Adaptive Linear Solvers and Eigensolvers

Jack Dongarra
University of Tennessee
Oak Ridge National Laboratory
University of Manchester
• 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>yes</td>
<td>X X</td>
<td>X</td>
<td>X</td>
<td>X C</td>
<td>M X</td>
<td></td>
<td></td>
<td>2018-09-15</td>
</tr>
<tr>
<td>DPLASMA</td>
<td>BSD</td>
<td>yes</td>
<td>X X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>M X</td>
<td></td>
<td></td>
<td>2014-04-14</td>
</tr>
<tr>
<td>Eigen</td>
<td>MPL2</td>
<td>yes</td>
<td>X X</td>
<td>X X</td>
<td>X</td>
<td>X</td>
<td>X X X</td>
<td>X</td>
<td>X</td>
<td>2018-07-23</td>
</tr>
<tr>
<td>Elemental</td>
<td>New BSD</td>
<td>yes</td>
<td>X X</td>
<td>X</td>
<td>X</td>
<td>M</td>
<td>X X X</td>
<td></td>
<td></td>
<td>2017-02-06</td>
</tr>
<tr>
<td>ELPA</td>
<td>LGPL</td>
<td>yes</td>
<td>X X</td>
<td>F90</td>
<td>X</td>
<td>X</td>
<td>M X</td>
<td></td>
<td></td>
<td>2018-06-01</td>
</tr>
<tr>
<td>FLENS</td>
<td>BSD</td>
<td>yes</td>
<td>X X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>2014-05-11</td>
</tr>
<tr>
<td>LAPACK</td>
<td>BSD</td>
<td>yes</td>
<td>X X</td>
<td>X X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>2017-11-12</td>
</tr>
<tr>
<td>LAPACK95</td>
<td>BSD</td>
<td>yes</td>
<td>X X</td>
<td>X X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>2000-11-30</td>
</tr>
<tr>
<td>libflame</td>
<td>New BSD</td>
<td>yes</td>
<td>X X</td>
<td>X X</td>
<td>X</td>
<td>X</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 X</td>
<td>X X X</td>
<td>X</td>
<td>X</td>
<td>C/O/X</td>
<td>X</td>
<td></td>
<td>2018-06-25</td>
</tr>
<tr>
<td>NAPACK</td>
<td>BSD</td>
<td>yes</td>
<td>X X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X X X</td>
<td></td>
<td></td>
<td>2018-06-25</td>
</tr>
<tr>
<td>PLAPACK</td>
<td>LGPL</td>
<td>yes</td>
<td>X X</td>
<td>X X</td>
<td>X</td>
<td>M</td>
<td>M X</td>
<td></td>
<td></td>
<td>2007-06-12</td>
</tr>
<tr>
<td>PLASMA</td>
<td>BSD</td>
<td>yes</td>
<td>X X</td>
<td>X X</td>
<td>X</td>
<td>X</td>
<td>M/P X</td>
<td></td>
<td></td>
<td>2018-09-04</td>
</tr>
<tr>
<td>ScALAPACK</td>
<td>BSD</td>
<td>yes</td>
<td>X X</td>
<td>X X</td>
<td>X</td>
<td>M</td>
<td>M X</td>
<td></td>
<td></td>
<td>2018-08-20</td>
</tr>
<tr>
<td>Triilinos/Pliris</td>
<td>BSD</td>
<td>yes</td>
<td>X X</td>
<td>X X</td>
<td>X</td>
<td>X</td>
<td>M X</td>
<td></td>
<td></td>
<td>2015-05-07</td>
</tr>
<tr>
<td>ViennaCL</td>
<td>MIT</td>
<td>yes</td>
<td>X X</td>
<td>X C/O/X</td>
<td>X</td>
<td>X</td>
<td>X X X</td>
<td>X</td>
<td></td>
<td>2016-01-20</td>
</tr>
</tbody>
</table>

[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
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>
<tbody>
<tr>
<td><strong>EISPACK (1970's)</strong> (Translation of Algol to F66)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>LINPACK (1980's)</strong> (Vector operations)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>LAPACK (1990's)</strong> (Blocking, cache friendly)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>ScaLAPACK (2000's)</strong> (Distributed Memory)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>PLASMA (2010's)</strong> New Algorithms</td>
</tr>
<tr>
<td>(many-core friendly)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>SLATE (2020's)</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
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. $x \leftarrow x + y \cdot z$ 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, Kaby Lake, Coffee Lake, ...
  - 32 flops/cycle/core DP & 64 flops/cycle/core SP

We are here
Commodity Processors ...

