12-2015

Batched Linear Algebra Problems on GPU Accelerators

Tingxing Dong
University of Tennessee - Knoxville, tdong@vols.utk.edu

To the Graduate Council:

I am submitting herewith a dissertation written by Tingxing Dong entitled "Batched Linear Algebra Problems on GPU Accelerators." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

Jack Dongarra, Major Professor

We have read this dissertation and recommend its acceptance:

Jian Huang, Gregory Peterson, Shih-Lung Shaw

Accepted for the Council:

Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

Batched Linear Algebra Problems on GPU Accelerators

A Dissertation Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Tingxing Dong

December 2015

For my family: my parents, brother, sister and my beloved nephew Luke

Acknowledgements

I would like to thank the University of Tennessee for letting me study and live here for five years. I have many beautiful memories left here. Knoxville is like my second hometown.

I would like to express my gratitude towards my advisor Jack Dongarra for providing me an incredible opportunity to do research in ICL and his supervision over my PhD career. I would also like to express my appreciation to Stanimire Tomov, Azzam Haidar, Piotr Luszczek for their guidance and advice, and being a source of motivation. Thank Mark Gate, Ichitaro Yamazaki and other people in ICL for their helpful discussion. Thank Tracy Rafferty and Teresa Finchum for their help in processing my paper work. ICL is a big family and like my home. I am alway proud to be an ICLer in my life.

I am grateful to my committee, Professor Jian Huang, Professor Gregory Peterson and Professor Shih-Lung Shaw for their valuable feedbacks during the writing of the dissertation.

I would also thank my friends in UT campus Chinese Bible study class. We had a wonderful fellowship every Friday night. I also thank Kai Wang for his effective proofreading of my dissertation.

Thank my friends for being there for me.

Love bears all things

Abstract

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.

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 low computing intensity, the large sequential operation fractions, and the big PCI-E overhead. These challenges entail redesigning the algorithms instead of merely porting the current LAPACK algorithms.

We consider two classes of problems. The first is linear systems with one-sided factorizations (LU, QR, and Cholesky) and their solver, forward and backward substitution. The second is a two-sided Householder bi-diagonalization. They are challenging to develop and are highly demanded in applications. Our main efforts focus on the same-sized problems. Variable-sized problems are also considered, though to a lesser extent.

Our contributions can be summarized as follows. First, we formulated a batched linear algebra framework to solve many data-parallel, small-sized problems/tasks. Second, we redesigned a set of fundamental linear algebra algorithms for high-performance, batched execution on GPU accelerators. Third, we designed batched

BLAS (Basic Linear Algebra Subprograms) and proposed innovative optimization techniques for high-performance computation. Fourth, we illustrated the batched methodology on real-world applications as in the case of scaling a CFD application up to 4096 nodes on the Titan supercomputer at Oak Ridge National Laboratory (ORNL). Finally, we demonstrated the power, energy and time efficiency of using accelerators as compared to CPUs. Our solutions achieved large speedups and high energy efficiency compared to related routines in CUBLAS on NVIDIA GPUs and MKL on Intel Sandy-Bridge multicore CPUs.

The modern accelerators are all Single-Instruction Multiple-Thread (SIMT) architectures. Our solutions and methods are based on NVIDIA GPUs and can be extended to other accelerators, such as the Intel Xeon Phi and AMD GPUs based on OpenCL.

Table of Contents

Chapter 1

Introduction

1.1 Background and Motivations

Solving many small linear algebra problems is called batched problem, which consists of a large number of independent matrices (e.g., from hundreds to millions) to be solved, where the size of each matrix is considered small. Various scientific applications require solvers that work on batched problems. For example, in magnetic resonance imaging (MRI), billions of 8x8 and 32x32 eigenvalue problems need to be solved. Also, a batched 200x200 QR decomposition is required to be computed in radar signal processing [5]. Hydrodynamic simulations with Finite Element Method (FEM) need to compute thousands of matrix-matrix (GEMM) and matrix-vector(GEMV) products [13]. The size of matrices increases with the order of methods, which can range from ten to a few hundred. As shown in Figure 1.1, high-order methods result in large-sized problems but can reveal more refined physical details. As another example, consider an astrophysics ODE solver with Newton-Raphson iterations [28]. Multiple zones are simulated in one MPI task, and each zone corresponds to a small linear system with each one resulting in multiple sequential solving with an LU factorization [28]. The typical matrix size is 150x150. If the matrix is symmetric and definite, the problem is reduced to a batched Cholesky

factorization, which is widely used in computer vision and anomaly detection in images [29, 10].

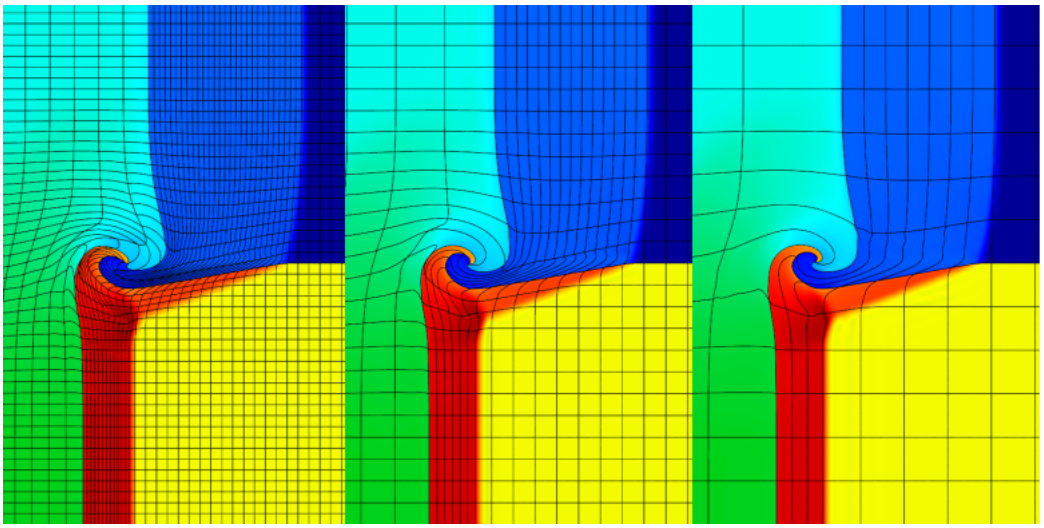

Figure 1.1: From left to right: shock triple-point problems using FEM with Q8Q7, Q4Q3, Q2Q1 methods, respectively.

High performance computing (HPC) is increasingly becoming power and energy constrained. The average power of TOP 10 supercomputers climbed from 3.2MW in 2010 to 6.6MW in 2013, which is enough to power a small town[43]. Department of Energy has set a goal of 50MW for Exascale systems, which require one watt to yield 20 GFLOPS. Limited by the power budget, more and more computing systems seek to install accelerators, such as GPUs, due to their high floating-point operation capability and energy efficiency advantage over CPUs, as shown in Figure 1.2. The co-processor accelerated computing has become a mainstream movement in HPC. This trend is indicated in the ranking of the TOP 500 and the Green 500. In the June 2013 TOP 500 ranking, 51 supercomputers are powered by GPUs[43]. Although accelerated systems make up only 10% of the systems, they accomplish 33% of the computing power. In the June 2013 Green 500 ranking, the most power efficient system accelerated by K20 GPUs surpassed 3 GFLOPS per watt, up from 2 GFLOPS per watt in the June 2012 ranking[18].

The vast difference between the computing capability of CPUs and GPUs (shown in Figure 1.3) is due to their architecture design. For CPUs, more transistors are used for caches or control units while they are devoted to arithmetic units for GPUs, as depicted in Figure 1.4. Different from CPUs, GPUs cannot run operating systems but are designed for compute-intensive, highly parallel computation purpose. Compared to CPUs, GPUs have limited cache size and cache level; therefore DRAM' latency is relatively high. Rather than caching data, GPUs launch thousands or even millions of light-weight threads for computation to hide the memory access latency.

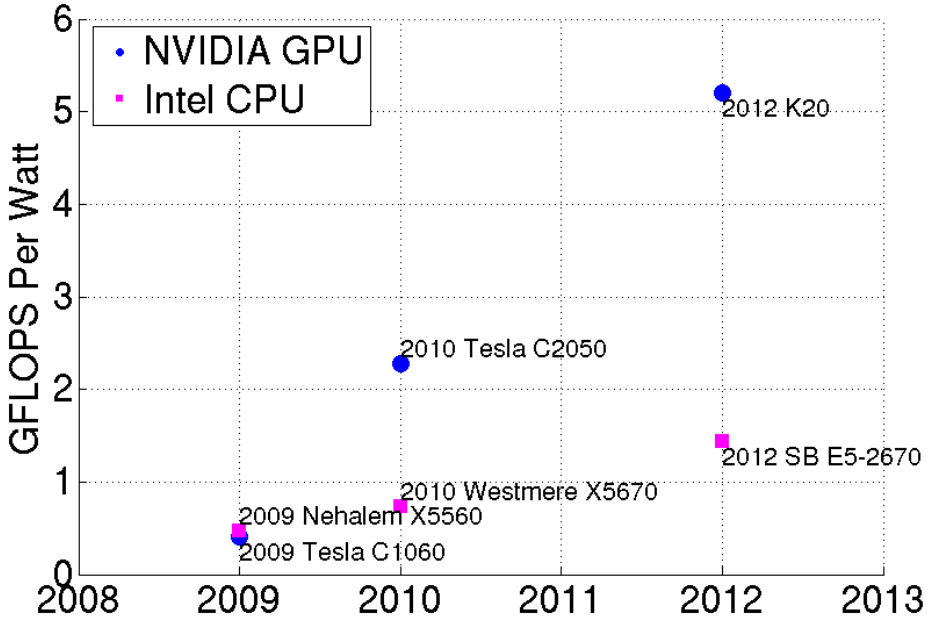

Figure 1.2: GFLOPS per watt of NVIDIA GPUs and Intel CPUs in double precision.

The development of CPUs, as noted in Sections 1.2 and 3.1, can be done easily using existing software infrastructure. On the other hand, GPUs, due to their SIMD design, are efficient for large data parallel computation; therefore, they have often been used in combination with CPUs, which handle the small and difficult to parallelize tasks. Although tons of linear algebra libraries are on CPUs, the lack of linear algebra software for small problems is especially noticeable for GPUs. The need to overcome the challenges of solving small problems on GPUs is also

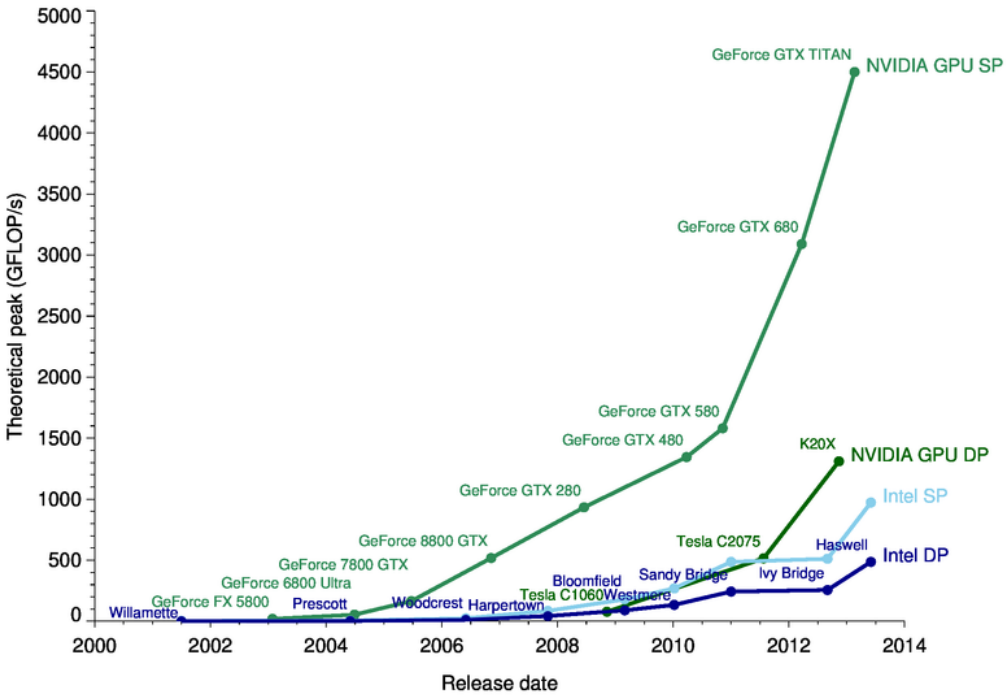

Figure 1.3: Single precision (SP) and double precision (DP) computing capability of NVIDIA GPUs and Intel CPUs [31].

related to the GPU's energy efficiency, often four to five times better than that of multicore CPUs. To take advantage of GPUs, code ported on GPUs must exhibit high efficiency. Thus, one of the main goals of this work is to develop GPU algorithms and their implementations on small problems to outperform multicore CPUs in raw performance and energy efficiency. In particular, we target three one-sided factorizations (LU, QR, and Cholesky) and one two-sided factorizations bi-diagonalization for a set of small dense matrices.

1.2 Related Work

The questions are what programming and execution model is best for small problems, how to offload work to GPUs, and what should interact with CPUs, if anything. The offload-based execution model and the accompanying terms, *host* and *device*,

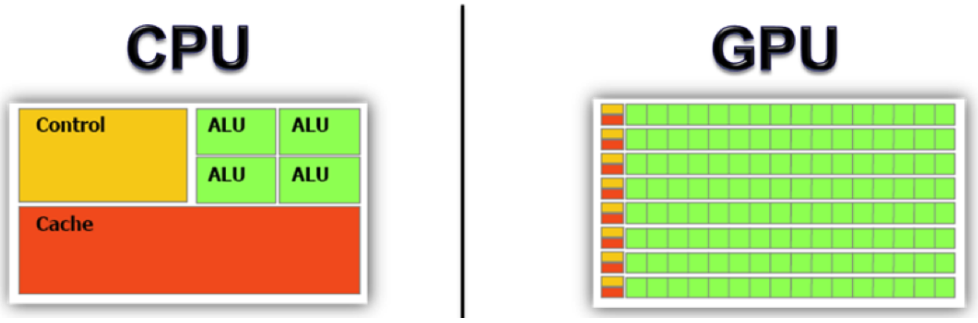

Figure 1.4: Differences between GPU and CPU.

have been established by the directive-based programming standards: OpenACC [35] and OpenMP [36]. While these specifications are *host-centric*, in the context of dense linear algebra computations, we recognize three different modes of operation: hybrid, native, and batched execution. The first employs both the host CPU and the device accelerator, be it a GPU or an Intel coprocessor, which cooperatively execute on a particular algorithm. The second offloads the execution completely to the accelerator. The third is the focus of this dissertation and involves execution of many small problems on the accelerator while the host CPU only sends the input data and receives the computed result in a pipeline fashion to alleviate the dearth of PCI-E bandwidth and long latency of the transfers.

Small problems can be solved efficiently on a single CPU core, e.g., using vendor supplied libraries such as MKL [23] or ACML [2] because the CPU's memory hierarchy would back a "natural" data reuse (small enough problems can fit into small, fast memory). Besides memory reuse, to further speed up the computation, vectorization to use SIMD processor supplementary instructions can be added either explicitly as in the Intel Small Matrix Library [22] or implicitly through the vectorization in BLAS. Batched factorizations then can be efficiently computed for multicore CPUs by having a single core factorize a single problem at a time (see Section 3.1). However, the energy consumption is higher than the GPU-based factorizations.

For GPU architectures, prior work has been concentrated on achieving high-performance for large problems through hybrid algorithms [42]. Motivations come

from the fact that the GPU's compute power cannot be used on panel factorizations as efficiently as on trailing matrix updates [44]. Because the panel factorization is considered a latency-bound workload, which faces a number of inefficiencies on throughput-oriented GPUs, it is preferred to be performed on the CPU. As a result, various hybrid algorithms are developed in which panels are factorized on the CPU while the GPU is used for trailing matrix updates (mostly GEMMs) [1, 14]. Note that a panel's data transfer to and from the CPU is required at each step of the loop. For large enough problems, the panel factorization and associated CPU-GPU data transfers can be overlapped with the GPU work. For small problems, however, this application is not possible; and our experience has shown that hybrid algorithms would not be as efficient as they are for large problems.

Most batched work on GPUs comes from NVIDIA and their collaborators. Villa et al. [37], [38] obtained good results for batched LU developed entirely for GPU execution, where a single CUDA thread, or a single thread block, was used to solve one linear system at a time. Their implementation targets very small problems (of sizes up to 128). Their work is released in CUBLAS as the batched LU routine. Similar techniques, including the use of a warp of threads for a single factorization, were investigated by Wainwright [45] for LU with full pivoting on matrices of size up to 32. Although the problems considered were often small enough to fit in the GPU's shared memory (e.g., 48 KB on a K40 GPU), and thus to benefit from data reuse (n^2 data for $\frac{2}{3}n^3$ flops for LU), the performance of these approaches was up to about 20 Gflop/s in double precision and did not exceed the maximum performance due to memory bound limitations (e.g., 46 Gflop/s on a K40 GPU for DGEMV's $2n^2$ flops on n^2 data; see Table A.1).

In version 4.1 released in January 2012, NVIDIA CUBLAS added a batched GEMM routine. In v5.0 released in October 2012, CUBLAS added a batched LU and a batched TRSM routine with the dimension of the matrix limited to 32x32. Version 5.5 removed the dimension limit but still restricted the matrix on square ones. The latest v6.5 included a batched QR routine. In the latest MKL v11.3

released in May 2015, Intel added its first batched routine - GEMM on the Xeon Phi accelerator.

To our best knowledge, implementations of batched two-sided bi-diagonalization factorizations, either on CPUs or GPUs, have not been reported. Here we review the related algorithms. A two-sided matrix bi-diagonalization for multicore CPU based on tile algorithms was studied in [26]. Their implementation was based on Householder reflectors. Ralha proposed a one-sided bi-diagonalization algorithm[39], which implicitly tridiagonalized the matrix A^TA with a one-sided orthogonal transformation of A . This approach suffers from numerical stability issues, and the resulting matrix may lose its orthogonality properties. Ralha's approach was improved by Barlow et al. to enhance the stability by merging the two distinct steps to compute the bidiagonal matrix B [6] . In our batched implementation, we adopt Householder reflectors to perform the orthogonal transformation to guarantee the numerical stability. It is different from less computational expensive but less stable transformations, for example, the Gaussian elimination.

Chapter 2

Algorithms for Related Linear Algebra Problems

2.1 One-sided Factorizations

In this section, we present a brief overview of the linear algebra algorithms for the development of either Cholesky, Gauss, or the Householder QR factorizations based on block outer-product updates of the trailing matrix. Conceptually, one-sided factorization maps a matrix A into a product of matrices X and Y:

$$\mathcal{F}: \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \mapsto \begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix} \times \begin{bmatrix} Y_{11} & Y_{12} \\ Y_{21} & Y_{22} \end{bmatrix}.$$

Algorithmically, this corresponds to a sequence of in-place transformations of A, whose storage is overwritten with the entries of matrices X and Y (P_{ij} indicates currently factorized panels):

$$\begin{bmatrix} A_{11}^{(0)} & A_{12}^{(0)} & A_{13}^{(0)} \\ A_{21}^{(0)} & A_{22}^{(0)} & A_{23}^{(0)} \\ A_{31}^{(0)} & A_{32}^{(0)} & A_{33}^{(0)} \end{bmatrix} \rightarrow \begin{bmatrix} P_{11} & A_{12}^{(0)} & A_{13}^{(0)} \\ P_{21} & A_{22}^{(0)} & A_{23}^{(0)} \\ P_{31} & A_{32}^{(0)} & A_{33}^{(0)} \end{bmatrix} \rightarrow$$

$$\rightarrow \begin{bmatrix} XY_{11} & Y_{12} & Y_{13} \\ X_{21} & A_{22}^{(1)} & A_{23}^{(1)} \\ X_{31} & A_{32}^{(1)} & A_{33}^{(1)} \end{bmatrix} \rightarrow \begin{bmatrix} XY_{11} & Y_{12} & Y_{13} \\ X_{21} & P_{22} & A_{23}^{(1)} \\ X_{31} & P_{32} & A_{33}^{(1)} \end{bmatrix} \rightarrow$$

$$\rightarrow \begin{bmatrix} XY_{11} & Y_{12} & Y_{13} \\ X_{21} & XY_{22} & Y_{23} \\ X_{31} & X_{32} & A_{33}^{(2)} \end{bmatrix} \rightarrow \begin{bmatrix} XY_{11} & Y_{12} & Y_{13} \\ X_{21} & X_{22} & Y_{23} \\ X_{31} & X_{32} & P_{33} \end{bmatrix} \rightarrow$$

$$\rightarrow \begin{bmatrix} XY_{11} & Y_{12} & Y_{13} \\ X_{21} & XY_{22} & Y_{23} \\ X_{31} & X_{32} & XY_{33} \end{bmatrix} \rightarrow \begin{bmatrix} XY \end{bmatrix},$$

where XY_{ij} is a compact representation of both X_{ij} and Y_{ij} in the space originally occupied by A_{ij}.

Table 2.1: Panel factorization and trailing matrix update routines. x represents the precision, which can be single (S), double (D), single complex (C) or double complex (Z).

	Cholesky	Householder	Gauss
PanelFactorize	xPOTF2 xTRSM	xGEQF2	xGETF2
TrailingMatrixUpdate	xSYRK2 xGEMM	xLARFB	xLASWP xTRSM xGEMM

There are two distinct phases in each step of the transformation from $[A]$ to $[XY]$: *panel factorization* (P) and trailing matrix update $A^{(i)} \rightarrow A^{(i+1)}$. Implementation of these two phases leads to a straightforward iterative scheme as shown in Algorithm 1. The panel factorization is accomplished by a non-blocked routine. Table 2.1 shows the BLAS and the LAPACK routines that should be substituted for the generic routines named in the algorithm.

Algorithm 1 is called blocked algorithm since every panel P is of size nb which allows the trailing matrix update to use the Level 3 BLAS routines. Note that if

$nb = 1$ the algorithm falls back to the standard non-blocked algorithm introduced by LINPACK in the 1980s.

for $P_i \in \{P_1, P_2, \ldots, P_n\}$ **do**
  PanelFactorize(P_i)
  TrailingMatrixUpdate($A^{(i)}$)
end

Algorithm 1: Two-phase implementation of a one-sided factorization.

We use a Cholesky factorization (POTF2) of a symmetric positive definite matrix to illustrate the Level 2 BLAS-based non-blocked algorithm, as outlined Figure 2.1. Due to the symmetry, the matrix can be factorized either as an upper triangular matrix or as a lower triangular matrix (e.g., only the shaded data is accessed if the lower side is to be factorized). Given a matrix A of size $n \times n$, there are n steps. Steps go from the upper-left corner to lower-right corner along the diagonal. At step j, the column vector $A(j : n, j)$ is to be updated. First, a dot product of the row vector $A(j, 0 : j)$ is needed to update the element $A(j, j)$(in black). Then the column vector $A(j+1 : n-1, j)$ (in red) is updated by a GEMV $A(j+1 : n-1, 0 : j-1) \times A(j, 0 : j-1)$ followed by a scaling operation. This non-blocked Cholesky factorization involves two Level 1 BLAS routines (DOT and SCAL) and a Level 2 BLAS routine GEMV. Since there are n steps, these routines are called n times; thus, one can expect that POTF2's performance will depend on Level 1 and Level 2 BLAS operations' performance. Hence, it is a slow memory-bound algorithm. The non-blocked algorithm of LU and householder QR can be found in LAPACK [4].

