2014 5th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems

Performance and Portability with OpenCL for Throughput-Oriented HPC Workloads Across Accelerators, Coprocessors, and Multicore Processors

Chongxiao Cao, Mark Gates, Azzam Haidar, Piotr Luszczek, Stanimire Tomov, Ichitaro Yamazaki
University of Tennessee, Knoxville, Tennessee, U.S.A.
Jack Dongarra
University of Tennessee, Oak Ridge National Laboratory, University of Manchester

***Abstract*—Ever since accelerators and coprocessors became the mainstream hardware for throughput-oriented HPC workloads, various programming techniques have been proposed to increase productivity in terms of both the performance and ease-of-use. We evaluate these aspects of OpenCL on a number of hardware platforms for an important subset of dense linear algebra operations that are relevant to a wide range of scientific applications. Our findings indicate that OpenCL portability has improved since our previous publication and many new and surprising usage scenarios are possible that rival those available after decades of software development on the CPUs. The combined performance-portability metric, even though not promised by the OpenCL standard, reflects the need for tuning performance-critical operations during the porting process and we show how a large portion of the available efficiency is lost if the tuning is not done correctly.**

I. Introductory Remarks

In this paper, we use OpenCL to implement a set of dense linear algebra operations that are relevant to a wide range of scientific applications, and evaluate two main aspects:

- Static versus dynamic scheduling of tasks;
- Cross-device portability in relation to performance portability for AMD GPUs, NVIDIA GPUs, Intel Xeon Phi (MICs), and many-core x86 CPUs.

It is a challenge to exploit the extreme level of parallelism and heterogeneity available on a modern computer. One way to tackle this programming challenge is to redesign and divide the algorithm of interest into carefully chosen computational tasks and then to schedule these tasks for execution on the computational components of the heterogeneous system using either static or dynamic scheduling. The latest release of MAGMA (version 1.4.1) [2], an extension of the popular LAPACK for heterogeneous systems, uses static scheduling. New results presented in this paper, on the other hand, use dynamic scheduling and show similar performance, while enabling the expression of parallelism through otherwise sequential code. Ideas to consider include dynamic code development that focuses on device-oriented development, emphasis on asynchronous execution, and straight-forward expression of algorithmic variants that enables fast selection of optimally tuned implementations – all taken into account here.

Our goal is to unify our numerical software for AMD and NVIDIA GPUs and Xeon Phi by using OpenCL programming that targets cross-platform portability. We show that obtaining high-performance still depends on hardware-specific optimizations. Our primary focus here is to study the extent of these non-portable optimizations within the framework of well understood and studied body of knowledge relating to the numerical dense linear algebra. The case of comparison is made through the AMD's clMath library (formerly APPML) [1], which currently is the only available complete BLAS implementation for OpenCL.

We note the draft release of OpenCL 2 [16] and its possible effect on our current and future work. Especially, the set of new features such as shared virtual memory, dynamic parallelism, generic address space, and C11 atomics will allow further strides in both portable and performance-efficient code development.

II. OpenCL LAPACK for Heterogeneous Systems

Over the last decade GPUs emerged as a viable hardware platform offered as discrete compute cards by the major vendors and system integrators. The success of the hardware resulted in an unfortunate proliferation of software solutions for easing the programming burden that accelerators bring to bear. Even with the recent coalescing of the plethora of products around major standardization themes, the end user is still left with the choice of one of the *open* specifications: OpenCL [15], OpenMP 4 [22], and OpenACC 2 [21]. From the very specific perspective of library development, only OpenCL offers a sufficient breadth of features essential for performance, portability, and maintainability, without the burden of extraneous software dependencies. An additional consideration is the fact that many algorithms, and in particular, the ones in the area of dense linear algebra, are designed to use the BLAS standard but there is no fully compliant BLAS implementation in OpenACC. Because of the aforementioned circumstances, we use OpenCL [15] to provide portability across a variety of accelerators and coprocessors. OpenCL is an open standard for off-loading computations to accelerators, coprocessors, and many-core processors that we use along with our task scheduling mechanisms such as runtime environments, auto-tuning frameworks, and coding techniques. The algorithms described here are being incorporated into our clMAGMA [7] library, which is a redesign and enhancement of the popular LAPACK for heterogeneous systems with OpenCL as the API to interface with various hardware accelerators.

For completeness, we would also like to mention C++ AMP [8], [11] – a standard that targets Microsoft's DirectX and DirectCompute frameworks. The recent porting efforts might make AMP much more relevant outside of the Windows and Visual Studio ecosystems as a port to LLVM is ongoing by

978-1-4799-7562-4/14 $31.00 © 2014 IEEE
DOI 10.1109/ScalA.2014.8

MultiCoreWare and overseen by the HSA Foundation[1].

The rest of the paper is organized as follows. After surveying related work in Section III, we first review, in Section IV, the one-sided factorization algorithms – LU, QR, and Cholesky of dense matrices – that are the focus of this paper. Then, in Section V, we present our implementations of the algorithms for multiple accelerators/coprocessors using OpenCL. Finally, in Section VI, we present the performance of our implementation on various architectures. We provide our final remarks in Section VII.

III. Related Work

Synthetic benchmarks of GPUs have been used extensively to understand graphics accelerators when technical details of the hardware remain an industrial secret [23]. In the context of scientific applications, such benchmarking efforts lead to algorithms that provide significant performance improvements [27].

The Ocelot [14] project did a performance-oriented study of NVIDIA's PTX (Parallel Thread eXecution) architecture [20]. Another project, MCUDA [26], applied code transformations to CUDA kernels, enabling them to run efficiently on multicore CPUs. Unforunately for legacy code maintainers, the reverse operation – porting multicore code to GPUs – proved difficult [14].

