MAGMA Batched: A Batched BLAS Approach for Small Matrix Factorizations and Applications on GPUs

Tingxing Dong, Azzam Haidar, Piotr Luszczek, Stanimire Tomov, Ahmad Abdelfattah and Jack Dongarra

Abstract—A particularly challenging class of problems arising in many applications, called batched problems, involves linear algebra operations on many small-sized matrices. We proposed and designed batched BLAS (Basic Linear Algebra Subroutines), Level-2 GEMV and Level-3 GEMM, to solve them. We illustrate how to optimize batched GEMV and GEMM to assist batched advance factorization (e.g. bi-diagonalization) and other BLAS routines (e.g. forward/back substitution) to achieve optimal performance on GPUs. Our solutions achieved up to 2.8-3× speedups compared to CUBLAS and MKL solutions, wherever possible. We applied our batched methodology in a real-world Hydrodynamic application by reformulating the tensor operations into batched BLAS GEMV and GEMM operations. A 2.5× speedup and a 1.4× greenup are obtained by changing 10% of the code. We accelerated and scaled it on Titan supercomputer to 4096 nodes.

Index Terms—GPU, Batched, Bi-diagonalization, Hydrodynamic.

1 Introduction

1.1 Background and Motivations

The emergence of multicore and heterogeneous architectures requires many linear algebra algorithms to be redesigned to take advantage of the 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. The size of these matrices is usually the same, up to a few hundred. The number of them can be thousands, even millions. Billions of 8x8 and 32x32 eigenvalue problems need to be solved in magnetic resonance imaging (MRI). Hydrodynamic simulations with Finite Element Method (FEM) need to compute thousands of matrix-matrix (GEMM) and matrix-vector(GEMV) products [1]. The size of matrices increases with the order of methods, which can range from ten to a few hundred. GEMM in at the heart of deep neural network (DNN). Rather than treating convolution as a problem of one large GEMM operation, it is much more efficient to to view it as many small GEMMs [2]. In an astrophysics ODE solver [3], multiple zones are simulated, and each zone corresponds to a small linear system with an LU factorization [3]. If the matrix is symmetric and definite, the problem is reduced to a batched Cholesky factorization [4], [5].

Compared to large matrix problems with more data parallel computation that are well suited on GPUs, the challenges of small matrix problems lie in: the computing intensity (flops per memory refs) is relatively small and the sequential operations (usually in the form of BLAS Level 2 routines) are relatively big. For large enough problems, the panel factorization and associated CPU-GPU data transfers can be overlapped with the GPU work [6]. For batched small problems, the data movement via PCI-E cannot be hidden by enough computation. Their operation should be carried on GPU only.

• *Authors are with the Innovative Computing Laboratory, University of Tennessee, Knoxville, TN, 37996.*
E-mail: tdong; haidar; luszczek; tomov; ahmad; dongarra@eecs.utk.edu

1.2 Related Work

Small problems can be solved efficiently on a single CPU core, e.g., using vendor supplied libraries such as MKL [7] or ACML [8] because the CPU's memory hierarchy would favor a natural data reuse in cache. Batched factorizations then can be efficiently computed for multicore CPUs by having a single core factorize a single problem at a time as investigated in our previous paper [9].

Getting high performance batched execution on accelerators remains a challenging problem. In Villa et al.'s batched LU implementation, a single CUDA thread or a single thread block was used to solve one linear system [10] [11]. Their implementation targets problems of sizes up to 128. Their work is released in CUBLAS as the batched LU routine. Similar techniques of a warp of threads for a single factorization, were investigated by Wainwright [12] for LU with full pivoting of size to 32. Recently, Intel added their first batched GEMM routine in MKL 11.3 on Xeon Phi acclerators [13].

Batched one-sided factorizations (LU, QR, and Cholesky) were developed in our previous papers [9] [14] [15]. The one-sided factorizations are rich in compute-bound Level 3 BLAS operations, therefore the main efforts lie in enhancing Level 3 BLAS operations algorithmically. Different from previous work, here we consider two new classes of linear algebra algorithms and a real application. The first algorithm is a two-sided Householder bi-diagonalization (GEBRD). The second is the forward and backward substitution (TRSV) which is usually applied after one-sided factorizations in solving linear systems. Instead, the two problems are memory-bound algorithms rich in Level 2 BLAS GEMV

operations. In this paper, we show how our batched BLAS approach minimizes memory transactions and improves the bandwidth in the two algorithms. We compare our solution with other solutions on CPU and GPU, if possible, and apply our approach in a real application. The same approach can be applied on other two-sided factorizations: Hessenberg reduction (GEHRD) and tri-diagonalization (SYTRD), as well. Beside of GEMV, GEHRD and SYTRD are also rich in the other two BLAS-2 routines TRMV (triangular matrix-vector multiplication) and SYMV (symmetric matrix-vector multiplication), respectively. Our main efforts focus on the same-sized problems, that is, all the matrices are of identical size. Variable-sized problems are also considered, though to a lesser extent.

Our contributions can be summarized as follows.

- First, we designed batched BLAS device functions and kernels. We examined the trade-offs between data reuse and degrees of parallelism. Memory alignment and auto-tuning are adopted to optimize their performance.
- Second, we redesigned two-sided bi-diagonalization and linear system solves for batched execution on GPUs based on the batched BLAS approach. To our best knowledge, this work has not been seen before. Together with our previous paper, we formulated a batched linear algebra framework, MAGMA batched, to solve many data-parallel, small-sized problems/tasks.
- Third, we illustrated the batched methodology on real-world applications and scaled it up to 4096 nodes on the Titan supercomputer at Oak Ridge National Laboratory.
- Finally, we analyzed the power and energy consumption of the application on GPUs. By improving the bandwidth of on-chip memory with batched operations, both time and power consumption can be lowered.

2 HOUSEHOLDER BI-DIAGONALIZATION

Singular value decomposition (SVD) is used to solve underdetermined and overdetermined systems of linear equations. A high order FEM CFD simulation requires solving SVD in a batched fashion [1]. SVD reduces the matrix to bi-diagonal form in the first stage and then diagonalizes it using the QR algorithm in the second stage. Most efforts focus on the more complicated first stage, bi-diagonalization(or GEBRD for short). Previous studies show that GEBRD portion takes 90% - 99% of the time if only singular values are needed, or 30% -75% if singular vectors are additionally required [16] .

The first stage of bi-diagonalization factorizes a $M \times N$ matrix A as $A = UBV^*$, where U and V are orthogonal matrix. B is in upper diagonal form with only the diagonal and upper superdiagonal elements being non-zero. Given a vector u with unit length, the matrix $H = I - 2uu^*$ is a Householder transformation (reflection). For a given vector x, there exists a Householder transformation to zero out all but the first element of the vector x. The classic stable Golub-Kahan method applies a sequence of Householder transformations from left to right to reduce a matrix into bi-diagonal form [17]. Algorithmically, this corresponds to a sequence of in-place transformations of A as follows, whose storage is overwritten with the entries of bi-diagonal matrix B, matrix U and V, where vectors defining the left and right Householder reflectors are stored in matrix U and V, respectively.