The floating-point operation counts and elements access of related Level 2 and 3 BLAS and one-sided factorization LAPACK routines are shown in Table A.1. Level 1 and Level 2 BLAS operations (e.g., GEMV) are memory bound since they have much lower flops per element compared to Level 3 BLAS (e.g.,GEMM). Note that both blocked and non-blocked algorithms inherently have the same floating-point operations. The difference is that the blocked algorithm explores the Level 3 BLAS

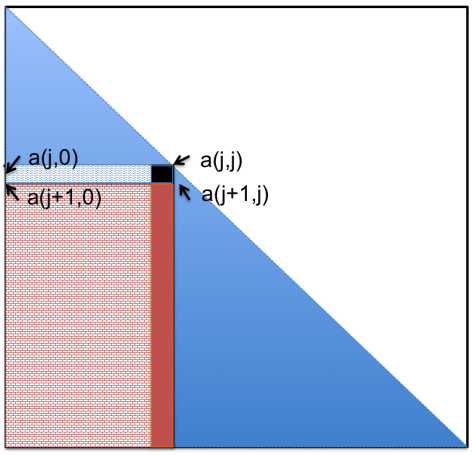

Figure 2.1: Non-blocked Cholesky factorization

by reducing the amount of Level 2 BLAS operations and achieves higher efficiency through the data and cache reuse [16].

The classical hybrid implementation as described in Algorithm 1 lacks efficiency because either the CPU or the GPU is working at a time and a data transfer to and from the CPU is required at each step. The MAGMA library further modified the algorithm to overcome this issue and to achieve closer-to-optimal performance. In fact, the ratio of the computational capability between the CPU and the GPU is orders of magnitude; thus, the common technique to alleviate this imbalance and keep the GPU loaded is to use lookahead.

for $P_i \in \{P_1, P_2, \ldots, P_n\}$ **do**
  CPU: PanelFactorize(P_i)
  GPU: TrailingMatrixUpdate of only next panel of ($A^{(i)}$ which is P_2)
  CPU and GPU work in parallel: CPU go to the next loop while GPU continue the update
  GPU: continue the TrailingMatrixUpdate of the remaining ($A^{(i-1)}$) using the previous panel (P_{i-1})
end

Algorithm 2: Lookahead of depth 1 for the two-phase factorization.

Algorithm 2 shows a very simple case of lookahead of depth 1. The update operation is split into an update of the next panel and an update of the rest of

the trailing matrix. The splitting is done to overlap the communication and the panel factorization with the update operation. This technique lets us hide the panel factorization' memory-bound operations and keep the GPU loaded by the trailing matrix update.

In the batched implementation, however, we cannot afford such a memory transfer between CPU and GPU at any step since the trailing matrix is small and the amount of computation is not sufficient to overlap it in time with the panel factorization. Many small data transfers will take away any performance advantage enjoyed by the GPU. In the next Chapter 3, we describe our proposed implementation and optimization of the batched algorithm.

The performance is recognized as Gflop/s across the dissertation. The floating-point counts of related BLAS and LAPACK routines are demonstrated in Table A.1 [8]. Asymptotically, the lower-degree terms (< 3) of the flops can be omitted if the size is big enough.

2.2 Forward/Backward Substitution

Solving linear systems $Ax = b$ is a fundamental problem in linear algebra, where A is a $n \times n$ 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 our implementation, we consider direct methods.

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. 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 CUDA, 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. Equation 2.2 is an example of forward substitution. The following solution depends all previous solutions. Therefore, the degree of parallelism in TRSV is limited. Although the operations' order cannot be changed, the sequential operations can be aggregated to improve the memory throughput by minimizing memory transactions in the blocked algorithm.

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

To solve it,

- Step 1: $x_1 = b_1/a_{11}$
- Step 2: $x_2 = (b_2 - a_{21} * x_1)/a_{22}$, x_2 depends on x_1
- Step 3: $x_3 = (b_3 - a_{31} * x_1 - a_{32} * x_2)/a_{33}$, x_3 depends on x_1 and x_2
- Step 4: $x_4 = (b_4 - a_{41} * x_1 - a_{42} * x_2 - a_{43} * x_3)/a_{44}$, x_4 dpends on x_1 to x_3
- Step n: x_n depends on all previous results $x_1, x_2, ..., x_{n-1}$

A blocked algorithm first sequentially computes $x_1, x_2, .., x_{nb}$ (nb is the blocking size), then applies a matrix-vector multiplication (GEMV) to obtain partial results of $x_{nb+1}, x_{nb+2}, .., x_{2nb}$. In the above example, after x_1 and x_2 are sequentially computed in Step 1 and 2, $a_{31} * x_1 - a_{32} * x_2$ and $a_{41} * x_1 - a_{42} * x_2$ in Step 3 and 4, can be done by one GEMV routine to get partial results of x_3 and x_4. x_3 and x_4 will be updated

to final ones in the next sequential solving. In GEMV, the computation is regular and there is no thread divergences.

The blocked algorithm overview is given in Figure 2.2. We use forward substitution as an example. 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 $i = 1, 2, ..., n/nb$. The triangular blocks are solved by the sequential algorithm. GEMV routines are applied on the rectangular blocks. The computation flow goes as follows. First, triangular block T1 is solved to get X1. A GEMV routine performs A2 * X1 to get the partial result of X2 which will be updated to the final result in the next T2 solving. After T2, another GEMV routine will take A3 and X1, X2 to get the partial result of X3 which will be updated in T3 solving. Iteratively, all the blocks Xi are solved. Backward substitution is in a reverse order. Each triangular matrix Ti can be further blocked recursively, which becomes a recursive blocked algorithm. The performance of TRSV is bounded by the performance of GEMV on blocks Ai and triangular blocks Ti that are in the critical path. It is easy to see that TRSV is a Level 2 BLAS routine. Its floating-point operation count is shown in Table A.1.

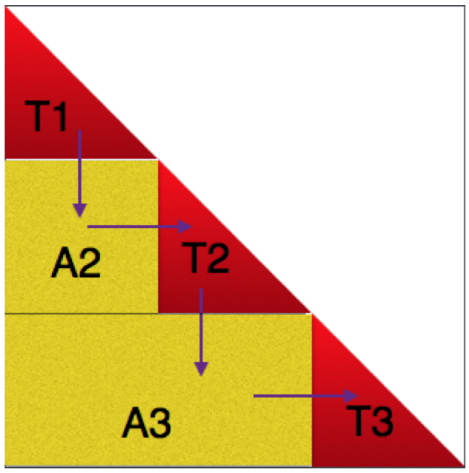

Figure 2.2: Overview of the blocked algorithm for forward substitution

2.3 Householder Bi-diagonalization

Two-sided factorizations, like the singular value decomposition (SVD) factorize a $M \times N$ matrix A as $A = UWV^*$, where U is an orthogonal $M \times M$ matrix and V is an orthogonal $N \times N$ matrix. The diagonal elements of matrix W are non-negative numbers in descending order and all off-diagonal elements are zeros. The first $min(m, n)$ columns of U and V are the left and right singular vectors of A. SVD is used to solve underdetermined and overdetermined systems of linear equations. It is also used to determine the rank, range and null space of a matrix. It is extensively used in signal processing and statistics. A high order FEM CFD simulation requires solving SVD in a batched fashion[13].

The singular decomposition algorithm 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 BRD for short). Previous studies show that BRD portion takes 90% - 99% of the time if only singular values are needed, or 30% -75% if singular vectors are additionally required [26] .

The first stage of bi-diagonalization factorizes a $M \times N$ matrix A as

$A = UBV^*$, where U and V are orthogonal matrices. 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 (GEBRD) applies a sequence of Householder transformations from left to right to reduce a matrix into bi-diagonal form [17]. See Figure 2.3. In the left update of each step, a column vector is annihilated with Householder transformation and then the Householder reflector is applied to update the remaining matrix. Vectors defining the left Householder reflectors are stored as columns of matrix U. In the right update, a row vector is annihilated and again applied to

update. The vectors defining the right Householder reflectors are stored in matrix V. 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 sequential algorithm can be blocked to aggregate the transformations to delay the update to improve the efficiency. The blocked algorithm is divided into two distinct phases: panel factorization and update of trailing matrix, as shown in Figure 2.4. The blocked two-phase algorithm is described in Algorithm 3. 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 saved left and right Householder reflectors are saved in matrix A in replace of annihilated elements. 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 total operations of GEBRD is $8n^3/3$, if we consider the square matrix size as n for simplicity. The sequential algorithm is rich in Level 1 and 2 BLAS operations. The blocked algorithm transforms half of the operations into Level 3 BLAS GEMM (for trailing matrix update) to make it overall similar to Level 2.5 BLAS. The other half is still Level 1 and 2 BLAS operations. Because Level 3 BLAS is much faster than Level 2 BLAS, the Level 2 BLAS in the panel factorization is the bottleneck. The peak performance of GEBRD is up to two times that of Level 2 BLAS GEMV as discussed Section 4.2.3.

for $i \in \{1, 2, 3, \ldots, n/nb\}$ **do**
  $\{A_i = A_{(i-1)\times nb:n,(i-1)\times nb:n}\}$
  $\{C_i = A_{i\times nb:n,i\times nb:n}\}$
  Panel Factorize LABRD(A_i), reduce A_i 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 3: Two-phase implementation of the Householder BRD algorithm.

$$U_1^*A = \begin{bmatrix} \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{x} \\ \mathbf{0} & \mathbf{x} & \mathbf{x} & \mathbf{x} \\ \mathbf{0} & \mathbf{x} & \mathbf{x} & \mathbf{x} \\ \mathbf{0} & \mathbf{x} & \mathbf{x} & \mathbf{x} \\ \mathbf{0} & \mathbf{x} & \mathbf{x} & \mathbf{x} \end{bmatrix} \longrightarrow U_1^*AV_1 = \begin{bmatrix} x & \mathbf{x} & \mathbf{0} & \mathbf{0} \\ 0 & \mathbf{x} & \mathbf{x} & \mathbf{x} \\ 0 & \mathbf{x} & \mathbf{x} & \mathbf{x} \\ 0 & \mathbf{x} & \mathbf{x} & \mathbf{x} \\ 0 & \mathbf{x} & \mathbf{x} & \mathbf{x} \end{bmatrix}$$

$$\longrightarrow U_2^*U_1^*AV_1 = \begin{bmatrix} x & x & 0 & 0 \\ 0 & \mathbf{x} & \mathbf{x} & \mathbf{x} \\ 0 & \mathbf{0} & \mathbf{x} & \mathbf{x} \\ 0 & \mathbf{0} & \mathbf{x} & \mathbf{x} \\ 0 & \mathbf{0} & \mathbf{x} & \mathbf{x} \end{bmatrix} \longrightarrow U_2^*U_1^*AV_1V_2 = \begin{bmatrix} x & x & 0 & 0 \\ 0 & x & \mathbf{x} & \mathbf{0} \\ 0 & 0 & \mathbf{x} & \mathbf{x} \\ 0 & 0 & \mathbf{x} & \mathbf{x} \\ 0 & 0 & \mathbf{x} & \mathbf{x} \end{bmatrix}$$

$$\longrightarrow U_3^*U_2^*U_1^*AV_1V_2 = \begin{bmatrix} x & x & 0 & 0 \\ 0 & x & x & 0 \\ 0 & 0 & \mathbf{x} & \mathbf{x} \\ 0 & 0 & \mathbf{0} & \mathbf{x} \\ 0 & 0 & \mathbf{0} & \mathbf{x} \end{bmatrix} \longrightarrow U_4^*U_3^*U_2^*U_1^*AV_1V_2 = \begin{bmatrix} x & x & 0 & 0 \\ 0 & x & x & 0 \\ 0 & 0 & x & x \\ 0 & 0 & 0 & \mathbf{x} \\ 0 & 0 & 0 & \mathbf{0} \end{bmatrix} = B.$$

Figure 2.3: A sequence of Householder transformations reduces the matrix into bi-diagonal form in the sequential algorithm.

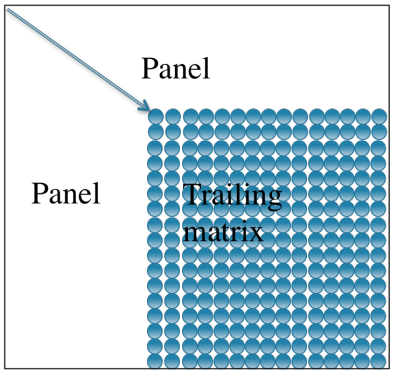

Figure 2.4: The matrix is divided into panel and trailing matrix in blocked bi-diagonalization algorithm.

Chapter 3

Methodology and Implementation

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.

3.1 Batched Design for Multicore CPUs

In broad terms, batched factorization on multicore CPUs can be approached in two main ways. The first is to parallelize each small factorization across all the cores, and the second is to execute each factorization sequentially on a single core with all the cores working independently on their own input data. With these two extremes clearly delineated, it is easy to see the third possibility: the in-between solution where each matrix is partitioned among a handful of cores, and multiple matrices are worked on at a time as the total number of available cores permits.

We tested various levels of nested parallelism to exhaust all possbilites of optimization available on CPUs. The two extremes mentioned above get about 40 Gflop/s (one outer task and all 16 cores working on a single problem at a time – 16-way parallelism for each matrix) and 100 Gflop/s (16 outer tasks with only a single core per task – sequential execution each matrix), respectively. The scenarios between these extremes achieve somewhere in between in terms of performance. For example, with eight outer tasks with two cores per task, we achieve about 50 Gflop/s. Given these results and to increase the presentation's clarity, we only report the extreme setups in the results shown below.

3.2 Batched Methodology and Implementation for GPUs

3.2.1 MAGMA

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, starting with current Multicore+GPU systems [21]. To address the complex challenges of the emerging hybrid environments, optimal software solutions will have to hybridize, combining the strengths of different algorithms within a single framework. Building on this idea, MAGMA aims to design linear algebra algorithms and frameworks for hybrid many-core and GPU systems that can enable applications to fully exploit the power that each of the hybrid components offers.

MAGMA is an open-sourced project. The latest release in May 2015 is v1.6.2. CUBLAS is the NVIDIA vendor CUDA library on GPUs. LAPACK is the Fortran library on CPUs. MAGMA calls some of CUBLAS and LAPACK routines but includes more advanced routines. MAGMA has several functionalities targeting corresponding types of problems, including dense, sparse, native and hybrid, as

shown in Figure 3.1. 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 in MAGMA. Throughout this dissertation, our batched routines are named as MAGMA batched routines. For example, our batched GEMM routine is referred to as MAGMA batched GEMM.

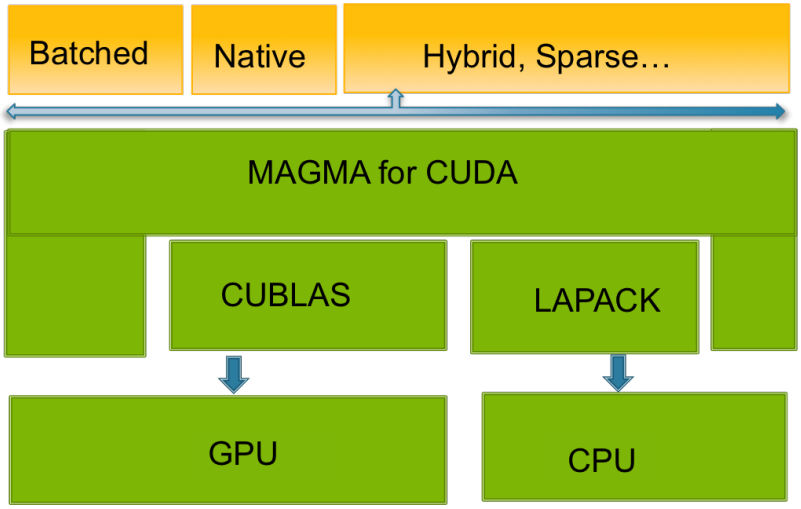

Figure 3.1: MAGMA software stack

3.2.2 Batched BLAS Kernel Design

Our batched routines are based on batched BLAS, the way they are implemented and all the relevant optimizations that have been incorporated to achieve performance. All routines are batched and denoted by the corresponding LAPACK routine names. We have implemented them in the four standard floating-point precisions – single real, double real, single complex, and double complex. For convenience, we use the double precision routine name throughout this study.

In a batched problem solution methodology that is based on batched BLAS, many small dense matrices must be factorized simultaneously (as illustrated in Figure 3.2), meaning that all the matrices will be processed simultaneously by the same kernel.

The batched kernel does 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 classic BLAS. Therefore, our batched BLAS is characterized by two levels of parallelism. The first level is the task-level parallelism among matrices. The second level of fine-grained data parallelism is inside each matrix through device functions to exploit the SIMT architecture.

The device function is templated with C++. The settings (like, the thread blocks size, tile size) are stored in C++ template parameters. In order to find the optimal setting for each type of problems, we adopt an auto-tuning technique, which will be discussed in Section 3.3.1 and 3.3.2.

Trade-offs between Data Reuse and Degrees of Parallelism

Shared memory is fast on-chip memory. The frequent accessed data of the matrix is loaded in shared memory before copying back to the main memory. However, shared memory can not live across multiple kernels and span thread blocks. When one kernel exits, the data in shared memory has to be copied back to the GPU main memory since the shared memory will be flushed. Therefore, many kernel launchings not only introduce launching overhead but potentially result in data movement, because the data has to be read again from GPU main memory in the next kernel, causing redundant memory access.

Besides, shared memory is private per thread block. In standard large-sized problems, the matrix is divided into tiles with each tile loaded in shared memory. Different thread blocks access the tiles in an order determined by the algorithm. Synchronization of the computation of the tiles is accomplished by finishing the current kernel and relaunching another in the GPU main 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. Therefore, in our design, each matrix is assigned with one thread block. The synchronization is accomplished in shared memory and by barriers inside the thread block. We call this setting big-tile setting. The naming is based 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 may have a higher degree of parallelism as different parts of the matrix are processed simultaneously, especially for matrices of big size. Thus, overall there is a trade-off between them. 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 and matrix size, and is usually selected by practical tuning.

Multiple device functions can reuse the same shared memory as long as they are called in the same kernel. This design, device functions instead of kernels serve as the basic component, allows the computation of BLAS routines to be merged easily in one kernel and takes advantage of shared memory. Merging codes usually demodulize the BLAS-based structure of LAPACK algorithm. However, since device functions preserve the BLAS-like interface, the BLAS-based structure can be gracefully maintained.

3.2.3 Implementation of One-sided Factorizations and Bi-diagonalization on GPUs

Algorithmically, one approach to the batched factorization problems for GPUs is to consider that the matrices are small enough, and, therefore factorize them using the non-blocked algorithm. The implementation is simple but the performance obtained turns out to be unacceptably low. Thus, the implementation of the batched factorization must also be blocked and thus must follow the same iterative scheme (*panel factorization* and *trailing matrix update*) shown in Algorithm 1. Note that the trailing matrix update consists of Level 3 BLAS operations (HERK for Cholesky, GEMM for LU and LARFB for QR) which are compute intensive and thus can perform very well on the GPU. Therefore, the most difficult phase of the algorithm is the panel factorization.

Figure 3.2 is a schematic view of the batched problem considered. Basic block algorithms, as the ones in LAPACK [4], factorize at step i a block of columns, denoted by panel P_i, followed by the application of the transformations accumulated in the panel factorization to the trailing sub-matrix A_i.

A recommended way of writing efficient GPU kernels is to use the GPU's shared memory – load it with data and reuse that data in computations as much as possible. The idea behind this technique is to perform the maximum amount of computation before writing the result back to the main memory. However, the implementation of such a technique may be complicated for the small problems considered as it depends on the hardware, the precision, and the algorithm. First, the current size of the shared memory is 48 KB per streaming multiprocessor (SMX) for the newest NVIDIA K40 (Kepler) GPUs, which is a low limit for the amount of batched problems data that can fit at once. Second, 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. Indeed, due to a limited parallelism in a small

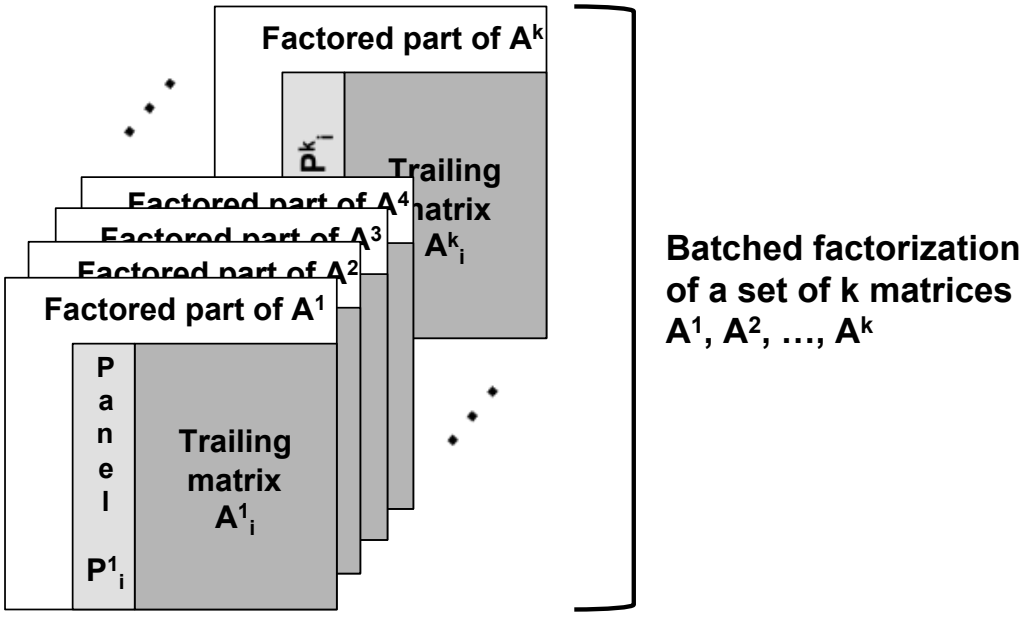

Figure 3.2: A batched one-sided factorization problem for a set of k dense matrices

panel's factorization, the number of threads used in the thread block will be limited, resulting in low occupancy, and subsequently poor core utilization.

Due to the SIMT programming model, all active threads execute the same instruction but on different data (operands). The best performance is achieved when all the processors cores in SMX are busy all the time, and the device memory access and latency can be hidden completely. 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 [41]. This process is similar to pipelining in CPU. In our study and analysis, 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 3.3. The performance obtained is three times better than the one in which the entire shared memory is saturated. Since the CUDA warp consists of 32 threads,

it is recommended to develop CUDA kernels that use multiples of 32 threads per thread block.

For good performance of Level 3 BLAS in trailing matrix update, panel width nb is increased. Yet, this increases 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 bi-diagonalization. A smaller nb is better because the panel operations in two-sided factorization are more significant than that in one-sided.

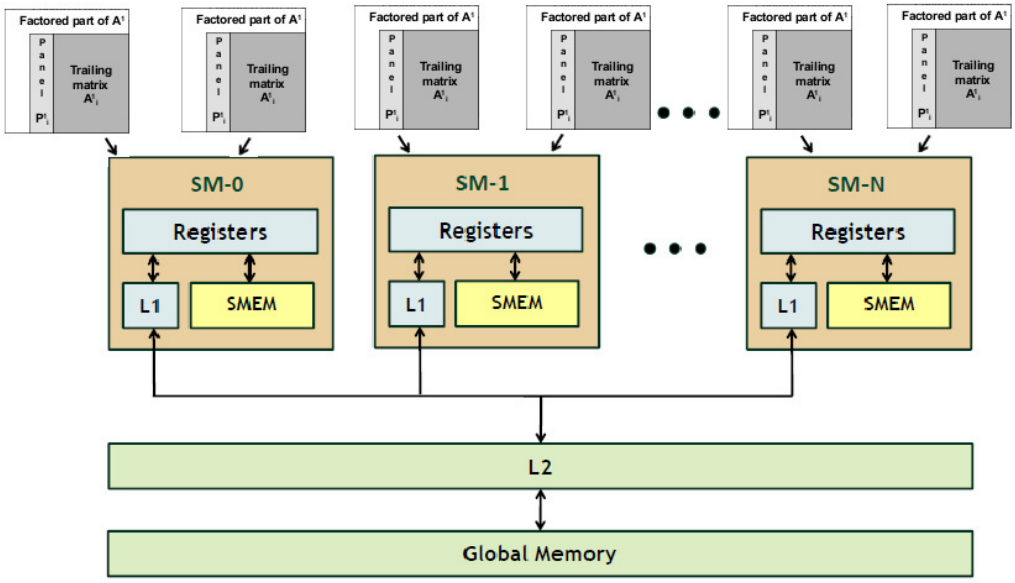

Figure 3.3: Multiple factorizations reside on one streaming-multiprocessor to allow the scheduler to swap to hide the memory latency.

