
This space is reserved for the Procedia header, do not use it

Optimizing the SVD Bidiagonalization Process for a Batch of Small Matrices

Tingxing Dong¹, Azzam Haidar², Stanimire Tomov², and Jack Dongarra^{2,3,4}

¹ Radeon Technologies Group, AMD, U.S.A

² University of Tennessee, Knoxville, USA

³ Oak Ridge National Laboratory, Oak Ridge, USA

⁴ University of Manchester, Manchester, U.K.

tingxing.dong@amd.com{haidar,tomov,dongarra}@icl.utk.edu

Abstract

A challenging class of problems arising in many GPU applications, called batched problems, involves linear algebra operations on many small-sized matrices. We designed batched BLAS (Basic Linear Algebra Subroutines) routines, and in particular the Level-2 BLAS GEMV and the Level-3 BLAS GEMM routines, to solve them. We proposed device functions and big-tile settings in our batched BLAS design. We adopted auto-tuning to optimize different instances of GEMV routines. We illustrated our batched BLAS approach to optimize batched bi-diagonalization progressively on a K40c GPU. The optimization techniques in this paper are applicable to the other two-sided factorizations as well.

Keywords:

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 8x8 and 32x32 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 [4]. 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]. In an astrophysics ODE solver [9], multiple zones are simulated, and each zone corresponds to a small linear system solve based on an LU factorization [9]. If the matrix is symmetric and definite, the problem is reduced to a batched Cholesky factorization [10, 3]. In [12], there are demands to compute one million 25x8 and n*20 SVD problems, where n ranges from 32 to 4096.

2 Related Work and Contributions

Efficient batched one-sided factorizations (LU, QR, and Cholesky) were developed in [8] [6] [5]. 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, etc.

In contrast to the compute-bound one-sided factorizations, we consider memory-bound two-sided Householder bi-diagonalizations (GEBRD). These routines are needed for 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. The goal is to develop efficient Level-2 BLAS that minimizes memory transactions and maximizes bandwidth. To accomplish this, we propose device functions and big-tile setting techniques to facilitate data reuse. Because there are various instances of GEMV to solve, we adopt auto-tuning to find the optimal setting for each instance.

Our main focus is on same-sized problems, that is, all the matrices are of identical size. Variable-sized BLAS are also considered, though to a lesser extent. 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 efficient implementations and optimization techniques; and **2)** Design two-sided bi-diagonalization for batched execution based on the batched BLAS approach.

3 Background

The SVD problem is to find orthogonal matrices U , 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. 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 $A = UB V^*$, 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 [7]. 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:

$$\begin{bmatrix} a_{11}^{(0)} & a_{12}^{(0)} & a_{13}^{(0)} & a_{14}^{(0)} \\ a_{21}^{(0)} & a_{22}^{(0)} & a_{23}^{(0)} & a_{24}^{(0)} \\ a_{31}^{(0)} & a_{32}^{(0)} & a_{33}^{(0)} & a_{34}^{(0)} \\ a_{41}^{(0)} & a_{42}^{(0)} & a_{43}^{(0)} & a_{44}^{(0)} \end{bmatrix} \rightarrow \begin{bmatrix} b_{11} & a_{12}^{(1)} & a_{13}^{(1)} & a_{14}^{(1)} \\ v_1 & a_{22}^{(1)} & a_{23}^{(1)} & a_{24}^{(1)} \\ v_1 & a_{32}^{(1)} & a_{33}^{(1)} & a_{34}^{(1)} \\ v_1 & a_{42}^{(1)} & a_{43}^{(1)} & a_{44}^{(1)} \end{bmatrix} \rightarrow \begin{bmatrix} b_{11} & b_{12} & u_1 & u_1 \\ v_1 & a_{22}^{(2)} & a_{23}^{(2)} & a_{24}^{(2)} \\ v_1 & a_{32}^{(2)} & a_{33}^{(2)} & a_{34}^{(2)} \\ v_1 & a_{42}^{(2)} & a_{43}^{(2)} & a_{44}^{(2)} \end{bmatrix} \rightarrow \begin{bmatrix} b_{11} & b_{12} & u_1 & u_1 \\ v_1 & b_{22} & b_{23} & u_2 \\ v_1 & v_2 & b_{33} & b_{34} \\ v_1 & v_2 & v_3 & b_{44} \end{bmatrix}$$