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

$$\rightarrow \begin{bmatrix} b_{11} & b_{12} & u_1 & u_1 \\ v_1 & a_{22}^{(2)} & a_{23}^{(2)} & a_{24}^{(2)} \\ v_1 & a_{32}^{(2)} & a_{33}^{(2)} & a_{34}^{(2)} \\ v_1 & a_{42}^{(2)} & a_{43}^{(2)} & a_{44}^{(2)} \end{bmatrix} \rightarrow \begin{bmatrix} b_{11} & b_{12} & u_1 & u_1 \\ v_1 & b_{22} & a_{23}^{(3)} & a_{24}^{(3)} \\ v_1 & v_2 & a_{33}^{(3)} & a_{34}^{(3)} \\ v_1 & v_2 & a_{43}^{(3)} & a_{44}^{(3)} \end{bmatrix} \rightarrow$$

$$\rightarrow \begin{bmatrix} b_{11} & b_{12} & u_1 & u_1 \\ v_1 & b_{22} & b_{23} & u_2 \\ v_1 & v_2 & a_{33}^{(4)} & a_{34}^{(4)} \\ v_1 & v_2 & a_{43}^{(4)} & a_{44}^{(4)} \end{bmatrix} \rightarrow \begin{bmatrix} b_{11} & b_{12} & u_1 & u_1 \\ v_1 & b_{22} & b_{23} & u_2 \\ v_1 & v_2 & b_{33} & a_{34}^{(5)} \\ v_1 & v_2 & v_3 & a_{44}^{(5)} \end{bmatrix} \rightarrow$$

$$\rightarrow \begin{bmatrix} b_{11} & b_{12} & u_1 & u_1 \\ v_1 & b_{22} & b_{23} & u_2 \\ v_1 & v_2 & b_{33} & b_{34} \\ v_1 & v_2 & v_3 & b_{44} \end{bmatrix} \rightarrow \begin{bmatrix} UBV^* \end{bmatrix},$$

This algorithm is sequential and rich in Level 2 BLAS GEMV routine that is 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_i proceeds in n/nb steps of blocking size nb. One step is composed by BLAS and LAPACK routines, with LABRD for panel factorization and GEMM for trailing matrix update. The panel factorization LABRD is still sequential. The accumulated transformations are saved in matrix X and Y, respectively. Once the transformations are accumulated within the panel, they can be applied to update trailing matrix once by Level 3 BLAS operations efficiently. The blocked algorithm transforms half of the operations into Level 3 BLAS GEMM (for trailing matrix update) to make it between Level 2 and 3.

for $i \in \{1, 2, 3, \ldots, n/nb\}$ **do**
  $\{A_{ix} = A_{(i-1)\times nb:(n-1),(i-1)\times nb:i\times nb}\}$
  $\{A_{iy} = A_{(i-1)\times nb:i\times nb,(i-1)\times nb:(n-1)}\}$
  $\{C_i = A_{i\times nb:(n-1),i\times nb:(n-1)}\}$
  Panel Factorize LABRD(A_i), reduce A_ix and A_iy to bi-diagonal form, returns matrices X, Y to update trailing matrix C_i, U, V are stored in factorized A
  Trailing Matrix Update $C_i = C_i - V * Y' - X * U'$ with gemm
end for

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.

3 LINEAR SYSTEM SOLVER

Solving linear systems $Ax = b$ is a fundamental problem in linear algebra, where A is an $n \times n$ matrix, b is the input vector of size n, and x is the unknown solution vector. Solving

linear systems can fall into two broad classes of methods: direct methods and iterative methods. Iterative methods are less expensive in terms of flops but hard to converge. Preconditioning is usually required to improve convergence. Direct methods are more robust but more expensive. In this paper, we consider direct methods of the one-sided factorizations, Cholesky, LU, and the Householder QR.

Forward/backward substitution (TRSV) is used in solving linear systems, after matrix A is factorized into triangular matrices by one of the three one-sided factorizations. For example, after an LU factorization, we get $A = LU$ where L is a lower triangular matrix with all the entries above the main diagonal are zero and U is a upper triangular one. Forward substitution is used to solve $Lz = b$, where $z = (Ux)$. Backward substitution is then applied on $Ux = z$, and solution x is obtained finally. Although many dense matrix algorithms have been substantially accelerated on GPUs, mapping TRSV on GPUs is not easy due to its inherently sequential nature. In GPU computing, execution of threads should be independent as much as possible to allow parallel execution. Orders among the threads in one warp (32 threads) should be avoided since any divergence will cause serialization execution. If one thread is in the divergence branch, the other 31 threads in the same warp will be idle. Unfortunately, in TRSV, computation (and thus, threads) must be ordered because of data dependence. The following is an example of forward substitution. Backward substitution is similar.

$$
\begin{aligned}
a_{11}x_1 &= b_1 \\
a_{21}x_1 + a_{22}x_2 &= b_2 \\
a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= b_3 \\
: \quad &\quad : \\
a_{n1}x_1 + a_{n2}x_2 + ... + a_{nn}x_n &= b_n
\end{aligned}
$$

It is easy to see that x_n depends on all previous results $x_1, x_2, ..., x_{n-1}$, due to $x_n = b_n - \sum_{k=1}^{n-1}(a_{nk}x_k)$. Although the operations' order cannot be changed, the rich multiplications $(a_{nk}x_k)$ in the right hand side of the formula can be aggregated in the form of matrix-vector multiplication (GEMV) to improve the memory throughput by minimizing memory transactions in a blocked algorithm. The blocked overview of forward substitution is given in Figure 1. The original matrix is divided into triangular blocks Ti (in red) and rectangular blocks Ai (in yellow). The solution vector X is also divided into blocks Xi, where $i = 1, 2, ..., n/(jb)$ with jb the blocking size. It first sequentially computes solution X1 = $\{x_1, x_2, .., x_{(jb)}\}$, then applies a GEMV with A2$*$X1 to obtain partial results of X2 = $\{x_{(jb)+1}, x_{(jb)+2}, .., x_{2(jb)}\}$. X2 will be updated to final results in solving T2. The DAG shows the solving orders. GEMV routines are applied on the rectangular blocks Ai with all previous solution blocks X1,X2,...,Xi-1 to get partial result of Xi except the first one. The computation of GEMV is regular and there is no thread divergences. The triangular blocks are solved by the sequential algorithm to update Xi to final results. Triangular blocks Ti lie in the critical path.

Each triangular blocks Ti can be viewed as a new TRSV problem and further blocked recursively, which becomes a recursive blocked algorithm. However, the performance of TRSV will be bounded by the performance of GEMV. It is easy to see that TRSV is also a Level 2 BLAS routine like GEMV.

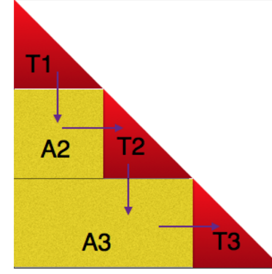

Fig. 1. Blocked overview of the forward substitution algorithm

4 Batched Methodology and Implementation for GPUs

The purpose of batched routines is to solve a set of independent problems in parallel. When one matrix is large enough to fully load the device with work, batched routines are not needed; the set of independent problems can be solved in serial as a sequence of problems. Moreover, it is preferred to solve it in serial rather than in a batched fashion, to better enforce locality of data and increase the cache reuse. However, when matrices are small (for example, matrices of size less than or equal to 512), the amount of work needed to perform the factorization cannot saturate the device, either the CPU or the GPU); thus, there is a need for batched routines.

Our batched work is part of the Matrix Algebra on GPU and Multicore Architectures (MAGMA) project, which aims to develop a dense linear algebra library similar to LAPACK but for heterogeneous architectures [18]. MAGMA has several functionalities targeting corresponding types of problems, including dense, sparse, native and hybrid. Their assumptions of problem size and hardware are different. The hybrid functionality exploits both the CPU and the GPU hardware for large problems. The native functionality only exploits the GPU for large problems. The batched functionality solving many small problems is recently integrated as **MAGMA batched**. Throughout this paper, our batched routines are named as MAGMA batched routines.

