cIMAGMA: High Performance Dense Linear Algebra with OpenCL

Stan Tomov
C. Cao, J. Dongarra, P. Du, M. Gates, and P. Luszczek

Innovative Computing Laboratory
University of Tennessee, Knoxville

1st International Workshop on OpenCL (IWOCL), May 13—14, 2013
Georgia Institute of Technology, Atlanta, GA
- **Methodology overview**
  - *Hybridization* of Linear Algebra Algorithms
  - Use both GPUs and multicore CPUs

- **clMAGMA**
  - OpenCL port of MAGMA
  - Performance results
  - Challenges and future directions

- **Conclusions**
cIMAGMA Software Stack

**CPU**
- Hybrid LAPACK / ScALAPACK & Tile Algorithms / StarPU / DAgUE

**HYBRID**
- MAGMA 1.3
  - Hybrid Tile (PLASMA) Algorithms
- cIMAGMA 1.0
  - MAGMA SPARSE
  - MAGMA BLAS

**GPU**
- StarPU run-time system
- AMD BLAS (APPML)
- OpenCL

---

*Linux, Windows, Mac OS X | C++/Fortran | Matlab, Python*

[AMD APPML -- Accelerated Parallel Processing Math Libraries
http://developer.amd.com/libraries/appmathlibs/]
cIMAGMA 1.0 Released

2012-10-24

cIMAGMA 1.0 is now available. cIMAGMA is an OpenCL port of the MAGMA library. This release adds the following new functionalities:

- Eigen and singular value problem solvers in both real and complex arithmetic, single and double (routines magma_zlcheevd, magma_zlseqvd, magma_zlclldsgeev, and magma_zlclldsgevd);
- Matrix inversion routines (routines magma_zlclldistri, magma_zlclldisotri, magma_zlclldisotri_gpu, magma_zlclldisotri_gpu);
- Orthogonal transformations routines (routines magma_zlclldunmqr_gpu, (dis)ormqr_gpu, (zlc)unmqr, (dis)ormqr, (zlc)unmql, (dis)ormql, (zlc)unmqr, and (dis)ormqr).

See the MAGMA software homepage for a download link.
DGEMM in OpenCL


**GPU:** Tahiti (AMD Radeon HD 7900)
- 264 GB/s memory bandwidth
- 3.79 Tflop/s SP, 947 Gflop/s DP
- $32 \times 64$ (2048 stream proc.)

![Graph showing performance vs. matrix size for DGEMM operations with different kernels and data formats.](image_url)
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 eigen-solvers

- Productivity
  - 1) High-level; 2) Leveraging prior developments; 3) Exceeding in performance homogeneous solutions
Hybrid Algorithms

One-sided factorizations (LU, QR, Cholesky)

- Hybridization
  - Panels (Level 2 BLAS) are factored on CPU using LAPACK
  - Trailing matrix updates (Level 3 BLAS) are done on the GPU using “look-ahead”
A Hybrid Algorithm Example

- Left-looking hybrid Cholesky factorization in clMAGMA

```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 9 (done on CPU) is overlapped with work on the GPU (from line 6)
Programming model

Host program

```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, j, dA(j, j), ldda, work, 0, jb, queue, &event);
        if (j+jb < n)
            magma_zgemm( MagmaConjTrans, MagmaNoTrans, jb, n-jjb, j, m_z_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, j, 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-jjb, z_one, da(j, j), ldda, da(j, j+jb), ldda, queue);
            }
    }
```

OpenCL interface – communications

```c
magma_err_t
magma_zgetmatrix_async(
    magma_int_t m, magma_int_t n,
    magmaDoubleComplex_const_ptr dA_src, size_t da_offset, magma_int_t lda,
    magmaDoubleComplex* dA_dst, size_t ha_offset, magma_int_t lda,
    magma_queue_t queue, magma_event_t* event
) {
    size_t buffer_origin[3] = { dA_offset*sizeof(magmaDoubleComplex), 0, 0 };
    size_t host_orig[3] = { 0, 0, 0 };
    size_t region[3] = { n*sizeof(magmaDoubleComplex), n, 1 };
    cl_int err = clEnqueueReadBufferRect(
        queue, dA_src, CL_FALSE, // non-blocking
        buffer_origin, host_orig, region,
        lda*sizeof(magmaDoubleComplex), 0,
        lda*sizeof(magmaDoubleComplex), 0,
        ha_dst, 0, NULL, event);
}
```

OpenCL interface – AMD APPML BLAS