Work on optimizing CUDA implementations of basic linear algebra kernels has demonstrated that the performance of a GPU is sensitive to the formulation of a kernel [30] and that an enormous amount of well-thought experimentation and benchmarking [27], [30] is needed in order to optimize the performance. Tuning OpenCL applications for a particular architecture faces the same challenges. Optimizing a fixed OpenCL code for several architectures is very difficult, perhaps impossible, and naturally, many authors claim that OpenCL does not provide performance portability. This, along with the fact that GPUs are quickly evolving in complexity, has made tuning numerical libraries for them challenging. One approach (that we explore) to systematically resolve these issues is the use of auto-tuning, a technique that in the context of OpenCL would involve collecting and generating multiple kernel versions, implementing the same algorithm optimized for different architectures, and heuristically selecting the best performing one. Auto-tuning has been used intensively on CPUs in the past to address these challenges to automatically generate near optimal numerical libraries, e.g., ATLAS [29], [9] and PHiPAC [4] used it to generate highly optimized BLAS. Work on auto-tuning CUDA kernels for NVIDIA GPUs [18], [19] has shown that the technique is a very practical solution to port existing algorithmic solutions on quickly evolving GPU architectures and to substantially speed up even highly tuned hand-written kernels. The challenge of providing performance portability is by no means limited to linear algebra.

Performance portability of OpenCL in applications was studied before [28] and the authors compared CUDA and OpenCL implementations of a Monte Carlo Chemistry application running on an NVIDIA GTX285. They also compared the same application written in ATI's now defunct Brook+ to an OpenCL version on a Firestream 9170 and Radeon 4870. They compared OpenCL to a C++ implementation running on multicore Intel processors. The paper showed OpenCL providing code portability but not necessarily providing *performance* portability. Furthermore, they showed that platform-specific languages often, but not always, outperformed OpenCL.

In our previous work [10], OpenCL was evaluated as a programming tool for developing performance-portable applications for GPGPU on BLAS kernels, in particular on the GEMM and TRSM operations. Higher level OpenCL-based routines for a single AMD GPU were presented in [6].

Compiling OpenCL kernels at runtime from source code introduces a significant amount of overhead. By caching to disk the Intermediate Representation (IR) resulting from clGetProgramInfo, and loading it at runtime, overhead can be effectively reduced [10]. AMD and NVIDIA's OpenCL implementations both allow such a maneuver, which is essential for the performance of clMAGMA since GPU kernels can be repeatedly called in different routines. An efficient way to handle the kernel compiling and caching is required. In clMAGMA, a runtime system is implemented to fulfill this task. On the Intel Xeon Phi, though, the hardware design is different than the one on traditional GPUs, and therefore optimizations for the two architectures require different techniques [13].

IV. One-Sided Factorizations

A. Right-looking LU and QR

The LU factorization of an m-by-n matrix A with partial pivoting has the form $PA = LU$, where P is an m-by-m row permutation matrix, L is an m-by-n lower-triangular matrix with unitary diagonal, and U is an n-by-n upper-triangular matrix. The LAPACK routine **xGETRF** computes this LU factorization, where **x** can either be **S**, **D**, **C**, or **Z** denoting either single, double, single-complex, or double-complex precision used for computing. LAPACK stores all the matrices in column-major order.

xGETRF first computes the partial factorization,

$$P_1 A = P_1 \begin{pmatrix} A_{1,1} & A_{1,2:n_t} \\ A_{2:m_t,1} & A_{2:m_t,2:n_t} \end{pmatrix}$$
$$= \begin{pmatrix} L_{1,1} & \\ L_{2:m_t,1} & I \end{pmatrix} \begin{pmatrix} I & \\ & \widehat{A} \end{pmatrix} \begin{pmatrix} U_{1,1} & U_{1,2:n_t} \\ & I \end{pmatrix}, \quad (1)$$

where P_1 is an m-by-m permutation matrix for the partial pivoting of the first n_b columns of A; $A_{1,1}$, $L_{1,1}$, and $U_{1,1}$ are the leading n_b-by-b_b blocks of A, L, and U, respectively; and m_t and n_t are the respective numbers of block rows and columns of A (i.e., $m_t = \frac{m}{n_b}$ and $n_t = \frac{n}{n_b}$).[2] **GETRF** computes factorization (1) in the following two phases, where the corresponding LAPACK/BLAS routines are shown in bold:

- 1) *Panel factorization.* **xGETF2** computes the LU factorization of the first m-by-n_b block column $A_{:,1}$ of A: $P_1 A_{1,:} = L_{1,:} U_{1,1}$, where $A_{:,j}$ is referred to as the j-th *panel* of the factorization.
- 2) *Trailing submatrix update.* The transformation computed by **xGETF2** is applied to the rest of the matrix, $A_{:,2:n_t}$:
  1) **xLASWP** applies the pivoting P_1 to the submatrix: $A_{:,2:n_t} := P_1 A_{:,2:n_t}$.
  2) **xTRSM** computes the off-diagonal blocks $U_{1,2:n_t}$ of U: $U_{1,2:n_t} := L_{1,1}^{-1} A_{1,2:n_t}$.
  3) **xGEMM** updates the trailing submatrix $A_{2:m_t,2:n_t}$: $\widehat{A} := A_{2:m_t,2:n_t} - L_{2:m_t,1} U_{1,2:n_t}$.

The LU factorization of A is then computed by recursively applying the above algorithm to the trailing submatrix $\widehat{A}$. This

[1] http://www.hsafoundation.com/bringing-camp-beyond-windows-via-clang-llvm/

[2] To simplify the exposition, we assume that m and n are multiples of n_b.

is referred to as a right-looking algorithm since at each step, the panel is used to update the trailing submatrix, which is on the right of the panel. The upper-triangular part of U and the strictly lower-triangular part of L are stored in the corresponding parts of A. An additional $\min(n,m)$ length integer array is required to store the pivots P compactly as a sequence of row interchanges rather than a full m-by-n permutation matrix. To properly account for the row interchanges performed during the factorization of the trailing submatrix, another call to **xLASWP** follows each trailing matrix factorization.

The QR factorization of the matrix A is of the form $A = QR$, where Q is an m-by-m orthonormal matrix, and R is an m-by-n upper-triangular matrix. The LAPACK routine **xGEQRF** implements a right-looking QR factorization algorithm, whose first step consists of the following two phases:

• 1) *Panel factorization.* The first panel $A_{:,1}$ is transformed into an upper-triangular matrix.

