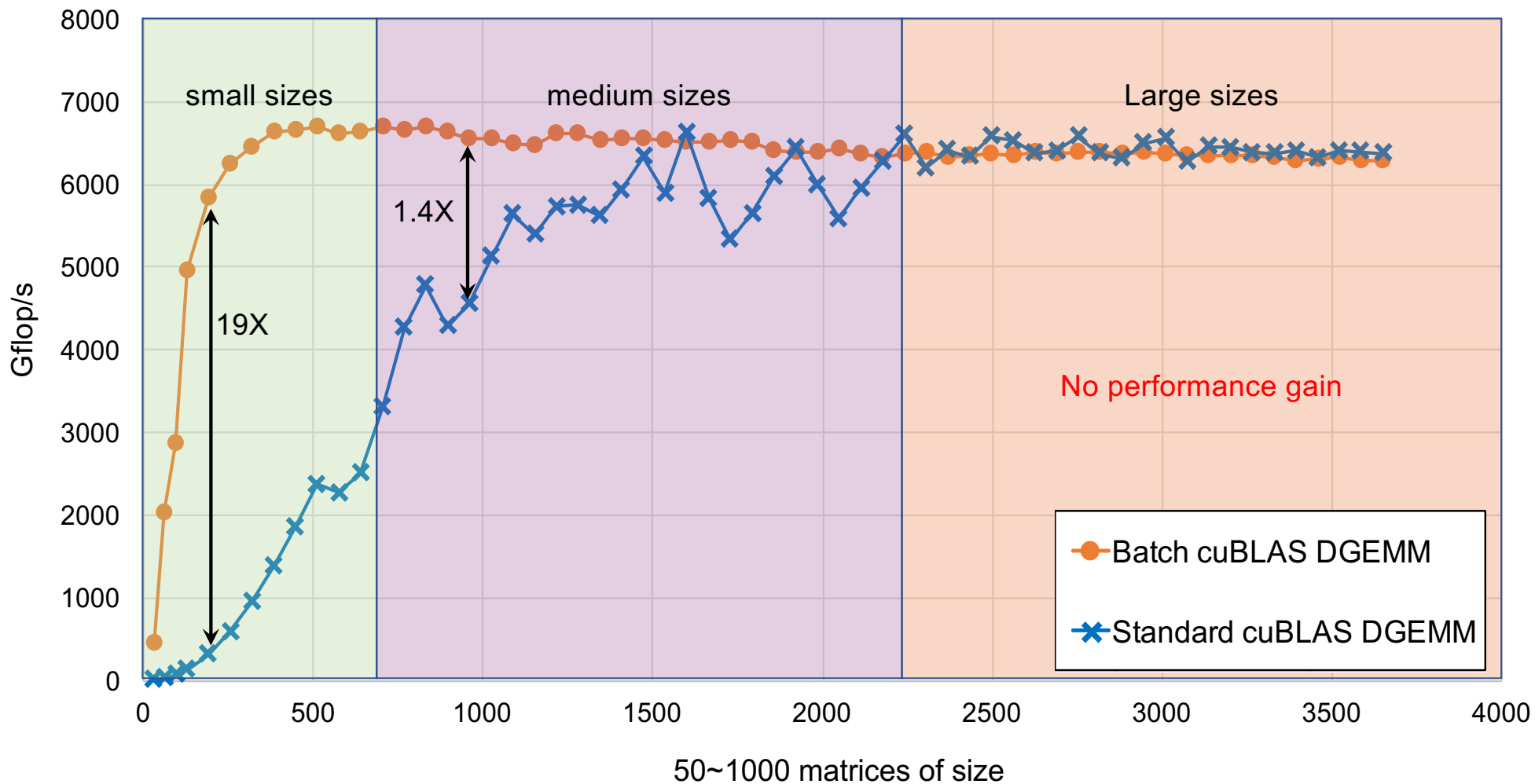
- For very small matrices, assign a matrix/core (CPU) or per TB for GPU
- For medium size a matrix go to a team of cores (CPU) or many TB's (GPU)
- For large size switch to multithreads classical 1 matrix per round.



Batched_dgemm (...)



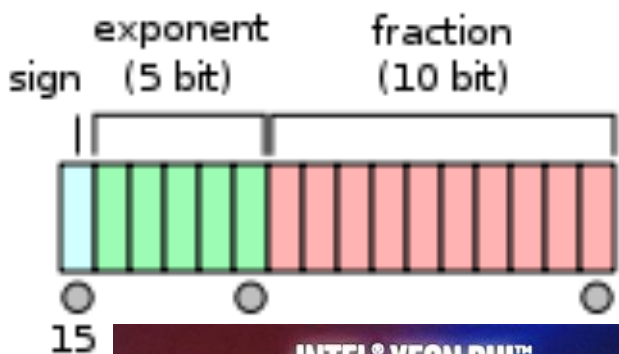
Nvidia V100 GPU





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

A lot of interest driven by "machine learning"

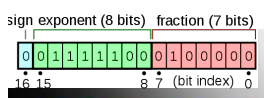


AMD Radeon Instinct			
	Instinct MI6	Instinct MI8	Instinct MI25
Memory Type	16GB GDDR5	4GB HBM	"High Bandwidth Cache and Controller"
Memory Bandwidth	224GB/sec	512GB/sec	?
Single Precision (FP32)	5.7 TFLOPS	8.2 TFLOPS	12.5 TFLOPS
Half Precision (FP16)	5.7 TFLOPS	8.2 TFLOPS	25 TFLOPS
TDP	<150W	<175W	<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



Google Tensor Processing Unit

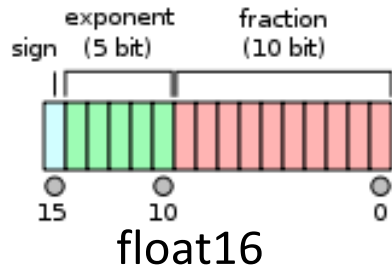
Google TPU different then IEEE bfloat16
 1 bit for the sign,
 8 bits for the exponent (same as SP)
 7 bits for the mantissa

Mixed Precision

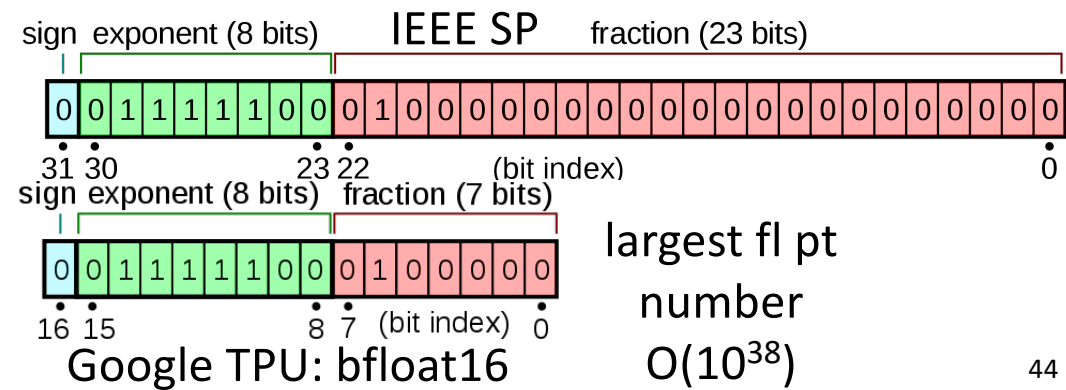
- Today many precisions to deal with (IEEE Standard)

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.)



largest fl pt
number
65,504



largest fl pt
number
 $O(10^{38})$



Nvidia Volta peak rates



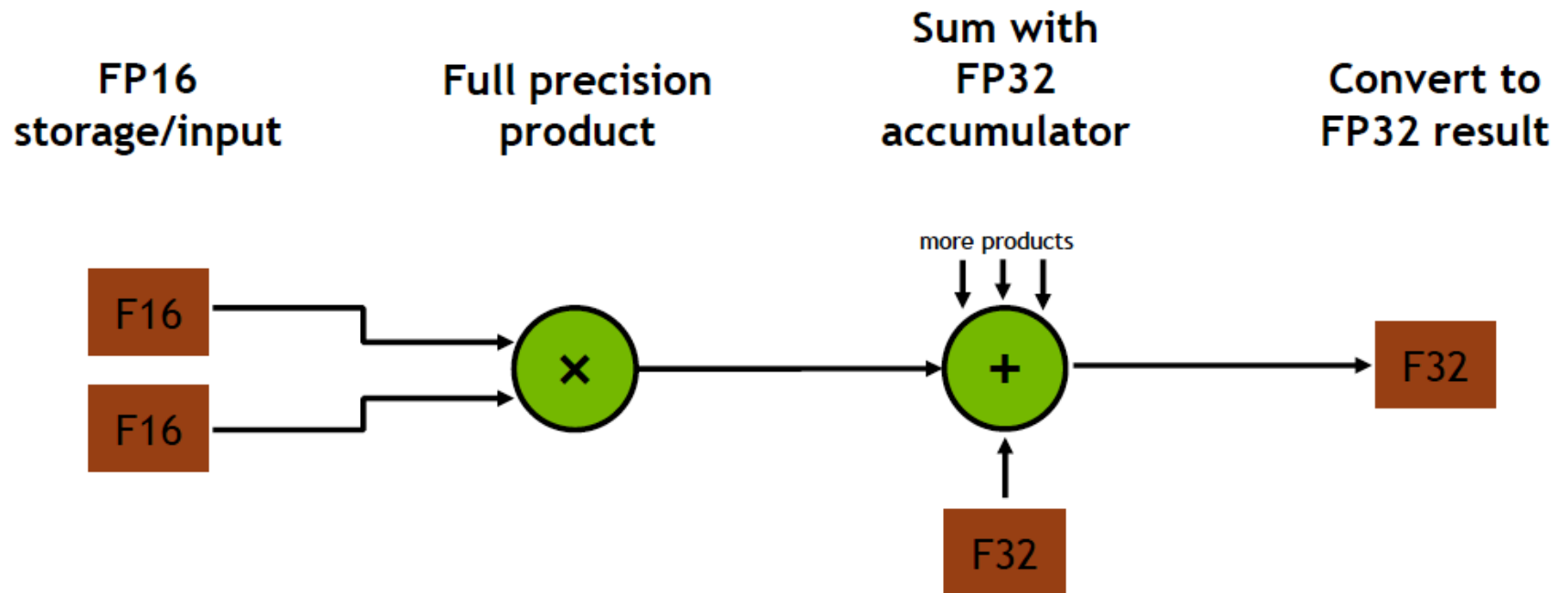
- 64 bit floating point (FMA): 7.5 Tflop/s
- 32 bit floating point (FMA): 15 Tflop/s
- 16 bit floating point (FMA): 30 Tflop/s
- 16 bit floating point with Tensor core: 120 Tflop/s