```c
magma_zherk(
    magma_uplo_t uplo, magma_trans_t trans,
    magma_int_t n, magma_int_t k,
    double alpha, magmaDoubleComplex_const_ptr dA, size_t da_offset,
    double beta, magmaDoubleComplex_ptr dC, size_t dc_offset,
    magma_queue_t queue )
{
    cl_int err = clMshadowZherk(
        clMshadowColumnMajor,
        amdblas_uplo const( uplo ),
        amdblas_trans const( trans ),
        n, k,
        alpha, dA, da_offset, lda,
        beta, dC, dc_offset, ldc,
        1, &queue, 0, NULL, NULL );
    return err;
}
```
Performance of cIMAGMA
Cholesky Factorization in double precision

GPU:
Tahiti (AMD Radeon 7970)
947 GFlop/s DP

CPU:
One socket six-core AMD Phenom II X6 1100T
@3.71GHz
Performance of cIMAGMA
LU Factorization in double precision

- **GPU:**
  - Tahiti (AMD Radeon 7970)
  - 947 GFlop/s DP

- **CPU:**
  - One socket six-core AMD Phenom II X6 1100T
  - @3.71GHz
Performance of cIMAGMA
QR Factorization in double precision

GPU:
Tahiti (AMD Radeon 7970)
947 GFlop/s DP

CPU:
One socket six-core AMD Phenom II X6 1100T
@3.71GHz
Hybrid Algorithms

Two-sided factorizations (Hessenberg, bi-, and tridiagonalization)

- Hybridization
  - Panels (Level 2 BLAS) are also hybrid, using both CPU & GPU (vs. just CPU as in the one-sided factorizations)
  - Trailing matrix updates (Level 3 BLAS) are done on the GPU using “look-ahead”
Performance of clMAGMA
Hessenberg Factorization in double precision

Graph showing the performance of clMAGMA and MKL11.1 for different matrix sizes.

- **GPU:** Tahiti (AMD Radeon 7970)
  947 GFlop/s DP

- **CPU:** One socket six-core AMD Phenom II X6 1100T
  @3.71GHz
Current work
Dynamic Scheduling

- Conceptually similar to out-of-order processor scheduling because it has:
  - Dynamic runtime DAG scheduler
  - Out-of-order execution flow of fine-grained tasks
  - Task scheduling as soon as dependencies are satisfied
  - Producer-Consumer

- Data Flow Programming Model
  - The DAG approach
  - Scheduling is data driven
  - Inherently parallel
Current Work
High Level of Productivity

From Sequential Nested-Loop Code to Parallel Execution:

```c
for (k = 0; k < \text{min}(\text{MT}, \text{NT}); \ k++){
    \text{zgeqrt}(A[k;k], \ldots);
    \text{for} (n = k+1; n < \text{NT}; n++)
        \text{zunmqr}(A[k;k], A[k;n], \ldots);
    \text{for} (m = k+1; m < \text{MT}; m++){
        \text{ztsqrt}(A[k;k], A[m;k], \ldots);
        \text{for} (n = k+1; n < \text{NT}; n++)
            \text{ztsmqr}(A[m;k], A[k;n], A[m;n], \ldots);
    }
}
```
Current Work
High Level of Productivity

From Sequential Nested-Loop Code to Parallel Execution:

for (k = 0; k < min(MT, NT); k++){
    Insert_Task(&zgeqrt, k, k, ...);
    for (n = k+1; n < NT; n++)
        Insert_Task(&zunmqr, k, n, ...);
    for (m = k+1; m < MT; m++){
        Insert_Task(&ztsqrt, m, k, ...);
        for (n = k+1; n < NT; n++)
            Insert_Task(&ztzmsqr, m, n, k, ...);
    }
}

Current work
Performance optimizations

- Overlap CPU work, GPU work, and CPU-GPU communications

A dgetrf trace example
Performance optimizations in LU

- **dgetrf**
- **dgetrf(flush)**
- **dgetrf(flush+2q)**
- **dgetrf(flush+2q+pinned mem)**

**GPU:**
Tahiti (AMD Radeon 7970)
947 GFlop/s DP

**CPU:**
One socket six-core AMD Phenom II X6 1100T
@3.71GHz
OpenCL-specific optimizations

- Benchmarks to discover OpenCL specifics

Latencies to launch a kernel

<table>
<thead>
<tr>
<th></th>
<th>cIAMDBlas async</th>
<th>CUBLAS async</th>
<th>CUBLAS sync</th>
</tr>
</thead>
<tbody>
<tr>
<td>DGEMM</td>
<td>5</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>DTRMM</td>
<td>5</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>DSYRK</td>
<td>5</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>DTRSM</td>
<td>5</td>
<td>5</td>
<td>10</td>
</tr>
</tbody>
</table>
Panels entirely on GPU?

• Important to have for both dense and certain sparse linear system and eigen-problem solvers

• Can we factor panels faster on GPU as panels are memory bound?

• Latencies may be a bottleneck
  ▪ e.g., 64 columns panel would require the invocation of ~400 kernels

<table>
<thead>
<tr>
<th>M</th>
<th>N</th>
<th>CUDA Time (ms)</th>
<th>OpenCL Time (ms)</th>
<th>16 Sandy Bridge (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>64</td>
<td>5</td>
<td>94</td>
<td>9</td>
</tr>
<tr>
<td>10,000</td>
<td>64</td>
<td>7</td>
<td>104</td>
<td>17</td>
</tr>
<tr>
<td>100,000</td>
<td>64</td>
<td>36</td>
<td>131</td>
<td>89</td>
</tr>
<tr>
<td>1,000,000</td>
<td>64</td>
<td>365</td>
<td>528</td>
<td>1,431</td>
</tr>
</tbody>
</table>

Difference is due to latencies (in our software/hardware configuration) as shown by increasing the problem size.
Summary and Future Directions

- A hybrid methodology and its application to DLA using OpenCL

- **clMAGMA: LAPACK for heterogeneous computing**
  - Achieving high-performance linear algebra using OpenCL
  - clMAGMA 1.0 includes the main
    - one- and two-sided factorizations
    - orthogonal transformation routines
    - linear and eigen-problem solvers

- **What is next?**
  - Further performance/efficiency improvements
  - MultiGPU and distributed environments
Collaborators / Support

- **MAGMA** [Matrix Algebra on GPU and Multicore Architectures] team

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