Journal of Computational Science 26 (2018) 237–245

Contents lists available at ScienceDirect

Journal of Computational Science

journal homepage: www.elsevier.com/locate/jocs

Accelerating the SVD bi-diagonalization of a batch of small matrices using GPUs☆

Tingxing Dong[a,*], Azzam Haidar[b], Stanimire Tomov[b], Jack Dongarra[b,c,d]

[a] Radeon Technologies Group, AMD, United States
[b] University of Tennessee, Knoxville, United States
[c] Oak Ridge National Laboratory, Oak Ridge, United States
[d] University of Manchester, Manchester, United Kingdom

ARTICLE INFO

Article history:
Received 16 October 2017
Accepted 28 January 2018
Available online 20 February 2018

Keywords:
Hardware accelerators
Batched
Two-sided factorization algorithms
Numerical linear algebra
Eigenvalue and singular value problems

ABSTRACT

The acceleration of many small-sized linear algebra problems has become extremely challenging for current many-core architectures, and in particular GPUs. Standard interfaces have been proposed for some of these problems, called batched problems, so that they get targeted for optimization and used in a standard way in applications, calling them directly from highly optimized, standard numerical libraries, like (batched) BLAS and LAPACK. While most of the developments have been for one-sided factorizations and solvers, many important applications – from big data analytics to information retrieval, low-rank approximations for solvers and preconditioners – require two-sided factorizations, and most notably the SVD factorization. To address these needs and the parallelization challenges related to them, we developed a number of new batched computing techniques and designed batched Basic Linear Algebra Subroutines (BLAS) routines, and in particular the Level-2 BLAS GEMV and the Level-3 BLAS GEMM routines, to solve them. We propose a *device functions*-based methodology and *big-tile* setting techniques in our batched BLAS design. The different optimization techniques result in many software versions that must be tuned, for which we adopt an auto-tuning strategy to automatically derive the optimized instances of the routines. We illustrate our batched BLAS approach to optimize batched SVD bi-diagonalization progressively on GPUs. The progression is illustrated on an NVIDIA K40c GPU, and also, ported and presented on AMD Fiji Nano GPU, using AMD's Heterogeneous–Compute Interface for Portability (HIP) C++ runtime API. We demonstrate achieving 80% of the theoretically achievable peak performance for the overall algorithm, and significant acceleration of the Level-2 BLAS GEMV and Level-3 BLAS GEMM needed compared to vendor-optimized libraries on GPUs and multicore CPUs. The optimization techniques in this paper are applicable to the other two-sided factorizations as well.

1. Introduction

The emergence of multicore and heterogeneous architectures requires many linear algebra algorithms to be redesigned to take advantage of accelerators, such as GPUs. A particularly challenging class of problems, arising in numerous applications, involves the use of linear algebra operations on many small-sized matrices. Their number can be thousands, even millions. For example, billions of 8×8 and 32×32 eigenvalue problems need to be solved in magnetic resonance imaging. Also, thousands of matrix–matrix (GEMM) and matrix–vector products (GEMV) are computed in hydrodynamic simulations with finite element method [1]. Here the size of matrices increases with the order of the numerical methods, and can range from ten to a few hundred. GEMM is at the heart of deep neural network (DNN) computations, where rather than treating convolution as one large GEMM problem, it is much more efficient to view it as many small GEMMs [2]. Thus, Batched BLAS, and Batched GEMM in particular, are central part of performing deep learning, and therefore can be used to accelerate frameworks like PaddlePaddle [3], Theano [4], TensorFlow [5], and Torch [6]. In an astrophysics ODE solver [7], multiple zones are simulated, and each zone corresponds to a small linear system solve based on an LU factorization [7]. If the matrix is symmetric and definite, the problem is reduced to a batched Cholesky factorization [8,9].

☆ This is an extended version of our conference paper [18] that was invited to the *JoCS* special issue (https://doi.org/10.1016/j.procs.2017.05.237).

* Corresponding author.

E-mail addresses: tingxing.dong@amd.com (T. Dong), haidar@icl.utk.edu (A. Haidar), tomov@icl.utk.edu (S. Tomov), dongarra@icl.utk.edu (J. Dongarra).

https://doi.org/10.1016/j.jocs.2018.01.007
1877-7503/© 2018 Elsevier B.V. All rights reserved.

In [10], there are demands to compute one million 25×8 and n*20 SVD problems, where n ranges from 32 to 4096.

The acceleration of the aforementioned many small-sized linear algebra problems has become extremely challenging for current many-core architectures, and in particular GPUs. To address the wide range of application needs and the parallelization challenges related to them, we developed a number of new batched computing techniques and designed batched Basic Linear Algebra Subroutines (BLAS) routines, and in particular the Level-2 BLAS GEMV and the Level-3 BLAS GEMM routines, to solve them. To describe these developments, we start with other related work and summary of our contributions (Section 2), followed by algorithmic background (Section 3) and the Householder Bi-diagonalization algorithm (Section 4), performance analysis and a roofline model that guides our design and optimizations (Section 5), the main batched BLAS design, optimization techniques and implementation for GPUs (Section 6), our auto-tuning strategy (Section 7), performance on NVIDIA GPUs (Section 8) and AMD GPUs (Section 9). Finally, conclusions and future work directions are given in Section 10.

2. Related work and contributions

The acceleration of many small-sized linear algebra problems has become extremely challenging for current many-core architectures, and in particular GPUs. Standard interfaces have been proposed for some of these problems, called batched problems, so that they get targeted for optimization and used in a standard way in application directly from highly-optimized, standard numerical libraries, like (batched) BLAS and LAPACK [11]. Indeed, vendors like NVIDIA and Intel started to provide certain batched functionalities in their cuBLAS and MKL libraries, respectively. MAGMA [12,13], an open source library, provides the most extended set of batched BLAS and LAPACK functionalities to date. In particular, efficient batched one-sided factorizations (LU, QR, and Cholesky) were developed in [14–17], and are now released through MAGMA. These factorizations are compute-bound and rich in Level-3 BLAS operations. Therefore, the main effort in developing them lied in algorithmically enhancing the percentage of Level-3 BLAS operations, using techniques such as recursive blocking, parallel swapping, and other batched BLAS techniques and optimizations.

While most of the developments have been for one-sided factorizations and solvers, many important applications – from big data analytics to information retrieval, low-rank approximations for solvers and preconditioners – require two-sided factorizations, and most notably the SVD factorization. To develop them, in contrast to the compute-bound one-sided factorizations, one must consider the acceleration of the memory-bound two-sided Householder bi-diagonalizations (GEBRD). These routines are the most time-consuming part that is needed for the singular value decompositions (SVD) in many applications. Instead of BLAS-3 GEMM, the Householder bi-diagonalization problem is rich in memory-bound Level-2 BLAS GEMV operations. Thus, the goal is to develop efficient batched Level-2 BLAS that minimizes memory transactions and maximizes bandwidth. To accomplish this, we propose a *device functions*-based methodology and *big-tile* setting techniques in our batched BLAS designs, in order to facilitate data reuse. The different optimization techniques, as well as the various instances of GEMV to accelerate, result in many software versions that must be tuned, for which we adopt an auto-tuning strategy to automatically derive the optimized instances of the routines.