Mixed Precision Matrix Multiply 4x4 Matrices

$$D = \begin{matrix} \text{FP16 or FP32} & \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} & \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} & \text{FP16} & + & \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix} & \text{FP16 or FP32} \end{matrix}$$

$$D = AB + C$$

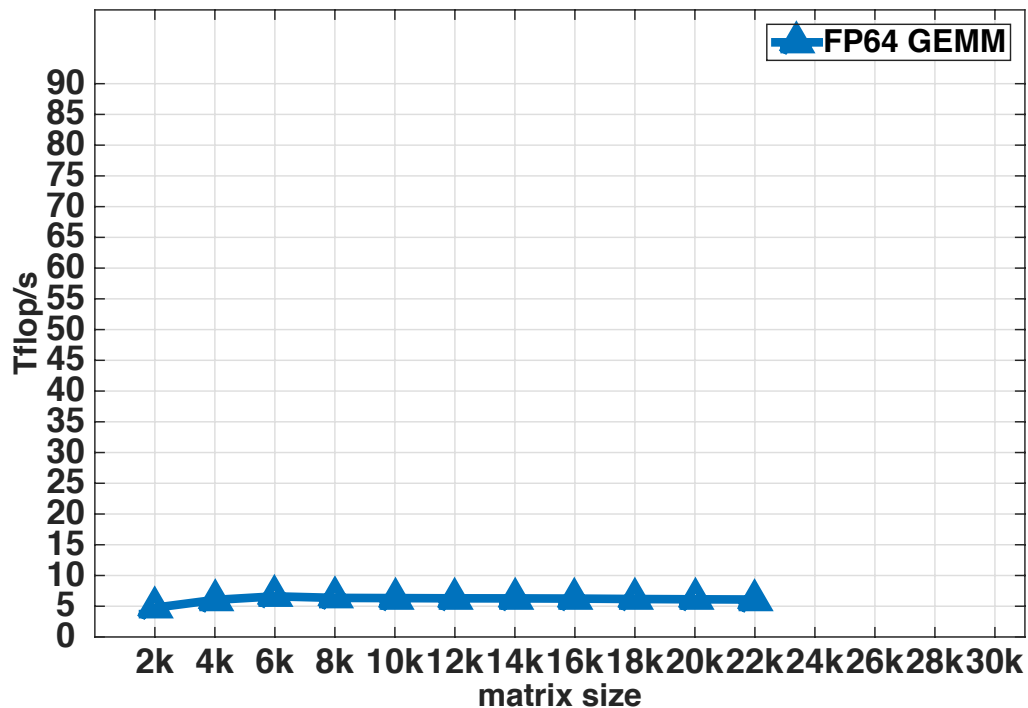
VOLTA TENSOR OPERATION



Also supports FP16 accumulator mode for inferencing

Leveraging Half Precision in HPC on V100

Study of the Matrix Matrix multiplication kernel on Nvidia V100



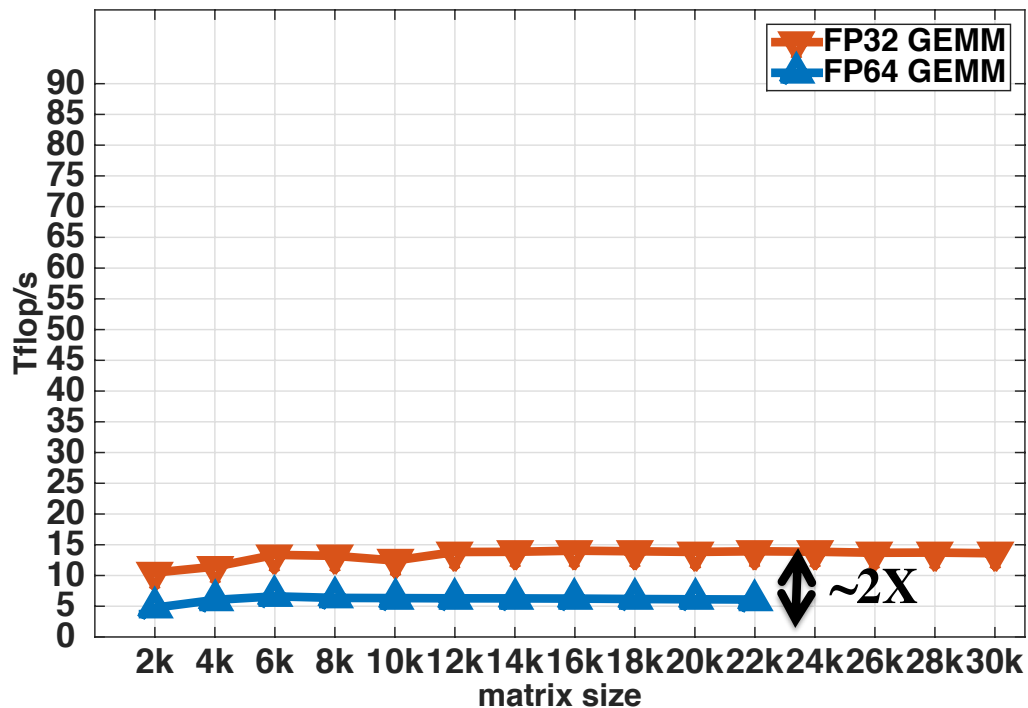
- dgemm achieve about 6.4 Tflop/s

Matrix matrix multiplication GEMM

$$C = \alpha A B + \beta C$$

Leveraging Half Precision in HPC on V100

Study of the Matrix Matrix multiplication kernel on Nvidia V100



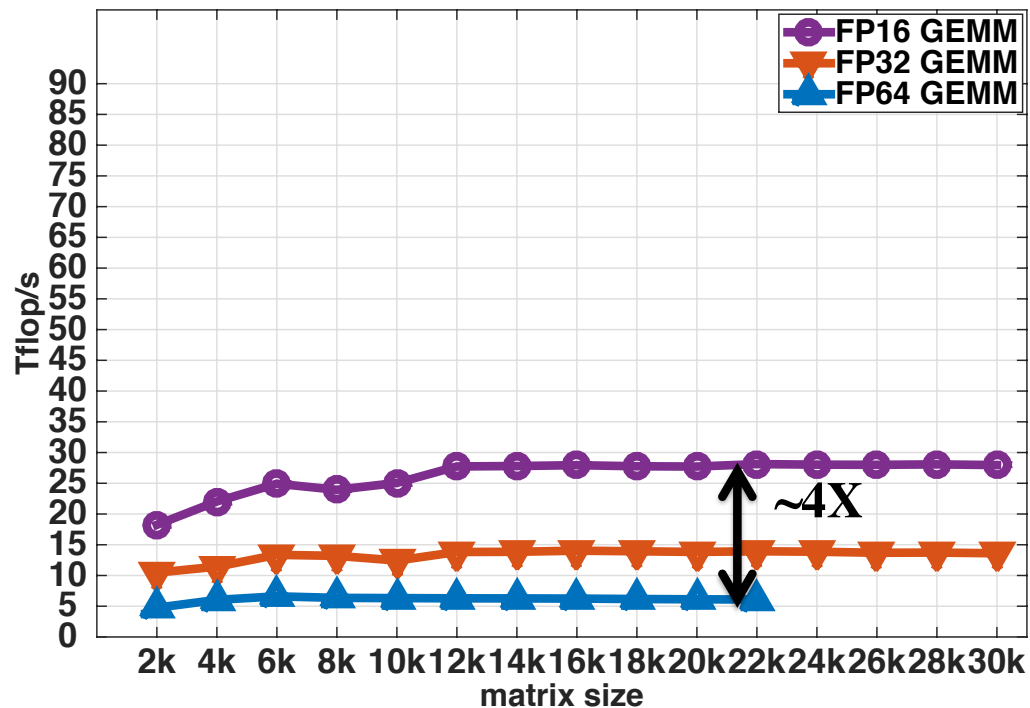
- dgemm achieve about 6.4 Tflop/s
- sgemm achieve about 14 Tflop/s

Matrix matrix multiplication GEMM

$$C = \alpha A B + \beta C$$

Leveraging Half Precision in HPC on V100

Study of the Matrix Matrix multiplication kernel on Nvidia V100



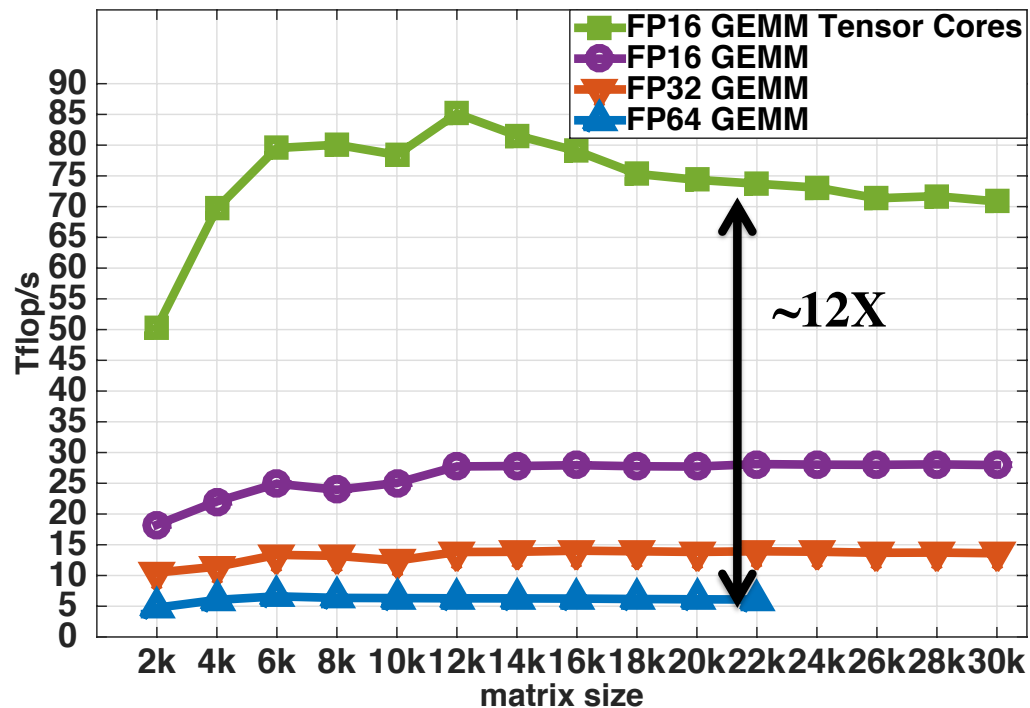
dgemm achieve about 6.4 Tfllop/s
sgemm achieve about 14 Tfllop/s
hgemm achieve about 27 Tfllop/s