1) **xGEQR2** computes an m-by-m Householder matrix H_1 such that $H_1^T A_{:,1} = \begin{pmatrix} R_{1,1} \\ 0 \end{pmatrix}$, and $R_{1,1}$ is an n_b-by-n_b upper-triangular matrix.
2) **xLARFT** computes a block representation of the transformation H_1, i.e., $H_1 = I - V_1 T_1 V_1^H$, where V_1 is an m-by-n_b matrix and T_1 is an n_b-by-n_b upper-triangular matrix.

• 2) *Trailing submatrix update.* **xLARFB** applies the transformation computed by **xLARFT** to the submatrix $A_{:,2:n_t}$:

$$\begin{pmatrix} R_{1,2:n_t} \\ \widehat{A} \end{pmatrix} := (I - V_1 T_1 V_1^H) \begin{pmatrix} A_{1,2:n_t} \\ A_{2:m_t,2:n_t} \end{pmatrix}.$$

Then, the QR factorization of A is computed by recursively applying the same transformation to the submatrix $\widehat{A}$. The transformations V_j are stored in the lower-triangular part of A, while R is stored in the upper-triangular part. Additional m-by-n_b storage is required to store T_j.

B. Left-looking Cholesky

The Cholesky factorization of a Hermitian positive-definite matrix A is of the form $A = RR^H$, where R is an n-by-n lower-triangular matrix with positive real diagonals. The LAPACK routine **xPOTRF** computes the Cholesky factor R, whose j-th step computes the j-th block column $R_{j:n_t,j}$ of R in the following two phases:

• 1) *Panel update.* The j-th panel is updated using the previously-computed columns of R,

1) **xSYRK** updates the diagonal block $A_{j,j}$, $A_{j,j} := A_{j,j} - R_{j,1:(j-1)} R_{j,1:(j-1)}^H$.
2) **xGEMM** updates the off-diagonal blocks, $A_{(j+1):n_t,j} := A_{(j+1):n_t,j} - R_{(j+1):n_t,1:(j-1)} R_{j,1:(j-1)}^H$.

• 2) *Panel factorization*. The j-th panel is factorized.

1) **xPOTRF2** computes the Cholesky factor $R_{j,j}$ of $A_{j,j}$, $A_{j,j} = R_{j,j} R_{j,j}^H$.
2) **xTRSM** computes the off-diagonal blocks $R_{(j+1):n_t,j}$, $R_{(j+1):n_t,j} := R_{j,j}^{-1} A_{(j+1):n_t,j}$.

This is known as a left-looking algorithm since at each step, the panel is updated using the previous columns, which are on the left of the panel. The above algorithm references only the lower-triangular part of A, which is overwritten by R. Alternatively, given the upper-triangular part of A, **xPOTRF** can compute R^H by block rows.

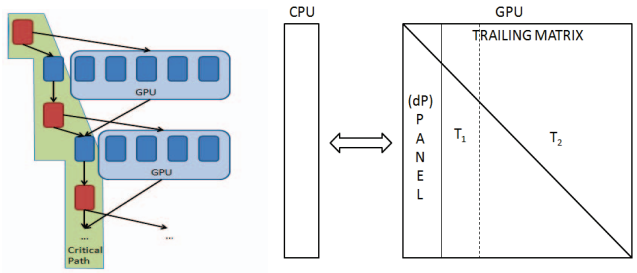


Fig. 1. Illustration of hybrid programming.

V. Implementation

A. Hybrid Programming

We extend in clMAGMA the LAPACK routines to utilize the computing power of CPUs, GPUs, and coprocessors as available in modern heterogeneous architectures. The algorithms design and implementation are based on hybrid programming and scheduling to match the different characteristics of the algorithm at the different phases of the factorization with the different performance strengths of the CPUs and GPUs available. For instance, during one-sided factorizations, the submatrix updates based on BLAS-3 exhibit high data parallelism and are ideal for running on a GPU. On the other hand, panel and diagonal factorizations are based on BLAS-1 and BLAS-2, and are often faster on a CPU. Hence, in clMAGMA we distribute the whole matrix A among the GPUs and use the GPUs to update their corresponding submatrices, while the panels and diagonal blocks are copied to and factorized on the multicore CPUs using a threaded version of LAPACK (see Figure 1).

B. Static Scheduling

To utilize the multiple GPUs for submatrix update, the matrix A is distributed in a 1D block column format for the LU and QR factorization, while it is distributed in a 1D block row or column format for the Cholesky factorization in the lower and upper triangular forms, respectively. Figure 2 shows the pseudocodes of our multi-GPU one-sided factorization algorithms. Note that the scheduling of the tasks in these algorithms is done statically. This is the default scheduling in the latest clMAGMA 1.1 release.

To effectively use the CPU for the panel factorization, we use a technique commonly-known as *lookahead*: as soon as the next panel is updated on the GPU, it is copied to the CPU so that the panel can be factorized on the CPU, while the GPUs continue updating the rest of the submatrix. Hence, for a large enough matrix, the BLAS-1 and BLAS-2 based panel factorization on the CPU can be hidden behind the BLAS-3 based submatrix update on the GPU. As a result, the lookahead brings the potential of having factorizations running at the speed of the BLAS-3 GPU kernels. We also use a separate queue for asynchronous data transfer between the CPU and GPU such that it can be overlapped with the computation either on the CPU or GPU. For instance, by using a separate queue, the transfer of the panel between the CPU and the GPU can be overlapped with the submatrix update. In addition, the transfer of the factor back to the CPU can be also overlapped with the computation (U-factor, R-factor, and off-diagonal blocks in the LU, QR, and Cholesky factorization, respectively). These techniques are easily implemented with static scheduling, where the main CPU thread goes through the

Algorithm 3.1
1. distribute A from CPU to GPUs.
2. `for` $j = 1,2,\ldots,n_t$ `do`
   a. use **xGETRF** to factorize panel on CPU.
   b. broadcast the panel from CPU to GPUs.
   c. apply pivoting to L on GPUs.
   d. use **magma_xTRSM** to compute $U_{j,(j+1):n_t}$ on GPUs.
   e. use **magma_xGEMM** to update trailing submatrix on GPUs.
   f. copy next panel from GPU to CPU.
   `end for`
3. gather U-factor from GPUs to CPU.

