Accelerating Linear Algebra with MAGMA

Stan Tomov
Mark Gates
Azzam Haidar
{tomov, mgates3, haidar}@icl.utk.edu

Innovative Computing Laboratory
University of Tennessee, Knoxville

Exascale Computing Project
2nd Annual Meeting
Knoxville, TN
February 9, 2018
Outline

• Part I
  Overview of dense linear algebra libraries
  Design principles and fundamentals

• Part II
  MAGMA Overview
  Availability, routines, code, testers, methodology

• Part III
  MAGMA Batched
  MAGMA Sparse
Dense Linear Algebra (DLA) is needed in a wide variety of science and engineering applications:

- **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.

- **DLA is crucial to the development of 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><strong>BLAS</strong></td>
<td>Kernels for dense linear algebra</td>
</tr>
<tr>
<td><strong>LAPACK</strong></td>
<td>Sequential dense linear algebra</td>
</tr>
<tr>
<td><strong>ScaLAPACK</strong></td>
<td>Parallel distributed dense linear algebra</td>
</tr>
<tr>
<td><strong>PLASMA</strong></td>
<td>Dense linear algebra (multicore)</td>
</tr>
<tr>
<td><strong>MAGMA</strong></td>
<td>Dense/batched/sparse linear algebra (accelerators)</td>
</tr>
<tr>
<td><strong>SLATE</strong></td>
<td>Dense linear algebra (distributed memory / multicore / accelerators)</td>
</tr>
</tbody>
</table>

**new software for multicore and accelerators**

Support from **ECP SLATE, CEED, PEEKS, xSDK**
Why use GPUs in HPC?

**PERFORMANCE & ENERGY EFFICIENCY**

MAGMA 2.3 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

**Matrix size N x N**

- Performance GFLOP/s
- Energy efficiency (under ~ the same power draw)

---

**Graph Details**

- Performance GFLOP/s vs Matrix size N x N
- Energy efficiency comparison between CPU, K40, P100, and V100 GPUs.
- 10x improvement in performance and energy efficiency.
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

$y = \alpha x + \beta y$
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
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+\alpha x )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Level 2</td>
<td>( n^2 )</td>
<td>2( n^2 )</td>
<td>2</td>
</tr>
<tr>
<td>( y=y+Ax )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Level 3</td>
<td>4( n^2 )</td>
<td>2( n^3 )</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

Matrix size (N), vector size (NxN)

C = C + A*B

4503 Gflop/s

31x

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
LAPACK – “Linear Algebra PACKage” - uses BLAS-3 (1989 – now)

Ex: Obvious way to express Gaussian Elimination (GE) is adding multiples of one row to other rows – BLAS-1

How do we reorganize GE to use BLAS-3?

Contents of LAPACK (summary)

Algorithms we can turn into (nearly) 100% BLAS 3

- Linear Systems: solve $Ax=b$ for $x$
- Least Squares: choose $x$ to minimize $||Ax - b||_2$

Algorithms that are only 50% BLAS 3 (so far)

- “Eigenproblems”: Find $\lambda$ and $x$ where $Ax = \lambda x$
- Singular Value Decomposition (SVD): $(A^TA)x=\sigma^2x$

Generalized problems (e.g., $Ax = \lambda Bx$)

Error bounds for everything

Lots of variants depending on $A$’s structure (banded, $A=A^T$, etc)

How much code? (Release 3.8, Nov 2017) (www.netlib.org/lapack)

Source: 1674 routines, 490K LOC, Testing: 448K LOC
A brief history of (Dense) Linear Algebra software

• **Is LAPACK parallel?**
  • Only if the BLAS are parallel (possible in shared memory)

• **ScaLAPACK – “Scalable LAPACK” (1995 – now)**
  • For distributed memory – uses MPI
  • More complex data structures, algorithms than LAPACK
    • Only (small) subset of LAPACK’s functionality available
  • All at www.netlib.org/scalapack
LAPACK

- http://www.netlib.org/lapack/

- LAPACK (Linear Algebra Package) provides routines for
  - solving systems of simultaneous linear equations,
  - least-squares solutions of linear systems of equations,
  - eigenvalue problems,
  - and singular value problems.

- LAPACK relies on BLAS

- The associated matrix factorizations (LU, Cholesky, QR, SVD, Schur, generalized Schur) are also provided, as are related computations such as reordering of the Schur factorizations and estimating condition numbers.

- Dense and banded matrices are handled, but not general sparse matrices. In all areas, similar functionality is provided for real and complex matrices, in both single and double precision.

LAPACK is in FORTRAN
Column Major

LAPACK is SEQUENTIAL

LAPACK is a REFERENCE implementation
Parallelism in LAPACK

- Most flops in gemm update
  - $2/3 n^3$ term
  - Easily parallelized using multi-threaded BLAS
  - Done in any reasonable software
- Other operations lower order
  - Potentially expensive if not parallelized

\[ \text{gemm multiply} \]
\[ \text{getrf panel} \]
\[ \text{luswp swap rows} \]
\[ \text{trsm solve} \]
Overview of Dense Numerical Linear Algebra Libraries

- **BLAS**: kernel for dense linear algebra
- **LAPACK**: sequential dense linear algebra
- **ScaLAPACK**: parallel distributed dense linear algebra

Scalable Linear Algebra PACKage
PBLAS

- Similar to BLAS in functionality and naming
- Built on BLAS and BLACS
- Provide global view of matrix

- LAPACK: \texttt{dge}___( m, n, A(ia, ja), lda, ... )
  - Submatrix offsets implicit in pointer

- ScaLAPACK: \texttt{pdge}___( m, n, A, ia, ja, descA, ... )
  - Pass submatrix offsets and matrix descriptor
ScaLAPACK structure

- ScaLAPACK
- PBLAS
- LAPACK
- BLAS
- BLACS
- MPI

- Global addressing
- Local addressing
- Platform independent
- Platform specific
ScaLAPACK routine, solve $AX = B$

- LAPACK:  
  $$ \text{dgesv}(n, \text{nrhs}, A, \text{lda}, \text{ipiv}, B, \text{ldb}, \text{info}) $$

- ScaLAPACK:  
  $$ \text{pdgesv}(n, \text{nrhs}, A, \text{ia}, \text{ja}, \text{descA}, \text{ipiv}, B, \text{ib}, \text{jb}, \text{descB}, \text{info}) $$

**Input:**

- $A_{11}$, $A_{12}$, $A_{13}$  
- $A_{21}$, $A_{22}$, $A_{23}$  
- $A_{31}$, $A_{32}$, $A_{33}$  
- $B_{11}$, $B_{12}$  
- $B_{21}$, $B_{22}$  
- $B_{31}$, $B_{32}$

**Output:**

- $L_{11}$, $U_{12}$, $U_{13}$  
- $L_{21}$, $L_{22}$, $U_{23}$  
- $L_{31}$, $L_{32}$, $L_{33}$, $U_{33}$  
- $B_{11}$, $B_{12}$  
- $B_{21}$, $B_{22}$  
- $B_{31}$, $B_{32}$

- L, U overwrite $A$
- $X$ overwrites $B$

- info (error code)
  - = 0: no error
  - < 0: invalid argument
  - > 0: numerical error (e.g., singular)