Matrix matrix multiplication GEMM

$$C = \alpha A B + \beta C$$

Leveraging Half Precision in HPC on V100

Study of the Matrix Matrix multiplication kernel on Nvidia V100



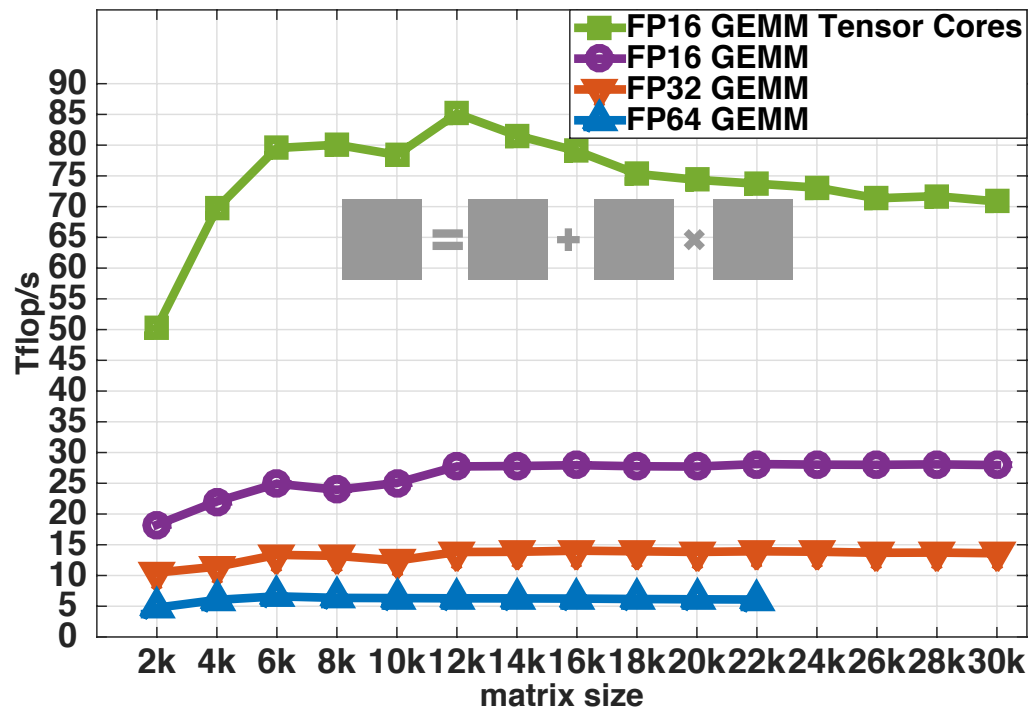
dgemm achieve about 6.4 Tflop/s
sgemm achieve about 14 Tflop/s
hgemm achieve about 27 Tflop/s
Tensor cores gemm reach about 85 Tflop/s

Matrix matrix multiplication GEMM

$$C = \alpha A B + \beta C$$

Leveraging Half Precision in HPC on V100

Study of the Matrix Matrix multiplication kernel on Nvidia V100



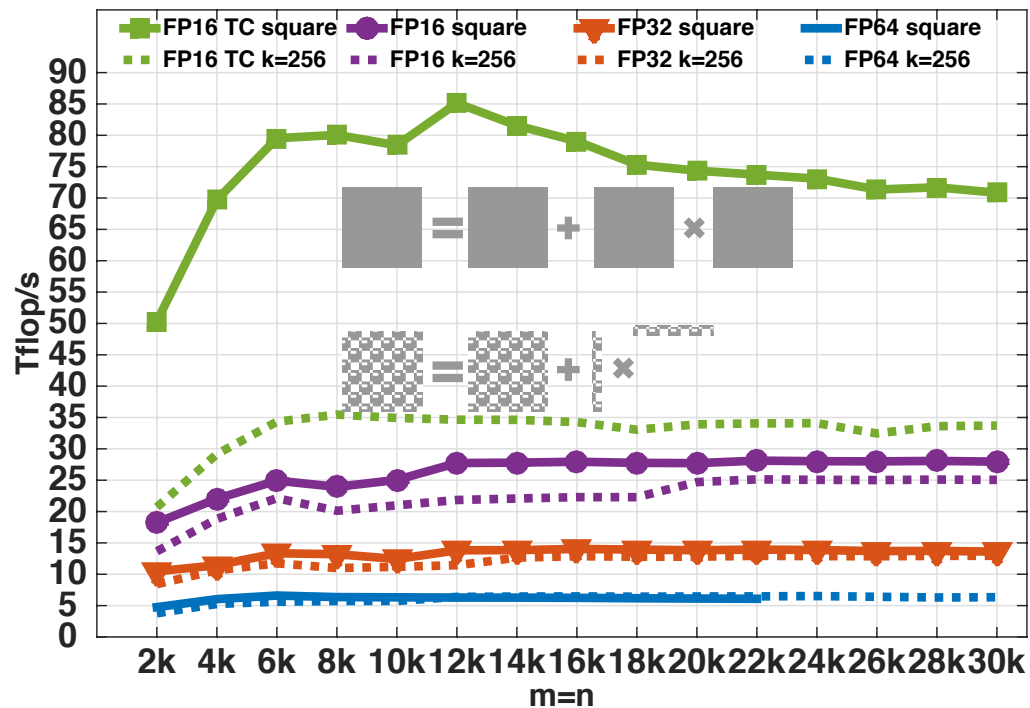
dgemm achieve about 6.4 Tflop/s
sgemm achieve about 14 Tflop/s
hgemm achieve about 27 Tflop/s
Tensor cores gemm reach about 85 Tflop/s

Matrix matrix multiplication GEMM

$$C = \alpha A B + \beta C$$

Leveraging Half Precision in HPC on V100

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

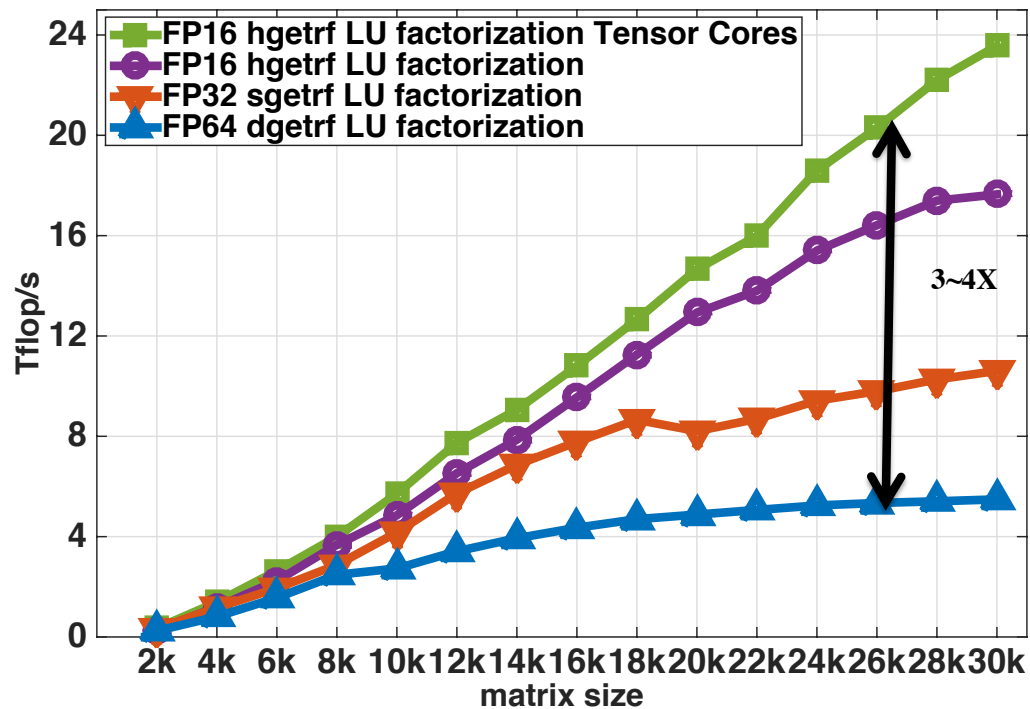


- In LU factorization need matrix multiple but operations is a rank-k update computing the Schur complement

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} = \begin{bmatrix} A & B \\ C & D \end{bmatrix} + \begin{bmatrix} I & 0 \\ 0 & -S \end{bmatrix} \times \begin{bmatrix} 0 & B \\ C & D \end{bmatrix}$$

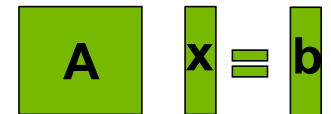
Leveraging Half Precision in HPC on V100

Study of the LU factorization algorithm on Nvidia V100



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

$$A x = b$$



$$LUx = b$$



$$Ly = b$$



then

$$Ux = y$$



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)

$x = U \setminus (L \setminus b)$

$r = b - Ax$

lower precision

$O(n^3)$

lower precision

$O(n^2)$

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:

➤ $z = U \setminus (L \setminus r)$

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

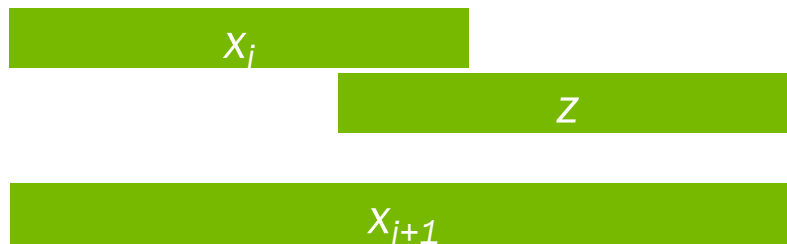
Higham and Carson showed can solve the inner problem with iterative method and not infect the solution.

E. Carson & N. Higham, "Accelerating the Solution of Linear Systems by Iterative Refinement in Three Precisions *SIAM J. Sci. Comput.*, 40(2), A817–A847.

