clMAGMA: High Performance Dense Linear Algebra with OpenCL*

Chongxiao Cao
Innovative Computing Laboratory
University of Tennessee
Knoxville, TN
ccao1@utk.edu

Jack Dongarra†
Innovative Computing Laboratory
University of Tennessee
Knoxville, TN
dongarra@cs.utk.edu

Peng Du‡
Amazon.com
Seattle, WA
pdu@utk.edu

Mark Gates
Innovative Computing Laboratory
University of Tennessee
Knoxville, TN
mgates3@utk.edu

Piotr Luszczek
Innovative Computing Laboratory
University of Tennessee
Knoxville, TN
luszczek@eecs.utk.edu

Stanimire Tomov
Innovative Computing Laboratory
University of Tennessee
Knoxville, TN
tomov@cs.utk.edu

ABSTRACT
This paper presents the design and implementation of several fundamental dense linear algebra (DLA) algorithms in OpenCL. In particular, these are linear system solvers and eigenvalue problem solvers. Further, we give an overview of the clMAGMA library, an open source, high performance OpenCL library that incorporates the developments presented, and in general provides to heterogeneous architectures the DLA functionality of the popular LAPACK library. The LAPACK-compliance and use of OpenCL simplify the use of clMAGMA in applications, while providing them with portably performant DLA. High performance is obtained through use of the high-performance OpenCL BLAS, hardware and OpenCL-specific tuning, and a hybridization methodology where we split the algorithm into computational tasks of various granularities. Execution of those tasks is properly scheduled over the heterogeneous hardware components by minimizing data movements and mapping algorithmic requirements to the architectural strengths of the various heterogeneous hardware components.

*This research was sponsored by the National Science Foundation through the *Keeneland: National Institute for Experimental Computing* grant (award #0910735), the Department of Energy, and AMD.

†Also affiliated with the Oak Ridge National Laboratory, TN, USA and the University of Manchester, UK

‡Research completed while at the University of Tennessee, Knoxville

Categories and Subject Descriptors
G.4 [**Mathematical software**]: Algorithm design and analysis, Efficiency, Parallel implementations, Portability; G.1.3 [**Numerical analysis**]: Numerical linear algebra—*linear systems, matrix inversion, eigenvalues and eigenvectors*

1. INTRODUCTION
Solving linear systems of equations and eigenvalue problems is fundamental to scientific computing. The popular LAPACK library [5], and in particular its vendor optimized implementations like Intel's MKL [13] or AMD's ACML [3], have been the libraries of choice to provide these solvers for dense matrices on shared memory systems. This paper considers a redesign of the LAPACK algorithms and their OpenCL implementation to add efficient support for heterogeneous systems of multicore processors with GPU accelerators and coprocessors. This is not the first time that DLA libraries have needed a redesign to be efficient on new architectures – notable examples being the move from LINPACK [10] to LAPACK [5] in the 80's to make algorithms cache friendly, ScaLAPACK [8] in the 90's to support distributed memory systems, and now the PLASMA and MAGMA libraries [1] targeting efficiency on multicore and heterogeneous architectures, respectively.

The development of new high-performance numerical libraries is complex, accounting for the extreme level of parallelism, heterogeneity, and wide variety of accelerators and coprocessors available in current architectures. Challenges vary from new algorithmic designs to choices of programming models, languages, and frameworks that ease development, future maintenance, and portability. This paper addresses these issues while presenting our approach and algorithmic designs in the development of the clMAGMA [9] library.

To provide a uniform portability across a variety of GPU accelerators and coprocessors (e.g., Intel Xeon Phi), clMAGMA uses OpenCL [14]. OpenCL is an open standard for off-loading computations to accelerators, coprocessors, and multi/

manycore processors, and is maintained by the Khronos group with the backing of major hardware and software computer industry vendors. It offers portability across hardware and OS software. Although the use of OpenCL provides programming portability, cross-device performance portability is not guaranteed; we specifically address this in Section 2.

To deal with the extreme level of parallelism and heterogeneity in current architectures, clMAGMA uses a hybridization methodology, described in Section 3, where we split the algorithms of interest into computational tasks of various granularities, and properly schedule those tasks' execution over the heterogeneous hardware. Thus, we use a Directed Acyclic Graph (DAG) approach to parallelism and scheduling that has been developed and successfully used for dense linear algebra libraries such as PLASMA and MAGMA [1], as well as in general task-based approaches to parallelism, such as runtime systems like StarPU [6] and SMPSs [7].

Besides the general cross-device considerations addressed in Section 2, obtaining high performance in OpenCL depends on a combination of algorithm and hardware-specific optimizations, discussed in Section 4. The implication of this on software, in order to maintain its performance portability across hardware, is the need to build in it algorithmic variations that are tunable, e.g., at installation time. This is the basis of autotuning, an example of these advanced optimization techniques.

A performance study on AMD hardware is presented in Section 5. Besides verifying our approaches and confirming the appeal of OpenCL and accelerators for high-performance DLA, the results open up a number of future work opportunities discussed in our conclusions.