Global matrix point of view

Implicit unit diagonal
### 2D block-cyclic layout

#### Global matrix view

| m | 21 | 22 | 23 | 31 | 32 | 33 | 44 | 42 | 43 | 51 | 52 | 53 | 61 | 62 | 63 | 71 | 72 | 73 | 81 | 82 | 83 | 91 | 92 | 93 |
|   | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 24 | 25 | 26 | 27 | 28 | 34 | 35 | 36 | 37 | 38 | 44 | 45 | 46 | 47 | 48 |

#### Local process point of view

<table>
<thead>
<tr>
<th>q processes</th>
<th>Process 1, 1</th>
<th>Process 1, 2</th>
<th>Process 1, 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>31</td>
<td>32</td>
<td>33</td>
</tr>
<tr>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
</tr>
<tr>
<td>38</td>
<td>44</td>
<td>45</td>
<td>46</td>
</tr>
<tr>
<td>47</td>
<td>48</td>
<td>51</td>
<td>52</td>
</tr>
<tr>
<td>54</td>
<td>55</td>
<td>56</td>
<td>57</td>
</tr>
<tr>
<td>58</td>
<td>61</td>
<td>62</td>
<td>63</td>
</tr>
<tr>
<td>64</td>
<td>65</td>
<td>66</td>
<td>67</td>
</tr>
<tr>
<td>68</td>
<td>71</td>
<td>72</td>
<td>73</td>
</tr>
<tr>
<td>74</td>
<td>75</td>
<td>76</td>
<td>77</td>
</tr>
<tr>
<td>78</td>
<td>81</td>
<td>82</td>
<td>83</td>
</tr>
<tr>
<td>84</td>
<td>85</td>
<td>86</td>
<td>87</td>
</tr>
<tr>
<td>88</td>
<td>91</td>
<td>92</td>
<td>93</td>
</tr>
<tr>
<td>94</td>
<td>95</td>
<td>96</td>
<td>97</td>
</tr>
<tr>
<td>98</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>p processes</th>
<th>Process 2, 1</th>
<th>Process 2, 2</th>
<th>Process 2, 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>25</td>
<td>26</td>
<td>27</td>
<td>41</td>
</tr>
<tr>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
</tr>
<tr>
<td>46</td>
<td>61</td>
<td>62</td>
<td>63</td>
</tr>
<tr>
<td>64</td>
<td>65</td>
<td>66</td>
<td>67</td>
</tr>
<tr>
<td>68</td>
<td>81</td>
<td>82</td>
<td>83</td>
</tr>
<tr>
<td>84</td>
<td>85</td>
<td>86</td>
<td>87</td>
</tr>
<tr>
<td></td>
<td>88</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2D block-cyclic layout

Global matrix view

Local process point of view

$m \times n$ matrix

$p \times q$ process grid
### 2D block-cyclic layout

#### Global matrix view

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
</tr>
<tr>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
</tr>
<tr>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
</tr>
<tr>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td>48</td>
</tr>
</tbody>
</table>

#### Local process point of view

- **Process 1, 1**
  - 11, 14, 17
  - 31, 34, 37
  - 51, 54, 57
  - 71, 74, 77

- **Process 1, 2**
  - 12, 15, 18
  - 32, 35, 38
  - 52, 55, 58
  - 72, 75, 78

- **Process 1, 3**
  - 13, 16
  - 33, 36
  - 53, 56
  - 73, 76

- **Process 2, 1**
  - 21, 24, 27
  - 41, 44, 47
  - 61, 64, 67
  - 81, 84, 87

- **Process 2, 2**
  - 22, 25, 28
  - 42, 45, 48
  - 62, 65, 68
  - 82, 85, 88

- **Process 2, 3**
  - 23, 26
  - 43, 46
  - 63, 66
  - 83, 86
2D block-cyclic layout

Global matrix view

Local process point of view

\( m \times n \) matrix

\( p \times q \) process grid
2D block-cyclic layout

Global matrix view

Local process point of view

$m \times n$ matrix
$p \times q$ process grid
2D block-cyclic layout

Global matrix view

Local process point of view

\( m \times n \) matrix

\( p \times q \) process grid
2D block-cyclic layout

Global matrix view

Local process point of view

$m \times n$ matrix

$p \times q$ process grid
2D block-cyclic layout

Global matrix view

Local process point of view

$m \times n$ matrix
$p \times q$ process grid
2D block-cyclic layout

Global matrix view

Local process point of view

$m \times n$ matrix
$p \times q$ process grid
Parallelism in ScaLAPACK

- Similar to LAPACK
- Bulk-synchronous
- Most flops in gemm update
  - $2/3 n^3$ term
  - Can use **sequential BLAS**, 
    - $p \times q = \#\text{ cores}$
    - $= \#\text{ MPI processes}$, 
    - $\text{num\_threads} = 1$
  - Or **multi-threaded BLAS**, 
    - $p \times q = \#\text{ nodes}$
    - $= \#\text{ MPI processes}$, 
    - $\text{num\_threads} = \#\text{ cores/node}$
Major Changes to Software

• **Must rethink the design of our software**
  • Another disruptive technology
  • Similar to what happened with cluster computing and message passing
  • Rethink and rewrite the applications, algorithms, and software

• **Numerical libraries for example are changing**
  • For example, both LAPACK and ScaLAPACK undergo major changes to accommodate this
## Software Projects

<table>
<thead>
<tr>
<th>netlib.org</th>
<th>icl.utk.edu/research</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAPACK</td>
<td>PLASMA</td>
</tr>
<tr>
<td>ScaLAPACK</td>
<td>MAGMA</td>
</tr>
<tr>
<td>BLAS</td>
<td>SLATE</td>
</tr>
<tr>
<td>CBLAS</td>
<td></td>
</tr>
<tr>
<td>LAPACKE</td>
<td></td>
</tr>
</tbody>
</table>

- **LAPACK**
- **ScaLAPACK**
- **BLAS**
- **CBLAS**
- **LAPACKE**

### dense linear algebra
- (multicore)
- (accelerators)
- (distributed memory / multicore / accelerators)

- new software for multicore and accelerators
Software Projects

- LAPACK
- ScaLAPACK
- BLAS
- CBLAS
- LAPACKE
- PLASMA
- MAGMA
- SLATE
- QUARK
- PaRSEC

netlib.org
icl.utk.edu/research

scheduling (multicore)
scheduling (distributed memory)
dynamic runtime schedulers
Software Projects

- **netlib.org**
  - LAPACK
  - ScaLAPACK
  - BLAS
  - CBLAS
  - LAPACKE

- **icl.utk.edu/research**
  - PLASMA
  - MAGMA
  - SLATE
  - OpenMP
  - PaRSEC

  - Scheduling (multicore)
  - Scheduling (distributed memory)
  - Dynamic runtime schedulers
dense linear algebra for multicore
- dataflow scheduling
- tile matrix layout
- tile algorithms

