Linear Algebra Preparation for Emergent Neural Network Architectures (LAPENNA)

Unit 11

MAGMA, BLAS, and Batched GPU Computing

Kwai Wong, Stan Tomov
Rocco Febbo, Julian Halloy

University of Tennessee, Knoxville

November 5, 2021
The major goal of The Linear Algebra Preparation for Emergent Neural Network Architectures (LAPENNA) program is to provide essential knowledge to advance literacy in AI to sustain the growth and development of the workforce in the cyberinfrastructure (CI) ecosystem on data-driven sciences. This program aims to prepare college researchers to enable, design, and direct their own in-house data-driven science programs and incorporate perspectives from their research activities into their course curricula. LAPENNA focuses to deliver algorithmic and computational techniques, numerical and programming procedures, and implementation of AI software on emergent CPU and GPU platforms.

- Support from NSF, UTK, JICS, ICL, NICS
- LAPENNA, [www.jics.utk.edu/lapenna](http://www.jics.utk.edu/lapenna), NSF award #202409
- [www.icl.utk.edu](http://www.icl.utk.edu), [cfdlab.utk.edu](http://cfdlab.utk.edu), [www.xsede.org](http://www.xsede.org), [www.jics.utk.edu/recsem-reu](http://www.jics.utk.edu/recsem-reu),
- MagmaDNN is a project grown out from the RECSEM REU Summer program supported under NSF award #1659502
Meeting Time

✓ Webinar Meeting time. Friday 4:00pm ET,
✓ Tentative schedule, www.jics.utk.edu/lapenna -- Fall 2021

Topic: LAPENNA Fall 2021 Meeting
Time: Aug 20, 2021 04:00 PM Eastern Time (US and Canada)
   Every week on Fri, 15 occurrence(s)
   Aug 20, 2021 04:00 PM
   Aug 27, 2021 04:00 PM
   Sep 3, 2021 04:00 PM
   Sep 10, 2021 04:00 PM
   Sep 17, 2021 04:00 PM
   Sep 24, 2021 04:00 PM
   Oct 1, 2021 04:00 PM
   Oct 8, 2021 04:00 PM
   Nov 5, 2021 04:00 PM

Please download and import the following iCalendar (.ics) files to your calendar system.
   Weekly: https://tennessee.zoom.us/meeting/tJwud-isrzouHtS8Ri0ny5ysjFEWJlktQuu-/ics?
   icsToken=98tyKuCgrTsrHtWUtB2HRow-A4igd-jzmH5bgrduxC3sUy5KNxrlPMRnBZhG8zh

Join from PC, Mac, Linux, iOS or Android: https://tennessee.zoom.us/j/98301411440
   Password: 893603

Or iPhone one-tap (US Toll): +13126266799,98301411440# or +16468769923,98301411440#
## Schedule of Fall 2021 – Friday 4:00-6:00pm

<table>
<thead>
<tr>
<th>Month</th>
<th>Week</th>
<th>Date</th>
<th>Arrangement</th>
</tr>
</thead>
<tbody>
<tr>
<td>August</td>
<td>Week 00</td>
<td>20</td>
<td>First Meeting, Logistics</td>
</tr>
<tr>
<td></td>
<td>Week 01</td>
<td>27</td>
<td>Computational Ecosystem</td>
</tr>
<tr>
<td>September</td>
<td>Week 02</td>
<td>3</td>
<td>Linear Algebra, Statistical Learning</td>
</tr>
<tr>
<td></td>
<td>Week 03</td>
<td>8 (4:45 - 6pm), 10</td>
<td>Q&amp;A (LA), DNN, Forward Path, Math</td>
</tr>
<tr>
<td></td>
<td>Week 04</td>
<td>17</td>
<td>Backward Path, MLP, example</td>
</tr>
<tr>
<td></td>
<td>Week 05</td>
<td>24</td>
<td>CNN computation</td>
</tr>
<tr>
<td>October</td>
<td>Week 06</td>
<td>Sept 29 (4:45-6pm), 1</td>
<td>Q&amp;A (MLP), Backpropagation example</td>
</tr>
<tr>
<td></td>
<td>Week 07</td>
<td>8</td>
<td>CNN network, Linear Algebra</td>
</tr>
<tr>
<td></td>
<td>Week 08</td>
<td>15</td>
<td>Object detection, optimization</td>
</tr>
<tr>
<td></td>
<td>Week 09</td>
<td>22</td>
<td>Image Segmentation</td>
</tr>
<tr>
<td></td>
<td>Week 10</td>
<td>27 (4:45-6pm), 29</td>
<td>Q&amp;A (RC vehicle), Magma, GPU CUDA, CUDNN</td>
</tr>
<tr>
<td>November</td>
<td>Week 11</td>
<td>5</td>
<td>Magma, CUBLAS, CUDNN</td>
</tr>
<tr>
<td></td>
<td>Week 12</td>
<td>12</td>
<td>RNN, LSTM, Transformer</td>
</tr>
<tr>
<td></td>
<td>Week 13</td>
<td>19</td>
<td>Recap Summary, closing</td>
</tr>
<tr>
<td>December</td>
<td>Workshop</td>
<td>10-13</td>
<td>Workshop</td>
</tr>
</tbody>
</table>
Matrix Algebra on GPU and Multicore Architectures (MAGMA)
Linear systems: Solve $Ax = b$
- Computational electromagnetics, material science, applications using boundary integral equations, airflow past wings, fluid flow around ship and other offshore constructions, and many more

Least squares: Find $x$ to minimize $|| Ax - b ||$
- Computational statistics (e.g., linear least squares or ordinary least squares), econometrics, control theory, signal processing, curve fitting, and many more

Eigenproblems: Solve $Ax = \lambda x$
- Computational chemistry, quantum mechanics, material science, face recognition, PCA, data-mining, marketing, Google Page Rank, spectral clustering, vibrational analysis, compression, and many more

SVD: $A = U \Sigma V^*$ ($Au = \sigma v$ and $A^*v = \sigma u$)
- Information retrieval, web search, signal processing, big data analytics, low rank matrix approximation, total least squares minimization, pseudo-inverse, and many more

Many variations depending on structure of $A$
- $A$ can be symmetric, positive definite, tridiagonal, Hessenberg, banded, sparse with dense blocks, etc.

Sparse solvers
Overview of Dense Numerical Linear Algebra Libraries

<table>
<thead>
<tr>
<th>netlib.org</th>
<th>icl.utk.edu/research</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLAS</td>
<td>Kernels for dense linear algebra</td>
</tr>
<tr>
<td>LAPACK</td>
<td>Sequential dense linear algebra</td>
</tr>
<tr>
<td>ScaLAPACK</td>
<td>Parallel distributed dense linear algebra</td>
</tr>
<tr>
<td>PLASMA</td>
<td>dense linear algebra (multicore)</td>
</tr>
<tr>
<td>MAGMA</td>
<td>Dense/batched/sparse linear algebra (accelerators)</td>
</tr>
<tr>
<td>SLATE</td>
<td>dense linear algebra (distributed memory / multicore / accelerators)</td>
</tr>
<tr>
<td></td>
<td>new software for multicore and accelerators</td>
</tr>
</tbody>
</table>
MAGMA on GPUs

PERFORMANCE & ENERGY EFFICIENCY

MAGMA 2.6.1 LU factorization in double precision arithmetic

- CPU: Intel Xeon E5-2650 v3 (Haswell) 2x10 cores @ 2.30 GHz
- K40: NVIDIA Kepler GPU 15 MP x 192 @ 0.88 GHz
- P100: NVIDIA Pascal GPU 56 MP x 64 @ 1.19 GHz
- V100: NVIDIA Volta GPU 80 MP x 64 @ 1.38 GHz
- A100: NVIDIA Ampere GPU 108 MP x 64 @ 1.41 GHz

Energy efficiency (under ~ the same power draw)

- A100: 22x
- V100: 10x
- P100: 10x

Matrix size N x N

Performance GFLOP/s

Energy efficiency

GFLOPs/Watt

CPU K40 P100 V100 A100
MAGMA Today

**MAGMA** – provides highly optimized LA well beyond LAPACK for GPUs;
– research vehicle for LA on new architectures for a number of projects.

- Collaboration and support from vendors NVIDIA, Intel, and AMD
- Two releases per year
  Latest MAGMA 2.6.1 released on July 13, 2021
  Number of downloads per release ~ 4K
  (MAGMA 2.5.4 – 6,111 downloads, MAGMA 2.6.x – 5,794 downloads)
- MAGMA Forum – now Google Groups
  72 members, 35 discussion topics
- MAGMA Issues
  48 issues total; 23 are still open, and 25 have been resolved
- MAGMA is incorporated/used in
  MATLAB (as of the R2010b),
  contributions in CUBLAS and MKL,
  AMD, Siemens (in NX Nastran 9.1), ArrayFire,
  ABINIT, Quantum-Espresso, R (in HiPLAR & CRAN),
  SIMULIA (Abaqus), MSC Software (Nastran and Marc),
  Cray (in LibSci for accelerators libsci_acc),
  Nano-TCAD (Gordon Bell finalist),
  Numerical Template Toolbox (Numscale), and others.
- MAGMA used in ECP – CEED, PEEKS, xSDK, ALEXA/TASMANIAN, SLATE & FFT,
  ExaSGD; Provides LAPACK:MAGMA and BLAS:MAGMA product

---

**for architectures in**

- CPUs + Nvidia GPUs (CUDA),
- CPUs + AMD GPUs (HIP & OpenCL),
- CPUs + Intel Xeon Phis,
- manycore (native: GPU or KNL/CPU),
- embedded systems, combinations, and
  software stack, e.g., since CUDA x

**for precisions in**

- s, d, c, z,
- half-precision (FP16),
- mixed, ...

