S7728 - MAGMA Tensors and Batched Computing for Accelerating Applications on GPUs

Stan Tomov  - Research Director, UTK
Azzam Haidar  - Research Scientist, UTK

Abstract: Learn how to accelerate your machine learning, data mining, and other algorithms through fast matrix and tensor operations on GPUs. There's an increasing demand for accelerated independent computations on tensors and many small matrices. Although common, these workloads cannot be efficiently executed using standard linear algebra libraries. To fill the gap, we developed the MAGMA Batched library that achieves dramatically better performance by repetitively executing the small operations in "batches." We'll describe a methodology on how to develop high-performance BLAS, SVD, factorizations, and solvers for both large- and small-batched matrices. We'll also present the current state-of-the-art implementations and community efforts to standardize an API that extends BLAS for Batched computations.
MAGMA Tensors and Batched Computing for Accelerating Applications on GPUs

Stan Tomov and Azzam Haidar
Innovative Computing Laboratory
Department of Electrical Engineering and Computer Science
University of Tennessee, Knoxville

In collaboration with:
LLNL, Livermore, CA, USA
University of Manchester, Manchester, UK
University of Paris-Sud, France

GTC 2017
San Jose, CA
May 8—11, 2017
Outline

- Introduction
- MAGMA library
  - Numerical Linear Algebra (NLA) for large problems
  - NLA for applications that need small problems
- MAGMA Tensor contraction computations
- MAGMA Batched Computing
- MAGMA-DNN NLA backend for DNN
- Algorithms and optimization techniques
- Conclusions
Wide range of Applications depend on Numerical Linear Algebra (NLA) Libraries

- Airplane wing design,
- Quantum chemistry,
- Geophysical flows,
- Stealth aircraft,
- Diffusion of solid bodies in a liquid,
- Adaptive mesh refinement,
- Computational materials research,
- Deep learning in neural networks,
- Stochastic simulation,
- Massively parallel data mining,
- ...

[Images and diagrams related to applications listed]
Numerical Linear Algebra (NLA) in Applications

**NLA is the backend** that accelerates a wide variety of science and engineering applications:

- **Linear system**
  - 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 $||Ax - b||$
  - Convex optimization, 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

- **Singular Value Decomposition (SVD):**
  - $A = U \Sigma V^*$
  - 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.

- **LA is crucial to the development of sparse solvers**
Numerical Linear Algebra (NLA) in Applications

**NLA is the backend** that accelerates a wide variety of science and engineering applications:

- For **big** NLA problems (BLAS, convolutions, SVD, linear system solvers, etc.)

In contemporary libraries:
- BLAS
- LAPACK
- ScaLAPACK
- **MAGMA** (for GPUs)
Numerical Linear Algebra (NLA) in Applications

**NLA is the backend** that accelerates a wide variety of science and engineering applications:

- **For big NLA problems**
  (BLAS, convolutions, SVD, linear system solvers, etc.)

- **Numerous important applications need NLA for small problems**
  - Machine learning / DNNs
  - Data mining / analytics
  - High-order FEM,
  - Graph analysis,
  - Neuroscience,
  - Astrophysics,
  - Quantum chemistry,
  - Signal processing, and more

In contemporary libraries:
- BLAS
- LAPACK
- ScaLAPACK
- **MAGMA** (for GPUs)

Where data can be multidimensional / relational
Numerical Linear Algebra (NLA) in Applications

**NLA is the backend** that accelerates a wide variety of science and engineering applications:

- **For big NLA problems** (BLAS, convolutions, SVD, linear system solvers, etc.)

  ![Large matrices](image)

  In contemporary libraries:
  - BLAS
  - LAPACK
  - ScaLAPACK
  - **MAGMA** (for GPUs)

- Adding in MAGMA application backends for **small problems**
  - Machine learning / DNNs
  - Data mining / analytics
  - High-order FEM,
  - Graph analysis,
  - Neuroscience,
  - Astrophysics,
  - Quantum chemistry,
  - Signal processing, and more

  ![Small matrices / tensors](image)

  Fixed-size batches
  Variable-size batches
  Dynamic batches
  Tensors
Key Features of MAGMA 2.2

**TASK-BASED ALGORITHMS**

MAGMA uses task-based algorithms where the computation is split into tasks of varying granularity and their execution scheduled over the hardware components. Scheduling can be static or dynamic. In either case, small non-parallelizable tasks, often on the critical path, are scheduled on the CPU, and larger more parallelizable ones, often Level 3 BLAS, are scheduled on the GPUs.