2. CROSS-DEVICE CONSIDERATIONS

A recommended approach to developing a high-performance and easy to maintain DLA library is to express the algorithms of interest in terms of the BLAS standard. Performance portability is then obtained through the use of architecture-specific, highly tuned BLAS implementations (e.g., MKL from Intel or ACML from AMD). LAPACK and ScaLAPACK have demonstrated this over the years, and now we see it in the new MAGMA and PLASMA libraries. The clMAGMA library takes the same approach, and therefore performance portability relies on the availability of portable OpenCL BLAS, discussed in Section 2.1. Specifics related to OpenCL and its implementation are also important for obtaining high-performance and must be addressed while designing and tuning OpenCL algorithms. Well designed microbenchmarks, shown in Section 2.2, can be used to obtain these key OpenCL specifics to achieving high performance.

2.1 Portable OpenCL BLAS

The Automatically Tuned Linear Algebra Software (ATLAS) library [19] is a BLAS implementation for CPUs. ATLAS achieves portable performance across CPUs mainly by relying on empirical autotuning. Still, vendor libraries like MKL and ACML, optimized for their specific architectures, provide higher performance implementations. The same is true with OpenCL BLAS implementations – OpenCL provides software portability, but unless tuned for a particular architecture, optimization opportunities can be missed.

Currently, the most complete OpenCL BLAS implementation is AMD's clAmdBlas, provided through the AMD's Accelerated Parallel Processing Math Libraries (APPML) [2]. It can be used on architectures other than AMD, but its tuning, and therefore highest efficiency, is on AMD hardware. The potential of OpenCL to express BLAS algorithms (vs. other, lower level access to the hardware languages) while obtaining high performance is evident through the clAmdBlas. Other implementations, e.g., from Nakasato et al. [16, 15], confirm this by obtaining impressive high performance matrix-matrix multiplication (GEMM). In particular, the highest performance that we are aware of has been demonstrated by Matsumoto et al. [15] – their OpenCL DGEMM reaches up to 848 Gflop/s, and SGEMM up to 2,646 Gflop/s, which is 90% and 70% of the double and single precision peak, respectively, of AMD's Tahiti GPU (Radeon HD 7970).

In previous work, we evaluated OpenCL as a programming tool for performance-portable BLAS [11]. Triangular solvers (TRSM) and GEMMs were developed in OpenCL, tuned for a specific device, and compared. The conclusion was that OpenCL environment setup overhead is large and should be minimized, e.g., by preprocessing or localized in library initialization routines. More importantly, the performance results presented confirmed the conclusion above that OpenCL is expressive enough for developing high performance BLAS, so long as architectural specifics are taken into account in the algorithm design. Even though good performance should not be expected from blindly running algorithms on a new platform, autotuning heuristics can help to improve performance on a single platform.

Autotuning mechanisms are already provided in clAmdBlas through a tuning tool that the user can run to produce optimized OpenCL BLAS on the architecture of interest. Thus, as performance portability of OpenCL BLAS can be obtained, organizing higher-level libraries like clMAGMA in terms of OpenCL BLAS can ensure their performance portability as well.

2.2 Microbenchmarks

We developed a number of microbenchmarks to help us gain a better understanding of OpenCL and to guide our algorithm design and tuning. We describe two benchmarks that can be key for performance – kernel launch overhead and CPU-GPU data transfer. To add some context to the measurements reported, we include comparisons with corresponding CUDA measurements.

2.2.1 Kernel launch overhead

The average time to asynchronously invoke an OpenCL 1.2 AMD-APP (1016.4) kernel on an AMD Tahiti GPU (Radeon HD 7900 Series) is 1.0–1.5μs. This was measured by asynchronously launching an empty kernel a large number of times and synchronizing at the end. The overhead increases to 120μs when synchronizing after each kernel invocation. (using a PCIe 2.0 CPU-GPU interface). Similar benchmarks for CUDA 4.2 [18] showed an overhead of 3–7μs with no synchronization between kernels, and 10–14μs with synchronization between kernels.

We also benchmarked the kernel launch overhead for four BLAS functions: DGEMM, DTRSM, DTRMM and DSYRK,

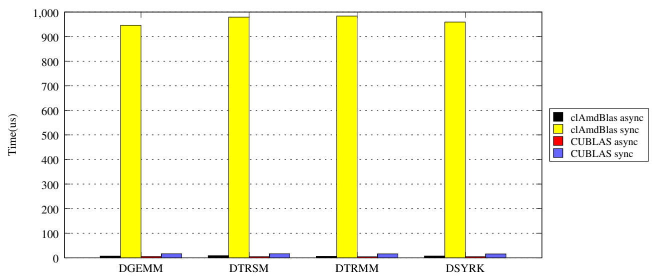

Figure 1: GPU BLAS functions launch overhead for clAmdBlas 1.8.286 with OpenCL 1.2 AMD-APP (1016.4) on Radeon HD 7970 and CUBLAS 4.2 on Tesla S2050, using PCIe 2.0 CPU-GPU interface.

which are used in the double precision LU, Cholesky and QR factorizations. In order to compare with OpenCL, the benchmark for CUDA was tested on a NVIDIA Fermi GPU (Tesla S2050). Results for kernel launch overhead of OpenCL and CUDA BLAS functions are shown in Figure 1. The OpenCL BLAS functions are from AMD's clAmdBlas 1.8.286 and the CUDA functions are from CUBLAS 4.2. The BLAS functions in clAmdBlas have 6–9 μs asynchronous launch overhead versus 4–5μs in CUBLAS. For synchronous launch overhead, CUBLAS takes only 14–15μs, while clAmdBlas increases hugely to 940–980μs. Both of these measurements are using a PCIe 2.0 CPU to GPU interface. We can see that the synchronous kernel launch overhead is very expensive in OpenCL, especially for the BLAS functions.