Cholesky panel: Provides the batched equivalent of LAPACK's POTF2 routine. At step j of a panel of size (m, nb), the column vector $A(j:m, j)$ must be computed. This computation requires a dot-product using row $A(j, 1:j)$ to update element $A(j, j)$, followed by a GEMV $A(j+1, 1)\ A(j, 1:j) = A(j+1:m, j)$, and finally a Scal on column $A(j+1:m, j)$. This routine involves two Level 1 BLAS calls

(Dot and Scal), as well as a Level 2 BLAS GEMV. 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. We used shared memory to load both row $A(j, 1 : j)$ and column $A(j+1 : m, j)$ to reuse them, and wrote a customized batched GEMV kernel to read and write these vectors from/into the shared memory.

LU panel: Provides the batched equivalent of LAPACK's GETF2 routine to factorize panels of size $m \times nb$ at each step of the batched LU factorizations. It consists of three Level 1 BLAS calls (Idamax, Swap and Scal) and one Level 2 BLAS call (GER). The GETF2 procedure is as follows: Find the maximum element of the i^{th} column, swap the i^{th} row with the row owning the maximum, and scale the i^{th} column. To achieve higher performance and minimize the effect on the Level 1 BLAS operation, we implemented a tree reduction to find the maximum where all the threads contribute to find the max. Since it is the same column that is used to find the max then scaled, we load it to the shared memory. This is the only data that we can reuse within one step.

QR panel: Provides the batched equivalent of LAPACK's GEQR2 routine to perform the Householder panel factorizations. It consists of nb steps where each step calls a sequence of the LARFG and the LARF routines. At every step (to compute one column), the LARFG involves a norm computation followed by a Scal that uses the norm computation's results 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 similar to the MPI_REDUCE operation, and the last 32 elements are reduced by only one thread. Another optimization is to allow more than one thread-block to execute the LARFG kernel meaning the kernel needs to be split over two: one for the norm and one for scaling in order to guarantee the synchronization. Custom batched implementations of both LARFG and the LARF have been developed.

BRD 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 a sequence of the LARFG and a set of GEMV and Scal routines. At every step, the LARFG computes one column and one row Householder reflectors, interleaved by a set of GEMV calls. 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 similar to the MPI_REDUCE operation, and the last 32 elements are reduced by only one thread. The Householder reflectors are frequently accessed and loaded in shared memory. The GEMV calls is auto-tuned.

Trailing matrix updates: Mainly Level 3 BLAS operations. However, for small matrices it might be difficult to extract performance from very small Level 3 BLAS kernels. The GEMM is the best Level 3 BLAS kernel: it is GPU-friendly, highly optimized, and achieves the highest performance among BLAS. High performance can be achieved if we redesign our update kernels to be represented by GEMMs. For Cholesky, the update consists of the HERK routine. It performs a rank-nb update on either the lower or the upper portion of A_{22}. Since CUBLAS does not provide a batched implementation of this routine, we implemented our own. It is based on a sequence of customized GEMMs in order to extract the best possible performance. The trailing matrix update for the Gaussian elimination (LU) is composed of three routines: the LASWP that swaps the rows on the left and the right of the panel in consideration, followed by the TRSM to update $A_{12} \leftarrow L_{11}^{-1}A_{12}$, and finally a GEMM for the update $A_{22} \leftarrow A_{22} - A_{21}L_{11}^{-1}A_{12}$. The swap (or pivoting) is required to improve the numerical stability of the Gaussian elimination. However, pivoting can be a performance killer for matrices stored in column major format because

rows, in that case, are not stored continuously in memory, and thus can not be read in a coalesced way. Indeed, a factorization stored in column-major format can be 2× slower (depending on hardware and problem sizes) than implementations that transpose the matrix in order to internally use a row-major storage format [44]. Nevertheless, experiments have shown that this conversion is too expensive for batched problems. Moreover, the swapping operations are serial, row by row, limiting the parallelism. To minimize this penalty, we propose a new implementation that emphasizes a parallel swap and allows coalescent read/write. We also developed a batched TRSM routine, which loads the small $nb \times nb$ L_{11} block into shared memory, inverts it with the TRTRI routine, and then GEMM accomplishes the A_{12} update. Generally, computing the inverse of a matrix may suffer from numerical stability; but since A_{11} results from the numerically stable LU with partial pivoting and its size is just $nb \times nb$, or in our case 32×32, we do not have this problem [11]. For the Householder QR decomposition, the update operation is referred by the LARFB routine. We implemented a batched LARFB that is composed of three calls to the batched GEMM: $A_{22} \leftarrow (I - VT^HV^H)A_{22} \equiv (I - A_{21}T^HA_{21}^H)A_{22}$.

For Householder BRD, the update is achieved by two GEMM routines. The first one is GEMM of a non-transpose matrix with a transpose matrix ($A = A - V * Y'$), followed by another GEMM of a non-transpose matrix with a non-transpose matrix ($A = A - X * U'$) . The update is directly applied on trailing matrix A.

3.2.4 Algorithmic Innovation

To achieve high performance of batched execution, the classic algorithms (like that in LAPACK) are reformulated to leverage the computing power of accelerators.

Parallel Swapping

Profiling the batched LU reveals that more than 60% of the time is spent in the swapping routine. Figure 3.4 shows the execution trace of the batched LU for $2,000$

matrices of size 512. We can observe on the top trace that the classic LASWP kernel is the most time-consuming part of the algorithm. The swapping consists of nb successive interchanges of two rows of the matrices. The main reason that this kernel is the most time consuming is because the nb row interchanges are performed in a sequential order. Moreover, the data of a row is not coalescent in memory, thus the thread warps do not read/write it in parallel. It is clear that the main bottleneck here is the memory access. Slow memory accesses compared to high compute capabilities have been a persistent problem for both CPUs and GPUs. CPUs alleviate the effect of the long latency operations and bandwidth limitations by using hierarchical caches. Accelerators, on the other hand, in addition to hierarchical memories, use thread-level parallelism (TLP) where threads are grouped into warps and multiple warps assigned for execution on the same SMX unit. The idea is that when a warp issues an access to the device memory, it stalls until the memory returns a value while the accelerator's scheduler switches to another warp. In this way, even if some warps stall, others can execute, keeping functional units busy while resolving data dependencies, branch penalties, and long latency memory requests. In order to overcome the bottleneck of swapping, we propose to modify the kernel to apply all nb row swaps in parallel. This modification will also allow the coalescent write back of the top nb rows of the matrix. Note that the first nb rows are those used by the TRSM kernel that is applied right after the LASWP, so one optimization is to use shared memory to load a chunk of the nb rows, and apply the LASWP followed by the TRSM at the same time. We changed the algorithm to generate two pivot vectors, where the first vector gives the final destination (e.g., row indices) of the top nb rows of the panel, and the second gives the row indices of the nb rows to swap and bring into the top nb rows of the panel. Figure 3.4 depicts the execution trace (bottom) when using our parallel LASWP kernel. The experiment shows that this optimization reduces the time spent in the kernel from 60% to around 10% of the total elapsed time. Note that the colors between the top and the bottom traces do not match each other because the NVIDIA

profiler always puts the most expensive kernel in green. As a result, the performance gain obtained is about 1.8×.

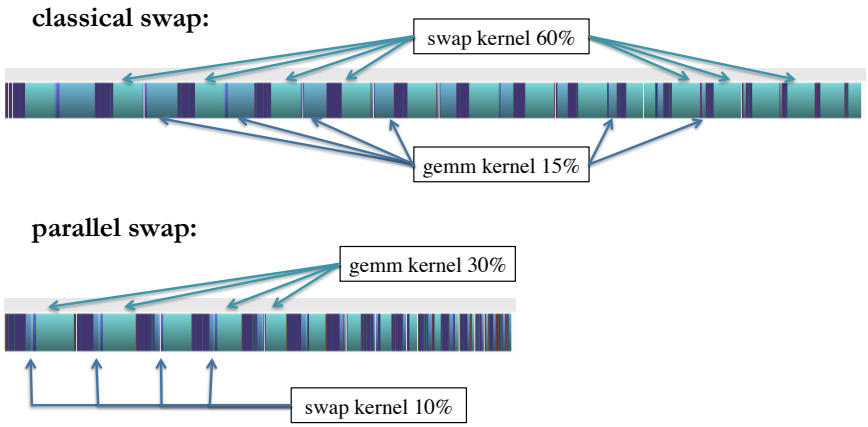

Figure 3.4: Execution trace of the batched LU factorization using either classic swap (top) or our new parallel swap (bottom).

Recursive Nested Blocking

The panel factorizations factorize the nb columns one after another, similar to the LAPACK algorithm. At each of the nb steps, either a rank-1 update is required to update the vectors to the right of the factorized column i (this operation is done by the GER kernel for LU and the LARF kernel for QR), or alternatively, a left looking update of column i by the columns on its left, before factorizing it (this operation is done by GEMV for the Cholesky factorization). Since we cannot load the entire panel into the GPU's shared memory, the columns to the right (in the case of LU and QR) or the left (in the case of Cholesky) are loaded back and forth from the main memory at every step. Thus, this is the most time-consuming part of the panel factorization. A detailed analysis using the profiler reveals that the GER kernel requires more than 80% and around 40% of the panel time and of the total LU factorization time respectively. Similarly for the QR decomposition, the LARF kernel used inside the panel computation needs 65% and 33% of the panel and the total QR factorization time respectively. Likewise, the GEMV kernel used within the

Cholesky panel computation needs around 91% and 30% of the panel and the total Cholesky factorization time, respectively. This inefficient behavior of these routines is also due to the memory access. To overcomes this bottleneck, we propose to improve the panel' efficiency and to reduce the memory access by using a recursive level of blocking technique as depicted in Figure 3.5. In principle, the panel can be blocked recursively until a single element remains. In practice, 2-3 blocked levels are sufficient to achieve high performance. The above routines must be optimized for each blocked level complicating the implementation. This optimization obtained more than 30% improvement in performance for the LU factorization. The same trend has been observed for both the Cholesky and the QR factorization.

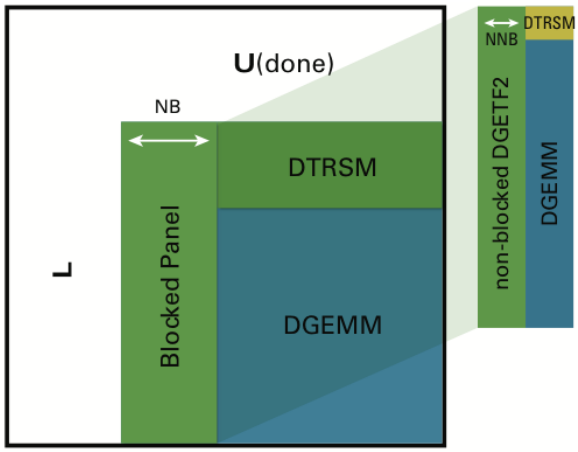

Figure 3.5: Recursive nested blocking

Trading Extra Flops for Higher Performance

The challenge discussed here is the following: for batched problems, the use of low-performance kernels must be minimized on the GPU even if they are Level 3 BLAS. For the Cholesky factorization, this concerns the SYRK routine that is used to update the trailing matrix. The performance of SYRK is important to the overall performance since it takes a big part of the run-time. We implemented the batched SYRK routine as a sequence of GEMM routines, each of size $M = m, N = K = nb$. In order to exclusively use the GEMM kernel, our implementation writes both the

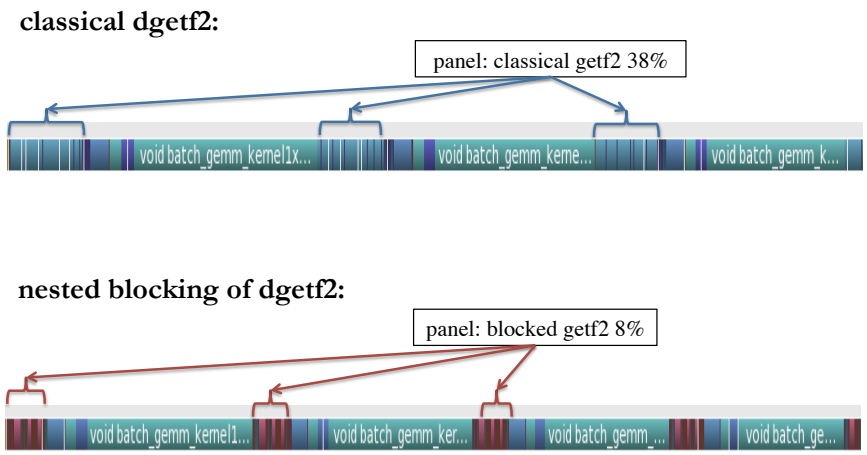

Figure 3.6: Execution trace of the batched LU factorization using either classic getf2 (top) or our recursive getf2 (bottom).

lower and the upper portion of the $nb \times nb$ diagonal blocks of the trailing matrix resulting in nb^3 extra operations for the diagonal block. However, since nb is small (e.g., $nb = 32$), these extra operations can be considered free. In practice, the extra operations allow us to use GEMM and thus achieve higher performance than the one that touches the lower/upper portion of the $nb \times nb$ diagonal blocks. Tests show that our implementation of SYRK is twice as fast as the GEMM kernel for the same matrix size. Thus, our SYRK is very well optimized to reach the performance of GEMM (which is twice as slow because it computes double the flops).

We applied the same technique in the LARFB routine used by the QR decomposition. The QR trailing matrix update uses the LARFB routine to perform $A_{22} = (I - VT^HV^H)A_{22} = (I - A_{21}T^HA_{21}^H)A_{22}$. The upper triangle of V is zero with ones on the diagonal. In the classic LARFB, A_{21} stores V in its lower triangular part and R (part of the upper A) in its upper triangular part. Therefore, the above is computed using TRMM for the upper part of A_{21} and GEMM for the lower part. The T matrix is upper triangular, and, therefore, the classic LARFB implementation uses TRMM to perform the multiplication with T. If one can guarantee that the lower portion of T is filled with zeroes and the upper portion of V is filled zeros and ones on the diagonal, TRMM can be replaced by GEMM. Thus, we implemented a

batched LARFB that uses three GEMM kernels by initializing the lower portion of T with zeros and filling up the upper portion of V with zeroes and ones on the diagonal. Note that this reformulation brings $3nb^3$ extra operations; but again, the overall time spent in the new LARFB update using the extra computation is around 10% less than the one using the TRMM.

Similar to LARFB and TRMM, we implemented the batched TRSM (that solves $AX = B$) by inverting the small $nb \times nb$ block A and using GEMM to get the final results $X = A^{-1}B$.

3.2.5 Optimization for Hardware Based on CUDA

In this section, we review some features of CUDA-based GPUs which have critical impacts on the performance of linear algebra problems. In CUDA, 32 consecutive threads are organized in one *warp* and are issued with the same instruction of memory access or execution. When a warp executes an instruction that accesses global memory, it coalesces the memory accesses into one transaction if the threads read consecutively from an aligned address. Otherwise, the warp may incur multiple transactions depending on the size of the word accessed by each thread and the distribution of the memory addresses across the threads. For example, if 32 words each of 4-byte are distributed in a striding manner in global memory such that each thread of one warp has to read a word separately, the throughput slows down 32 times compared to coalesced memory access.

The number of transactions affected also varies with other factors, like the compute capability of the device, alignment, and cache. Generally, the higher the compute capability, the lower the memory coalescing requirement. From computing capability 2.0, the cache is introduced to reduce the possibility of non-coalescing.

Figure 3.7 is an example of a warp of threads accessing global memory, with 4-byte for each, from an aligned address. In CUDA, this 128-byte segment is aligned in GPU memory . If the 128-byte segment is cached in L1, there is only one 128-byte

transaction for this warp. If a cache miss happens in L1, the L2 cache will service four 32-byte memory transactions since L2 has a cache line size of 32 bytes. A mis-aligned example is shown in Figure 3.8. If a cache hit, it incurs two memory transactions for this warp on computing capability of 2.0 and above as the data are located in two segments due to the mis-alignment. If a cache miss, again, there will be six 32-byte memory transactions compared to five in the aligned situation. The mis-aligned problem is serious in dense linear algebra. If the starting thread is from a mis-aligned address, the following threads (and thus warps) are all mis-aligned.

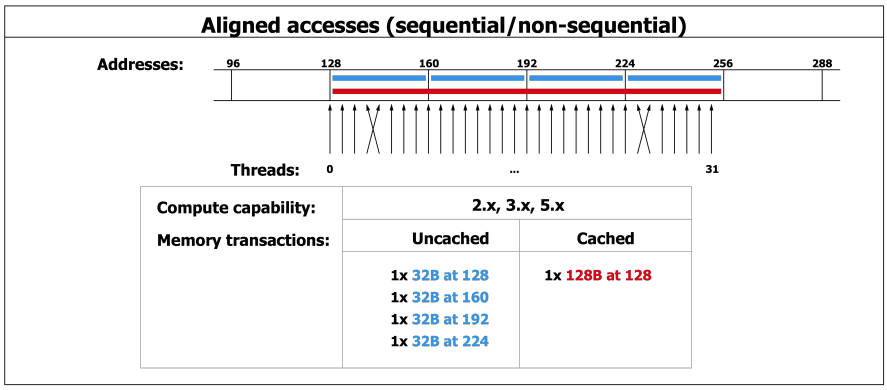

Figure 3.7: Aligned memory accesses by a warp of threads

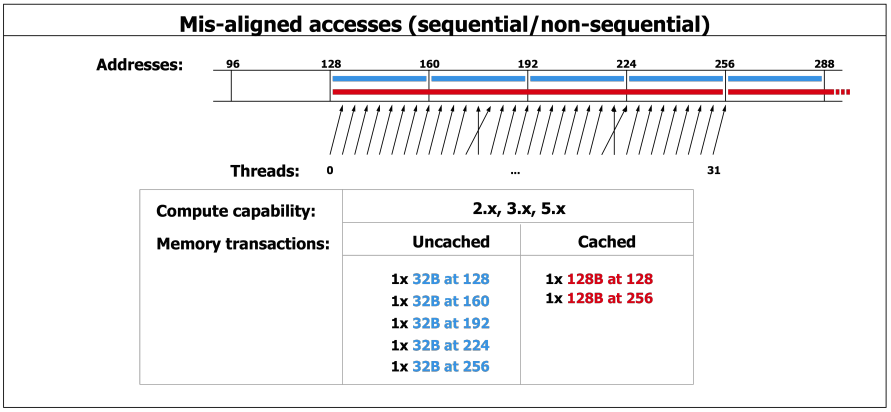

Figure 3.8: Mis-aligned memory accesses by a warp of threads

When the compiler allocates new space for the matrix, the starting address is always aligned in GPU memory. In Figure 3.9 and 3.10, the blue curves indicate

the performance of GEMV transpose and non-transpose of double precision in the aligned situation, respectively. However, when the algorithm iterates the sub-portion of the matrix, the starting address may not be aligned, as shown in Figure 3.11 by the bi-diagonalization (BRD) algorithm. In Figure 3.9 and 3.10, the green curves depict the performance of the two GEMV in this situation. It 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 warps are in a mis-aligned way.

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. By padding the corresponding elements in the multiplied vector as zeros, extra results are computed but finally discarded in the writing stage. Figure 3.9 and 3.10 show that our padding technique enables the GEMV in the BRD algorithm to run at a speed close to the aligned address' speed.

3.3 Auto-tuning

3.3.1 Batched Level 3 BLAS GEMM Tuning

The efforts of maximizing 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

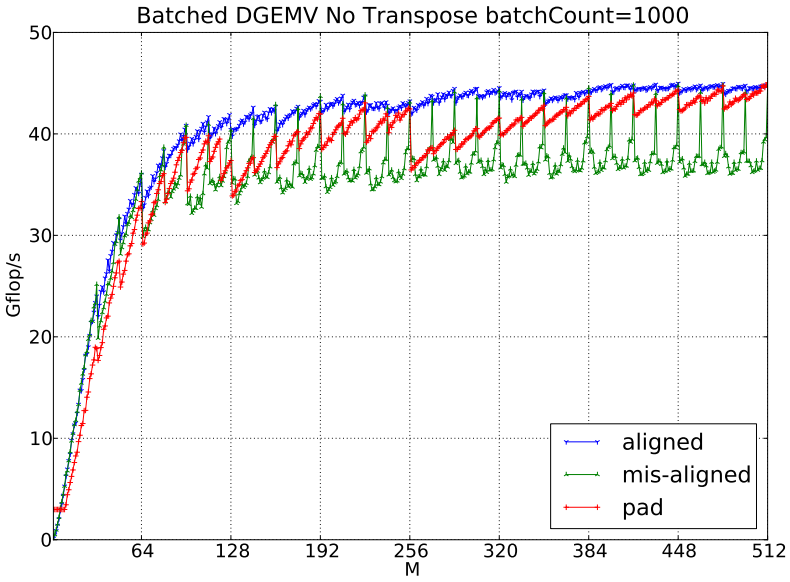

Figure 3.9: Performance of batched DGEMV (non-transpose) in three situations: aligned, mis-aligned, and pad.

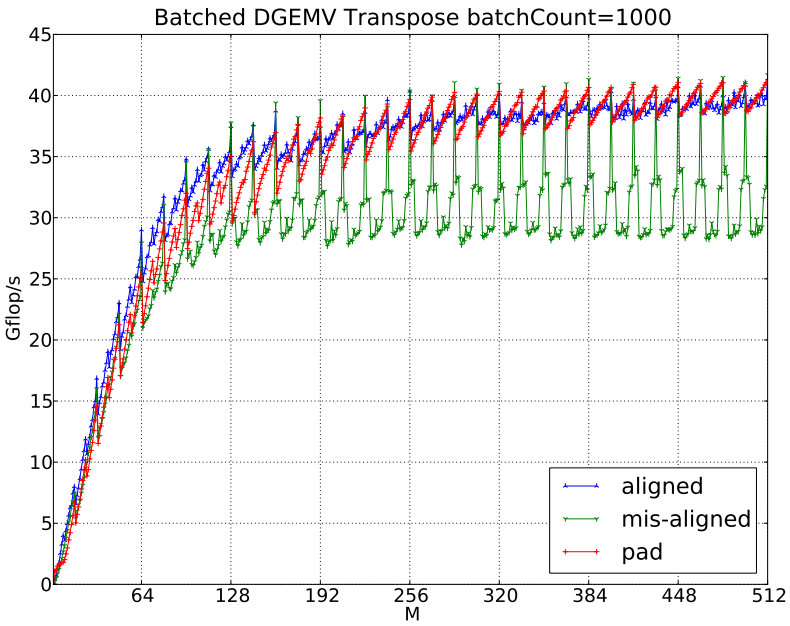

Figure 3.10: Performance of batched DGEMV(transpose) in three situations: aligned, mis-aligned, and pad.

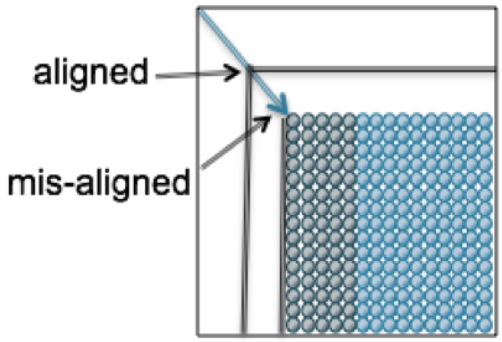

Figure 3.11: The BRD algorithm accesses the sub-matrix step by step.

vendors maintain the performance portability across different generations of their architectures [46].

