MAGMA: Development of High-Performance Linear Algebra for GPUs

Stan Tomov
University of Tennessee, Knoxville
Outline

• Methodology
• Dense linear system and eigen-problem solvers
• Multi-GPU algorithms
  – Dynamic scheduling
  – Distributed MAGMA
• MAGMA Sparse
• Future Directions
## Next Generation of DLA Software

Software/Algorithms follow hardware evolution in time

<table>
<thead>
<tr>
<th>Time Period</th>
<th>Software/Criteria</th>
<th>Rely on</th>
</tr>
</thead>
<tbody>
<tr>
<td>1970’s</td>
<td>LINPACK (Vector operations)</td>
<td>Level-1 BLAS operations</td>
</tr>
<tr>
<td>1980’s</td>
<td>LAPACK (Blocking, cache friendly)</td>
<td>Level-3 BLAS operations</td>
</tr>
<tr>
<td>1990’s</td>
<td>ScaLAPACK (Distributed Memory)</td>
<td>PBLAS Mess Passing</td>
</tr>
<tr>
<td>2000’s</td>
<td>PLASMA (New Algorithms) (many-core friendly)</td>
<td>a DAG/scheduler, block data layout, some extra kernels</td>
</tr>
</tbody>
</table>

**MAGMA**

Hybrid Algorithms (heterogeneity friendly)

Rely on
- hybrid scheduler (of DAGs)
- hybrid kernels (for nested parallelism)
- existing software infrastructure
MAGMA: LAPACK for GPUs

• MAGMA
  - Matrix algebra for GPU and multicore architecture
  - The LAPACK/ScaLAPACK on hybrid architectures
  - [http://icl.cs.utk.edu/magma/](http://icl.cs.utk.edu/magma/)

• MAGMA 1.4.1
  - For NVIDIA CUDA GPUs on shared memory systems
  - Hybrid dense linear algebra (for CPUs and GPUs)
    • One-sided factorizations and linear system solvers
    • Two-sided factorizations and eigenproblem solvers
    • A subset of BLAS and auxiliary routines in CUDA

• MAGMA developers & collaborators
  - UTK, UC Berkeley, UC Denver, INRIA (France), KAUST (Saudi Arabia)
  - Community effort, similarly to LAPACK/ScaLAPACK
Key Features of MAGMA 1.4.1

- High performance
- Multiple precision support (Z, C, D, S, and MP)
- Hybrid algorithms
- Out-of-GPU memory algorithms
- MultiGPU support
Key Features of MAGMA 1.4.1

HYBRID ALGORITHMS
MAGMA uses a hybridization methodology where algorithms of interest are split into tasks of varying granularity and their execution scheduled over the available 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 GPU.

PERFORMANCE

MAGMA on Kepler K40
LU factorization in double precision arithmetic

<table>
<thead>
<tr>
<th>GPU</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVIDIA K40 (Atlas)</td>
<td>Intel Xeon ES-2670 (Sandy Bridge)</td>
</tr>
<tr>
<td>15 MP x 192 @ 0.88 GHz</td>
<td>2 x 8 cores @ 2.60 GHz</td>
</tr>
</tbody>
</table>

FEATURES AND SUPPORT

- **MAGMA 1.4.1** FOR CUDA
- **clIMAGMA 1.1** FOR OpenCL
- **MAGMA MIC 1.1** FOR Intel Xeon Phi

- Linear system solvers
- Eigenvalue problem solvers
- Auxiliary BLAS
- CPU Interface
- GPU Interface
- Multiple precision support
- Non-GPU-resident factorizations
- Multicore and multi-GPU support
- LAPACK testing
- Linux
- Windows
- Mac OS
Multiple precision support

Performance of the LU factorization in various precisions

- CGETRF_GPU
- SGETRF_GPU
- ZGETRF_GPU
- DGETRF_GPU

Matrix size

Keeneland GPU M2090 (14 MP @1.3 GHz, peak 583 GFlop/s)
CPU Intel Xeon X5660 (2x6 cores @2.8GHz)
Methodology overview

A methodology to use all available resources:

• MAGMA uses **hybridization** methodology based on
  
  – Representing linear algebra algorithms as collections of **tasks** and **data dependencies** among them
  
  – Properly **scheduling** tasks' execution over multicore and GPU hardware components

• Successfully applied to fundamental linear algebra algorithms
  
  – One- and two-sided factorizations and solvers
  
  – Iterative linear and eigensolvers

• Productivity
  
  – 1) High level; 2) Leveraging prior developments; 3) Exceeding in performance homogeneous solutions