2.2.2 CPU-GPU data transfer overhead

Transfer time for contiguous data between CPU and GPU can be modeled as

$$\text{time} = \text{latency} + \frac{\text{bytes transferred}}{\text{PCIe bandwidth}}. \quad (1)$$

On our system, an AMD Radeon HD 7970 card on a PCIe 2.0 interface, the measured PCIe bandwidth was 2.82 GB/s from CPU to GPU and 3.29 GB/s from GPU to CPU. We found that the latency was 50–60μs from CPU to GPU and 140–150μs from GPU to CPU. Benchmarks for CUDA [18] showed that it had 10–17μs latency, which we verified on our CUDA system (NVIDIA Tesla S2050 on the PCIe 2.0 interface) as 13–14μs latency in both directions, which is much smaller than OpenCL.

3. DENSE LINEAR ALGEBRA IN OPENCL

3.1 Hybridization methodology

The hybridization methodology used in MAGMA [17] and now in clMAGMA is an extension of the task-based approach for parallelism and developing DLA on homogeneous multicore systems [1]. In particular,

- The computation is split into BLAS-based tasks of various granularities, with their data dependencies, as shown in Figure 2.
- Small, non-parallelizable tasks with significant control-flow are scheduled on the CPUs.
- Large, parallelizable tasks are scheduled on GPUs.

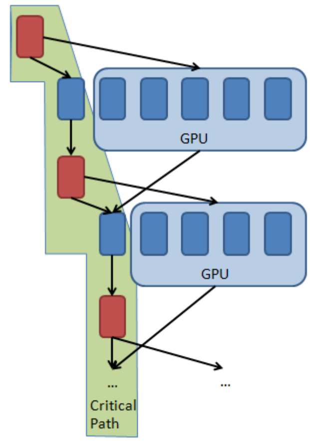

Figure 2: DLA algorithm as a collection of BLAS-based tasks and their dependencies. The algorithm's critical path is, in general, scheduled on the CPUs, and large data-parallel tasks on the GPUs.

The difference with multicore algorithms is the task splitting, which here are of various granularities to make different tasks suitable for particular architectures, and the scheduling itself. Specific algorithms using this methodology, and covering the main classes of DLA, are described in the subsections below.

3.2 The clMAGMA design and functionality

The clMAGMA interface is similar to LAPACK. For example, compare LAPACK's LU factorization interface vs. clMAGMA's:

```
lapackf77_dgetrf(&M,&N, hA,   &lda, ipiv, &info)
magma_dgetrf_gpu( M, N, dA,0, ldda, ipiv, &info, queue)
```

Here hA is the typical CPU pointer (`double *`) to the matrix of interest in the CPU memory and dA is a pointer in the GPU memory (`magmaDouble_ptr`). The last argument in every clMAGMA call is an OpenCL queue, through which the computation will be streamed on the GPU (`magma_queue_t`).

To abstract the user from knowing OpenCL, all OpenCL data types and main functions, such as BLAS, CPU-GPU data transfers, and memory allocations and deallocations, are redefined in terms of clMAGMA data types and functions. This design allows us to more easily port the MAGMA library to clMAGMA, and eventually to merge them while maintaining a single source. Also, the clMAGMA wrappers are often simpler than the corresponding OpenCL functions, and provide a complete set of functions for programming hybrid high-performance numerical libraries. Thus, not only users but application developers as well can opt to use the clMAGMA wrappers without knowing OpenCL.

clMAGMA provides the standard four floating point arithmetic precisions – single real, double real, single complex, and double complex. There are routines for the so called one-sided factorizations (LU, QR, and Cholesky), two-sided factorizations (Hessenberg, bi-, and tridiagonal reductions), linear system and least squares solvers, matrix inversions, symmetric and nonsymmetric standard eigenvalue problems, SVD, and orthogonal transformation routines, all described in the subsections below.

As discussed in [11], compiling OpenCL kernel from source file introduces significant amount of overhead. By caching the Intermediate Representation (IR) resulting from clGetProgramInfo to disk and loading at runtime, overhead can be effectively reduced. AMD and NVIDIA's OpenCL implementations both allow such maneuver, which is essential for the performance of clMAGMA since GPU kernels could be repeated called in different routines. An efficient way to handle the kernel compiling and catching is required. In clMAGMA, a runtime system is implemented to fulfill this task.

The runtime system, coded in C++ as a singleton class, provides two functionalities depending on usage phases: during installation, runtime system compiles OpenCL source files into IRs and stores them to disk; during execution time, the runtime system loads IRs to memory and further builds them into platform specific executables. At the beginning of user level program, the runtime system compiles IR loaded from disk and setups mapping between the name of the OpenCL kernel and its platform specific executables through a series of hashtables. This initialization process only executes once to avoid repeated compiling and allow reusing executables across different higher level routines.

3.3 LU, QR, and Cholesky factorizations

The one-sided factorizations routines implemented and currently available through clMAGMA are:

`magma_zgetrf_gpu` computes an LU factorization of a general M-by-N matrix A using partial pivoting with row interchanges;

`magma_zgeqrf_gpu` computes a QR factorization of a general M-by-N matrix A;

`magma_zpotrf_gpu` computes the Cholesky factorization of a complex Hermitian positive definite matrix A.