- 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

Improving Solution

- z is the correction or $(x_{i+1} - x_i)$
- Computed in lower precision and then added to the approximate solution in higher precision $x_i + z$



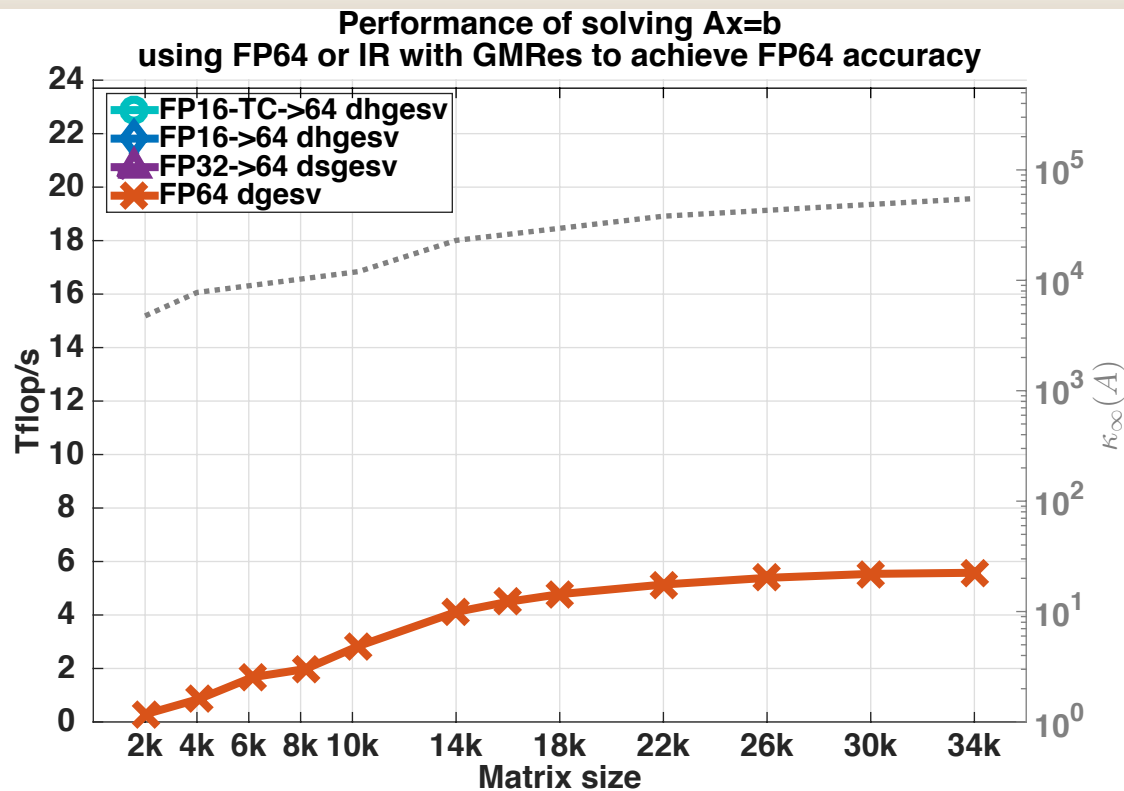
- Can be used in situations like this, i.e.

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

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

Leveraging Half Precision in HPC on V100

Performance Behavior



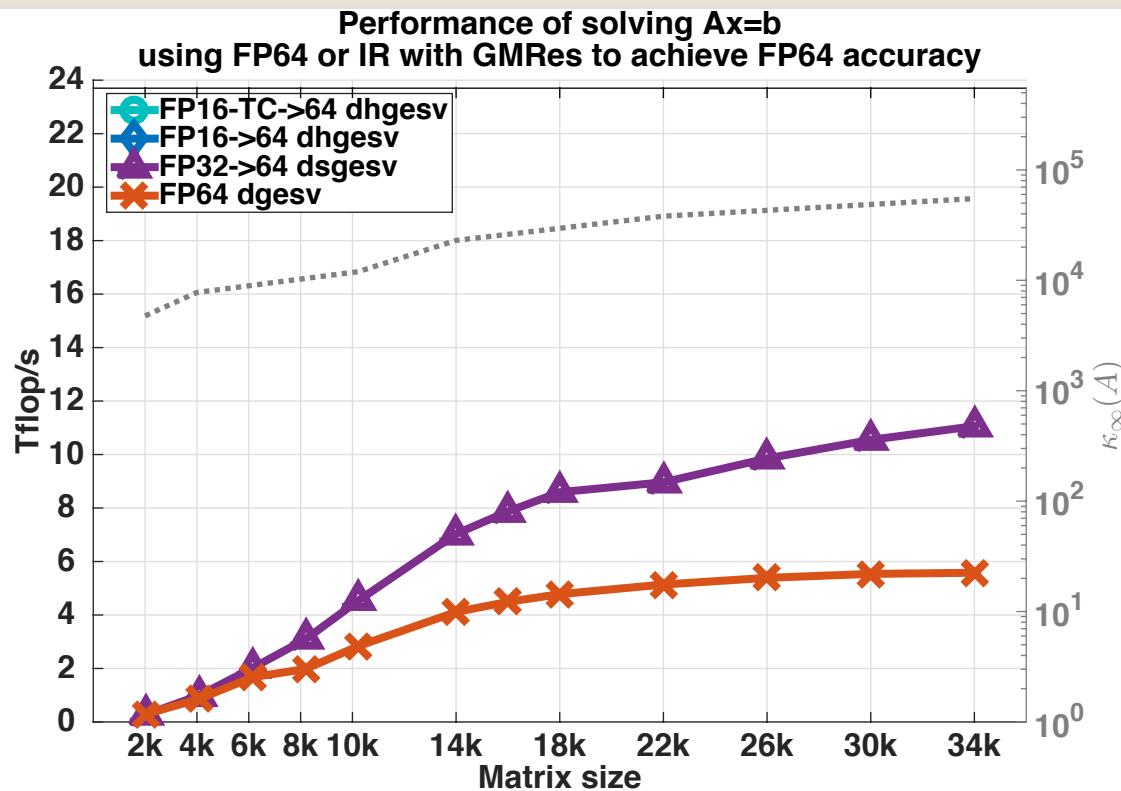
Flops = $2n^3/(3 \text{ time})$
meaning twice higher is twice faster

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

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

Leveraging Half Precision in HPC on V100

Performance Behavior



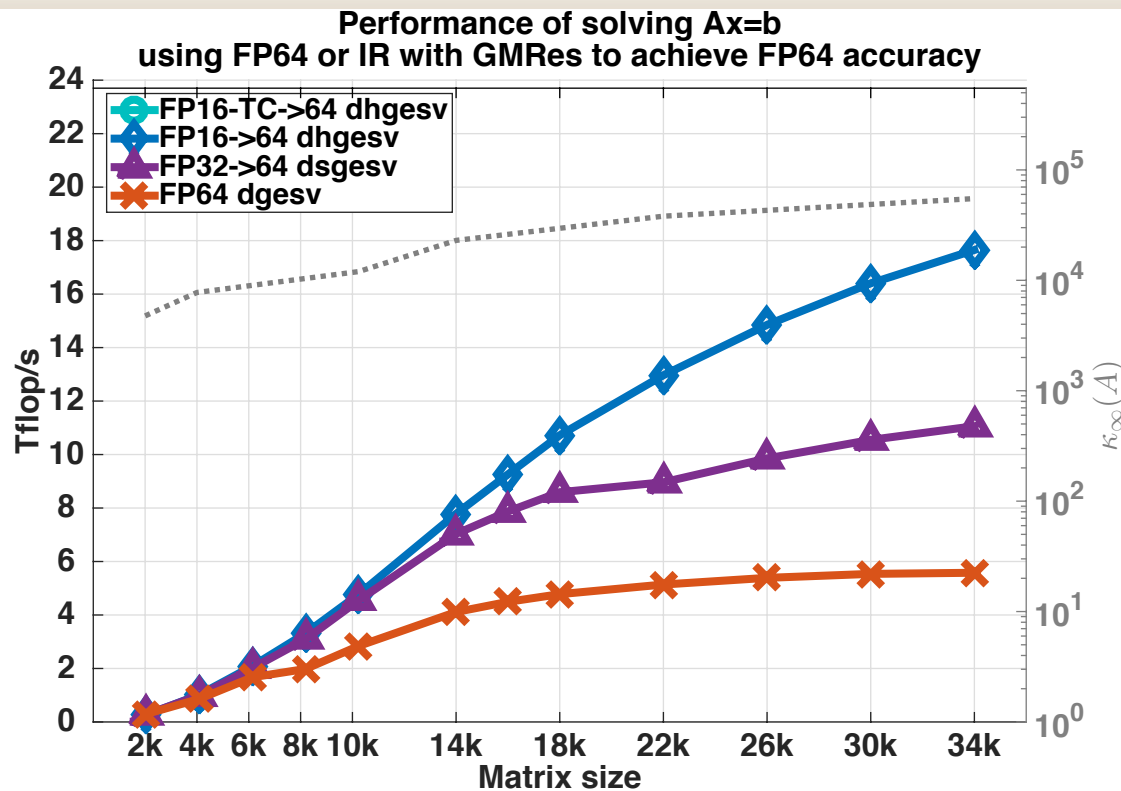
Flops = $2n^3/(3 \text{ time})$
meaning twice higher is twice faster

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

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

Leveraging Half Precision in HPC on V100

Performance Behavior



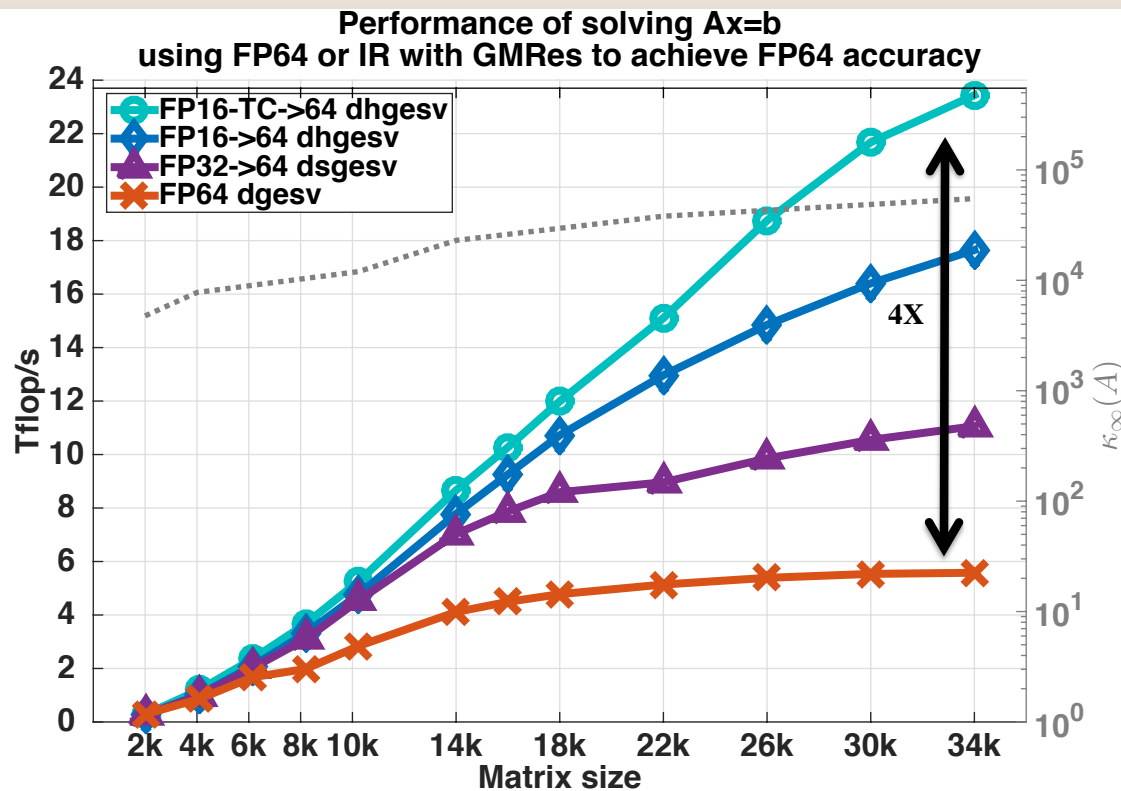
Flops = $2n^3/(3 \text{ time})$
meaning twice higher is twice faster