**for interfaces**

- heterogeneous CPU/GPU, native, ...
  - LAPACK
  - BLAS
  - Batched LAPACK
  - Batched BLAS
  - Sparse
  - Tensors
  - MAGMA-DNN
  - Templates
  - ...

---

---

---
• **Availability**
  - [http://icl.utk.edu/magma/](http://icl.utk.edu/magma/) download (latest MAGMA 2.6.1), documentation, forum
  - [https://bitbucket.org/icl/magma](https://bitbucket.org/icl/magma) Git repo

• **Support**
  - Linux, macOS, Windows
  - CUDA >= 7.0; recommend latest CUDA
  - CUDA architecture >= 2.0 (Fermi, Kepler, Maxwell, Pascal, Volta, Turing)
  - AMD GPUs through HIP
  - BLAS & LAPACK: Intel MKL, OpenBLAS, macOS Accelerate, ...

• **May be pre-installed on supercomputers**
  
  `titan-ext1> module avail magma`
  `------------- /sw/xk6/modulefiles -------------`
  `magma/1.3       magma/1.6.2(default)`
  `------------- /sw/xk6/modulefiles -------------`
Installation options

1. Makefile
   - Edit make.inc for compilers and flags (see make.inc examples)
   - magma> **make && make install**

2. CMake
   - magma> **mkdir build && cd build**
   - magma/build> **cmake .. or ccmake ..**
   - Adjust settings, esp. LAPACK_LIBRARIES and GPU_TARGET
   - magma/build> **make && make install**

3. Spack
   - **spack install magma**
BLAS: Basic Linear Algebra Subroutines

• Level 1 BLAS — vector operations
  – $O(n)$ data and flops (floating point operations)
  – Memory bound: $O(1)$ flops per memory access

\[ y = \alpha x + \beta y \]
BLAS: Basic Linear Algebra Subroutines

• Level 1 BLAS — vector operations
  – $O(n)$ data and flops (floating point operations)
  – Memory bound: $O(1)$ flops per memory access

• Level 2 BLAS — matrix-vector operations
  – $O(n^2)$ data and flops
  – Memory bound: $O(1)$ flops per memory access
BLAS: Basic Linear Algebra Subroutines

• Level 1 BLAS — vector operations
  – $O(n)$ data and flops (floating point operations)
  – Memory bound:
    $O(1)$ flops per memory access

• Level 2 BLAS — matrix-vector operations
  – $O(n^2)$ data and flops
  – Memory bound:
    $O(1)$ flops per memory access

• Level 3 BLAS — matrix-matrix operations
  – $O(n^2)$ data, $O(n^3)$ flops
  – Surface-to-volume effect
  – Compute bound:
    $O(n)$ flops per memory access
BLAS: Basic Linear Algebra Subroutines

- Level 1 BLAS
  \[ y = \alpha x + \beta y \]

- Level 2 BLAS
  \[ y = \alpha A x + \beta y \]

- Level 3 BLAS
  \[ C = \alpha A B + \beta C \]

Use of BLAS for portability

<table>
<thead>
<tr>
<th>Software/Algorithms follow hardware evolution in time</th>
<th>Level 1 BLAS</th>
</tr>
</thead>
<tbody>
<tr>
<td>LINPACK (70's)</td>
<td></td>
</tr>
<tr>
<td>(Vector operations)</td>
<td></td>
</tr>
<tr>
<td>LAPACK (80's)</td>
<td></td>
</tr>
<tr>
<td>(Blocking, cache friendly)</td>
<td></td>
</tr>
<tr>
<td>ScALAPACK (90's)</td>
<td></td>
</tr>
<tr>
<td>(Distributed Memory)</td>
<td></td>
</tr>
<tr>
<td>PLASMA (00's)</td>
<td></td>
</tr>
<tr>
<td>New Algorithms</td>
<td></td>
</tr>
<tr>
<td>(many-core friendly)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PBLAS</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLAS on tiles + DAG scheduling</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MAGMA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hybrid Algorithms</td>
</tr>
<tr>
<td>(heterogeneity friendly)</td>
</tr>
</tbody>
</table>

| BLAS tasking +                                   |
| (CPU / GPU / Xeon Phi)                           |
| hybrid scheduling                                |
Why higher level BLAS?

- By taking advantage of the principle of locality:
- Present the user with as much memory as is available in the cheapest technology.
- Provide access at the speed offered by the fastest technology.
- Can only do arithmetic on data at the top of the hierarchy
- Higher level BLAS lets us do this

<table>
<thead>
<tr>
<th>BLAS</th>
<th>Memory Refs</th>
<th>Flops</th>
<th>Flops/ Memory Refs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 1</td>
<td>3n</td>
<td>2n</td>
<td>2/3</td>
</tr>
<tr>
<td>y = y + αx</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Level 2</td>
<td>n²</td>
<td>2n²</td>
<td>2</td>
</tr>
<tr>
<td>y = y + Ax</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Level 3</td>
<td>4n²</td>
<td>2n³</td>
<td>n/2</td>
</tr>
<tr>
<td>C = C + AB</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Level 1, 2 and 3 BLAS
Nvidia P100, 1.19 GHz, Peak DP = 4700 Gflop/s

- Registers
- L 1 Cache
- L 2 Cache
- Local Memory
- Remote Memory
- Secondary Memory
MAGMA Overview

• Hybrid LAPACK-style functions
  – Matrix factorizations: LU, Cholesky, QR, eigenvalue, SVD, ...
  – Solve linear systems and linear least squares, ...
  – Nearly all are synchronous: return on CPU when computation is finished

• GPU BLAS and auxiliary functions
  – Matrix-vector multiply, matrix norms, transpose (in-place and out-of-place), ...
  – Most are asynchronous: return immediately on CPU; computation proceeds on GPU

• Wrappers around CUDA and cuBLAS
  – BLAS routines (gemm, symm, symv, ...)
  – Copy host ⇔ device, queue (stream) support, GPU malloc & free, ...
Naming example

• magma_ or magmablas_ prefix

• Precision (1–2 characters)
  – Single, Double, single Complex, “Z” double complex, Integer
    Mixed precision (DS and ZC)

• Matrix type (2 characters)
  – GEneral SYmmetric HErmetian POSitive definite
    ORthogonal UNitary Triangular

• Operation (2–3+ characters)
  – SV solve
    TRF triangular factorization
    EV eigenvalue problem
    GV generalized eigenvalue problem
    etc.

• _gpu suffix for interface
# Linear solvers

- **Solve linear system:** $AX = B$
- **Solve linear least squares:** minimize $\|AX - B\|_2$

<table>
<thead>
<tr>
<th>Type</th>
<th>Routine</th>
<th>Mixed precision routine</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>dgesv</td>
<td>dsgesv</td>
<td>✓</td>
</tr>
<tr>
<td>Positive definite</td>
<td>dposv</td>
<td>dsposv</td>
<td>✓</td>
</tr>
<tr>
<td>Symmetric</td>
<td>dsyev</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Hermitian</td>
<td>zhesv</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Least squares</td>
<td>dgels</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

Selected routines; complete documentation at [http://icl.utk.edu/magma/](http://icl.utk.edu/magma/)
Eigenvalue / singular value problems

- Eigenvalue problem: \( Ax = \lambda x \)
- Generalized eigenvalue problem: \( Ax = \lambda Bx \) (and variants)
- Singular value decomposition: \( A = U \Sigma V^H \)

<table>
<thead>
<tr>
<th>Matrix type</th>
<th>Operation</th>
<th>Routine</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>SVD</td>
<td>dgesvd, dgesdd</td>
<td>✓</td>
</tr>
<tr>
<td>General non-symmetric</td>
<td>Eigenvalue</td>
<td>dgeev</td>
<td>✓</td>
</tr>
<tr>
<td>Symmetric</td>
<td>Eigenvalue</td>
<td>dsyevd / zheevd</td>
<td>✓</td>
</tr>
<tr>
<td>Symmetric</td>
<td>Generalized</td>
<td>dsygvd / zhegvd</td>
<td>✓</td>
</tr>
</tbody>
</table>

Additional variants; complete documentation at [http://icl.utk.edu/magma/](http://icl.utk.edu/magma/)
Fastest are divide-and-conquer (gesdd, syevd) and 2-stage versions.
Computational routines

- Computational routines solve one part of problem

<table>
<thead>
<tr>
<th>Matrix type</th>
<th>Operation</th>
<th>Routine</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>LU</td>
<td>dgetrf</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Solve (given LU)</td>
<td>dgetrs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Inverse</td>
<td>dgetri</td>
<td></td>
</tr>
<tr>
<td>SPD</td>
<td>Cholesky</td>
<td>dpotrf</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Solve (given LLᵀ)</td>
<td>dpotrs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Inverse</td>
<td>dpotri</td>
<td>✓</td>
</tr>
<tr>
<td>General</td>
<td>QR</td>
<td>dgeqrf</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Generate Q</td>
<td>dorgqr / zungqr</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Multiply by Q</td>
<td>dormqr / zunmqr</td>
<td>✓</td>
</tr>
</tbody>
</table>

Selected routines; complete documentation at [http://icl.utk.edu/magma/](http://icl.utk.edu/magma/)
Testers/benchmarks
Every routine has tester, also used as benchmark