Throughout this paper, our batched routines are named as MAGMA batched routines, and released through the MAGMA library. Our main contributions are: (1) design batched BLAS device functions and kernels, as well as efficient implementations and optimization techniques; (2) design two-sided bi-diagonalization for batched execution based on the batched BLAS approach; and (3) port and tune the developments to a number of high-end GPUs, including both NVIDIA and AMD GPUs. Our batched BLAS can run on NVIDIA GPU with a CUDA code version that we extend from [18], but also on AMD GPUs through a HIP code version that we developed. HIP programming on AMD GPU is rather new [19]. To our best knowledge, no similar work has been presented before.

3. Background

The SVD problem is to find orthogonal matrices U and V, and a diagonal matrix Σ with nonnegative elements, such that $A = U\Sigma V^T$, where A is an $m \times n$ matrix. The diagonal elements of Σ are singular values of A, the columns of U are called left singular vectors of A, and the columns of V are called right singular vectors of A. Such problem is solved by a three-phase process:

1. *Reduction phase*: orthogonal matrices Q and P are applied on both the left and the right side of A to reduce it to a bi-diagonal matrix – hence these are called "two-sided factorizations."
2. *Solution phase*: a singular value solver further computes the singular values Σ and the left and right vectors $\tilde{U}$ and $\tilde{V}^T$ of the bi-diagonal matrix;
3. *Back transformation phase*: if required, the left and the right singular vectors of A are computed by multiplying $\tilde{U}$ and $\tilde{V}^T$ by the orthogonal matrices Q and P used in the reduction phase.

It is well known that the first phase is the most time consuming portion of the SVD problem [20]. Benchmarks show that it consists of more than 70% or 90% of the total time when all singular vectors or only singular values are computed on modern architectures, respectively. For that, we focus in this paper on the reduction phase for a batch of small problems, and study its limitations.

4. Householder bi-diagonalization

The bi-diagonalization factorizes $A = U\,BV^T$, where U and V are orthogonal matrices, and B is bi-diagonal with non-zeros only on the diagonal and upper superdiagonal. This is done by the classic stable Golub–Kahan method that applies a sequence of Householder transformations [21]. Algorithmically, this corresponds to a sequence of in-place transformations, where A is overwritten by the entries of the bi-diagonal matrix B, as well as by the U and V holding the vectors defining the left and right householder reflectors, respectively:

$$\begin{bmatrix} a_{11}^{(0)} & a_{12}^{(0)} & a_{13}^{(0)} & a_{14}^{(0)} \\ a_{21}^{(0)} & a_{22}^{(0)} & a_{23}^{(0)} & a_{24}^{(0)} \\ a_{31}^{(0)} & a_{32}^{(0)} & a_{33}^{(0)} & a_{34}^{(0)} \\ a_{41}^{(0)} & a_{42}^{(0)} & a_{43}^{(0)} & a_{44}^{(0)} \end{bmatrix} \rightarrow \begin{bmatrix} b_{11} & a_{12}^{(1)} & a_{13}^{(1)} & a_{14}^{(1)} \\ v_1 & a_{22}^{(1)} & a_{23}^{(1)} & a_{24}^{(1)} \\ v_1 & a_{32}^{(1)} & a_{33}^{(1)} & a_{34}^{(1)} \\ v_1 & a_{42}^{(1)} & a_{43}^{(1)} & a_{44}^{(1)} \end{bmatrix} \rightarrow$$

$$\rightarrow \begin{bmatrix} b_{11} & b_{12} & u_1 & u_1 \\ v_1 & a_{22}^{(2)} & a_{23}^{(2)} & a_{24}^{(2)} \\ v_1 & a_{32}^{(2)} & a_{33}^{(2)} & a_{34}^{(2)} \\ v_1 & a_{42}^{(2)} & a_{43}^{(2)} & a_{44}^{(2)} \end{bmatrix} \rightarrow \begin{bmatrix} b_{11} & b_{12} & u_1 & u_1 \\ v_1 & b_{22} & b_{23} & u_2 \\ v_1 & v_2 & b_{33} & b_{34} \\ v_1 & v_2 & v_3 & b_{44} \end{bmatrix}.$$

This algorithm is sequential and rich in Level-2 BLAS GEMV routine calls that are applied in every step for updating the rest of the matrix. The blocked two-phase algorithm is described in Algorithm 1. The factorization of the panel A_{ix}, A_{iy} proceeds in n/n_b steps of blocking size n_b. Each step is composed of BLAS and LAPACK routines, e.g., the Level-3 BLAS GEMM routine is used for the trailing matrix update, and the LAPACK's LABRD routine is used for the panel factorization. LABRD is still sequential and composed of Level-2 BLAS GEMV. LABRD saves Householder transformations in matrices X and Y, respectively. Once the transformations are accumulated within the panel, they are applied to the trailing matrix using Level-3 BLAS operations. The blocked algorithm casts half of the flops of the original sequential algorithm from Level-2 BLAS to Level-3 BLAS GEMM operations.

Algorithm 1. Two-phase implementation of the Householder GEBRD algorithm. Without loss of generality, A is assumed to be of size $n \times n$. $A(i:j, m:k)$ is the submatrix of A consisting of ith through jth row and mth through kth column with 0-based indexing.

```
for $i \in \{1, 2, 3, \ldots, n/n_b\}$ do
    $A_{ix} := A_{(i-1)\times n_b:(n-1),(i-1)\times n_b:i\times n_b}$
    $A_{iy} := A_{(i-1)\times n_b:i\times n_b,(i-1)\times n_b:(n-1)}$
    $C_i := A_{i\times n_b:(n-1),i\times n_b:(n-1)}$
    Panel Factorize using LABRD to reduce $A_{ix}$ and $A_{iy}$ to bi-diagonal form;
    returns matrices X, Y to update trailing matrix $C_i$ in the next phase; $U$, $V$ are
    stored in the factorized part of $A$.
    Trailing Matrix Update $C_i = C_i - V \times Y^T - X \times U^T$ with gemm
end for
```

5. Performance bound analysis and roofline model

In order to evaluate the performance behavior of the reduction to bi-diagonal and to analyze if there are opportunities for improvements, we present a performance bound analysis and the associated with it roofline model. Similar to the one-sided factorizations (LU, Cholesky, QR), the two-sided factorizations (in particular, the bi-diagonal reduction) are split into a *panel factorization* and a *trailing matrix update*. Unlike the one-sided factorizations, the panel factorization requires computing Level-2 BLAS matrix–vector products involving the entire trailing matrix. This requires loading the entire trailing matrix into memory, and thus, incurring a significant amount of memory-bound operations. The application of two-sided transformations creates data dependencies and produces artificial synchronization points between the panel factorization and the trailing submatrix update. This makes it impossible to overlap the panel and the trailing submatrix update. Therefore, we can model the performance of our algorithm by the performances of its basic kernels (as they have to be executed in order).

The algorithm proceeds by steps of size n_b. We give the detailed *panel* and *update* costs per step:

- The panel is of size n_b columns. The factorization of every column is primarily dominated by two matrix–vector products with the trailing matrix. Thus, the cost of a panel is $4\, n_b\, l^2 + \Theta(n)$, where l is the size of the trailing matrix at step i. For simplicity, we omit $\Theta(n)$ and roundup the cost of the panel by the cost of the matrix–vector product;
- The update of the trailing matrix consists of applying the householder reflectors generated during the panel factorization to the trailing matrix from both the left and the right side using Level-3 BLAS routines: $A_{i*n_b:n-1,i*n_b:n-1} \longleftarrow A_{i*n_b:n-1,i*n_b:n-1} - V \times Y^T - X \times U^T$, where V and U are the householder reflectors computed during the panel phase, X and Y are two rectangular matrices needed for the update and also computed during the panel phase. This update phase can be performed by two matrix–matrix products using the gemm routine and its cost is $2 \times 2\, n_b\, k^2$, where k is the size of the trailing matrix at step i.