- 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

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

Leveraging Half Precision in HPC on V100

Performance Behavior



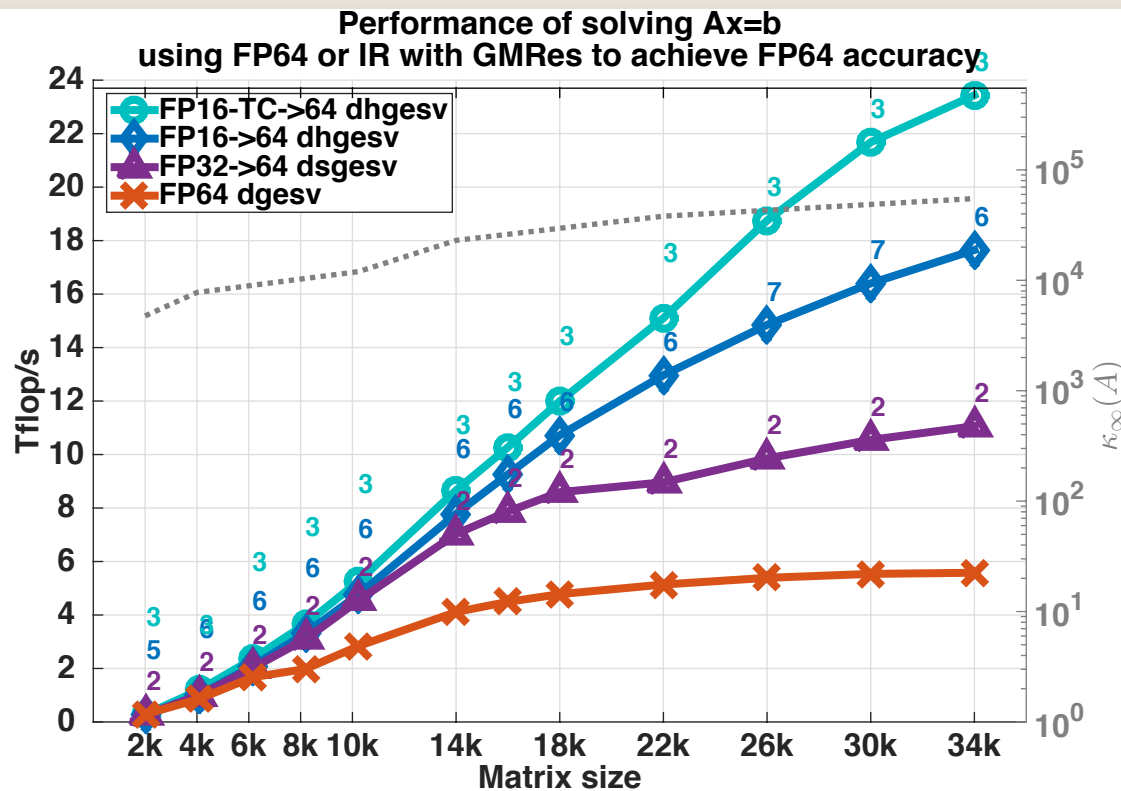
Flops = $2n^3/(3 \text{ time})$
 meaning twice higher is twice faster

- 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

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

Leveraging Half Precision in HPC on V100

Performance Behavior



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

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

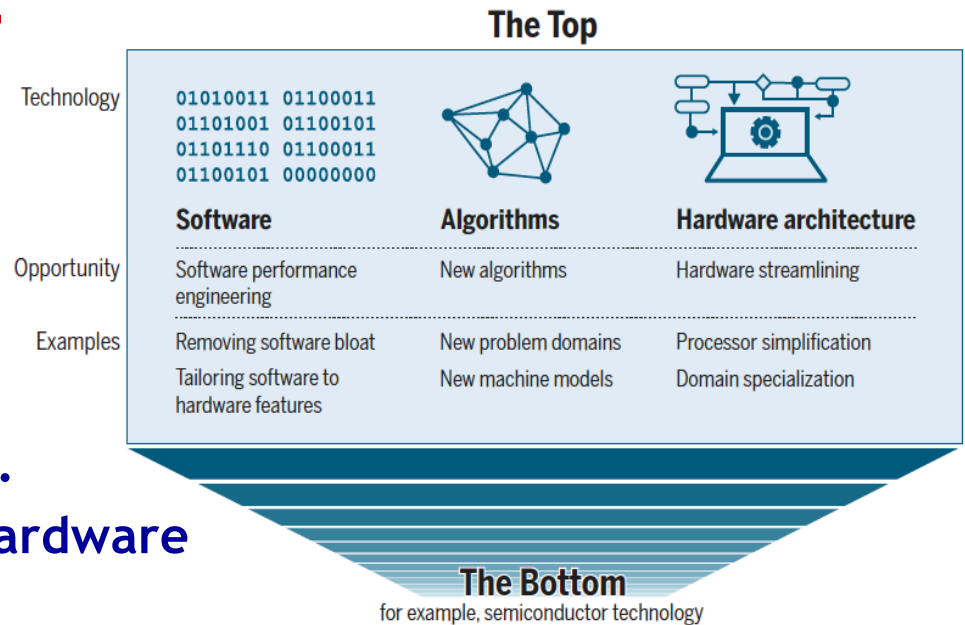


Critical Issues at Exascale for Algorithm and Software Design

- **Synchronization-reducing algorithms**
 - Break Fork-Join model
- **Communication-reducing algorithms**
 - Use methods which have lower bound on communication
- **Mixed precision methods (half (16bit), single(32 bit), & double precision (64))**
 - 2x - 10x speed of ops and 2x - 4x speed for data movement
- **Autotuning - Performance Debugging**
 - Today's machines are very 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.

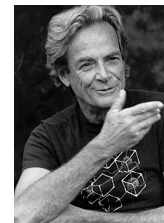
The Take Away

- **HPC Constantly Changing**
 - **Scalar**
 - **Vector**
 - **Distributed**
 - **Accelerated**
 - **Mixed precision**
- **Data movement critical for performance.**
- **Algorithm / Software advances follows hardware**
 - **And there is “plenty of room at the top”**
 - **“There's life in the old dog yet”**



Leiserson *et al.*, *Science* **368**, 1079 (2020) 5 June 2020

“There's plenty of room at the Top: What will drive computer performance after Moore's law?”



Collaborators / Software / Support

- ◆ **PLASMA**

<http://icl.cs.utk.edu/plasma/>



- ◆ **MAGMA**

<http://icl.cs.utk.edu/magma/>



- ◆ **SLATE**

- ◆ <https://icl.utk.edu/slate/>

- ◆ <https://bitbucket.org/icl/slate/src/default/>



- ◆ **PaRSEC** (Parallel Runtime Scheduling & Execution Control)

- ◆ <http://icl.cs.utk.edu/parsec/>



Also see: <http://www.netlib.org/utk/people/JackDongarra/papers.htm>

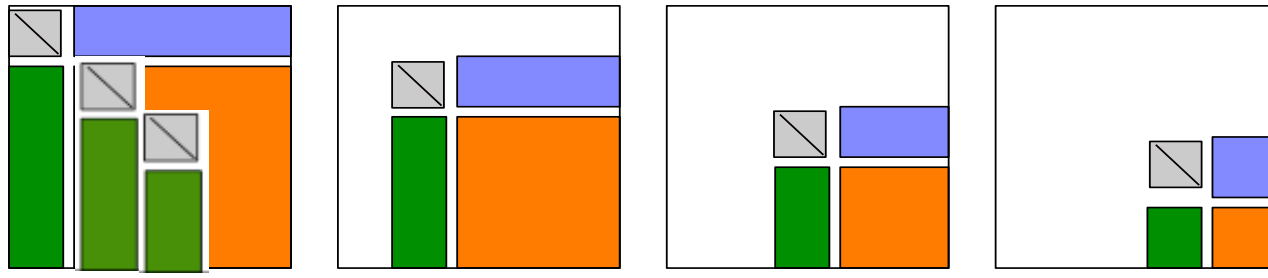
Looking for Grad Students and
Post-Docs for work in this area.



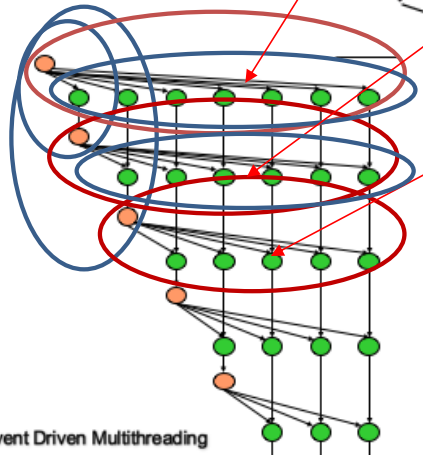
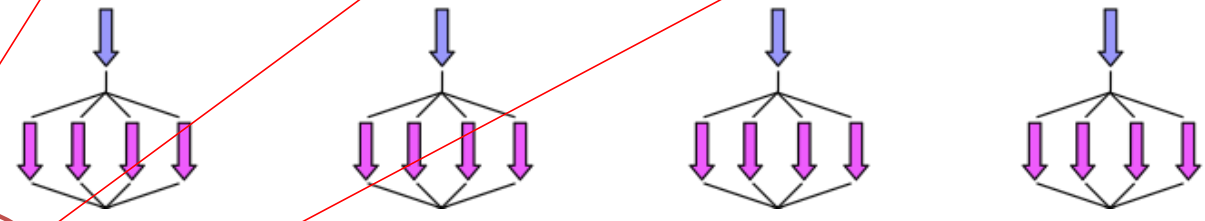
- ◆ Collaborating partners
University of Tennessee, Knoxville
University of California, Berkeley
University of Colorado, Denver



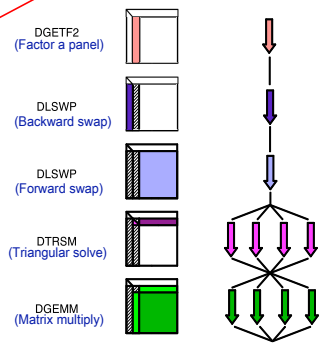
Synchronization (in LAPACK)



Step 1 → Step 2 → Step 3 → Step 4 ...