# (abbreviated output)
magma> cd testing
magma/testing> ./testing_dsymv -n 123 -n 1000:18000:1000 --lapack --check
% MAGMA 2.5.0 compiled for CUDA capability >= 6.0, 32-bit magma_int_t, 64-bit pointer.
% device 0: Tesla P100-PCIE-16GB, 1328.5 MHz clock, 16276.2 MiB memory, capability 6.0

<table>
<thead>
<tr>
<th>uplo</th>
<th>N</th>
<th>MAGMA</th>
<th>Atomics</th>
<th>cuBLAS</th>
<th>CPU</th>
<th>error</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Gflop/s</td>
<td>Gflop/s</td>
<td>Gflop/s</td>
<td>Gflop/s</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>-----</td>
<td>---------</td>
<td>---------</td>
<td>---------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td></td>
<td>123</td>
<td>0.76</td>
<td>0.76</td>
<td>0.51</td>
<td>0.58</td>
<td>ok</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>27.44</td>
<td>34.41</td>
<td>29.88</td>
<td>8.86</td>
<td>ok</td>
</tr>
<tr>
<td></td>
<td>2000</td>
<td>45.74</td>
<td>70.83</td>
<td>33.91</td>
<td>12.78</td>
<td>ok</td>
</tr>
<tr>
<td></td>
<td>3000</td>
<td>75.30</td>
<td>108.51</td>
<td>40.09</td>
<td>17.76</td>
<td>ok</td>
</tr>
<tr>
<td></td>
<td>4000</td>
<td>100.64</td>
<td>131.23</td>
<td>41.13</td>
<td>17.57</td>
<td>ok</td>
</tr>
<tr>
<td></td>
<td>5000</td>
<td>118.17</td>
<td>162.35</td>
<td>41.46</td>
<td>16.33</td>
<td>ok</td>
</tr>
<tr>
<td></td>
<td>6000</td>
<td>141.21</td>
<td>180.43</td>
<td>42.16</td>
<td>17.55</td>
<td>ok</td>
</tr>
<tr>
<td></td>
<td>7000</td>
<td>157.81</td>
<td>200.44</td>
<td>41.94</td>
<td>19.32</td>
<td>ok</td>
</tr>
<tr>
<td></td>
<td>8000</td>
<td>169.54</td>
<td>198.21</td>
<td>41.78</td>
<td>19.12</td>
<td>ok</td>
</tr>
<tr>
<td></td>
<td>9000</td>
<td>188.40</td>
<td>216.07</td>
<td>42.28</td>
<td>21.50</td>
<td>ok</td>
</tr>
<tr>
<td></td>
<td>10000</td>
<td>195.92</td>
<td>224.50</td>
<td>42.36</td>
<td>17.44</td>
<td>ok</td>
</tr>
<tr>
<td></td>
<td>11000</td>
<td>214.93</td>
<td>237.51</td>
<td>45.91</td>
<td>21.30</td>
<td>ok</td>
</tr>
<tr>
<td></td>
<td>12000</td>
<td>219.33</td>
<td>233.44</td>
<td>45.76</td>
<td>20.81</td>
<td>ok</td>
</tr>
<tr>
<td></td>
<td>13000</td>
<td>217.52</td>
<td>241.45</td>
<td>42.49</td>
<td>21.29</td>
<td>ok</td>
</tr>
<tr>
<td></td>
<td>14000</td>
<td>231.26</td>
<td>249.06</td>
<td>45.84</td>
<td>19.71</td>
<td>ok</td>
</tr>
<tr>
<td></td>
<td>15000</td>
<td>232.12</td>
<td>255.98</td>
<td>45.87</td>
<td>19.60</td>
<td>ok</td>
</tr>
<tr>
<td></td>
<td>16000</td>
<td>239.26</td>
<td>250.89</td>
<td>45.58</td>
<td>22.61</td>
<td>ok</td>
</tr>
<tr>
<td></td>
<td>17000</td>
<td>240.74</td>
<td>257.13</td>
<td>45.69</td>
<td>23.15</td>
<td>ok</td>
</tr>
<tr>
<td></td>
<td>18000</td>
<td>242.45</td>
<td>265.05</td>
<td>45.75</td>
<td>19.09</td>
<td>ok</td>
</tr>
</tbody>
</table>
## BLAS and auxiliary routines

<table>
<thead>
<tr>
<th>Category</th>
<th>Operation</th>
<th>Routine (all GPU interface)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Level 1 BLAS</strong></td>
<td>$y = \alpha x + y$</td>
<td>daxpy</td>
</tr>
<tr>
<td></td>
<td>$r = x^T y$</td>
<td>ddot</td>
</tr>
<tr>
<td><strong>Level 2 BLAS</strong></td>
<td>$y = \alpha Ax + \beta y$, general $A$</td>
<td>dgemv</td>
</tr>
<tr>
<td></td>
<td>$y = \alpha Ax + \beta y$, symmetric $A$</td>
<td>dsymv</td>
</tr>
<tr>
<td><strong>Level 3 BLAS</strong></td>
<td>$C = \alpha AB + \beta C$</td>
<td>dgemm</td>
</tr>
<tr>
<td></td>
<td>$C = \alpha AB + \beta C$, symmetric $A$</td>
<td>dsymmm</td>
</tr>
<tr>
<td></td>
<td>$C = \alpha AA^T + \beta C$, symmetric $C$</td>
<td>dsyrk</td>
</tr>
<tr>
<td><strong>Auxiliary</strong></td>
<td>$|A|<em>1$, $|A|</em>{\infty}$, $|A|<em>{\text{fro}}$, $|A|</em>{\max}$</td>
<td>dlange (norm, general $A$)</td>
</tr>
<tr>
<td></td>
<td>$B = A^T$ (out-of-place)</td>
<td>dlansy (norm, symmetric $A$)</td>
</tr>
<tr>
<td></td>
<td>$A = A^T$ (in-place, square)</td>
<td>dtranspose</td>
</tr>
</tbody>
</table>

Selected routines; complete documentation at [http://icl.utk.edu/magma/](http://icl.utk.edu/magma/)
Matrix layout

General m-by-n matrix, LAPACK column-major layout

Symmetric / Hermitian / Triangular n-by-n matrix
  - uplo = Lower or Upper
  - Entries in opposite triangle ignored
Simple example

- **Solve** $AX = B$
  - Double precision, **GEneral matrix, SolVe** (**DGESV**)
- Traditional LAPACK call

```cpp
#include "magma_lapack.h"

int main( int argc, char** argv )
{
    int n = 100, nrhs = 10;
    int lda = n, ldx = n;
    double *A = new double[ lda*n ];
    double *X = new double[ ldx*nrhs ];
    int* ipiv = new int[ n ];

    // ... fill in A and X with your data
    // A[ i + j*lda ] = A_ij
    // X[ i + j*ldx ] = X_ij

    // solve AX = B where B is in X
    int info;
    lapackf77_dgesv( &n, &nrhs, A, &lda, ipiv, X, &ldx, &info );
    if (info != 0) {
        throw std::exception();
    }

    // ... use result in X

    delete[] A;
    delete[] X;
    delete[] ipiv;
}
```
Simple example

- MAGMA CPU interface
  - Input & output matrices in CPU host memory
- Add MAGMA init & finalize
- MAGMA call direct replacement for LAPACK call

```c
#include "magma_v2.h"

int main( int argc, char** argv )
{
  magma_init();

  int n = 100, nrhs = 10;
  int lda = n, ldx = n;
  double *A = new double[ lda*n ];
  double *X = new double[ ldx*nrhs ];
  int* ipiv = new int[ n ];

  // ... fill in A and X with your data
  // A[ i + j*lda ] = A_ij
  // X[ i + j*ldx ] = X_ij

  // solve AX = B where B is in X
  int info;
  magma_dgesv( n, nrhs,
                 A, lda, ipiv,
                 X, ldx, &info );
  if (info != 0) {
    throw std::exception();
  }

  // ... use result in X
  delete[] A;
  delete[] X;
  delete[] ipiv;

  magma_finalize();
}
```
Simple example

- MAGMA GPU interface
  - Add `_gpu` suffix
  - Input & output matrices in GPU device memory ("d" prefix on variables)
  - ipiv still in CPU memory
  - Set GPU stride (ldda) to multiple of 32 for better performance
  - roundup returns ceil (n / 32) * 32

- MAGMA malloc & free
  - Type-safe wrappers around cudaMalloc & cudaFree

```cpp
// tutorial2_gpu_interface.cc
int main( int argc, char** argv )
{
    magma_init();

    int n = 100, nrhs = 10;
    int ldda = magma_roundup( n, 32 );
    int lddx = magma_roundup( n, 32 );
    int* ipiv = new int[ n ];
    double *dA, *dX;
    magma_dmalloc( &dA, ldda*n );
    magma_dmalloc( &dX, lddx*nrhs );
    assert( dA != nullptr );
    assert( dX != nullptr );

    int info;
    magma_dgesv_gpu( n, nrhs, dA, ldda, ipiv, dX, lddx, &info );
    if (info != 0) {
        throw std::exception();
    }

    magma_finalize();
}
```
BLAS example

