Performance, Design, and Autotuning of Batched GEMM for GPUs

Ahmad Abdelfattah[1], Azzam Haidar[1], Stanimire Tomov[1], and Jack Dongarra[1 2 3]

[1] Department of Electrical Engineering and Computer Science
University of Tennessee, Knoxville, USA
[2] Oak Ridge National Laboratory, Oak Ridge, USA
[3] University of Manchester, UK
{aahmad2,haidar,tomov,dongarra}@eecs.utk.edu

Keywords: GEMM, Batched GEMM, HPC, GPU computing, Autotuning

Abstract. The general matrix-matrix multiplication (GEMM) is the most important numerical kernel in dense linear algebra. It is the key component for obtaining high performance in most LAPACK routines. As batched computations on relatively small problems continue to gain interest in many scientific applications, there becomes a need to have a high performance GEMM kernel for a batch of small matrices. Such kernel should be well designed and tuned to handle small sizes, and to maintain high performance for realistic test cases found in the higher level LAPACK routines, and scientific computing applications in general.
This paper presents a high performance batched GEMM kernel on Graphics Processing Units (GPUs). We address batched problems with both fixed and variable sizes, and show that specialized GEMM designs and a comprehensive autotuning process are needed to handle problems of small sizes. For most performance test reported in this paper, the proposed kernels outperform state-of-the-art approaches using a K40c GPU.

1 Introduction

Scientific computing applications extract their high-performance (HP) and efficient use of modern computing architecture excessively through fast linear algebra libraries, and most notably the GEMM routine. Indeed, in the area of dense linear algebra (DLA), algorithms are designed as much as possible to use GEMM, e.g., as in the LAPACK library. For example, direct solvers for large dense linear system and least squares problems require $O(n^3)$ floating point operations (flops), of which $O(n^3)$ are in GEMM. Consequently, they run as fast/efficiently as running GEMM. Application areas that rely on DLA, and therefore GEMM, are computational electromagnetics, material science, airflow past wings, fluid flow around ship and other offshore constructions, applications using boundary integral equations, computational statistic, econometrics, control theory, signal processing, curve fitting, and many more. Therefore, even a slight improvement in GEMM, is extremely valuable and has great impact.

Besides the scientific computing areas that directly need large DLA, numerous other applications, e.g., that will normally require sparse linear algebra computations, use domain decomposition type of frameworks where the overall computation tend to be cast in terms of many but small enough problems/tasks to fit into certain levels of the machines' memory hierarchy. Many times it is advantageous to represent these small tasks as DLA problems on small matrices, as in applications such as astrophysics [18], metabolic networks [13], CFD and the resulting PDEs through direct and multifrontal solvers [23], high-order FEM schemes for hydrodynamics [6], direct-iterative preconditioned solvers [11], and some image [19] and signal processing [4]. Moreover, even in the area of DLA itself, large dense matrices can be broken into tiles and the algorithms expressed in terms of small tasks over them [3]. Also note that implementation-wise, large GEMMs are parallelized on current computing architectures, including GPUs, as many small GEMMs. Under these circumstances, the only way to achieve good performance is to find a way to group these small inputs together and run them in large "batches." The most needed and performance-critical kernel here is a batched GEMM [5, 8, 10]. Finally, tensor contractions, used to model multilinear relations in recent areas of high interest like big-data analytics and machine learning, as well as large scale high-order FEM simulations, can also be reduced to batched GEMMs [1].

To address the needs for batched linear algebra on new architectures, as outlined above, we designed high-performance batched GEMM algorithms for GPUs. We consider batched problems with both fixed and variable sizes. While we leverage optimization techniques from the classic GEMM kernel for one multiplication at a time, we also developed a different design scheme for the tuning process that can flexibly select the best performing set of tuning parameters. For variable size problems, we propose new interfaces, as well as techniques, to address the irregularity of the computation. We show that besides the critical for performance algorithmic designs and innovations, a comprehensive autotuning process is needed in order to handle the enormous complexity of tuning all GEMM variants resulting from our designs. The complexity is further exacerbated by targeting problems for entire ranges of small sizes (*vs.* for a few discrete sizes). Using a K40c GPU, the proposed kernels outperform state-of-the-art approaches (e.g. cuBLAS and MKL libraries) in most of the performance tests reported in this work.

2 Related Work

To enable GPUs for a large-scale adoption in the HP scientific computing area, a fast GEMM had to be developed. This became feasible with the introduction of shared memory in the GPUs. While general purpose GPU computing was possible before that, performance was memory bound, as data once read could not be reused in many computations. The availability of shared memory made data reuse possible, and the first compute-bound GEMM for GPUs was developed [22] (in 2008). As the GPUs continued improving, new GEMM

algorithms had to be developed to better use to the evolving architecture, especially its memory hierarchy. In particular, [20] presented a GEMM algorithm and implementation (in MAGMA, later incorporated in cuBLAS) that applied hierarchical communications/blocking on all memory levels available at the time, including a new register blocking. Blocking sizes, along with other performance-critical choices were parametrized and used in autotuning frameworks [16, 14] but improvements were limited to certain, very specific matrix sizes. Coding these multilevel blocking types of algorithms in native machine language was used to overcome some limitations of the CUDA compiler or warp scheduler (or both) to achieve better performance [21]. Similarly, assembly implementations [15, 7] are used today in cuBLAS for Kepler and Maxwell GPUs to obtain higher performance than corresponding CUDA codes.

Besides the batched GEMM in cuBLAS, there have been a number of research papers on batched GEMM, developed as needed for particular applications. For example, a batched GEMM for very small sizes (up to 16) was developed for high-order finite element method (FEM) [12]. Tensor contraction computations for large scale high-order FEM simulations were reduced to batched GEMM [1], obtaining close to peak performance for very small matrices (90+% of a theoretically derived peak) using some of the techniques that we developed and describe in detail here. Matrix exponentiation from the phylogenetics domain was reduced to batched GEMMs on small square matrices [17], obtaining very good performance for fixed sizes (4, 20, and 60) in single precision.

3 Batched GEMM Design and Implementation Details

This section discusses the main design and tuning approaches for batched GEMM kernels that support both fixed and variable sizes. From now on, variable size batched GEMM is abbreviated as *vbatched* GEMM. Our goal is to minimize coding effort and to design one kernel that could be easily adapted for use in both fixed and variable size batched GEMM. We begin by considering only fixed size batched problems. We then discuss the modifications we incorporated to handle a variable size problem at the end of the section.