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.

We tune our batched kernels under BEAST (Bench-testing Environment for Automated Software Tuning), which is an auto-tuning framework to explore and optimize the performance of computational kernels on accelerators [7]. The programmer needs to supply a templated kernel and define tuning parameter search space. The 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. See Table 3.2 for the meaning of the parameters.

The search space can be very big, yet it would be efficiently pruned with a set of constraints. The derived constraints 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. For example, the maximum shared memory size is 48KB per SMX on Kepler GPUs. Based on these metrics, configurations violating the requirement will be discarded. The constraints

may be soft in terms of performance. We require at least 512 threads per GPU Streaming Multiprocessor (SM) to ensure a reasonable occupancy.

After pruning, there are hundreds of valid configurations as shown in Table 3.1, reduced from thousands in search space. GEMM of single real precision (SGEMM) has the most valid configurations while GEMM of double complex precision (ZGEMM) has the least. An element of double complex precision is four times bigger than one in single precision. Many configurations in ZGEMM exceed 48KB shared memory hardware constraints and are eliminated in pruning.

However, tuning is a challenge as programmers face a typical conundrum of multi-variable optimization. Not only is the number of parameters large and, therefore, so is the resulting search space, but the parameters are also usually linked by counterintuitive relationships (i.e., a seemingly beneficial setting for one prevents a reasonable setting for another). Decisions are usually made by finding piecewise optimums and trade-offs.

We consider a batched GEMM of double precision (DGEMM) with the rank-32 update ($K = 32$) as an example. This routine is called by batched LU factorization. Other precisions and shape are tuned and analyzed in the same way. There are 157 valid configurations for batched DGEMM with a non-transpose matrix and a non-transpose matrix. During tuning, each kernel is launched with one configuration. The four most performant kernels are shown in Figure 3.12. The kernel with configuration 111 outperforms others most of the time for matrices of size larger than 128 and is more stable than configuration 116, though the latter is able to reach the peak at certain size. For sizes less than 128, configuration 93 is more stable than configuration 111. Therefore, there is a switchover between configuration 111 and 93 at size 128. All the four configurations outperforms CUBLAS batched GEMM routine a lot. The details of the four configurations in Table 3.2 explain their behaviors. Configuration 93 has a smaller thread block and tile size, and, therefore, performs best for matrices of small size. The performance curve of configuration 116 shakes at every step size of 64 because its tile size is 64 (DIM-M). Configuration 107 and 111 are very similar except

exchanging BLK-M and BLK-N, resulting in very similar performance. Configuration 111 is preferred since it proves to be optimal in other cases like $K = 8$ and $K = 16$.

Figure 3.13 and 3.14 show the tuning results of batched DGEMM with rank-8 and rank-16 update ($K = 8$ and $K = 16$) on a K40c GPU. These two routines are called by GEBRD factorization for trailing matrix update. Figure 3.15 shows our batched DGEMM (denoted as the MAGMA batched) performance against other solutions after auto-tuning. The number of matrices is 500. The 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 500 matrices. 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 75Gflop/s or 30% faster than CUBLAS on average and more than 3× faster than the CPU solution.

Note that the performance of batched is lower than that of the standard GEMM with the same amount of input data since 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 [33].

Table 3.1: Numbers of valid configurations for batched GEMM.

Precision	SGEMM	DGEMM	CGEMM	ZGEMM
Valid configurations	632	157	538	76

Streamed GEMM

Another way of parallelizing many small size matrices for GEMM computation is to launch a CUDA kernel for each matrix. All the kernels are put into multiple streams. CUDA allows up to 32 simultaneous streams per GPU. The kernels in the same stream are still sequentially processed. Our main goal is to achieve higher performance;

Table 3.2: DIM-X and DIM-Y denote the number of threads in x-dimension (row) and y-dimension (column) of the thread block, respectively. BLK-M(N,K) denotes the tile size in the matrix along each corresponding dimension.

Index	DIM-X	DIM-Y	BLK-M	BLK-N	BLK-K
93	16	8	32	24	16
107	16	16	32	48	16
111	16	16	48	32	16
116	16	16	64	32	16

therefore, we performed deep analysis of every kernel of the algorithm. We found that 70% of the time is spent in the batched GEMM kernel. An evaluation of the GEMM kernel' performance using either batched or streamed GEMM is illustrated in Figure 3.16.

The curves let us conclude that the streamed GEMM was performing better than the batched one for some cases (e.g., for $K = 32$ when the matrix size is of an order of $M > 200$ and $N > 200$). We note that the performance of the batched GEMM is stable and does not depend on K, in the sense that the difference in performance between $K = 32$ and $K = 128$ is minor. However, it is bound by 300 Gflop/s. Therefore, we proposed to use the streamed GEMM whenever it is faster and to roll back to the batched one otherwise.

Figure 3.17 shows the trace of the batched LU factorization of 2,000 matrices of size 512 each, using either the batched GEMM (top trace) or the combined streamed/batched GEMM (bottom trace). We will see that the use of the streamed GEMM (when the size allows it) can speed up the factorization by about 20%, as confirmed by the performance curve.

3.3.2 Batched Level 2 BLAS GEMV 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

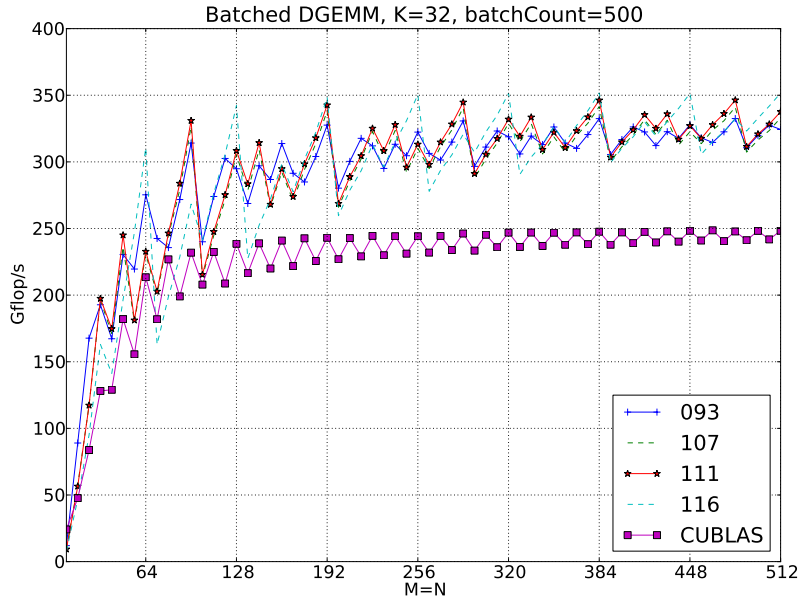

Figure 3.12: The four most performant batched DGEMM kernels (K=32) in our tuning. CUBLAS is given for comparison.

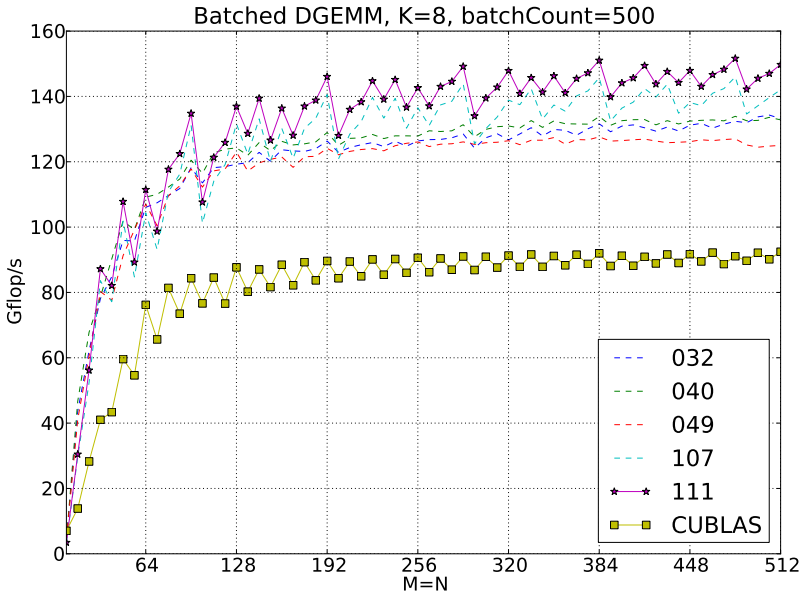

Figure 3.13: The five most performant batched DGEMM kernels (K=8) in our tuning. CUBLAS is given for comparison.

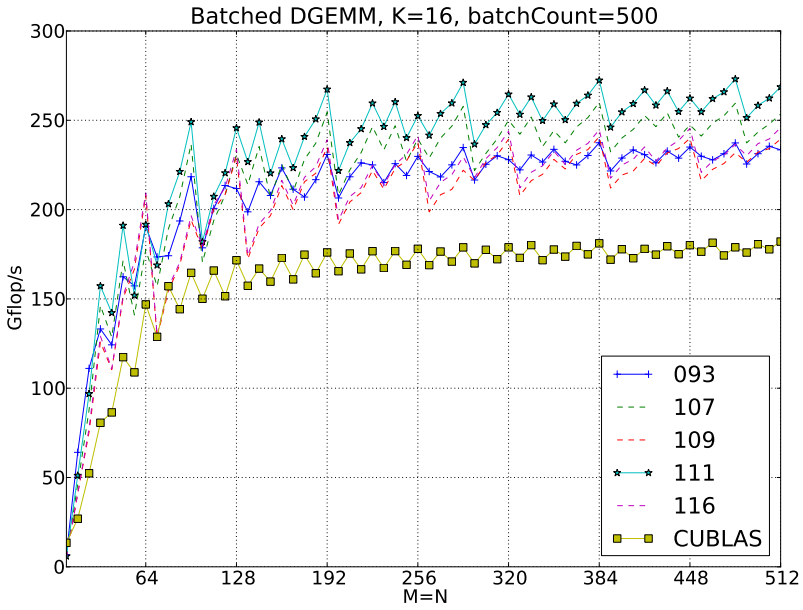

Figure 3.14: The five most performant batched DGEMM kernels (K=16) in our tuning. CUBLAS is given for comparison.

the reduction result. Since matrices are stored in column-major format and elements in a column are stored consecutively in memory, 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 as discussed in Section 3.2.5. 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.

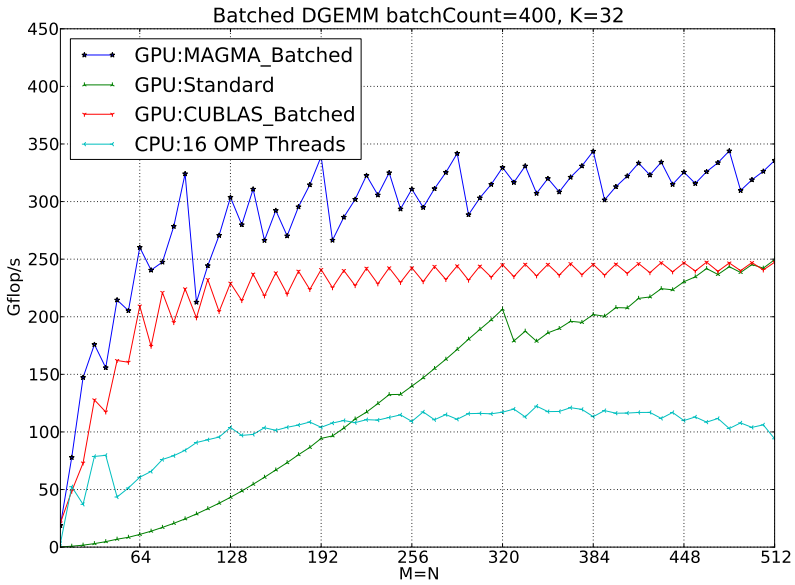

Figure 3.15: Performance of our batched DGEMM (K=32) vs. other solutions on CPUs or GPUs.

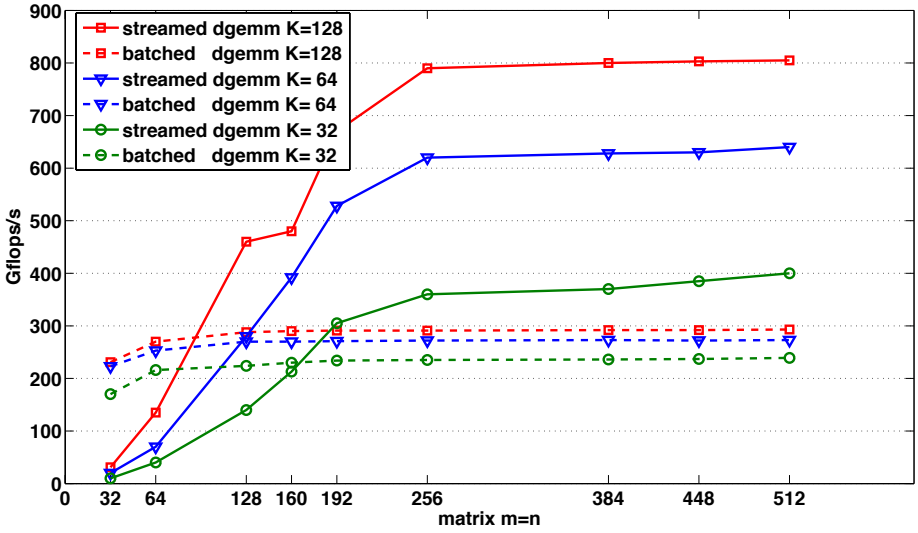

Figure 3.16: Performance of streamed and batched DGEMM kernels for different values of K and different matrix sizes where M=N

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

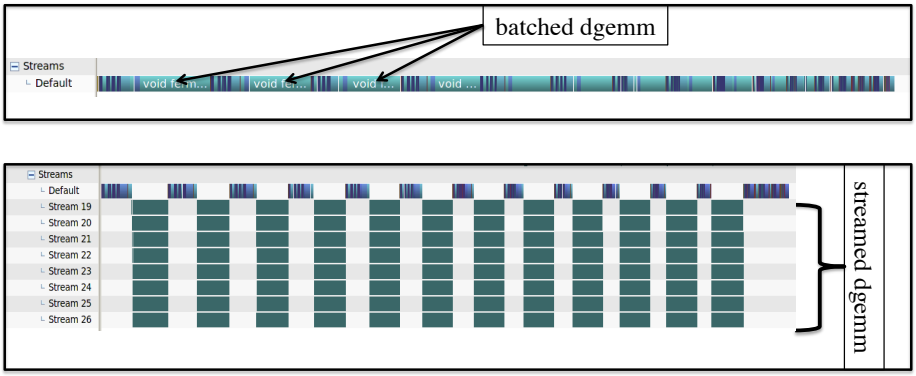

Figure 3.17: Execution trance of either batched or streamed GEMM in batched LU factorization

wide matrix (or called fat matrix, with both terms being interchangeable throughout this dissertation) 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 six tall matrix GEMVN calls, two tall matrix GEMVT calls, two wide matrix GEMVT calls, one wide matrix GEMVN call, one square GEMVN call and one square GEMVT call in one step of BRD panel factorization. See Figure 3.18. For a matrix of size n, there are n steps in the BRD algorithm. Since they are called extensively, the overall BRD performance highly relies on efficient implementations of these GEMV variants. We proposed auto-tuning to optimize them. By auto-tuning, the four precisions, complex/real and double/single, are automatically tackled. The source code is templated in C++. The template parameters decide the configurations. Each kernel is associated with one configurations. These kernels are then launched one by one in auto-tuning. From the tuning results, we find the optimal configuration. In this dissertation, we differentiate the term configuration and setting in this way: configuration is particularly used in auto-tuning. The term setting is used in a broad

sense. When an optimal configuration is finalized by auto-tuning, the setting is fixed. Table 3.3 shows examples of settings in GEMVN and GEMVT. As discussed in Section 3.2.2, a big-tile setting is proposed to take advantage of data reuse through shared memory. The big-tile setting requires one thread block to process one matrix. The big-tile setting is used when multiple GEMV device function are in the same kernel to synchronize and maximize the data reuse of the matrix. It is against to the classic setting where one matrix may be divided into tiles and processed by multiple thread blocks which synchronize through the GPU main memory.

Figures 3.19 to 3.29 show the tuning results of different batched DGEMV variants (wide, tall, square, transpose, non-transpose in double precision) on a K40c GPU. The number of matrices is 400. The classic setting result is on the left side, and the big-tile setting is on the right side of each figure. For tall and wide matrices of DGEMVT, the big-tile setting is slightly slower than the classic setting when $K < 16$, but becomes big when $K >= 32$, where K is row M in the wide matrix or column N in the tall matrix. For square matrices, the big-tile setting is about 5Gflop/s slower than that classic setting at size 256. For DGEMVN, the big-tile and the classic setting have little differences. In fact, a classic setting with tile size 512 is reduced to a big-tile setting since the testing matrix size is up to 512.

In DGEMVT, the classic setting is more performant than the big-tile setting because of a higher degree of parallelism. In the transpose case, columns are independent, and parallelism is exploited among the columns. For the same column size N, compared to the classic setting, the big-tile setting loses $(N/BLK\_N) - 1$ thread blocks which, alternatively, compensates with a loop. Therefore, the degree of parallelism is less in the big-tile setting in the transpose case. For the non-transpose DGEMVN, rows are independent and the parallelism is exploited among the rows. Although the number of thread blocks is fewer, the degree of parallelism is preserved by the number of threads which is the same as the number of rows.

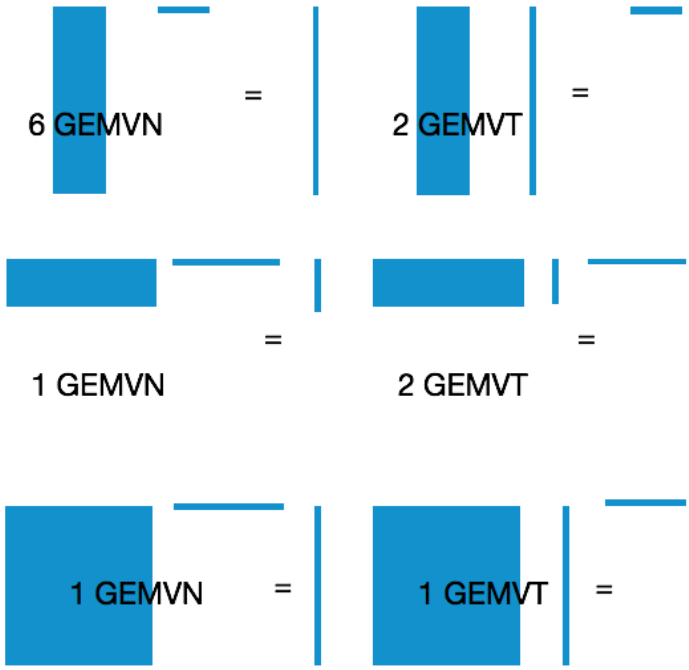

Figure 3.18: Number of GEMV calls in one step of the BRD algorithm.

3.4 Batched Problems of Variable 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 3.2.2).

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,

Table 3.3: DIM-X and DIM-Y denote the number of threads in x-dimension (per row) and y-dimension (per column), respectively. BLK-M(N) denotes the tile size in row (column) dimension, respectively. Tiling concept in the the reduction dimension (which is column for the non-transpose, row for the transpose) is not applicable. 10000 is selected to represent the tile size in big-tile setting as a size beyond 10000 is so big that the concept of batching is no longer applicable. Index is the configuration ID.

Variant	Setting	Index	DIM-X	DIM-Y	BLK-M	BLK-N
Square GEMVN	classic	120	128	4	512	N/A
Square GEMVT	classic	90	16	8	N/A	8
Wide GEMVN	big-tile	23	128	1	10000	N/A
Tall GEMVT	big-tile	11	16	8	N/A	10000

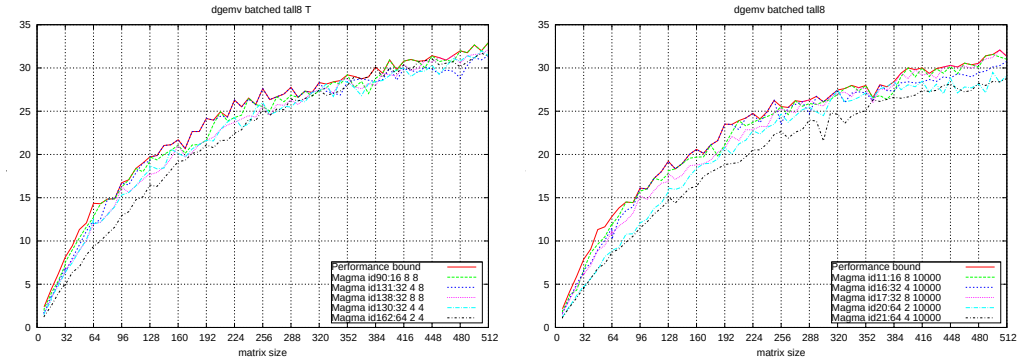

Figure 3.19: Batched DGEMVT for the tall matrix with 8 columns.

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 3.30 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

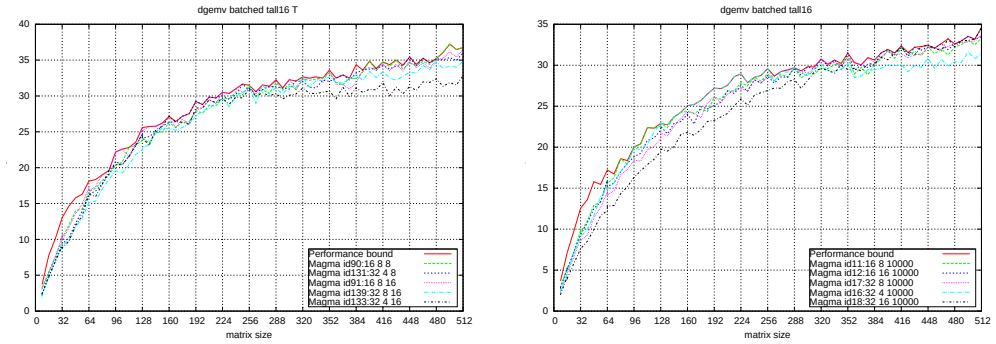

Figure 3.20: Batched DGEMVT for the tall matrix with 16 columns.

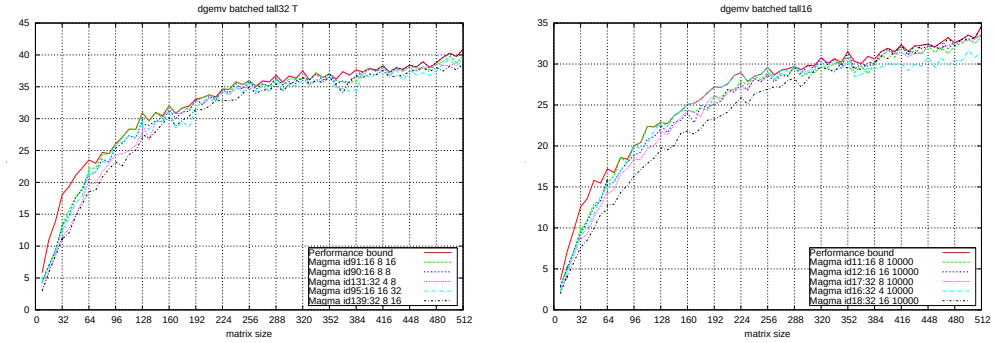

Figure 3.21: Batched DGEMVT for the tall matrix with 32 columns.

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.