Routines in all standard four floating point precision arithmetics are available, following LAPACK's naming convention. Namely, the first letter of the routine name (after the prefix `magma_`) indicates the precision – z, c, d, or s for correspondingly double complex, single complex, double real, or single real. The suffix `_gpu` indicates that the input matrix and the output are on the GPU memory.

The typical hybrid computation and communication pattern for the one-sided factorizations (LU, QR and Cholesky) is shown in Figure 3. At a given iteration, panel dP is copied to the CPU and factored using LAPACK, and the result is copied back to GPU. The trailing matrix, consisting of the next panel T_1 and submatrix T_2, is updated on the GPU. After receiving dP back from the CPU, T_1 is updated first using dP and the result is sent to the CPU (as being the next panel to be factored there). While the CPU starts the factorization of T_1, the rest of trailing matrix, T_2, is updated on the GPU in parallel with the CPU factorization of panel T_1. In this pattern, only data to the right of the current

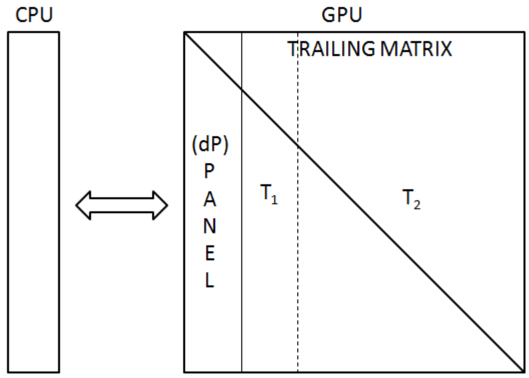

Figure 3: Typical computational pattern for the hybrid one-sided factorizations in clMAGMA.

panel is accessed and modified, and the factorizations that use it are known as right-looking. The computation can be organized differently – to access and modify data only to the left of the panel – in which case the factorizations are known as left-looking.

An example of a left-looking factorization, demonstrating a hybrid algorithm implementation, is given in Figure 4 for the Cholesky factorization. Copying the panel to the CPU, in this case just a square block on the diagonal, is done on line 4. The data transfer is asynchronous, so before we factor it on the CPU (line 8), we synchronize on line 7 to enforce that the data has arrived. Note that the CPU work from line 8 is overlapped with the GPU work on line 6. This is indeed the case because line 6 is an asynchronous call/request from the CPU to the GPU to start a ZGEMM operation. Thus control is passed to lines 7 and 8 while the GPU is performing the ZGEMM. The resulting factored panel from the CPU work is sent to the GPU on line 11 and used on line 14, after making sure that it has arrived (the sync on line 13).

```
1   for ( j=0; j<n; j += nb) {
2         jb = min(nb, n – j);
3         magma_zherk( MagmaUpper, MagmaConjTrans,
                          jb, j, m_one, dA(0, j), ldda, one, dA(j, j), ldda, queue );
4         magma_zgetmatrix_async( jb, jb, dA(j,j), ldda, work, 0, jb, queue, &event );
5         if ( j+jb < n )
6             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 );
7         magma_event_sync( event );
8         lapackf77_zpotrf( MagmaUpperStr, &jb, work, &jb, info );
9         if ( *info != 0 )
10              *info += j;
11        magma_zsetmatrix_async( jb, jb, work, 0, jb, dA(j,j), ldda, queue, &event );
12        if ( j+jb < n ) {
13            magma_event_sync( event );
14            magma_ztrsm( MagmaLeft, MagmaUpper, MagmaConjTrans, MagmaNonUnit,
                          jb, n-j-jb, z_one, dA(j, j), ldda, dA(j, j+jb), ldda, queue );
          }
    }
```

Figure 4: Cholesky factorization in clMAGMA.

3.4 Orthogonal transformation routines

The orthogonal transformation routines implemented and currently available through clMAGMA are:

magma_zungqr[_gpu] generates an M-by-N matrix Q with orthonormal columns, which is defined as the first N columns of a product of K elementary reflectors of order M as returned by `magma_zgeqrf_gpu`;

magma_zunmqr[_gpu] overwrites a general complex M-by-N matrix C with QC or CQ, where Q can also be transposed or not.

The routines are available in all four precisions, and in both CPU (input and output is on the CPU) and GPU interfaces.

Typical uses of the QR factorization require computing the product QC for some matrix C (the `zunmqr` routine). For efficiency, the matrix Q is represented implicitly as a product of block Householder reflectors of the form $I - V_i T_i V_i^T$, for $i = 1, \ldots, k$. Instead of forming Q explicitly and then performing a matrix-matrix multiplication, it is cheaper to apply the block Householder reflectors directly. Applying each reflector requires three matrix-matrix multiplies, which clMAGMA performs on the GPU. The V matrices are tall and skinny, with the upper triangle logically zero, as shown in Figure 5. In LAPACK, the upper triangle of each V contains of the R matrix; in clMAGMA, when the V is copied to the GPU, the upper triangle is explicitly set to zero. This allows us to simplify the code and improve performance using a single GEMM, instead of a less-efficient triangular multiply (TRMM) and a GEMM. The only work on the CPU is computing the T_i matrices when necessary.