4.1 Batched BLAS Kernel Design

In a batched problem solution methodology 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.

4.1.1 Two-level Parallelism and Device-kernel Mode

Our batched BLAS kernels do not make any assumption about the layout of these matrices in memory. The batched matrices are not necessarily stored continuously in memory. The starting addresses of every matrix is stored in an array of pointers. The batched kernel takes the array of pointers as input. 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 and execute only on GPUs.

The **device function** only sees a matrix by the batched ID and thus still maintains the same interface as the standard BLAS. Different from [10] where one thread is used for one matrix factorization, each matrix problem is still parallelized by CUDA threads in our design. Therefore, our batched BLAS is characterized by two levels of parallelism. The first level is the **task-level parallelism** among the independent matrices that are simultaneously processed. The second level of **fine-grained data parallelism** is per each matrix to exploit the SIMT architecture through device functions.

The device function is templated with CUDA C++. The settings (thread blocks size, tile size, see Section 5.2) are stored in C++ template parameters. Figure 2 shows that the same GEMV device function can be called by multiple kernels, standard/batched GEMV and LABRD, TRSV kernels. (Standard GEMV targets a large matrix instead of many small ones.) We use auto-tuning techniques (see Section 5.2) to find the best setting for each type of kernel. Only one copy of device function is maintained, and optimization of the GEMV device function other than setting will propagate to upper kernels.

Multiple device functions can be called in one kernel. The usage of device functions allows multiple BLAS routines to be merged easily in one kernel without **demodulizing** the BLAS-based structure of LAPACK algorithm. Because device functions preserve the BLAS-like interface, the BLAS-based structure can be gracefully maintained. Multiple device functions can load data from the same shared memory to improve data reuse. In order to do it, we propose a big-tile setting which will be described in next section.

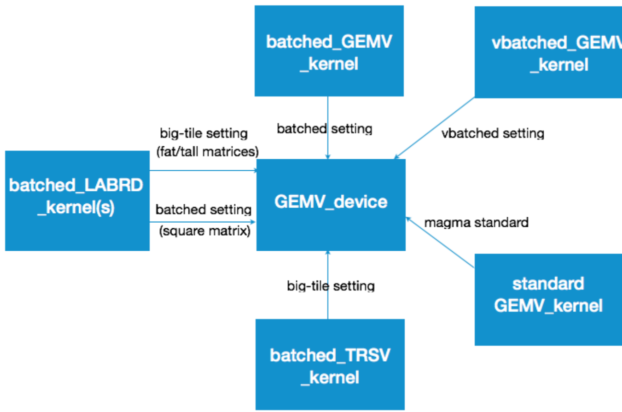

Fig. 2. The same GEMV device function is called by various kernels. The setting information is stored in C++ template parameters. vbatched is for batched matrix computation of variable sizes. See Section 5.3

4.1.2 Data Reuse and Degrees of Parallelism

As an important feature in CUDA programming, the frequent accessed data is loaded in shared memory to perform computation as much as possible before copying back to the GPU main memory. However, shared memory is private per thread block. In standard one large matrix problem, the matrix is divided into tiles with each tile loaded in shared memory. Different thread block accesses the tiles in an order determined by the algorithm. Synchronization of the computation of the tiles is accomplished by ending and relaunching kernels. When one kernel exits, the data in shared memory has to be copied back to the GPU main memory as the shared memory will be flushed. Therefore, the data dependency is resolved by synchronization of these kernels in the GPU main memory (device memory). However, in small-sized batched problems, too many kernel launchings should be avoided, especially for panel factorization where each routine has a small workload and a high probability of data reuse in shared memory.

In our batched design, each matrix is assigned with one thread block. The synchronization of all threads computation is accomplished by barriers inside per thread block. We call this setting big-tile setting. The naming is from this observation: if the tile is big enough 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 simultaneously, especially for large square matrices. Thus, overall there is a trade-off. Big-tile setting allows data to be reused through shared memory readily 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 practical tuning. Our experience shows that for the panel factorization, the big-tile setting has advantage. While for the big trailing matrix update with GEMM computation, the classic setting is preferred.

4.2 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. However, the implementation of such a technique is complicated for the small problems considered as it depends on the hardware, the precision, and the algorithm. 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. Completely saturating the shared memory per SMX can decrease the memory-bound routines' performance since only one thread-block will be mapped to that SMX at a time. Due to a limited parallelism in a small panel's factorization, 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 [20]. This process is similar to pipelining in CPU to hide the device memory access and latency.

We found that redesigning the algorithm to use *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. See Figure 3.

For good performance of Level 3 BLAS in trailing matrix update, panel width nb is increased. Yet, increasing nb in-

creases tension as the panel is a sequential operation because a larger panel width results in larger Amdahl's sequential fraction. 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, and 16 or 8 for two-sided bidiagonalization. A smaller nb for two-sided is better because the panel operations (mainly GEMV operations) in two-sided factorization are more significant than that in one-sided.

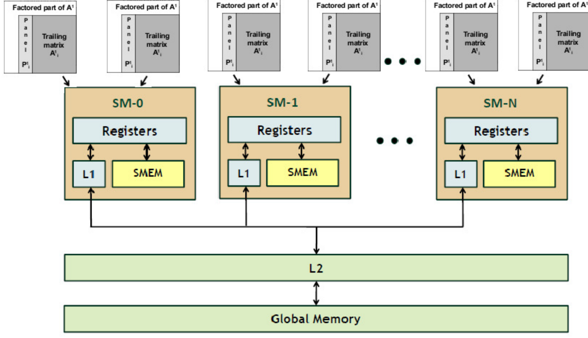

Fig. 3. Multiple factorizations reside on one streaming-multiprocessor to allow the scheduler to swap to hide the memory latency.

GEBRD panel: Provides the batched equivalent of LAPACK's LABRD routine to reduce the first nb rows and columns of a m by n matrix A to upper or lower real bidiagonal form by a Householder transformation, and returns the matrices X and Y that later are required 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 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 algorithm.

Trailing matrix updates: For Householder GEBRD, the update is achieved by two GEMM routines with the returned matrices X and Y from 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 small matrices it might be difficult to extract performance from very small Level 3 BLAS kernels.

5 AUTO-TUNING

The efforts of maximizing BLAS, especially GEMM, performance generally fall into two directions: writing assembly code and the 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 architectural specific. The vendors maintain the performance portability across different generations of their architectures [21].

Another direction is to explore the source level code auto-tuning to achieve optimal performance. Different from assembly code, source code auto-tuning relies on the compilers to allocate registers and schedule instructions. The advantage is source code is architecturally independent and is easy to maintain. Our effort focuses on source code auto-tuning.

5.1 Batched Level 3 BLAS GEMM

Linear algebra routines' performance highly reply on the Level 3 BLAS GEMM. The trailing matrix update of bidiagonalization is GEMM operations. Our batched GEMM is modified from the standard MAGMA GEMM [22]. The template parameters of our batched GEMM include the number of threads, the size of shared memory, and the data tile size. Therefore, the search space size is DIM-X * DIM-Y * BLK-M * BLK-N * BLK-K. The search space is big but 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.

Figure 4 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. In the non-batched GPU solution, it is solved by 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× faster than the CPU solution.

Note that batches of small matrices cannot achieve the same FLOPS as one large matrix. One n^2 matrix performs n^3 operations, but k^2 small $(\frac{n}{k})^2$ matrices only perform $k^2(\frac{n}{k})^3 = \frac{n^3}{k}$ operations with the same input size [23].