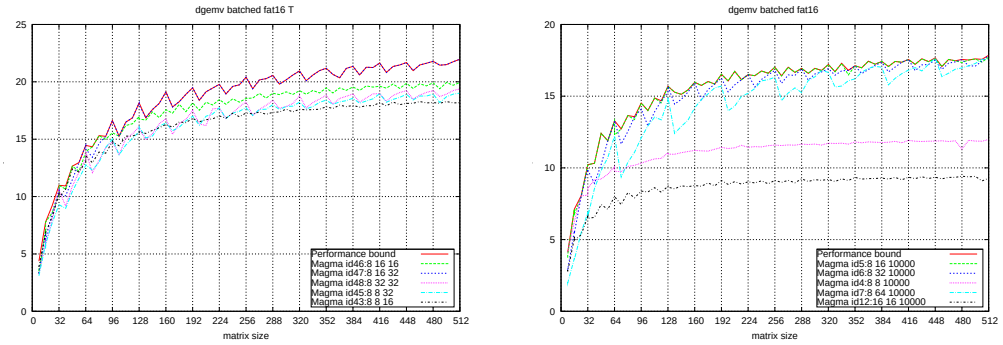

Figure 3.22: Batched DGEMVT for the wide matrix with 16 rows.

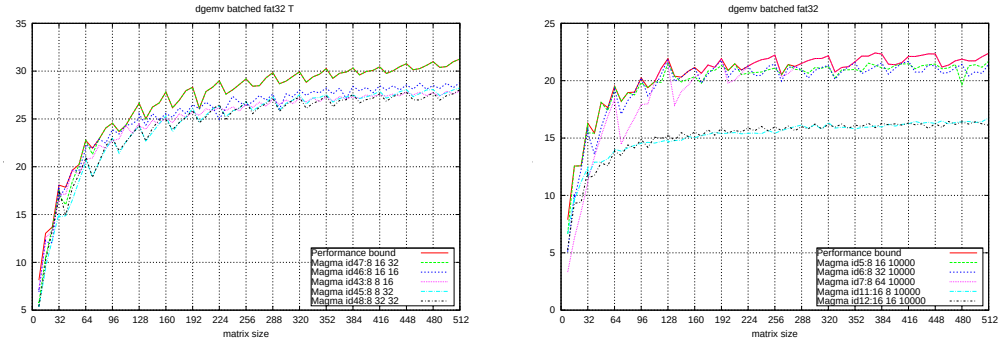

Figure 3.23: Batched DGEMVT for the wide matrix with 16 rows.

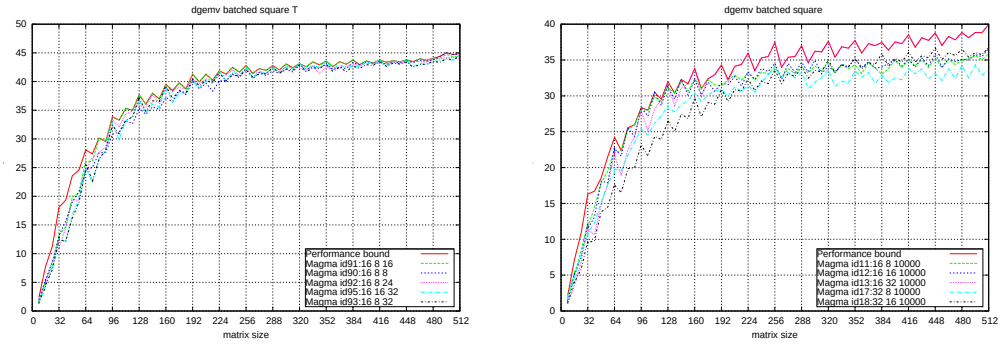

Figure 3.24: Batched DGEMVT for the square matrix.

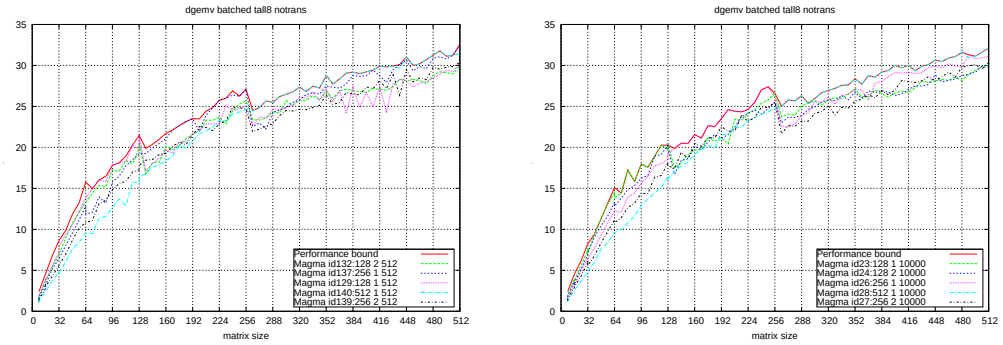

Figure 3.25: Batched DGEMVN for the tall matrix with 8 columns.

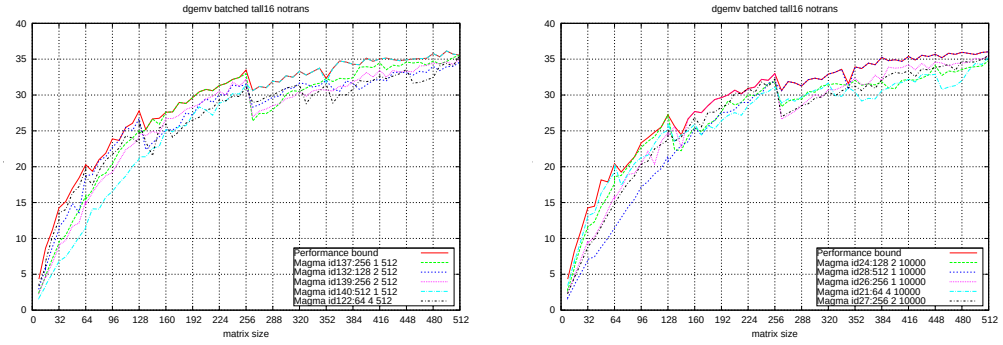

Figure 3.26: Batched DGEMVN for the tall matrix with 16 columns.

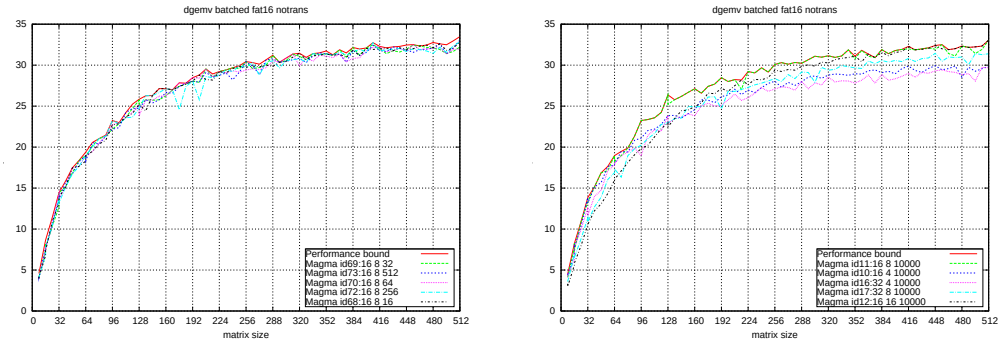

Figure 3.27: Batched DGEMVN for the wide matrix with 16 rows.

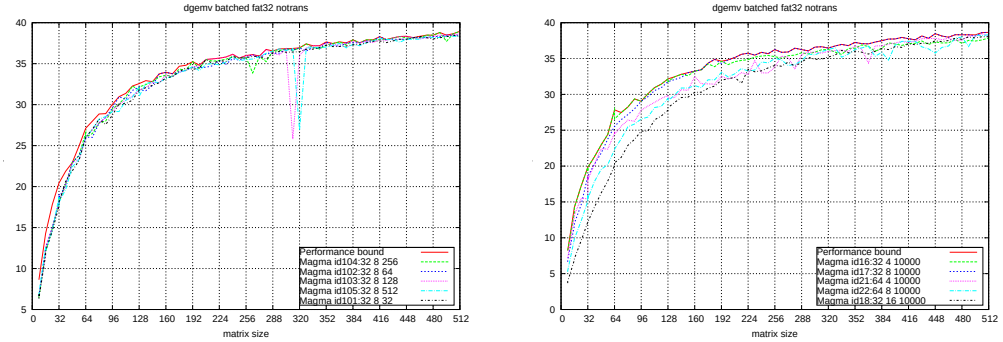

Figure 3.28: Batched DGEMVN for the wide matrix with 32 rows.

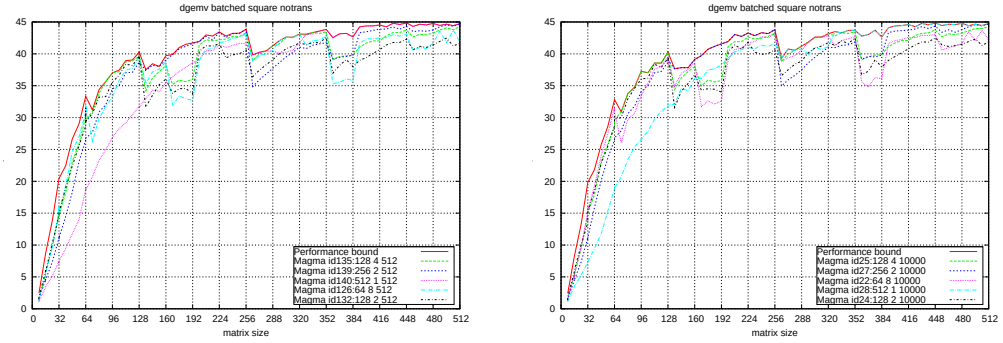

Figure 3.29: Batched DGEMVN for the square matrix.

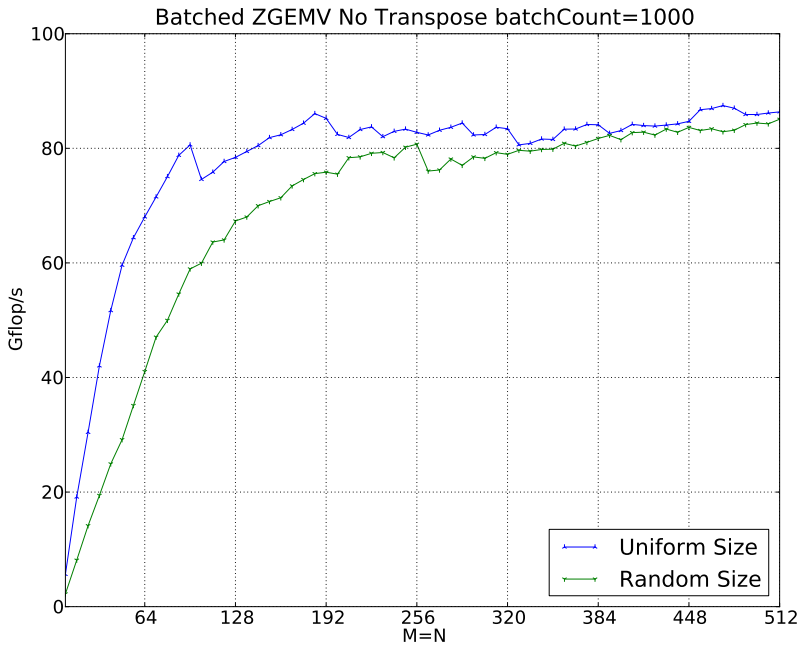

Figure 3.30: Performance of batched GEMV in double complex precision with uniform size and random size, respectively.

Chapter 4

Results and Discussions

4.1 Hardware Description and Setup

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 52 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.

A number of software packages are used for the experiments. On the CPU side, we use the MKL (Math Kernel Library) [23] with the Intel ICC compiler (version 2013.sp1.2.144) and on the GPU accelerator, we use CUDA toolkits of version 6.0.37.

We note that in this particular setup, the CPU and the GPU have about the same theoretical power draw. In particular, the Thermal Design Power (TDP) of the Intel Sandy Bridge is 115 W per socket, or 230 W in total, while the TDP of the K40c GPU is 235 W. Therefore, we roughly expect that a GPU would have a

power consumption advantage if it outperforms the 16 Sandy Bridge cores in terms of time. Note that based on the theoretical peaks, the GPU's advantage should be about 4×. This advantage is observed in practice as well, especially for workloads on large data-parallel problems that can be efficiently implemented on GPUs.

Table 4.1: Overview of the Intel E5-2670 CPU

8-cores Intel Sandy Bridge E5-2670 CPU

Frequency	L1 Cache	L2 Cache	L3 Cache	TDP	Peak Peformance
2.6GHz	64KB	256KB	20MB	115W	20.8Gflop/s per core

Table 4.2: Overview of the NVIDIA K40c GPU

Frequency	Cores	TDP	Peak Performance (in double)
0.825GHz	2880	235W	1430Gflop/s

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.

4.2 Performance on the K40c GPU

4.2.1 Performance of One-sided Factorizations

Getting high performance on accelerators remains a challenging problem that we address with the algorithmic and programming techniques described in previous chapters of this dissertation. Efficient strategies are used to exploit parallelism and to increase the use of Level 3 BLAS operations across the GPU. We highlight them through a set of experiments on our systems. We compare our batched implementations with the CUBLAS library whenever possible [33]. Our experiments

are performed on batches of $2,000$ matrices of different sizes going from 32×32 to 512×512.

Figure 4.1 shows the performance of the batched LU factorization. The dgetrfBatched version, marked as "CUBLAS", reaches around 70 Gflop/s for matrices of size 512×512. We first compare it to a naive implementation that is based on the assumption that the size (< 512) is very small for block algorithms, and, therefore, uses the non-blocked version. For LU, the non-blocked algorithm is the batched dgetf2 routine. The routine is very slow, and the performance is less than 30 Gflop/s. Note that although low, it is the optimal performance achievable by this type of memory-bound algorithms.

Our second comparison is to the *classic* LU factorization, i.e., the one that follows LAPACK's two-phase implementation described in Algorithm 1. This algorithm achieves 63 Gflop/s as shown in Figure 4.1.

To reach beyond 100 Gflop/s, we use the technique that optimizes pivoting with *parallel swap*. Next step in performance improvement is the use of two-level blocking of the panel which enables performance to go slightly above 130 Gflop/s. The last two improvements are *streamed/batched GEMM*, which moves the performance beyond 160 Gflop/s, and the *two-levels blocking update*, (also we called *recursive blocking*) completes the set of optimizations and takes the performance beyond 180 Gflop/s. Thus, our batched LU achieves up to 2.5× speedup compared to its counterpart from the CUBLAS library. These improvement techniques are described in Section 3.2.4.

For Cholesky, the performance improvement is shown in Figure 4.2. Non-blocked algorithm achieves 30 Gflop/s, similar to LU, bounded by Level 2 BLAS performance. The classic blocked one achieves less than 50 Gflop/s because there are still considerable Level 2 BLAS operations in panel factorizations. By introducing recursive blocking, the sub-panel is recursively blocked to a size that can fit into shared memory that is fast on-chip memory. The triangular solve is a TRSM routine (that solves $Ax = B$) and trailing matrix update is an HERK routine. We implement the batched TRSM by inverting the small nb by nb blocks of A and using GEMM to

get the final result. Both TRSM and GEMM are efficient BLAS-3 routines. Overall, the performance moves to 200 Gflop/s.

The progress of batched QR shows the same behavior. See Figure 4.3. The classic blocked algorithm does not exceed 60 Gflop/s. The recursive blocking improves the performance to 105Gflop/s. The optimized triangular T gets up to 125 Gflop/s. The other optimizations including replacing TRMM with GEMM with streamed/batched GEMM in trailing matrix update bring to 170 Gflop/s. TRMM is a triangular matrix-matrix vector multiplication. The triangular layout will introduce the branches inside a warp. By filling the corresponding portion of the matrix as zeros and saving the data, we can replace the TRMM with GEMM. The extra flops introduced are negligible. This optimization is described in details in Section 3.2.4.

The performance of CUBLAS v7.5 dgeqrfBatched for batched QR does not exceed 27 Gflop/s at the best time and slows down after size 64. For size less than 80, CUBLAS is faster than our batched QR (denoted as MAGMA in Figure 4.12) because memory space has to be allocated in order to use the optimization techniques for big size. The memory allocation is an overhead for small matrices of size less than 80 in the QR factorization.

4.2.2 Performance of Forward/Backward Substitution

Different solutions of batched forward/backward substitutions (solving $Ax = b$, where A is triangular, and b is a vector) in double precision (DTRSV) are given in Figures 4.4 and 4.5, respectively. They are used in solving linear systems after one-sided factorizations. The solution of inverting matrix A and then solving it with a GEMV routine ($x = A^{-1}b$ [15]) 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 to 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

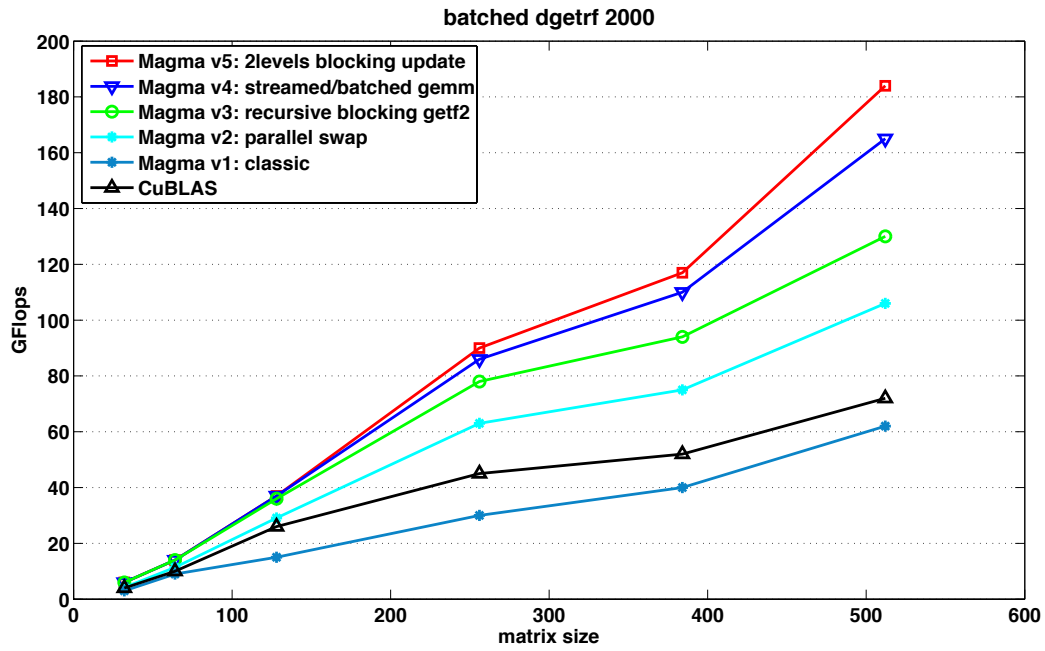

Figure 4.1: Performance in Gflops/s of different versions of our batched LU factorization compared to the CUBLAS implementation for different matrix sizes where $m = n$.

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.

4.2.3 Performance of Bi-diagonalization

Amdahl's law tells the maximum speedup achieved on multiple processors compared to one processor [3]. It is expressed by

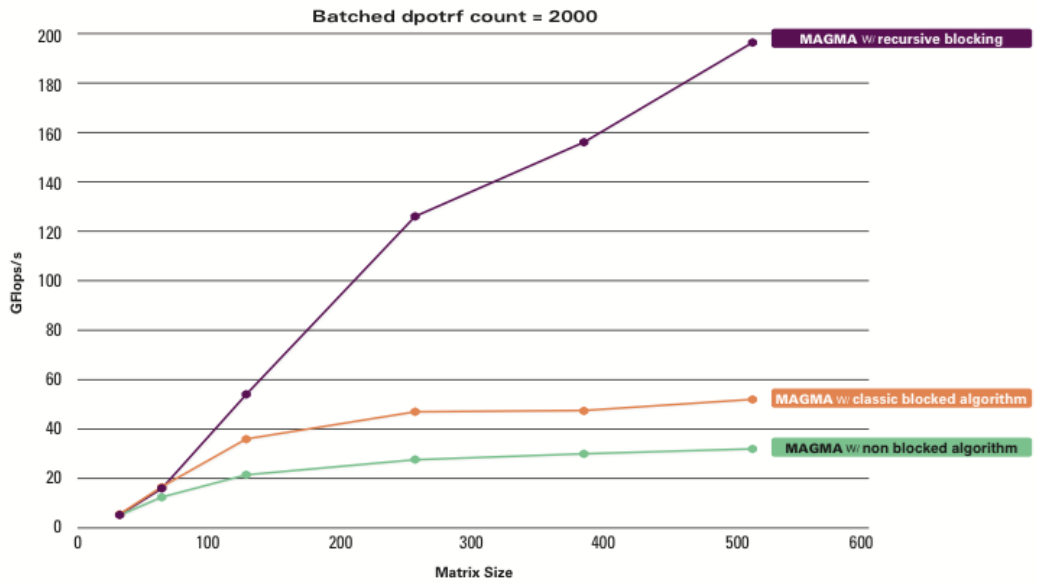

Figure 4.2: Performance in Gflops/s of different versions of our batched Cholesky factorization on a K40c GPU for different matrix sizes.

$$Speedup = \frac{1}{(1-F) + \frac{F}{N}}$$

, where $F >= 0, <= 1$ is the fraction of the code can be accelerated and $N > 0$ is number of processors.

For example, if $F = 0.5, N = \infty$, the maximum speedup is 2. As discussed in Section 2.3, half of the GEBRD operations is Level 2 BLAS. They are memory bound and do not scale with the number of processors. Amdahl's law indicates that the performance of GEBRD does not exceed 2× of Level 2 BLAS GEMV. However, in practice, the other half operations can not be infinitely accelerated and ignored. Therefore, the speedup is less than 2.

If viewed in a mathematical way, the total time of GEBRD includes the time spending on GEMV and GEMM. The performance of GEBRD (in flop/s) can be calculated by the following equation:

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

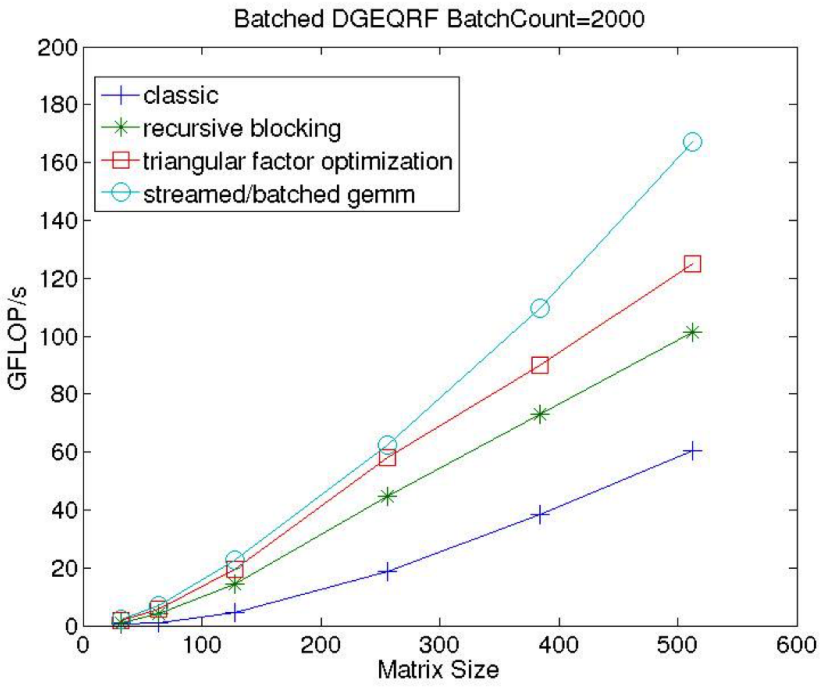

Figure 4.3: Performance in Gflops/s of different versions of our batched QR factorization on a K40c GPU for different matrix sizes where $m = n$.