Over provisioned for floating point operations

Today it's all about data movement

Each Core: 32 Flops per core / cycle

With 2.6 GHz

Each Core Peak DP 83.2 Gflop/s
Each Socket Peak 665.6 Gflop/s

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

Need Cache Friendly Algorithms
Matrix Multiply and Data Reuse
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

56 GFLOP/sec/core x 2 cores

CPU

Cache
(6 MB)

25.6 GB/sec

Main memory
(16 GB)

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 \( n=375,000 \).

\[
\text{DOT: } \alpha \leftarrow x^T y
\]

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

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

• Time to perform computation of DOT:
  – \( (2n \text{ flops}) / (56 \text{ Gflop/sec}) = 0.013 \text{ ms} \)
Vector Operations

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

Performance = \( \frac{(2 \times 375,000 \text{ flops})}{0.23\text{ms}} \) = 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: $C \leftarrow \alpha A B + \beta C$

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} / \text{ double}) \approx 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

total_time ≥ max ( time_comm , time_comp )

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

Performance = (2 x 860² flops)/.23ms = 6.4 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 vectors $x$ and $y$ of size $n=500$.

• Data size:
  – $(500^2 \text{ double}) \times (8 \text{ Bytes} / \text{ double}) = 2 \text{ MBytes per matrix}$
  (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

\[
\text{total_time} \geq \max (\text{time_comm}, \text{time_comp})
\]
\[
= \max(0.23\text{ms}, 4.46\text{ms}) = 4.46\text{ms}
\]
For this example, communication time is less than 6% of the computation time.

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

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

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

\[
\text{Performance for GEMM} \approx 55.5 \text{ Gflop/sec}
\]

\[
\text{Performance for DOT} \leq 3.2 \text{ Gflop/s}
\]
\[
\text{Performance for GEMV} \leq 6.4 \text{ Gflop/s}
\]

(Out of 56 Gflop/sec possible, so that would be 99% peak performance efficiency.)
Level 1, 2 and 3 BLAS

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

1 core Intel Haswell i7-4850HQ, 2.3 GHz, Memory: DDR3L-1600MHz
6 MB shared L3 cache, and each core has a private 256 KB L2 and 64 KB L1.
The theoretical peak per core double precision is 56 Gflop/s per core.
Compiled with gcc and using Veclib
Level 1, 2 and 3 BLAS

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

- Compiled with `icc`
- Using Intel MKL 2018

**Diagram Details:**
- **dgemm** (BLAS Level 3): Computation bound
- **dgemv** (BLAS Level 2):
- **daxpy** (BLAS Level 1):

**Performance Metrics:**
- **C = C + A * B**
- **y = y + A * x**
- **y = α * x + y**

**Graph:**
- X-axis: Matrix size (N), vector size (NxN)
- Y-axis: Gflop/s
- Key points:
  - Compute bound: 1050 Gflop/s
  - Memory bound: 8 Gflop/s

**Notes:**
- 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
• 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**
  - 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>
</tr>
</thead>
<tbody>
<tr>
<td>LINPACK (70’s) (Vector operations)</td>
</tr>
<tr>
<td>Rely on - Level-1 BLAS operations</td>
</tr>
<tr>
<td>LAPACK (80’s) (Blocking, cache friendly)</td>
</tr>
<tr>
<td>Rely on - Level-3 BLAS operations</td>
</tr>
<tr>
<td>ScaLAPACK (90’s) (Distributed Memory)</td>
</tr>
<tr>
<td>Rely on - PBLAS Mess Passing</td>
</tr>
</tbody>
</table>