TABLE 1
Parameter searching space: DIM-X and DIM-Y denote the number of threads per thread block. BLK-M(N,K) denotes the data tile size. Tiling in the reduction dimension of GEMV transpose (GEMVT) and GEMV non-transpose (GEMVN) is not applicable. GEMV does not have a third dimension BLK-K.

Name	DIM-X	DIM-Y	BLK-M	BLK-N	BLK-K
GEMM	✓	✓	✓	✓	✓
GEMVN	✓	✓	✓(∞)	n/a	n/a
GEMVT	✓	✓	n/a	✓(∞)	n/a

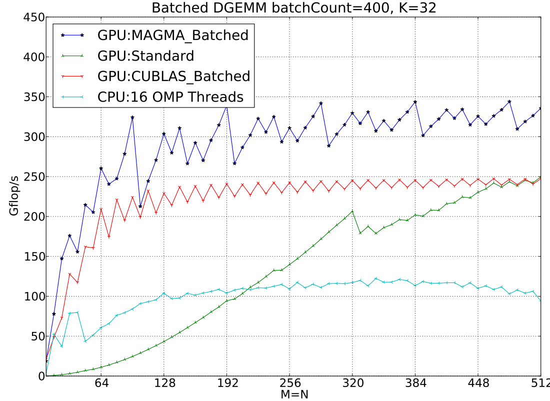

Fig. 4. Performance of our batched DGEMM (K=32) vs. other solutions on CPUs or GPUs.

5.2 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 by the warp is in a coalescing manner in GEMVN. However, in GEMV using a transpose matrix (GEMVT), the reduction has to be performed on each column. Assigning a thread to a column will make the reduction easy but 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 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.

For the GEMVN, if there is only thread in y-dimension, the result will be accumulated naturally in one thread falling back to the previous case; otherwise, a final reduction among threads in y-dimension is demanded.

The matrices can be in different shapes, like wide with row $m >> $ column n, or tall with $m <<$ column n or square with $m = n$. There are 13 GEMV instances in one step of BRD panel factorization (see Table 2). The overall BRD performance highly relies on efficient implementations and tuning of these GEMV instances. By auto-tuning, the four precisions, complex/real and double/single, are automatically tackled.

TABLE 2
Different GEMV instances needs to be optimized

Number of calls	Wide matrix	Tall matrix	Square
GEMVN	1	6	1
GEMVT	2	2	1

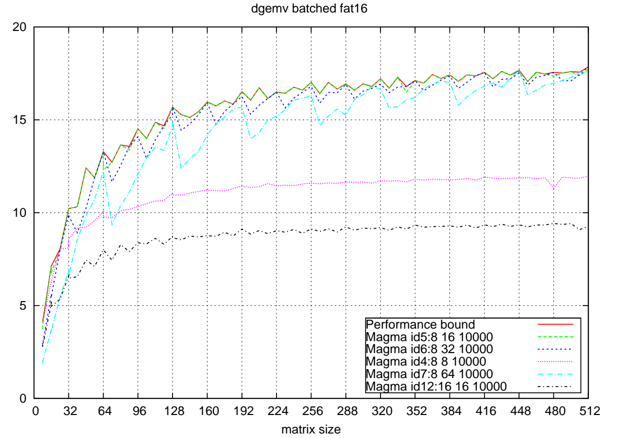

Fig. 5. Tuning results of batched DGEMVT in a wide matrix instance with row $M = 16$ but different columns. It is a big-tile setting with BLK-N as ∞, where we put 10000 as a big number.

5.3 Batched Problems of Variable Matrix Size

Applications like Geographic Information System (GIS) need to calculate a set of matrices. For each GIS object, an independent matrix is associated with it. The matrix size may be different since the geometric shape of objects varies. The batched problem with variable matrix size is another class of problems. The use of device function makes the implementation of variable batched algorithms easy in our two-level parallelism design of batched BLAS (see Section 4.1).

We consider a variable-sized batched GEMV as an example to explain our implementation. Different from uniform-sized problem, each matrix has different metadata, like sizes and leading dimension. Inside each kernel, each matrix is assigned a unique batch ID and called by a device function. Each device function only takes care of one matrix and its associated metadata.

The main challenge of variable-sized problem is that the optimal setting for one matrix size may not to be optimal for another. In CUDA, when a kernel is launched, the number of threads per thread block is fixed if without using dynamic

parallelism, indicating the same setting for every matrix. We pick up one setting optimal for the most ranges of sizes. Yet, some matrices are not running at the optimal speed, especially if the size distribution is in a worst case of random distribution. Figure 6 describes two batched problems with uniform size and random size, respectively. The matrices are square and the number of them is 1000. For uniform curve, M in x-axis denotes the matrix size, which is the same for all 1000 matrices. For random curve, M refers to the *maximum* size of the 1000 matrices. For example, $M = 256$ on the x-axis indicates 1000 random matrices with their row/column ranging from 1 to 256. The value of y-axis denotes the 1000 uniform/random size matrices' overall performance in Gflop/s. The uniform curve grows fast below size 128 and levels off in performance beyond 128. Below size 192, there is an obvious gap between the two curves since small matrices in the random problem are not running at the speed of biggest size M. Above 192, the gap becomes smaller and the random curve also levels off, as more matrices run at the speed of bigger size.

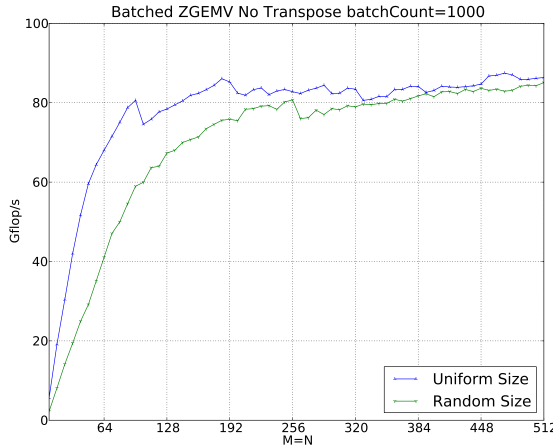

Fig. 6. Performance of batched GEMV in double complex precision with uniform size and random size, respectively.

6 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 testings, 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.

6.1 Performance of Forward/Backward Substitution

Different solutions of batched forward substitutions (solving $Ax = b$, where A is triangular, and b is a vector) in double precision (DTRSV) is shown in Figures 8. Backward substitution has the similar behavior. The solution of inverting matrix A and then solving it with a GEMV routine ($x = A^{-1}b$ [24]) proves to be the slowest because inverting matrix is expensive. An implementation using CUBLAS TRSM routine (solving $Ax = B$, where B is a matrix) is to call dtrsmBatched. By setting the number of column 1, the right-hand side matrix B is reduced to a vector, and the TRSM routine is reduced to TRSV. The performance of CUBLAS dtrsmBatched levels off at 12 Gflop/s beyond size 320. Our two implementations, one-level blocking and recursive blocking, scale with the size and reaches 30 Gflop/s and 34 Gflop/s, respectively. Recursive blocking is comparable or better than one-level blocking most of the time in performance. In the blocking algorithm, the solution vector x is loaded in shared memory. The required shared memory is proportional to the size of x. The blocked curve shakes down after size 512 because over shared memory usage decreases the occupancy of GPU SMX. The recursive algorithm blocks the shared memory usage of x to a fixed size 256. Beyond 256, x is recursively blocked and solved. It overcomes the shaky problem and continues to scale beyond size 512.