PLASMA

LAPACK Layout

Tile Layout
Programming with Quark tasking

#include <quark.h>

int main(int argc, char** argv) {
    Quark* quark = QUARK_New( nthreads);
    ...
    for (int m = 1; m <= 8; m++) {
        for (int n = 1; n <= 7; n++) {
            dgemm_tile_quark( quark, NULL,
                CblasColMajor, CblasNoTrans, CblasNoTrans, nb, nb, nb, -1.0,
                A(m, 0), nb, A(0, n), nb, 1.0, A(m, n), nb);
        }
    }
    ...
    QUARK_Delete( quark);
}

void dgemm_tile_task(Quark* quark ) {
    enum CBLAS_ORDER order; enum CBLAS_TRANSPOSE transa,
    enum CBLAS_TRANSPOSE transb, enum CBLAS_TRANSPOSE transc,
    int m, n, k; double alpha, beta, *A, *B, *C, int ldc )
    QUARK_Insert_Task( quark, dgemm_tile_task, task_flags,
        sizeof(enum CBLAS_ORDER), &order, VALUE,
        sizeof(enum CBLAS_TRANSPOSE), &transa, VALUE,
        sizeof(enum CBLAS_TRANSPOSE), &transb, VALUE,
        sizeof(enum CBLAS_TRANSPOSE), &transc, VALUE,
        sizeof(int), &m, VALUE,
        sizeof(int), &n, VALUE,
        sizeof(int), &k, VALUE,
        sizeof(double), &alpha, VALUE,
        sizeof(double *), &A, INPUT, 
        sizeof(int), &lda, VALUE,
        sizeof(double *), &B, INPUT,
        sizeof(int), &ldb, VALUE,
        sizeof(double), &beta, VALUE,
        sizeof(double *), &C, INOUT,
        sizeof(int), &ldc, VALUE, 0);
    }

    quark_unpack_args_15(quark, order, transa, transb, transc,
        m, n, k,
        alpha, A, lda,
        B, ldb,
        beta, C, ldc );

    cblas_dgemm(order, transa, transb, transc,
        m, n, k,
        alpha, A, lda,
        B, ldb,
        beta, C, ldc );
Programming with OpenMP4 tasking

```c
#include <quark.h>

int main(int argc, char** argv) {
    Quark * quark = QUARK_New( nthreads );
    ...
    for (int m = 1; m <= 8; m++) {
        for (int n = 1; n <= 7; n++) {
            dgemm_tile_quark( quark, NULL,
                CblasColMajor, CblasNoTrans, CblasNoTrans,
                nb, nb, nb, -1.0,
                A(m, 0), nb, A(0, n), nb, 1.0, A(m, n), nb);
        }
    }
    ...
    QUARK_Delete( quark );
}

void dgemm_tile_quark(Quark* quark, Quark_Task_Flags * task_flags,
    enum CBLAS_ORDER order, enum CBLAS_TRANSPOSE transa,
    enum CBLAS_TRANSPOSE transb, enum CBLAS_TRANSPOSE transc,
    int m, int n, int k, double alpha, double *A, int lda, double *B, int ldb,
    double beta, double *C, int ldc ) {
    QUARK_Insert_Task( quark, dgemm_tile_task, task_flags,
        sizeof(enum CBLAS_ORDER), &order, VALUE,
        sizeof(enum CBLAS_TRANSPOSE), &transa, VALUE,
        sizeof(enum CBLAS_TRANSPOSE), &transb, VALUE,
        sizeof(enum CBLAS_TRANSPOSE), &transc, VALUE,
        sizeof(int), &m, VALUE,
        sizeof(int), &n, VALUE,
        sizeof(int), &k, VALUE,
        sizeof(double), &alpha, VALUE,
        sizeof(double *), &A, INPUT,
        sizeof(int), &lda, VALUE,
        sizeof(double *), &B, INPUT,
        sizeof(int), &ldb, VALUE,
        sizeof(double), &beta, VALUE,
        sizeof(double *), &C, INOUT,
        sizeof(int), &ldc, VALUE, 0 );
}
```

```c
#include <omp.h>

int main(int argc, char** argv) {
    #pragma omp parallel
    #pragma omp master
    {
        ...
        for (int m = 1; m <= 8; m++)
            for (int n = 1; n <= 7; n++) {
                #pragma omp task depend( in:A(m,0)[0:nb*nb] ) \ 
                    depend( in:A(0,n)[0:nb*nb] ) \ 
                    depend( in:out:A(m,0)[0:nb*nb] )
                cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans,
                    nb, nb, nb, -1.0,
                    A(m, 0), nb, A(0, n), nb, 1.0, A(m, n), nb);
            }
        ...
    }
}

void dgemm_tile_task( Quark* quark ) { 
    enum CBLAS_ORDER order;
    enum CBLAS_TRANSPOSE transa, transb, transc;
    int m, n, k;
    quark_unpack_args_15(quark, order, transa, transb, transc,
        m, n, k,
        alpha, A, lda,
        B, ldb,
        beta , C, ldc );
    cblas_dgemm(order, transa, transb, transc,
        m, n, k,
        alpha, A, lda,
        B, ldb,
        beta, C, ldc );
}
#pragma omp parallel
#pragma omp master
{
    for (k = 0; k < nt; k++) {
        #pragma omp task depend(inout:A(k,k)[0:nb*nb])
        info = LAPACKE_dpotf_work(
            LAPACK_COL_MAJOR,
            lapack_const(PlasmaLower),
            nb, A(k,k), nb);
    }
    for (m = k+1; m < nt; m++) {
        #pragma omp task depend(inout:A(k,k)[0:nb*nb])
        
        cblas_dsyrk(
            CblasColMajor,
            CblasLower, CblasNoTrans,
            nb, nb,
            -1.0, A(k,k), nb,
            1.0, A(m,k), nb);
    }
    for (n = k+1; n < m; n++) {
        #pragma omp task depend(inout:A(m,k)[0:nb*nb])
        
        cblas_dgemm(
            CblasColMajor,
            CblasNoTrans, CblasTrans,
            nb, nb, nb,
            -1.0, A(m,k), nb,
            1.0, A(n,k), nb,
            1.0, A(m,n), nb);
    }
}
This research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy’s Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation’s exascale computing imperative.
## SLATE Objectives

<table>
<thead>
<tr>
<th><strong>Coverage</strong></th>
<th>ScaLAPACK and beyond</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Modern Hardware</strong></td>
<td>DOE CORAL (pre Exascale) → DOE Exascale</td>
</tr>
<tr>
<td><strong>Portability</strong></td>
<td>Intel Xeon (&amp;Phi), IBM POWER, ARM, NVIDIA, AMD, ...</td>
</tr>
<tr>
<td><strong>Modern Language</strong></td>
<td>C++11/14/17 (templates, STL, overloading, polymorphism, ...)</td>
</tr>
<tr>
<td><strong>Modern Standards</strong></td>
<td>MPI 3, OpenMP 4/5 (&amp;omp target)</td>
</tr>
<tr>
<td><strong>Performance</strong></td>
<td>80-90% of peak (asymptotic)</td>
</tr>
<tr>
<td><strong>Scalability</strong></td>
<td>full machine (tens of thousands of nodes)</td>
</tr>
<tr>
<td><strong>Productivity</strong></td>
<td>ca. 4 full time developers</td>
</tr>
<tr>
<td><strong>Maintainability</strong></td>
<td>part time developers + community</td>
</tr>
</tbody>
</table>

**can be built:**
- serial
- OpenMP multithreading
- MPI message passing
- GPU acceleration

---

38
SLATE Stack

molecular dynamics  computational chemistry  quantum mechanics  quantum chemistry  sparse solvers

EXAALT  NWChemEx  QMCPACK  GAMESS  FBSS

parallel dense linear algebra routines
distributed memory
multicore accelerators

SLATE

PARALLEL DENSE LINEAR ALGEBRA ROUTINES
DISTRIBUTED MEMORY
MULTICORE ACCELERATORS

PaRSEC  MPI  OpenMP

Exa MPI  SOLLVE

OMPI-X

BLAS++  LAPACK++  batch BLAS++

MKL  ESSL  cuBLAS  ACML

ECP  standards  vendor  SLATE
SLATE Resources

- main ECP website: https://exascaleproject.org
- main SLATE website: http://icl.utk.edu/slate/
- main SLATE repository: https://bitbucket.org/icl/slate
- BLAS++ repository: https://bitbucket.org/icl/blaspp
- LAPACK++ repository: https://bitbucket.org/icl/lapackpp
- SLATE Working Notes: http://www.icl.utk.edu/publications/series/swans
- Research Gate project: https://www.researchgate.net/project/ECP-SLATE
- SLATE User https://groups.google.com/a/icl.utk.edu/forum/#!forum/slate-user
Designing SLATE: Software for Linear Algebra Targeting Exascale

C++ API for BLAS and LAPACK

Roadmap for the Development of a Linear Algebra Library for Exascale Computing:

SLATE: Software for Linear Algebra Targeting Exascale
std::map<std::tuple<int64_t, int64_t, int>, Tile<FloatType>*> *tiles_;

- collection of tiles
- **individually allocated**
- only allocate what is needed
- accommodates: symmetric, triangular, band, ...

While in the PLASMA library the matrix is also stored in tiles, the tiles are laid out contiguously in memory.

In contrast, in SLATE, the tiles are individually allocated, with no correlation of their locations in the matrix to their addresses in memory.
std::map<std::tuple<int64_t, int64_t, int>, Tile<FloatType>*> *tiles_;

- distributed matrix
- global indexing of tiles
- only allocate the local part
- any distribution is possible (2D block cyclic by default)

The same structure, used for single node representation, naturally supports distributed memory representation.
GEMM Efficiency

LAPACK
MAGMA

SLATE

C = C – A \times B
C = C − A × B with small k, i.e., the DGEMM called in LU factorization.

The matrix fills out the GPU memory. The X axis shows the k dimension.
nested parallelism

top level: 

bottom level:

# pragma omp task

# pragma omp task

batch GEMM
### SLATE GPU Performance

**asymptotic scaling**

- **112 K × 112 K**: 1 node 4 GPUs
- **225 K × 225 K**: 4 nodes 16 GPUs
- **450 K × 450 K**: 16 nodes 64 GPUs

---

**SummitDev @ OLCF**

- **3 × 18 = 54 nodes** (IBM S822LC) ca. 0.5 TFLOPS (2.5%)
- **2 × 10 = 20 cores** (IBM POWER8) ca. 20 TFLOPS (97.5%)
- **4 GPUs** (NVIDIA P100)
- **256 GB DDR4**
- **4 × 16 = 64 GB HBM2**
- **NVLink 1.0** 80 GBPS (advertised)
- **GCC 7.1.0**
- **ESSL 5.5.0**
- **CUDA 8.0.54**
- **Spectrum MPI 10.1.0.3.**
SLATE GPU Trace

Cholesky factorization
20 cores + 4 GPUs
112 K × 112 K matrix
tile size of 512
SLATE Timeline

2016
Q1 research
Q2 design
Q3 prototyping

2017
Q4 C++ APIs for BLAS and LAPACK
Q1 parallel BLAS
Q2 parallel norms
Q3 linear systems (LU, LLT, LDLT)
Q4 least squares (CA-QR/LQ)

2018
Q1 mixed precision linear systems
Q2 matrix inversion
Q3 SVD

2019
Q4 EVP
Availability

- [http://icl.utk.edu/magma/](http://icl.utk.edu/magma/) download, documentation, forum
- [https://bitbucket.org/icl/magma](https://bitbucket.org/icl/magma) Mercurial repo

Support

- Linux, macOS, Windows
- CUDA >= 5.0; recommend latest CUDA
- CUDA architecture >= 2.0 (Fermi, Kepler, Maxwell, Pascal, Volta)
- 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)
```
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
   - Part of xSDK
   - `spack install magma`
   - Caveat: nvcc is picky about its host compiler (gcc, ...)