Hybrid CPU+GPU algorithms
(small tasks for multicores and large tasks for GPUs)
A Hybrid Algorithm Example

• Left-looking hybrid Cholesky factorization in MAGMA

```c
for (j=0; j<n; j += nb) {
    jb = min(nb, n - j);
    magma_zherk( MagmaUpper, MagmaConjTrans,
                 jb, j, m_one, dA(0, j), ldda, one, dA(j, j), ldda, queue);
    magma_zgetmatrix_async(jb, jb, dA(j,j), ldda, work, 0, jb, queue, &event);
    if (j+jb < n)
        magma_zgemm(MagmaConjTrans, MagmaNoTrans, jb, n-j-jb, j, mz_one,
                     dA(0, j), ldda, dA(0, j+jb), ldda, z_one, dA(j, j+jb), ldda, queue);
    magma_event_sync(event);
    lapackf77_zpotrf(MagmaUpperStr, &jb, work, &jb, info);
    if (*info != 0)
        *info += j;
    magma_zsetmatrix_async(jb, jb, work, 0, jb, dA(j,j), ldda, queue, &event);
    if (j+jb < n) {
        magma_event_sync(event);
        magma_ztrsm(MagmaLeft, MagmaUpper, MagmaConjTrans, MagmaNonUnit,
                    jb, n-j-jb, z_one, dA(j, j), ldda, dA(j, j+jb), ldda, queue);
    }
}
```

• The difference with LAPACK – the 4 additional lines in red
• Line 8 (done on CPU) is overlapped with work on the GPU (from line 6)
Mixed precision iterative refinement

Solving general dense linear systems using mixed precision iterative refinement

- SP Solve
- DP Solve (MP Iter.Ref.)
- DP Solve

Matrix size

GFlp/s

SP Solve
DP Solve (MP Iter.Ref.)
DP Solve

Keeneland
GPU  M2090 (14 MP @1.3 GHz, peak 583 GFlp/s)
CPU  Intel Xeon X5660@2.80GHz (2 x 6 cores)
Out of GPU Memory Algorithms

Solving large problems that do not fit in the GPU memory

Matrices of size that do not fit in a specified GPU memory

Keeneland
GPU  M2090 (14 MP @1.3 GHz, peak 583 GFlop/s)
CPU  Intel Xeon X5660@2.80GHz (2 x 6 cores)
Out of GPU Memory Algorithms

Solving large problems that do not fit in the GPU memory

Keeneland

**GPU**  M2090 (14 MP @1.3 GHz, peak 583 GFlop/s)

**CPU**  Intel Xeon X5660@2.80GHz (2 x 6 cores)

Out-of-GPU-memory Algorithms can now solve large problems
Out of GPU Memory Algorithm

- Perform left-looking factorizations on sub-matrices that fit in the GPU memory (using existing algorithms)
- The rest of the matrix stays on the CPU
- Left-looking versions minimize writing on the CPU

1. Copy $A_2$ to the GPU
2. Update $A_2$ using $A_1$ (a panel of $A_1$ at a time)
3. Factor the updated $A_2$ using existing hybrid code
4. Copy factored $A_2$ to the CPU

Trivially extended to multiGPUs:
$A_2$ is “larger” with 1-D block cyclic distribution, again reusing existing algorithms
MultiGPU Support

• Data distribution
  – 1-D block-cyclic distribution

• Algorithm
  – GPU holding current panel is sending it to CPU
  – All updates are done in parallel on the GPUs
  – Look-ahead is done with GPU holding the next panel
LU on multiGPUs in DP

Matrix size

GFlop/s

Keeneland
GPU M2090 (14 MP @1.3 GHz, peak 583 GFlop/s)
CPU Intel Xeon X5660@2.80GHz (2 x 6 cores)
LU on multiGPUs in DP

- 2 GPUs
- 1 GPU
- CPU (MKL)

**Keeneland**
- **GPU** M2090 (14 MP  @1.3 GHz, peak  583 GFlop/s)
- **CPU** Intel Xeon X5660@2.80GHz (2 x 6 cores)
LU on multiGPUs in DP

- 3 GPUs
- 2 GPUs
- 1 GPU
- CPU (MKL)

Matrix size

GFlop/s

Keeneland
- GPU: M2090 (14 MP @1.3 GHz, peak 583 GFlop/s)
- CPU: Intel Xeon X5660@2.80GHz (2 x 6 cores)
LU on Kepler in DP