- Matrix multiply $C = -AB + C$
  - Double-precision, GEneral Matrix Multiply (DGEMM)
- Asynchronous
  - BLAS take queue and are async
  - Return to CPU immediately
  - Queue wraps CUDA stream and cuBLAS handle
  - Can create queue from existing CUDA stream and cuBLAS handle, if required

```c++
// tutorial3_blas.cc
int main( int argc, char** argv )
{
    // ... setup matrices on GPU:
    // m-by-k matrix dA,
    // k-by-n matrix dB,
    // m-by-n matrix dC.
    int device;
    magma_queue_t queue;
    magma_getdevice( &device );
    magma_queue_create( device, &queue );

    // C = -A B + C
    magma_dgemm( MagmaNoTrans,
                 MagmaNoTrans, m, n, k,
                 -1.0, dA, ldda,
                 dB, lddb,
                 1.0, dC, lddc, queue );

    // ... do concurrent work on CPU
    magma_queue_sync( queue );

    // ... use result in dC
    magma_queue_destroy( queue );
    // ... cleanup
}
```
Copy example

- Copy data host ↔ device
  - setmatrix (host to device)
  - getmatrix (device to host)
  - copymatrix (device to device)
  - setvector (host to device)
  - getvector (device to host)
  - copyvector (device to device)

- Default is synchronous
  - Return when transfer is done

- Strides (lda, ldda) can differ on CPU and GPU
  - Set GPU stride (ldda) to multiple of 32 for better performance

```c
// tutorial4_copy.cc
int main( int argc, char** argv )
{
  // ... setup A, X in CPU memory;
  // dA, dX in GPU device memory

  int device;
  magma_queue_t queue;
  magma_getdevice( &device );
  magma_queue_create( device, &queue );

  // copy A, X to dA, dX
  magma_dsetmatrix( n, n, A, lda, dA, ldda, queue );
  magma_dsetmatrix( n, nrhs, X, ldx, dX, lddx, queue );

  // ... solve AX = B

  // copy result dX to X
  magma_dgetmatrix( n, nrhs, dX, lddx, X, ldx, queue );

  // ... use result in X
  magma_queue_destroy( queue );
}
```
Async copy

- Add _async suffix
- Use pinned CPU memory
  - Page locked, so DMA can access it
  - Better performance
  - Required by CUDA for async behavior
  - But pinned memory is limited resource, and expensive to allocate

- Overlap:
  - Sending data (host to device)
  - Getting data (device to host)
  - Host computation
  - Device computation

```c
// tutorial5_copy_async.cc
int main( int argc, char** argv )
{
    // ... setup dA, dX, queue

    // allocate A, X in pinned CPU memory
    double *A, *X;
    magma_dmalloc_pinned( &A, lda*n );
    magma_dmalloc_pinned( &X, ldx*nrhs );

    // ... fill in A and X

    // copy A, X to dA, dX, then wait
    magma_dsetmatrix_async( n, n,
        A, lda, dA, ldda, queue );
    magma_dsetmatrix_async( n, nrhs,
        X, ldx, dX, lddx, queue );
    magma_queue_sync( queue );

    // ... solve AX = B

    // copy result dX to X, then wait
    magma_dgetmatrix_async( n, nrhs,
        dX, ldx, X, lddx, queue );
    magma_queue_sync( queue );

    // ... use result in X

    magma_free_pinned( A );
    magma_free_pinned( X );

    // ... cleanup
}
```
magma> cd testing
magma/testing> ./testing_dgetrf -n 123 -n 1000:20000:1000 --lapack --check

% MAGMA 2.2.0 compiled for CUDA capability >= 6.0, 32-bit magma_int_t, 64-bit pointer.
% device 0: Tesla P100-PCIE-16GB, 1328.5 MHz clock, 16276.2 MiB memory, capability 6.0

| M    | N    | CPU Gflop/s (sec) | GPU Gflop/s (sec) | |PA-LU|/(N*|A|) |
|------|------|-------------------|-------------------|-----------------|-------|
123   123 | 0.20 ( 0.01) | 0.40 ( 0.00) | 3.59e-18 | ok    |
1000  1000 | 10.40 ( 0.06) | 43.50 ( 0.02) | 2.76e-18 | ok    |
2000  2000 | 111.64 ( 0.05) | 218.26 ( 0.02) | 2.68e-18 | ok    |
3000  3000 | 288.38 ( 0.06) | 280.28 ( 0.06) | 2.65e-18 | ok    |
4000  4000 | 305.58 ( 0.14) | 545.90 ( 0.08) | 2.81e-18 | ok    |
5000  5000 | 396.16 ( 0.21) | 838.09 ( 0.10) | 2.71e-18 | ok    |
6000  6000 | 413.37 ( 0.35) | 1088.14 ( 0.13) | 2.71e-18 | ok    |
7000  7000 | 426.71 ( 0.54) | 1288.60 ( 0.18) | 2.67e-18 | ok    |
8000  8000 | 447.85 ( 0.76) | 1514.43 ( 0.23) | 2.66e-18 | ok    |
9000  9000 | 461.05 ( 1.05) | 1621.29 ( 0.30) | 2.87e-18 | ok    |
10000 10000 | 524.06 ( 1.27) | 1802.39 ( 0.37) | 2.84e-18 | ok    |
11000 11000 | 554.16 ( 1.60) | 1965.85 ( 0.45) | 2.84e-18 | ok    |
12000 12000 | 559.33 ( 2.06) | 2090.42 ( 0.55) | 2.82e-18 | ok    |
13000 13000 | 563.56 ( 2.60) | 2223.62 ( 0.66) | 2.80e-18 | ok    |
14000 14000 | 566.58 ( 3.23) | 2323.04 ( 0.79) | 2.78e-18 | ok    |
15000 15000 | 567.17 ( 3.97) | 2431.59 ( 0.93) | 2.77e-18 | ok    |
16000 16000 | 556.86 ( 4.90) | 2539.66 ( 1.08) | 2.79e-18 | ok    |
17000 17000 | 579.82 ( 5.65) | 2593.40 ( 1.26) | 2.75e-18 | ok    |
18000 18000 | 584.93 ( 6.65) | 2694.57 ( 1.44) | 2.76e-18 | ok    |
19000 19000 | 585.78 ( 7.81) | 2768.67 ( 1.65) | 2.75e-18 | ok    |
20000 20000 | 587.08 ( 9.08) | 2821.48 ( 1.89) | 2.74e-18 | ok    |
Testers: LU factorization (dgetrf)
# (abbreviated output)
magma> cd testing
magma/testing> ./testing_dsymv -n 123 -n 1000:20000:1000 --lapack --check
% MAGMA 2.5.0 compiled for CUDA capability >= 6.0, 32-bit magma_int_t, 64-bit pointer.
% device 0: Tesla P100-PCIE-16GB, 1328.5 MHz clock, 16276.2 MiB memory, capability 6.0

% uplo = Lower
%! N | MAGMA | Atomics | cuBLAS | CPU | error
%! ---- | ------- | ------- | ------- | ---- | ------
%! 123  | 0.76 | 0.76 | 0.51 | 0.58 | ok     # warmup run
%! 1000 | 27.44 | 34.41 | 29.88 | 8.86 | ok
%! 2000 | 45.74 | 70.83 | 33.91 | 12.78 | ok
%! 3000 | 75.30 | 108.51 | 40.09 | 17.76 | ok
%! 4000 | 100.64 | 131.23 | 41.13 | 17.57 | ok
%! 5000 | 118.17 | 162.35 | 41.46 | 16.33 | ok
%! 6000 | 141.21 | 180.43 | 42.16 | 16.33 | ok
%! 7000 | 157.81 | 200.44 | 41.94 | 19.32 | ok
%! 8000 | 169.54 | 216.07 | 42.28 | 19.12 | ok
%! 9000 | 188.40 | 224.50 | 42.36 | 17.44 | ok
%! 10000 | 195.92 | 224.50 | 42.36 | 17.44 | ok
%! 11000 | 214.93 | 237.51 | 45.91 | 21.30 | ok
%! 12000 | 219.33 | 233.44 | 45.76 | 20.81 | ok
%! 13000 | 217.52 | 241.45 | 42.49 | 21.29 | ok
%! 14000 | 231.26 | 249.06 | 45.84 | 19.71 | ok
%! 15000 | 232.12 | 255.98 | 45.87 | 19.60 | ok
%! 16000 | 239.26 | 250.89 | 45.58 | 22.61 | ok
%! 17000 | 240.74 | 257.13 | 45.69 | 23.15 | ok
%! 18000 | 242.45 | 265.05 | 45.75 | 19.09 | ok
%! 19000 | 242.53 | 262.48 | 45.81 | 22.42 | ok
%! 20000 | 239.53 | 258.24 | 45.63 | 22.83 | ok
Testers: symmetric matrix-vector multiply

Symmetric matrix-vector multiply (dsymv)
P100 GPU, 2 × 10-core 2.3 GHz Haswell

Gflop/s

matrix dimension
Test everything: run_tests.py

- Python script to run:
  - All testers
  - All possible options (left/right, lower/upper, ...)
  - Various size ranges (small, medium, large; square, tall, wide)
- Occasionally, tests fail innocuously
  - E.g., error = 1.1e-15 > tol = 1e-15
- Some experimental routines are known to fail
  - E.g., gegqr_gpu, geqr2x_gpu
  - See magma/BUGS.txt
- Running ALL tests can take > 24 hours
Test everything: run_tests.py

magma/testing> python ./run_tests.py *trsm --xsmall --small > trsm.txt

testing_strsm -SL -L -DN -c ok # left, lower, non-unit, [no-trans]
testing_dtrsm -SL -L -DN -c ok
testing_ctrsm -SL -L -DN -c ok
testing_ztrsm -SL -L -DN -c ok

testing_strsm -SL -L -DU -c ok # left, lower, unit, [no-trans]
testing_dtrsm -SL -L -DU -c ok
testing_ctrsm -SL -L -DU -c ok
testing_ztrsm -SL -L -DU -c ok

testing_strsm -SL -L -C -DN -c ok # left, lower, non-unit, conj-trans

testing_dtrsm -SL -L -C -DN -c ok

testing_ctrsm -SL -L -C -DN -c ok
testing_ztrsm -SL -L -C -DN -c ok
...