52
Outline

Availability
Routines
Code
Testers
Methodology
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
- Hermitian
- Orthogonal
- Unitary
- Positive definite
- Triangular

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

_gpu suffix for interface

Example: magma_zgesv_gpu
### 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</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>routine</td>
<td>CPU</td>
</tr>
<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>zhsv</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: \( A = \mathbf{U} \Sigma \mathbf{V}^H \)

<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/
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 $LL^T$)</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 / zunqgr</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/)
## 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>$|A|<em>1$, $|A|</em>{\text{inf}}$, $|A|<em>{\text{fro}}$, $|A|</em>{\text{max}}$</td>
<td>dlange (norm, general $A$)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dlansy (norm, symmetric $A$)</td>
</tr>
<tr>
<td></td>
<td>$B = A^T$ (out-of-place)</td>
<td>dtranspose</td>
</tr>
<tr>
<td></td>
<td>$A = A^T$ (in-place, square)</td>
<td>dtranspose_inplace</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

- \text{uplo} = \text{Lower or Upper}
- Entries in opposite triangle ignored
Simple example

Solve $AX = B$

- **Double precision, G**Eneral matrix, S**o**lVe (**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
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 );

    // ... fill in dA and dX (on GPU)

    // solve AX = B where B is in X
    int info;
    magma_dgesv_gpu( n, nrhs, 
                      dA, ldda, ipiv, 
                      dX, lddx, &info );
    if (info != 0) {
        throw std::exception();
    }

    // ... use result in dX
    magma_free( dA );
    magma_free( dX );
    delete[] ipiv;
    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

    // wait for gemm to finish
    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

// 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 );

  // ... cleanup
}
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
}
```
Outline
Availability
Routines
Code
Testers
Methodology
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|)
%========================================================================
| 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 | # warmup run |
| 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)
```plaintext
# (abbreviated output)
magma> cd testing
magma/testing> ./testing_dsymv -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

% uplo = Lower
%   N     MAGMA   Atomics    cuBLAS       CPU   error
%       Gflop/s   Gflop/s   Gflop/s   Gflop/s
%=====================================================================
| 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 | 17.55 | ok     |
| 7000 | 157.81| 200.44| 41.94 | 19.32 | ok     |
| 8000 | 169.54| 198.21| 41.78 | 19.12 | ok     |
| 9000 | 188.40| 216.07| 42.28 | 21.50 | 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 (dsymv)
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
...