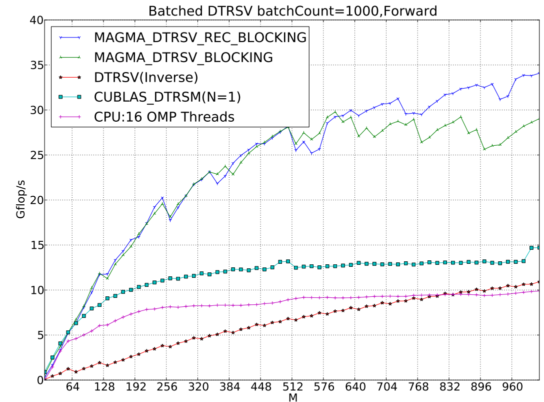

Fig. 7. Performance in Gflops/s of different solutions of batched DTRSV (forward substitution) for different matrix sizes

6.2 Performance of Bi-diagonalization

The total time of bi-diagonalization (GEBRD) includes the time spending on GEMV and GEMM. The total floating point counts of GEBRD is $8n^3/3$ [?] assuming the matrix is of size n by n. The performance of GEBRD (in flop/s) can be calculated by the following equation:

$$\frac{8n^3/3}{(GEBRD_{perf})} = \frac{4n^3/3}{(GEMV_{perf})} + \frac{4n^3/3}{(GEMM_{perf})}$$

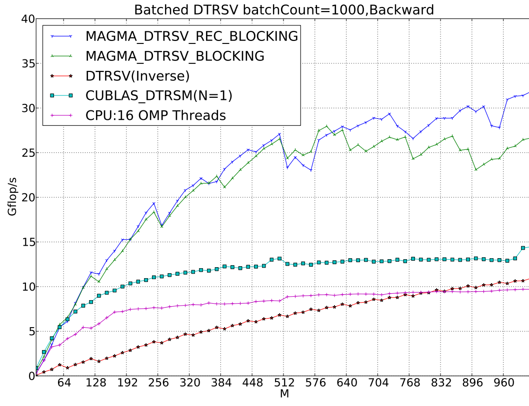

Fig. 8. Performance in Gflops/s of different solutions of batched DTRSV (backward substitution) for different matrix sizes

By reforming it, we get

$$(GEBRD_{perf}) = \frac{2 * (GEMV_{perf}) * (GEMM_{perf})}{(GEMV_{perf}) + (GEMM_{perf})}$$

From Figure 4 and 10, by averaging all size, the GEMM is 7× faster than GEMV. Therefore, we obtain

$$(GEBRD_{perf}) \simeq \frac{7(GEMV_{perf})}{4}$$

Since the performance of GEMV at size 512 on K40c is around 40 Gflop/s, the GEBRD will be bounded by 70 Gflop/s according to the equation.

Figure 12 demonstrates the performance improvement progress of our implementation. The **non-blocked version** rich in Level 2 BLAS operations does not scale any more after size 256. The first non-optimized blocked version 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 1** is even slower than the on-blocked due to the overhead. Beyond 224, it starts to grow steadily because of GEMM performance.

The main issue of the first blocked version is that GEMV routines are not optimized for all the instances in Table 2. By tuning these GEMV routines as described in Section 5.2 , the performance is doubled in **version 2**. 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 to call these GEMV device functions.

As discussed in Section 4.1.2, there is a trade-off between data reuse (with big-tile setting) and the degree of parallelism (with classic setting). As demonstrated in Figure 9, the classic setting is 5Gflop/s faster than big-tile setting for square matrix at size 512 without considering any shared memory effect. Yet, the classic setting can not fully take advantage of shared memory due to spanning thread blocks. By taking into account of shared memory, we found there is a switch over at size 128 for square matrix computation. We adopt classic setting beyond size 128 and big-tile for size less than 128. The big-tile setting is still adopted for other wide/tall instances no matter the size is 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 after we adopt this technique.

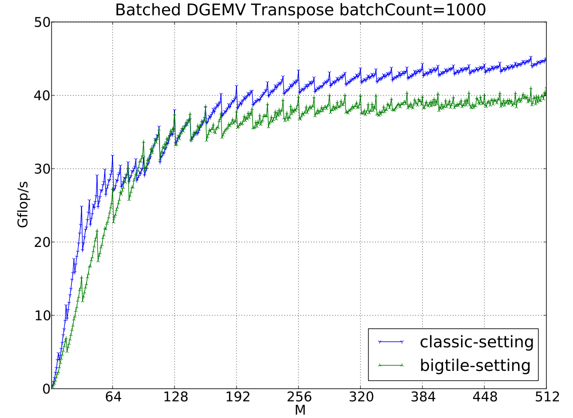

Fig. 9. Performance of batched DGEMVT kernels of square matrix with two settings. The matrices and vectors are loaded from device memory. In GEBRD, the GEMV (in the form device functions) may read/write data from shared memory in the big-tile setting. Each matrix is processed by multiple thread blocks in classic-setting but only one thread block in big-tile setting.

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 10, 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 11. Since the transactions decreases, the bandwidth is improved accordingly. By padding the corresponding elements in the multiplied vector as zeros, extra results are computed but finally discarded in the writing stage. Figure 10 show that our padding technique enables the GEMV in the GEBRD algorithm to run at a speed close to the aligned address' speed.

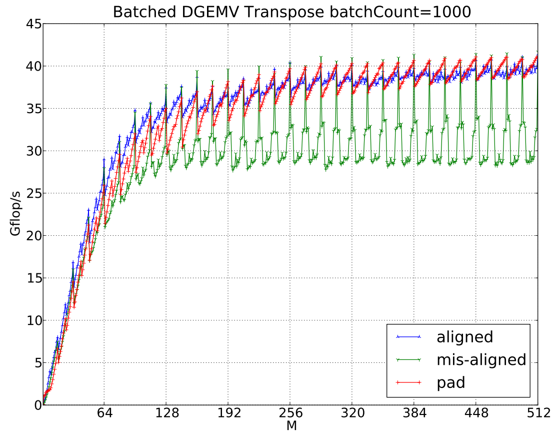

Fig. 10. Performance of batched DGEMV(transpose) in three situations: aligned, mis-aligned, and pad.

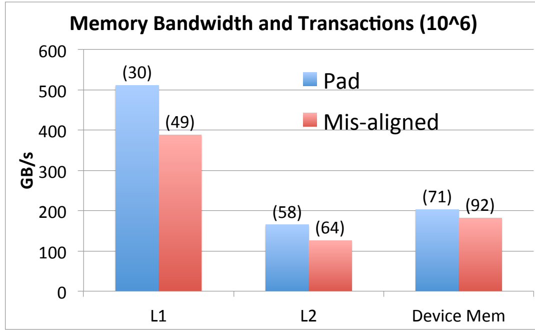

Fig. 11. Number of transactions (on top of the bar, in millions) and bandwidth of L1, L2 cache and device memory (in GB/s) before and after padding optimization. Memory transactions in L2 is low compared to other memory types because the L2 cache is designed to be shared by all SMXs; while each matrix is processed by one SMX independently, the reuse of L2 cache is very low. Data is collected by CUDA Profiler.

By padding the corresponding elements in the multiplied vector as zeros, extra results were computed but finally discarded in the writing stage. By padding, **version 4** reaches 56 Gflop/s at size 512 which is 80% of the upper bound of the performance. Overall, we find the GEMV takes 90% of the total time, GEMM takes 10% at size 512, though both take half of the floating-point operations.