testing_strsm -SR -U -T -DU -c ok # right, upper, unit, trans
testing_dtrsm -SR -U -T -DU -c ok
testing_ctrsm -SR -U -T -DU -c ok
testing_ztrsm -SR -U -T -DU -c ok

************************************************************************************

summary

6240 tests in 192 commands passed
96 tests failed accuracy test
0 errors detected (crashes, CUDA errors, etc.)

routines with failures:
   testing_ctrsm --ngpu 2 -SL -L -C -DN -c
   testing_ctrsm --ngpu 2 -SL -L -C -DU -c

...
CUDA Software Stack

(Source: NVIDIA CUDA Programming Guide)
Grid of thread blocks
(blocks of the same dimension, grouped together to execute the same kernel)

Thread block
(a batch of threads with fast shared memory executes a kernel)

Sequential code launches asynchronously GPU kernels

---

```
// set the grid and thread configuration
Dim3 dimGrid(2,3);
Dim3 dimTBlock(3,5);

// Launch the device computation
MatVec<<<dimGrid, dimTBlock>>>(...);
```

```
__global__ void MatVec(...) {
    // Block index
    int bx = blockIdx.x;
    int by = blockIdx.y;

    // Thread index
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    ... 
}
```
One-sided factorizations

- LU, Cholesky, QR factorizations for solving linear systems
Execution trace

- **Panels** on CPU *(green)* and **set/get communication** *(brown)* overlapped with **trailing matrix updates** *(teal)* on GPU

- Goal to keep GPU busy all the time; CPU may idle

LU factorization (dgetrf), n = 20000
P100 GPU, 2 x 10-core 2.3 GHz Haswell

- Optimization: for LU, we transpose matrix on GPU so row-swaps are fast
Two-sided factorizations

- Hessenberg, tridiagonal, bidiagonal factorizations for eigenvalue and singular value problems

\[ y_i = A \nu_i \]

column \( a_i \)

Panel

Trailing matrix \( A = Q^T A Q \)
Small red rectangles (to overlap communication & computation) are of size 32 x 4 and are red by 32 x 2 threads.
A thread computes part of a row (16 values) of the C block

A thread computes a block of C (4 x 4 values in this case)

*Small red rectangles (to overlap communication & computation) are of size 32 x 4 and are red by 32 x 2 threads


- Add register blocking
- Parameterized for autotuning for particular size GEMMs and portability across GPUs
## CUDA vs. HIP

<table>
<thead>
<tr>
<th>CUDA</th>
<th>HIP</th>
</tr>
</thead>
<tbody>
<tr>
<td>cudaMalloc, cudaMemcpy, ... -&gt;</td>
<td>hipMalloc, hipMemcpy, ...</td>
</tr>
<tr>
<td>cuCadd, cuCsub, ... -&gt;</td>
<td>hipCadd, hipCsub, ...</td>
</tr>
<tr>
<td>func&lt;&lt;&lt;B, T, sz, st&gt;&gt;&gt;(a, b); -&gt;</td>
<td>hipLaunchKernelGGL(func, B, T, sz, st, a, b);</td>
</tr>
<tr>
<td>extern <strong>shared</strong> T name; -&gt;</td>
<td>HIP_DYNAMIC_SHARED(T, name); (*)</td>
</tr>
</tbody>
</table>

(*) Must be declared inside a function, and per function
There is a tool supplied by AMD, called ‘hipify’ (actually, there are 2; hipify-perl & hipify-clang)

‘hipify-perl’ performs basic substitution, i.e. ‘cudaMemcpy -> hipMemcpy’, and not much else. It has some trouble with macros & nested expressions

‘hipify-clang’ is built on top of LLVM’s parser/compiler interface, so can handle any amount of parentheses/calls/etc. However, it is harder to tweak & install, as ‘hipify-perl’ is just a perl script with regex

Let’s look at an example, which is performing a 2D box filter
CUDA Box Filter (device)

```c
__global__
void mykernel(int M, int N, const float* inp, float* out) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    int j = blockIdx.y*blockDim.y + threadIdx.y;
    if (i < 0 || i >= M || j < 0 || j >= N) return;

    const int k = 1;
    float sumr = 0.0f;

    for (int px = max(0, i-k); px <= min(M-1, i+k); ++px)
        for (int py = max(0, j-k); py <= min(N-1, j+k); ++py)
            sumr += inp[py * M + px];

    out[j * M + i] = sumr / ((2 * k + 1) * (2 * k + 1));
}
```
HIP Box Filter (device)

```cpp
__global__
void mykernel(int M, int N, const float* inp, float* out) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    int j = blockIdx.y*blockDim.y + threadIdx.y;
    if (i < 0 || i >= M || j < 0 || j >= N) return;

    const int k = 1;
    float sumr = 0.0f;

    for (int px = max(0, i-k); px <= min(M-1, i+k); ++px)
        for (int py = max(0, j-k); py <= min(N-1, j+k); ++py)
            sumr += inp[py * M + px];

    out[j * M + i] = sumr / ((2 * k + 1) * (2 * k + 1));
}
```
CUDA Box Filter (host)

```c
int main() {
    int M = 20, N = 8;
    float *x = (float*)malloc(M*N * sizeof(float));
    float *y = (float*)malloc(M*N * sizeof(float));
    for (int i = 0; i < M * N; ++i) x[i] = i;

    float *d_x, *d_y;
    cudaMalloc(&d_x, M*N * sizeof(float));
    cudaMalloc(&d_y, M*N * sizeof(float));

    cudaMemcpy(d_x, x, M*N * sizeof(float), cudaMemcpyHostToDevice);

    dim3 dimBlock(16, 16);
    dim3 dimGrid((M+dimBlock.x-1)/dimBlock.x, (N+dimBlock.y-1)/dimBlock.y);

    mykernel<<< dimGrid, dimBlock >>>(M, N, d_x, d_y);

    cudaMemcpy(y, d_y, M*N * sizeof(float), cudaMemcpyDeviceToHost);
}
```
HIP Box Filter (host)

```c
int main() {
    int M = 20, N = 8;
    float* x = (float*)malloc(M*N * sizeof(float));
    float* y = (float*)malloc(M*N * sizeof(float));
    for (int i = 0; i < M * N; ++i) x[i] = i;

    float *d_x, *d_y;
    hipMalloc(&d_x, M*N * sizeof(float));
    hipMalloc(&d_y, M*N * sizeof(float));

    hipMemcpy(d_x, x, M*N * sizeof(float), hipMemcpyHostToDevice);

    dim3 dimBlock(16, 16);
    dim3 dimGrid((M+dimBlock.x-1)/dimBlock.x, (N+dimBlock.y-1)/dimBlock.y);

    hipLaunchKernelGGL(mykernel, dim3(dimGrid), dim3(dimBlock), 0, 0, M, N, d_x, d_y);

    hipMemcpy(y, d_y, M*N * sizeof(float), hipMemcpyDeviceToHost);
}
```
MAGMA for AMD GPUs

- Easy functional and performance portability through use of BLAS
  - Provided about 2,000 routines
    - LAPACK, BLAS, Auxiliary, Batched
  - Ported through BLAS and auto-source translation tools
  - Added some BLAS still missing or underperforming in hipBLAS

- MAGMA has been ported before to OpenCL and Intel Xeon Phi

Testers/benchmarks for AMD GPUs
dgemm

DGEMM on the Mi50 GPU with ROCm 3.5

DGEMM on the Mi25 GPU with ROCm 3.5
Testers/benchmarks for AMD GPUs
dgemv and dsymv

DGEMV on the Mi50 GPU with ROCm 3.5

DSYMV on the Mi50 GPU with ROCm 3.5
Testers/benchmarks for AMD GPUs
dgemv and dsymv

**DSYRK on the Mi50 GPU with ROCm 3.5**

![Graph showing DSYRK performance](image1)

**DTRMM on the Mi50 GPU with ROCm 3.5**

![Graph showing DTRMM performance](image2)
Mixed Precision Solvers in MAGMA
Motivation on design of mixed precision algorithms

Table 4: Parameters for the IEEE FP16, FP32, and FP64 arithmetic precisions, and their respective peak performances on NVIDIA V100 and AMD MI100 GPUs. “Range” denotes the order of magnitude of the smallest subnormal ($x_{min,s}$), and largest and smallest positive normalized floating-point numbers. The peak 16-bit performances reported* by vendors vary depending on the instructions and special hardware acceleration used, e.g., the peak performance of 125 Tflop/s for Nvidia V100 uses FP16-TC (tensor cores) with inputs FP16, while the outputs and the computations are performed in full (FP32) precision.