(a) Right-looking LU algorithm.

Algorithm 3.2
1. distribute A from CPU to GPUs.
2. `for` $j = 1,2,\ldots,n_t$ `do`
   a. use **xGEQRF** to factorize the panel on CPU.
   b. use **xLARFT** to compute T_j and V_j.
   c. copy T_j and V_j from CPU to GPU.
   d. use **magma_xLARFB** to apply the transformation on GPU.
   e. copy next panel $A_{j:n_t,j}$ from GPU to CPU.
   `end for`
3. gather R-factor from GPUs to CPU.

(b) Right-looking QR algorithm.

Algorithm 3.3
1. distribute lower-triangular of A from CPU to GPUs.
2. `for` $j = 1,2,\ldots,n_t$ `do`
   a. use **magma_xSYRK** to update $A_{j,j}$ on GPU.
   b. copy $A_{j,j}$ from GPU to CPU.
   c. use **magma_xGEMM** to update $A_{(j+1):n_t,j}$ on GPUs.
   d. use **xPOTRF** to compute $R_{j,j}$ on CPU.
   e. broadcast $R_{j,j}$ from CPU to GPUs.
   f. use **magma_xTRSM** to compute $R_{j,(j+1):n_t}$ on GPU.
   `end for`
3. gather off-diagonal blocks of R from GPUs to CPU.

(c) Left-looking Cholesky algorithm (lower triangular).

Algorithm 3.3
1. distribute lower-triangular of A from CPU to GPUs.
2. `for` $j = 1,2,\ldots,n_t$ `do`
   a. use **xPOTRF** to compute $R_{j,j}$ on CPU.
   b. broadcast $R_{j,j}$ from CPU to GPUs.
   c. use **magma_xTRSM** to compute $R_{j,(j+1):n_t}$ on GPU.
   d. use **magma_xSYRK** to update trailing submatrix on GPU.
   e. copy $A_{j,j}$ from GPU to CPU.
   `end for`
3. gather off-diagonal blocks of R from GPUs to CPU.

(d) Right-looking Cholesky algorithm (lower triangular).

Fig. 2. MAGMA one-sided factorization algorithms.

algorithm, first queuing large task/work requests to the GPUs through OpenCL queues, while second, performing tasks on the critical part.

C. Dynamic Scheduling

The use of multiple heterogeneous devices complicates the development using static scheduling. Instead, the use of a light-weight runtime system may be preferred as it can keep scheduling overhead low, while enabling the expression of parallelism through sequential-like code. The runtime system relieves the developer from keeping track of the computational activities that, in the case of heterogeneous systems, are further exacerbated by the separation between the address spaces of the main memory of the CPUs, GPU accelerators, and coprocessors. Our runtime model is build on the QUARK [31] superscalar execution environment that has been originally used with great success for linear algebra software on multicore platforms [17]. The conceptual work though could be replicated within other models such as StarPU [3], OmpSS [24], Cilk [5], and Jade [25], to mention just a few.

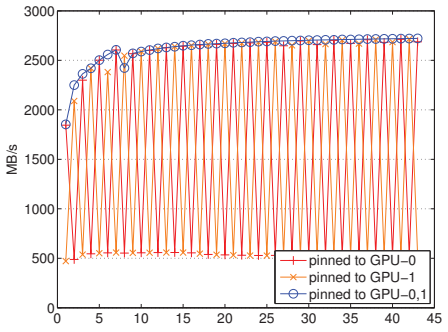
```
cl_mem clCreateBuffer( cl_context context,
  cl_mem_flags flags, size_t size, void *host_ptr,
  cl_int *errcode_ret)
creates a buffer object with the following arguments:
  context (input) : valid OpenCL context,
  flags   (input) : bit-field to specify allocation and usage
                    information,
  size    (input) : size in bytes of the buffer memory object to be
                    allocated,
  host_ptr(input) : pointer to a buffer that has been allocated
                    by the application,
  errcode (output): error code.

void * clEnqueueMapBuffer( cl_command_queue queue,
  cl_mem buffer, cl_bool blocking,
  cl_map_flags maping, size_t offset, size_t cb,
  cl_uint num_events, const cl_event *event_list,
  cl_event *event, cl_int *errcode_ret)
maps a region of buffer into host address space with the following
arguments:
  queue     (input) : valid command-queue,
  buffer    (input) : valid buffer object,
  blocking  (input) : flag to indicate if this operation is blocking
                      or non-blocking,
  maping    (input) : bit-field to indicate if buffer is for reading
                      or/and writing,
  offset,cb (input) : offset in bytes and size of region in the buffer
                      object that is being mapped,
  num_events(input) : number of events in enent_list,
  event_list(input) : events that must be completed before executing
                      this command,
  event     (output): event object that identifies this command, and
  errcode   (output): error code.
```

Fig. 3. OpenCL interfaces to map CPU memory.

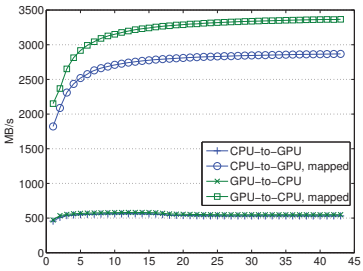
Compared to static scheduling, the dynamic approach allows us to unroll and pipeline for execution a larger part of an algorithm, which creates more opportunities for minimizing CPU-GPU synchronizations and CPU-GPU communication latencies. Static scheduling on the other hand does not have scheduling overheads but is more susceptible to cause CPU or GPU idle time. For example, GPU idle time will occur if insufficient work is queued to the GPU before the CPU is dedicated to execution of the next CPU task.