For all steps (n/n_b), the trailing matrix size varies from n to n_b by steps of size n_b, where l varies from n to n_b and k varies from $(n - n_b)$ to 2 n_b. Thus, the total cost for the n/n_b steps is:

$$\approx 4n_b \sum_{n_b}^{n/n_b} l^2 + 4n_b \sum_{2n_b}^{\frac{n-n_b}{n_b}} k^2 \approx \frac{4}{3} n^3_{\text{gemv}} + \frac{4}{3} n^3_{\text{gemm}} \approx \frac{8}{3} n^3.$$

According to the above equation, we derive below the maximum performance P_{max} that can be reached by the bi-diagonal reduction algorithm as a function of the performances P_{gemm} and P_{gemv} for the gemm and gemv kernels, respectively. In particular, for large matrix sizes n:

$$P_{\max} = \frac{flops}{t_{\min}} = \frac{\frac{8}{3}n^3}{\frac{4}{3}n^3 * \frac{1}{P_{gemv}} + \frac{4}{3}n^3 * \frac{1}{P_{gemm}}} \qquad (1)$$
$$= \frac{2 * P_{gemm} * P_{gemv}}{P_{gemm} + P_{gemv}} < 2P_{gemv}, \quad \text{when} \quad P_{gemm} >> P_{gemv}.$$

The performance of the Level-2 BLAS routines such as the batched matrix–vector multiplication (gemv) is memory bound and very low compared to the Level-3 BLAS dgemm. For example, on a K40c GPU the performance of batched_dgemv is about 40 Gflop/s as shown in Fig. 4(a), while for batched_dgemm it is about 323 Gflop/s as illustrated in Fig. 2. Thus, one can expect from Eq. (1), that the performance of the reduction algorithm is bound by the performance of the Level-2 BLAS operations. This explains the well known low performance behavior observed for the algorithm.

6. Batched BLAS design and implementation for GPUs

In a batched problem that is based on batched BLAS, many small dense matrices must be factorized simultaneously, meaning that all the matrices will be processed simultaneously by the same kernel.

6.1. Two-level parallelism and device-kernel mode

Our batched BLAS kernels do not make any assumption about the layout of the matrices in memory, e.g., the matrices are not necessarily stored continuously. The starting address of every matrix is stored in an array of pointers, and the batched kernel takes the array of pointers as input. Note that to use the array of pointers interface, extra memory must be allocated as workspace, compared to the assumption of consecutive matrix storage. Inside the kernel, each matrix is assigned to a unique batch ID and processed by one device function. Device functions are low-level and callable only by CUDA or HIP kernels. The *device function* only sees a matrix by the batched ID and thus still maintains the same interface as the standard BLAS. Moreover, we use multiple GPU threads per matrix factorization, which is different from [22], where only one thread is used. Thus, our batched BLAS is characterized by two levels of parallelism. The first level is a *task-level parallelism* among the independent matrices that are simultaneously processed. The second is *fine-grained data parallelism* within the computation of each matrix and its goal is to exploit the SIMT architecture of the GPU through device functions. This strategy yields higher parallelism in our algorithms (occupancy) that results in better use of the GPU, and therefore higher performance.

The device functions are templated with CUDA and HIP C++ for NVIDIA and AMD GPUs, respectively. A number of tunable parameters are selected – thread blocks size, tile size, etc.; see Section 7.2 – and stored in C++ template parameters. The use of device functions brings multiple advantages. First, merging kernels can reduce kernel launching overhead but that usually demodulizes the BLAS-based structure of an LAPACK algorithm. However, device functions

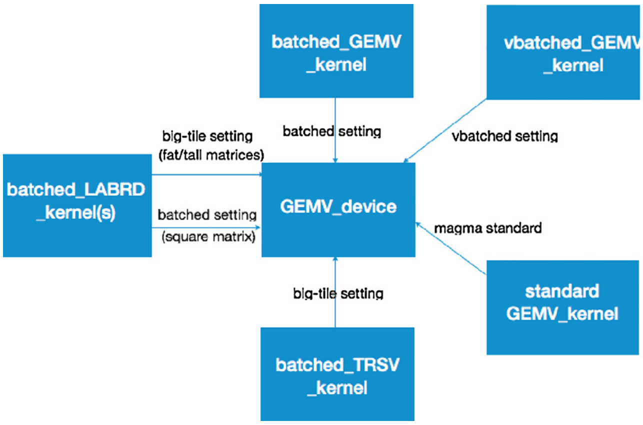


Fig. 1. The same GEMV device function is called by various kernels.

preserve the BLAS interface. Multiple small workload device function can be merged in one kernel easily but with the BLAS-based LAPACK algorithm structure still gracefully maintained. Second, since shared memory is alive per kernel-life-time, multiple device functions can access the same shared memory to improve data reuse. Merging kernels and data reuse is important to GEBRD. They are possible because the panel factorization stage has many small computations that if merged have a good possibility of data reuse, e.g., in reusing the Householder vector. In order to reuse data in shared memory, we propose a big-tile setting which will be described in the next section. Third, if the underlying computation is the same, only one copy of device function is maintained for different kernels. Fig. 1 shows that the same GEMV device function can be called by different types of kernels, standard GEMV (targeting a large matrix instead of many small ones), batched GEMV, LABRD, and TRSV kernels. Each type of kernel requires optimization accordingly. We use auto-tuning techniques (see Section 7.2) to find the optimal setting for each particular kernel as shown in the figure. Thus, using a *device functions*-based methodology to designs for our internal APIs and to build higher-level algorithms on top of these APIs, is a central characteristic of our developments towards taking advantage of the aforementioned benefits.

6.2. Data reuse and degrees of parallelism

An important optimization technique in GPU programming is to load frequently accessed data in shared memory to perform computations as much as possible before storing back results to the GPU main memory. However, shared memory is private per thread block. When solving just one large matrix problem, the matrix is divided into tiles with each tile loaded in shared memory. Different thread blocks access different tiles in an order determined by the algorithm. Synchronization of the computation of the tiles is accomplished by ending and re-launching kernels. When one kernel exits, the resulting data in shared memory must be stored back to the GPU main memory as the shared memory will be flushed. However, in small-sized batched problems, too many kernel launches should be avoided, especially in the panel factorization, where each routine has a small workload, and a high probability of data reuse exists in shared memory (if kernels are merged).

Therefore, to avoid many kernel launches, in our design, each matrix is assigned to a thread block, and the synchronization is accomplished by barriers inside the thread block. We call this thread block setting *big-tile setting*. The naming is from this observation: if the tile (i.e., thread block size) is big enough that a whole matrix (from the batch) is inside the tile, that matrix computation reduces to the point that one thread block accesses the whole matrix.

