Adaptive Linear Solvers and Eigensolvers

Jack Dongarra
University of Tennessee
Oak Ridge National Laboratory
University of Manchester

Dense Linear Algebra

- **Common Operations**

\[ Ax = b; \quad \min_{x} \|Ax - b\|; \quad Ax = \lambda x \]

- A major source of large dense linear systems is problems involving the solution of boundary integral equations.
  - The price one pays for replacing three dimensions with two is that what started as a sparse problem in \(O(n^3)\) variables is replaced by a dense problem in \(O(n^2)\).

- Dense systems of linear equations are found in numerous other applications, including:
  - airplane wing design;
  - radar cross-section studies;
  - flow around ships and other off-shore constructions;
  - diffusion of solid bodies in a liquid;
  - noise reduction; and
  - diffusion of light through small particles.
## Existing Math Software - Dense LA

<table>
<thead>
<tr>
<th>DIRECT SOLVERS</th>
<th>License</th>
<th>Support</th>
<th>Type</th>
<th>Language</th>
<th>Mode</th>
<th>Dense</th>
<th>Sparse Direct</th>
<th>Sparse Iterative</th>
<th>Sparse Eigenvalue</th>
<th>Last release date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chameleon</td>
<td>CeCILL-C</td>
<td>No</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>C</td>
<td>M</td>
<td></td>
<td>2014-01-21</td>
</tr>
<tr>
<td>DPLASMA</td>
<td>BSD</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>C</td>
<td>M</td>
<td></td>
<td></td>
<td>2014-04-14</td>
</tr>
<tr>
<td>Eigen</td>
<td>Mozilla</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>2015-01-21</td>
</tr>
<tr>
<td>Elemental</td>
<td>New BSD</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>M</td>
<td>X</td>
<td>X</td>
<td>2014-11-08</td>
</tr>
<tr>
<td>ELPA</td>
<td>LGPL</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>F90</td>
<td>X</td>
<td>M</td>
<td></td>
<td></td>
<td>2015-05-29</td>
</tr>
<tr>
<td>FLINS</td>
<td>BSD</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>2014-05-11</td>
</tr>
<tr>
<td>form-cx</td>
<td>GPL</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>2015-03-10</td>
</tr>
<tr>
<td>LAPACK</td>
<td>BSD</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>2013-11-26</td>
</tr>
<tr>
<td>LAPACK95</td>
<td>BSD</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>2000-11-30</td>
</tr>
<tr>
<td>lotamere</td>
<td>New BSD</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>2014-03-18</td>
</tr>
<tr>
<td>MAGMA</td>
<td>BSD</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>2015-05-05</td>
</tr>
<tr>
<td>NAPACK</td>
<td>BSD</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>2007-06-12</td>
</tr>
<tr>
<td>PLAPACK</td>
<td>LGPL</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>M</td>
<td></td>
<td></td>
<td>2015-04-27</td>
</tr>
<tr>
<td>PLASMA</td>
<td>BSD</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>2013-10-01</td>
</tr>
<tr>
<td>pqitix</td>
<td>by-nc-sa</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>P</td>
<td></td>
<td></td>
<td>2012-05-01</td>
</tr>
<tr>
<td>ScalAPACK</td>
<td>BSD</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>M</td>
<td></td>
<td></td>
<td>2015-05-07</td>
</tr>
<tr>
<td>Trilinos/Pliris</td>
<td>BSD</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>M</td>
<td></td>
<td></td>
<td>2015-05-07</td>
</tr>
<tr>
<td>ViennaCL</td>
<td>MIT</td>
<td>yes</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>2014-12-11</td>
</tr>
</tbody>
</table>

For more information, visit: [http://www.netlib.org/utk/people/JackDongarra/la-sw.html](http://www.netlib.org/utk/people/JackDongarra/la-sw.html)

- LINPACK, EISPACK, LAPACK, ScalAPACK
- PLASMA, MAGMA

7/27/18
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

<table>
<thead>
<tr>
<th>Software/Algorithms follow hardware evolution in time</th>
</tr>
</thead>
</table>
| **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  
- 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