7 APPLICATION

BLAST is a software package simulating hydrodynamics problems [25]. The BLAST C++ code uses high-order Finite Element Method (FEM) in a moving Lagrangian frame to solve the Euler equations of compressible hydrodynamics. It supports 2D (triangles, quads) and 3D (tets, hexes) unstructured curvilinear meshes.

On a semi-discrete level, the conservation laws of Lagrangian hydrodynamics can be written as [25]:

$$\text{Momentum Conservation:} \quad \mathbf{M}_{\mathcal{V}} \frac{\mathrm{d}\mathbf{v}}{\mathrm{d}t} = -\mathbf{F} \cdot \mathbf{1}, \tag{1}$$

$$\text{Energy Conservation:} \quad \frac{\mathrm{d}\mathbf{e}}{\mathrm{d}t} = \mathbf{M}_{\mathcal{E}}^{-1} \mathbf{F}^T \cdot \mathbf{v}, \tag{2}$$

$$\text{Equation of Motion:} \quad \frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t} = \mathbf{v}, \tag{3}$$

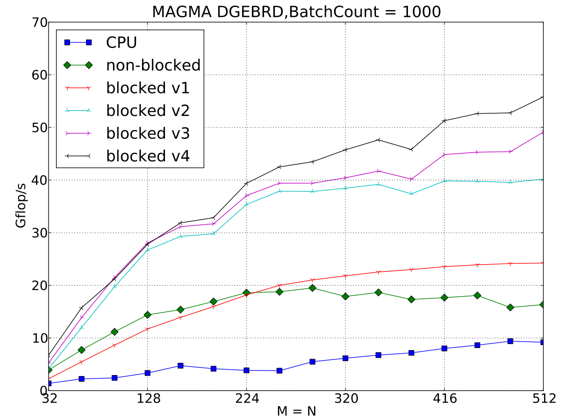

Fig. 12. Performance progresses of different versions of batched DGEBRD on a K40c GPU.

where $\mathbf{v}$, $\mathbf{e}$, and $\mathbf{x}$ are the unknown velocity, specific internal energy, and grid position, respectively. The kinematic mass matrix $\mathbf{M}_{\mathcal{V}}$ is the density weighted inner product of *continuous* kinematic basis functions and is therefore global, symmetric, and sparse.

We solve the linear system of (1) by using a preconditioned conjugate gradient (PCG) iterative method at each time step. We solve the linear system of (2) by precomputing the inverse of each local dense matrix at the beginning of a simulation and applying it at each time step using sparse linear algebra routines.

$\mathbf{F}$, called the generalized *force*, depends on the hydrodynamic state $(\mathbf{v}, \mathbf{e}, \mathbf{x})$, and needs to be evaluated at every time step. $\mathbf{F}$ is a tensor of rank-3 and can be assembled from the generalized *corner force matrices* $\{\mathbf{F}_z\}$ computed in every zone (or element) of the computational mesh. Evaluating $\mathbf{F}_z$ is a locally FLOP-intensive process based on transforming each zone back to the reference element where we apply a quadrature rule with points $\{\hat{q}_k\}$ and weights $\{\alpha_k\}$:

$$(\mathbf{F}_z)_{ij} = \int_{\Omega_z(t)} (\sigma : \nabla \vec{w}_i)\, \phi_j$$

$$\approx \sum_k \alpha_k \hat{\sigma}(\hat{\vec{q}}_k) : J_z^{-1}(\hat{\vec{q}}_k) \hat{\nabla} \hat{\vec{w}}_i(\hat{\vec{q}}_k)\, \hat{\phi}_j(\hat{\vec{q}}_k) |J_z(\hat{\vec{q}}_k)|. \tag{4}$$

where, J_z is the Jacobian matrix, and the hat symbol indicates the quantity is on the reference zone. Each zone computes a component of the corner forces associated with it independently. A local corner force matrix $\mathbf{F}_z$ can be written in a compact GEMM form

$$\mathbf{F}_z = \mathbf{A}_z \mathbf{B}^{\mathbf{T}},$$

with

$$(\mathbf{A}_z)_{ik} = \alpha_k \hat{\sigma}(\hat{\vec{q}}_k) : J_z^{-1}(\hat{\vec{q}}_k) \hat{\nabla} \hat{\vec{w}}_i(\hat{\vec{q}}_k)\, |J_z(\hat{\vec{q}}_k)|, \tag{5}$$

and

$$(\mathbf{B})_{jk} = \hat{\phi}_j(\hat{\vec{q}}_k)\,. \tag{6}$$

Therefore, in the CPU code, $\mathbf{F}$ is constructed by a loop over zones (for each z) in the domain with each zone is associated with a GEMM problem for $\mathbf{F}_z$.

Finite element zones are defined by a parametric mapping Φ_z from a reference zone. The Jacobian matrix J_z is non-singular. Its determinant $|J_z|$ represents the local volume. The stress tensor $\hat{\sigma}(\vec{\hat{q}}_k)$ requires evaluation at each time step and is rich in FLOPs at each quadrature point (see [25] for more details).

A finite element solution is specified by the order of the kinematic and thermodynamic bases, Q_k-Q_{k-1} with $k \geq 1$. By increasing the order of the finite element method, k, we can arbitrarily increase the floating point intensity of the corner force kernel of (4) as well as the overall algorithm of (1) - (3). The corner force computation only takes 10% amount of the code but takes more than 60% of the running time.

7.1 CUDA Implementation

Our implementation has two layers of parallelism: (1) MPI-based parallel domain-partitioning and communication between CPUs; (2) CUDA based parallel corner force calculation on GPUs inside each MPI task.

We redesigned the corner force CPU code into CUDA code. The key observation is the rich rank-3 tensor operations in the right hand side can be translated into batched matrix operations. A tensor is a 3-dimensional array. A matrix stores two dimension (not necessary the first two dimensions), while the batch number (not necessary the third dimension in the array) can be viewed the last dimension. For example, $\mathbf{F}$ can be viewed as batched $\mathbf{F}_z$. The number of batches is number of zones. Therefore, in the GPU code, $\mathbf{F}$ can be computed by a batched GEMM given $\mathbf{A}_z$ and $\mathbf{B}$ are ready.

We identify three important properties of batched problem. First, to allow parallel execution, these matrices must be independent. Otherwise, they still have to be solved with a loop. Second, the size of them should be small. If the matrix is big enough to saturate the GPU, the batched launching will fall back to sequential launching (see Figure 4, the cuBLAS batched GEMM merges with standard GEMM at size 512). Third, the batch number should be large to allow a higher degrees of parallelism to hide the memory latency.

The challenge is to recognize batched matrices to satisfy these properties in a 3-dimension tensor in $\mathbf{A}_z$ (see Eq 5). For example, a batched GEMM is involved in computing $\hat{\sigma}(\vec{\hat{q}}_k)$. The number of batch is number of points which is much bigger than zones. Moreover, there are different choices to organize the matrix products because the matrix multiplications are associative, which further complicates the implementation. For example, $A*B*C$ can be organized as either $A*(B*C)$ or $(A*B)*C$. Although they are the same mathematically, they might be drastically different in performance. In our optimal implementation, we designed six batched kernels **1-6**. Kernel 4-8 are characterized by batched DGEMM or DGEMV routines as shown in Table 3 . Depending on the order of the methods, most matrices are with size ranging from 2 to 126.

In our CUDA code development, we also developed a base implementation. The base implementation is a simple kernel only exploring the task level parallelism (each matrix problem is considered as a task) but each matrix problem is still sequentially solved by one thread. In addition to the task level parallelism, the optimal solution explore the fine-grained data parallelism per matrix with multi-threading. The related optimization techniques of configuring thread blocks and shared memory are similar to those in Section 5.1 and 5.2.