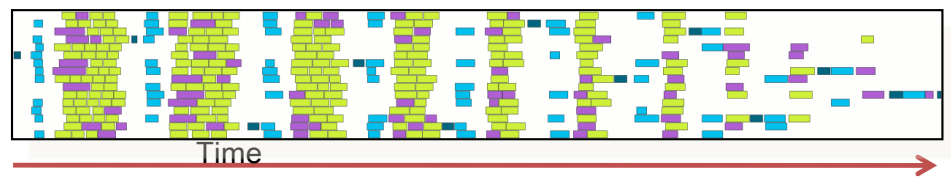


Event Driven Multithreading



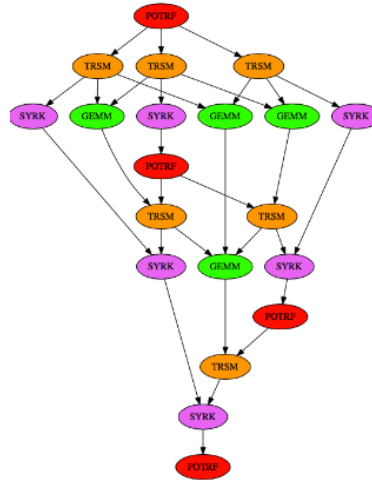
LAPACK
LAPACK
LAPACK
BLAS
BLAS 65

➤ fork join
➤ bulk synchronous processing

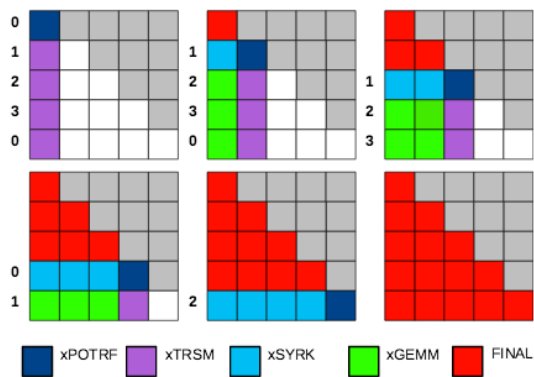


OpenMP tasking

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



Tiled Cholesky Decomposition

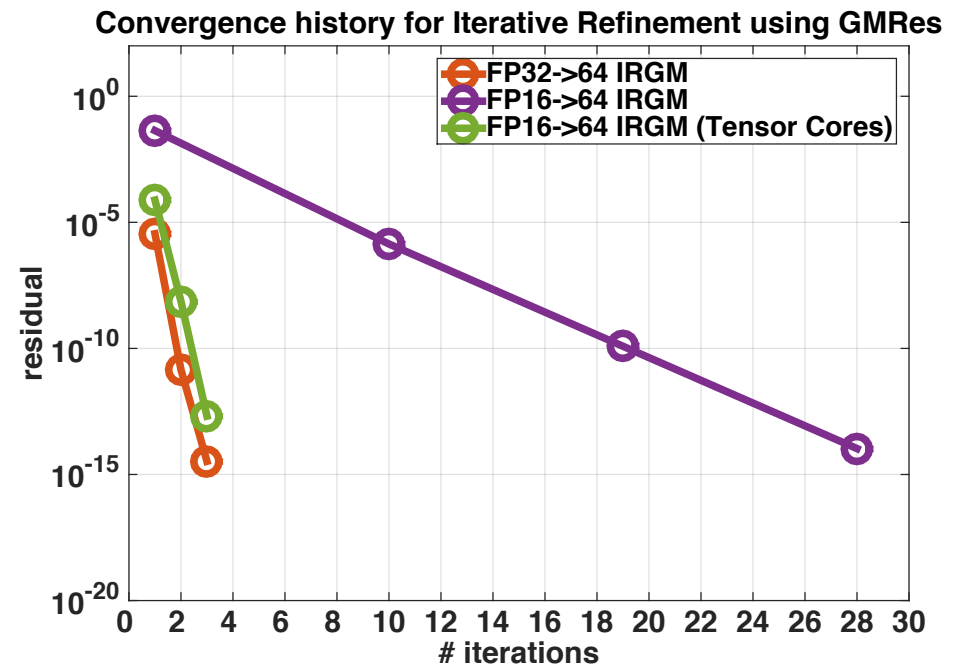
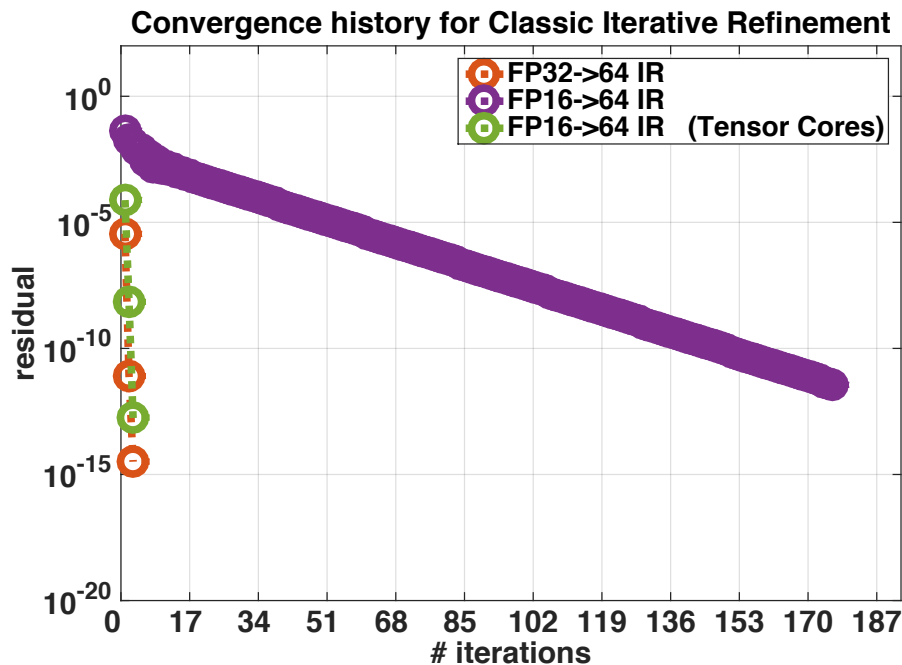


```

#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) ); }
      }
    }
  }
}

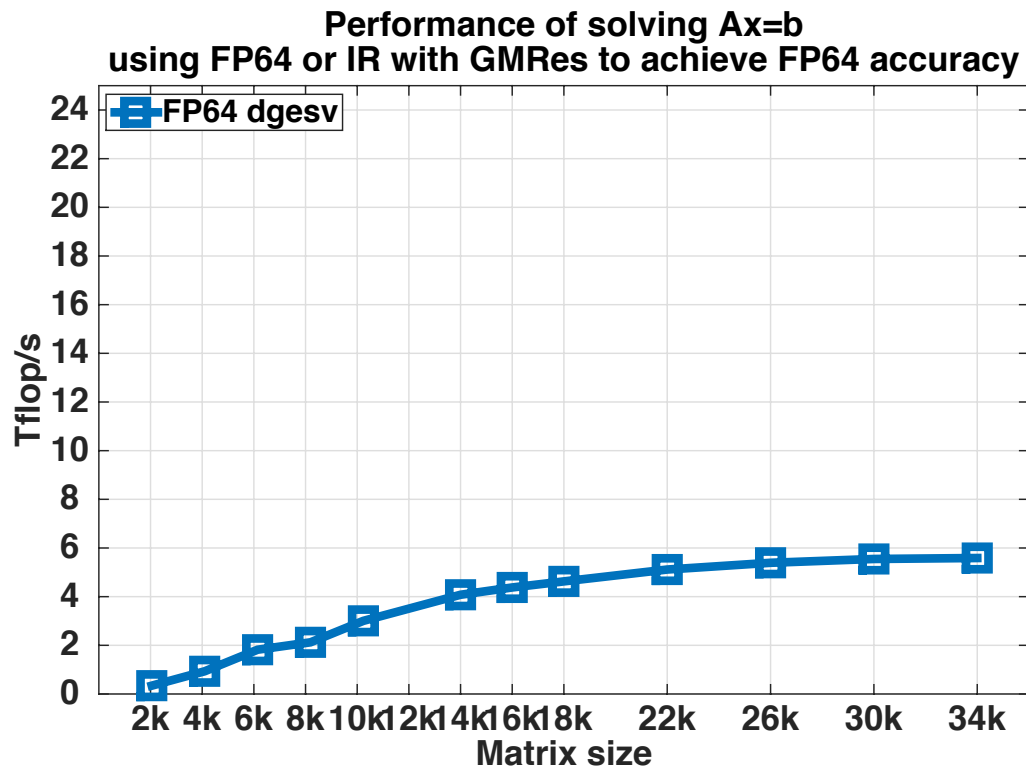
```