D. Mapped memory for CPU-GPU data transfer.

Mapping the CPU memory to the GPU allows us to get a higher data transfer throughput between the CPU and GPU. In OpenCL, `clCreateBuffer` and `clEnqueueMapBuffer` provide the required interfaces to map the CPU memory (see Figure 3 for their interfaces). In order to map memory for multiple GPUs, we first conducted experiments to verify if `clEnqueueMapBuffer` must be called either per `gContext` that includes multiple GPUs, or per device specified by `queue`, or for each `queue` that uses the mapped memory. For example, Figure 4(a) compares the data transfer rates of cyclically distributing the block columns of $11,008$-by-$11,008$ double-precision upper-triangular matrix among two GPUs with the block size $n_b = 256$, when the CPU memory that stores the matrix is mapped to GPU-0, to GPU-1, or to both GPU-0 and GPU-1. The figure clearly indicates that the memory must be mapped to both GPUs by invoking clEnqueueMapBuffer with each queue associated with the GPU. For instance, the following piece of the code maps the memory to multiple GPUs, where `queue[2*d]` is associated



(a) Mapping memory to multiple GPUs.



(b) Data transfer rate with/without mapping.

Fig. 4. Effects of mapping on the data transfer rate ($n = 11,008, n_b = 256$).

with GPU-d:

```
cl_mem buffer = clCreateBuffer(gContext,
  CL_MEM_READ_WRITE | CL_MEM_ALLOC_HOST_PTR,
  sizeof(magmaDoubleComplex) * sizePanel *
  (1+num_gpus), NULL, NULL);
for (d=0; d < num_pinned; d++)
  work = (magmaDoubleComplex*)
    clEnqueueMapBuffer( queues[2*d], buffer,
      CL_TRUE, CL_MAP_READ | CL_MAP_WRITE, 0,
      sizeof(magmaDoubleComplex) * sizePanel *
      (1+num_gpus), 0, NULL, NULL, NULL);
```

We have also verified that the memory is mapped to the device that is associated with the queue, and we can use the mapped memory with a queue that is associated with the same device but is different from the one used to map the memory. Finally, to use the mapped memory for data transfer, the pointer to the beginning of the mapped memory should be used.

Figure 4(b) compares the data transfer rates from the CPU to the GPU with those from the GPU to the CPU, and also shows the transfer rates of cyclically distributing the block rows of the lower triangular matrix.

E. Multiple buffer to utilize GPU memory.

The current OpenCL driver limits the amount of the device memory that can be allocated per `clCreateBuffer` call. To overcome this limitation and utilize the whole device memory for factorizing a large matrix, we could run our multi-GPU factorization routine on one GPU. However, this requires redundant communication between the CPU and the GPU, which can be expensive. To avoid this overhead, we developed a multi-buffer implementation, where the local submatrix on each GPU is stored in multiple buffers. As shown in Figure 5, this not only allows us to factorize a large matrix, but it also

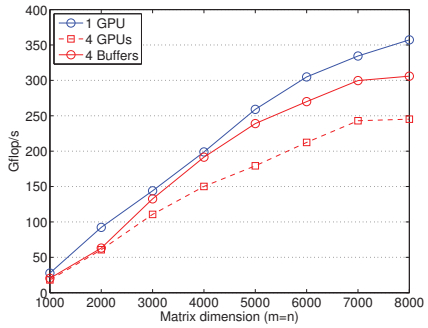
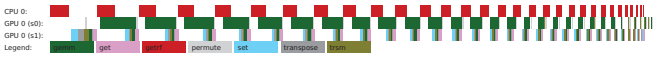
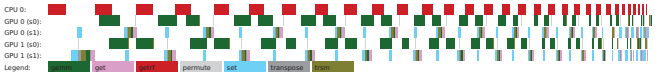


Fig. 5. Performance of LU factorization with multiple buffers.

(a) One buffer.



(b) Two buffers.



(c) Two GPUs.

Fig. 6. Execution trace of `dgetrf_msub` on one AMD GPU ($n = 8,000$).

avoids the additional communication required when running the multi-GPU algorithm on a single GPU. The trace from Fig. 6 could offer additional inside into this issue.

VI. Experimental Results

A. Hardware Description and Setup

Our experiments were performed on a number of shared-memory systems. They are representative of a vast class of servers and workstations commonly used for computationally intensive workloads. We conducted our experiments on three different systems:

- System A was six-core AMD Phenom™ II X6 1055T Processor, running at 2.8 GHz with 8 GiB of main memory. It featured two AMD HD7970 cards with 3 GB per card running at 1000 MHz
- System B was composed of an four-socket AMD Opteron™ 6380 with 16 cores each and sharing two FPU units per core for the total of 64 cores running at 1.4 GHz with 128 GiB of main memory. It was equipped with a single AMD HD7970 card with 3 GiB of main GPU memory and running at 1 GHz.
- System C was an Intel multicore system with dual-socket, 8-core Intel Xeon E5-2670 (Sandy Bridge) processor running at 2.6 GHz. Each socket had 24 MiB of shared Level 3 cache, and each core had a private 256 KiB Level 2 and 64 KB Level 1 cache. The system was equipped with 52 GB of memory and the theoretical peak in double precision was 20.8 Gflop/s per core. System C also featured three NVIDIA K40c cards with 11.5 GB per card running at 875 MHz, connected to the host via two PCIe I/O hubs measured at 10 GB/s of achievable bandwidth. In terms of software, on the CPU side, we used the MKL (Math Kernel Library) [12] and the Intel compiler 13.1.1 20130313, which comes with the Composer XE 2013.4.183 suite. On the AMD HD7970 GPU side, we used OpenCL implementation for version 1.2 of

the standard and the driver version 1348.5. On NVIDIA K40c GPU, OpenCL version 1.1 used with the driver version 331.22. And finally, as BLAS implementation, we used the open-source clBLAS library which provides code for the OpenCL BLAS portion of clMath.

B. Performance results

Getting good performance across multiple accelerators remains a challenging problem that we address with the algorithmic and programming techniques described in this paper. The efficient strategies used to schedule and exploit parallelism across multiple devices will be highlighted in this subsection through the extensive set of experiments that we performed.