Kepler (K20X) 3 GPUs 2 GPUs 1 GPU CPU (MKL)

Keeneland GPU M2090 (14 MP @1.3 GHz, peak 583 GFlop/s)
CPU Intel Xeon X5660@2.80GHz (2 x 6 cores)
Eigenproblem Solvers in MAGMA

\[ A x = \lambda x \]

- Quantum mechanics (Schrödinger equation)
- Quantum chemistry
- Principal component analysis (in data mining)
- Vibration analysis (of mechanical structures)
- Image processing, compression, face recognition
- Eigenvalues of graph, e.g., in Google’s page rank

\[ \ldots \]

• Need to solve it fast

Current MAGMA results:

MAGMA with 1 GPU can be 12x faster vs. vendor libraries on state-of-art multicore systems


Toward fast Eigensolvers

Keeneland system, using one node
3 NVIDIA GPUs (M2090 @ 1.1 GHz, 5.4 GB)
2 x 6 Intel Cores (X5660 @ 2.8 GHz, 23 GB)

flops formula: $n^3/3 \times \text{time}$
Higher is faster

Characteristics
- Too many Blas-2 op,
- Relies on panel factorization,
- Bulk sync phases,
- Memory bound algorithm.

Toward fast Eigensolvers

Keeneland system, using one node
3 NVIDIA GPUs (M2090@ 1.1 GHz, 5.4 GB)
2 x 6 Intel Cores (X5660 @ 2.8 GHz, 23 GB)

flops formula: $n^3/3 \times \text{time}$
Higher is faster

Characteristics
- Blas-2 GEMV moved to the GPU,
- Accelerate the algorithm by doing all BLAS-3 on GPU,
- \text{Bulk sync phases},
- \text{Memory bound algorithm}.

**Toward fast Eigensolvers**

- **Characteristics**
  - **Stage 1:** BLAS-3, increasing computational intensity,
  - **Stage 2:** BLAS-1.5, new cache friendly kernel,
  - **4X/12X faster** than standard approach,
  - **Bottleneck:** if all Eigenvectors are required, it has 1 back transformation extra cost.


**Keeneland system, using one node**
3 NVIDIA GPUs (M2090@ 1.1 GHz, 5.4 GB)
2 x 6 Intel Cores (X5660 @ 2.8 GHz, 23 GB)

**Acceleration w/ 3 GPUs:**
15 X vs. 12 Intel cores

**flops formula: n^3/3*time**
Higher is faster
Toward fast Eigensolvers for Non-symmetric Matrices

- \( A \) is \( n \times n \), nonsymmetric
- \( Ax = \lambda x \)
- Three phases:
  - Hessenberg reduction
    \( H = Q_1^T A Q_1 \)
  - QR iteration to triangular form
    \( T = Q_2^T H Q_2 \)
  - Compute eigenvectors \( Z \) of \( T \) and back-transform to eigenvectors \( X \) of \( A \)

n = 16000, 2x8 core Intel Sandy Bridge, NVIDIA Kepler K40 GPU
Current work

• Schedule task execution using Dynamic Runtime Systems

48 cores
POTRF, TRTRI and LAUUM. The matrix is 4000 x 4000, tile size is 200 x 200
Current work

High-productivity w/ Dynamic Runtime Systems
From Sequential Nested-Loop Code to Parallel Execution

for (k = 0; k < min(MT, NT); k++) {
    zgeqrt(A[k;k], ...);
    for (n = k+1; n < NT; n++)
        zunmqr(A[k;k], A[k;n], ...);
    for (m = k+1; m < MT; m++) {
        ztsqrt(A[k;k], A[m;k], ...);
        for (n = k+1; n < NT; n++)
            ztsmqr(A[m;k], A[k;n], A[m;n], ...);
    }
}
Current work

High-productivity w/ Dynamic Runtime Systems
From Sequential Nested-Loop Code to Parallel Execution

for (k = 0; k < min(MT, NT); k++){
    Insert_Task(&cl_zgeqrt, k , k, ...);
    for (n = k+1; n < NT; n++)
        Insert_Task(&cl_zunmqr, k, n, ...);
    for (m = k+1; m < MT; m++){
        Insert_Task(&cl_ztsqrt, m, k, ...);
        for (n = k+1; n < NT; n++)
            Insert_Task(&cl_ztsmqr, m, n, k, ...);
    }
}

Various runtime systems can be used:

• StarPU
  http://icl.cs.utk.edu/projectsdev/morse
• PaRSEC
  https://icl.cs.utk.edu/parsec/
• QUARK
  http://icl.cs.utk.edu/quark/
Scalability and efficiency:

- Snapshot of the execution trace of the Cholesky factorization on System A for a matrix of size 40K using six GPUs K20c.