Routine Interface. Each GEMM in a batch routine has the form of the standard BLAS GEMM:

$$C = \alpha \cdot op(A) \times op(B) + \beta \cdot C.$$

The interface of a batched/vbatched kernel must manage independent multiplications of matrices that are not necessarily stored contiguously in memory. As a result, the batched kernel requires the address of every individual matrix. It also requires the size and the leading dimension of every matrix. While such information can be passed using single integers in the fixed sizes case, arrays of integers are needed for the vbatched problems. Our kernels support multiplications with different values for α and β. We also add an extra input argument batchCount that indicates the number of matrices in the batch. Table 1 summarizes an example of the interface written in the C language for the batched/vbatched DGEMM routine.

Argument	Description	BLAS	Batched	Vbatched
TRANSA	$op(A)$	`char`	`char`	`char`
TRANSB	$op(B)$	`char`	`char`	`char`
M	Rows of $op(A)/C$	`int`	`int`	`int*`
N	Columns of $op(B)/C$	`int`	`int`	`int*`
K	Columns of $op(A)$/rows of $op(B)$	`int`	`int`	`int*`
α	Alpha	`double`	`double*`	`double*`
A	Input matrix	`double*`	`double**`	`double**`
LDA	Leading dimension of A	`int`	`int`	`int*`
B	Input matrix	`double*`	`double**`	`double**`
LDB	Leading dimension of B	`int`	`int`	`int*`
β	Beta	`double`	`double*`	`double*`
C	Input/output matrix	`double*`	`double**`	`double**`
LDC	Leading dimension of C	`int`	`int`	`int*`
batchCount	Number of matrices	`N/A`	`int`	`int`

Table 1: Interface of batched and vbatched matrix multiplication kernel against standard BLAS interface (GEMM: $C = \alpha \cdot op(A) \times op(B) + \beta \cdot C$)

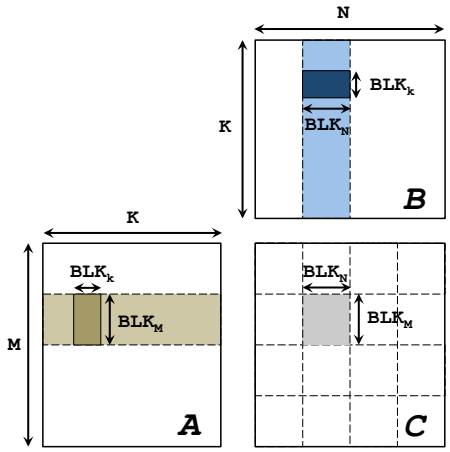


Fig. 1: Example of blocking in the GEMM kernel.

Kernel Design. To design a GEMM kernel in CUDA and take advantage of the available threads, thread blocks and multiprocessors of a GPU, the computation must be partitioned into blocks of threads (also called thread blocks, or simply TBs) that execute independently from each other on the multiprocessors of the GPU. To do that, as shown in Figure 1, the matrix C can be subdivided into rectangular blocks of size $\texttt{BLK}_M \times \texttt{BLK}_N$, and each of these blocks computed by one TB. Specifics on how to do this efficiently, e.g., using hierarchical blocking of both communications and computations, as noted in Section 2, are given in a design by Nath et al. [20], which is also available in the MAGMA library [2]. We use these ideas to build an extended CUDA kernel that is efficient for batched computations (note that the batched GEMM in cuBLAS also uses this early MAGMA GEMM kernel). However, some rules change here in the case of small

matrices. For example, the standard GEMM kernel design tries to maximize the use of shared memory while for batched small GEMM, we should minimize the use of shared memory to allow more than one TB to be executed on the same multiprocessor. The results obtained by our autotuning framework, described below, prove this choice.

The TBs computing a single matrix C can be specified as a 2D grid of size $(\left\lceil \frac{M}{\texttt{BLK}_M} \right\rceil, \left\lceil \frac{N}{\texttt{BLK}_N} \right\rceil)$. A TB processes an entire slice of A and an entire slice of B to perform the necessary multiplication. The reading from global memory is blocked, so that the kernel loads a $\texttt{BLK}_M \times \texttt{BLK}_K$ block of A and a $\texttt{BLK}_K \times \texttt{BLK}_N$ block of B into shared memory, where the multiplication can benefit from the fast shared memory bandwidth. Moreover, a double buffering technique is used to enforce data prefetching into registers, where the computation is additionally blocked. For multiple/batched GEMMs, each C can be computed independently by its 2D grid of TBs, similarly to the standard case. Thus, we design a batched GEMM for a 3D grid of TBs, where one dimension specifies a particular GEMM, and the 2D subgrid specifies the TBs for computing that particular GEMM.

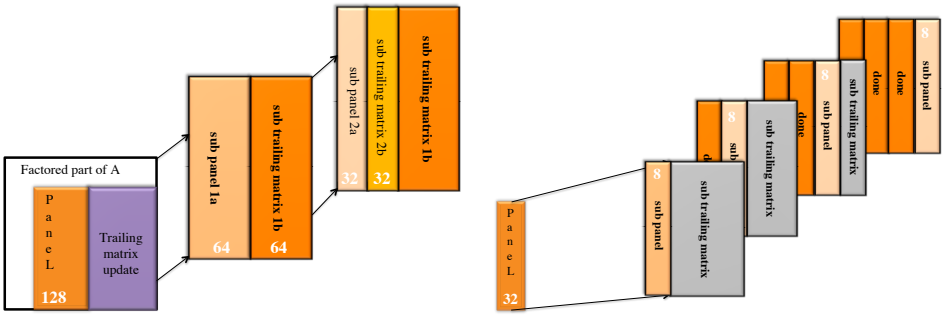
The kernel has many tuning parameters such as the `BLK_M`, `BLK_N`, and `BLK_K` illustrated in Figure 1, `DIM_X` and `DIM_Y` used to configure the number of threads in a TB, among others to specify algorithmic variations. For example, a key distinction with the case of single GEMM is that matrices can be very small, e.g., sub-warp in size. Therefore, instead of having multiple TBs working on a single C matrix, we have parametrized the basic kernel to allow configurations where a TB computes several GEMMs. This design is critical for obtaining close to peak performances for very small sizes [1].

Search Space Generation and Pruning. The MAGMA batched GEMM kernel has a total of 10 tuning parameters, which can produce millions of combinations if we use a brute-force generator. In can be computationally infeasible to search in an enormous design space like this. Therefore, to reduce it, we use generator rules that accept two sets of constraints in order to prune the parameter space. The first set corresponds to the hardware constraints, as defined by the GPU generation and model. Two examples of such constraints are the maximum number of threads in a TB (e.g., $1,024$ for a Kepler GPU), and the amount of shared memory required per TB (48KB). Violation of hardware constraints usually leads to compilation errors or kernel launch failures.