**PERFORMANCE & ENERGY EFFICIENCY**

MAGMA LU factorization in double precision arithmetic

<table>
<thead>
<tr>
<th>CPU</th>
<th>Intel Xeon E5-2650 v3 (Haswell)</th>
<th>2x10 cores @ 2.30 GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>K40</td>
<td>NVIDIA K40 GPU</td>
<td>15 MP x 192 @ 0.88 GHz</td>
</tr>
<tr>
<td>P100</td>
<td>NVIDIA Pascal GPU</td>
<td>56 MP x 64 @ 1.19 GHz</td>
</tr>
</tbody>
</table>

![Graph showing performance GFLOP/s vs. matrix size N x N](image)

- Green dots: P100
- Red dots: 2 K40
- Orange dots: 1 K40
- Blue dots: CPU

![Bar chart showing GFLOPs/Watt](image)

- Blue bar: CPU
- Orange bar: K40
- Green bar: P100

---

**GPU TECHNOLOGY CONFERENCE**

**CEED**

EXASCALE DISCRETIZATIONS

**ICL INNOVATIVE COMPUTING LABORATORY**

Department of Electrical Engineering and Computer Science
MAGMA — designed to use Level 3 BLAS as much as possible

Nvidia P100, 1.19 GHz, Peak DP = 4700 Gflop/s

**Matrix size (N), vector size (NxN)**

- **C = C + A*B**
  - 4503 Gflop/s

- **y = y + A*x**
  - 145 Gflop/s

- **y = α * x + y**
  - 52 Gflop/s

Nvidia P100
The theoretical peak double precision is 4700 Gflop/s
CUDA version 8.0
MAGMA Algorithms (influenced by hardware trend)
Hybrid (using CPU + GPUs) and/vs. GPU-only

MAGMA LU factorization in double precision arithmetic

CPU Intel Xeon E5-2650 v3 (Haswell)
2x10 cores @ 2.30 GHz

K40 NVIDIA K40 GPU
15 MP x 192 @ 0.88 GHz

P100 NVIDIA Pascal GPU
56 MP x 64 @ 1.19 GHz

magma native (opt)
magma native
magma hybrid
cusolver
mkl-knl
mkl-cpu
**MAGMA Algorithms** (influenced by hardware trend)

Mixed-precision iterative refinement

Solving general dense linear systems using mixed precision iterative refinement

![Graph showing performance comparison between different algorithms and platforms.](image)

**Graph Key:**
- CPOSV
- ZCPOSV
- ZPOSV

**Platform Details:**
- **GPU:** TITAN X (3,072 CUDA cores @ 1.076 GHz)
  - Z/C GEMM peak ~ 190 / 5,600 GFlop/s
- **CPU:** Intel Xeon X5660@2.80GHz (2 x 6 cores)

**Mixed Precision Iterative Refinement Formula:**

$$x_{n+1} = x_i + (LU_{sp})^T P (b - A x_i)$$

**Text:**

- Direct solvers
  - Factor and solve in working precision
- Mixed Precision Iterative Refinement
  - Factor in single (i.e. the bulk of the computation in fast arithmetic) and use it as preconditioner in simple double precision iteration, e.g.
Support for various Batched and/or Tensor contraction routines

e.g., Convolutional Neural Networks (CNNs) used in computer vision

Key computation is convolution of Filter \( F_i \) (feature detector) and input image \( D \) (data):

\[
O_{n,k} = \sum_i D_{k,i} F_{n,i}
\]

- Plenty of parallelism; small operations that must be batched
- With data “reshape” the computation can be transformed into a batched GEMM (for efficiency; among other approaches)
Lagrangian Hydrodynamics in the BLAST code\cite{1}

On semi-discrete level our method can be written as

\[ \frac{d\mathbf{v}}{dt} = -\mathbf{M}_v^{-1}\mathbf{F} \cdot \mathbf{1} \]

\[ \frac{de}{dt} = \mathbf{M}_e^{-1}\mathbf{F}^T \cdot \mathbf{v} \]

\[ \frac{dx}{dt} = \mathbf{v} \]

where \(\mathbf{v}, e, \) and \(x\) are the unknown velocity, specific internal energy, and grid position, respectively; \(\mathbf{M}_v\) and \(\mathbf{M}_e\) are independent of time velocity and energy mass matrices; and \(\mathbf{F}\) is the generalized corner force matrix depending on \((\mathbf{v}, e, x)\) that needs to be evaluated at every time step.