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>Size (bits)</th>
<th>$x_{min,s}$</th>
<th>$x_{min}$</th>
<th>$x_{max}$</th>
<th>Unit roundoff</th>
<th>Peak Tflop/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFloat16</td>
<td>16</td>
<td>$9.2 \times 10^{-41}$</td>
<td>$1.2 \times 10^{-38}$</td>
<td>$3.4 \times 10^{38}$</td>
<td>$3.9 \times 10^{-3}$</td>
<td>N/A</td>
</tr>
<tr>
<td>FP16</td>
<td>16</td>
<td>$6.0 \times 10^{-8}$</td>
<td>$6.1 \times 10^{-5}$</td>
<td>$6.6 \times 10^{4}$</td>
<td>$4.9 \times 10^{-4}$</td>
<td>125*</td>
</tr>
<tr>
<td>FP32</td>
<td>32</td>
<td>$1.4 \times 10^{-45}$</td>
<td>$1.2 \times 10^{-38}$</td>
<td>$3.4 \times 10^{38}$</td>
<td>$6.0 \times 10^{-8}$</td>
<td>15.7</td>
</tr>
<tr>
<td>FP64</td>
<td>64</td>
<td>$4.9 \times 10^{-324}$</td>
<td>$2.2 \times 10^{-308}$</td>
<td>$1.8 \times 10^{308}$</td>
<td>$1.1 \times 10^{-16}$</td>
<td>7.8</td>
</tr>
</tbody>
</table>
Mixed precision iterative refinement

- Iterative refinement for dense systems, \( Ax = b \), can work this way.

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

\[ \text{WHILE } \| r \| \text{ not small enough} \]
\[ z = L \backslash (U \backslash r) \]
\[ x = x + z \]
\[ r = b - Ax \]

\[ \text{END} \]

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

- Requires extra storage, total is 1.5 times normal;
- \( O(n^3) \) work is done in lower precision
- \( O(n^2) \) work is done in high precision
- Problems if the matrix is ill-conditioned in sp; \( O(10^8) \)
Mixed precision iterative refinement solvers

- Implemented mixed-precision iterative refinement solvers based on LU, Cholesky, and QR factorizations in MAGMA since 2010
- Algorithms and stopping criteria follow LAPACK (Langou et al. 2006)

\[ \frac{\|Ax - b\|}{\|A\| \|x\|} < \varepsilon^w \sqrt{n}, \text{ where } \varepsilon^w \text{ is the machine precision at the working precision} \]
GPU Tensor Cores – accelerated FP16 matrix-multiply-and-accumulate units

- Up to 120 teraFLOP/s on NVIDIA Volta V100 GPUs
Tensor Core Accelerated IRS
solving linear system $Ax = b$

For $s = 0, nb, .. N$

1. panel factorize
2. update trailing matrix

LU factorization requires $O(n^3)$
most of the operations are spent in GEMM

Panel

step 1  step 2  step 3  step 4

panel

update
Tensor Core Accelerated IRS Motivation

Study of the Matrix Matrix multiplication kernel on Nvidia V100

- \texttt{dgemm} achieve about 6.4 Tflop/s
- \texttt{sgemm} achieve about 14 Tflop/s
- \texttt{hgemm} achieve about 27 Tflop/s
- Tensor cores \texttt{gemm} reach about 85 Tflop/s

\textbf{Rank-k GEMM needed by LU does not perform as well as square but still OK}

\textbf{Matrix matrix multiplication GEMM}

\begin{align*}
    C &= \alpha A \times B + \beta C
\end{align*}
Mixed-precision factorizations

**Input:** $A$ in precision $\mathbf{u}^h = \text{FP32}$

$\mathbf{u}^f \leq \mathbf{u}^h$

for $P_i \in \{P_1, P_2, \ldots, P_n\}$ do

1. PanelFactorize $P_i$ in precision $\mathbf{u}^h$ (e.g., FP32)
2. Triangular solve $T_i$ in precision $\mathbf{u}^h$ (e.g., FP32)
3. convert $P_i^h \rightarrow P_i^f$ (e.g., from precision FP32 to FP16)
4. convert $T_i^h \rightarrow T_i^f$ (e.g., from precision FP32 to FP16)
5. TrailingMatrixUpdate $A_i^h = A_i^h - P_i^f T_i^f$ (e.g., $A_i^{FP32} = A_i^{FP32} - P_i^{FP16} T_i^{FP16}$) using tensor cores

- A is never converted to FP16
- Sensitive for accuracy panel factorization and triangular solve are done in FP32
- For many problems of interest we get about two more digits of accuracy compared to factorizations using fixed FP16 arithmetic!
Mixed-precision HGEMM accuracy

**Figure 3:** Forward error of HGEMM with respect to MKL SGEMM ($C = \alpha AB + \beta C$). Results are shown for square sizes using cuBLAS 9.1 and MKL 2018.1. The forward error is computed as $\frac{||R_{\text{cuBLAS}} - R_{\text{MKL}}||_F}{\sqrt{k+2||A||_F||B||_F+2||\beta||C||_F}}$, where $k$ is equal to the matrix size.
Leveraging Half Precision in HPC on V100

Motivation

Study of the LU factorization algorithm on Nvidia V100

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

$A \ x = b$

$Lx = y$

then

$Ux = y$
Tensor Core Accelerated IRS solving linear system $Ax = b$

Performance Behavior

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.

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

Figure 4: Performance in Tflop/s for the three linear solver

Investigating Half Precision Arithmetic to Accelerate Dense Linear System Solvers ScalA17, November 12–17, 2017, Denver, CO, USA
Problem generated with an arithmetic distribution of the singular values and positive eigenvalues.

Performance Behavior

- 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

---

$\sigma_i = 1 - \left(\frac{i-1}{n-1}\right)(1 - \frac{1}{\text{cond}})$ and positive eigenvalues.
Batched Linear Algebra
Many applications need LA on many small matrices

Data Analytics and associated with it Linear Algebra on small LA problems are needed in many applications:

- Machine learning,
- Data mining,
- High-order FEM,
- Numerical LA,
- Graph analysis,
- Neuroscience,
- Astrophysics,
- Quantum chemistry,
- Multi-physics problems,
- Signal processing, etc.

Applications using high-order FEM

- Matrix-free basis evaluation needs efficient tensor contractions,

\[ C_{i1,i2,i3} = \sum_k A_{k,i1} B_{k,i2,i3} \]

- Within ECP CEED Project, designed MAGMA batched methods to split the computation in many small high-intensity GEMMs, grouped together (batched) for efficient execution:

\[ \text{Batch}_{i3} \{ C_{i3} = A^T B_{i3}, \text{ for range of } i3 \} \]
Batched Linear Algebra Computations

Non-batched computation
- Data parallel computation (e.g., over independent matrices $A_i$)
- Loop over the matrices one by one and compute in parallel

```
for (int i=0; i<batchcount; i++)
    la_computation($A_i$, ...)
```

Call a numerical library (e.g., BLAS, LAPACK, etc.) to compute $A_i$ in parallel
Batched Linear Algebra Computations

Non-batched computation
- Data parallel computation (e.g., over independent matrices $A_i$)
- Loop over the matrices one by one and compute in parallel
  (since matrices are of small sizes there is not enough work for all the cores, so expect low performance as well as threads contention).

```c
for (int i=0; i<batchcount; i++)
  dgemm(...)
```
Batched Linear Algebra Computations

Non-batched computation

- Data parallel computation (e.g., over independent matrices $A_i$)
- Loop over the matrices one by one and compute in parallel (since matrices are of small sizes there is not enough work for all the cores, so expect low performance as well as threads contention).

```
for (int i=0; i<batchcount; i++)
    la_computation(A_i, ...)
```

Batched computation

- Given an interface as a single Batched routine
- Distribute all the matrices over the available resources by assigning a matrix to each group of core/TB to operate on it independently.

```
Batched_la_computation(A, batchcount, ...)
```
Batched Linear Algebra Computations

Non-batched computation
- Data parallel computation (e.g., over independent matrices $A_i$)
- Loop over the matrices one by one and compute in parallel (since matrices are of small sizes there is not enough work for all the cores, so expect low performance as well as threads contention).

Batched computation
- Given an interface as a single Batched routine
- Distribute all the matrices over the available resources by assigning a matrix to each group of core/TB to operate on it independently

Batched \_dgemm (...)
Batched Linear Algebra Computations

**Non-batched computation**
- Data parallel computation (e.g., over independent matrices $A_i$)
- Loop over the matrices one by one and compute in parallel (since matrices are of small sizes there is not enough work for all the cores, so expect low performance as well as threads contention).

```c
for (int i=0; i<batchcount; i++)
    la_computation(A_i, ...)
```

**Batched computation**
- Given an interface as a single Batched routine
- Distribute all the matrices over the available resources by assigning a matrix to each group of core/TB to operate on it independently

```c
Batched_la_computation(A,batchcount,...)
```

---

**Performance Comparison**

- **Matrix sizes (fixed) in the batch**
  - **Batch size 1,000**
  - **Batch size 300**
  - **Batch size 50**

- **Nvidia V100 GPU**
  - **small sizes**
    - Batch $dgemm$ BLAS 3: 19X
    - Standard $dgemm$ BLAS 3: $1.4X$
  - **medium sizes**
  - **Large sizes**

- **Switch to non-batch**
Batched Linear Algebra Computations

Non-batched computation
- Data parallel computation (e.g., over independent matrices $A_i$)
- Loop over the matrices one by one and compute in parallel (since matrices are of small sizes there is not enough work for all the cores, so expect low performance as well as threads contention).

Batched computation
- Given an interface as a single Batched routine
- Distribute all the matrices over the available resources by assigning a matrix to each group of core/TB to operate on it independently.