However, compared to the big-tile setting, the classic setting with multiple thread blocks processing one matrix has a higher degree of parallelism as different parts of the matrix are processed simultaneously, especially for large square matrices. Thus, overall, there is a trade-off. Big-tile setting allows data to be reused through shared memory, but suffers a lower degree of parallelism. The classic setting has a higher degree of parallelism, but may lose the data reuse benefits. The optimal setting depends on many factors, including the algorithm type and matrix size, and is often selected by auto-tuning (as in Section 7.2). Our experience shows that for the panel factorization, the big-tile setting has advantage. While for the trailing matrix update with GEMM computation, the classic setting is preferred.

6.3. Batched bi-diagonalization implementations on GPUs

One approach to the batched problems is to consider that the entire matrix is small enough to fit into shared memory. For example, the current size of the shared memory is 48 KB per streaming multiprocessor (SMX, or computing unit on AMD GPU) for the high-end NVIDIA K40c GPUs, which is a low limit for the amount of batched problems data that can fit at once. However, completely saturating the shared memory per SMX can decrease the performance of memory-bound routines since only one thread-block will be mapped to that SMX at a time. Due to a limited parallelism in the factorization of a small panel, the number of threads used in the thread block will be limited, resulting in low occupancy, and subsequently poor core utilization. The advantages of multiple blocks residing on the same SMX is that the scheduler can swap out a thread block waiting for data from memory and push in the next block that is ready to execute [23]. We found that using *a small amount* of shared memory per kernel (less than 10 KB) not only provides an acceptable data reuse, but also allows many thread-blocks to be executed by the same SMX concurrently, thus taking better advantage of its resources.

For good performance of Level-3 BLAS in trailing matrix updates, panel width n_b needs to be increased. Yet, increasing n_b increases tension as the panel is a sequential operation – a larger panel width results in larger Amdahl's sequential fraction which governs the maximum speedup. The best panel size is usually a trade-off product by balancing the two factors and is obtained by tuning. We discovered empirically that the best value of n_b for one-sided factorizations is 32 [14–16]. However, 16 or 8 is optimal for the two-sided bi-diagonalization. A smaller n_b is better because the panel operations (mainly GEMV operations) in the two-sided factorizations are more significant than the panel operations in the one-sided factorizations.

GEBRD panel with LABRD: This provides the batched equivalent of LAPACK's LABRD routine that reduces the first n_b rows and columns of an m by n matrix A to upper or lower real bi-diagonal form by Householder transformations, and returns the matrices X and Y that later are used to apply the transformation to the unreduced trailing matrix. It consists of n_b steps where each step calls two routines generating householder reflectors (LARFG), one for column and one for row householder reflector, and a set of GEMV and scaling SCAL routines. The LARFG involves a norm computation followed by a SCAL that uses the results of the norm computation in addition to some underflow/overflow checking. The norm computation is a sum reduce and thus a synchronization step. To accelerate it, we implemented a two-layer tree reduction where for sizes larger than 32, all 32 threads of a warp progress to do a tree reduction to reduce to 32 elements. The last 32 elements are reduced to one by only one thread. The householder reflectors are frequently accessed and are loaded in shared memory. A set of GEMV routines are called to update the rest of the panel and matrices X and Y. Since there are n_b steps, these routines are called n_b times; thus, one can expect that the performance depends on the performances of Level-2 and Level-1 BLAS operations. Hence, it is a slow, memory-bound routine.

Trailing matrix updates with GEMM: The update is achieved by two GEMMs with the matrices X and Y returned from the panel factorization. The first one is a GEMM with a non-transpose matrix and a transpose matrix ($A = A - V * Y'$), followed by another GEMM with a non-transpose matrix and a non-transpose matrix ($A = A - X * U'$). The update is directly applied on trailing matrix A. However, for very small matrices it might be still difficult to extract performance from Level-3 BLAS kernels.

7. Auto-tuning

The efforts of maximizing the performance of BLAS, especially GEMM, generally fall into two directions: writing assembly code and source level code tuning. The vendor libraries (e.g., Intel MKL, AMD ACML and hipBLAS, NVIDIA CUBLAS) supply their own routines on their hardware. To achieve performance, the GEMM routine is implemented in assembly code, like the CUBLAS GEMM on Kepler GPUs. The assembly code usually delivers high performance. A disadvantage is that it is highly architecture-specific. The vendors maintain the performance portability across different generations of their architectures [24]. Another direction is to explore the source level code auto-tuning to achieve optimal performance (within a preset kernel design space). Different from assembly code, source code auto-tuning relies on the compilers to allocate registers and schedule instructions. The advantage is that source code is architecturally independent and is easy to maintain. Our effort focuses on source code auto-tuning.

7.1. Batched level-3 BLAS GEMM

The performance of linear algebra routines highly relies on the Level-3 BLAS GEMM. Our batched GEMM is modified from the standard MAGMA GEMM [25]. The template parameters of our batched GEMM include the number of threads, the size of shared memory, and the data tile size. The product of the sizes for these parameters produces a large search space, but it can be powerfully pruned by constraints. The derived constraints of the search space include correctness, as well as hardware constraints and soft constraints. Hardware constraints stem from the realities of the accelerator architecture, like registers and shared memory size. Based on these metrics, invalid kernels violating the hardware requirement (like exceeding 48 KB shared memory) will be discarded. The constraints may be soft in terms of performance. We require at least 512 threads per GPU SMX (CU on AMD) to ensure a reasonable occupancy. More details about tuning batched GEMM can be found in [26].

Fig. 2 shows our batched DGEMM (denoted as the MAGMA batched) performance against other solutions after auto-tuning. The number of matrices is 400. The best CPU solution is to parallelize with 16 OpenMP threads on a 16-core Sandy Bridge CPU. Its performance is stable around 100 Gflop/s. The non-batched GPU solution is a loop over the 400 matrices by calling standard GEMM routine, where the GPU sequentially processes each matrix and relies on the multi-threading per matrix to achieve performance. The non-batched curve linearly grows below size 320 and catches up with CUBLAS batched GEMM around size 448. Our MAGMA batched GEMM outperforms other solutions. It is 75 Gflop/s faster (i.e., 30% faster) than CUBLAS on average and more than 3× faster than the CPU solution. Fig. 3 also show the autotuning of the DGEMV routine for wide matrices meaning matrices with small number of row (e.g., corresponding to the panel size of GEBRD) and large number of columns.

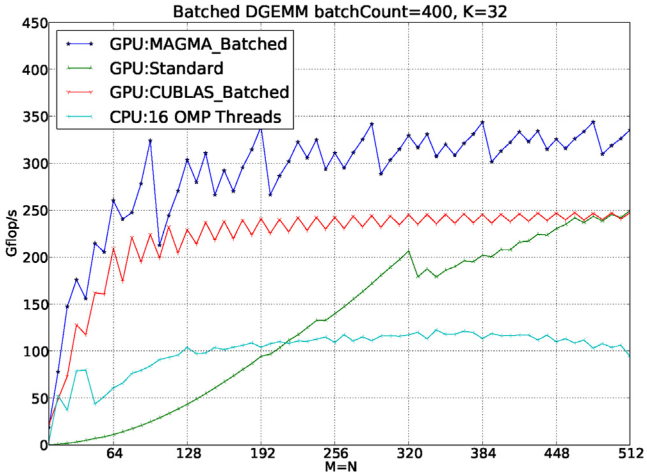


Fig. 2. Performance of our batched DGEMM ($K = 32$) vs. other solutions on CPUs or GPUs.