Floating point operations per cycle per core

- Most of the recent computers have FMA (Fused multiple add):
  (i.e. $x \leftarrow x + y \times z$ 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

  - 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

FLOPS = cores $\times$ clock $\times$ FLOPs$/$cycle
In 167 cycles can do 2672 DP Flops

- Main memory: 167 cycles
- L3 Cache Full Random access: 38 cycles
- L3 Cache In Page Random access: 18 cycles
- L3 Cache sequential access: 14 cycles
- L2 Cache Full Random access: 11 cycles
- L2 Cache In Page Random access: 11 cycles
- L2 Cache sequential access: 11 cycles
- L1 Cache Full Random access: 4 cycles
- L1 Cache In Page Random access: 4 cycles
- L1 Cache sequential access: 4 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

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

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

total_time ≥ max ( time_comm, time_comp )

= max ( 0.23ms, 0.01ms ) = 0.23ms

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

- **AXPY:** $y \leftarrow \alpha x + y$
- **DOT:** $\alpha \leftarrow x^T y$

- 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

- **GEMV:** $y \leftarrow \alpha A x + y$

- 2n^2 FLOPs
- n^2 memory references
- RATIO Fl Pt Ops to Memory Ops: 2:1

Level 3 BLAS Matrix-Matrix operations

- **GEMM:** $C \leftarrow \alpha A + \beta B C$

- 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:
  - $(860^2 + 2 \times 860 \text{ double}) \times (8 \text{ Bytes / double}) \sim 6 \text{ MBytes}$
  
  Matrix and two vectors fit in cache (6 MBytes).

• Time to move the data from memory to cache:
  - $(6 \text{ MBytes}) / (25.6 \text{ GBytes/sec}) = 0.23 \text{ ms}$

• Time to perform computation of GEMV:
  - $(2n^2 \text{ flops}) / (56 \text{ Gflop/sec}) = 0.026 \text{ ms}$
Matrix - Vector Operations

\[ \text{total}_{-}\text{time} \geq \max ( \text{time}_{-}\text{comm} , \text{time}_{-}\text{comp} ) \]

\[ = \max ( 0.23\text{ms} , 0.026\text{ms} ) = 0.23\text{ms} \]

Performance = \( \frac{2 \times 860^2 \text{ flops}}{0.23\text{ms}} \) = 6.4 Gflop/s

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

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

Peak is 56 Gflop/s

We say that the operation is communication bounded. Very little reuse of data.
• Take two double precision vectors x and y of size n=500.

• Data size:
  – \((500^2 \text{ double}) \times (8 \text{ Bytes / double}) = 2 \text{ MBytes per matrix}
  (Three matrices fit in cache (6 MBytes). OK.)

• Time to move the matrices in cache:
  – \((6 \text{ MBytes}) / (25.6 \text{ GBytes/sec}) = 0.23 \text{ ms}\)

• Time to perform computation in GEMM:
  – \((2n^3 \text{ flops}) / (56 \text{ Gflop/sec}) = 4.5 \text{ ms}\)
Matrix Matrix Operations

\[ \text{total_time} \geq \max( \text{time}_\text{comm} , \text{time}_\text{comp} ) \]
\[
= \max(0.23\text{ms}, 4.46\text{ms}) = 4.46\text{ms}
\]

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

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

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

If we assume total_time \(\approx\) time_comm + time_comp, we get

**Performance for GEMM \(\approx\) 55.5 Gflop/sec**

Performance for DOT \(\leq\) 3.2 Gflop/s
Performance for GEMV \(\leq\) 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.
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
• 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

<table>
<thead>
<tr>
<th>Software/Algorithms follow hardware evolution in time</th>
<th>LINPACK (70's) (Vector operations)</th>
<th>Rely on</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>- Level-1 BLAS operations</td>
</tr>
<tr>
<td>LAPACK (80's) (Blocking, cache friendly)</td>
<td><img src="Diagram1.png" alt="Diagram" /></td>
<td>Rely on</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Level-3 BLAS operations</td>
</tr>
<tr>
<td>ScaLAPACK (90's) (Distributed Memory)</td>
<td><img src="Diagram2.png" alt="Diagram" /></td>
<td>Rely on</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- PBLAS Mess Passing</td>
</tr>
</tbody>
</table>