TABLE 3
Optimal Implementations of the BLAST code on GPUs. Kernel 9 is a set of kernels instead of one single kernel.

No.	Kernel Type	Purpose
1	custom kernel	SVD,Eigen,Ajugate
2	custom kernel	EoS, $\hat{\sigma}(\vec{\hat{q}}_k)$
3	batched GEMM	$\hat{\nabla}\vec{\hat{w}}_i(\vec{\hat{q}}_k)$, $\mathbf{J}_z(\vec{\hat{q}}_k)$
4	batched GEMM	$\hat{\sigma}(\vec{\hat{q}}_k)$
5	batched GEMM	Auxiliary
6	batched GEMM	Auxiliary
7	batched GEMM	$\mathbf{A}_z\mathbf{B}^\mathbf{T}$
8	batched GEMVN	$-\mathbf{F}\cdot\mathbf{1}$
10	batched GEMVT	$\mathbf{F}^T\cdot\mathbf{v}$
9	CUDA_PCG	Solve linear system(1)
11	SpMV	Solve linear system(2)

We profile the bandwidth of the base and optimized kernels on a K20c GPU. Figure 13 shows the bandwidth of all the three level memory, from on-chip memory L1/Shared, off-chip L2 to device memory. All the optimized kernels exceeded the base implementation in bandwidth of L1/Shared and device memory except kernel 3 which instead achieved very high bandwidth in L1/shared memory. The four optimized batched kernels (kernel 3-6) achieved much higher bandwidth in L1/Shared memory as they exploited shared memory a lot. Because on-chip memory is much faster than off-chip memory, the bandwidth of on-chip memory is more critical to performance. Again, batched kernels do not achieve high bandwidth in L2 cache compared to non-batched Kernel 1 and 2 as discussed in Section **??**.

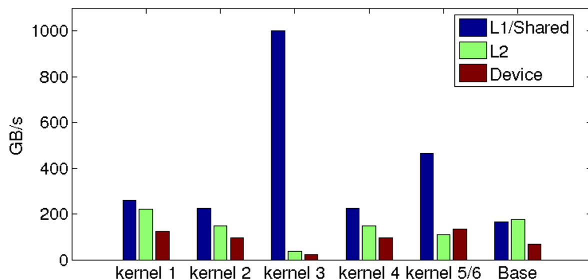

Fig. 13. Memory bandwidth of base and optimized kernels. The theoretical peak bandwidth of device memory of K20 is 208GB/s.

Kernel 8 and 10 are batched DGEMV. An implementation involving CUBLAS is to put `cublasDgemv` in streams, as recommended in the User Guide [23] since there is no batched DGEMV routine in CUBLAS. However, the performance is very poor, as shown in Table 4. Our kernel is 90x faster than that of `cublasDgemv`, achieving 50 % of theoretical peak performance of batched DGEMV on C2050. The theoretical peak performance of DGEMV is one fourth of bandwidth because it reads a 64-bit (8-byte) element and only performs two operations (one multiplication and one adding).

We also developed a custom conjugate gradient solver (kernel 9) to solve Equation (1) as $\{\mathbf{M}_\mathcal{V}\}$ is a sparse matrix. It is with a diagonal preconditioner (PCG) [26]. It is constructed with CUSPARSE SpMV and cublasDdot [27]. The CUDA-PCG solver is outside of corner force. Kernel 11 is a sparse matrix-vector multiplication routine in CUSPARSE.

TABLE 4
Custom kernel 8 and streamed `cublasDgemv` implement batched DGEMV on one C2050. In this test case, each small matrix is 81 by 8 and each vector is 8. The number of matrices is 4096.

	streamed `cublasDgemv`	kernel 8	theoretical
Gflop/s	0.2	18	35.5

7.2 Single Node Performance and Scalability

In our test, the CPU is a Intel 8-core Sandy Bridge E5-2670 and the GPU is a K20c GPU. We consider a Q_2-Q_1 and a Q_4-Q_3 method. Only corner force component is accelerated on GPU (noted as core speedup). The other parts are still performed on the CPU. Table **??** shows the speedup achieved by the CPU-GPU over the CPU only. The core speedups are $5\times$ and $3\times$ for the Q_2-Q_1 and Q_4-Q_3 method, respectively. However, the overall speedups are 1.9× and 2.5×, because the corner force take a more significant portion in Q_4-Q_3 methods.

We tested our code on the ORNL Titan supercomputer, which has 16 AMD CPU cores and one K20m GPU per node. We scaled it up to 4096 computing nodes. Eight nodes is the base line. For a 3D problem, one more refinement level increases the domain size $8\times$. We achieved weak scaling by fixing the domain size 512 for each computing node and increasing $8\times$ more nodes for every refinement step. From 8 nodes to 512 nodes, the curve is almost flat in Figure 14. From 512 nodes to 4096 nodes, 5-cycle time increases from 1.05 to 1.83 seconds. The limiting factor is the MPI global reduction to find the minimum time step after the corner force computation and MPI communication in MFEM (Step 5 in Section 7).

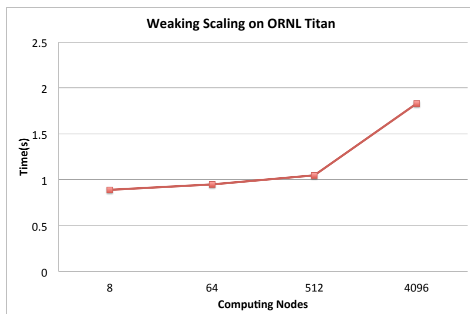

Fig. 14. Weak scaling of the BLAST code on the Titan supercomputer. The time is of 5 cycles of steps.

8 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.

For batched linear algebra problems, we designed batched BLAS CUDA kernels. We propose one-matrix-visible device functions as the underlying components of batched kernels. The use of device functions allows the data to be easily reused through shared memory, which is critical to panel factorizations performance in advance routines, but without demodulizing the BLAS-based structure. The device functions are CUDA C++ templated. Auto-tuning is used to help find the optimal setting for different types of kernels. Since device function only sees one matrix, the variable sized batched problem is easily extended from uniform sized problems.

We consider a batched two-sided bi-diagonalization and a batched triangular solve problem based on the batched BLAS approach. They are optimized for coalescing and alignment to improve the GPU memory throughput. Other solutions of batched triangular solves are also examined. Our best implementation achieves $3\times$ speedups compared to optimal MKL implementations on two Intel Sandy Bridge CPU. Compared with NVIDIA CUBLAS routines, our triangular solve achieves up to $2.8\times$ speedups. For a memory-bound Householder bi-diagonalization, we achieve 56Gflop/s, 80% of the theoretical performance bounded by GEMV on a K40c GPU. Our methodology applies to other two-sided factorizations as well, for example, Hessenberg reduction using Level 2 BLAS GEMV and tri-diagonalization using Level 2 BLAS SYMV.

Furthermore, we redesigned a real world hydrodynamic application with the batched methodology onto CPU-GPU systems. The tensor operations are translated into batched BLAS GEMV and GEMM operations. By changing less 10% of the code, we see $1.9\times$ to $2.5\times$ speedups compared to CPU only code. Compared to a base implementation, our optimal implementation is twice faster and lowers the power by 10% by efficiently utilizing the on-ship GPU memory. Weak scaling is achieved up to 4096 computing nodes on the ORNL Titan Supercomputer.