Leveraging Half Precision in HPC on V100



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

Leveraging Half Precision in HPC on V100



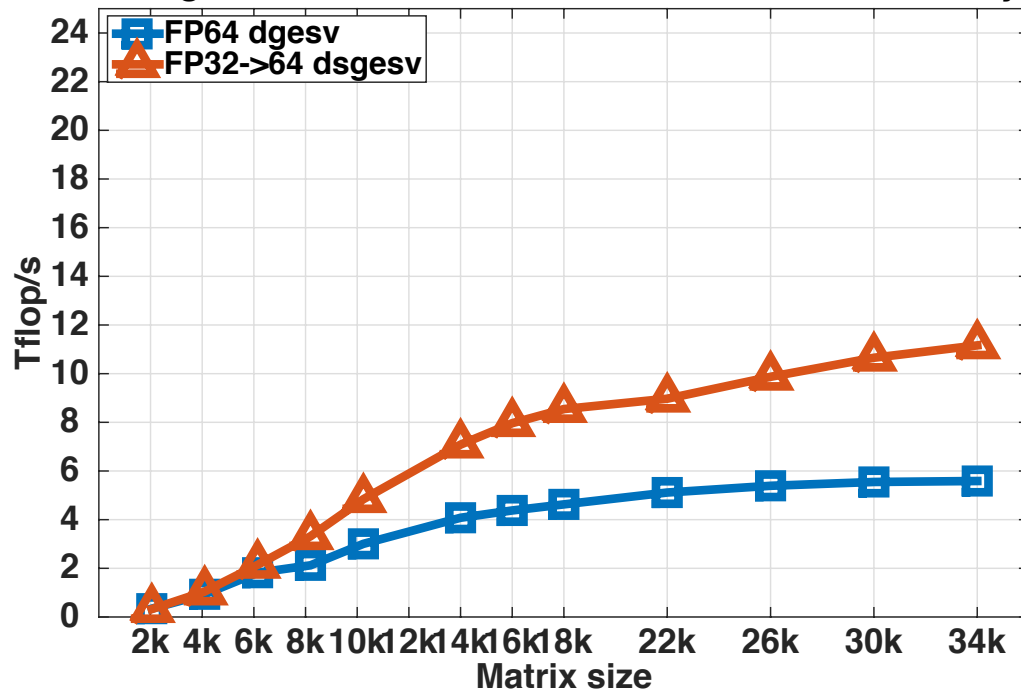
Flops = $2n^3 / (3 \text{ time})$
meaning twice higher is twice faster

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

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

Leveraging Half Precision in HPC on V100

Performance of solving $Ax=b$
using FP64 or IR with GMRes to achieve FP64 accuracy

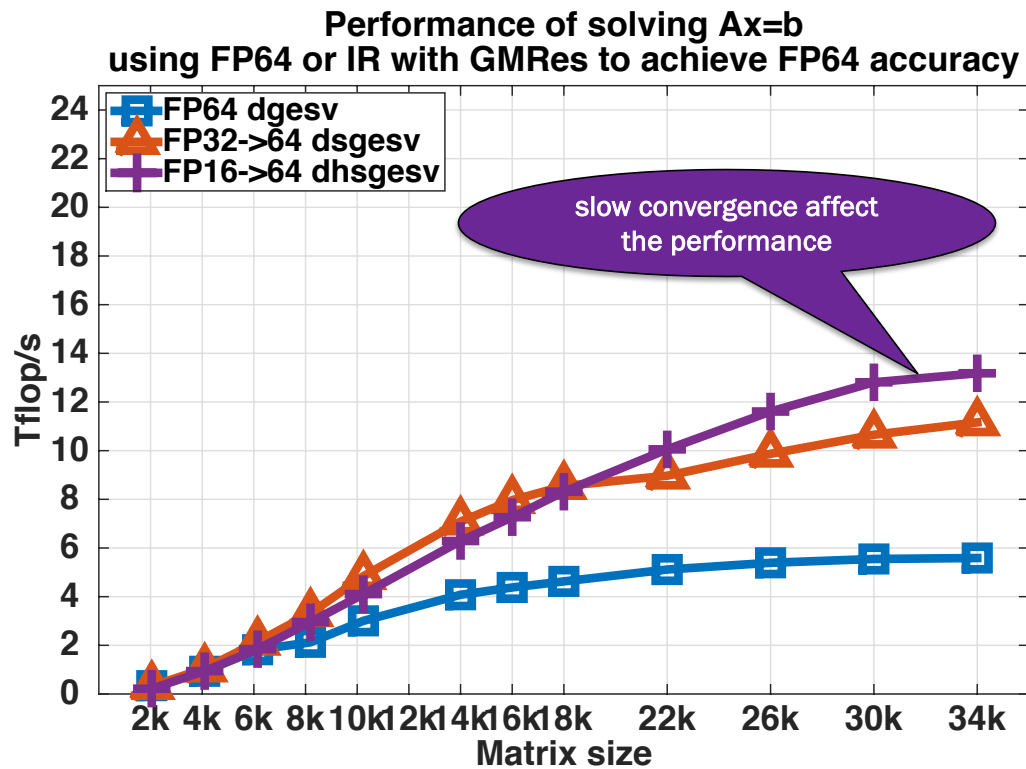


Flops = $2n^3 / (3 \text{ time})$
meaning twice higher is twice faster

- 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 $\sigma_i = (1, \dots, 1, \frac{1}{cond})$ and where its condition number is equal to 10^2 .

Leveraging Half Precision in HPC on V100

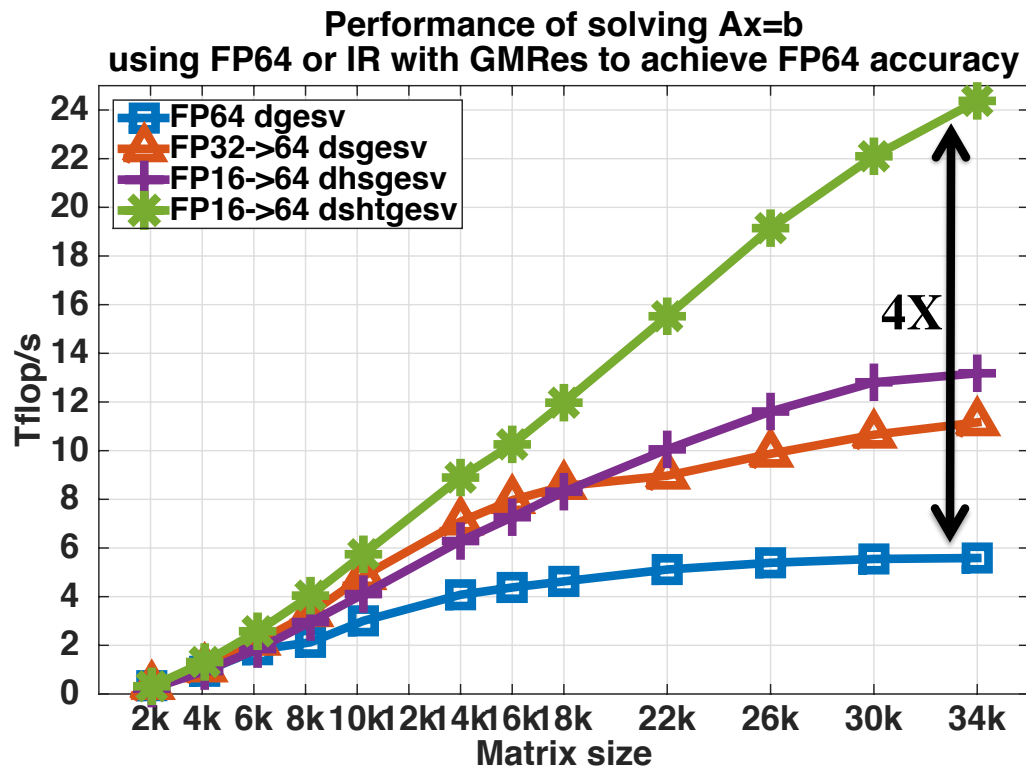


Flops = $2n^3 / (3 \text{ time})$
meaning twice higher is twice faster

- 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 $\sigma_i = (1, \dots, 1, \frac{1}{cond})$ and where its condition number is equal to 10^2 .

Leveraging Half Precision in HPC on V100



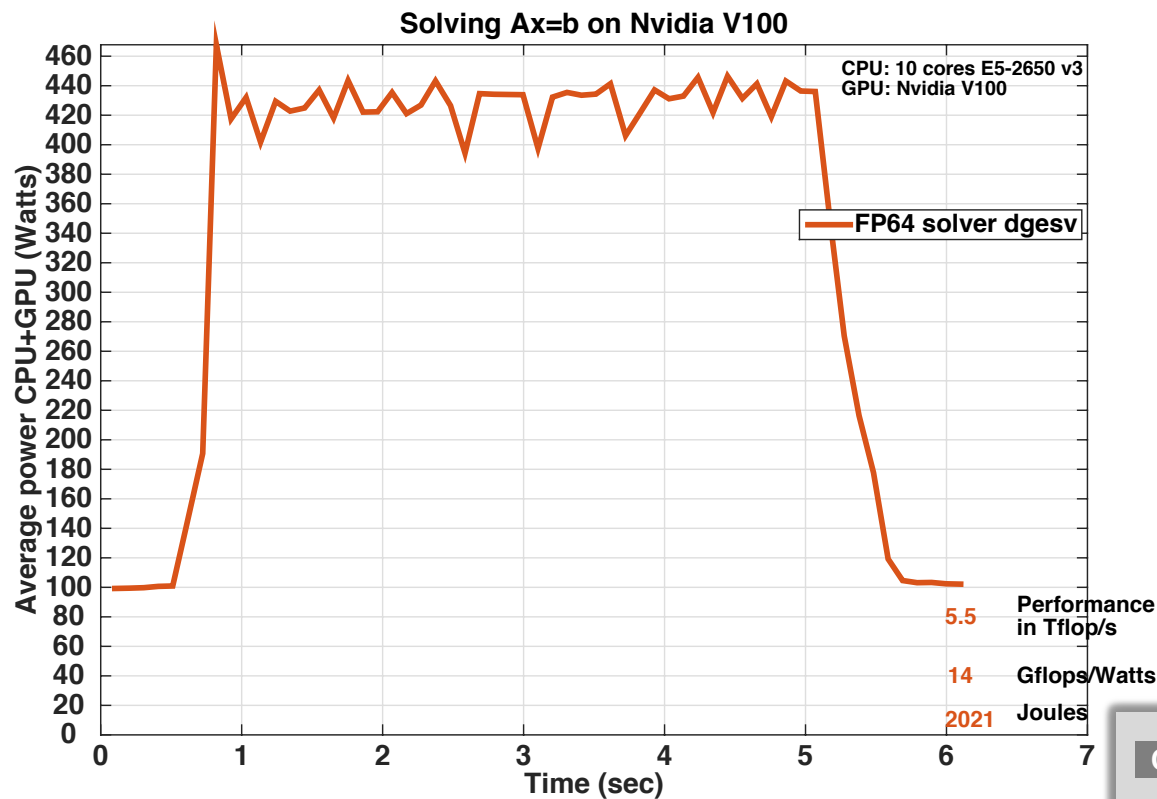
Flops = $2n^3 / (3 \text{ time})$
meaning twice higher is twice faster