One-sided factorizations

LU, Cholesky, QR factorizations for solving linear systems

Level 2
BLAS on
CPU

Level 3
BLAS on
GPU

Panel

DAG

critical
path
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 × 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

\[ y_i = A v_i \]

Panel

Trailing matrix

\[ A = Q^T A Q \]

column \( a_i \)
Numerical Linear Algebra (NLA) in 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

Large matrices

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

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

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

• 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

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

Small matrices / tensors
  - Fixed-size batches
  - Variable-size batches
  - Dynamic batches
  - Tensors
What about DLA on many small matrices?

Batched routines

- MAGMA provide a set of GPU only routines for batched computation
MAGMA Batched Computations

Need of Batched routines for Numerical LA
[e.g., sparse direct multifrontal methods, preconditioners for sparse iterative methods, tiled algorithms in dense linear algebra, etc.;]
[collaboration with Tim Davis at al., Texas A&M University]

Sparse / Dense Matrix System

DAG-based factorization

To capture main LA patterns needed in a numerical library for Batched LA

- LU, QR, or Cholesky on small diagonal matrices
- TRSMs, QRs, or LUs
- TRSMs, TRMMs
- Updates (Schur complement) GEMMs, SYRKs, TRMMs

- Example matrix from Quantum chromodynamics
- Reordered and ready for sparse direct multifrontal solver
- Diagonal blocks can be handled in parallel through batched LU, QR, or Cholesky factorizations
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):

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

- For every filter \( F_n \) and every channel, the computation for every pixel value \( O_{n,k} \) is a tensor contraction:

- 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)
MAGMA Batched: Tensor contractions for high-order FEM

Lagrangian Hydrodynamics in the BLAST code

On semi-discrete level our method can be written as

Momentum Conservation: \[
\frac{dv}{dt} = -M_\nu^{-1}F \cdot 1
\]

Energy Conservation: \[
\frac{de}{dt} = M_e^{-1}F^T \cdot v
\]

Equation of Motion: \[
\frac{dx}{dt} = v
\]

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


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

1. Non-batched computation
   - loop over the matrices one by one and compute using multithread (note that, since matrices are of small sizes there is not enough work for all the cores). So we expect low performance as well as threads contention might also affect the performance

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

MAGMA Batched Computations
1. Batched computation
Distribute all the matrices over the available resources by assigning a matrix to each group of core/TB to operate on it independently

- For very small matrices, assign a matrix/core (CPU) or per TB for GPU
- For medium size a matrix go to a team of cores (CPU) or many TB’s (GPU)
- For large size switch to multithreads classical 1 matrix per round.

Batched_dgemm(...)
MAGMA Batched Computations

1. Batched computation
Distribute all the matrices over the available resources by assigning a matrix to each group of core/TB to operate on it independently
• For very small matrices, assign a matrix/core (CPU) or per TB for GPU
• For medium size a matrix go to a team of cores (CPU) or many TB’s (GPU)
• For large size switch to multithreads classical 1 matrix per round.

Batched_dgemm(...)
1. Batched computation

Distribute all the matrices over the available resources by assigning a matrix to each group of core/TB to operate on it independently

- For very small matrices, assign a matrix/core (CPU) or per TB for GPU
- For medium size a matrix go to a team of cores (CPU) or many TB’s (GPU)
- For large size switch to multithreads classical 1 matrix per round.

Based on the kernel design that decide the number of TB or threads (GPU/CPU) and through the Nvidia/OpenMP scheduler

Batched_dgemm(...)

High percentage of the resources is used
MAGMA Batched Computations

68 cores Intel Xeon Phi KNL 7250, 1.3 GHz. DP peak is 2662 Gflop/s compiled with icc and using Intel MKL 2017

Batched dgemm BLAS 3
Standard dgemm BLAS 3

Small sizes
medium sizes
large sizes

Switch to non-batched

C = C + A*B

50~1000 matrices of size

Gflop/s

0 200 400 600 800 1000 1200 1400 1600 1800 2000

64 256 512 1000 1800 2200 2500 3000
MAGMA Batched Computations

Batched $dgemm$ BLAS 3

Standard $dgemm$ BLAS 3

Switch to non-batched

$C = C + A \times B$

50~1000 matrices of size

64 256 512 1000 1500 2000 2,500 3,000 3,500 4,000

Gflop/s

Small sizes

medium sizes

large sizes

1.2X

30X
MAGMA Batched Computations

Nvidia V100 GPU

Gflop/s

50~1000 matrices of size

- **Small sizes**: 1.4X
- **Medium sizes**: 20X
- **Large sizes**

Switch to non-batched

C = C + A*B

- **Batch dgemm BLAS 3**
- **Standard dgemm BLAS 3**
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)
Implementation on current hardware is becoming challenging

**Draft Reports**
Batched BLAS Draft Reports:  

**Batched BLAS Poster:**  

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

**Webpage on ReproBLAS:**  
[http://bebop.cs.berkeley.edu/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](http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-229.pdf)

---

**Workshop on Batched, Reproducible, and Reduced Precision BLAS**

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

Toward a standard Batched BLAS API

Status and goal

- Batched BLAS functionalities becomes a major factor in our community
  - Batched routines gradually make their steps into vendor libraries (Intel, Nvidia, etc) as well as into research software (MAGMA, Kokkos, etc)
- Today’s API differ significantly which can lead to poor portability
- Thus the community needs to make an effort to standardize the Batched BLAS API
Toward a standard Batched BLAS API

**Status and goal**

- Heterogeneity in the hardware (GPU, Phi, CPU) deeply complicates efforts to provide a standard interface
  - The calling interface may affect the implementation (performance) which depend on the architecture
- **Our objective today, is to try to define a cross-architecture standard without a severe performance penalty**
- Other API’s could be considered as auxiliary API’s or API with extra features
Matrices are stored BLAS-like \textit{“the usual storage that we know”}\footnote{array of pointers: that consists of a pointer to each matrix}

- Data could belong to one memory allocation
- Data could be anywhere, different allocations
- Matrices could be equidistant or not from each other
- Is suitable for CPU, GPU, Phi
- Accommodate most of the cases

- User has to fill-up the array of pointers
Matrices are stored BLAS like “the usual storage that we know”

- array of pointers: that consists of a pointer to each matrix
- strided: as one pointer to a memory and matrices are strided inside
  - Fixed stride
  - Variable stride
- Suitable for CPU, GPU, Phi

- For variable stride, user has to fill-up the array
- Cannot accommodate data that was not been allocated within the same chunk of memory. Think about adding matrices to the batch.
Toward a standard Batched BLAS API

Matrices are stored BLAS like “the usual storage that we know”
  - array of pointers: that consists of a pointer to each matrix
  - strided: as one pointer to a memory and matrices are strided inside

Matrices are stored in interleaved fashion or compact
  - data can be interleaved by batchcount or by chunk (SIMD, AVX, Warp)
  - Is only good for sizes less than 20 and only for some routines such as GEMM, TRSM, while it has performance and implementation issues for routines like LU or QR factorization
  - Requires user or implementation to convert/reshuffle the memory storage since most of the storage are BLAS-like
Toward a standard Batched BLAS API

API discussion

- Same or separate API for fixed and variable size batches?
  - Have two separate API’s?
  - Have a flag that switch between fixed and variable?