The second set represents soft constraints that rule out kernel instances that are unlikely to achieve good performance for batched workloads. Violation of such constraints can still produce runnable kernels, but they are predictably not good candidates from a performance perspective. Specifying the rules is important in order not to mispredict and consequently rule out good candidates. For example, our experience shows that configurations that use small number of threads per TB and small amounts of shared memory can be very efficient for batched computations. The explanation for this observation is that multiple TBs can run concurrently on the same Streaming Multiprocessor (SM), thus maximizing throughput. Therefore, we consider kernels that use a number of threads as small as 32, and rule out kernels that tend to maximize the occupancy

per TB, e.g., the ones using more than 512 threads per TB. We point out that this is the opposite of a previous work that targeted classic GEMM operations [14], where the soft constraints were set to rule out kernels using less than 512 threads. Our search space generator ended up with 6,400 eligible GEMM kernels.

Test cases. A classical test case for a GEMM kernel is to tune for square matrices which seems to be good choice for other shape for large matrices. However, this scenario rarely appears in higher-level LAPACK routines, such as the LU and QR factorizations, where the multiplication usually involves rectangular matrices (tall-skinny and short-wide matrices), with relatively small values of K compared to M and N. For small matrices computation, K gets even smaller. For example, the batched LU factorization [9] uses a panel of width up tp 128, but it performs the panel factorization recursively as two panels of width 64, each factorized as two panels of width 32. Eventually, each panel of width 32 is factorized as four panels of size 8. Figure 2 shows this recursive nested blocking in the batched LU factorization for small matrices. As a results, in addition to the square sizes, we define our test cases to have discrete small values of K (8, 16, 32, etc.), while varying M and N.



(a) Recursive nested blocking fashion. (b) Classical blocking fashion.

Fig. 2: Recursive nested panel factorization in batched LU

For simplicity, all performance tests are conducted for fixed size batched computations, so that we can specify a winning kernel instance for every tested size. The vbatched GEMM kernel is assumed to have the same tuning decision as the fixed size batched GEMM.

Autotuning output anaylsis. For every test case – specified by precision, transposition mode that we call *shape*, and (M, N, K) sizes – we run all eligible GEMM kernels. We developed an automated selection process that sorts all kernels according to their performances at each data point, and stores the ID of the kernel instance with the best performance. After repeating the same step for all data points, the automated process selects the five (this number can be chosen by the user) most frequent kernel instances that scored the best performance across all data points. We plot also the maximal and the minimal performance obtained by all the kernels at every data point. For a fixed size GEMM: for

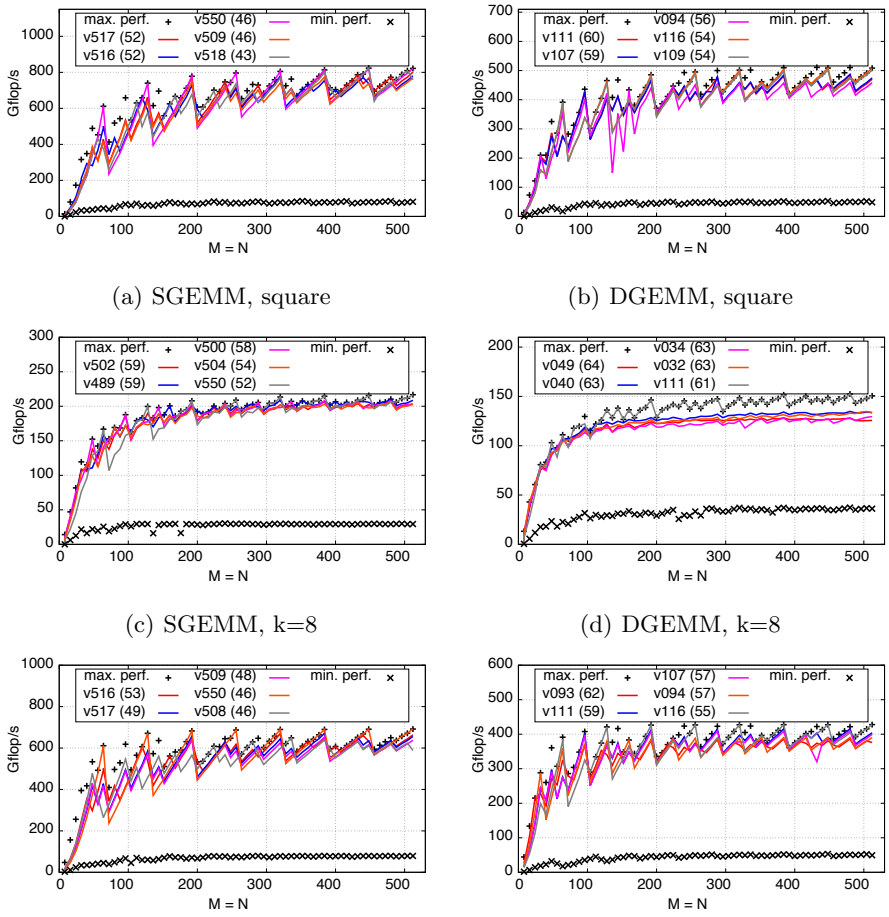
every shape (e.g., NN, NT, etc), every test case (e.g., square, tall-skinny k=8 tall-skinny k=32, wide, etc), one or multiple winning version can be selected such a way to provide the best performance for all the range of sizes. For variable size GEMM: for every shape, we select one winning version that scores a performance within 5-10% of the best achievable performance and that fit all the sizes for a specific test case. The details for these choices are described below.

Performance Sensitivity and Software Framework. Figure 3 shows example performance graphs for some test cases, where the five best performing kernel instances are nominated by our selection process. We observe that not only different test cases have different winning versions, but also a single test case may have two or three winning versions according to the ranges of M and N. Unlike tuning for big matrices [14], which ended up with four kernels across all test cases, we observe that the performance is very sensitive for small matrices that it is required to have an efficient software framework that can call the correct winning version for each test case. Such framework should be able to handle large number of versions while preserving reasonable programming and coding effort. It should also provide an easy-to-modify code structure for future tuning experiments.