Reference:

- Contractions can often be implemented as index reordering plus batched GEMM (and hence, be highly efficient)

Reference:
Batched routines released in MAGMA

**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)
Implementation on current hardware is becoming challenging.

Memory hierarchies for different type of architectures.

<table>
<thead>
<tr>
<th></th>
<th>Haswell E5-2650 v3</th>
<th>KNL 7250 DDR5</th>
<th>ARM</th>
<th>K40c</th>
<th>P100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>16/core AVX2</td>
<td>32/core AVX-512</td>
<td>32/core</td>
<td>256 KB/SM</td>
<td>256 KB/SM</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L1 Cache &amp; GPU Shared Memory</td>
<td>32 KB/core</td>
<td>32 KB/core</td>
<td>32 KB/core</td>
<td>64 KB/SM</td>
<td>64 KB/SM</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L2 Cache</td>
<td>256 KB/core</td>
<td>1024 KB/2 cores</td>
<td>2 MB</td>
<td>1.5 MB</td>
<td>4 MB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L3 Cache</td>
<td>25 MB</td>
<td>0...16 GB</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Main Memory</td>
<td>64 GB</td>
<td>384</td>
<td>16 GB</td>
<td>4 GB</td>
<td>12 GB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Main Memory Bandwidth</td>
<td>68 GB/s</td>
<td>115</td>
<td>421 GB/s</td>
<td>26 GB/s</td>
<td>288 GB/s</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interconnect</td>
<td>16 GB/s</td>
<td>16 GB/s</td>
<td>16 GB/s</td>
<td>16 GB/s</td>
<td>16 GB/s</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Interconnect</td>
<td>6 GB/s</td>
<td>6 GB/s</td>
<td>6 GB/s</td>
<td>6 GB/s</td>
<td>6 GB/s</td>
</tr>
</tbody>
</table>

Workshop on Batched, Reproducible, and Reduced Precision BLAS

Georgia Tech
Computational Science and Engineering
Atlanta, GA
February 23—25, 2017


Draft Reports
Batched BLAS Draft Reports:
https://www.dropbox.com/s/olocmipyxfvcaui/batched_api_03_30_2016.pdf?dl=0

Batched BLAS Poster:
https://www.dropbox.com/s/ddkym76fapddf5c/Batched%20BLAS%20Poster%2012.pdf?dl=0

Batched BLAS Slides:
https://www.dropbox.com/s/kz4fhcipz3e56ju/BatchedBLAS-1.pptx?dl=0

Webpage on ReproBLAS:
http://bebop.cs.berkeley.edu/reproblas/

Efficient Reproducible Floating Point Summation and BLAS:
http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-229.pdf
Tensor contractions – performance

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.

Reference:
MAGMA-DNN

Motivation

• Deep learning architectures show promising results in abstract tasks like image classifications

• They inherently consist of same old neural networks as before except
  o The size of the networks has increased drastically, more compute power,
  o Training dataset is huge, but fast

• Any improvement in the core modules of such networks will greatly influences the training and increases performance, also helps in understanding the network well

• Spatial convolution in convnets takes up to 70% of the total execution.

• We optimize spatial convolution module specifically for convnets.
Introduction to Spatial Convolution:

- Input and weights are 3D tensors
- As the filter traverses across the input volume horizontally and vertically it generates a 2D activation map
- Multiple filters generate multiple 2D output frames
- These output frames are stacked to form a 3D output tensor
MAGMA-DNN