To simplify user life and avoid a combinatorial combination of parameter, we propose to **distinguish between fixed and variable size APIs**

```c
void batchedblas_dgemm_vbatched (  
    batched_trans_t transA, batched_trans_t transB,  
    batched_int_t *m, batched_int_t *n, batched_int_t *k,  
    double alpha,  
    double const * const *dA_array, batched_int_t *ldda,  
    double const * const *dB_array, batched_int_t *lddb,  
    double beta,  
    double **dC_array, batched_int_t *lddc,  
    batched_int_t batchCount, batched_queue_t queue );
```
Towards a standard Batched BLAS API

API discussion

- Same or separate API for fixed and variable size batches?
  - Have two separate API’s?
  - Have a flag that switch between fixed and variable?

- To simplify user life and avoid a combinatorial combination of parameter, we propose to **distinguish between fixed and variable size APIs**
# Toward a standard Batched BLAS API

- Consider GEMM as an example
- ‘F’ = Fixed, unified across a batch/group of problem
- ‘V’ = Variable, each problem has its assigned value

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>option args (transA, transB)</th>
<th>scaling args (alpha, beta)</th>
<th>sizes, ld's, inc’s (m, n, k) (lda, ldb, ldc)</th>
<th>data pointers (A, B, C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flag-based Flat</td>
<td>Standard</td>
<td>BATCH_FIXED</td>
<td>F</td>
<td>F</td>
<td>V</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BATCH_VAR</td>
<td>V</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>Group</td>
<td>MKL</td>
<td>per group</td>
<td>F</td>
<td>F</td>
<td>V</td>
</tr>
<tr>
<td></td>
<td></td>
<td>across groups</td>
<td>V</td>
<td>V</td>
<td>V</td>
</tr>
<tr>
<td>Flat</td>
<td>MAGMA</td>
<td>Fixed API</td>
<td>F</td>
<td>F</td>
<td>V</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Var. API</td>
<td>F</td>
<td>F</td>
<td>V</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cuBLAS</td>
<td>F</td>
<td>F</td>
<td>V</td>
</tr>
</tbody>
</table>
The goal is to have one API that allows any argument to be unified or varied across the batch/group.

We can support different scenarios through overloading and/or advanced standard containers:
- Overloading works, but will result in several APIs per each routine.

The main idea is to use the std::vector container:
- Every argument is std::vector<…>
- The size of the vector determines if it is unified or varied.

MAGMA Batched moving forward: C++ API
This API supports over 1000 possibilities for GEMM
• The size of C must be equal to batchCount
• The size of other arguments can be either 1 or batchCount
• Error checking through std::vector<blas_int> info
• The group API can be supported by promoting batchCount to be a std::vector

template<typename T>
void gemm_batch(
    std::vector<Op> const &transA, std::vector<Op> const &transB,
    std::vector<blas_int> const &m, std::vector<blas_int> const &n, std::vector<blas_int> const &k,
    std::vector<T> const &alpha,
    std::vector<T*> const &A, std::vector<blas_int> const &lda,
    std::vector<T*> const &B, std::vector<blas_int> const &ldb,
    std::vector<T> const &beta,
    std::vector<T*> const &C, std::vector<blas_int> const &ldc,
    const blas_int batchCount);
MAGMA Batched moving forward: C++ API

template<typename T>
void gemm_batch(
    std::vector<Op> const &transA, std::vector<Op> const &transB,
    std::vector<blas_int> const &m, std::vector<blas_int> const &n, std::vector<blas_int> const &k,
    std::vector<T> const &alpha,
    std::vector<T*> const &A, std::vector<blas_int> const &lda,
    std::vector<T*> const &B, std::vector<blas_int> const &ldb,
    std::vector<T> const &beta,
    std::vector<T*> const &C, std::vector<blas_int> const &ldc,
    const blas_int batchCount);

• **Pros:**
  
  1) Many possibilities through a single interface (opposed to overloading)
  2) Duplicate argument values are entirely eliminated
  3) Using a widely-used standard container in C++
     - Function to get the size, max., min., ... etc are already available
     - Also available for GPUs (e.g. Thrust Library for CUDA)
  4) A reference implementation for all possibilities is easily feasible
     - BLAS + OpenMP on CPUs
     - BLAS + Streams on GPUs
Cons:
1) Error checking can be an overhead
2) Should also check size consistencies
   • E.g. If the size of $A$ is 1, then $m$, $k$, and $\text{lda}$ should be of size 1 as well
3) The user has to be careful about the vector sizes
   • This is the price of a flexible API

```cpp
template<typename T>
void gemm_batch(
    std::vector<Op> const &transA, std::vector<Op> const &transB,
    std::vector<blas_int> const &m, std::vector<blas_int> const &n, std::vector<blas_int> const &k,
    std::vector<T> const &alpha,
    std::vector<T*> const &A, std::vector<blas_int> const &lda,
    std::vector<T*> const &B, std::vector<blas_int> const &ldb,
    std::vector<T> const &beta,
    std::vector<T*> const &C, std::vector<blas_int> const &ldc,
    const blas_int batchCount);
```
MAGMA Batched API

• API unification
  • Current batch APIs are divergent
    • MKL, MAGMA, cuBLAS, and others have their own distinct APIs
    • Flat vs group API?
    • Fixed vs. variable size?
    • Lots of possibilities -> explosion of interfaces
  • Standardization effort is ongoing
    • Community effort
    • Two batch BLAS workshops so far
    • First standard C API (http://eprints.maths.manchester.ac.uk/2464/)
    • More details (http://icl.utk.edu/bblas/)
MAGMA Batched API

• C++ API for batch routines
  • An effort to have one unified (standard) API
  • Overcomes C/Fortran shortcomings, by using standard containers and overloading
  • One API that supports fixed/variable/group interfaces
  • Heavily relies on `std::vector` container in C++
  • First draft is published (http://www.icl.utk.edu/publications/swan-004)
Classical strategies design

• For large problems the strategy is to prioritize the data-intensive operations to be executed by the accelerator and keep the memory-bound ones for the CPUs since the hierarchical caches are more appropriate to handle it.

Challenges

• Cannot be used here since matrices are very small and communication becomes expensive.

Proposition

• Develop a GPU-only implementation.
MAGMA Batched Computations

Classical strategies design

• For large problems performance is driven by the Level 3 BLAS (GEMM)

Challenges

• For batched small matrices it is more complicated

Proposition

• Rethink and Redesign both phases in a tuned efficient way
Key observations and current situation:

Classical strategies design

• A recommended way of writing efficient GPU kernels is to use the whole GPU’s shared memory, registers/TB – load it with data and reuse that data in computations as much as possible.

