Fast Batched Matrix Multiplication for Small Sizes using Half-Precision Arithmetic on GPUs

Ahmad Abdelfattah, Stanimire Tomov
Innovative Computing Laboratory
University of Tennessee, USA
{ahmad,tomov}@icl.utk.edu

Jack Dongarra
University of Tennessee, USA
Oak Ridge National Laboratory, USA
University of Manchester, UK
dongarra@icl.utk.edu

***Abstract*—Matrix multiplication (GEMM) is the most important operation in dense linear algebra. Because it is a compute-bound operation that is rich in data reuse, many applications from different scientific domains cast their most performance-critical stages to use GEMM. With the rise of batch linear algebra, batched GEMM operations have become increasingly popular in domains other than dense linear solvers, such as tensor contractions, sparse direct solvers, and machine learning. In particular for the latter, batched GEMM in reduced precision (i.e., FP16) has been the core operation of many deep learning frameworks.**

This paper introduces an optimized batched GEMM for FP16 arithmetic (HGEMM) using graphics processing units (GPUs). We provide a detailed design strategy that takes advantage of the Tensor Core technology that was recently introduced in CUDA-enabled GPUs. The developed solution uses low-level APIs provided by the vendor in an optimized design that overcomes the limitations imposed by the hardware (in the form of discrete configurations). The outcome is a highly flexible GPU kernel that provides a lot of controls to the developer, despite the aforementioned restrictions. The paper also pays particular attention to multiplications of very small matrices that cannot fully occupy the Tensor Core units. Our results show that the proposed design can outperform the highly optimized vendor routine for sizes up to 100 by factors between $1.2\times$ and $10\times$ using a Tesla V100 GPU. For extremely small matrices, the observed speedups range between $1.8\times$ and $26\times$.

***Index Terms*—Matrix multiplication, batched linear algebra, FP16 arithmetic, GPU computing**

I. Introduction

High-performance linear algebra libraries empower many scientific applications to run efficiently on today's massively parallel architectures. Considering dense linear algebra software, achieving high performance is usually possible through algorithmic designs that express as many computational stages as possible in terms of compute-bound routines in general, and matrix multiplication (GEMM) in particular. The latter combines two properties that make it an extremely important kernel in many high-performance computing (HPC) applications. The first is that it is an embarrassingly parallel operation. The second is its high operational intensity [1], which is defined as the ratio between the amount of floating-point operations (FLOPs) and the number of bytes transferred to/from the main memory. A standard GEMM operation updates a matrix C, where $C_{\mathrm{M}\times\mathrm{N}} = \alpha A_{\mathrm{M}\times\mathrm{K}} \times B_{\mathrm{K}\times\mathrm{N}} + \beta C_{\mathrm{M}\times\mathrm{N}}$. Both α and β are scalars. The total number of FLOPs of this equation is equal to $(2MNK)$. The amount of bytes transferred is equal to $P \times [K(M+N)+2MN]$, where P is the number of bytes required to represent a floating-point number in a specific precision. We consider only half-precision arithmetic in this paper, so $P = 2$. If β is zero, there is no need to read C, so the amount of memory transfer becomes $2 \times [K(M+N)+MN]$, while the amount of FLOPs remains the same. GEMM is a compute-bound kernel, since the FLOP complexity is cubic, while the memory traffic is only quadratic.

Outside the field of dense linear solvers, matrix multiplication can still be important in many scientific domains, where the computational workload can be broken down into a large number of independent GEMM operations of relatively small sizes. The workload is often called a "batch workload," and dedicated routines have been optimized for such workloads (e.g., batched GEMMs). It turns out that the batched GEMM kernel is almost as important as the regular non-batched GEMM, since it has been featured in many applications, such as sparse direct solvers [2], and tensor contractions [3]. Some hardware vendors now provide optimized batched GEMM kernels such as the Intel Math Kernel Library (MKL) library[1] and the NVIDIA cuBLAS library.[2]