Background or existing technique: Unfold and GEMM

VGG-16 D conv modules

<table>
<thead>
<tr>
<th>Conv</th>
<th>Weight Matrix</th>
<th>Data Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>64x27</td>
<td>27x50176</td>
</tr>
<tr>
<td>2</td>
<td>64x576</td>
<td>576x50176</td>
</tr>
<tr>
<td>3</td>
<td>128x576</td>
<td>576x12544</td>
</tr>
<tr>
<td>4</td>
<td>128x1152</td>
<td>1152x12544</td>
</tr>
<tr>
<td>5-7</td>
<td>256x2304</td>
<td>2304x3136</td>
</tr>
<tr>
<td>8</td>
<td>512x2304</td>
<td>2304x784</td>
</tr>
<tr>
<td>9-10</td>
<td>512x4608</td>
<td>4608x784</td>
</tr>
<tr>
<td>10-13</td>
<td>512x4608</td>
<td>4608x196</td>
</tr>
</tbody>
</table>

\[ N = 1 \]
\[ C = 3 \]
\[ H = 3 \]
\[ W = 3 \]
\[ K = 2 \]
\[ R = 2 \]
\[ S = 2 \]
\[ u=v = 1 \]
\[ \text{pad}_h = 0 \]
\[ \text{pad}_w = 0 \]
Background or existing technique: Unfold and GEMM

Advantages:
- Unfold involves streaming memcpy and can be made parallel by having many threads working on many sections of the input
- Many BLAS libraries contain fine tuned GEMM routines that can be used
- The output format is consistent with the actual convolution output

Disadvantages:
- Unfold operation requires extra memory
- Matrix shapes can be greatly skewed
Convolution using transformation techniques

**FFT method:**
- Convolution becomes elementwise product in frequency domain
- Complexity in 2D is $O(RS \log(HW))$, better than $O(RSHW)$ in direct convolution
- Caution !! filter dimension should be similar to image dimension but in convnets that are used widely $RS \ll HW$

**Winograd Minimal filtering:**
- Best suited for small filters, $RS \ll HW$
- Reduces the arithmetic operations by constant factor thus improves the asymptotic timing
Winograd algorithm

Steps:
1. Transform a 4x4 image tile
2. Transform a 3x3 filter
3. Perform element wise product between the transformed tiles
4. Inverse transform on the product tile

1. \[
\begin{bmatrix}
1 & 1 & 1 & 0 \\
0 & 1 & -1 & -1 \\
\end{bmatrix}
\begin{bmatrix}
1 & 1 & 1 & 0 \\
0 & 1 & -1 & -1 \\
\end{bmatrix}
\begin{bmatrix}
1 & 0 & -1 & 0 \\
0 & 1 & 1 & 0 \\
0 & -1 & 1 & 0 \\
0 & 1 & 0 & -1 \\
\end{bmatrix}
\begin{bmatrix}
1 & 0 & -1 & 0 \\
0 & 1 & 1 & 0 \\
0 & -1 & 1 & 0 \\
0 & 1 & 0 & -1 \\
\end{bmatrix}^T
\]

2. \[
\begin{bmatrix}
0 & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0 \\
\end{bmatrix}
\begin{bmatrix}
1 & 0 & -1 & 0 \\
0 & 1 & 1 & 0 \\
0 & -1 & 1 & 0 \\
0 & 1 & 0 & -1 \\
\end{bmatrix}
\begin{bmatrix}
1 & 0 & -1 & 0 \\
0 & 1 & 1 & 0 \\
0 & -1 & 1 & 0 \\
0 & 1 & 0 & -1 \\
\end{bmatrix}^T
\]

3. \[
\begin{bmatrix}
1 & 0 & -1 & 0 \\
0 & 1 & 1 & 0 \\
0 & -1 & 1 & 0 \\
0 & 1 & 0 & -1 \\
\end{bmatrix}
\begin{bmatrix}
1 & 0 & -1 & 0 \\
0 & 1 & 1 & 0 \\
0 & -1 & 1 & 0 \\
0 & 1 & 0 & -1 \\
\end{bmatrix}^T
\]
Winograd algorithm: reduction to GEMM’s

- Each Image tensor has 16 matrices of size $T \times C$
- The K filters are reduced to 16 $C \times K$ matrices
- For a batch size of $N$ there are 16N GEMMs, for example $N=64$ gives 1024 GEMMs.
- 16 filter matrices are common for all the GEMMs, so better last level cache efficiency
- Once GEMM are performed, “Gather” the elements from the 16 output matrices to form a 4x4 output tile
- Apply inverse transform on the output tile to obtain 2x2 convolution output
Winograd algorithm: Advantage of GEMM’s

• Each transformed image tile is reused with K filters. Similarly, each filter tile is reused with all the input tiles across the batch of N.