We released and maintained this new functionality through the MAGMA batched on NVIDIA GPU accelerators. The batched is a GPU-only implementation rather than hybrid solution. When there is only a single matrix size, our batched solution is reduced to a GPU native solution. The native solution can have a performance advantage over hybrid one where the host CPU is much weaker than the accelerator. In mobile devices featuring ARM CPUs with discrete GPUs, the total GPU implementations have significant advantages in both energy consumption and performance [33].

Acknowledgments

This work is supported by the National Science Foundation, the Department of Energy, and NVIDIA.

References

[1] T. Dong, V. Dobrev, T. Kolev, R. Rieben, S. Tomov, and J. Dongarra, "A step towards energy efficient computing: Redesigning a hy-

drodynamic application on CPU-GPU," in *IEEE 28th International Parallel Distributed Processing Symposium (IPDPS)*, 2014.

[2] "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] 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.

[4] 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.

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

[6] S. Tomov, R. Nath, and J. Dongarra, "Dense linear algebra solvers for multicore with GPU accelerators," in *Proc. of the IEEE IPDPS'10*, Atlanta, GA, April 19-23 2014.

[7] "Intel Math Kernel Library," 2014, available at http://software.intel.com/intel-mkl/.

[8] "ACML - AMD Core Math Library," 2014, available at http://developer.amd.com/tools-and-sdks/cpu-development/amd-core-math-library-acml.

[9] 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*, 2015, pp. 31–47.

[10] 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*, ser. Lecture Notes in Computer Science, vol. 8097, Aachen, Germany, August 26-30 2013, pp. 813–825.

[11] V. Oreste, N. A. Gawande, and A. Tumeo, "Accelerating subsurface transport simulation on heterogeneous clusters," in *IEEE International Conference on Cluster Computing (CLUSTER 2013)*, Indianapolis, Indiana, September, 23-27 2013.

[12] I. Wainwright, "Optimized LU-decomposition with full pivot for small batched matrices," April, 2013, gTC'13 – ID S3069. [Online]. Available: http://on-demand.gputechconf.com/gtc/2013/presentations/S3069-LU-Decomposition-Small-Batched-Matrices.pdf

[13] "Introducing batch gemm operations," 2015, at https://software.intel.com/en-us/articles/introducing-batch-gemm-operations.

[14] 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*, 2014, pp. 432–440.

[15] 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.

[16] H. Ltaief, P. Luszczek, and J. J. Dongarra, "High performance bidiagonal reduction using tile algorithms on homogeneous multicore architectures," *ACM Transactions on Mathematical Software*, vol. 39, no. 3, pp. 16:1–16:22, May 2013. [Online]. Available: http://dx.doi.org/10.1145/2450153.2450154

[17] G. Golub and W. Kahan, "Calculating the singular values and pseudo-inverse of a matrix," 1965. [Online]. Available: http://www.jstor.org/stable/2949777

[18] "Matrix algebra on GPU and multicore architectures (MAGMA)," 2014, available at http://icl.cs.utk.edu/magma/.

[19] "Available at http://developer.download.nvidia.com/assets/cuda/files/CUDADownloads/TechBrief-Dynamic-Parallelism-in-CUDA.pdf," 2014.

[20] 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)*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer-Verlag, 2014.

[21] 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*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 25:1–25:12. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503219

[22] R. Nath, S. Tomov, and J. Dongarra, "An improved magma gemm for fermi graphics processing units," *Int. J. High Perform. Comput. Appl.*, vol. 24, no. 4, pp. 511–515, Nov. 2010. [Online]. Available: http://dx.doi.org/10.1177/1094342010385729

[23] "CUBLAS," 2015, at http://docs.nvidia.com/cuda/cublas/.

[24] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, "From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming," *Parallel Comput.*, vol. 38, no. 8, pp. 391–407, Aug. 2012. [Online]. Available: http://dx.doi.org/10.1016/j.parco.2011.10.002

[25] V. Dobrev, T. V. Kolev, and R. N. Rieben, "High-order curvilinear finite element methods for lagrangian hydrodynamics," *SIAM J. Scientific Computing*, vol. 34, no. 5, 2012. [Online]. Available: http://dx.doi.org/10.1137/120864672

[26] M. Naumov, "Incomplete-lu and cholesky preconditioned iterative methods using cusparse and cublas," 2011.

[27] "Cusparse," 2014, at http://docs.nvidia.com/cuda/cusparse/.

[28] "Intel® 64 and IA-32 architectures software developer's manual," July 20 2014, available at http://download.intel.com/products/processor/manual/.

[29] E. Rotem, A. Naveh, D. Rajwan, A. Ananthakrishnan, and E. Weissmann, "Power-management architecture of the intel microarchitecture code-named sandy bridge," *IEEE Micro*, vol. 32, no. 2, pp. 20–27, March/April 2012, iSSN: 0272-1732, http://dx.doi.org/10.1109/MM.2012.1210.1109/MM.2012.12.

[30] "Available at https://developer.nvidia.com/nvidia-management-library-nvml," 2014.

[31] "Cuda programming guide v5.0," http://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[32] J. Choi and R. W. Vuduc, "How much (execution) time and energy does my algorithm cost?" *ACM Crossroads*, vol. 19, no. 3, pp. 49–51, 2013. [Online]. Available: http://doi.acm.org/10.1145/2425676.2425691

[33] A. Haidar, S. Tomov, P. Luszczek, and J. Dongarra, "Magma embedded: Towards a dense linear algebra library for energy efficient extreme computing," *2015 IEEE High Performance Extreme Computing Conference (HPEC 2015)*, 2015.

Tingxing Dong Tingxing Dong obtained a PhD degree in computer science at the Innovative Computing Laboratory (ICL) at the University of Tennessee, Knoxville (UTK) in 2015. He received a master degree in computer science from university of Chinese academy of science in 2010. His research interests include linear algebra problems and computational fluid dynamics implementations on hybrid architectures. He is now a Senior Software Engineer in AMD, Austin, TX.

Azzam Haidar received a Ph.D. in 2008 from CERFACS, France. He is Research Scientist at the Innovative Computing Laboratory (ICL) at the University of Tennessee, Knoxville (UTK). His research interests focus on the development and implementation of parallel linear algebra routines for scalable multi-core architectures, for largescale dense and sparse problems, as well as approaches that combine direct and iterative algorithms to solve large linear systems as well as eigenvalue problems.

Piotr Luszczek is a Research Director at the University of Tennessee. His research interests are in large-scale parallel algorithms, numerical analysis, and highperformance computing. He has been involved in the development and maintenance of widely used software libraries for numerical linear algebra. In addition, he specializes in computer benchmarking of supercomputers using codes based on linear algebra, signal processing and PDE solvers.

Stanimire Tomov received a Ph.D. in Mathematics from Texas A M University in 2002. He is a Research Director in ICL and Adjunct Assistant Professor in the EECS at UTK. His research interests are in parallel algorithms, numerical analysis, and high-performance scientific computing (HPC). Currently, his work is concentrated on the development of numerical linear algebra software for emerging architectures for HPC.

Jack Dongarra holds appointments at the University of Tennessee, Oak Ridge National Laboratory, and the University of Manchester. He specializes in numerical algorithms in linear algebra, parallel computing, use of advanced computer architectures, programming methodology and tools for parallel computers. His contributions to the HPC field have received numerous recognitions including the IEEE Sid Fernbach Award (2004), the first IEEE Medal of Excellence in Scalable Computing (2008), the first SIAM Special Interest Group on Supercomputing?s award for Career Achievement (2010) and the IEEE IPDPS 2011 Charles Babbage Award. He is a fellow of the AAAS, ACM, IEEE and SIAM and a member of the National Academy of Engineering.