Challenges

• Our study and experience shows that this procedure provides very good performance for classical GPU kernels but is not that appealing for batched algorithm for different reasons.
MAGMA Batched Computations

Challenges

• Completely **saturating the shared memory** per SMX can decrease the performance of memory bound operations, since only one thread-block will be mapped to that SMX at a time (**low occupancy**)

• Due to the **limited parallelism** in the small matrices, the number of threads used in the thread block will be limited, resulting in **low occupancy**, and subsequently poor core utilization

• **Shared memory is small** (48KB/SMX) to fit the whole panel

• The panel involves **Non-GPU friendly** operations:
  • Vectors column (find the max, scale, norm, reduction)
  • Row interchanges (swap)
  • Small number of vectors (apply)

**Proposition:** **custom design per operations type**
MAGMA Batched Computations

GPU Optimization Summary

• **Hardware concepts**
  - CUDA core
  - Warp
  - Half-warp
  - Register file
  - Shared memory
  - Atomics
  - Shuffles
  - SMX

• **Software concepts**
  - Stream
  - Thread block
  - Kernel
  - Inlining
  - Intrinsics

• **Algorithmic concepts**
  - Blocking
  - Recursive blocking
  - Kernel replacement
  - Out-of-place operations
MAGMA Batched Computations

Batch GEMM, Batch = 100k, Tesla V100 GPU

Graph showing the performance of MAGMA and cuBLAS for batched GEMM operations with a batch size of 100k on a Tesla V100 GPU. The x-axis represents the matrix size, and the y-axis represents Gflop/s.
MAGMA Batched Computations

Batch LU, Batch = 1M, Tesla V100 GPU

Diagram showing the performance comparison between MAGMA and cuBLAS for batched LU operations on a Tesla V100 GPU, with Gflop/s on the y-axis and matrix size on the x-axis.
MAGMA Batched Computations

Batch QR, Batch = 1M, Tesla V100 GPU

![Graph showing performance comparison between MAGMA and cuBLAS for batch QR computations.](image-url)
MAGMA Batched Computations

Batch Cholesky, Batch = 500, Tesla V100 GPU, 20-Core Haswell CPU

- MAGMA
- mkl+openmp
MAGMA Batched Computations

Batch LU, Batch = 500, Tesla V100 GPU, 20-Core Haswell CPU

Matrix size vs. Gflop/s for MAGMA, mkl+openmp, and cublas.
MAGMA Batched Computations

Batch QR, Batch = 500, Tesla V100 GPU, 20-Core Haswell CPU

![Graph showing performance comparison between MAGMA, mkl+openmp, and cublas.](image)
MAGMA Batched Computations

Problem sizes influence algorithms & optimization techniques

Used in High-order (HO) Finite Element Methods (FEM)

Lagrangian Hydrodynamics in the BLAST code[1]
On semi-discrete level our method can be written as

\[
\begin{align*}
\text{Momentum Conservation:} & \quad \frac{dv}{dt} = -M_v^{-1}F - 1 \\
\text{Energy Conservation:} & \quad \frac{de}{dt} = M_e^{-1}F^T \cdot v \\
\text{Equation of Motion:} & \quad \frac{dx}{dt} = v
\end{align*}
\]

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


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:

Batch \{ \( C_{i3} = A^T B_{i1} \), for range of \( i3 \) \}

http://ceed.exascaleproject.org/
MAGMA Batched: fused kernels

Problem sizes influence algorithms & optimization techniques

Used in High-order (HO) Finite Element Methods (FEM)

Lagrangian Hydrodynamics in the BLAST code

On semi-discrete level our method can be written as

Momentum Conservation: \[
\frac{dv}{dt} = -M_e^{-1} F \cdot \mathbf{1}
\]

Energy Conservation: \[
\frac{de}{dt} = M_e^{-1} F^T \cdot \mathbf{v}
\]

Equation of Motion: \[
\frac{dx}{dt} = \mathbf{v}
\]

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

Matrix-free basis evaluation needs efficient tensor contractions,

\[
C_{i_1,i_2,j_3} = \sum_{k,j} A_{k,j,1} B_{k,j_2,j_3}
\]

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

Batch \{ \(C_{i_3} = A^T B_{i_3}\), for range of \(i_3\) \}

http://ceed.exascaleproject.org/
MAGMA now has GPU-only routines

GPU only routines

- MAGMA provide a set of GPU only routines such as Cholesky, LU and QR.

Reference:

1. A. Haidar, S. Tomov, P. Luszczek, and J. Dongarra,
   MAGMA Embedded: Towards a Dense Linear Algebra Library for Energy Efficient Extreme Computing,

2. Haidar, A. Abdelfatah, S. Tomov, and J. Dongarra.
   High-performance Cholesky factorization for GPU-only execution.
   In Proceedings of the General Purpose GPUs (GPGPU-10). ACM, New York, NY, USA, 42-52. DOI: https://doi.org/10.1145/3038228.3038237

3. A. Haidar, A. Abdelfattah, M. Zounon, S. Tomov, J. Dongarra,
   A Guide For Achieving High Performance With Very Small Matrices on GPU: A case Study of Batched LU and Cholesky Factorizations,
   IEEE Transactions on Parallel and Distributed Systems (TPDS in press 2018)

4. A. Abdelfattah, A. Haidar, S. Tomov, J. Dongarra,
   Analysis and Design Techniques for High-Performance and Energy-Efficient Dense Linear Solvers on GPUs
   Submitted to IEEE Transactions on Parallel and Distributed Systems (TPDS 2017)
MAGMA now has GPU-only routines

MAGMA LU factorization in double precision arithmetic

Reference:
DOI: https://doi.org/10.1145/3038228.3038237
Leveraging Half Precision in HPC on V100

MAGMA Mixed Precision algorithms

Idea: use lower precision to compute the expensive flops \((LU O(n^3))\) and then iteratively refine the solution in order to achieve the FP64 arithmetic

- Achieve higher performance \(\rightarrow\) faster time to solution
- Reduce power consumption by decreasing the execution time \(\rightarrow\) Energy Savings !!!

Reference:
   Investigating Half Precision Arithmetic to Accelerate Dense Linear System Solvers,  
2. Haidar, A., Tomov, S., Dongarra, J.  
   Leveraging Half Precision in HPC,  
   In preparation for ACM TOMS
LU factorization is used to solve a linear system $Ax=b$.

$A \ x = b$

$LUx = b$

$Ly = b$

then

$Ux = y$

Study of the LU factorization algorithm on Nvidia V100

Motivation

Leveraging Half Precision in HPC on V100
Iterative refinement for dense systems, $Ax = b$, can work this way.

$L U = lu(A)$
$x = U\backslash (L\backslash b)$
$r = b - Ax$

WHILE $\| r \|$ not small enough

1. find a correction “z” to adjust $x$ that satisfy $Az = r$

   solving $Az = r$ could be done by either:
   
   - $z = U\backslash (L\backslash r)$ Classical Iterative Refinement
   - GMRes preconditioned by the LU to solve $Az = r$ Iterative Refinement using GMRes

2. $x = x + z$

3. $r = b - Ax$

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.
Leveraging Half Precision in HPC on V100

Performance Behavior

Performance of solving $Ax=b$ using FP64 or IR with GMRes to achieve FP64 accuracy

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

• solving $Ax = b$ using FP64 LU
Leveraging Half Precision in HPC on V100
Performance Behavior

Performance of solving $Ax=b$ using FP64 or IR with GMRes to achieve FP64 accuracy

- $\text{FP64 dgesv}$
- $\text{FP32->64 dgesv}$

$\text{Flops} = \frac{2n^3}{3\text{ time}}$ 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
Leveraging Half Precision in HPC on V100
Performance Behavior

Performance of solving $Ax=b$ using FP64 or IR with GMRes to achieve FP64 accuracy

- 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

$\text{Flops} = \frac{2n^3}{3 \text{ time}}$ meaning twice higher is twice faster
Leveraging Half Precision in HPC on V100

Performance Behavior

Performance of solving $Ax = b$ using FP64 or IR with GMRes to achieve FP64 accuracy

- 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

Flops = $2n^3/(3 \times time)$ meaning twice higher is twice faster

Flops = $2n^3/(3 \times time)$ meaning twice higher is twice faster
Leveraging Half Precision in HPC

Power awareness

Mixed precision techniques can provide a large gain in energy efficiency

- Power consumption of the FP64 algorithm to solve $Ax=b$ for a matrix of size 34K, it achieve 5.5 Tflop/s and requires about 2021 joules providing about 14 Gflops/Watts.