If the Q matrix is needed explicitly, clMAGMA can compute it (the `zungqr` routine) by multiplying the implicitly-represented Q with identity matrix I. This is done in a block-by-block fashion in order to be done in-place, overwriting the implicit Q (the V Householder vectors) with the explicit Q.

Similar routines are used by clMAGMA in the eigenvalue and SVD problems, where orthogonal transformations are applied to back transform eigenvectors and singular vectors.

3.5 Hessenberg, bi- and tridiagonal reductions

The two-sided factorizations routines implemented and currently available through clMAGMA are:

magma_zgehrd reduces a general matrix A to upper Hessenberg form H by orthogonal similarity transformations;

magma_zhetrd reduces a Hermitian matrix A to real symmetric tridiagonal form T by orthogonal similarity transformations;

magma_zgebrd reduces a general M-by-N matrix A to upper or lower bidiagonal form B by orthogonal transformations.

The routines are available in all four precisions.

The Hessenberg, bidiagonal, and tridiagonal reductions are two-sided factorizations used in the nonsymmetric eigenvalue, symmetric eigenvalue, and SVD problem, respectively. The standard one-stage approach to solving the nonsymmetric eigenvalue problem applies an orthogonal transformation Q on both sides of the matrix A to reduce it to upper Hessenberg form, $H = QAQ^T$. QR iteration is then used to find the eigenvalues and eigenvectors of H; the eigenvalues of H are the same as the eigenvalues of A, while the eigenvectors can be back-transformed using Q to find the eigenvectors of A.

Unlike the QR factorization, where the panel factorization is independent of the trailing matrix, in the Hessenberg reduction, each column of the panel requires a matrix-vector product (GEMV) with the trailing matrix. We take advantage of the high bandwidth of GPUs to accelerate these memory-bound GEMV operations during the panel factorization. The algorithm is shown schematically in Figure 5. A panel dP_i is copied from the GPU to the CPU (step 1). For each column j of the panel, a Householder vector v_j is computed (step 2) and the matrix-vector product $y_j = A_j v_j$ is computed with the trailing matrix on the GPU (step 3). After the panel factorization, the block Householder reflector is applied with several GEMMs to update the trailing matrix, and completed portions of the trailing matrix are copied back to the CPU (step 4). Note that in this pattern the communication-to-computation is in a surface-to-volume ratio – sending a vector of length n is followed by $2n^2$ flops (in the inner loop), and sending a panel of size $n \times nb$ is followed by $O(n^2 \times nb)$ flops (in the outer loop).

Similarly, the symmetric eigenvalue problem involves an initial reduction to tridiagonal form, and the SVD involves an initial reduction to bidiagonal form. The exact details differ from the Hessenberg factorization, but the panel factorization similarly involves matrix-vector products (GEMV or SYMV), which clMAGMA performs on the GPU to take advantage of its high memory bandwidth.

Recent success in MAGMA with two-stage algorithms for the tridiagonal reduction [12] demonstrate that we can recast it using compute-bound Level-3 BLAS SYMM operations, instead of memory-bound Level-2 BLAS SYMV operations. This provides a large speed boost compared to the traditional one-stage algorithm. Future work for clMAGMA involves porting these two-stage algorithms, where we expect a similar

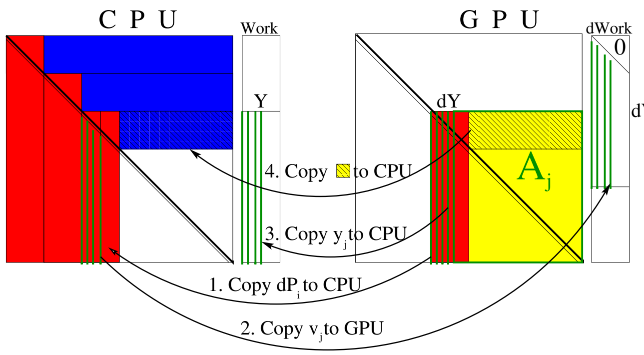

Figure 5: Typical communication pattern for the hybrid two-sided factorizations in clMAGMA.

speed increase.

3.6 Linear system and eigenproblem solvers

The one- and two-sided factorizations are the major building blocks for developing correspondingly linear system and eigenproblem solvers. We have developed the following solvers:

`magma_zpotrs_gpu` solves a system of linear equations $Ax = B$ with a Hermitian positive definite matrix A using the Cholesky factorization of A;

`magma_zgetrs_gpu` solves a system of linear equations with general N-by-N matrix A using the LU factorization of A;

`magma_zgels_gpu` solves the overdetermined least squares problem, min $||Ax - B||$, using the QR factorization of A;

`magma_zheevd` computes all eigenvalues and, optionally, eigenvectors of a complex Hermitian matrix A. If eigenvectors are desired, it uses a divide and conquer algorithm;

`magma_zgeev` computes the eigenvalues and, optionally, the left and/or right eigenvectors for an N-by-N complex nonsymmetric matrix A;

`magma_zgesvd` computes the singular value decomposition (SVD) of a complex M-by-N matrix A, optionally computing the left and/or right singular vectors.

The routines are available in all four precisions. The linear solvers use the hybrid clMAGMA one-sided factorization routines and triangular matrix solvers, as provided from OpenCL BLAS implementations. The eigenproblem solvers use the hybrid clMAGMA two-sided factorizations, which are the most time consuming parts of the algorithms. The rest is run on the CPUs, using vendor optimized LAPACK.