Template-based Design. The original tuning of the classic GEMM kernels [20] resulted in finding a few versions, best for different cases. Each version is instantiated in a separate file where the corresponding tuning parameters are listed using compile-time macros (`#define`). This structure is impractical if we have a large number of kernel versions. As an example, assume a kernel that has only five tuning parameters, and is defined in a header file called `gemm.h`. Figures 4 and 5 show an example code for the generation of a single GEMM version (`dgemm_v0`). Another drawback of such design is that a kernel version must have all shapes covered. This is an unnecessary restriction, since we might need more kernels for the `NN` shape than the `NT` shape, for example. It is more flexible to decouple GEMM shapes from each other.

Therefore, we use CUDA C++ templates to enable a unified code base for the batched/vbatched GEMM kernels. Templates enable an easy instantiation of a kernel with a specific precision and tuning parameters. Figure 6 shows an example for the DGEMM routine using templates. Each set of tuning parameters is described as an array of integers. In addition, switching among versions becomes as simple as changing a single number, namely the kernel ID passed to the `instance` macro. The only cost, which is paid once, is the need to generate all possible combinations of tuning parameters using the space generator. Once this step is finished, any future changes to the code in Figure 6 become very simple. As opposed to the previous approach, there is no need to keep the same number of kernels across all shapes, or keep different DGEMM versions in separate files.

Now we describe how we move from the fixed size batched GEMM to the **vbatched GEMM.** There are two main approaches to address a vbatched problem on GPUs. The first assumes that a vbatched kernel is launched directly from the CPU side. Since the launch involves configuration of TBs, the kernel must be configured to accommodate the largest matrix dimensions in the batch.



(a) SGEMM, square

(b) DGEMM, square

(c) SGEMM, k=8

(d) DGEMM, k=8

(e) SGEMM, k=64

(f) DGEMM, k=64

Fig. 3: GEMM performance of the five most frequent, best performing kernels in selected test cases. Each instance is associated with an ID and the number of occurrences. batchCount=500.

As a result, subgrids assigned to smaller matrices will have some threads (or even full TBs) with no work. We developed an *Early Termination Mechanisms (ETMs)* to solve this problem. An ETM is a lightweight software layer that identifies, at the beginning of a kernel launch, threads with no work and immediately terminates them to avoid over-occupancy and memory access violations. ETMs are implemented at the level of a thread, so that each thread can independently determine whether it should proceed with execution or no. Note that such approach requires these maximal dimensions to be known on the CPU side prior to the kernel launch.

```
#define TYPE          double

#define PARAM_NN_0    (2)
#define PARAM_NN_1    (32)
#define PARAM_NN_2    (16)
#define PARAM_NN_4    (8)
#define PARAM_NN_5    (1)

#define PARAM_NT_0    (4)
#define PARAM_NT_1    (16)
#define PARAM_NT_2    (32)
#define PARAM_NT_4    (4)
#define PARAM_NT_5    (0)
/* Parameters of other shapes */

#include "gemm.h"

void dgemm_v0(/* input arguments */)
{
    /* some code */
    if(shape == "nn")
        dgemm_v0_nn(/* arg */)
    else if (shape == "nt")
        dgemm_v0_nt(/* arg */)
    /* other shapes */
}
```

Fig. 4: Generating one version of the DGEMM kernel for all shapes. Each version must be stored in a separate file.

```
void dgemm(/* input arguments */)
{
    if(/* condition 1 */)
        dgemm_v0(/* arg */)
    else if (/* condition 2 */)
        dgemm_v1(/* arg */)
    /*
     Repeat as many times as necessary
    */
}
```

Fig. 5: DGEMM wrapper that calls different DGEMM versions.

The second approach is based on the relatively new CUDA GPUs technology called *dynamic parallelism*. It enables a GPU kernel to launch another GPU kernel. In this case, a vbatched kernel is launched from the GPU side. The CPU role is to launch a *parent kernel* with a total number of CUDA threads equal to

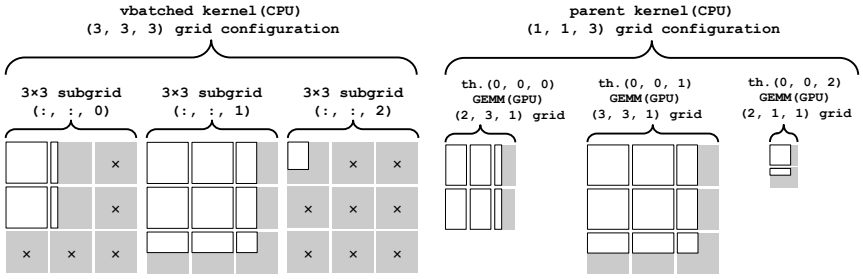
```
#define NN_V_0    2, 32, 16, 8, 1
#define NN_V_1    4,  8, 32, 4, 1

#define NT_V_0    4, 16, 32, 4, 0
#define NT_V_1    8, 24, 16, 2, 1
/* other version definitions */

#define instance(shape,v) shape ## _V_ ## v
#include "gemm_kernel_template.h"

void dgemm(/* input arguments */){
    /* some code */
    switch(shape)
    {
        case "nn":
        if(/* condition nn-1 */)
            gemm_template<double, instance(NN,0)>(/* arg */);
        else if (/* condition nn-2 */)
            gemm_template<double, instance(NN,1)>(/* arg */);
        /* other conditions */
        break;
        case "nt":
        if(/* condition nt-1 */)
            gemm_template<double, instance(NT,0)>(/* arg */);
        else if (/* condition nt-2 */)
            gemm_template<double, instance(NT,1)>(/* arg */);
        /* other conditions */
        break;
        /* Repeat for all shapes */
    }
    /* some code */
}
```

Fig. 6: DGEMM routines using templates with flexible switching.

the number of matrices in the batch. Each CUDA thread then launches a GPU GEMM kernel for one matrix based on its dimensions. As opposed to the first approach, dynamic parallelism waives the need to know the largest dimensions across all matrices. However, it assumes that the underlying CUDA runtime will schedule execution of the *child kernels* efficiently on the GPU, which is not always the case, as described in Section 4. Dynamic parallelism is a technology that is available only on GPUs with compute capability 3.5 (Kepler) or higher.

The vbatched GEMM kernel uses the same code base as the fixed size batched routine, with the use of either ETMs or dynamic parallelism. Examples for both approaches are highlighted in Figure 7. Shown are the output matrices of three independent GEMMs. The first approach (ETMs) requires knowledge about the maximum values of M, N, and K across all matrices. Note that such values do not

necessarily belong to one matrix. Based on these values, it determines the GEMM kernel version to be called. As shown in Figure 7(a), all matrices are processed using a single kernel that is called from the CPU. Each subgrid is responsible for one matrix. All matrices are subdivided using the same blocking size. The ETM layer is responsible for terminating TBs marked by ×, which do not have any work. The second approach, which is based on dynamic parallelism, lets the CPU launch a parent kernel with a number of *master threads*. Each master thread launches a GEMM kernel for its assigned matrix, and it chooses the best working GEMM instance for it. Consequently, this approach allows matrices to be subdivided using different blocking sizes.