- Power consumption of the mixed precision FP32→64 algorithm to solve $Ax=b$ for a matrix of size 34K, it achieve 10.7 Tflop/s and requires about 1041 joules providing about 30 Gflops/Watts.

- Power consumption of the mixed precision FP16→64 algorithm to solve $Ax=b$ for a matrix of size 34K, it achieve 16.8 Tflop/s and requires about 609 joules providing about 48 Gflops/Watts.

- Power consumption of the mixed precision FP16→64 TC algorithm using Tensor Cores to solve $Ax=b$ for a matrix of size 34K, it achieve 24 Tflop/s and requires about 470 joules providing about 74 Gflops/Watts.
MAGMA Sparse

- C-based, uses structures to pack complex objects
- Comprehensive set of SpMV formats/ routines
- Vector operations and building blocks
- Iterative solvers (Krylov, relaxation)
- Preconditioners
- Focus on GPU hardware

SpMV performance (DP)
Matrices from SuiteSparse Matrix Collection
NVIDIA P100, 5.3 GFLOPs peak (DP), 720 GB/s
Sparse linear algebra objects are handled with `magma_z_matrix` structure containing info about:
- memory location
- storage format
- data, size, nnz, ...
  - `magma_zmconvert(A, &B, Magma_CSR, Magma_ELL, queue);`
  - `magma_zmtransfer(A, &B, Magma_CPU, Magma_DEV, queue);`
  - `magma_z_spmv(alpha, A, x, beta, y, queue);`
  ...
Sparse linear algebra objects are handled with magma_z_matrix structure containing info about:

- memory location
- storage format
- data, size, nnz, ...

- magma_zmconvert( A, &B, Magma_CSR, Magma_ELL, queue );
- magma_zmtransfer( A, &B, Magma_CPU, Magma_DEV, queue );
- magma_z_spmv( alpha, A, x, beta, y, queue );

For calling a solver from MAGMA-sparse, an additional structure is used, containing information about the solver and the preconditioner:

- opts.solver_par.solver = Magma_CG;
- opts.solver_par.rtol = 1e-10;
- opts.solver_par.maxiter = 1000;
- opts.precond_par.solver = Magma_JACOBI;
- magma_z_solver( A, b, &x, &opts, queue );

All structures are defined and documented in magmasparse_types.h
MAGMA Sparse

// Initialize MAGMA and create some LA structures.
magma_init();
magma_queue_t queue; magma_queue_create( 0, &queue );

// Pass the system to MAGMA.
magma_dm_matrix A={Magma_CSR}, dA={Magma_CSR}, b={Magma_CSR}, db={Magma_CSR}, x={Magma_CSR}, dx={Magma_CSR};
magma_dmcsrset( m, n, row, col, val, &A, queue );
magma_dmvset( m, 1, rhs, &b, queue );
magma_dmvset( m, 1, sol, &x, queue );

// Copy the system to the device (optional, only necessary if using the GPU)
magma_dmvtransfer( A, &dA, Magma_CPU, Magma_DEV, queue );
magma_dmvtransfer( b, &db, Magma_CPU, Magma_DEV, queue );
magma_dmvtransfer( x, &dx, Magma_CPU, Magma_DEV, queue );

// Choose a solver, preconditioner, etc. - see documentation for options.
magma_dopts opts; opts.solver_par.solver = Magma_PCG; opts.precond_par.solver = Magma_JACOBI;
magma_dsvinfo_init( &opts.solver_par, &opts.precond_par, queue );
magma_dprecondsetup( dA, db, &opts.solver_par, &opts.precond_par, queue );

// to solve the system, run:
magma_dsvsolve( dA, db, &dx, &opts, queue );

// Then copy the solution back to the host and extract it to the application code
magma_dmvfree( &x, queue );
magma_dmvtransfer( dx, &x, Magma_CPU, Magma_DEV, queue );
magma_dmvget( x, &m, &n, &sol, queue );

// Free the allocated memory and finalize MAGMA
magma_dmvfree( &dx, queue ); magma_dmvfree( &db, queue ); magma_dmvfree( &dA, queue );
magma_queue_destroy( queue );
magma_finalize();

// From here on, the application code may continue with the solution in sol...
for (i = 0; i < 20; ++i) {
    printf("%.4f\n", sol[i]);
}
Parallel Preconditioning in MAGMA-sparse

Parallel Incomplete Factorization Preconditioners

- Kernel for computing \textbf{ILU preconditioners} in parallel.
- Implementations for hardware supporting \textbf{OpenMP} or \textbf{CUDA} available in \textbf{MAGMA-sparse}.
- Reported speedups up to 10x when comparing against NVIDIA’s \textbf{cuSPARSE} library (version 9.0).

Parallel Triangular solves

1. Based on approximating inverse of triangular factors (ISAI).
   - Replaces \texttt{trsv} with \texttt{SPAI}
   - Sparsity pattern & Accuracy of \texttt{SPAI} user-controlled

2. Based on Relaxation steps of parallel iterative method.
   - Jacobi / block-Jacobi
   - Block-Jacobi based on batched matrix inversion
   - Flexible block size, supervariable amalgation

Performance comparison on NVIDIA’s Volta architecture.

Diagonal blocks are inverted in-place
Sparse direct multifrontal solvers

DOD CREATE

“High-Performance Numerical Libraries with Support for Low-Rank Matrix Computations”

Accelerating StruMF HSS solver

- StruMF computes preconditioner for Hierarchical Semiseparable (HSS) matrices using low-rank approximations in a direct multifrontal solver
  - We developed and use a GPU-accelerated QP3 algorithm for the low-rank approximations
  - 2x performance improvement using K40c GPU vs. 16 Intel Sandy Bridge cores@2.6 GHz


Factorization times in a double precision HSS solver. The pairs of bars show times from using one to 16 cores. The bar on the left is the time on the CPU cores vs. the time on the right using the same number of CPU cores plus a Kepler GPU.
MAGMA-sparse interface to Trilinos

- **ECP PEEKS** effort develops interface between Trilinos and MAGMA-sparse.

- Higher goal: **Design a general standard interface for sparse LA.**

- Initial **interface design document** published in June 2017. (available on PEEKS project webpage [http://icl.utk.edu/peeks/](http://icl.utk.edu/peeks/))

- **Living document** - may be updated over time...
  
  Your feedback is welcome!

- Populate all functionality via the xSDK ecosystem.
Collaborators and Support

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

PLASMA team
http://icl.cs.utk.edu/plasma

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