By reforming it, we get

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

Supposing GEMM is 7× faster than GEMV, we obtain

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

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.

Table 4.3: Different shape GEMV calls in GEBRD

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

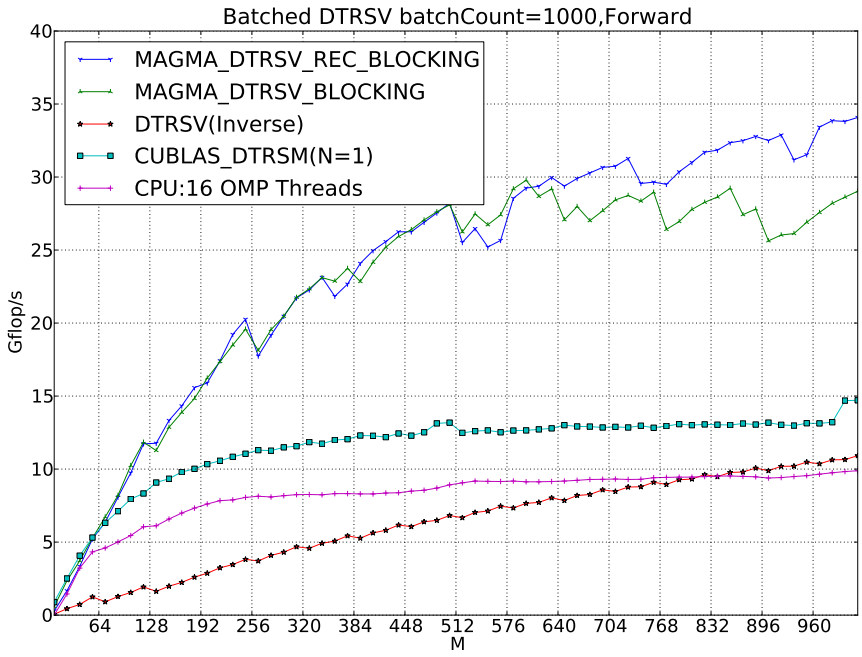

Figure 4.4: Performance in Gflops/s of different solutions of batched DTRSV (forward substitution) for different matrix sizes

Figure 4.8 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 LAPACK's two-phase implementation as depicted in Algorithm 3 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 tall/ wide matrices in the panel operation. There are 13 GEMV routine calls in GEBRD, as shown in Table 4.3. By tuning these GEMV routines for tall/wide matrices as described in Section 3.3.2 , the performance doubled in

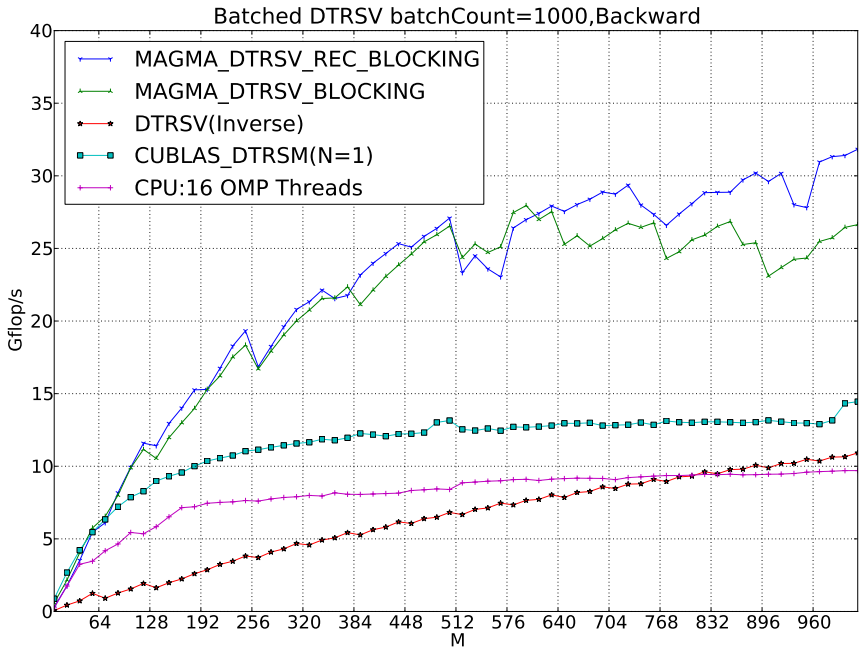

Figure 4.5: Performance in Gflops/s of different solutions of batched DTRSV (backward substitution) for different matrix sizes

version 2. These GEMV routines are called in the form of device functions in the panel factorization kernel to minimize the kernel launching overhead. The column/ row vector of Householder reflectors and the to-be-updated column in matrix X and Y (see Figure 2.3) 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 3.2.2, there is a trade-off between data reuse and the degree of parallelism. The classic setting with multiple thread blocks processing one matrix is better than the big-tile setting when a higher degree of parallelism is more desired than data reuse. Figure 4.6 describes the performance of batched GEMV (transpose) in double precision (DGEMVT) for square matrices with the two settings, respectively. Compared to the classic setting (in blue), the big-tile setting (in red)

losses $N/16 - 1$ thread blocks per matrix, and, therefore, has the lower degrees of parallelism, resulting in a 5Gflop/s loss at size 512, where N is the number of columns.

In version 3, we launch new GEMV kernels to compute the square matrix with the class setting. We roll back to the big-tile setting for the GEMV computation of wide/ tall matrices in order to reuse data in shared memory where the data caching proves to be more important. The performance of version 3 boosts to 50 Gflop/s from 40 Gflops in version 2 at size 512 after we adopt this technique.

Figure 4.7 describes the upper bound performance of GEBRD. If the computation of wide/tall matrices and GEMM in trailing matrix update is assumed to be free, the performance of GEMV of the square matrix should reach the upper bound. Instead, the GEMV in GEBRD does not run at the optimal speed but has a constant gap to the upper bound. The gap is 5Gflop/s. By profiling the GEMV step by step, we find this gap is from the memory mis-aligned problem because the BRD algorithm iterates the matrix step by step as demonstrated in Figure 3.11.

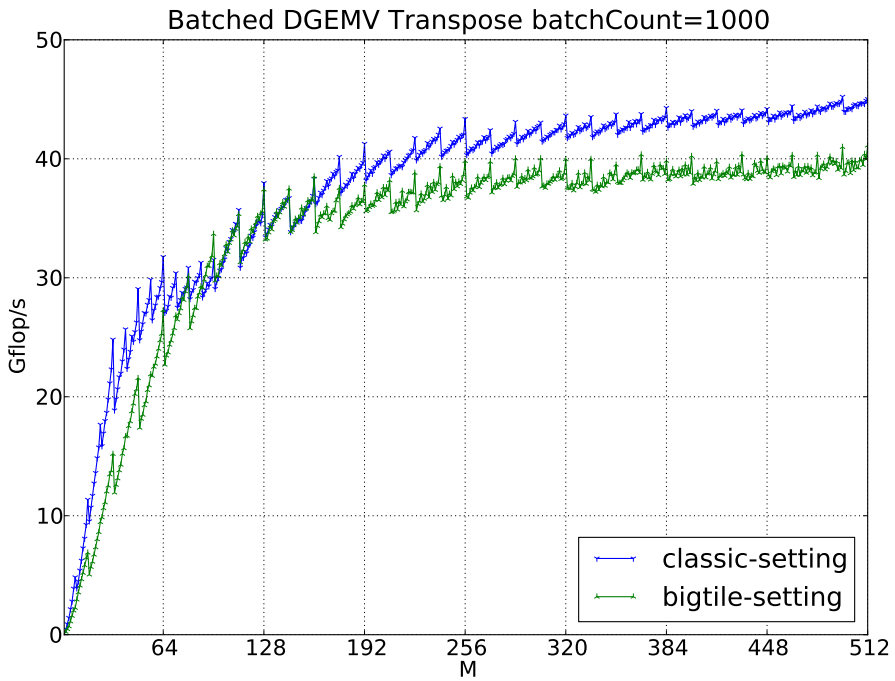

Figure 4.6: Performance of batched DGEMVT with two settings.

To overcome the mis-aligned issue, we adopt a padding technique in version 4 as described in Section 3.2.5. By padding the corresponding elements in the multiplied vector as zeros, extra results were computed but finally discarded in the writing stage. Compared to version 3, version 4 successfully earns the 5Gflop/s back and reaches 56 Gflop/s at size 512.

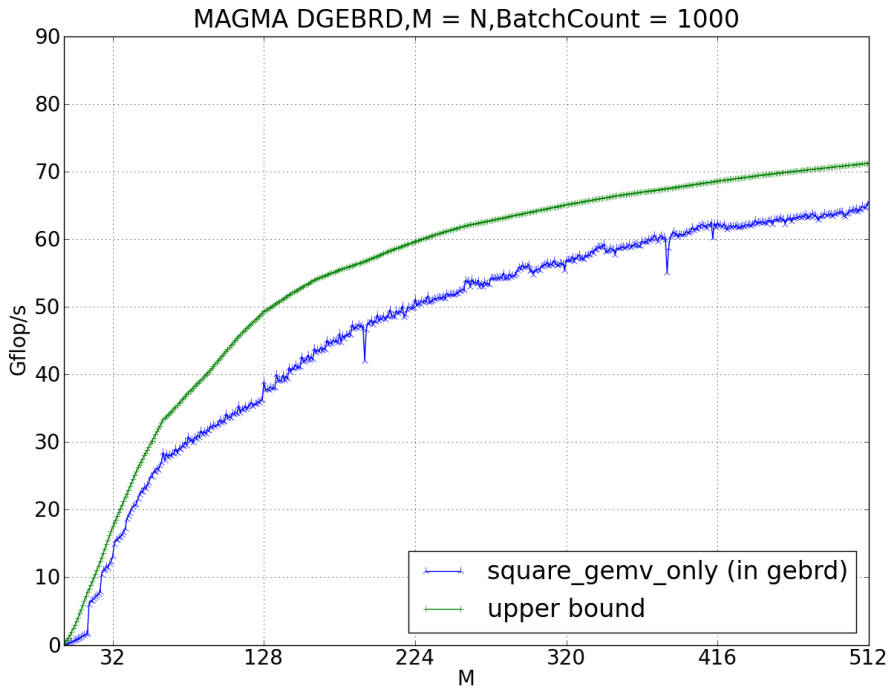

Figure 4.7: A constant gap exists between the GEMV of the square matrix and the upper bound of GEBRD before optimization on the memory mis-aligned problem.

A breakdown of different components contributing to the overall time is depicted in Figure 4.9. As the matrix size ($M = N$) increases, the computation time of the square matrix (in blue) begins to dominate. At a smaller size, the ratio of the wide/tall matrix (of panel width nb) to the square matrix (of matrix size N) is larger, and, thus, the time of wide/tall matrix (in red) is more prominent. The percentage of the time on GEMM is stable across different sizes. Optimization of batched GEMM is described in Section 3.3.1.

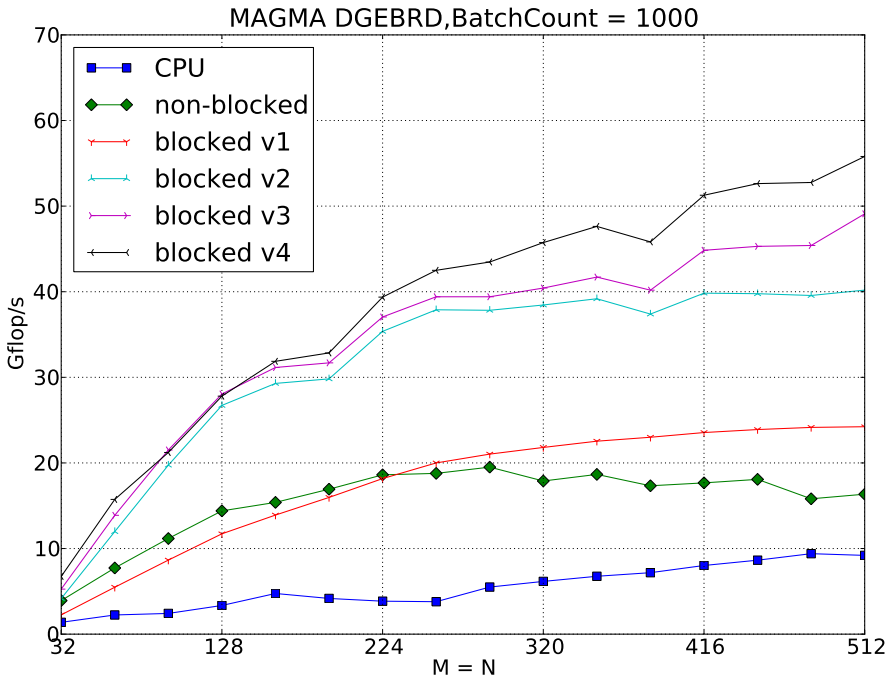

Figure 4.8: Performance progresses of different versions of batched DGEBRD on a K40c GPU.

4.3 Comparison to Multicore CPU Solutions

Here we compare our batched LU to the two CPU implementations proposed in Section 3.1. The simple CPU implementation is to go in a loop style to factorize matrix after matrix, where each factorization is using the multi-thread version of the MKL Library. This implementation is limited regarding performance and does not achieve more than 50 Gflop/s. The main reason for this low performance is the fact that the matrix is small – it does not exhibit parallelism, and so the multithreaded code is not able to feed with workload all 16 threads demand. Hence, we proposed another version of the CPU implementation. Since the matrices are small (< 512) and at least 16 of them fit in the L3 cache level, one of the best technique is to use each thread to factorize a matrix independently. This way 16 factorizations are conducted independently in parallel. We think that this implementation is one of the best

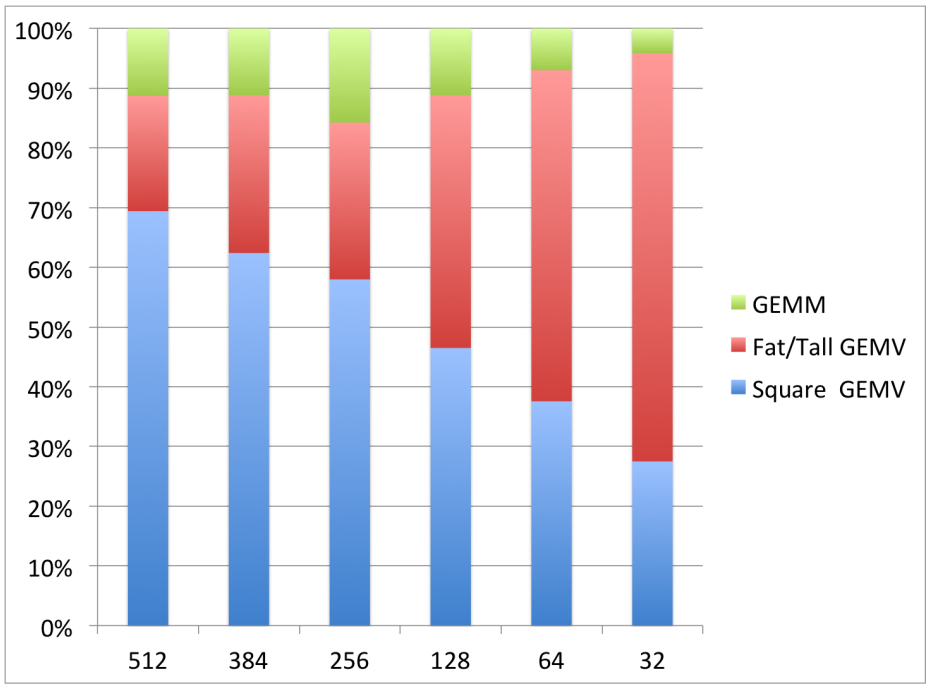

Figure 4.9: A breakdown of the time for different components in batched DGEBRD.

optimized implementations for the CPU. This later implementation is twice faster than the simple implementation. It reaches around 100 Gflop/s in factorizing $2,000$ matrices of size 512×512. Experiments show that our GPU batched LU factorization is able to achieve a speedup of $1.8\times$ compared to the best CPU implementation using 16 Sandy Bridge cores, and $4\times$ than the simple one.

The performances obtained for the batched Cholesky and QR factorizations are similar to the results for LU. A comparison against the two CPU implementations for Cholesky and QR are given in Figures 4.11 and 4.12, respectively. The two CPU implementations behave similarly to the ones for LU. The simple CPU implementation achieves around 60 Gflop/s while the optimized one reaches 100 Gflop/s. Our GPU batched Choleksy yields a speedup of $2\times$ against the best CPU implementation using 16 Sandy Bridge cores.

The simple CPU implementation of the QR decomposition does not exceed 50Gflop/s. The optimized one reaches 100Gflop/s. Despite the CPU's hierarchical memory advantage, our GPU batched implementation is about $1.7\times$ faster.

The optimized CPU implementation of batched TRSV is to use 16 parallel threads with each one calling sequential MKL. The CPU performance is stable and around 10Gflop/s. Our batched TRSV delivers a 3× speedup on the GPU as shown in Figures 4.4 and 4.5.

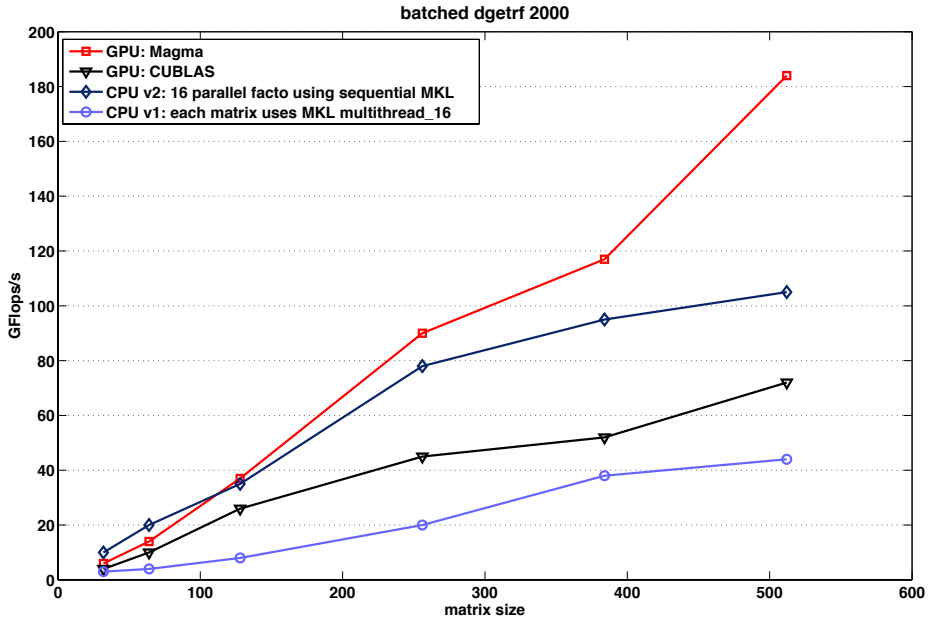

Figure 4.10: Performance in Gflops/s of different versions of the batched LU factorization compared to the CUBLAS implementation for different matrix sizes where $m = n$.

4.4 Power and Energy Consumption

For energy efficiency measurements, we use power and energy estimators built into the modern hardware platforms. In particular, on the tested Intel Xeon CPU E5-2690, we use RAPL (Runtime Average Power Limiting) hardware counters [24, 40]. By the vendor's own admission, the reported power/energy numbers are based on a model tuned to match the actual measurements for various workloads. Given this caveat, we can report that the idle power of the tested Sandy Bridge CPU, running at a fixed frequency of 2600 MHz, consumes about 20 W of power per socket. Batched operations raise the consumption to above 125 W-140 W per socket, and the large

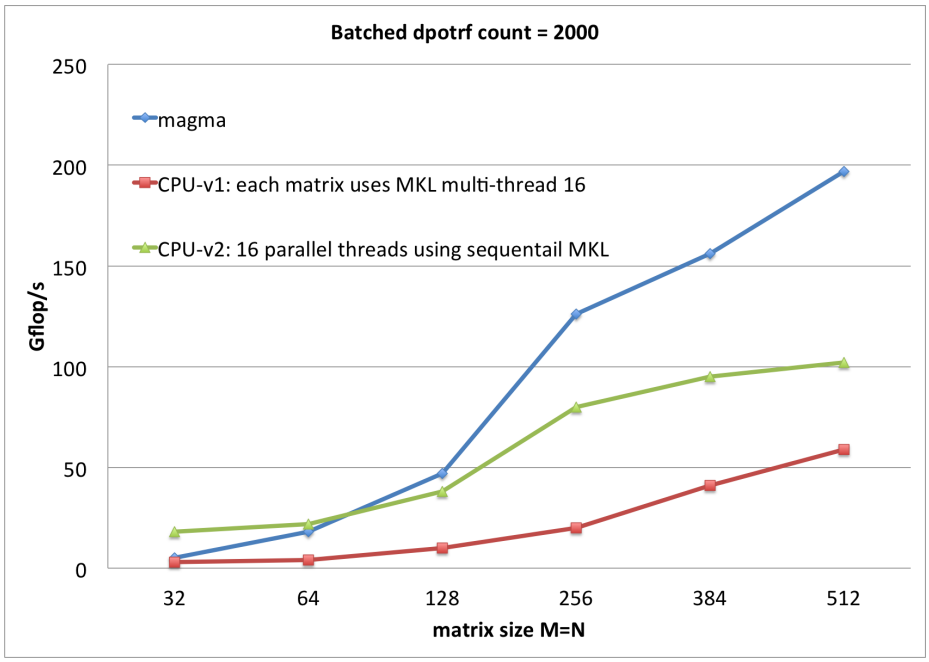

Figure 4.11: Performance in Gflops/s of the GPU compared to the CPU versions of our batched Cholesky decomposition for different matrix sizes where $m = n$

dense matrix operations that reach the highest fraction of the peak performance raise the power draw to about 160 W per socket.

For the GPU measurements, we use NVIDIA's NVML (NVIDIA Management Library) library [32]. NVML provides a C-based programmatic interface to monitor and manage various states within NVIDIA GPUs. On Fermi and Kepler GPUs (like the K40c used), the readings are reported to be within +/-5% accuracy of current power draw. The idle state of the K40c GPU consumes about 20 W. Batched factorizations raise the consumption to about $150 - 180$ W, while large dense matrix operations raise the power draw to about 200 W.

Figure 4.13 depict the comparison of the power consumption of the three implementations of the batched QR decomposition: the best GPU and the two CPU implementations. Here, the batched problem solves $4,000$ matrices of uniform size 512×512. The green curve shows the power required by the simple CPU implementation. In this case, the batched QR proceeds as a loop over the $4,000$ matrices where each matrix is factorized using the multithreaded DGEQRF routine

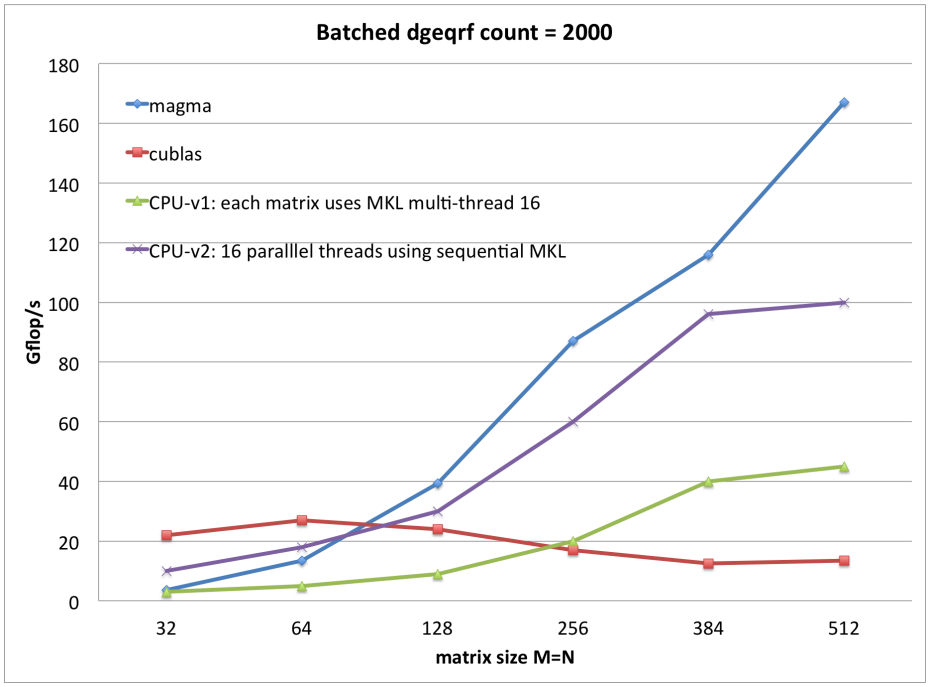

Figure 4.12: Performance in Gflops/s of the GPU compared to the CPU versions of our batched QR decomposition for different matrix sizes where $m = n$.

from the Intel MKL library on the 16 Sandy Bridge cores. The blue curve shows the power required by the optimized CPU implementation. Here, the code proceeds by a sweep of 16 parallel factorizations with each using the sequential DGEQRF routine from the Intel MKL library. The red curve shows the power consumption of our GPU implementation of the batched QR decomposition. The GPU implementation is attractive because it is around 2× faster than the optimized CPU implementation, and moreover it consumes 3× less energy.