Figures 7 show the performance scalability of the Cholesky factorization in double precision on either a single GPU or two GPUs of System A. The curves show performance in terms of Gflop/s. We note that this also directly reflects the elapsed time: performance that is two times higher corresponds to an elapsed time that is two times shorter. Our heterogeneous implementation shows very good scalability. For example for a 24,000 matrix, the Cholesky factorization achieves around 800 Gflop/s when using the 2 AMD HD7970 GPUs. Note that both the dynamic and the static version of our implementation achieves the same trend of performance curves. We observed similar performance trends for both the LU factorization and the QR decomposition. Figures 8 and 9 illustrate the scalability performance obtained on system A. The three amigos achieves very good scalability with the respect to the performance of the system. This was expected from the analysis of the tracing figures depicted above. The trace is compact and shows that the GPU is always busy, which means that the algorithm reaches close to the peak of the Level 3 BLAS routine performance on the GPU. The difference in performance between the three amigos can be explained by a detailed examination of the algorithm. The trailing matrix update of the LU factorization consists mainly of gemm's kernel which provides the highest performance on GPU. We can thus expect that a good implementation of LU trend to be asymptotically close to thegemm's peak. However, for the QR algorithm the trailing matrix update consists of a call to larfb. The larfb is composed of one inner-product and one outer-product gemm and a small trmm of size *nb*, where *nb* is the blocking factor (or the width of the panel). We note that in the current implementation of BLAS for AMD GPUs, the inner-product gemm as well as trmm do not perform at the same speed as the classical square or outer-product gemm. For this reason, the performance of QR trails by a bit the LU code. Due to a related reason, the Cholesky trailing matrix update does not perform well either, as it consists of syrk not gemm.

C. Programming Model Across Multiple Devices

In this section, we discuss the programming model that raises the level of abstraction above the hardware and its accompanying software stack to offer a uniform approach for algorithmic development. GPU accelerators and coprocessors have different capabilities, which makes it challenging to develop an algorithm that can achieve high performance and reach good scalability. The key features of OpenCL allow us to develop stable implementation across many different architectures. We ported our implementation with little effort to run on NVIDIA devices. We used the open source AMD clBLAS library and compiled it on System C. We first evaluated the

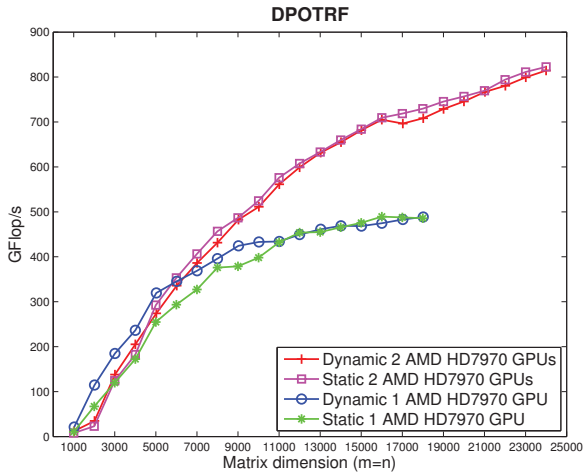


Fig. 7. Performance of double precision Cholesky factorization on up to two AMD HD7970 GPUs.

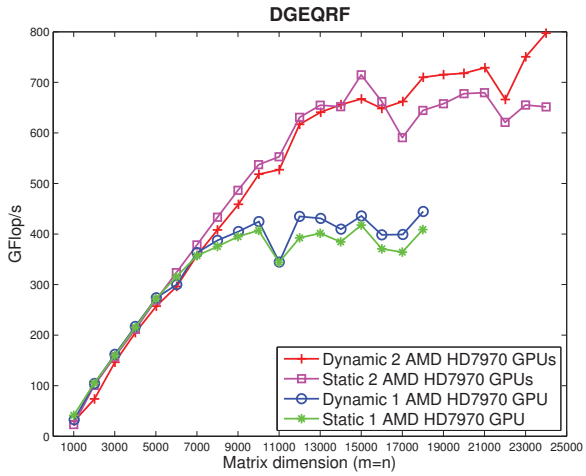


Fig. 8. Performance of double precision QR factorization on up to two AMD HD7970 GPUs.

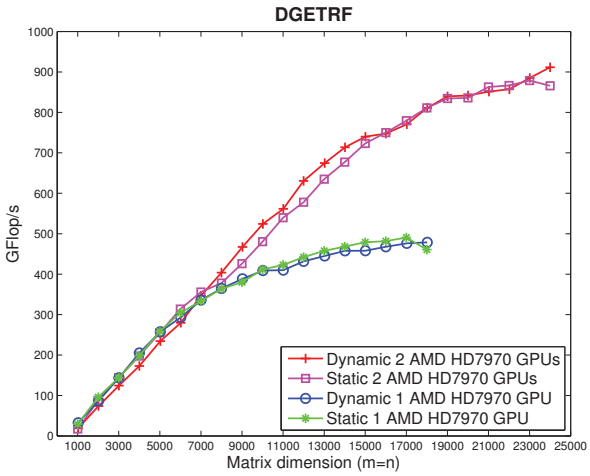


Fig. 9. Performance of double precision LU factorization on up to two AMD HD7970 GPUs.

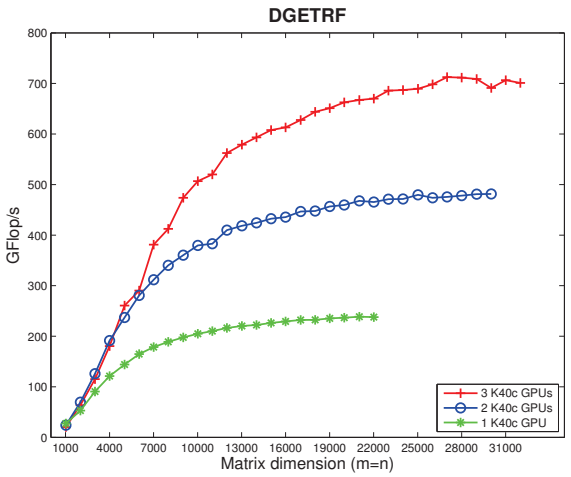


Fig. 10. Performance of double precision LU factorization on up to three Nvidia Tesla K40c GPUs.