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., with the Level-3 BLAS GEMM routine used for the trailing matrix update, and LAPACK's LABRD routine 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 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 $n \times n$. $A(i : j, m : n)$ is the submatrix of A consisting of i -th through j -th row and m -th through n -th column with 0-based indexing.

for $i \in \{1, 2, 3, \dots, n/n_b\}$ **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 LABRD, 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 factorized 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 proceed 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: $A_{i:n_b:n-1, i:n_b:n-1} \leftarrow A_{i:n_b:n-1, i:n_b:n-1} - V \times Y^T - X \times U^T$, 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:

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

According to the above equation, we derive below the maximum performance P_{max} that can be reached by the bi-diagonal reduction algorithm. In particular, for large matrix sizes n :

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

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 Figure 3a, while for `batched_dgemm` it is about 323 Gflop/s as illustrated in Figure 2a. Thus, one can expect from Equation (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., they are not necessarily stored continuously. The starting addresses 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 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 CUDA threads per matrix factorization, which is different from [13], 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.

The device function is templated with CUDA C++. 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 usually demodulize the BLAS-based structure of LAPACK algorithm. However, device functions 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. Figure 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.

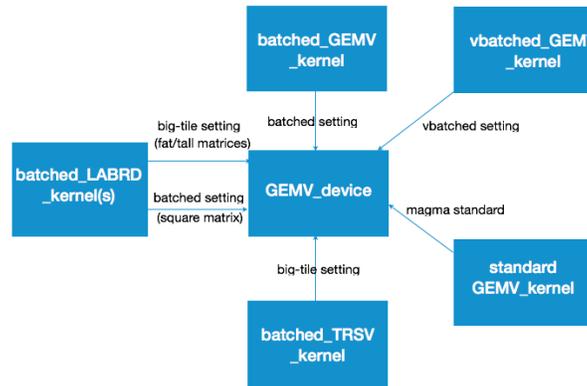


Figure 1: The same GEMV device function is called by various kernels.

6.2 Data Reuse and Degrees of Parallelism

An important optimization technique in CUDA 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).

In our design, each matrix is assigned to a thread block. The synchronization is accomplished by barriers inside the thread block. We call this setting *big-tile setting*. The naming is from this observation: if the tile is big enough (e.g., 10,000, that is well beyond the scope of a batched problem) that the whole matrix is inside the tile, it 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 simultane-

ously, 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. The current size of the shared memory is 48 KB per streaming multiprocessor (SMX) for the high-end NVIDIA K40 (Kepler) 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 [14]. We found that using *a small amount* of shared memory per kernel (less than 10KB) 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 nb needs to be increased. Yet, increasing nb 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 nb for one-sided factorizations is 32 [8] [6] [5]. However, 16 or 8 is optimal for the two-sided bi-diagonalization. A smaller nb 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 nb 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 nb 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 nb steps, these routines are called nb 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, 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 [15]. 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 [11]. 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 48KB shared memory) will be discarded. The constraints may be soft in terms of performance. We require at least 512 threads per GPU SM to ensure a reasonable occupancy. More details about tuning batched GEMM can be found in [1].

Figure 2a 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 or 30% faster than CUBLAS on average and more than $3\times$ faster than the CPU solution. Figure 2b 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.

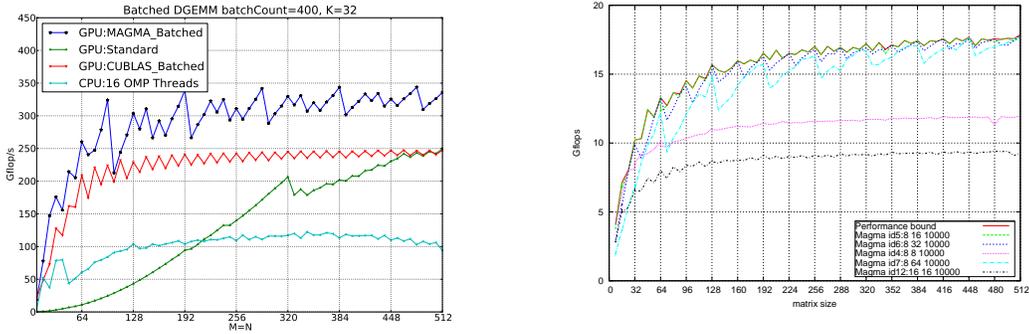
7.2 Different Batched Level 2 BLAS GEMV instances Tuning

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.