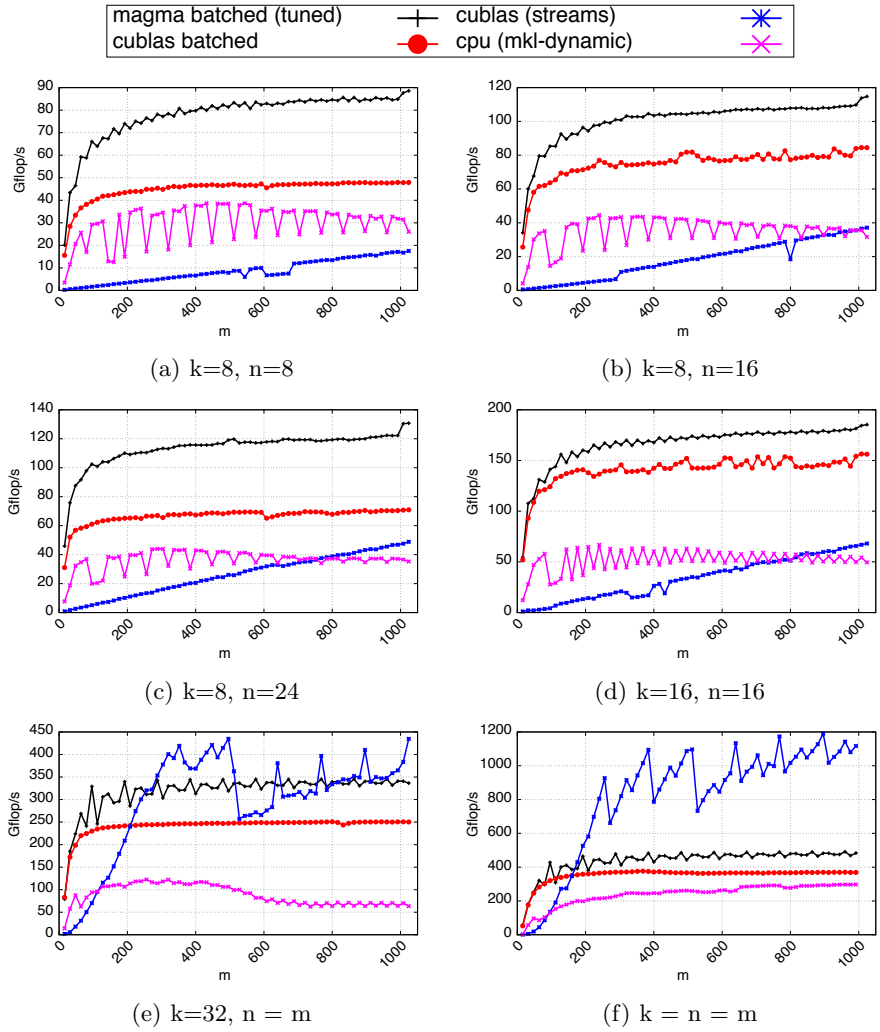
(a) Using max. information with ETMs (b) Using dynamic parallelism

Fig. 7: Approaches for vbatched GEMM.

4 Performance Results and Analysis

System Setup. Performance tests are conducted on a machine equipped with two 8-core Intel Sandy Bridge CPUs (Intel Xeon E5-2670, running at 2.6 GHz), and a Kepler generation GPU (Tesla K40c, running at 745 MHz, with ECC on). CPU performance tests use Intel MKL Library 11.3.0. GPU performance tests use CUDA Toolkit 7.0. Due to space limitations, we show results for double precision only. We point out that the proposed tuned kernels support all other precisions, with roughly similar performance behavior. The performance of the MAGMA GEMM kernel is compared against the cuBLAS batched GEMM kernel, the cuBLAS classic GEMM kernel offloaded to concurrent streams, and the MKL GEMM kernel running on 16 CPU cores. The MKL library is configured to assign one core per matrix at a time, and is used within an OpenMP parallel loop that is dynamically unrolled to balance the workload among cores.

Fixed size. Figure 8 shows the performance for the NN shape, with different problem sizes that are typically used in higher-level factorization and solve algorithms. The tuned MAGMA kernel achieves the best performance when K is small, regardless of M, and N. In Figures 8(a) through 8(d), it scores speedups up to 87%, 38%, 86%, and 26% against the best competitor (cuBLAS batched),