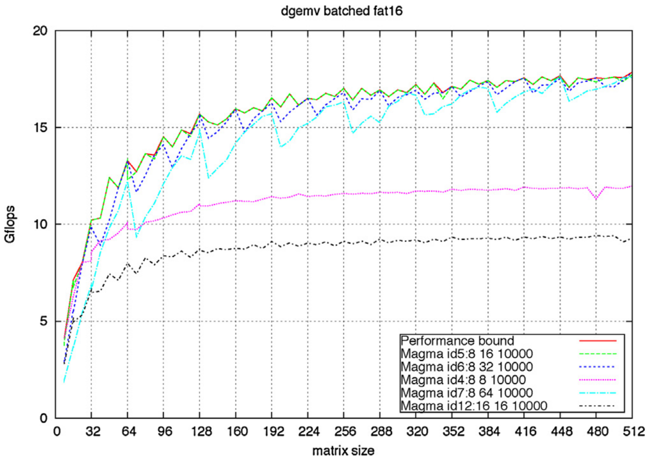


Fig. 3. Tuning results of batched DGEMV for a wide matrix where $m = 16$ and n is reported in the x-axis.

7.2. Different batched level 2 BLAS GEMV instances tuning

In matrix–vector multiplication using a non-transpose matrix (GEMVN), a reduction is performed per row. Each thread is assigned to a row and a warp of threads is assigned to a column. Each thread iterates row-wise in a loop and naturally owns the reduction result. Since matrices are stored in column-major format, the data access in GEMVN by the warp is in a coalescing manner.

However, in GEMV using a transpose matrix (GEMVT), the reduction must be performed on each column. Assigning a thread to a column will make the reduction easy, but will lead to memory access in a striding way. To overcome the non-coalescing problem in GEMVT, a two-dimension thread block configuration is adopted. Threads in x-dimension are assigned per row. These threads access data row-wise to avoid the memory non-coalescing penalty. A loop of these threads over the column is required in order to do the column reduction in GEMVT. Partial results owned by each thread are accumulated in every step of the loop. At the final stage, a tree reduction among the threads is performed to obtain the final result, similar to MPI_REDUCE.

Threads in y-dimension are assigned per column. A outside loop is required to finish all the columns. Threads in x-dimension ensure the data access is in a coalescing pattern. Threads in *y*-dimension preserve the degree of parallelism, especially for the wide matrix (or called fat matrix, with both terms being interchangeable throughout this paper) where the parallelism is more critical to performance.

8. Performance on NVIDIA GPU

We conducted our experiments on a multicore system with two 8-cores socket Intel Xeon E5-2670 (Sandy Bridge) processors with each running at 2.6 GHz. Each socket has a shared 20 MB L3 cache, and each core has a private 256 KB L2 and a 64 KB L1 cache. The system is equipped with 64 GB of memory and the theoretical peak in double precision is 20.8 Gflop/s per core, i.e., 332.8 Gflop/s in total for the two sockets. It is also equipped with an NVIDIA K40c GPU with 11.6 GB GDDR memory per card running at 825 MHz. The theoretical peak in double precision is 1430 Gflop/s. The GPU is connected to the CPU via PCIe I/O hubs with 6 GB/s bandwidth.

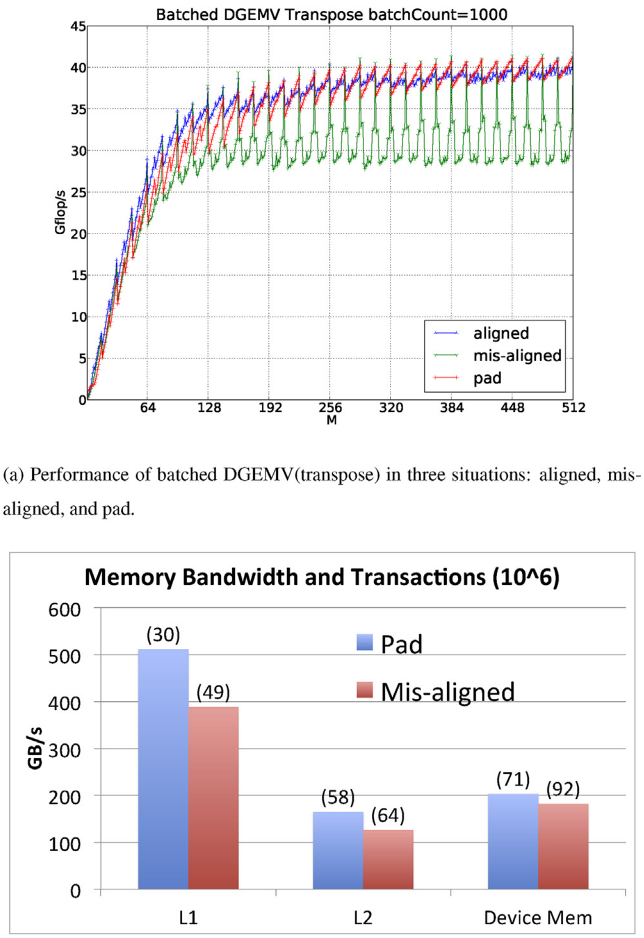
In our testing, we assume the data already resided in the processor's memory. Unless explicitly noted, the memory transfer time between processors is not considered. We believe this is a reasonable assumption since the matrices are usually generated and processed on the same processor. For example, in the high order FEMs, each zone assembles one matrix on the GPU. The conjugation is performed immediately, followed by a batched GEMM. All the data is generated and computed on the GPU [1].

8.1. Performance study and optimization of the bi-diagonalization

Since the performance of the batched GEMV on K40c is around 40 Gflop/s (as shown in Fig. 4(a)), the GEBRD roofline bound is 80 Gflop/s according to Eq. (1). A sharper bound, using Eq. (1) and that the Batched GEMM performance is 323 Gflop/s, is $\frac{2*323*40}{323+40} \approx$ 71 Gflop/s.

Fig. 5 demonstrates the performance improvement progress of our implementation. The *non-blocked version* purely composed of Level 2 BLAS operations does not scale any more after size 256. The first non-optimized blocked *version v1* follows the LAPACK's two-phase implementation as depicted in Algorithm 1 in which the trailing matrix is updated with Level 3 BLAS operations. Additional memory allocation overhead has to be introduced in order to use the array of pointers interfaces in the blocked algorithm. Below size 224, the performance of version v1 is even slower than the non-blocked due to the overhead. Beyond 224, it starts to grow steadily because of GEMM performance.

The main issue of the blocked version v1 is that the GEMVs are not optimized for instances required by the GEBRD. By tuning these GEMVs, as described in Section 7.2, the performance is immediately doubled in *version v2*. These GEMV routines are called in the form of device functions in the panel factorization kernel. The column/row vector of householder reflectors and the to-be-updated column in matrices *X* and *Y* are repeatedly accessed at each step. We load them into fast on-chip shared memory. In order to reuse and synchronize data in shared memory, one matrix can not span multiple thread blocks. Therefore, we adopt the big-tile setting for the GEMV device functions in v2. As discussed in Section 6.1, there is a trade-off between data reuse (with big-tile setting) and the degree of parallelism (with classic setting). We found there is a switch over at size 128 for the two settings. We adopt classic setting beyond size 128 and big-tile for size less than 128 for square instances. The big-tile setting is still adopted for other wide/tall instances because the data caching proves to be more important. By this switch-over, the performance of *version 3* boosts to 50 Gflop/s from 40 Gflops in version 2 at size 512.