### LINPACK (70's)
- Vector operations

### LAPACK (80's)
- Blocking, cache friendly

### ScaLAPACK (90's)
- Distributed Memory

#### 2D Block Cyclic Layout

**Matrix point of view**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

**Processor point of view**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
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
- PBLAS
- LAPACK
- BLAS
- BLACS
- MPI

Global addressing
Local addressing
Platform independent
Platform specific
ScaLAPACK routine, solve $AX = B$

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

input:

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$i_p$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_{11}$</td>
<td>$B_{11}$</td>
<td>$i_p_1$</td>
</tr>
<tr>
<td>$A_{12}$</td>
<td>$B_{12}$</td>
<td>$i_p_2$</td>
</tr>
<tr>
<td>$A_{13}$</td>
<td></td>
<td>$i_p_3$</td>
</tr>
<tr>
<td>$A_{21}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A_{22}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A_{23}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A_{31}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A_{32}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A_{33}$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

output:

<table>
<thead>
<tr>
<th>$L$</th>
<th>$U$</th>
<th>$i_p$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_{11}$</td>
<td>$U_{11}$</td>
<td>$i_p_1$</td>
</tr>
<tr>
<td>$L_{12}$</td>
<td>$U_{12}$</td>
<td>$i_p_2$</td>
</tr>
<tr>
<td>$L_{13}$</td>
<td></td>
<td>$i_p_3$</td>
</tr>
<tr>
<td>$L_{21}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$L_{22}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$L_{23}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$L_{31}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$L_{32}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$L_{33}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$U_{12}$</td>
<td>$U_{13}$</td>
<td></td>
</tr>
<tr>
<td>$U_{22}$</td>
<td>$U_{23}$</td>
<td></td>
</tr>
<tr>
<td>$U_{32}$</td>
<td>$U_{33}$</td>
<td></td>
</tr>
</tbody>
</table>

Global matrix point of view

- 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

Global matrix view

Local process point of view
2D block-cyclic layout \( m \times n \) matrix

\( p \times q \) process grid

<table>
<thead>
<tr>
<th>Global matrix view</th>
<th>Local process point of view</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 11 ) ( 12 ) ( 13 )</td>
<td>Process 1, 1 ( \quad ) Process 1, 2 ( \quad ) Process 1, 3</td>
</tr>
<tr>
<td>( 21 ) ( 22 ) ( 23 ) ( 14 ) ( 15 ) ( 16 ) ( 17 ) ( 18 )</td>
<td>( 12 ) ( 15 ) ( 18 ) ( 13 ) ( 16 )</td>
</tr>
<tr>
<td>( 31 ) ( 32 ) ( 33 ) ( 34 ) ( 35 ) ( 36 ) ( 37 ) ( 38 )</td>
<td>( 32 ) ( 35 ) ( 38 ) ( 33 ) ( 36 )</td>
</tr>
<tr>
<td>( 44 ) ( 42 ) ( 43 ) ( 44 ) ( 45 ) ( 46 ) ( 47 ) ( 48 )</td>
<td>( 52 ) ( 55 ) ( 58 ) ( 53 ) ( 56 )</td>
</tr>
<tr>
<td>( 51 ) ( 52 ) ( 53 ) ( 54 ) ( 55 ) ( 56 ) ( 57 ) ( 58 )</td>
<td>( 71 ) ( 74 ) ( 77 ) ( 73 ) ( 76 )</td>
</tr>
<tr>
<td>( 61 ) ( 62 ) ( 63 ) ( 64 ) ( 65 ) ( 66 ) ( 67 ) ( 68 )</td>
<td>( 72 ) ( 75 ) ( 78 ) ( 93 ) ( 96 )</td>
</tr>
<tr>
<td>( 71 ) ( 72 ) ( 73 ) ( 74 ) ( 75 ) ( 76 ) ( 77 ) ( 78 )</td>
<td>( 91 ) ( 94 ) ( 97 ) ( 91 ) ( 94 )</td>
</tr>
<tr>
<td>( 81 ) ( 82 ) ( 83 ) ( 84 ) ( 85 ) ( 86 ) ( 87 ) ( 88 )</td>
<td>( 92 ) ( 95 ) ( 98 ) ( 92 ) ( 95 )</td>
</tr>
<tr>
<td>( 91 ) ( 92 ) ( 93 ) ( 94 ) ( 95 ) ( 96 ) ( 97 ) ( 98 )</td>
<td>( 94 ) ( 95 ) ( 96 ) ( 94 ) ( 95 )</td>
</tr>
</tbody>
</table>