According to the power experiments we conduct, we find that the GPU implementations of all the batched one-sided factorizations deliver around a 2× speedup over the best CPU counterpart and are 3× less expensive in term of energy.

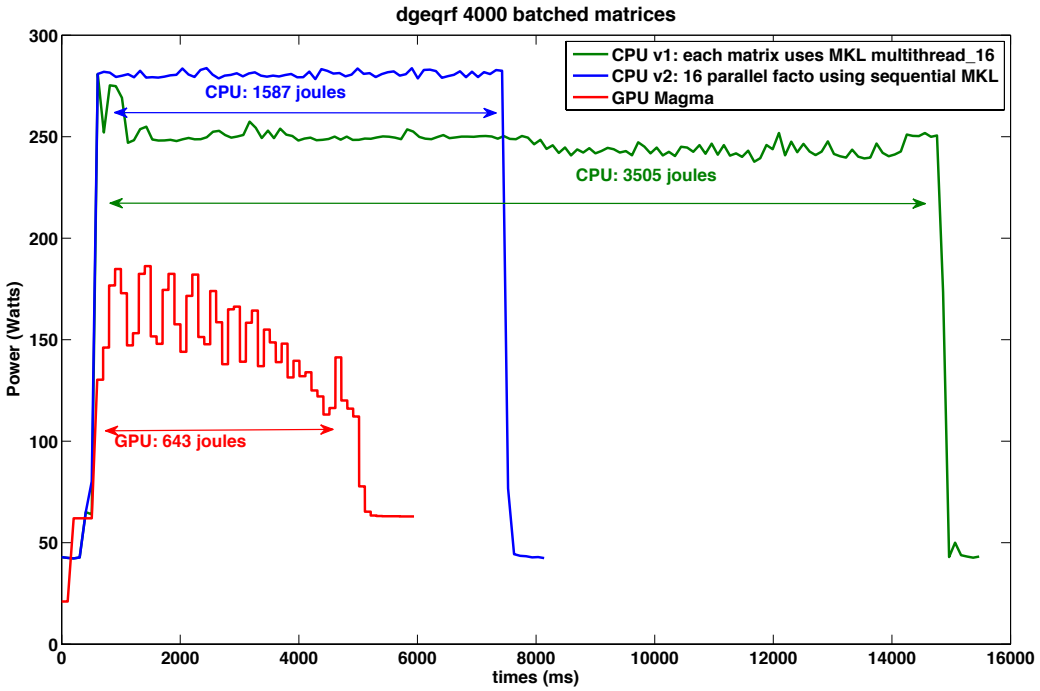

Figure 4.13: Comparison of the power consumption for the QR decomposition of $4,000$ matrices of size 512×512.

Chapter 5

Applications

5.1 The BLAST Algorithm

BLAST is a software package simulating hydrodynamics problems. 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:

$$\text{Momentum Conservation:} \quad \mathbf{M}_{\mathcal{V}} \frac{\mathrm{d}\mathbf{v}}{\mathrm{d}t} = -\mathbf{F} \cdot \mathbf{1}, \tag{5.1}$$

$$\text{Energy Conservation:} \quad \frac{\mathrm{d}\mathbf{e}}{\mathrm{d}t} = \mathbf{M}_{\mathcal{E}}^{-1} \mathbf{F}^T \cdot \mathbf{v}\,, \tag{5.2}$$

$$\text{Equation of Motion:} \quad \frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t} = \mathbf{v}, \tag{5.3}$$

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 (5.1) by using a preconditioned conjugate gradient (PCG) iterative method at each time step. The thermodynamic mass matrix $\mathbf{M}_{\mathcal{E}}$ is

the density weighted inner product of *discontinuous* thermodynamic basis functions and is therefore symmetric and block diagonal, with each block consisting of a local dense matrix. We solve the linear system of (5.2) by pre-computing 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. The rectangular matrix $\mathbf{F}$, called the generalized *force matrix*, depends on the hydrodynamic state $(\mathbf{v}, \mathbf{e}, \mathbf{x})$, and needs to be evaluated at every time step.

The matrix $\mathbf{F}$ 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 $\{\alpha_k\}$:

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

$$\approx \sum_k \alpha_k \hat{\sigma}(\hat{\vec{q}}_k) : J_z^{-1}(\hat{\vec{q}}_k) \hat{\nabla} \hat{\vec{w}}_i(\hat{\vec{q}}_k)\, \hat{\phi}_j(\hat{\vec{q}}_k) |J_z(\hat{\vec{q}}_k)|. \tag{5.4}$$

where, J_z is the Jacobian matrix, and the hat symbol indicates the quantity is on the reference zone. In the CPU code, $\mathbf{F}$ is constructed by two loops: an outer loop over zones (for each z) in the domain and an inner loop over the quadrature points (for each k) in each zone. Each zone and quadrature point compute a component of the corner forces associated with it independently.

A local corner force matrix $\mathbf{F}_z$ can be written as

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

with

$$(\mathbf{A}_z)_{ik} = \alpha_k \hat{\sigma}(\hat{\vec{q}}_k) : J_z^{-1}(\hat{\vec{q}}_k) \hat{\nabla} \hat{\vec{w}}_i(\hat{\vec{q}}_k)\, |J_z(\hat{\vec{q}}_k)|, where J_z = \hat{\nabla} \Phi_z \tag{5.5}$$

and

$$(\mathbf{B})_{jk} = \hat{\phi}_j(\hat{\vec{q}}_k)\,. \tag{5.6}$$

The matrix $\mathbf{B}$ contains the values of the thermodynamic basis functions sampled at quadrature points on the reference element $\hat{\phi}_j(\hat{\vec{q}}_k)$ and is of dimension number of thermodynamic basis functions by number of quadrature points. Matrix $\mathbf{B}$ is constant with time steps. Matrix $\mathbf{A}_z$ contains the values of the gradient of the kinematic basis functions sampled at quadrature points on the reference element $\hat{\nabla}\hat{\vec{w}}_i(\hat{\vec{q}}_k)$ and is of dimension number of kinematic basis functions by number of quadrature points. Matrix $\mathbf{A}_z$ depends on the geometry of the current zone, 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 $\hat{\sigma}(\hat{\vec{q}}_k)$ requires evaluation at each time step and is rich in FLOPs including singular value decomposition (SVD), eigenvalue, eigenvector, equation of state (EOS) evaluations, at each quadrature point (see [12] for more details).

A finite element solution is specified by the order of the kinematic and thermodynamic bases. In practice, we choose the order of the thermodynamic basis to be one less than the kinematic basis, where a particular method is designated as Q_k-Q_{k-1}, $k \geq 1$, corresponding to a continuous kinematic basis in space Q_k and a discontinuous thermodynamic basis in space Q_{k-1}. High order methods (as illustrated in Figure 5.1) can lead to better numerical approximations at the cost of more basis functions and quadrature points in the evaluation of (5.1). By increasing the order of the finite element method, k, we can arbitrarily increase the floating point intensity of the corner force kernel of (5.1) as well as the overall algorithm of (5.1) - (5.3).

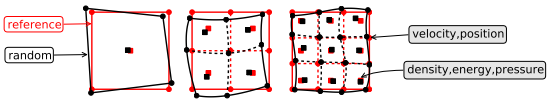

Figure 5.1: Schematic depiction of bilinear (Q_1-Q_0), biquadratic (Q_2-Q_1), and bicubic (Q_3-Q_2) zones.

Here we summarize the basic steps of the BLAST MPI-based parallel algorithm:

1) Read mesh, material properties and input parameters;

2) Partition domain across MPI tasks and refine mesh;

3) Compute initial time step;

4) Loop over zones in the sub-domain of each MPI task:

(4.1)Loop over quadrature points in each zone;

(4.2)Compute corner force associated with each quadrature point and update time step;

5) Find minimum time step and assemble zone contribution to global linear system;

6) Solve global linear system for new accelerations;

7) Update velocities, positions and internal energy;

8) Go to 4 if final time is not yet reached, otherwise exit.

Step 4 is associated with the corner force calculation of (5.1) which is a computational hot spot. Step 6 solves the linear equation of (5.1) with a PCG solver. Table 5.1 shows the timing data for various high order methods in 2D and 3D. Both corner force and CG solver time increase as the order of the method k and dimension increase, but the timing of corner force grows faster than that of CG solver.

Table 5.1: Profile of the BLAST code on the Xeon CPU. The corner force calculation consumes $55\% - 75\%$ of total time. The CG solver takes $20\% - 34\%$.

Method	Corner Force	CG Solver	Total time
2D: Q_4-Q_3	198.6	53.6	262.7
2D: Q_3-Q_2	72.6	26.2	103.7
3D: Q_2-Q_1	90.0	56.7	164.0

5.2 Hybrid Programming Model

Multi-GPU communication relies on CPU-GPU communication on a single node and CPU-CPU communication across nodes. Therefore, a multi-GPU implementation requires CUDA to interact with other CPU programming models like MPI, OpenMP or Pthreads. 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.

5.2.1 CUDA Implementation

We implemented the momentum (5.1) and energy (5.2) equations on the GPU. In the CUDA programming guide [31], the term host is used to refer to CPU and device to GPU. We follow this practice.

CUDA Code Redesign

The CPU code loops over the points in each zone and performs operations on the variables, most of which are represented as matrices. `kernel_loop_quadrature_point` is a kernel to unroll the loop over zones to $\mathbf{A}_z$. The kernel on a Fermi GPU is comparable with a six-core Westmere X5660 CPU in terms of performance. Yet, it is still inefficient and dominated most of the GPU time. We replaced it with six new designed kernels **1-6**. The reformulation of these CUDA kernels is based on the considerations that they can be translated into standard linear algebra routines and thus can be further reused. Except kernel 1-2, the other kernels exhibit standard LAPACK interface and of general purpose. Thus, it is easy for developers to maintain and reuse the code. A major change from the CPU code to our newly designed CUDA code is that loops become batch-processed. Thus, the challenge is to write GPU-efficient massively parallel batched matrix operations.

Kernel 1,2 are used in evaluations of $\hat{\sigma}(\hat{\vec{q}}_k)$, and adjugate of $\mathbf{J}_z$. Independent operations are performed on each quadrature point. Each thread solves SVD, eigenvalue problems for $DIM \times DIM$ matrices by implementing a formula.

Kernel 3,4 evaluate $\hat{\nabla}\hat{\vec{w}}_i(\hat{\vec{q}}_k)$, $\mathbf{J}_z(\hat{\vec{q}}_k)$. This kernel can be expressed by a batched DGEMM $\mathbf{G}_k = \mathbf{D^T}_z\mathbf{W}$, where $\mathbf{D}_z$ is used to define the gradient operator, $\mathbf{W}$ is the basis function.

Kernel 5,6 are batched DGEMM with all matrices size of $DIM \times DIM$. These kernels multiply Jacobin $\mathbf{J}_z$, gradient of basis functions $\hat{\nabla}\hat{\vec{w}}_i$, stress values $\hat{\sigma}$ together.

Kernel 7 performs $\mathbf{F}_z = \mathbf{A}_z\mathbf{B}^\mathbf{T}$, where $\mathbf{A}_z$ is the output of the last kernel. This kernel is a batched DGEMM as well. The batch count is the number of zones.

Kernel 8 and **Kernel 10** compute $-\mathbf{F} \cdot \mathbf{1}$ from (5.1) and $\mathbf{F}^T \cdot \mathbf{v}$ from (5.2), respectively. Each thread block does a matrix-vector multiplication (DGEMV) and computes part of a big vector. The resulting vector is assembled by all thread blocks. The two kernels can be expressed as batched DGEMV.

Kernel 9 is a custom conjugate gradient solver for (5.1) with a diagonal preconditioner (PCG) [30]. It is constructed with CUBLAS/CUSPARSE routines [34].

Kernel 11 is a sparse (CSR) matrix multiplication by calling a CUSPARSE SpMV routine [34]. The reason for calling SpMV routine instead of using a CUDA-PCG solver as in kernel 9 is that the matrix $M_\mathcal{E}$ is a blocked diagonal one as described in Section 5.1. The inverse of $M_\mathcal{E}$ is only computed once at the initialization stage.

Table 5.2: Implementations of the BLAST code on GPUs. Kernel 9 is a set of kernels instead of one single kernel.

No.	Kernel Name	Purpose
1	kernel_CalcAjugate_det	SVD,Eigen,Ajugate
2	kernel_loop_grad_v	EoS, $\hat{\sigma}(\hat{\vec{q}}_k)$
3	kernel_PzVz_Phi_F_Batched	$\hat{\nabla}\hat{\vec{w}}_i(\hat{\vec{q}}_k)$, $\mathbf{J}_z(\hat{\vec{q}}_k)$
4	kernel_Phi_sigma_hat_z	$\hat{\sigma}(\hat{\vec{q}}_k)$
5	kernel_NN_dgemmBatched	Auxiliary
6	kernel_NT_dgemmBatched	Auxiliary
7	kernel_loop_zones	$\mathbf{A}_z\mathbf{B}^\mathbf{T}$
8	kernel_loop_zones_dv_dt	$-\mathbf{F} \cdot \mathbf{1}$
10	kernel_dgemvt	$\mathbf{F}^T \cdot \mathbf{v}$
9	CUDA_PCG	Solve linear system(5.1)
11	SpMV	Solve linear system(5.2)

Memory Transfer and CPU Work

Input vectors $(\mathbf{v}, \mathbf{e}, \mathbf{x})$ are transferred from the host to the device before kernel 1, and output vectors $\frac{\mathrm{d}\mathbf{e}}{\mathrm{d}t}$ are transferred back from the device to the host after kernel 11. Whether vector $\frac{\mathrm{d}\mathbf{v}}{\mathrm{d}t}$ after kernel 9 or vector $-\mathbf{F} \cdot \mathbf{1}$ after kernel 8 is transferred to host depends on turning on/off the CUDA-PCG solver. The time integration of the output right-hand-side vectors in the momentum (5.1) and energy (5.2) equations, together with the motion (5.3) equation are still performed on the CPU to update new velocity, energy and position states $(\mathbf{v}, \mathbf{e}, \mathbf{x})$.

With kernel 8,10, we avoid transferring the full large matrix $\mathbf{F}$, significantly reducing the amount of data transfer between the CPU and GPU via the slow PCI-E bus.

5.2.2 MPI Level Parallelism

The MPI level parallelism in BLAST is based on MFEM, which is a modular C++ finite element library [25]. At the initialization stage (Step 2 in Section 5.1), MFEM takes care of the domain splitting and parallel mesh refinement as shown in Figure 5.2. Each MPI task is assigned a sub-domain consisting of a number of elements (zones). Finite element degrees of freedom (DOFs) shared by multiple MPI tasks are grouped by the set (group) of tasks sharing them and each group is assigned to one of the tasks in the group (the master), see Figure 5.3. This results in a non-overlapping decomposition of the global vectors and matrices and typical FEM and linear algebra operations, such as matrix assembly and matrix-vector product, require communications only within the task groups.

After computing the corner forces, a few other MPI calls are needed to handle the translation between local finite element forms and global matrix / vector forms in MFEM (Step 5 in Section 5.1). An MPI reduction is used to find the global minimum time step.

Because computing the corner forces can be done locally, the MPI parallel layer and the CUDA/OpenMP parallel corner force layer are independent. Each layer can be enabled or disabled independently. However, the kinematic mass matrix $\mathbf{M}_\mathcal{V}$ in (5.1) is global and needs communication across processors, because the kinematic basis is continuous and components from different zones overlap. The modification of MFEM's PCG implementation needed to enable the CUDA-PCG solver to work on multi-GPU is beyond the scope of the present work. With the higher order of the methods, CG time will be less significant compared to the corner force time. Therefore, we only consider the CUDA-PCG solver for (5.1) on a single GPU.

Figure 5.2: Parallel mesh splitting and parallel mesh refinement

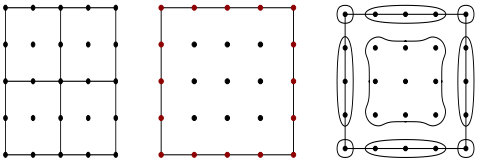

Figure 5.3: Zones assigned to one MPI task and associated Q_2 DOFs (left); the DOFs at the boundary of this subdomain are shared with neighboring tasks (middle); groups of DOFs, including the local group of internal DOFs (right).

5.3 Results and Discussions

For our test cases, we consider the 3D Sedov blast wave problem (see [12] for further details on the nature of these benchmarks). In all cases, we use double precision. The GCC compiler and NVCC compiler under CUDA v5.0 are used for the CPU and GPU codes, respectively.

Table 5.3: Results of CPU and GPU code for a 2D triple-pt problem using a Q_3-Q_2 method; the total energy includes kinetic energy and internal energy. Both CPU and GPU results preserve the total energy to machine precision.

Procs	Kinetic	Internal	Total	Total Change
CPU	5.04235968e-01	9.54576403e+00	1.00500000e+01	-9.2192919e-13
GPU	5.04186180e-01	9.54581381e+00	1.00500000e+01	-4.9382720e-13

5.3.1 Validation of CUDA Code

We get consistent results on the CPU and the GPU. Both the CPU and the GPU code preserved the total energy of each calculation to machine precision, as shown in Table 5.3.

5.3.2 Performance on a Single Node

Due to the new feature Hyper-Q on Kepler GPUs, multiple MPI processes can run on a K20 GPU simultaneously. A K20 GPU can set up to 32 work queues between the host and the device. Each MPI process is assigned to a different hardware work queue and run concurrently on the same GPU.

In our test, the CPU is a 8-core Sandy Bridge E5-2670 and the GPU is a K20. Unless explicitly noted, we always use them to perform our tests in the following sections. In our configuration, 8 MPI tasks share one K20 GPU. Only corner force is accelerated on the GPU. Figure 5.4 shows the speedup achieved by the CPU-GPU over the CPU. We tested two methods Q_2-Q_1 and Q_4-Q_3. When the order is higher, the percentage of the corner force is higher, and the BLAST code enjoys more performance gain from the GPU. GPU speedups Q_2-Q_1 1.9×, but 2.5× for Q_4-Q_3.

5.3.3 Performance on Distributed Systems: Strong and Weak Scalability

We tested our code on the ORNL Titan supercomputer, which has 16 AMD cores and one K20m GPU per node. We scaled it up to 4096 computing nodes. Eight

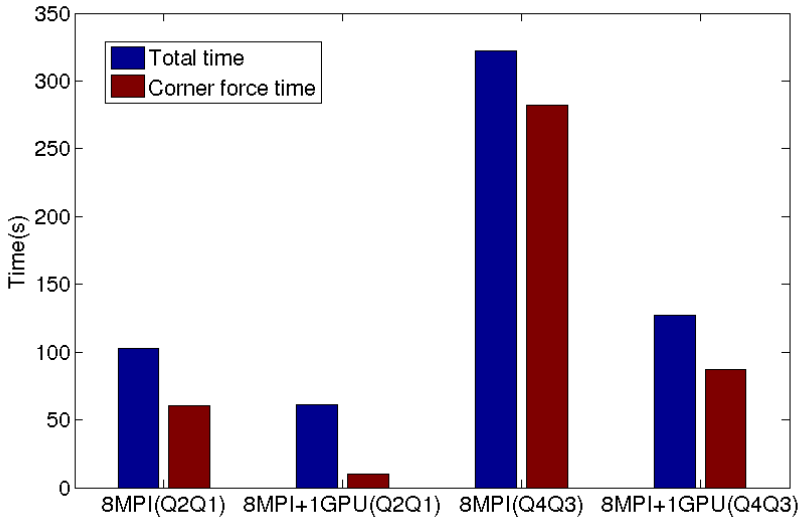

Figure 5.4: Speedups of the CPU-GPU code over the CPU-only code. A 1.9× overall speedup is obtained for the Q_2-Q_1 method and 2.5× is obtained for the Q_4-Q_3.

nodes is the base line. For a 3D problem, one more refinement level increases the domain size 8×. We achieved weak scaling by fixing the domain size 512 for each computing node and increasing 8× more nodes for every refinement step. From 8 nodes to 512 nodes, the curve is almost flat in Figure 5.5. 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 5.1).

We also tested the strong scalability on a small cluster, Shannon machine installed in Sandia national laboratories (SNL). It has 30 computing nodes, with two K20m and two sockets of Intel E5-2670 CPU per node. Figure 5.6 shows the linear strong scaling on this machine. The domain size is 32^3.

5.4 Energy Efficiency

Generally, there are two ways to measure power. The first is attaching an external power meter to the machine. This method is accurate, but it can only measure

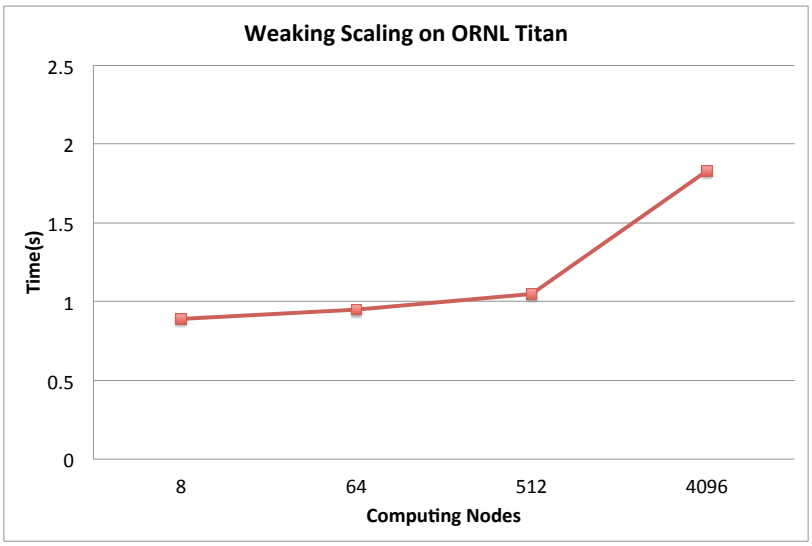

Figure 5.5: Weak scaling of the BLAST code on the Titan supercomputer. The time is of 5 cycles of steps.