• If N and K are large enough, the transformation cost is amortized because of max re-usage of transformed tiles.

• Instead of applying inverse across C and then accumulating, the natural form of GEMM accumulates the result across C and then inverse can be applied once to the GEMM output tile.

• This is possible because of the linearity property for Winograd convolution. Fine-tuned GEMM APIs are available.

• Good cache efficiency. Good arithmetic intensity.
**Winograd algorithm**

<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>
Optimizing GEMM’s: Kernel design

\[ C = \beta C + \alpha AB \]
Optimizing GEMM’s: Kernel design

\[ C = \beta C + \alpha AB \]

- Assign every block of \( C_{ii} \) to a TB
- Hold the block \( C_{ii} \) in register/sm
- Slide the green tile of A and B and compute \( C = \beta C + \alpha AB \)
- This design guarantees reproducibility of results
- The kernel is parameterized to allow tuning and optimization
Optimizing GEMM’s: Kernel design

\[ C = \beta C + \alpha AB \]

\[ T = A_1 B_1 \]
\[ T += A_2 B_2 \]
\[ T += A_3 B_3 \]
\[ T += A_4 B_4 \]
\[ C_{22} = \beta C_{22} + \alpha T \]
• 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
Optimizing GEMM’s: Kernel design

- Prefetching
Optimizing GEMM’s: Kernel design

Are we done, we have our best kernel?

- for most of the case LA algorithms, Deep Learning, etc., the matrices A,B,C are not squares which requires autotuning
Optimizing GEMM’s: Kernel tuning

![Graph showing performance optimization for different gemm configurations.](image)

- **Performance bound**
- **gemm 739 (6)**
- **gemm 711 (6)**
- **gemm 742 (5)**
- **gemm 741 (4)**
- **gemm 738 (4)**

**Intel Xeon E5-2650 v3 (Haswell)**
- 2x10 cores @ 2.30 GHz

---

**MAGMA-DNN**

---

**GPU TECHNOLOGY CONFERENCE**

---

**CEED**

---

**ICL INNOVATIVE COMPUTING LABORATORY**

---

**THE UNIVERSITY OF TENNESSEE**

Department of Electrical Engineering and Computer Science
Optimizing GEMM’s: Kernel tuning

Gflops vs. matrix size for different Gemm kernels:
- Gemm 636 (4)
- Gemm 777 (3)
- Gemm 765 (3)
- Gemm 745 (3)
- Gemm 732 (3)

Performance bound and min for a CPU Intel Xeon E5-2650 v3 (Haswell):
- 2x10 cores @ 2.30 GHz
Optimizing GEMM’s: Kernel tuning

sgemm NN square

Gflops

matrix size
Optimizing GEMM's: Kernel tuning

sgemm NN custom1

Gflops vs matrix size

Performance bound:
- gemm 447 (4)
- gemm 557 (3)
- gemm 496 (3)
- gemm 440 (3)
- gemm 837 (2)

Performance min:

Intel Xeon E5-2650 v3 (Haswell)
2x10 cores @ 2.30 GHz
Conclusions and future work

In conclusion:

- Developed a number of NLA in MAGMA targeting applications
  - High-order FEM, DNN, and data analytics;
  - Tensor abstractions and high-performance tensor contractions (for high-order FEM)
- Multidisciplinary effort
- Achieve 90+% of theoretical maximum on GPUs and multicore CPUs
- Use on-the-fly tensor reshaping to cast tensor contractions as small but many GEMMs, executed using batched approaches
- Custom designed GEMM kernels for small matrices and autotuning

Future directions:

- To release a tensor contractions package through the MAGMA library
- To release NLA backend for DLA and data analytics
- Integrate developments in applications
- Complete autotuning and develop all kernels
Collaborators and Support

MAGMA team
http://icl.cs.utk.edu/magma

Collaborating partners
University of Tennessee, Knoxville
Lawrence Livermore National Laboratory, Livermore, CA
LLNL led ECP CEED:
Center for Efficient Exascale Discretizations
University of Manchester, Manchester, UK
University of Paris-Sud, France
INRIA, France