(a) Performance of our batched DGEMM ($K=32$) vs. other solutions on CPUs or GPUs. (b) Tuning results of batched DGEMV for a wide matrix where $m = 16$ and n is reported in the "x" axis.

Figure 2: Performance of the main batched BLAS routines.

8 Performance

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 Glop/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 1,430 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 [4].

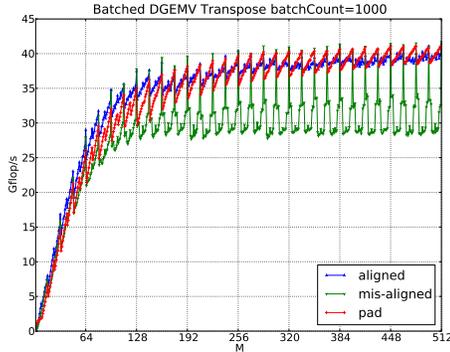
8.1 Performance study and Optimization of the bidiagonalization

Since the performance of the batched GEMV on K40c is around 40 Gflop/s (as shown in Figure 3a), the GEBRD roofline bound is 70 Gflop/s according to equation (1).

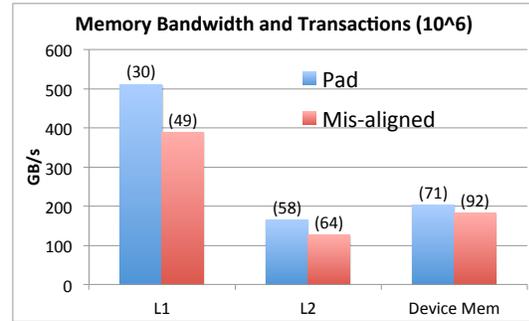
Figure 4a demonstrates the performance improvement progress of our implementation. The **non-blocked version** purely composed of Level 2 BLAS operations does not scale any more after size 256.

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 6.2, 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.

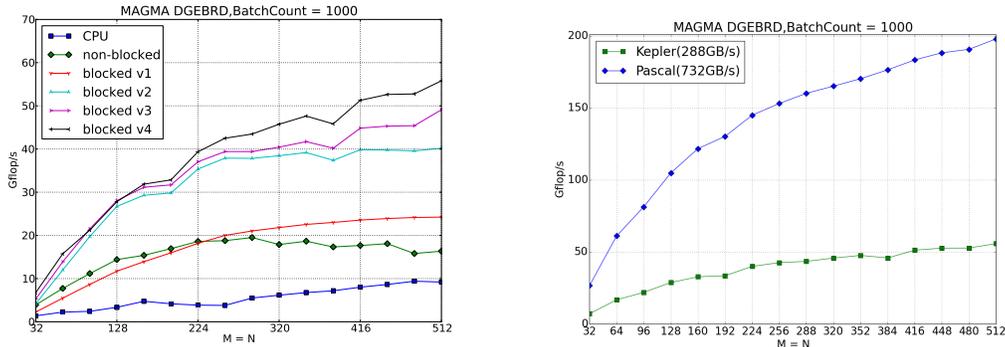


(b) Number of transactions (on top of the bar, in millions) and achieved bandwidth of the y axis.

Figure 3: 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 Figure 3a, 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 (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 Figure 3b. 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



(a) Progress of different versions on a K40c GPU.

(b) Performance on Kepler (K40c) and Pascal P100 GPU.

stage. Figure 3a show 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.

Our original development targeted Kepler GPUs. The recent Pascal GP100-PCIe GPU (16GB) has 732 GB/s bandwidth HBM2 memory, which is $2.5\times$ improvement over the Kepler K40c's 288 GB/s GDDR5 memory. Testing our memory-bound GEBRD (v4) on the Pascal GPU, even without any tuning, reveals a 3-3.5 \times speedup. This demonstrates that although we targeted a previous generation GPU, our batched approach is scalable with the future GPU architecture.