Related to the linear solvers, clMAGMA provides matrix inversion routines as well. These are the:

`magma_ztrtri_gpu` for computing the inverse of a real upper or lower triangular matrix;

`magma_zgetri_gpu` for computing the inverse of a matrix using the LU factorization computed by `magma_zgetrf_gpu`;

`magma_zpotri_gpu` for computing the inverse of a real symmetric positive definite matrix using its Cholesky factorization computed by `magma_zpotrf_gpu`.

The triangular inverse routine is a hybrid, derived from the corresponding block LAPACK algorithm. The diagonal blocks of the matrix are sent and inverted on the CPU, and everything else is on the GPU. The LU inversion uses `magma_ztrtri_gpu` to invert U and then computes inv(A) by solving the system inv(A)L = inv(U) for inv(A) (entirely on the GPU). The `magma_zpotri_gpu` also uses `magma_ztrtri_gpu` to invert the upper (U) or lower (L) factor of the Cholesky factorization, and a hybrid code (`magma_zlauum_gpu`) to compute the product UU' or $L'L$.

4. ADVANCED OPTIMIZATIONS

We highlight three optimization techniques that are crucial for obtaining high performance. The first one, overlapping CPU-GPU communications with GPU computations, is important because of the slow CPU-GPU interconnect relative to the GPU performance capabilities. For example, sending $O(1)$ bytes between the CPU and GPU without overlap can result in losing the opportunity to compute $O(100)$ double precision flops on the GPU. The second one, overlapping CPU and GPU work, allows us to use the entire system more efficiently. Finally, autotuning is a technique that saves tuning time and enables cross-device performance portability.

4.1 Overlapping CPU-GPU communications with GPU computation

In Section 2, we saw that OpenCL can have higher CPU-GPU data transfer latency overhead than CUDA, which can reduce the effective bandwidth when a small size of data is transferred between the CPU and GPU. Thus, this can become a performance bottleneck, unless overlapped with GPU work (or minimized with other optimization techniques). Figure 6 shows part of the trace of a double precision LU factorization in clMAGMA: the first row is the CPU work, where the black color represents the time of panel factorization; the second row is the GPU work, where the red color represents DGEMM operations and green color represents DTRSM. Yellow is copying data from GPU to CPU and grey is copying data from CPU to GPU. Although computation on the CPU has overlapped with the GPU, communication and computation on the GPU are executed sequentially.

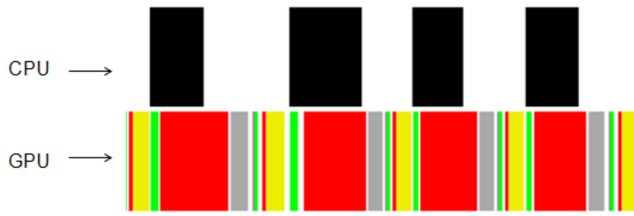

Figure 6: Partial CPU-GPU execution trace of a hybrid LU factorization in clMAGMA. Yellow and gray represent CPU-GPU communications that in this case are not overlapped with GPU work.

In OpenCL, performing work with a device, such as executing kernels or moving data to and from the device's local memory, is done using a corresponding command queue [4]. A command queue is an interface for a specific device and its associated work. A way to overlap CPU-GPU communication and GPU computation is by creating two command queues. One queue is used for data transfers and the other is used for kernel computations. Figure 7 shows part of the trace of double precision LU factorization similar to Figure 6, but here we have applied the two queues optimization. The first row is again the CPU work, the second row is the computation work of queue 1 on the GPU, and the third row is the communication work of queue 2. All color definitions are the same as in Figure 7. Note that based on this two queues technique, we made the communication overlap with the GPU computation work. Experiments showed that this approach lead to about 10% increase of performance for double precision LU factorization.

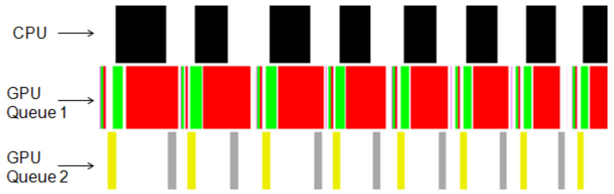

Figure 7: Partial CPU-GPU execution trace of a hybrid LU factorization in clMAGMA based on the two command queues' optimization, overlapping CPU-GPU data transfers (the yellow and gray transfers in GPU Queue 2) with GPU work (in GPU Queue 1).

From the above two traces, we also notice that there are some blank gaps between different kernels on the GPU. Those represent overheads of kernel switching on the GPU.

4.2 Overlapping CPU and GPU work

In OpenCL, the host creates a data structure called a command-queue to coordinate execution of the kernels on the devices. The host places commands into the command-queue which are then scheduled onto the devices. For example, in Figure 4, line 6 puts a ZGEMM in the command-queue `queue`. The host still must submit the ZGEMM to the device for execution, but this may not happen immediately. As a result, the CPU can start the computation at line 8 while the device has not started the ZGEMM. Thus, although our high-level algorithm is designed to overlap CPU and GPU work, overlap may not happen in practice. In order to force the command-queue to immediately submit the command queued to the appropriate device, one must call `clFlush(queue)` [4]. Therefore, all clMAGMA BLAS wrappers first queue the corresponding OpenCL BLAS and immediately post a `clFlush` on the queue.