(a) k=8, n=8 (b) k=8, n=16

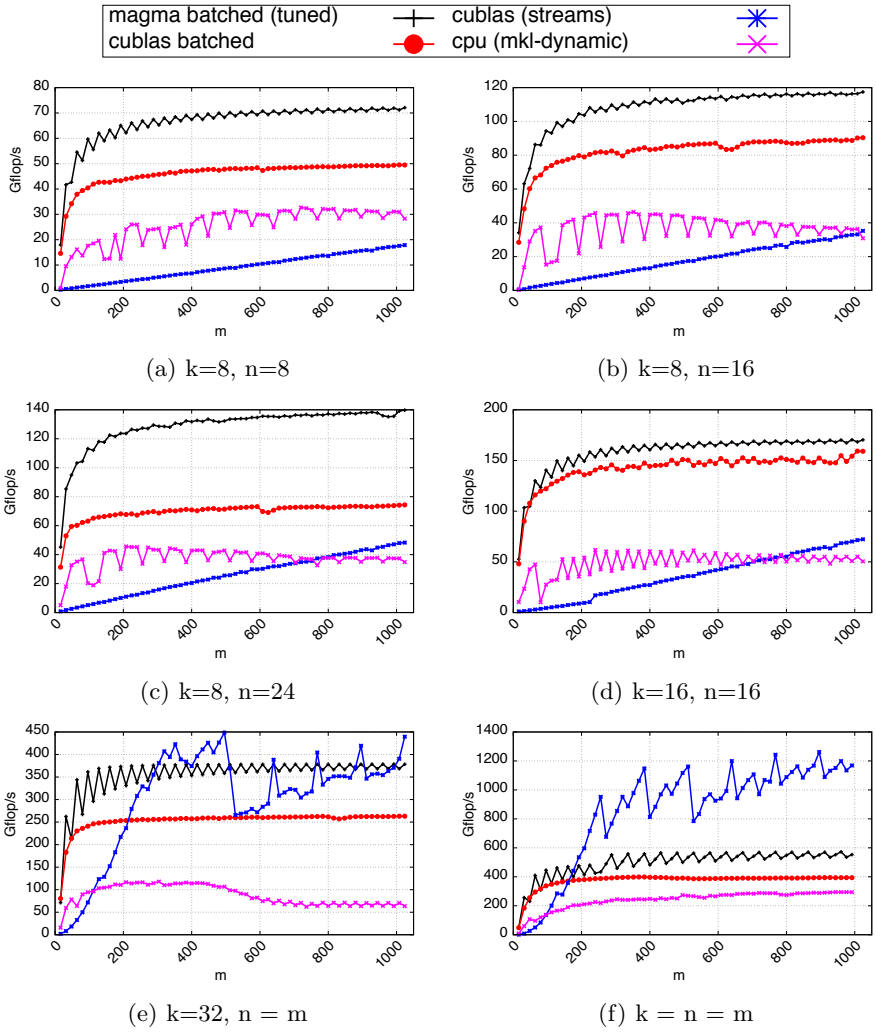
(c) k=8, n=24 (d) k=16, n=16

(e) k=32, n = m (f) k = n = m

Fig. 8: Fixed size batched DGEMM performance for shape NN.

respectively. Starting K = 32, the MAGMA DGEMM kernel loses its advantage to the streamed GEMM, except for the small range of M and N, which is of particular importance for batched computation. In Figures 8(e) and 8(f), MAGMA is generally faster than the batched cuBLAS kernel, achieving up to 43% and 35% speedups, respectively. However, the streamed GEMM, apart from some drops in Figure8(e), becomes the best performing kernel when M and N are around 200.

A similar behavior is observed in Figure 9 for the NT shape. MAGMA scores speedups up to 48%, 39%, 96%, and 16% against the batched cuBLAS kernel, for Figures9(a) through 9(d), respectively. When K gets larger as in Figures 9(e) and 9(f), MAGMA has the advantage for relatively small values of M and N,



(a) k=8, n=8 (b) k=8, n=16

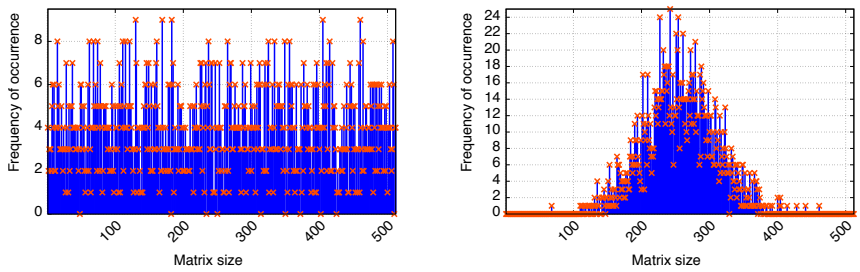
(c) k=8, n=24 (d) k=16, n=16

(e) k=32, n = m (f) k = n = m

Fig. 9: Fixed size batched DGEMM performance for shape NT.

with a 45% speedup against batched cuBLAS for $k = 32$, and a slightly better better performance for the square case. Otherwise, the streamed cuBLAS kernel mostly achieves the best performance, expect for the midrange in Figure 9(e), where MAGMA takes over.

Variable size. Now considering the matrix test suites for the vbatched GEMM, each point M on the x-axis in Figures 11 and 12 represents a distribution of sizes where M is the maximum size in the batch. We did our tests based on two random distributions: uniform and Gaussian. Examples of both distributions are shown in Figure 10. We drop the batched cuBLAS kernel because it does not support variable size computation.



(a) Uniform Distribution (b) Gaussian Distribution

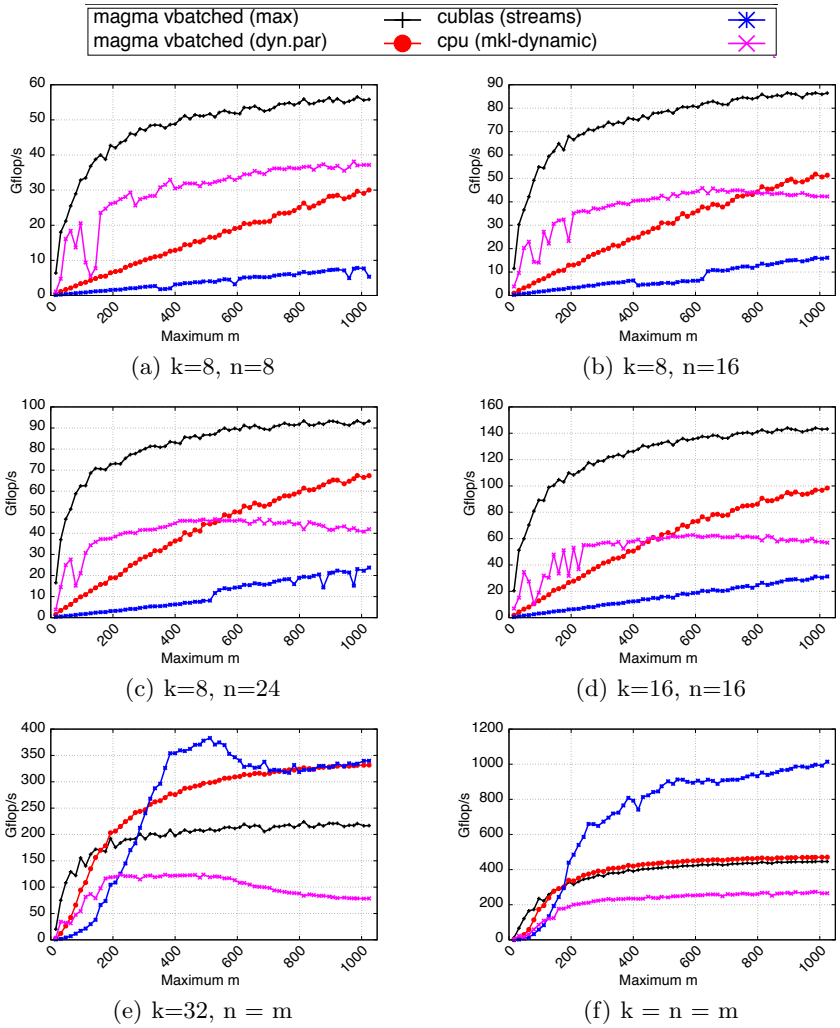
Fig. 10: Histograms of the size distribution for a batch count equal to 2000 with maximum matrix size set to 512