(a) Performance of batched DGEMV(transpose) in three situations: aligned, mis-aligned, and pad.

(b) Number of transactions (on top of the bar, in millions) and achieved bandwidth of the y axis.

Fig. 4. Effect of the padding.

By profiling the GEMV time in GEBRD step by step, we find it does not match the optimal performance obtained in our auto-tuning. In Fig. 4(a), the blue curves depicts the performance of GEMV transpose of double precision with every matrix being aligned in memory. However, when the algorithm iterates the sub-matrix as in GEBRD factorization, the starting address may not be aligned

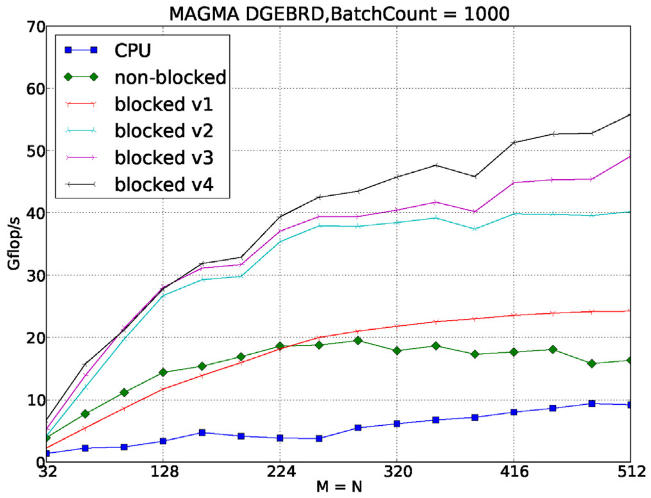


Fig. 5. Performance of batched dgebrd: progress of different versions on a K40c GPU.

(green curve). The performance curve fluctuates because when the starting address of the sub-matrix is aligned in memory, the peak performance is reached; otherwise, it drops drastically. The fluctuation is more serious for bigger matrices since most threads are mis-aligned as more threads are used in large size.

To overcome the fluctuation issue, we adopt a padding technique. The starting thread always reads from the recent upper aligned address. It introduces extra data reading. The extra reading is up to 15 elements per row because 16 threads fit in an aligned 128-byte segment as a double element is of 8 byte. Although more data is read, it is coalescing that the 128-byte segment can be fetched by only one transaction. Overall the number of memory transactions is reduced as shown in Fig. 4(b). Since the number of memory transactions decreases, the bandwidth is improved accordingly. By padding elements in the multiplied vector as zeros, extra results are computed but finally discarded in the writing stage. Fig. 4(a) shows that our padding technique enables the GEMV in the GEBRD algorithm to run at a speed close to the aligned speed. By padding, *version 4* reaches 56 Gflop/s at size 512 which is 80% of the upper bound of the performance.

A breakdown of different components contributing to the overall time of GEBRD version 4 is depicted in Fig. 6. As the matrix size (n) increases, the time of the square matrix (in blue) begins to dominate. At a smaller size (n), the percentage of the wide/ tall matrix (asymptotically of size n_b by n) to the square matrix (asymptotically of size n by n) is larger, since n_b is fixed. Thus, the time spending on the wide/tall matrix (in red) is more prominent. The GEMM time is relative stable around 10% across different size problems.

9. Batched GEMV and GEMM performance on AMD GPU

Recently, AMD introduced the Radeon Open Compute Platform (ROCm) open-sourced platform for GPU-based high-performance computing [27]. The ROCm ecosystem includes Linux kernels, runtimes, the HCC compiler, and high level mathematical libraries. ROCm introduces a Heterogeneous–Compute Interface for Portability (HIP) layer allowing users to create portable applications that run on AMD and NVIDIA GPUs with similar source code. Compared to the OpenCL C language, HIP is a C/C++ runtime API and kernel language that is very familiar to CUDA users [19]. Most GPU kernel optimizations techniques on OpenCL and CUDA equally apply on HIP programming.

AMD provides their vendor-optimized GPU BLAS, called hipBLAS [28]. Like cuBLAS, hipBLAS provides standard GEMM and GEMV, but no batched GEMV. Different than cuBLAS, hipBLAS only provides a strided batched GEMM routine, where the batched matrices are not stored in an array of pointers. Instead, a pointer is provided that points to the very first matrix, and the following matrices are located with a fixed address offset called stride.

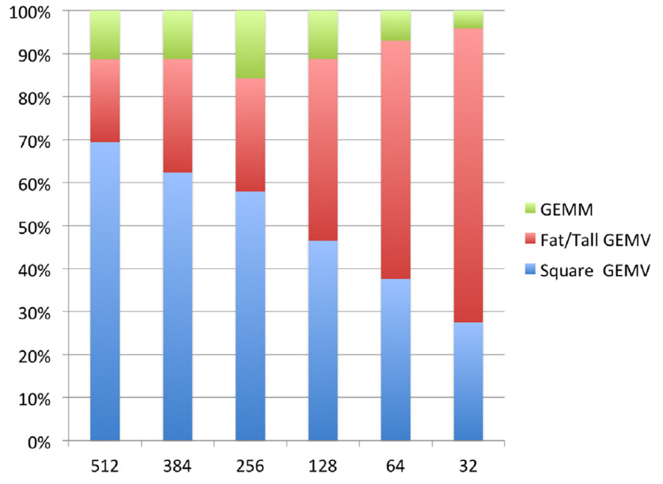


Fig. 6. A time breakdown of batched GEBRD on a K40c GPU. (For interpretation of the references to color in text, the reader is referred to the web version of the article.)

Table 1
Hardware specification of NVIDIA K40c and AMD R9 Fiji Nano.

GPU	Bandwidth (GB/s)	DP peak performance (Gflop/s)	Memory size (GB)
K40c	288	1430	12
Fiji Nano	512	512	4

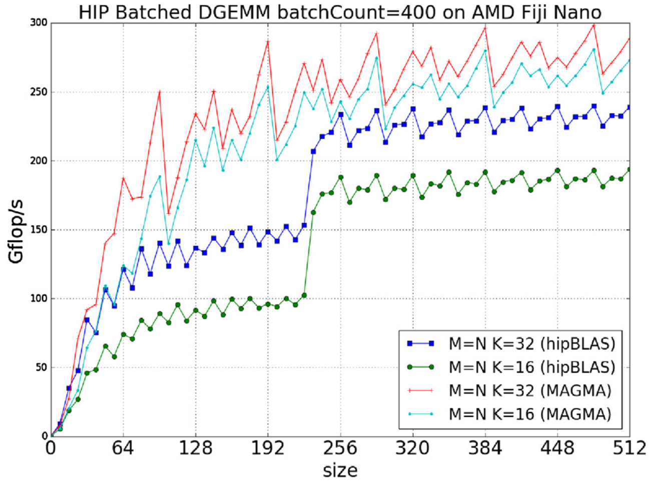


Fig. 7. Performance of hipBLAS and MAGMA batched DGEMM for $K = 32$ and $K = 16$ on AMD Fiji Nano.