### 2D Block Cyclic Layout

<table>
<thead>
<tr>
<th>Matrix point of view</th>
<th>Processor point of view</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>1 2 3 4 5 6 7 8</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>2 3 4 5 6 7 8 9</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>3 4 5 6 7 8 9 10</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>4 5 6 7 8 9 10 11</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>5 6 7 8 9 10 11 12</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>6 7 8 9 10 11 12 13</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td>7 8 9 10 11 12 13 14</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
</tbody>
</table>

8/2/19
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
  – $2/3 \ n^3$ term
  – Can use **sequential BLAS**, $p \times q = \# \ cores$
    = \# \ MPI \ processes, num_threads = 1
  – Or **multi-threaded BLAS**, $p \times q = \# \ nodes$
    = \# \ MPI \ processes, num_threads = \# \ cores/node
Today’s HPC Environment for Numerical Libraries

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

- 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
Tile Algorithms: Matrix Decomposition

LAPACK Algorithm (right looking)

Tile Algorithm

= chol

= trsm

= syrk

= trsm

= gemm

= syrk

= gemm

= syrk
Track dependencies — Directed acyclic graph (DAG)

Fork-join schedule on 4 cores with artificial synchronizations

Reorder without synchronizations

Critical path
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

Assume a t by t matrix tiling then Cholesky Factorization alone: 3t-2

Total: 25(7t-3)

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

Cholesky $A = LL^T$

Invert $L^{-1}$

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

Cholesky matrix inverse

Total: 18(3t+6)
• 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.

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

Low percentage of the resources is used.
Batched Computation

- **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

50~1000 matrices of size

64 256 512 1000 1800 2200 2500 3000

Gflop/s

0 200 400 600 800 1000 1200 1400 1600 1800 2000

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

2~3x

C = C + A*B

Switch to non-batched

Small sizes

medium sizes

large sizes

50~1000 matrices of size
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
IEEE 754 Half Precision (16-bit) Floating Point 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

<table>
<thead>
<tr>
<th>AMD Radeon Instinct</th>
<th>Instinct M10</th>
<th>Instinct M10</th>
<th>Instinct M12X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Type</td>
<td>16GB GDDR5</td>
<td>4GB HBM</td>
<td>&quot;High Bandwidth Cache and Controller&quot;</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>224GB/sec</td>
<td>512GB/sec</td>
<td>?</td>
</tr>
<tr>
<td>Single Precision</td>
<td>5.7 TFLOPS</td>
<td>8.0 TFLOPS</td>
<td>12.5 TFLOPS</td>
</tr>
<tr>
<td>Half Precision</td>
<td>5.7 TFLOPS</td>
<td>8.2 TFLOPS</td>
<td>25 TFLOPS</td>
</tr>
<tr>
<td>TDP</td>
<td>&lt;150W</td>
<td>&lt;175W</td>
<td>&lt;225W</td>
</tr>
<tr>
<td>Cooling</td>
<td>Passive</td>
<td>Passive</td>
<td>Passive</td>
</tr>
<tr>
<td>GPU</td>
<td>Polaris 1000</td>
<td>Fiji</td>
<td>Vega</td>
</tr>
<tr>
<td>Manufacturing Process</td>
<td>QioFo 14nm</td>
<td>TGMC 20nm</td>
<td>?</td>
</tr>
</tbody>
</table>