Some simple examples from BLAS

```c
// Batch of adding vector entries
// dy(:) += da * dx(:)
for (int i=0; i<n; i++)
    dy[i] = dy[i] + da * dx[i];

// Level 1 BLAS daxpy routine
daxpy(n, da, dx, 1, dy, 1);
```
Batched Linear Algebra Computations

Non-batched computation
- Data parallel computation (e.g., over independent matrices $A_i$)
- Loop over the matrices one by one and compute in parallel (since matrices are of small sizes there is not enough work for all the cores, so expect low performance as well as threads contention).

Batched computation
- Given an interface as a single Batched routine
- Distribute all the matrices over the available resources by assigning a matrix to each group of core/TB to operate on it independently

Some simple examples from BLAS

// Batch of Level 1 BLAS dot-prod.
// $y_i = \alpha \cdot A_i \cdot x + \beta \cdot y_i$
for (int i=0; i<m; i++)
    double res = 0.0;
    for (int j=0; j<n; j++)
        res += A(i,j) * x[j];
    y[i] = alpha * res + beta * y[i];

// Level 2 BLAS dgemv routine
dgemv('N', m, n,
    alpha, A, lda, x, 1,
    beta, y, 1);
Batched Linear Algebra Computations

Non-batched computation
- Data parallel computation (e.g., over independent matrices $A_i$)
- Loop over the matrices one by one and compute in parallel (since matrices are of small sizes there is not enough work for all the cores, so expect low performance as well as threads contention).

Batched computation
- Given an interface as a single Batched routine
- Distribute all the matrices over the available resources by assigning a matrix to each group of core/TB to operate on it independently

Some simple examples from BLAS

```c
// Batch of Level 2 BLAS dgemv.
// $C(:,i) = \alpha \cdot A B_i + \beta \cdot C(:,i)$
for (int i=0; i<n; i++)
    dgemv('N', m, k,
        alpha, A, lda, B+i*ldb, 1,
        beta, C+i*ldc, 1);

// Level 3 BLAS dgemm routine
dgemm('N', 'N', m, n, k,
    alpha, A, lda, B, ldb,
    beta, C, ldc);
```
Batched Linear Algebra Computations

Non-batched computation
- Data parallel computation (e.g., over independent matrices $A_i$)
- Loop over the matrices one by one and compute in parallel (since matrices are of small sizes there is not enough work for all the cores, so expect low performance as well as threads contention).

Batched computation
- Given an interface as a single Batched routine
- Distribute all the matrices over the available resources by assigning a matrix to each group of core/TB to operate on it independently

Some simple examples from BLAS

// Batch of rank-k Level3 BLAS
// $C_{ij} = \alpha A_i B_j + \beta C_{ij}$
for (int $i=0; i<n; i++$)
  for (int $j=0; j<m; j++$)
    $C_{ij} = A_i B_j + \beta C_{ij}$

// Level 3 BLAS dgemm routine
Batched_dgemm(...);

or, in this case, one big dgemm

dgemm( ... );
Batched Linear Algebra Computations

Non-batched computation
- Data parallel computation (e.g., over independent matrices $A_i$)
- Loop over the matrices one by one and compute in parallel (since matrices are of small sizes there is not enough work for all the cores, so expect low performance as well as threads contention).

Batched computation
- Given an interface as a single Batched routine
- Distribute all the matrices over the available resources by assigning a matrix to each group of core/TB to operate on it independently

Example from tensor computations

```c
// Tensor computations as batches of LA, e.g., batched gemms
// C_{i_1,i_2,i_3} = \sum_k A_{k,i_1} B_{k,i_2,i_3}

for (int i3 = 0; i < batch; i++)
    C_{i3} = A^T B_{i3}

// Batched dgemms
Batched_dgemm(‘T’, ‘N’, m, n, k, 1, A, lda, B, ldb, 0, C, ldc, batch);
```
Batched Linear Algebra Computations

**Non-batched computation**
- Data parallel computation (e.g., over independent matrices $A_i$)
- Loop over the matrices one by one and compute in parallel (since matrices are of small sizes there is not enough work for all the cores, so expect low performance as well as threads contention).

**Batched computation**
- Given an interface as a single Batched routine
- Distribute all the matrices over the available resources by assigning a matrix to each group of core/TB to operate on it independently

**Example from CNNs**

```c
// Apply convolutions with filters F
// on batch of images I
for (int i=0; i<batch; i++)
    for (int x=0; x<xdim-1; x++)
        for (int y=0; y<ydim-1; y++)
            R[i][x][y] = 0;
    for (int l=0; l<m-1; l++)
        for (int k=0; k<n-1; k++)
            R[i][x][y]+=F[l][k]*I[i][x][y];
```

// Batched dgemms
```
Unfold_data(...);
Batched_dgemm(..., batch);
Put_results_back(...);
```
Example from DNN Accelerate DNNs through GEMMs of various sizes

- Convolutions can be accelerated in various ways:
  - Unfold and GEMM
  - FFT
  - Winograd minimal filtering – reduction to batched GEMMs

<table>
<thead>
<tr>
<th>Layer</th>
<th>m</th>
<th>n</th>
<th>k</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12544</td>
<td>64</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>12544</td>
<td>64</td>
<td>64</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>12544</td>
<td>128</td>
<td>64</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>12544</td>
<td>128</td>
<td>128</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6272</td>
<td>256</td>
<td>128</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>6272</td>
<td>256</td>
<td>256</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>6272</td>
<td>256</td>
<td>256</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>3136</td>
<td>512</td>
<td>256</td>
<td>16</td>
</tr>
<tr>
<td>9</td>
<td>3136</td>
<td>512</td>
<td>512</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>3136</td>
<td>512</td>
<td>512</td>
<td>16</td>
</tr>
<tr>
<td>11</td>
<td>784</td>
<td>512</td>
<td>512</td>
<td>16</td>
</tr>
<tr>
<td>12</td>
<td>784</td>
<td>512</td>
<td>512</td>
<td>16</td>
</tr>
<tr>
<td>13</td>
<td>784</td>
<td>512</td>
<td>512</td>
<td>16</td>
</tr>
</tbody>
</table>

- Use autotuning to handle complexity of tuning

Required matrix-matrix products of various sizes, including batched GEMMs
Batched LA Interface Design
Large scientific community effort (industry, national labs, academia) to define Batched BLAS and LAPACK standard


Batched LA Interface Design

- Single interface and proposed API standard for **Batched BLAS and LAPACK** for functional and performance portability across architectures
- Main design issues include
  - **Data layout for single problem**
    (row/column major, symmetric, band, structure, sparse formats ...)
  - **Data layout interface for the batched problems**
    (groups, variable/constant size, one matrix after another or strided, etc.)
  - **Supported precisions**

Batched GEMM (general matrix-matrix multiply) C language declarations in multiple precisions

```c
int BLAS_gemm_batched_r16(BLAS_layout layout, BLAS_op * A_trans, BLAS_op * B_trans, int * m, int * n, int * k, _Float16 * alpha, _Float16 * beta, _Float16 * A, int * A_ld, _Float16 * B, int * B_ld, _Float16 * C, int * C_ld, int * group_count, int * group_sizes, int * info);
int BLAS_gemm_batched_c16(BLAS_layout layout, BLAS_op * A_trans, BLAS_op * B_trans, int * m, int * n, int * k, _Complex_Float16 * alpha, _Complex_Float16 * beta, _Complex_Float16 * A, int * A_ld, _Complex_Float16 * B, int * B_ld, _Complex_Float16 * C, int * C_ld, int * group_count, int * group_sizes, int * info);
int BLAS_gemm_batched_r32(BLAS_layout layout, BLAS_op * A_trans, BLAS_op * B_trans, int * m, int * n, int * k, float * alpha, float * beta, float * A, int * A_ld, float * B, int * B_ld, float * C, int * C_ld, int * group_count, int * group_sizes, int * info);
int BLAS_gemm_batched_c32(BLAS_layout layout, BLAS_op * A_trans, BLAS_op * B_trans, int * m, int * n, int * k, _Complex Float * alpha, _Complex Float * beta, _Complex Float * A, int * A_ld, _Complex Float * B, int * B_ld, _Complex Float * C, int * C_ld, int * group_count, int * group_sizes, int * info);
int BLAS_gemm_batched_r64(BLAS_layout layout, BLAS_op * A_trans, BLAS_op * B_trans, int * m, int * n, int * k, double * alpha, double * beta, double * A, int * A_ld, double * B, int * B_ld, double * C, int * C_ld, int * group_count, int * group_sizes, int * info);
int BLAS_gemm_batched_c64(BLAS_layout layout, BLAS_op * A_trans, BLAS_op * B_trans, int * m, int * n, int * k, _Complex Double * alpha, _Complex Double * beta, _Complex Double * A, int * A_ld, _Complex Double * B, int * B_ld, _Complex Double * C, int * C_ld, int * group_count, int * group_sizes, int * info);
int BLAS_gemm_batched_r128(BLAS_layout layout, BLAS_op * A_trans, BLAS_op * B_trans, int * m, int * n, int * k, _Float128 * alpha, _Float128 * beta, _Float128 * A, int * A_ld, _Float128 * B, int * B_ld, _Float128 * C, int * C_ld, int * group_count, int * group_sizes, int * info);
int BLAS_gemm_batched_c128(BLAS_layout layout, BLAS_op * A_trans, BLAS_op * B_trans, int * m, int * n, int * k, _Complex_Float128 * alpha, _Complex_Float128 * beta, _Complex_Float128 * A, int * A_ld, _Complex_Float128 * B, int * B_ld, _Complex_Float128 * C, int * C_ld, int * group_count, int * group_sizes, int * info);
```
MAGMA Batched Computations