31
2D block-cyclic layout $m \times n$ matrix
$p \times q$ process grid
Parallelism in ScaLAPACK

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

- 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**

DAG scheduled parallelism

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

\[
\begin{align*}
\text{Tile Algorithm} & \quad \Rightarrow \\
\text{Tile Algorithm} & \quad \Rightarrow
\end{align*}
\]
Track dependencies – Directed acyclic graph (DAG)

Fork-join schedule on 4 cores with artificial synchronizations

Reorder without synchronizations

Critical path
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(u,k)[0:tilesize]) \
        depend(inout:A(m,n)[0:tilesize])
        {   GEMM( A(m,k), A(n,k), A(m,n) ); } 
    }
} }
Algorithms

Cholesky

- **Algorithm**
  - equivalent to LAPACK

- **Numerics**
  - same as LAPACK

- **Performance**
  - comparable to vendor on few cores
  - much better than vendor on many cores

![Graph showing Cholesky Performance (double prec.)](image)

AMD Istanbul, 2.8 GHz, 8 sockets (48 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: $3t^2 - 2t$
Total: $25(7t^3 - 3)$

sync:
770 Gflop/s
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 $L$
Compute $A^{-1} = L^{-T}L^{-1}$

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

$sync: 770 \text{ Gflop/s}$
$async: 1001 \text{ Gflop/s}$

Total: $18(3t^3 + 6)$

PLASMA – Inverse of the Variance-Covariance Matrix
Emerging software solutions

- **PLASMA**
  - Tile layout & algorithms
  - Dynamic scheduling — OpenMP 4

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

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

- **SLATE — DOE ECP Project**
  - DPLAMA Hybrid
  - C++
  - Update to state-of-the-art algorithms
API for Batching BLAS Operations

- We are proposing, as a community standard, an API for Batched Basic Linear Algebra Operations
- The focus is on multiple independent BLAS operations
  - Think “small” matrices (n<500) that are operated on in a single routine.
- Goal to be more efficient and portable for multi/manycore & accelerator systems.
- We can show 2x speedup and 3x better energy efficiency.
Level 1, 2 and 3 BLAS

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
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 by with 16-bit floating point

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
- TRSMs, QRs, or LUs
- TRSMs, TRMMs
- Updates (Schur complement) GEMMs, SYRKs, TRMMs

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

**Tasks manager dispatcher**

Based on the kernel design that decide the number of TB or threads (GPU/CPU) and through the Nvidia/OpenMP scheduler.

High percentage of the resources is used.
Batched Computations: How do we design and optimize

68 cores Intel Xeon Phi KNL 7250, 1.3 GHz. DP peak is 2662 Gflop/s compiled with icc and using Intel MKL 2017

Batched dgemm BLAS 3
Standard dgemm BLAS 3

Small sizes
medium sizes
large sizes

Switch to non-batched

C = C + A*B

Gflop/s

50~1000 matrices of size

2~3x

100X
Batched Computations: How do we design and optimize

**Nvidia P100**

- **Small sizes**
  - Switch to non-batched
  - $C = C + A*B$

- **Medium sizes**
  - $1.2x$

- **Large sizes**

<table>
<thead>
<tr>
<th>Matrix Size (in 64 256 512 1000 1500 2000 2500 3000 3500 4000)</th>
<th>Gflop/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>0</td>
</tr>
<tr>
<td>256</td>
<td>0</td>
</tr>
<tr>
<td>512</td>
<td>0</td>
</tr>
<tr>
<td>1000</td>
<td>0</td>
</tr>
<tr>
<td>1500</td>
<td>0</td>
</tr>
<tr>
<td>2000</td>
<td>0</td>
</tr>
<tr>
<td>2500</td>
<td>0</td>
</tr>
<tr>
<td>3000</td>
<td>0</td>
</tr>
<tr>
<td>3500</td>
<td>0</td>
</tr>
<tr>
<td>4000</td>
<td>0</td>
</tr>
</tbody>
</table>