Figure 11 shows the performance for the vbatched DGEMM kernel against a uniform distribution for the NN shape, while Figure 12 considers the NT shape. In both shapes, the MAGMA DGEMM based on ETMs has a clear advantage in Figures 11(a) through 11(d), and 12(a) through 12(d). The MAGMA DGEMM kernel based on dynamic parallelism is either equal to or better than the former approach for relatively large sizes in the cases of $k = 32$ and square matrices. The asymptotic speedups scored by the ETM-based kernel against streamed GEMM/MKL are 6.73×/5.47×, 5.45×/2.18×, 3.75×/10.20×, and 4.34×/11.06× in Figures 11(a) through 11(d), and 8.34×/10.52×, 4.82×/7.86×, 4.20×/9.38×, and 3.80×/9.86× in Figures 12(a) through 12(d), respectively. In Figures 11(e) and 12(e), there is no winning kernel for all sizes. The two MAGMA kernels outperform other competitors for Maximum m up to 300. The streamed GEMM dominates the midrange, and then gets nearly matched or slightly outperformed by the MAGMA kernel based on dynamic parallelism. For the case of square matrices (Figures 11(f) and 12(f)), the streamed GEMM achieves the best performance unless matrices are too small, where the ETM-based MAGMA kernel is the best choice. We observe a similar behavior when we repeat all the above test cases based on the Gaussian distribution. For space limitations, we highlight only two test cases for the NN shape in Figure 13.

Sub-warp sizes. Finally, we want to point out that the framework presented was also used to find batched GEMM kernels for very small (sub-warp in size) matrices. Performance there is memory bound and can be modeled. Results show that we obtain close to peak performance [1] (90+% of the theoretically derived peak) to significantly outperform cuBLAS on GPUs and MKL on CPUs.

5 Conclusion and Future Work

This paper presented a design and autotuning framework for fixed and variable size batched matrix-matrix multiplication using GPUs. Similarly to the GEMM routine, batched GEMMs on small matrices are needed in many applications from big-data analytics to data mining, and more. The work focused on the



(a) k=8, n=8 (b) k=8, n=16

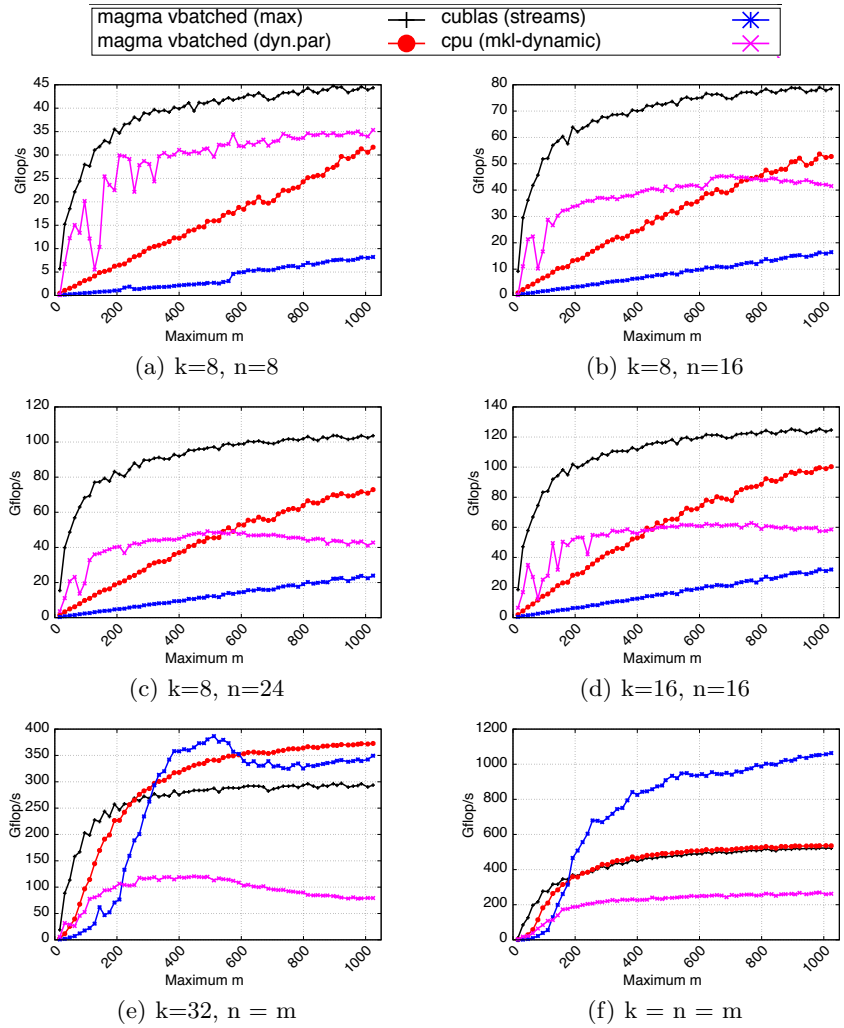
(c) k=8, n=24 (d) k=16, n=16

(e) k=32, n = m (f) k = n = m

Fig. 11: Vbatched DGEMM performance for shape NN with uniform distribution

algorithmic design and performance autotuning for small fixed and variable sizes on test cases found in batched LAPACK factorization and solve algorithms. With a comprehensive autotuning process and a flexible software framework, we are able to find and call the best kernel configuration (within our design space) according to many deciding factors. The flexible software scheme ensures minimal coding effort if future changes are required, and can be used efficiently for other computational kernels that have large number of tuning parameters.

Future directions include adding support for multiplications with different shapes within the same GPU kernel, thorough testing of the vbatched routine against different size distributions, and performance analysis and profiling of the



(a) k=8, n=8 (b) k=8, n=16

(c) k=8, n=24 (d) k=16, n=16

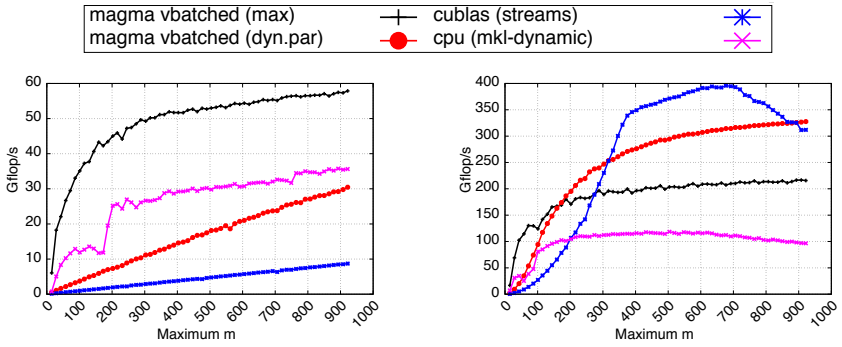
(e) k=32, n = m (f) k = n = m

Fig. 12: Vbatched DGEMM performance for shape NT with uniform distribution

dynamic-parallelism based kernels in order to analyze and understand its behavior and overhead. Work on applying and tuning the batched GEMMs in specific applications, e.g., using application-specific knowledge, especially in computing applications requiring variable sizes like direct multifrontal solvers for sparse matrices, are of high interest and subject to future work.