- As expected the pattern of the trace looks compressed which means that our implementation is able to schedule and balance the tasks on the GPU devices (six).
Dynamic MAGMA with QUARK

![Graph showing performance of DPOTRF 1 K20c with MAGMA and QUARK](image)
Dynamic MAGMA with QUARK

![Graph showing Gflop/s vs Matrix size for DPOTRF 2 K20c and DPOTRF 1 K20c]
Dynamic MAGMA with QUARK
Dynamic MAGMA with QUARK
Dynamic MAGMA with QUARK

- DPOTRF 6 K20c
- DPOTRF 4 K20c
- DPOTRF 3 K20c
- DPOTRF 2 K20c
- DPOTRF 1 K20c
Distributed MAGMA

- Preliminary work on distributed memory systems
- Extensions of the Dynamic MAGMA
  - ScalAPACK 2D block-cyclic data distribution
  - Lightweight “local” (node) scheduling with QUARK + MPI communications
  - Match in performance previous results using “tile” algorithms


Single node performance (N = 34 K)

- Kernel peakx(#cores+#GPUs)
- Cholesky factorization

Weak scaling (N = 340 K)

- DGEMM UB
- Distri. GPUs

75%
Sparse Solvers for GPUs
Sparse HPC on modern architectures

• Important scientific applications rely on sparse linear algebra

• HPCG – a new benchmark proposal to complement Top500 (HPL)
  – To solve $A x = b$, where $A$ is large and sparse
  – To show essential communication & computation patterns in solving PDEs
  – To encourage the focus on architecture features and application needs
  – In collaboration with Sandia National Laboratory

• MAGMA Sparse
  – Develop GPU-aware Sparse Solvers
  – Support from DOE, DOD, and Nvidia
MAGMA Sparse

• Recently added MAGMA component for sparse linear algebra
• Under evaluation for release (friendly users and collaborators)
• Current MAGMA Sparse functionality:
  – Krylov subspace iterative linear system and eigen-problem solvers
  – Support for various matrix formats
  – Sparse BLAS GPU kernels
  – Dense LA building blocks for preconditioners
Challenges on modern architectures

- The explosion of parallelism
e.g., single K40 has 2,880 CUDA cores; algorithms must account for these levels of parallelism

- The growing gap of compute vs. data-movement capabilities

K40 GPU computing efficiency on
Compute intensive (dense LU) vs. Memory-bound computation (SpMV)

Focus is on architecture-aware algorithms of high-parallelism and improved data access patterns
GPU-aware Sparse Solvers
[ braking the memory bound performance limitation ! ]

- Reference as well as optimized (kernels & reduced communication) implementations

[included are SpMV / SpMM in various formats, e.g., DENSE, CSR, Block-CSR, ELLPACK, ELLPACKT, ELLPACKRT, HYB, COO, CSC, SELLC/SELLC-σ; and other kernels/building blocks ...]

Performance of SpMM for various matrices and a block of 32 vectors on a K40 GPU
(GPU MAGMA & CUBLAS)

# GPU-aware Sparse Solvers

- Reference as well as optimized (kernels & **reduced communication**) implementations

  [included are CG, BiCGSTAB, GMRES, preconditioned versions, CA-GMRES, and LOBPCG]

<table>
<thead>
<tr>
<th>BiCGSTAB Method implementation</th>
<th>Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>while ( (k &lt; \text{maxiter}) ) &amp;&amp; (res &gt; \tau) )</td>
<td></td>
</tr>
<tr>
<td>( k := k + 1 )</td>
<td></td>
</tr>
<tr>
<td>( \rho_k := \rho_{k-1} )</td>
<td></td>
</tr>
<tr>
<td>( \beta_{k+1} := \frac{\rho_k}{\rho_k - \alpha_k} )</td>
<td></td>
</tr>
<tr>
<td>( p_k := r_{k-1} + \beta (p_{k-1} - \omega_{k-1} v_{k-1}) )</td>
<td></td>
</tr>
<tr>
<td>( v_k := Ap_k )</td>
<td></td>
</tr>
<tr>
<td>( \alpha_k := \frac{p_k^T v_k}{v_k} )</td>
<td></td>
</tr>
<tr>
<td>( s_k := r_{k-1} - \alpha v_k )</td>
<td></td>
</tr>
<tr>
<td>( t_k := As_k )</td>
<td></td>
</tr>
<tr>
<td>( \omega_k := \frac{1}{s_k^T r_k} )</td>
<td></td>
</tr>
<tr>
<td>( x_{k+1} := x_k + \alpha_k p_k + \omega_k s_k )</td>
<td></td>
</tr>
<tr>
<td>( r_k := s_k - \omega t_k )</td>
<td></td>
</tr>
<tr>
<td>( \text{res} = r_k^T r_k )</td>
<td></td>
</tr>
</tbody>
</table>