- **Batched dgemm BLAS 3**
- **Standard dgemm BLAS 3**

- **30X**
- **4500**
- **5000**
IEEE 754 Half Precision (16-bit) Floating Pt Standard

A lot of interest driven by “machine learning”

Google TPU different than 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

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Range</th>
<th>$u = 2^{-t}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>half</td>
<td>16 bits</td>
<td>$10^{±5}$</td>
<td>$2^{-11} \approx 4.9 \times 10^{-4}$</td>
</tr>
<tr>
<td>single</td>
<td>32 bits</td>
<td>$10^{±38}$</td>
<td>$2^{-24} \approx 6.0 \times 10^{-8}$</td>
</tr>
<tr>
<td>double</td>
<td>64 bits</td>
<td>$10^{±308}$</td>
<td>$2^{-53} \approx 1.1 \times 10^{-16}$</td>
</tr>
<tr>
<td>quadruple</td>
<td>128 bits</td>
<td>$10^{±4932}$</td>
<td>$2^{-113} \approx 9.6 \times 10^{-35}$</td>
</tr>
</tbody>
</table>

• 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

FP16 storage/input  Full precision product  Sum with FP32 accumulator  Convert to FP32 result

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
Leveraging Half Precision in HPC on V100

Study of the Matrix Matrix multiplication kernel on Nvidia V100

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

~12X
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
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

\[
\begin{array}{c|c|c|c|c|c}
2k & 4k & 6k & 8k & 10k & 12k \\
\hline
\text{FP16 TC square} & \text{FP16 square} & \text{FP32 square} & \text{FP64 square} & \text{FP16 k=256} & \text{FP16 k=256} \\
\text{FP16 TC square} & \text{FP16 square} & \text{FP32 square} & \text{FP64 square} & \text{FP16 k=256} & \text{FP16 k=256} \\
\text{FP16 TC square} & \text{FP16 square} & \text{FP32 square} & \text{FP64 square} & \text{FP16 k=256} & \text{FP16 k=256} \\
\hline
\text{FP16 TC square} & \text{FP16 square} & \text{FP32 square} & \text{FP64 square} & \text{FP16 k=256} & \text{FP16 k=256} \\
\hline
\text{FP16 TC square} & \text{FP16 square} & \text{FP32 square} & \text{FP64 square} & \text{FP16 k=256} & \text{FP16 k=256} \\
\hline
\text{FP16 TC square} & \text{FP16 square} & \text{FP32 square} & \text{FP64 square} & \text{FP16 k=256} & \text{FP16 k=256} \\
\hline
\end{array}
\]
LU factorization is used to solve a linear system \( Ax = b \)

\[
L \cdot U \cdot x = b \quad \text{then} \quad U \cdot x = y
\]

Study of the LU factorization algorithm on Nvidia V100
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 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,
Iterative refinement for dense systems, \( Ax = b \), can work this way:

\[
L U = lu(A) \\
x = U \backslash (L \backslash b) \\
r = b - Ax
\]

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:
   - \( z = U \backslash (L \backslash r) \) Classical Iterative Refinement
   - GMRes preconditioned by the LU to solve \( Az = r \) Iterative Refinement using GMRes
2. \( x = x + z \)
3. \( r = b - Ax \)

END


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.

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

The reason is that such software is optimized to handle large sizes of data with embarrassingly parallel computational kernels such as matrix multiplication (GEMM). In dense linear algebra, the idea of blocking enables the use of compute-intensive trailing matrix updates that are well-suited for GPUs [19]. However, when the sizes are very small, the LAPACK-style blocking cannot be applied, since the blocking sizes will be very small, leading to memory-bound trailing matrix updates. We need a different design mindset in order to develop high performance GPU kernels to handle such type of workloads.