- 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 $\sigma_i = (1, \dots, 1, \frac{1}{cond})$ 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 Tflops/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

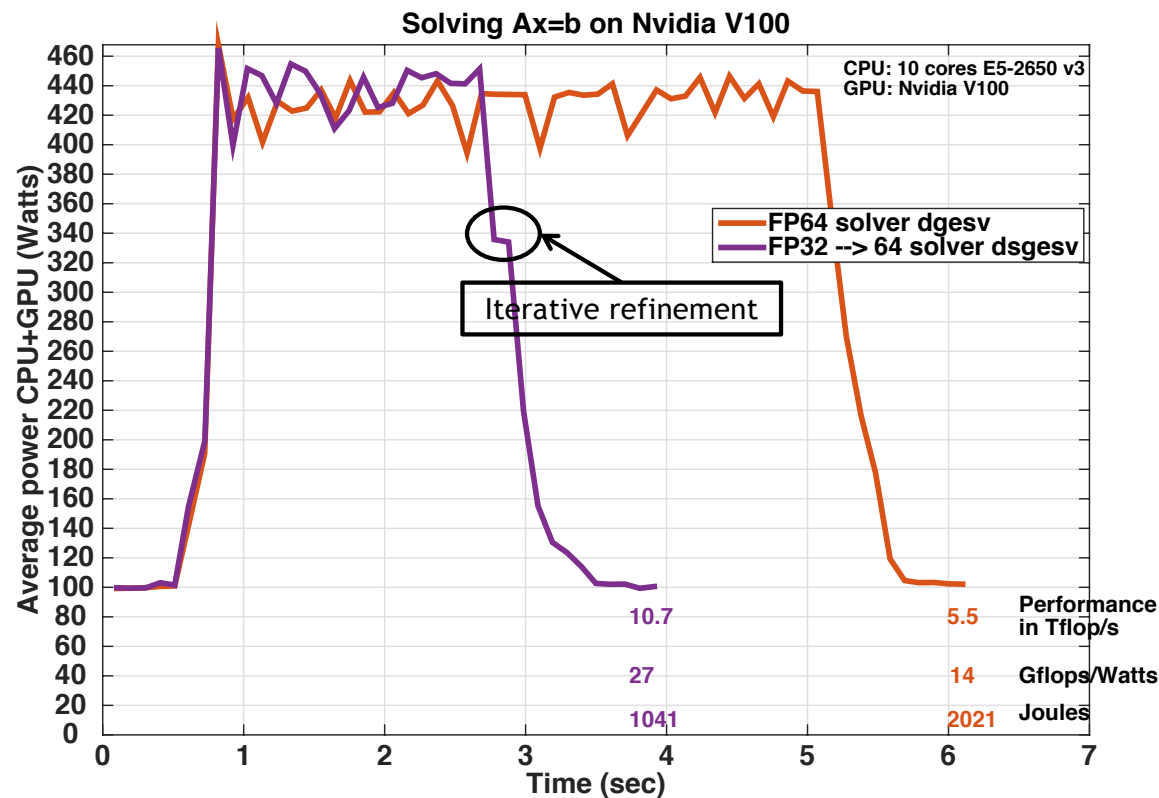
V100 NVIDIA Volta GPU
80 MP x 64 @ 1.38 GHz

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

Leveraging Half Precision in HPC

Power awareness

Mixed precision techniques can provide a large gain in energy efficiency



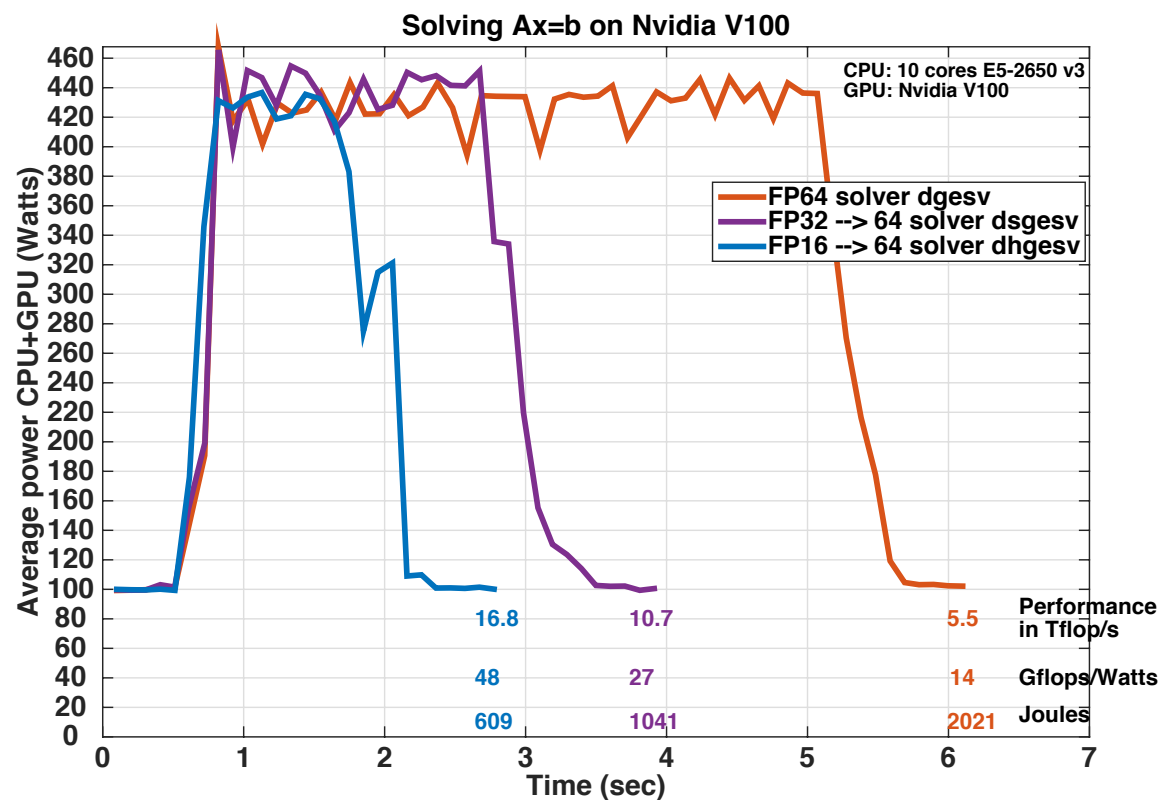
- Power consumption of the **FP64 algorithm** to solve $Ax=b$ for a matrix of size 34K, it achieve **5.5 Tflop/s** and requires about **2021 joules** providing about **14 Gflops/Watts**.
- Power consumption of the mixed precision **FP32→64 algorithm** to solve $Ax=b$ for a matrix of size 34K, it achieve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

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

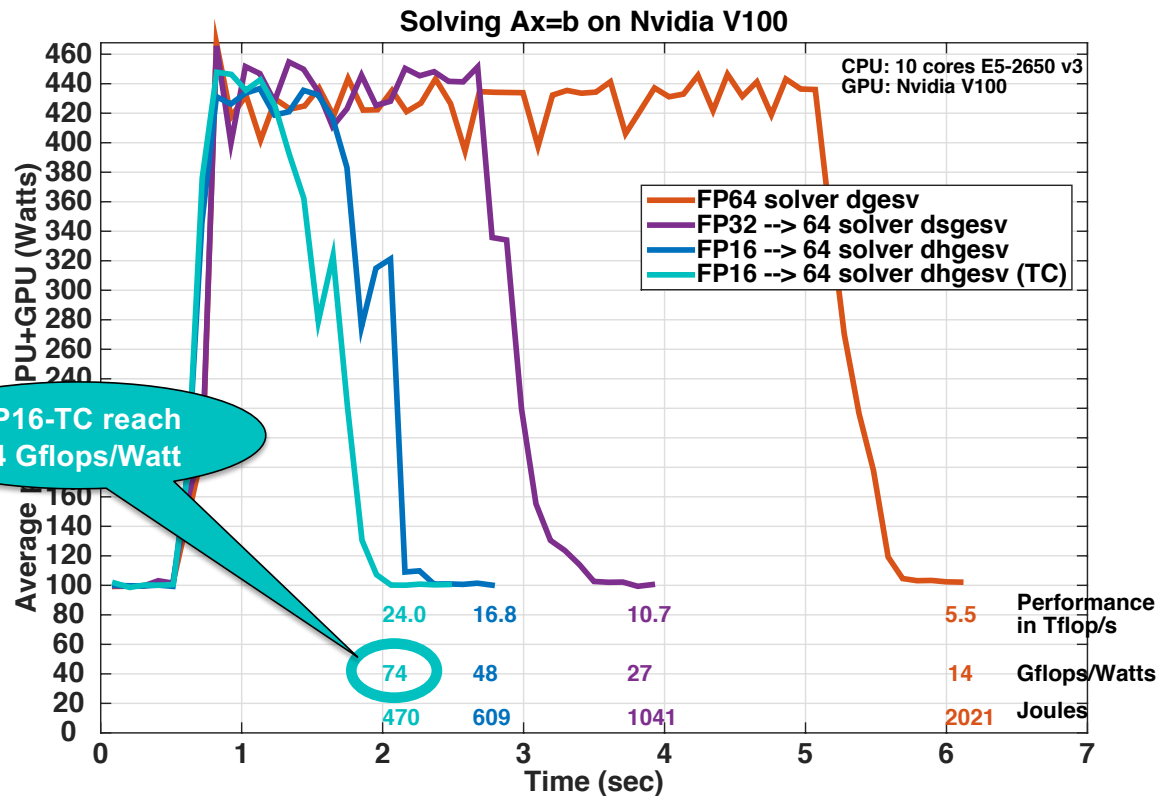
Problem generated with an arithmetic distribution of the singular values $\sigma_i = 1 - \left(\frac{i-1}{n-1}\right)\left(1 - \frac{1}{cond}\right)$ and positive eigenvalues.

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**.



FP16-TC reach
74 Gflops/Watt

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

- ◆ SLATE

<http://icl.cs.utk.edu/slate/>

- ◆ PLASMA

<http://icl.cs.utk.edu/plasma/>

- ◆ MAGMA

<http://icl.cs.utk.edu/magma/>

- ◆ 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



ICL is hiring!

Projects include

- SLATE — distributed dense linear algebra
- CEED — tensor algebra, batched operations
- PEEKS — Krylov methods
- heFFTe — distributed FFT
- PAPI — performance measurement and modeling
- ParSEC — distributed tasking for exascale

www.icl.utk.edu/jobs



THE UNIVERSITY OF
TENNESSEE
KNOXVILLE