The AMD GPU that we use to test and present our results is a Radeon R9 Fiji Nano. Fiji Nano is equipped with 4 GB HBM memory. The specification of Fiji Nano vs. K40c is outlined in Table 1. Fiji Nano has much less double precision (DP) peak performance, but otherwise owns higher bandwidth. The software environment is the ROCm v1.6.3 with the HCC compiler based on Clang 6.0 [29].

To run on AMD GPUs, we need to *hipify* our CUDA source code into HIP code with the assistance of the AMD provided *hipify-perl* script. We found that most of our batched GEMV and GEMM CUDA code, including device functions, and templates, can be hipified successfully except that we had to manually add an extra “hipLaunchParm lp” parameter to every HIP kernel.

We compare our batched GEMM against hipBLAS in three different instances: square matrix, $K = 32$, and $K = 16$, as these are of interest in the GEBRD routine. From Fig. 7, the MAGMA HIP code outperforms hipBLAS for every $M = N$ size for $K = 32$ and $K = 16$. MAGMA can reach 300 Gflop/s while hipBLAS fluctuates around 240 Gflop/s. One interesting thing is, there is a sudden jump at size 224 for hipBLAS, indicating that there is either a tuning or algorithm change at that switch-over. Compared to CPU in Fig. 2, cuBLAS and hipBLAS (after size 250) both climb close to 250 Gflop/s and exceed the MKL with 16 threads on Sandy Bridge CPU, which achieves only half of the performance on GPU.

The square matrix result is shown in Fig. 8. MAGMA wins over hipBLAS before size 250. Beyond it, hipBLAS and MAGMA are very close, and both stable at round 300 Gflop/s. If we compare this figure with Fig. 2, the MAMGA HIP code fluctuates around 275 Gflop/s, while the MAGMA CUDA code fluctuates at around 325 Gflop/s. This can be partially explained by the fact that the K40c GPU owns higher double precision computing capability (1.4 Tflop/s), which is about 2.8× that of the Fiji Nano (0.5 Tflop/s). However, the 2.8× cannot fully justify the fact that K40c is only 50 Gflop/s (18%) faster here. The reason is that batched GEMM is not totally compute-bound as

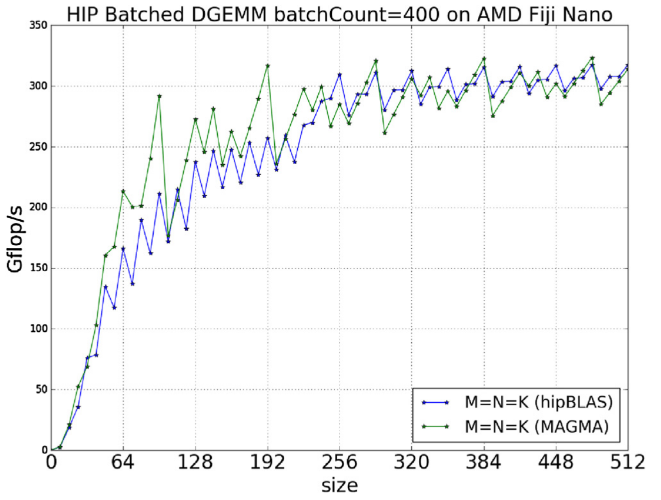


Fig. 8. Performance of hipBLAS and MAGMA batched DGEMM for square matrix on AMD Fiji Nano.

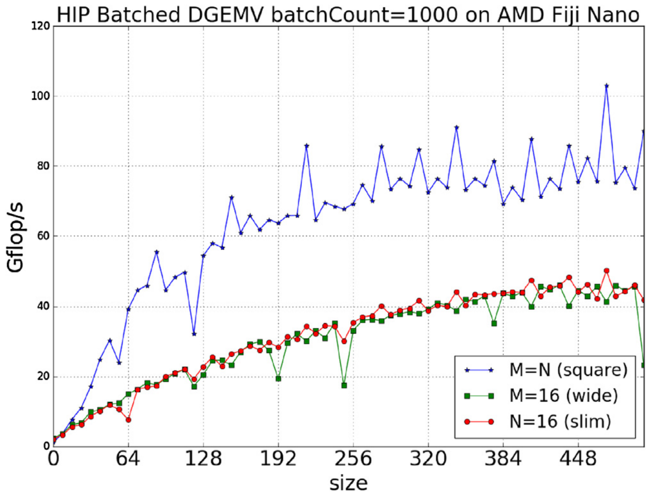


Fig. 9. Performance of MAGMA HIP batched DGEMV on AMD Fiji Nano.

the standard BLAS Level-3 GEMM, but a mix of compute-bound and bandwidth-bound routine. For the same byte access, the batched GEMM on small matrices has much less floating-point operations than the standard GEMM for larger matrices, as explained in [1]. When the matrix size is smaller, the performance is more bounded by bandwidth than the compute capability, which is the case for batched BLAS where matrices are much smaller than in standard BLAS. Although Fiji Nano has less computing capability, it owns higher bandwidth than the K40c, which offsets its double precision shortage.

The bandwidth advantage of the Fiji Nano is further shown in the fully memory-bounded BLAS Level 2 batched GEMV routine, as in Fig. 9. Because neither cuBLAS nor hipBLAS provides such routine, we compare the MAGMA CUDA code and the HIP code. For square matrices, the MAGMA HIP code fluctuates around 80 Gflop/s and reaches up to 100 Gflop/s on the Fiji Nano, while the MAGMA CUDA code stabilizes at around 40 Gflop/s, as in Fig. 4(a). For wide (or fat) matrices (e.g., $M = 16$), the MAMGA HIP code reaches more than 40 Gflop/s on the Fiji Nano GPU. In contrast, the MAGMA CUDA code stabilizes at around 17 Gflop/s on the K40c, as in Fig. 3.

10. Conclusions and future work

The use of GPUs to accelerate scientific codes has been observed extensively on large dense and sparse linear algebra problems that feature abundant data parallelism. Solving small linear algebra problems on current high-end many-core systems is more challenging. Still, small problems can be implemented relatively easy for multicore CPUs by taking advantage of the large CPU caches for data reuse. On the other hand, the development of small problems on GPUs is not as straightforward. We demonstrated that with a batched approach, small problems can be accelerated on GPUs to have a significant advantage over CPUs, as well.

We consider a batched two-sided bi-diagonalization based on the batched BLAS approach. We first adopt optimal blocking algorithm to maximize the GPU-friendly GEMM operations. We then propose device functions as the underlying components of the batched BLAS kernels. The use of device functions allows the data to be reused through shared memory and avoids multiple small kernel launches, but without demodulizing the BLAS-based LAPACK algorithm structure at the same time. The device functions are CUDA/HIP C++ templated. Auto-tuning is used to help find the optimal template setting for different types of kernels, e.g., standard, batched, or different instances for a type of kernel, like transpose, wide, slim for GEMV. Because the GEBRD algorithm iterates the sub-matrix resulting in a mis-aligned starting address, we adopt padding techniques to overcome the fluctuation. Using these techniques, we achieve 56 Gflop/s, which is 80% of the upper bound performance for the bi-diagonalization problem on a Kepler K40c GPU.