| while( \( (k < \text{maxiter}) \) && (\text{res}_\text{host} > \epsilon) \) ) \{ |
| magma_dbicgmerge_p_update\(<<<\text{Gs}, \text{Bs}, 0>>>\) |
| (n, skp, v, r, p); |
| magma_dbicgmerge_spmv1\(<<<\text{Gs}, \text{Bs}, \text{Ms1}>>>\) |
| (n, valA, rowA, colA, p, r, v, d1); |
| magma_dbicgmerge_reduce1( n, Gs, Bs, d1, d2, skp ); |
| magma_dbicgmerge_s_update\(<<<\text{Gs}, \text{Bs}, 0>>>\) |
| (n, skp, r, v, s ); |
| magma_dbicgmerge_spmv2\(<<<\text{Gs}, \text{Bs}, \text{Ms2}>>>\) |
| (n, valA, rowA, colA, s, t, d1); |
| magma_dbicgmerge_reduce2( n, Gs, Bs, d1, d2, skp ); |
| magma_dbicgmerge_xr_update\(<<<\text{Gs}, \text{Bs}, 0>>>\) |
| (n, skp, r\_hat, r, p, s, t, x, d1); |
| magma_dbicgmerge_reduce3( n, Gs, Bs, d1, d2, skp ); |
| magma_memcpy( 1, skp+5, res\_host ); |
| k++; |

---

GPU-aware Sparse Solvers

- Communication avoiding GMRES (CA-GMRES)
  - Replacing GMRES’ SpMV $\rightarrow$ Matrix Powers Kernel (MPK):
    \[ v_{k+1} = A v_k \quad \text{for } k = j, \ldots, j+s \]

  BLAS-2 $\rightarrow$ BLAS-3 based orthogonalization (next ...)

MPK to generate 100 vectors for various $s$  Overall performance improvement on up to 3 GPUs

Orthogonalization procedures

- Mixed-precision Cholesky QR
  - CholQR obtains BLAS-3 performance, but error is bounded by $\varepsilon \kappa(V)^2$
  - Remove the “square” by selectively using double-double (doubled) precision

GPU-aware Sparse Eigen-Solvers

- Locally Optimal Block PCG (LOBPCG)
  - Find a set of smallest eigen-states of a sparse SPD matrix (\( A x = \lambda x \))
  - Replace finding the states one-by-one by a block algorithm
    - finds them simultaneously; needs fast SpMM, re-orthogonalization, and GEMM of particular sizes

Performance of SpMM with various matrices (x 32 vec.)

Overall speedup vs. LOBPCG from BLOPEX on CPUs
Batched DLA & Other Building Blocks

- Many small DLA problems solved in parallel
  - Needed in preconditioners [5,6], orthogonalization routines [4,5], some sparse direct solvers, high-order FEMs [8], and batched higher-level DLA [7]


Future Directions

• Distributed multi-GPU solvers
  – Scheduling
  – Reproducibility

• Further tuning
  – partitioning, comm/comp overlaps
  – autotuning

• Extended functionality
  – Other mixed precision techniques
  – Other communication-avoiding techniques
  – Sparse direct multifrontal solvers & preconditioners
  – Other techniques of bringing BLAS-3 performance to sparse solvers
Collaborators / Support

• MAGMA [Matrix Algebra on GPU and Multicore Architectures] team http://icl.cs.utk.edu/magma/

• PLASMA [Parallel Linear Algebra for Scalable Multicore Architectures] team http://icl.cs.utk.edu/plasma

• Collaborating partners
  – University of Tennessee, Knoxville
  – University of California, Berkeley
  – University of Colorado, Denver
  – INRIA, France
  – KAUST, Saudi Arabia