9 Conclusions and Future Work

GPU improvements have been observed extensively on large dense and sparse linear algebra problems which have more data parallelism. Small problems, taking advantage of CPU cache reuse, can be implemented relatively easily for multicore CPUs. On the other hand, the development of small problems on GPUs is not straightforward. We demonstrated that with a batched approach, small problems can have an 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 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 C++ templated. Auto-tuning is used to help find the optimal template setting for different types of kernels, like standard, batched, or different instances for a type of kernel, like transpose, wide, slim for GEMV. We demonstrate that variable sized batched BLAS can be easily extended from uniform sized problems by using device function. We achieved 56 Gflop/s, 80% of the upper bound performance for the bi-diagonalization problem on a Kepler K40c GPU. By testing on Pascal GPU, we show that our approach has very good scalability on different architectures.

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. In addition to the GEMV demonstrated here, GEHRD and SYTRD have BLAS-2 TRMV (triangular matrix-vector multiplication) and SYMV (symmetric matrix-vector multiplication) operations, respectively. The same optimization techniques of GEMV are applicable to TRMV and SYMV.

References

- [1] A. Abdelfattah, A. Haidar, S. Tomov, and J. J. Dongarra. Performance, design, and autotuning of batched GEMM for gpus. In *High Performance Computing - 31st International Conference, ISC High Performance 2016, Frankfurt, Germany, June 19-23, 2016, Proceedings*, pages 21–38, 2016.
- [2] L. Brown. Accelerate machine learning with the cudnn deep neural network library, 2015. at <http://devblogs.nvidia.com/parallelforall/accelerate-machine-learning-cudnn-deep-neural-network-library/>.
- [3] N. Corporation. <https://devtalk.nvidia.com/default/topic/527289/help-with-gpu-cholesky-factorization-/>.
- [4] T. Dong, V. Dobrev, T. Kolev, R. Rieben, S. Tomov, and J. Dongarra. A step towards energy efficient computing: Redesigning a hydrodynamic application on CPU-GPU. In *IEEE 28th International Parallel Distributed Processing Symposium (IPDPS)*, 2014.
- [5] T. Dong, A. Haidar, P. Luszczek, A. Harris, S. Tomov, and J. Dongarra. LU Factorization of Small Matrices: Accelerating Batched DGETRF on the GPU. In *16th IEEE International Conference on High Performance and Communications (HPCC 2014)*, August 2014.
- [6] T. Dong, A. Haidar, S. Tomov, and 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*, pages 432–440, 2014.
- [7] G. Golub and W. Kahan. Calculating the singular values and pseudo-inverse of a matrix. 1965.
- [8] A. Haidar, T. T. Dong, S. Tomov, P. Luszczek, and J. Dongarra. A framework for batched and gpu-resident factorization algorithms applied to block householder transformations. In *High Performance Computing - 30th International Conference, ISC High Performance 2015, Frankfurt, Germany, July 12-16, 2015, Proceedings*, pages 31–47, 2015.
- [9] O. Messer, J. Harris, S. Parete-Koon, and M. Chertkow. Multicore and accelerator development for a leadership-class stellar astrophysics code. In *Proceedings of "PARA 2012: State-of-the-Art in Scientific and Parallel Computing."*, 2012.
- [10] J. Molero, E. Garzón, I. García, E. Quintana-Ortí, and A. Plaza. Poster: A batched Cholesky solver for local RX anomaly detection on GPUs, 2013. PUMPS.
- [11] R. Nath, S. Tomov, and J. Dongarra. An improved magma gemm for fermi graphics processing units. *Int. J. High Perform. Comput. Appl.*, 24(4):511–515, Nov. 2010.
- [12] Batched svd, 2015. at <https://devtalk.nvidia.com/default/topic/851534/batched-svd-/>.
- [13] V. Oreste, M. Fatica, N. A. Gawande, and A. Tumeo. Power/performance trade-offs of small batched LU based solvers on GPUs. In *19th International Conference on Parallel Processing, Euro-Par 2013*, volume 8097 of *Lecture Notes in Computer Science*, pages 813–825, Aachen, Germany, August 26-30 2013.
- [14] B. Rymut and B. Kwolek. Real-time multiview human body tracking using gpu-accelerated pso. In *Int. Conf. on Parallel Processing and Applied Mathematics (PPAM 2013)*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, Heidelberg, 2014.
- [15] Q. Wang, X. Zhang, Y. Zhang, and 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*, pages 25:1–25:12, New York, NY, USA, 2013. ACM.