The importance of overlapping CPU and GPU work is quantified in Figure 8 for the case of LU factorization in double precision (the dgetrf routine). The blue curve is the performance of dgetrf without CPU and GPU work overlap. It achieves up to 195 Gflop/s. The red curve is the performance of dgetrf with overlapping CPU and GPU work, using `clFlush`. It achieves up to 280 Gflop/s, i.e., getting about 1.4× speedup.

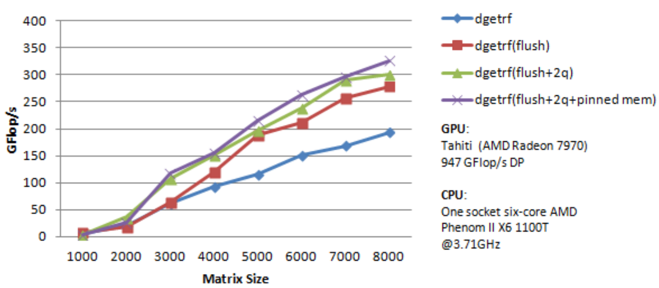

Figure 8: Advanced performance optimizations of dgetrf in clMAGMA.

Figure 8 also shows the effect of further optimizations, and in particular the techniques of using two queues to overlap CPU-GPU communications with GPU computation (from the previous subsection), and using pinned memory to get higher transfer throughput between CPU and GPU. Putting all these optimizations together, the performance of dgetrf is shown by the purple curve. It achieves up to 326 Gflop/s, which is almost 1.6× speedup compared to the original version without any optimizations.

4.3 Autotuning

While functionality of OpenCL is portable, the resulting performance often is not. Furthermore, it is commonly sufficient to rely on highly optimized BLAS that are provided by the vendor to guarantee transportable efficiency with respect to the peak performance. This is clearly predicated on the fact that the BLAS is of high quality and is capable of providing very efficient execution across a wide range of input parameters including matrix dimensions and data-dependent characteristics such as symmetry or transposition. In practice, this requirement is often not fulfilled and it is necessary to use customized versions of some of the kernels or maybe just one specific instance of the kernel for particular matrix shapes.

5. PERFORMANCE STUDY

The performance results provided in this section use AMD's Radeon HD 7970 card and its multicore host, a single socket six-core AMD Phenom II X6 1100T CPU running at 3.71 GHz. Kernels executed on the CPU use LAPACK and BLAS from MKL 11.1, and BLAS kernels executed on the GPU are from clAmdBlas 1.8. The OpenCL version is 1.2. We installed AMD-APP 1016.4 as the OpenCL driver. Currently the AMD OpenCL driver for Linux has a 512 MB maximum limitation for a single memory allocation on the GPU, so in our experiment we only tested matrix sizes of up to 8,000 (in double precision arithmetic).

The performance of double precision LU factorization in clMAGMA is given in Figure 9. It achieves up to 326 Gflop/s, getting about 5.7× speedup versus the CPU host.

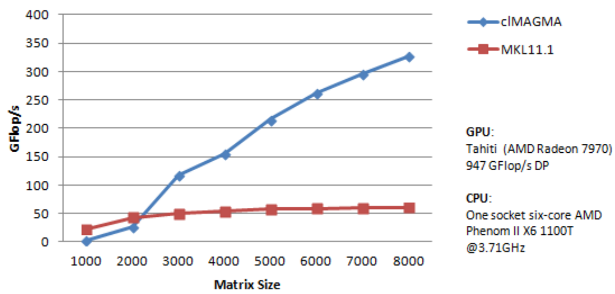

Figure 9: Performance of clMAGMA's LU factorization in double precision vs. MKL 11.1

The performance of the double precision Cholesky factorization in clMAGMA is shown in Figure 10. It achieves up to 344 Gflop/s, getting about 5.4× speedup versus the CPU host.

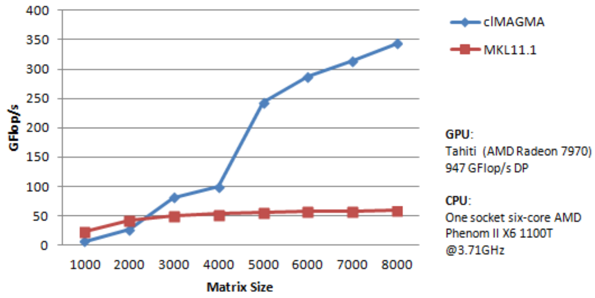

Figure 10: Performance of clMAGMA's Cholesky factorization in double precision vs. MKL 11.1

The performance of the double precision QR factorization in clMAGMA is shown in Figure 11. It achieves up to 347 Gflop/s, getting about 5.9× speedup versus the CPU host.

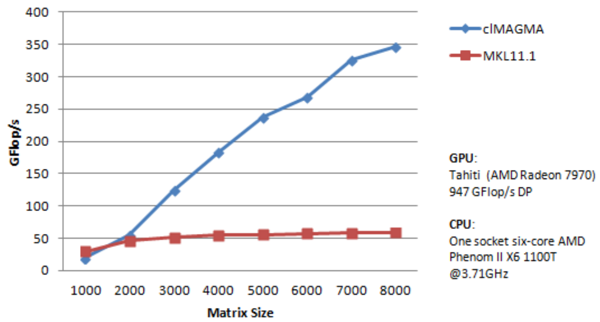

Figure 11: Performance of clMAGMA's QR factorization in double precision vs. MKL 11.1

The performance of the double precision Hessenberg factorization in clMAGMA is shown in Figure 12. It achieves up to 40 Gflop/s, getting about 5.5× speedup versus the CPU host.