**MAGMA Batched**

Batched Factorization of a set of small matrices in parallel

Numerous applications require factorization of many small matrices

- Deep learning
- Structural mechanics
- Astrophysics
- Sparse direct solvers
- High-order FEM simulations

**ROUTINES**

- LU, QR, and Cholesky
- Solvers and matrix inversion
- All BLAS 3 (fixed + variable)
- SYMV, GEMV (fixed + variable)

**APPLICATIONS / LIBRARIES**

- MAGMA Batched Framework & Abstractions
  - Autotuning
  - Inlining & Code Generation
  - Kernel Design
  - Algorithmic Variants

**DEVICES**

- CPUs
- GPUs
- Coprocessors (KNC/KNL)
API for Batched BLAS in MAGMA

Batch of fixed-size problems:

```c
extern "C" void
magmablas_dgemm_batched( magma_trans_t transA, magma_trans_t transB,
                          magma_int_t m, magma_int_t n, magma_int_t k,
                          double alpha,
                          double const * const * dA_array, magma_int_t ldda,
                          double const * const * dB_array, magma_int_t lddb,
                          double beta,
                          double **dC_array, magma_int_t lddc,
                          magma_int_t batchCount, magma_queue_t queue )
```

Batch of variable-size problems:

```c
extern "C" void
magmablas_dgemm_vbatched( magma_trans_t transA, magma_trans_t transB,
                           magma_int_t* m, magma_int_t* n, magma_int_t* k,
                           double alpha,
                           double const * const * dA_array, magma_int_t* ldda,
                           double const * const * dB_array, magma_int_t* lddb,
                           double beta,
                           double **dC_array, magma_int_t* lddc,
                           magma_int_t batchCount, magma_queue_t queue )
```
API for Batched LAPACK in MAGMA

Batch of fixed-size problems:

```c
extern "C" magma_int_t
magma_zpotrf_batched(
    magma_uplo_t uplo, magma_int_t n,
    magmaDoubleComplex **dA_array, magma_int_t ldda,
    magma_int_t *info_array, magma_int_t batchCount,
    magma_queue_t queue)
```

Batch of variable-size problems:

```c
extern "C" magma_int_t
magma_zpotrf_vbatched(
    magma_uplo_t uplo, magma_int_t *n,
    magmaDoubleComplex **dA_array, magma_int_t *ldda,
    magma_int_t *info_array, magma_int_t batchCount,
    magma_queue_t queue)
```
Implementation of Batched LA in CUDA

- Grid of thread blocks
  (blocks of the same dimension, grouped together to execute the same kernel)
- Thread block
  (a batch of threads with fast shared memory executes a kernel)
- Sequential code launches asynchronously GPU kernels

```c
void LA_computation(double *A, ...) {
  // set the grid and thread configuration
  Dim3 dimGrid(2,3);
  Dim3 dimTBlock(3,5);

  // Launch the device computation
  LA_computation_kernel<<<dimGrid, dimTBlock>>>(A, ..);
}

__global__ void LA_computation_kernel(double *A, ...) {
  // Get Block index that this TB is computing
  int bx = blockIdx.x;
  int by = blockIdx.y;

  // Get the Thread index that this thread is computing
  int tx = threadIdx.x;
  int ty = threadIdx.y;

  LA_device(A, bx, by, tx, ty, ...);
```

(Source: NVIDIA CUDA Programming Guide)
Implementation of Batched LA in CUDA

- Grid of thread blocks
  (blocks of the same dimension, grouped together to execute the same kernel)

- Thread block
  (a batch of threads with fast shared memory executes a kernel)

- Sequential code launches asynchronously GPU kernels

Assume we want to run LA_computation on a batch of independent problems

One way is to redesign LA_computation by running not just one 2 x 3 Grid but on a 2 x 3 x batch Grid where each sub-problem is done on a 2 x 3 Grid

```c
void LA_computation(double *A, ... ) {
    // set the grid and thread configuration
    Dim3 dimGrid(2,3);
    Dim3 dimTBlock(3,5);

    // Launch the device computation
    LA_computation_kernel<<<dimGrid, dimTBlock>>>(A, ...);
}

__global__ void LA_computation_kernel( . . . ) {
    // Get Block index that this TB is computing
    int bx = blockIdx.x;
    int by = blockIdx.y;

    // Get the Thread index that this thread is computing
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    LA_device(A, bx, by, tx, ty, ...);
}
```

```c
void Batched_LA_computation(double **A, ..., int batchCount ) {
    // set the grid and thread configuration
    Dim3 dimGrid(2,3, batchCount);
    Dim3 dimTBlock(3,5);

    // Launch the device computation
    Batched_LA_computation_kernel<<<dimGrid, dimTBlock>>>(A, ...);
}

__global__ void Batched_LA_computation_kernel( . . . ) {
    // Get Block index that this TB is computing
    int bx = blockIdx.x, by = blockIdx.y;
    int bz = blockIdx.z;

    // Get the Thread index that this thread is computing
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    LA_device( A[bz], bx, by, tx, ty, ...);
}
```
Other considerations in implementing fast batched LA

- Can not rely on single implementation to get portable high-performance
- Multiple kernels are designed for various scenarios and parameterized for subsequent autotuning
- **Fused kernels**

Optimize communications:

\[ \text{batch}\{ e=0..nelems\} \{ B_e^T D_e \cdot (B_e A_e B_e^T) B_e \} \]

VS.

\[ \begin{align*} 
\text{batch}\{ e=0..nelems\} & \{ C_e = A_e B_e^T \} ; \\
\text{batch}\{ e=0..nelems\} & \{ C_e = B_e C_e \} ; \\
\text{batch}\{ e=0..nelems\} & \{ C_e = D_e \cdot C_e \} ; \\
\text{batch}\{ e=0..nelems\} & \{ C_e = C_e B_e \} ; \\
\text{batch}\{ e=0..nelems\} & \{ C_e = B_e^T C_e \} ; 
\end{align*} \]

**Performance improvements in Batched LU through various levels of kernel fusion**

(most recent developments in MAGMA by A. Abdelfattah)

How to code fused kernels:
- JIT (nvtrc for NVIDIA GPUs)
- Constructed as DAGs
- Sequence of device BLAS routines
How to implement fast batched DLA?

**Problem sizes influence algorithms & optimization techniques**

- Can not rely on single implementation to get portable high-performance

Kernels are designed various scenarios and parameterized for autotuning framework to find “best” performing kernels

**Optimizing GEMM’s: Kernel design**

- Reading/writing the elements is based on the TB size (# threads) and so is an extra parameter.
- Also it could be different for A, B and C
GEMM in MAGMA


- Add register blocking
- Parameterized for autotuning for particular size GEMMs and portability across GPUs

*Small* red rectangles (to overlap communication & computation) are of size 32 x 4 and are red by 32 x 2 threads

A thread computes part of a row (16 values) of the C block

A thread computes a block of C (4 x 4 values in this case)
1) **Kernel variants:** performance parameters are exposed through a templated kernel interface

```cpp
template< typename T, int DIM_X, int DIM_Y,
         int BLK_M, int BLK_N, int BLK_K,
         int DIM_XA, int DIM_YA, int DIM_XB, int DIM_YB,
         int THR_M, int THR_N, int CONJA, int CONJB >
static __device__ void tensor_template_device_gemm_nn( int M, int N, int K, ...
```

2) **CPU interfaces that call the GPU kernels as a Batched computation**

```cpp
void tensor_template_batched_gemm_nn( int m, int n, int k, ... ) {
...
    tensor_template_device_gemm__nn<T, DIM_X, DIM_Y, ... >>><dimGrid, dimBlock, 0, queue>>>(m, n, k,...);
}
```

3) **Python scripts that generate the search space for the parameters DIM_X, DIM_Y ...**

<table>
<thead>
<tr>
<th>index,</th>
<th>DIM_X,</th>
<th>DIM_Y,</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>#define NN_V_0</td>
<td>4,</td>
<td>8,</td>
<td>24,</td>
</tr>
<tr>
<td></td>
<td>8,</td>
<td>1,</td>
<td>4,</td>
</tr>
<tr>
<td></td>
<td>8,</td>
<td>4,</td>
<td>8,</td>
</tr>
<tr>
<td>#define NN_V_1</td>
<td>4,</td>
<td>8,</td>
<td>32,</td>
</tr>
<tr>
<td></td>
<td>8,</td>
<td>1,</td>
<td>4,</td>
</tr>
<tr>
<td></td>
<td>8,</td>
<td>4,</td>
<td>8,</td>
</tr>
<tr>
<td>#define NN_V_2</td>
<td>4,</td>
<td>8,</td>
<td>40,</td>
</tr>
<tr>
<td></td>
<td>8,</td>
<td>1,</td>
<td>4,</td>
</tr>
<tr>
<td></td>
<td>8,</td>
<td>4,</td>
<td>8,</td>
</tr>
</tbody>
</table>

4) **Scripts that run all versions in the search space, analyze the results, and return the best combination of parameters, which is stored in the library for subsequent use.**
Performance comparison of tensor contraction versions using batched $C = \alpha AB + \beta C$ on 100,000 square matrices of size $n$ on a K40c GPU and 16 cores of Intel Xeon E5-2670, 2.60 GHz CPUs.

Effect of a Thread Block Concurrency (tbc) techniques where several DGEMMs are performed on one TB simultaneously.
Performance ...

**Batched DGEMM on CPUs**

Intel Xeon E5-2650 v3 (Haswell), 10 cores

2 x Intel Xeon E5-2650 v3 (Haswell), 20 cores

The End

• The End!