Acknowledgment

This work is based upon work supported by the National Science Foundation under Grants No. ACI-1339822 and CSR 1514286, NVIDIA, the Department of



(a) k=8, n=8 (b) k=32, n = m

Fig. 13: Vbatched DGEMM performance for shape NN with Gaussian distribution

Energy (LLNL subcontract under DOE contract DE-AC52-07NA27344), and in part by the Russian Scientific Foundation, Agreement N14-11-00190.

References

1. Abdelfattah, A., Baboulin, M., Dobrev, V., Dongarra, J., Earl, C., Falcou, J., Haidar, A., Karlin, I., Kolev, T., Masliah, I., Tomov, S.: High-Performance Tensor Contractions for GPUs. University of Tennessee Computer Science Technical Report (UT-EECS-16-738) (01-2016 2016)
2. Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Ltaief, H., Luszczek, P., Tomov, S.: Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. J. Phys.: Conf. Ser. 180(1) (2009)
3. Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Ltaief, H., Luszczek, P., Tomov, S.: Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. J. Phys.: Conf. Ser. 180(1) (2009)
4. Anderson, M., Sheffield, D., Keutzer, K.: A predictive model for solving small linear algebra problems in gpu registers. In: IEEE 26th International Parallel Distributed Processing Symposium (IPDPS) (2012)
5. Dong, T., Haidar, A., Luszczek, P., Harris, A., Tomov, S., Dongarra, J.: LU Factorization of Small Matrices: Accelerating Batched DGETRF on the GPU. In: Proceedings of 16th IEEE International Conference on High Performance and Communications (HPCC 2014) (August 2014)
6. Dong, T., Dobrev, V., Kolev, T., Rieben, R., Tomov, S., Dongarra, J.: A step towards energy efficient computing: Redesigning a hydrodynamic application on CPU-GPU. In: IEEE 28th International Parallel Distributed Processing Symposium (IPDPS) (2014)
7. Gray, S.: A full walk through of the SGEMM implementation. https://github.com/NervanaSystems/maxas/wiki/SGEMM (2015)
8. Haidar, A., Dong, T., Luszczek, P., Tomov, S., Dongarra, J.: Batched matrix computations on hardware accelerators based on GPUs. International Journal of High Performance Computing Applications doi:10.1177/1094342014567546 (02/2015)
9. Haidar, A., Dong, T., Luszczek, P., Tomov, S., Dongarra, J.: Batched matrix computations on hardware accelerators based on gpus. International Journal of

High Performance Computing Applications (2015), http://hpc.sagepub.com/content/early/2015/02/06/1094342014567546.abstract
10. Haidar, A., Dong, T., Tomov, S., Luszczek, P., Dongarra, J.: A framework for batched and gpu-resident factorization algorithms applied to block householder transformations. In: Kunkel, J.M., Ludwig, T. (eds.) High Performance Computing, Lecture Notes in Computer Science, vol. 9137, pp. 31–47. Springer International Publishing (2015), http://dx.doi.org/10.1007/978-3-319-20119-1_3
11. Im, E.J., Yelick, K., Vuduc, R.: Sparsity: Optimization framework for sparse matrix kernels. Int. J. High Perform. Comput. Appl. 18(1), 135–158 (Feb 2004), http://dx.doi.org/10.1177/1094342004041296
12. Jhurani, C., Mullowney, P.: A GEMM interface and implementation on NVIDIA gpus for multiple small matrices. CoRR abs/1304.7053 (2013), http://arxiv.org/abs/1304.7053
13. Khodayari A., A.R. Zomorrodi, J.L., Maranas, C.: A kinetic model of escherichia coli core metabolism satisfying multiple sets of mutant flux data. Metabolic engineering 25C, 50–62 (2014)
14. Kurzak, J., Tomov, S., Dongarra, J.: Autotuning GEMM kernels for the Fermi GPU. IEEE Transactions on Parallel and Distributed Systems 23(11), 2045–2057 (November 2012)
15. Lai, J., Seznec, A.: Performance upper bound analysis and optimization of sgemm on fermi and kepler gpus. In: Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). pp. 1–10. CGO '13, IEEE Computer Society, Washington, DC, USA (2013), http://dx.doi.org/10.1109/CGO.2013.6494986
16. Li, Y., Dongarra, J., Tomov, S.: A note on auto-tuning GEMM for GPUs. In: Proceedings of the 2009 International Conference on Computational Science, ICCS'09. Springer, Baton Roube, LA (May 25-27 2009)
17. Lopez, M., Horton, M.: Batch matrix exponentiation. In: Kindratenko, V. (ed.) Numerical Computations with GPUs, pp. 45–67. Springer International Publishing (2014), http://dx.doi.org/10.1007/978-3-319-06548-9_3
18. Messer, O., Harris, J., Parete-Koon, S., Chertkow, M.: 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)
19. Molero, J., Garzón, E., García, I., Quintana-Ortí, E., Plaza, A.: Poster: A batched Cholesky solver for local RX anomaly detection on GPUs (2013), PUMPS
20. Nath, R., Tomov, S., Dongarra, J.: An Improved Magma Gemm For Fermi Graphics Processing Units. Int. J. High Perform. Comput. Appl. 24(4), 511–515 (Nov 2010), http://dx.doi.org/10.1177/1094342010385729
21. Tan, G., Li, L., Triechle, S., Phillips, E., Bao, Y., Sun, N.: Fast implementation of dgemm on fermi gpu. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 35:1–35:11. SC '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/2063384.2063431
22. Volkov, V., Demmel, J.: Benchmarking GPUs to tune dense linear algebra. In: SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing. pp. 1–11. IEEE Press, Piscataway, NJ, USA (2008)
23. Yeralan, S.N., Davis, T.A., Ranka, S.: Sparse mulitfrontal QR on the GPU. Tech. rep., University of Florida Technical Report (2013), http://faculty.cse.tamu.edu/davis/publications_files/qrgpu_paper.pdf