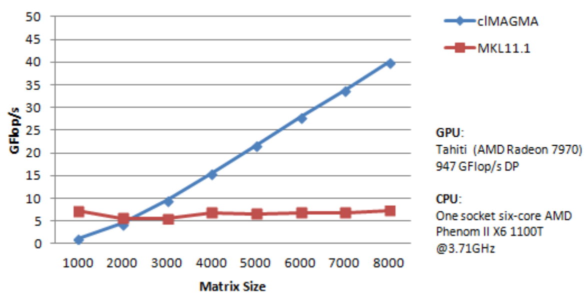

Figure 12: Performance of clMAGMA's Hessenberg factorization in double precision vs. MKL 11.1

The performance of the double precision matrix inversion in clMAGMA (`magma_zgetri_gpu`) is shown in Figure 13. It achieves up to 48 Gflop/s, getting about 1.2× speedup versus the CPU host.

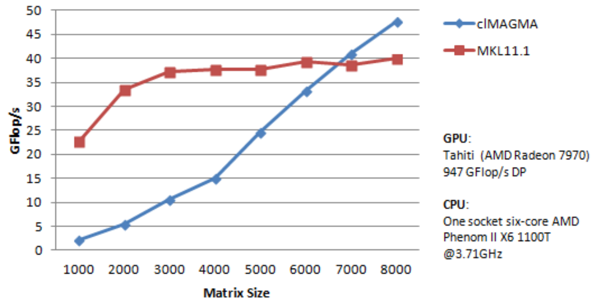

Figure 13: Performance of clMAGMA's Matrix Inversion in double precision vs. MKL 11.1

6. CONCLUSIONS AND FUTURE WORK

We have presented high performance linear algebra routines for a wide range linear transformation. The routines were implemented efficiently on AMD's Tahiti GPUs with the use of the OpenCL standard and optimized BLAS routines from the hardware vendor. Our optimization techniques show a wide applicability and yield many-fold performance improvement over highly tuned codes that constitute state-of-the-art libraries for the current generation of multicore CPUs. With the success we achieved in porting our high performance kernels to OpenCL implementation on GPUs, we are encouraged to look into extending our porting efforts to the emerging platforms such as Intel Xeon Phi and ARM's Aarch64 as well as the supported editions of multicore x86 hardware that are targeted by CPU-oriented implementations of OpenCL.

Acknowledgments

The authors would like to thank the National Science Foundation (award #0910735), the Department of Energy, and AMD for supporting this research effort.

7. REFERENCES

[1] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *J. Phys.: Conf. Ser.*, 180(1), 2009.

[2] AMD. Accelerated Parallel Processing Math Libraries (APPML). Available at http://developer.amd.com/tools/heterogeneous-computing/.

[3] AMD. AMD Core Math Library (ACML). Available at http://developer.amd.com/tools/.

[4] AMD. AMD Accelerated Parallel Processing OpenCL Programming Guide (v2.4), 2012. Available at: http://developer.amd.com.

[5] E. Anderson, Z. Bai, C. Bischof, S. L. Blackford, J. W. Demmel, J. J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. C. Sorensen. *LAPACK User's Guide*. SIAM, Philadelphia, Third edition, 1999.

[6] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency Computat. Pract. Exper.*, 2010. (to appear).

[7] Barcelona Supercomputing Center. *SMP Superscalar (SMPSs) User's Manual, Version 2.0*, 2008. http://www.bsc.es/media/1002.pdf.

[8] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, Philadelphia, PA, 1997. http://www.netlib.org/scalapack/slug/.

[9] Software distribution of clMAGMA version 1.0. http://icl.cs.utk.edu/magma/software/, October 24 2012.

[10] J. Dongarra, J. Bunch, C. Moler, and G. Stewart. *LINPACK Users' Guide*. SIAM, Philadelphia, PA, 1979.

[11] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Comput.*, 38(8):391–407, Aug. 2012.

[12] A. Haidar, S. Tomov, J. Dongarra, R. Solca, and T. Schulthess. A novel hybrid CPU-GPU generalized eigensolver for electronic structure calculations based on fine grained memory aware tasks. *International Journal of High Performance Computing Applications*, September 2012. (accepted).

[13] Intel. Math Kernel Library. Available at http://software.intel.com/en-us/articles/intel-mkl/.

[14] Khronos OpenCL Working Group. The opencl specification, version: 1.0 document revision: 48, 2009.

[15] K. Matsumoto, N. Nakasato, S. Sedukhin, I. Tsuruga, and A. City. Implementing a code generator for fast matrix multiplication in OpenCL on the GPU. July 2012.

[16] N. Nakasato. A fast GEMM implementation on the Cypress GPU. *SIGMETRICS Performance Evaluation Review*, 38(4):50–55, 2011.

[17] S. Tomov and J. Dongarra. *Scientific Computing with Multicore and Accelerators*, chapter Dense Linear Algebra for Hybrid GPU-based Systems. Chapman and Hall/CRC, 2010.

[18] V. Volkov and J. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Supercomputing 08*. IEEE, 2008.

[19] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.