This paper presents highly optimized GPU kernels for batched LU factorization and matrix inversion. The kernels can handle millions of matrices of sizes up to 32. We show a step-by-step methodology, where incremental improvements in the kernel design lead to incremental performance gains. We justify all of our design choices by showing the performance before and after every incremental improvement. One of the main advantages of our design is the elimination of the expensive intermediate row interchanges by delaying them to the end of the kernel, which leads to a much faster kernel that produces the exact same result of a LAPACK-style LU-factorization and inversion. The performance results show significant speedups against the vendor-supplied CUBLAS kernels using a Pascal P100 GPU.

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

![Convergence history for Classic Iterative Refinement](image1)

![Convergence history for Iterative Refinement using GMRes](image2)
Leveraging Half Precision in HPC on V100

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

Flops = $2n^3/3$ time meaning twice higher is twice faster

- solving $Ax = b$ using FP64 LU

Matrices generated with positive $\lambda$ and arithmetic distribution of its singular values $\sigma_i = 1 - \left(\frac{i-1}{n-1}\right)(1 - \frac{1}{\text{cond}})$ 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 $\lambda$ and arithmetic distribution of its singular values

$$\sigma_i = 1 - \left(\frac{i-1}{n-1}\right)(1 - \frac{1}{\text{cond}})$$

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

- 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

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

Matrices generated with positive $\lambda$ and arithmetic distribution of its singular values $\sigma_i = 1 - \frac{(i-1)}{n-1}(1 - \frac{1}{\text{cond}})$ 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

- 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 $\lambda$ and arithmetic distribution of its singular values $\sigma_i = 1 - \left(\frac{i-1}{n-1}\right)\left(1 - \frac{1}{\text{cond}}\right)$ and where its condition number is equal to $10^2$. 

Flops = $2n^3/(3 \text{ time})$ meaning twice higher is twice faster
Leveraging Half Precision in HPC on V100

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

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

- solving $Ax = b$ using FP64 LU

This paper presents highly optimized GPU kernels for batched LU factorization and matrix inversion. The kernels can handle millions of matrices of sizes up to 32. We show a step-by-step methodology, where incremental improvements in the kernel design lead to incremental performance gains. We justify all of our design choices by showing the performance before and after every incremental improvement. One of the main advantages of our design is the elimination of the expensive intermediate row interchanges by delaying them to the end of the kernel, which leads to a much faster kernel that produces the exact same result of a LAPACK-style LU-factorization and inversion. The performance results show significant speedups against the vendor-supplied CUBLAS kernels using a Pascal P100 GPU.
Leveraging Half Precision in HPC on V100

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

\[ \text{Flops} = \frac{2n^3}{3 \text{ time}} \]

meaning twice higher is twice faster

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

- solving $Ax = b$ using FP64 LU
- solving $Ax = b$ using FP32 LU and iterative refinement to achieve FP64 accuracy
Leveraging Half Precision in HPC on V100

Matrices generated with positive $\lambda$ and clustered distribution of its singular values $\sigma_i=(1, \cdots, 1, \frac{1}{\text{cond}})$ 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

- 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 \( \lambda \) and clustered distribution of its singular values \( \sigma_i = (1, \cdots, 1, \frac{1}{\text{cond}}) \) and where its condition number is equal to \( 10^2 \).
Leveraging Half Precision in HPC

Power awareness

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.

- 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

Solving \( Ax = b \) on Nvidia V100

- CPU: 10 cores E5-2650 v3
- GPU: Nvidia V100

Figure 4: Performance in Tflop/s for the three linear solver (FP16->64 dhgesv, FP32->64 dsgesv, FP64 dgesv)
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 T flop/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 T flop/s and requires about 1041 joules providing about 30 Gflops/Watts.

Problem generated with an arithmetic distribution of the singular values $\sigma_i = 1 - \frac{i-1}{n-1}(1 - \frac{1}{\text{cond}})$, and positive eigenvalues.
Leveraging Half Precision in HPC Power awareness

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.

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$\rightarrow$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$\rightarrow$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 \( \sigma_i = 1 - \left( \frac{i - 1}{n - 1} \right) \left( 1 - \frac{1}{\text{cond}} \right) \) and positive eigenvalues.

**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\( \rightarrow \)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\( \rightarrow \)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\( \rightarrow \)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.
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