We compare our batched BLAS across different platforms and vendor-optimized libraries, whenever they have the corresponding routines, including Intel MKL with 16 threads on Sandy Bridge CPU, Nvidia cuBLAS on K40c GPU, and AMD hipBLAS on R9 Fiji Nano GPU. Our test shows that our batched GEMM implementation exceeds all of them in performance for the sizes of interest. For GPUs, we found that the HIP code on Fiji Nano is 2× faster than the CUDA code on K40c for the memory-bound GEMV routine. For batched DGEMM, Fiji Nano has inherent disadvantages in double precision computing ability, but heavily alleviated by its high bandwidth advantage.

The methods in this paper can be applied to other two-sided factorizations, e.g., the Hessenberg reduction (GEHRD) and the tri-diagonalization (SYTRD), as well. Besides the GEMV, GEHRD and SYTRD use additional BLAS-2 triangular matrix–vector multiplication (TRMV) and symmetric matrix–vector multiplication (SYMV) operations, respectively. The same optimization techniques used for GEMV are applicable to TRMV and SYMV.

Acknowledgements

This work is partially supported by NSF Grant No. SI2:SSE 1740250, and the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering and early testbed platforms, in support of the nations exascale computing imperative.

References

[1] T. Dong, V. Dobrev, T. Kolev, R. Rieben, S. Tomov, J. Dongarra, A step towards energy efficient computing: redesigning a hydrodynamic application on CPU–GPU, IEEE 28th International Parallel Distributed Processing Symposium (IPDPS) (2014).

[2] L. Brown, Accelerate Machine Learning with the cuDNN Deep Neural Network Library, 2015 http://devblogs.nvidia.com/parallelforall/accelerate-machine-learning-cudnn-deep-neural-network-library/.

[3] PArallel Distributed Deep LEarning (Paddle), Available at https://github.com/PaddlePaddle/Paddle, 2017.

[4] Theano Development Team, Theano: A Python Framework for Fast Computation of Mathematical Expressions, arXiv e-prints http://arxiv.org/abs/1605.02688.

[5] TensorFlow Development Team, TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, Software Available at http://tensorflow.org/.

[6] R. Collobert, K. Kavukcuoglu, C. Farabet, Torch7: a Matlab-like environment for machine learning, BigLearn, NIPS Workshop (2011).

[7] O. Messer, J. Harris, S. Parete-Koon, M. Chertkow, Multicore and accelerator development for a leadership-class stellar astrophysics code, Proceedings of PARA 2012: State-of-the-Art in Scientific and Parallel Computing (2012).

[8] J. Molero, E. Garzón, I. García, E. Quintana-Ortí, A. Plaza, Poster: A Batched Cholesky Solver for Local RX Anomaly Detection on GPUs, PUMPS, 2013.

[9] N. Corporation, https://devtalk.nvidia.com/default/topic/527289/help-with-gpu-cholesky-factorization-/.

[10] Batched SVD, Available at https://devtalk.nvidia.com/default/topic/851534/batched-svd-/, 2015.

[11] J. Dongarra, I. Duff, M. Gates, A. Haidar, S. Hammarling, N.J. Higham, J. Hogg, P. Valero-Lara, S.D. Relton, S. Tomov, M. Zounon, A proposed API for batched basic linear algebra subprograms, in: MIMS EPrint 2016.25, Manchester Institute for Mathematical Sciences, The University of Manchester, UK, 2016 http://eprints.ma.man.ac.uk/2464/.

[12] S. Tomov, J. Dongarra, M. Baboulin, Towards dense linear algebra for hybrid GPU accelerated manycore systems, Parellel Comput. Syst. Appl. 36 (5–6) (2010) 232–240, http://dx.doi.org/10.1016/j.parco.2009.12.005.

[13] J. Dongarra, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, I. Yamazaki, Accelerating numerical dense linear algebra calculations with GPUs, Numer. Comput. GPUs (2014) 1–26.

[14] A. Haidar, T.T. Dong, S. Tomov, P. Luszczek, J. Dongarra, A framework for batched and GPU-resident factorization algorithms applied to block householder transformations, in: 30th Proceedings of the International Conference on High Performance Computing, ISC High Performance 2015, Frankfurt, Germany, July 12–16, 2015, 2015, pp. 31–47.

[15] T. Dong, A. Haidar, S. Tomov, J. Dongarra, A fast batched Cholesky factorization on a GPU, in: 43rd International Conference on Parallel Processing, ICPP 2014, Minneapolis, MN, USA, September 9–12, 2014, 2014, pp. 432–440.

[16] T. Dong, A. Haidar, P. Luszczek, A. Harris, S. Tomov, J. Dongarra, LU factorization of small matrices: accelerating batched DGETRF on the GPU, 16th IEEE International Conference on High Performance and Communications (HPCC 2014) (2014).

[17] A. Haidar, T. Dong, S. Tomov, P. Luszczek, J. Dongarra, Framework for batched and GPU-resident factorization algorithms to block householder transformations, in: ISC High Performance, Springer, Frankfurt, Germany, 2015.

[18] T. Dong, A. Haidar, S. Tomov, J.J. Dongarra, Optimizing the SVD bidiagonalization process for a batch of small matrices, in: International Conference on Computational Science (ICCS'17), 12–14 June 2017, Zurich, Switzerland, 2017, pp. 1008–1018, http://dx.doi.org/10.1016/j.procs.2017.05.237.

[19] Hip, Available at https://github.com/ROCm-Developer-Tools/HIP, 2016.

[20] K. Kabir, A. Haidar, S. Tomov, J. Dongarra, On the Design, Development, and Analysis of Optimized Matrix–Vector Multiplication Routines for Coprocessors, Springer International Publishing, Cham, 2015, pp. 58–73, http://dx.doi.org/10.1007/978-3-319-20119-1_5.

[21] G. Golub, W. Kahan, Calculating the Singular Values and Pseudo-Inverse of a Matrix, http://www.jstor.org/stable/2949777.

[22] V. Oreste, M. Fatica, N.A. Gawande, A. Tumeo, Power/performance trade-offs of small batched LU based solvers on GPUs, in: 19th International Conference on Parallel Processing, Euro-Par 2013, Vol. 8097 of Lecture Notes in Computer Science, Aachen, Germany, 2013, pp. 813–825.

[23] B. Rymut, B. Kwolek, Real-time multiview human body tracking using GPU-accelerated PSO, in: International Conference on Parallel Processing and Applied Mathematics (PPAM 2013), Lecture Notes in Computer Science, Springer-Verlag, Berlin, Heidelberg, 2014.

[24] Q. Wang, X. Zhang, Y. Zhang, Q. Yi, AUGEM: automatically generate high performance dense linear algebra kernels on x86 CPUs, in: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '13, ACM, New York, NY, USA, 2013, pp. 25:1–25:12, http://dx.doi.org/10.1145/2503210.2503219.

[25] R. Nath, S. Tomov, J. Dongarra, An improved magma Gemm for Fermi graphics processing units, Int. J. High Perform. Comput. Appl. 24 (4) (2010) 511–515, http://dx.doi.org/10.1177/1094342010385729.

[26] A. Abdelfattah, A. Haidar, S. Tomov, J.J. Dongarra, Performance, Design, and Autotuning of Batched GEMM for GPUs, 2016, pp. 21–38, http://dx.doi.org/10.1007/978-3-319-41321-1_2.

[27] ROCm, https://github.com/RadeonOpenCompute/ROCm, 2016.

[28] hipBLAS, https://github.com/ROCmSoftwarePlatform/hipBLAS, 2016.

[29] HCC, https://github.com/RadeonOpenCompute/hcc, 2016.