performance of the gemm kernel which is supposed to reach the highest fraction of the peak performance of the device. Our experiments showed that the double precision dgemm kernel from AMD clBLAS library achieves about 100 Gflop/s on NVIDIA M2090 and about 260 Gflop/s on NVIDIA K40c (System C) while cuBLAS dgemm reaches 1250 Gflop/s on the faster card. From a programming model point of view, it is nearly impossible to hide the distinction between the two devices (AMD GPU and NVIDIA K40c). Also, the BLAS library was designed for older of GPU architecture and by a different vendor. In particular, clBLAS does not map well to the architectural features of NVIDIA hardware such as the shared memory, register file, and cache structure. But the undeniable advantage of such a model is portability, even though, an effort was still required to make the kernel scalable. We have performed a set of LU factorization experiments using our System C on either one, two or three NVIDIA GPUs. Figure 10 shows the performance obtained by this implementation. It is able to reach close to the performance of AMD clBLAS' dgemm on this architecture and also it is able to scale on up to three available GPUs. Our LU factorization reaches around 230 Gflop/s on a single NVIDIA K40c and about 700 Gflop/s on three of them. We also tried to install the AMD clBLAS library and our code on the Intel Xeon Phi coprocessor. However, the portability of AMD's clBLAS was not paired with good performance. The Intel card reached less than 30 Gflop/s – an unacceptable level. This result, however, is to be expected due to much more drastic differences between the two architectures than was the case for the AMD and NVIDIA comparison. To get better performance on Phi, significant architecture-specific optimizations must be applied [13].

D. Heterogeneous-Device Computation

In this section, we propose to optimize our implementation to benefit from all the available resources on the platform. When using a platform composed of many kinds of GPUs or a GPU and a number of multicore processors, our implementation can run on all of them simultaneously. We force the data layout distribution to be hardware-guided, so that the data will either be distributed in a manner that allows each device to receive an appropriate volume of data to match its capabilities.

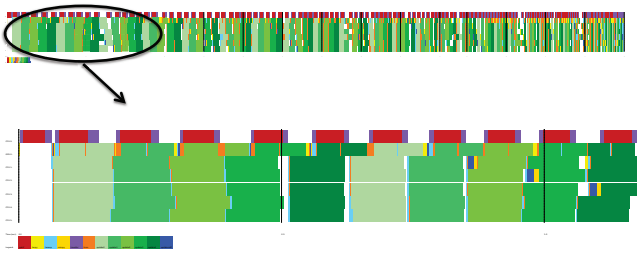


Fig. 11. LU factorization trace on multicore CPU and one accelerator AMD HD7970 GPU, using the Heterogeneous-Device Computation to achieve higher hardware usage.

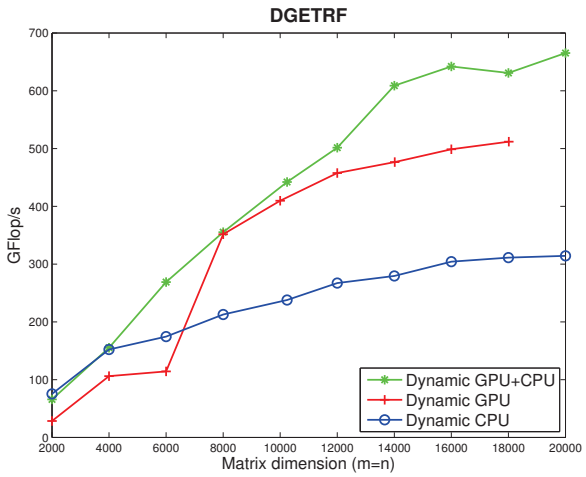
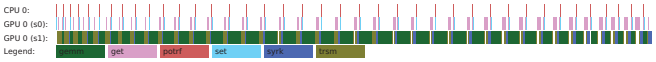


Fig. 12. Performance of double precision LU factorization on one AMD HD7970 GPU and one AMD Opteron(tm) 6380 CPU.

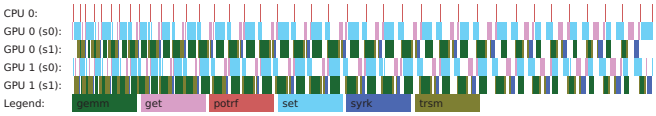
Figure 11 shows the trace of the LU factorization for a matrix of size $20K$ on System A while using the Heterogeneous Device Computation Strategy (HDCS). It is clear that the execution trace is more compact – all of the heterogeneous hardware is fully occupied with useful work and one can expect an increase in the total performance. This is shown to be the case in Figure 12 and the performance curves of the LU factorization for either CPU-only or GPU-only or using our HDCS method. The curves in green show the performance obtained on one AMD HD7970 GPU and 32 CPU cores of System A. The blue line corresponds to our implementation using only the 32 CPU cores, while the red line illustrates the performance using the AMD GPU without taking advantage of the CPUs to factorize the panel of each factorization step. We observe that we can reach an improvement of about 20% when using the HDCS technique.

Figure 13 shows the execution trace of the lower-triangular Cholesky factorization. As observed on other traces from this hardware, we see gaps between the useful computation or between the data transfers which indicate idle periods that are a sign of the sub-optimal performance mentioned earlier.

Figure 14 shows the performance of the Level 3 BLAS kernels used for the Cholesky factorization. It is worth noting that this performance profile is quite typical of CPU-only implementations on either shared or distributed memory. The slow decay in performance of dgemm is characteristic for the decreasing size of the input parameters for the routine. And the somewhat periodic nature of the performance is indicative



(a) One GPU.



(b) Two GPUs.

Fig. 13. Execution trace of `dpotrf_msub` (lower-triangular, $n = 10,000$).

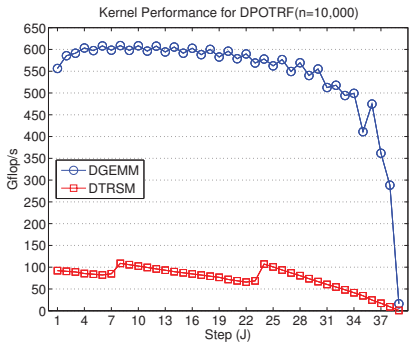


Fig. 14. Kernel performance for `dpotrf_msub` ($n = 10,000$).

of the internal blocking factors of the BLAS implementation. Finally, the drastic performance difference between dgemm and dtrsm is the testimony of the latter routine's more demanding loop dependence pattern that poses a challenge on latency-sensitive accelerators.