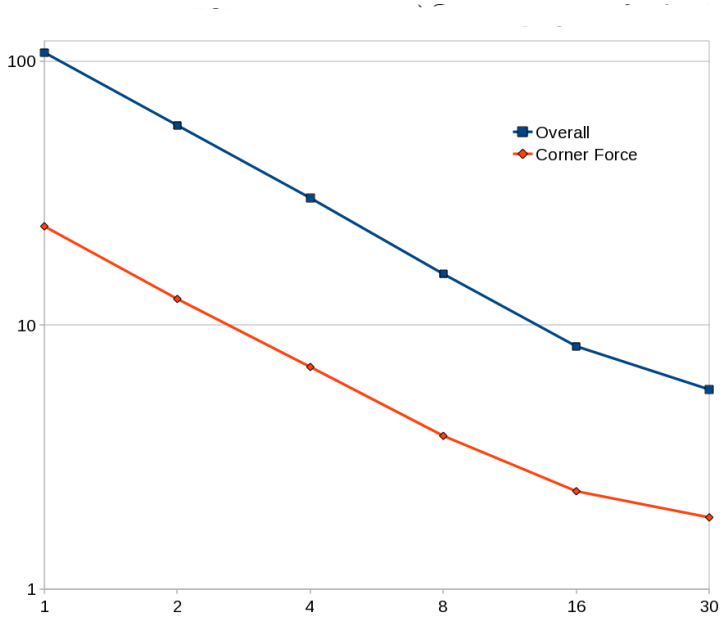

Figure 5.6: Strong scaling. The x-axis is the number of nodes. The y-axis is the logged time.

the power of the whole machine. It is not able to profile power usage of individual processor or memory. The other way is to estimate from the software aspect. We adopt the second way in our measurement.

From Sandy Bridge CPUs, Intel supports onboard power measurement via the Running Average Power Limit (RAPL) [24]. The internal circuitry can estimate current energy usage based on a model accessing Model (or sometimes called Machine) Specific Registers(MSRs), with an update frequency of milliseconds. The power model has been validated by Intel [40] to actual energy.

RAPL provides measurement of the total package domain, the PP0 (Power Plane 0) which refers to the processor cores, and the directly-attached DRAM. Figure 5.7 shows the power of two CPU packages and their DRAM. For comparison purpose, we let one processor to be busy and the other idle. The fully loaded package power is 95W with DRAM at 15W. The idle power is slightly lower than 20W with DRAM almost 0. The test case is a 3D Q_2-Q_1 problem with 8 MPI tasks.

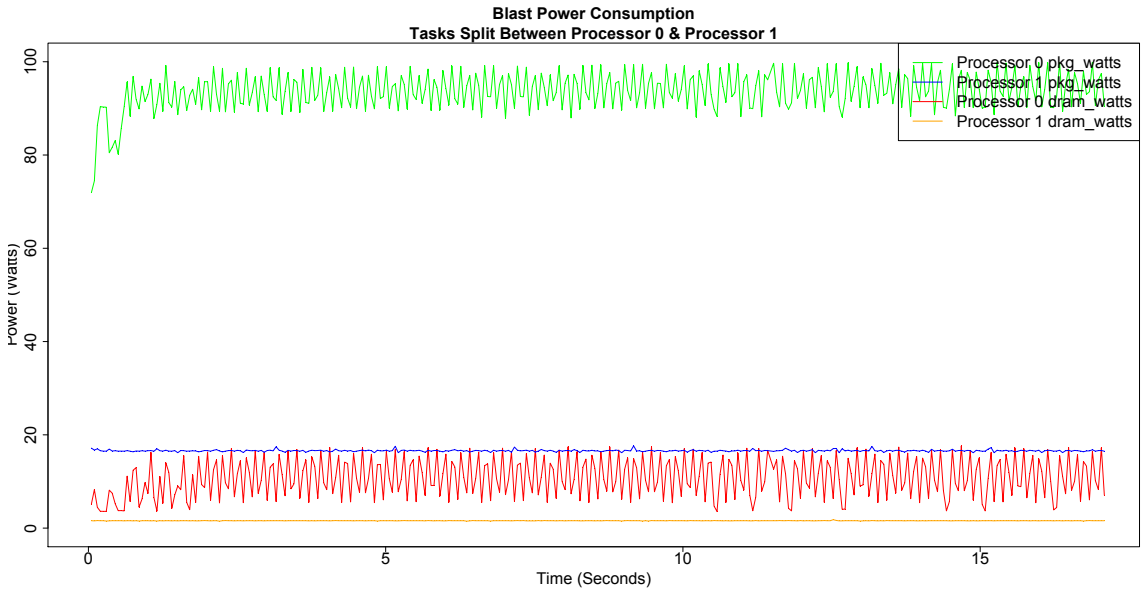

Figure 5.7: Power of two packages of Sandy Bridge CPU. Package 0 is fully loaded. Package 1 is idle.

Recent NVIDIA GPUs support power management via the NVIDIA Management Library (NVML). NVML provides programmers APIs to instrument the code and a high level utility NVIDIA-SMI for users. It only reports the entire board power, including GPU and its associated memory. The update frequency is per millisecond.

Our CUDA kernels' time is around several to tens milliseconds, so the computation will not be missed by NVML.

We test the power of GPU in six scenarios in Figure 5.8. (1,2) Base versus optimized implementation with both corner force and CUDA_PCG solver enabled with one MPI task. The base implementation is the kernel only unrolling the loop. The optimized implementation refers to the redesigned kernels in a batched approach. (3) Optimized corner force (with a Q_2-Q_1 method) with one MPI task. (4,5) Optimized corner force (with Q_2-Q_1 and Q_4-Q_3 methods, respectively) with eight MPI tasks running on the same GPU. (6) CUDA_PCG (Q_2-Q_1) only with one MPI task. The test case is a 3D Sedov problem with the domain size 16^3, which is the maximum size we can allocate with the Q_4-Q_3 method because of the memory limitation of the K20c GPU. The GPU is warmed up by a few runs to reduce noise. Our test shows that the startup power is around 50W by launching any kernel. The idle power is 20W if the GPU is doing nothing for a long time. The TDP of K20c is 225W.

Based and optimized implementation both perform the same FLOPs. The difference between them is the way to exploit the GPU memory hierarchy. The optimized implementation not only runs faster but also lowers the power consumption. The reason is the GPU device memory's power consumption is much higher than that of on-chip memory. In the micro-benchmarks performed in [20], the device memory power is 52 (normalized unit), while shared memory is 1 with FP and ALU only 0.2. Accessing on-chip shared memory can only take 1 cycle while accessing device memory may take 300 cycles [31]. It requires much more energy to drive data across to DRAM and back than to fetch it from on-chip RAM. Because the memory utilization and bandwidth is significantly improved in optimized code, the power consumption is reduced.

When the GPU is shared by eight MPI tasks, its power is higher than one MPI (with the same domain size and problem). We did not find any previous reports

about this situation, but obviously this additional power cost should come from the overhead of Hyper-Q.

The power of CUDA-PCG solver is higher than that of corner force in Figure 5.8. Partly because CG(SpMV) is very memory bound due to its sparse structure, it is very hard for SpMV to achieve comparable memory bandwidth as dense matrices operations in the corner force.

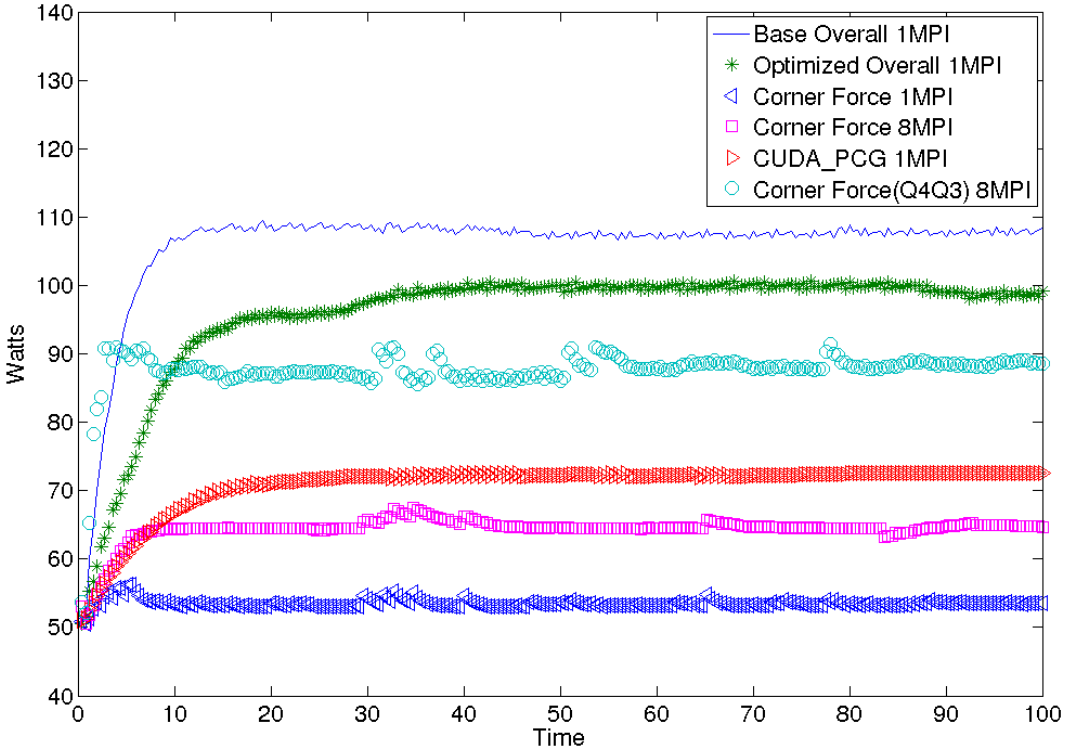

Figure 5.8: Power consumptions of different components on a K20c GPU

Similar to the notion of speedup that is usually used to describe the performance boost, the notion of greenup is used to quantify the energy efficiency [27] [9].

$$
\begin{aligned}
Greenup &= \frac{CPU_{energy}}{(CPU+GPU)_{energy}} = \\
&= \frac{CPU_{power} \cdot CPU_{time}}{(CPU+GPU)_{power} \cdot (CPU+GPU)_{time}} \\
&= \frac{CPU_{power}}{(CPU+GPU)_{power}} \cdot Speedup \\
&= Powerup \cdot Speedup
\end{aligned}
$$

where powerup and speedup are larger than 0. Powerup may be less than 1, since CPU+GPU power may exceed that of CPU only. Yet, the speedup is greater than 1. Therefore the greenup will be larger than 1. Table 5.4 outlines the greenup, powerup and speedup of the BLAST code. The hybrid CPU-GPU solution is greener. It save 27% and 42% of energy, respectively for the two methods, compared to the CPU-only solution. However, it is more than mere energy save. Because the CPU power decreases, the power leakage and failure rate of cores are also reduced. Applications are more fault tolerant and runs faster, so the frequency of checking points can be reduced.

Table 5.4: The CPU-GPU greenup over CPU-only for the BLAST code in 3D Sedov problems.

Method	Power Efficiency	Speedup	Greenup
Q_2-Q_1	0.67	1.9	1.27
Q_4-Q_3	0.57	2.5	1.42

Chapter 6

Conclusions and Future Work

Designing algorithms to work on small problems is a concept that can deliver high performance through an improved data reuse. Many applications have relied on this design concept to get better hardware efficiency, and users have requested it as a supported functionality in linear algebra libraries. We demonstrated how to accomplish this in the case of two classes of batched dense linear algebra problems, one-sided and two-sided factorizations, for GPU architectures.

We showed that small problems can be implemented relatively easily for multicore CPUs, relying on existing high-performance libraries like MKL as building blocks. For GPUs, on the other hand, the development is not straightforward. Our literature review pointed out that the pre-existing solutions were either memory-bound or, even if optimized, did not exceed the corresponding CPU versions' performance. We demonstrated that GPUs, with proper algorithmic enhancements and a batched BLAS approach, can have an advantage over CPUs. Our algorithmic innovations include blocking, variations of blocking like the recursive nested blocking, parallels swapping, regularity of the computation, streaming, and other batched algorithm-specific improvements. Our batched BLAS is characterized by two levels of parallelism: task level parallelism among matrices and fine-grained data parallelism inside each matrix exploiting the underlying SIMT architecture. Our batched

implementations consider the hardware feature and are optimized to take advantage of memory coalescing and alignment to maximize the GPU memory throughput.

GPU Improvements have been observed on large classic numerical algorithms in both dense and sparse linear algebra, where efficient GPU implementations are relatively easy. We demonstrated that GPUs can be taken advantage of for this workload as well. In particular, we achieved 1.8× to 3× speedups compared to our optimized CPU implementations for batched one-sided factorizations and the triangular solve. We also compared our results with NVIDIA CUBLAS, where they have corresponding routines. Our implementations achieved up to 2.5×, 12× and 2.8× speedups for batched LU, QR factorization and the triangular solve, respectively. For a memory-bound Householder bi-diagonalization, we achieved 56Gflop/s, 80% of the theoretical performance bounded by matrix-vector multiplications on a K40c GPU. Furthermore, we redesigned a real world hydrodynamic application with the batched methodology onto CPU-GPU systems. We achieved good strong scaling on a local computing cluster. Weak scaling is achieved up to 4096 computing nodes on the ORNL Titan Supercomputer.

We envision that users will further demand batched availability in high-performance numerical libraries and that batched solvers will become a standard feature in libraries for new architectures. We released and maintained this new functionality through the MAGMA library for NVIDIA GPU accelerators. Our plan is to extend it onto Intel Xeon Phi coprocessors and AMD GPUs based on OpenCL.

The batched is a total GPU implementation. It can have a performance advantage over hybrid implementations where the host CPU is much slower than the accelerator in future systems. For example, in mobile devices featuring ARM CPUs enhanced with GPUs, the total GPU implementations have significant advantages in both energy consumption and performance [19].

Bibliography

[1] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Hatem Ltaief, Raymond Namyst, Samuel Thibault, and Stanimire Tomov. Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs. In Wen mei W. Hwu, editor, *GPU Computing Gems*, volume 2. Morgan Kaufmann, September 2010. 6

[2] ACML - AMD Core Math Library, 2014. Available at http://developer.amd.com/tools-and-sdks/cpu-development/amd-core-math-library-acml. 5

[3] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM. 57

[4] Edward Anderson, Zhaojun Bai, Christian Bischof, Suzan L. Blackford, James W. Demmel, Jack J. Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven J. Hammarling, Alan McKenney, and Danny C. Sorensen. *LAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, Third edition, 1999. 10, 23

[5] M.J. Anderson, D. Sheffield, and K. Keutzer. A predictive model for solving small linear algebra problems in gpu registers. In *IEEE 26th International Parallel Distributed Processing Symposium (IPDPS)*, 2012. 1

[6] Jesse L. Barlow, Nela Bosner, and Zlatko Drma? A new stable bidiagonal reduction algorithm. *Linear Algebra and its Applications*, 397:35 – 84, 2005. 7

[7] http://icl.cs.utk.edu/beast/overview/index.html. 37

[8] Susan Blackford and Jack J. Dongarra. Installation guide for LAPACK. Technical Report 41, LAPACK Working Note, June 1999. originally released March 1992. 12

[9] JeeWhan Choi and Richard W. Vuduc. How much (execution) time and energy does my algorithm cost? *ACM Crossroads*, 19(3):49–51, 2013. 83

[10] NVIDIA Corporation. https://devtalk.nvidia.com/default/topic/527289/help-with-gpu-cholesky-factorization-/. 2

[11] Du Croz, Jack J. Dongarra, and N. J. Higham. Stability of methods for matrix inversion. *IMA J. Numer. Anal.*, 12(119), 1992. 28

[12] Veselin Dobrev, Tzanio V. Kolev, and Robert N. Rieben. High-order curvilinear finite element methods for lagrangian hydrodynamics. *SIAM J. Scientific Computing*, 34(5), 2012. 72, 77

[13] Tingxing Dong, Veselin Dobrev, Tzanio Kolev, Robert Rieben, Stanimire Tomov, and Jack Dongarra. A step towards energy efficient computing: Redesigning a hydrodynamic application on CPU-GPU. In *IEEE 28th International Parallel Distributed Processing Symposium (IPDPS)*, 2014. 1, 15

[14] J. Dongarra, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, and A. YarKhan. Model-driven one-sided factorizations on multicore accelerated systems. *International Journal on Supercomputing Frontiers and Innovations*, 1(1), June 2014. 6

[15] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming. *Parallel Comput.*, 38(8):391–407, August 2012. 56

[16] K. Gallivan, W. Jalby, and U. Meier. The use of BLAS3 in linear algebra on a parallel processor with a hierarchical memory. *SIAM J. Sci. Stat. Comp.*, 8, 1987. 10791084. 11

[17] G. Golub and W. Kahan. Calculating the singular values and pseudo-inverse of a matrix. 1965. 15

[18] Green500, 2013. Available at http://www.green500.org,2013. 2

[19] Azzam Haidar, Stanimire Tomov, Piotr Luszczek, and Jack Dongarra. Magma embedded: Towards a dense linear algebra library for energy efficient extreme computing. *2015 IEEE High Performance Extreme Computing Conference (HPEC 2015)*, 2015. 86

[20] Sunpyo Hong and Hyesoon Kim. An integrated gpu power and performance model. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 280–289, New York, NY, USA, 2010. ACM. 82

[21] Matrix algebra on GPU and multicore architectures (MAGMA), 2014. Available at http://icl.cs.utk.edu/magma/. 19

[22] Intel Pentium III Processor - Small Matrix Library, 1999. Available at http://www.intel.com/design/pentiumiii/sml/. 5

[23] Intel Math Kernel Library, 2014. Available at http://software.intel.com/intel-mkl/. 5, 53

[24] Intel® 64 and IA-32 architectures software developer's manual, July 20 2014. Available at http://download.intel.com/products/processor/manual/. 66, 81

[25] Mfem. Available at http://mfem.googlecode.com/. 76

[26] Hatem Ltaief, Piotr Luszczek, and Jack J. Dongarra. High performance bidiagonal reduction using tile algorithms on homogeneous multicore architectures. *ACM Transactions on Mathematical Software*, 39(3):16:1–16:22, May 2013. 7, 15

[27] Dimitar Lukarski and Tobias Skoglund. A priori power estimation of linear solvers on multi-core processors, 2013. 83

[28] O.E.B. Messer, J.A. Harris, S. Parete-Koon, and M.A. 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. 1

[29] J.M. Molero, E.M. Garzón, I. García, E.S. Quintana-Ortí, and A. Plaza. Poster: A batched Cholesky solver for local RX anomaly detection on GPUs, 2013. PUMPS. 2

[30] M. Naumov. Incomplete-lu and cholesky preconditioned iterative methods using cusparse and cublas. 2011. 75

[31] Cuda programming guide v5.0. http://docs.nvidia.com/cuda/cuda-c-programming-guide/. 4, 74, 82

[32] Available at https://developer.nvidia.com/nvidia-management-library-nvml, 2014. 67

[33] CUBLAS, 2014. Available at http://docs.nvidia.com/cuda/cublas/. 39, 54

[34] Cusparse, 2014. Available at http://docs.nvidia.com/cuda/cusparse/. 75

[35] The OpenACC™ application programming interface version 1.0, November 2011. 5

[36] OpenMP application program interface, July 2013. Version 4.0. 5

[37] Villa Oreste, Massimiliano Fatica, Nitin A. Gawande, and Antonino 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. 6

[38] Villa Oreste, Nitin A. Gawande, and Antonino Tumeo. Accelerating subsurface transport simulation on heterogeneous clusters. In *IEEE International*

Conference on Cluster Computing (CLUSTER 2013), Indianapolis, Indiana, September, 23-27 2013. 6

[39] Rui Ralha. One-sided reduction to bidiagonal form. *Linear Algebra and its Applications*, 358(1?3):219 – 238, 2003. 7

[40] Efraim Rotem, Alon Naveh, Doron Rajwan, Avinash Ananthakrishnan, and Eliezer Weissmann. Power-management architecture of the intel microarchitecture code-named sandy bridge. *IEEE Micro*, 32(2):20–27, March/April 2012. ISSN: 0272-1732, 10.1109/MM.2012.12. 66, 81

[41] Boguslaw Rymut and Bogdan 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. 24

[42] Stanimire Tomov, Rajib Nath, and Jack Dongarra. Dense linear algebra solvers for multicore with GPU accelerators. In *Proc. of the IEEE IPDPS'10*, Atlanta, GA, April 19-23 2014. 5

[43] Top500, 2013. Available at `http://www.top500.org,2013`. 2

[44] Vasily Volkov and James W. Demmel. LU, QR and Cholesky factorizations using vector capabilities of GPUs. Technical Report UCB/EECS-2008-49, University of California, Berkeley, May 13 2008. Also available as LAPACK Working Note 202. 6, 28

[45] Ian Wainwright. Optimized LU-decomposition with full pivot for small batched matrices, April, 2013. GTC'13 – ID S3069. 6

[46] Qian Wang, Xianyi Zhang, Yunquan Zhang, and Qing 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. 37

Appendix

Appendix A

Floating-Point Operation Counts

Floating-point operations counts of Level 2, Level 3 BLAS routines and LAPACK routines discussed in this article are shown in Table A.1. For simplicity, we consider the matrix size as n. Level 1 BLAS routines perform scalar, vector and vector-vector operations. Level 2 BLAS routines perform matrix-vector operations. Level 2 BLAS routines take $O(n^2)$ input data and perform $O(n^2)$ operations. Level 1 and Level 2 BLAS do not scale with number of cores and are limited by the bandwidth. Level 3 BLAS perform matrix-matrix operations and are compute intensive routines. They take $O(n^2)$ input data and perform $O(n^3)$ operations.

Table A.1: Floating-point operation counts of related BLAS and LAPACK routines.

	Multiplications	Additions	Total	Elements Accessed
Level 2 BLAS				
GEMV	mn	mn	$2mn$	mn
TRSV	$n(n+1)/2$	$n(n-1)/2$	n^2	$n^2/2$
Level 3 BLAS				
GEMM	mnk	mnk	$2mnk$	$mn+$ $mk+nk$
SYRK	$kn(n+1)/2$	$kn(n+1)/2$	$kn(n+1)$	$nk+n^2/2$
TRSM (L)	$nm(m+1)/2$	$nm(m-1)/2$	nm^2	$mn+m^2/2$
TRSM (R)	$mn(n+1)/2$	$mn(n-1)/2$	mn^2	$mn+n^2/2$
One-sided Factorizations				
POTRF	$1/6n^3+1/2n^2+1/3n$	$1/6n^3-1/6n$	$1/3n^3+1/2n^2+$ $1/6n$	$n^2/2$
GETRF	$1/2mn^2-1/6n^3+$ $1/2mn-1/2n^2+2/3n$	$1/2mn^2-1/6n^3$ $-1/2mn+1/6n$	$mn^2-1/3n^3$ $-1/2n^2+5/6n$	mn
GEQRF ($m >= n$)	$mn^2-1/3n^3+$ $mn+1/2n^2+23/6n$	$mn^2-1/3n^3+$ $1/2mn+6/6n$	$2mn^2-2/3n^3+$ $mn+n^2+14/3n$	mn
GEQRF ($m < n$)	$nm^2-1/3m^3+$ $2nm-1/2m^2+23/6m$	$nm^2-1/3m^3+$ $nm-1/2m^2+5/6m$	$2nm^2-2/3m^3+$ $3nm-m^2+14/3n$	mn
Two-sided Bi-diagonalization				
GEBRD ($m >= n$)	$2mn^2-2/3n^3+$ $2n^2+20/3n$	$2mn^2-2/3n^3+$ $n^2-mn+5/3n$	$4mn^2-4/3n^3+$ $3n^2-mn+25/3n$	mn

Vita

Tingxing Dong was born in Hebi, Henan, China, to the parents of Fu Dong and Xiuqin Wei. He attended Xunxian No.1 High School in Xunxian, China. He obtained a Bachelor degree in Physics from Zhengzhou University in Zhengzhou, Henan in 2007. After finishing his Master degree in Computer Software and Theory from University of Chinese Academy of Sciences in Beijing, China in July 2010, he started to pursue a PhD degree in Computer Science in the University of Tennessee, Knoxville from August 2010.