GPU PERFORMANCE COMPARISON

<table>
<thead>
<tr>
<th></th>
<th>P100</th>
<th>V100</th>
<th>RTX3080</th>
</tr>
</thead>
<tbody>
<tr>
<td>DL Training FP16</td>
<td>10 TFLOPS</td>
<td>120 TFLOPS</td>
<td>12x</td>
</tr>
<tr>
<td>DL Inference FP16</td>
<td>21 TFLOPS</td>
<td>120 TFLOPS</td>
<td>6x</td>
</tr>
<tr>
<td>FP32</td>
<td>5/10 TFLOPS</td>
<td>7.5/15 TFLOPS</td>
<td>1.5x</td>
</tr>
<tr>
<td>HBM2 Bandwidth</td>
<td>720 GB/s</td>
<td>900 GB/s</td>
<td>1.2x</td>
</tr>
<tr>
<td>STREAM Triad Perf</td>
<td>557 GB/s</td>
<td>855 GB/s</td>
<td>1.5x</td>
</tr>
<tr>
<td>NVLink Bandwidth</td>
<td>160 GB/s</td>
<td>300 GB/s</td>
<td>1.9x</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>4 MB</td>
<td>6 MB</td>
<td>1.5x</td>
</tr>
<tr>
<td>L1 Caches</td>
<td>1.3 MB</td>
<td>10 MB</td>
<td>7.7x</td>
</tr>
</tbody>
</table>
Mixed Precision

- Today many precisions to deal with (IEEE Standard)

<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.)

- float16
- largest fl pt number 65,504

- Google TPU: bfloat16
- 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

\[
D = \begin{pmatrix}
A_{11} & A_{12} & A_{13} \\
A_{21} & A_{22} & A_{23} \\
A_{31} & A_{32} & A_{33} \\
\end{pmatrix}
+ \begin{pmatrix}
B_{11} & B_{12} & B_{13} \\
B_{21} & B_{22} & B_{23} \\
B_{31} & B_{32} & B_{33} \\
\end{pmatrix}
+ \begin{pmatrix}
C_{11} & C_{12} & C_{13} \\
C_{21} & C_{22} & C_{23} \\
C_{31} & C_{32} & C_{33} \\
\end{pmatrix}
\]

\[
D = AB + C
\]
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

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

Matrix matrix multiplication GEMM

\[ C = \alpha \cdot A + B + \beta \cdot C \]
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

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

~12X

Matrix matrix multiplication GEMM

\[ C = \alpha A \beta B + C \]
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.

![Graph showing performance of LU factorization with different precisions and ranks.](graph.png)
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 $Ax=b$
  
  $A \cdot x = b$

- Then:
  
  $Ly = b$
  
  $Ux = y$
Iterative refinement for dense systems, \( Ax = b \), can work this way.

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

**WHILE** \( \| r \| \) not small enough

1. find a correction “z” to adjust x that satisfy \( Az=r \)
   - Classical Iterative Refinement
     \[
     z = U\backslash(L\backslash r)
     \]
   - GMRes preconditioned by the LU to solve \( Az=r \)

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.

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 $x_i + z$

$z$

$x_i$

$x_{i+1}$

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

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

$$x_{i+1} - x_i = -\frac{f(x_i)}{f'(x_i)}$$
Leveraging Half Precision in HPC on V100

Performance Behavior

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

Problem generated with an arithmetic distribution of the singular values
$\sigma_i = 1 - \left(\frac{i-1}{n-1}\right)(1 - \frac{1}{\text{cond}})$ 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 $\sigma_i = 1 - (\frac{i-1}{n-1})(1 - \frac{1}{\text{cond}})$ and positive eigenvalues.
Leveraging Half Precision in HPC on V100

Performance Behavior

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
- 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

Problem generated with an arithmetic distribution of the singular values and positive eigenvalues.

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

- solving \(Ax = b\) using \(\text{FP64 LU}\)
- solving \(Ax = b\) using \(\text{FP32 LU}\) and iterative refinement to achieve \(\text{FP64 accuracy}\)
- solving \(Ax = b\) using \(\text{FP16 LU}\) and iterative refinement to achieve \(\text{FP64 accuracy}\)
- solving \(Ax = b\) using \(\text{FP16 Tensor Cores LU}\) and iterative refinement to achieve \(\text{FP64 accuracy}\)
Leveraging Half Precision in HPC on V100

Performance Behavior

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
- 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 \(\sigma_i = 1 - (i-1)(1 - \frac{1}{\text{cond}})\) 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.
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/

Looking for Grad Students and Post-Docs