VII. Conclusions and Future Work

We have presented algorithms, their implementation, and performance for common dense linear algebra factorizations on top of the OpenCL implementations from major hardware accelerator vendors. At this point in time we can safely conclude that OpenCL does meet the cross-platform portability requirements, albeit with a need for some architecture-specific tuning and possibly performance penalty. With these limitations in mind, it could be considered a viable alternative for library development challenging industry or de-facto standards.

Our goal is to continue to monitor the quality of OpenCL implmenations across the hardware spectrum and broaden the algorithm choice of our implementation efforts of to the so-called "two-sided" factorizations that aim at eigenvalue-preserving transformations. The challenge for this other important class of linear algebra factorizations is the available memory bandwidth and this may pose additional challenges with respect to the OpenCL software stack.

Acknowledgments. This research was supported in part by the NSF under grants OCI-1032815, ACI-1339822; DOE – under grants DE-SC0004983, DE-SC0010042; and AMD.

References

[1] clmath libraries: clblas 2.0. , Aug. 13 2013.

[2] MAGMA version 1.4.1. , January 8 2014.

[3] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[4] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using phipac: a portable, high-performance, ansi c coding methodology. In *ICS '97: Proc. of the 11th international conference on Supercomp.*, pp. 340–347, NY, NY, USA, 1997. ACM.

[5] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall and Y. Zhou. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not.*, 30:207–216, Aug. 1995.

[6] C. Cao, J. Dongarra, P. Du, M. Gates, P. Luszczek, and S. Tomov. clmagma: High performance dense linear algebra with opencl. (LAWN (LAPACK Working Note) 275, UT-CS-13-706), 2013. Int. Workshop on OpenCL, Atlanta, GA, May 13-14.

[7] Software distribution of clMAGMA version 1.1. , January 6 2014.

[8] M. Corp.. C++ AMP : Language and programming model, 2012. Version 1.0, Aug..

[9] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R. C. Whaley, and K. Yelick. Self-adapting linear algebra algorithms and software. *Proc. IEEE*, 93(2):293–312, February 2005.

[10] 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. *Par. Comput.*, 38(8):391–407, Aug. 2012.

[11] K. Gregory and A. Miller. *C++ AMP: Accelerated Massive Par.ism with Microsoft Visual C++*. Microsoft Press, 1st edition, 2012. ISBN-13: 978-0735664739 ISBN-10: 0735664730.

[12] Intel. Math Kernel Library. .

[13] Intel Corp.. Intel SDK for OpenCL applications XE 2013 R2 optimization guide. , 2013. Document Number: 326542-003US.

[14] A. Kerr, G. Diamos, and S. Yalamanchili. A characterization and analysis of ptx kernels. In *Proc. of 2009 IEEE Int. Symposium on Workload Characterization (IISWC)*, pp. 3–12, 2009.

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

[16] Khronos OpenCL Working Group. The opencl specification, version: 2.0 document revision: 22. , March 18 2014.

[17] J. Kurzak, P. Luszczek, A. YarKhan, M. Faverge, J. Langou, H. Bouwmeester, and J. Dongarra. Multithreading in the PLASMA Library. In *Handbook of Multi and Many-Core Processing: Architecture, Algorithms, Programming, and Applications*, Comp. and Inf. Sci. Series. Chapman and Hall/CRC, April 26 2013.

[18] Y. Li, J. Dongarra, and S. Tomov. A note on auto-tuning GEMM for GPUs. In *Computational Sci. - ICCS 2009*, volume 5544 of *Lecture Notes in Comp. Sci.*, pp. 884–892. Springer Berlin / Heidelberg, 2009.

[19] R. Nath, S. Tomov, and J. J. Dongarra. Accelerating GPU kernels for dense linear algebra. In *Proc. of VECPAR'10*, Berkeley, CA, June 22-25 2010.

[20] NVIDIA. *NVIDIA Compute PTX: Par. Thread Execution*. NVIDIA Corp., Santa Clara, California, 1st edition, Oct. 2008.

[21] OpenACC Non-Profit Corp.. The OpenACC application programming interface version 2.0. , June 2013.

[22] OpenMP Architecture Review Board. OpenMP application program interface version 4.0. , July 2013.

[23] M. Papadopoulou, M. Sadooghi-Alvandi, and H. Wong. Micro-benchmarking the GT200 GPU. Tech. Rep., Comp. Group, ECE, Univ. of Toronto, 2009.

[24] J. M. Pérez, R. M. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Proc. of the 2008 IEEE Int. Conf. on Cluster Comp., 29 Sept. - 1 October 2008, Tsukuba, Japan*, pp. 142–151. IEEE, 2008.

[25] M. C. Rinard, D. J. Scales, and M. S. Lam. Jade: a high-level, machine-independent language for parallel programming. *Comp.*, 26(6):28–38, 1993. DOI: 10.1109/2.214440.

[26] J. Stratton, S. Stone, and W. mei Hwu. MCUDA: An efficient implementation of CUDA kernels on multi-cores. Tech. Rep. IMPACT-08-01, Univ. of Illinois at Urbana-Champaign, Mar. 2008. .

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

[28] R. Weber, A. Gothandaraman, R. J. Hinde, and G. D. Peterson. Comparing hardware accelerators in scientific applications: A case study. *IEEE Transactions on Par. and Dist. Systems*, 99(RapidPosts), 2010.

[29] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Par. Comp.*, 27(1–2):3–35, 2001.

[30] M. Wolfe. Special-purpose hardware and algorithms for accelerating dense linear algebra. *HPC Wire*, 10 2008. .

[31] A. YarKhan. *Dynamic Task Execution on Shared and Dist. Memory Architectures*. PhD thesis, Univ. of Tenn., Dec. 2012.