Since the booming of machine learning applications and artificial intelligence (AI), there has been an interest in developing high-performance half-precision arithmetic (16-bit floating-point format), since most AI applications do not necessarily require the accuracy of single or double precisions [4]. Half precision also enables machine learning applications to run faster, not only because of the faster arithmetic, but also because of the reduction in memory storage and traffic by a factor of $2\times$ against single precision, and by a factor of $4\times$ against double precision. NVIDIA's graphics processing units (GPUs) introduced half-precision arithmetic with the Pascal architecture. They implement the "binary16" format which is defined by the IEEE-754 standard [5]. While the Pascal GPU architecture introduced hardware support for FP16 arithmetic, the Volta architecture, which powers the Summit supercomputer,[3] comes with hardware acceleration units (called Tensor Cores) for matrix multiplication in FP16. These Tensor Cores

[1] https://software.intel.com/mkl
[2] https://developer.nvidia.com/cublas
[3] https://www.olcf.ornl.gov/summit/

are theoretically $12\times$ faster than the theoretical FP16 peak performance of the preceding architecture. Applications taking advantage of the Tensor Cores can run up to $4\times$ faster than using the regular FP16 arithmetic on the same GPU.

In this paper, we investigate the use of the Tensor Cores to provide a general purpose batched matrix multiplication in FP16 arithmetic (batched HGEMM). The paper addresses some challenges in using Tensor Cores programmatically in a GPU kernel, such as discrete sizes and restricted thread configurations. The proposed design allows a highly flexible and robust implementation that takes advantage of the hardware acceleration while mitigating the limitations imposed by the hardware. We prioritize design flexibility in order to withstand potential future changes to such a recent technology. The developed solution is equipped with many tuning parameters that control different aspects of the kernel. The paper presents a tuning process that exposes a tradeoff between performance tolerance and the number of compiled kernel instances. The paper also thoroughly investigates batched HGEMM on extremely small matrices that cannot fill the Tensor Cores with enough work. While the vendor routine is very optimized for relatively large sizes, we observe that the developed kernel outperforms cuBLAS for sizes less than 100 with speedups that range between $1.2\times$ and $10\times$. The range of speedups for very small sizes is even larger, ranging between $1.8\times$ and $26\times$.

II. Related Work

GPUs are throughput-oriented processors capable of delivering very high performance in tasks with high degrees of parallelsim. Since GEMM is a very good example of such tasks, GPUs have become very popular in the field of dense linear solvers. Research efforts go back almost a decade ago, when GPUs started to have programmable shared memories (i.e., user-controlled caches). This enabled researchers to develop the first compute-bound GEMM on GPUs [6]. Since then, the GEMM kernel has been subject to continuous improvements, like register and shared memory blocking and prefetching [7]. Such developments sparked many efforts in providing fast high-level dense linear solvers on GPUs, such as the MAGMA library [8], ViennaCL [9], and Chameleon [10]. Performance portability of GEMM was achieved through performance-critical tuning parameters that control different aspects of the GEMM design [11] [12]. Following the publicly available developments from the research community, the GPU vendor started providing highly optimized GEMM implementations that are written in a low-level language [13] in order to overcome some limitations imposed by the compiler and the hardware scheduler. Similarly, assembly implementations [14], [15] are available today in the cuBLAS library, with the ability to achieve a performance that is very close to the GPU theoretical peak. Similar to the libraries mentioned above, the vendor also provides a library called cuSOLVER[4] for high-level dense linear algebra algorithms.

[4]https://developer.nvidia.com/cusolver

All the aforementioned efforts address the problem of one GEMM operation that is relatively large enough to provide enough parallel work for the GPU. The recent application-driven interests in batch linear solvers have encouraged vendors and library developers to design dedicated routines that can address a large number of small matrix problems. Algorithmically, a batched GEMM is still a very important operation, since it remains the performance key to higher-level algorithms such as the batched one-sided factorizations [16]. However, the importance of the batched GEMM goes beyond the boundaries of dense linear algebra to affect other scientific domains, such as sparse direct solvers [2], tensor contractions [17] [3], and machine learning [18]. The challenges in optimizing batched GEMM are different from the regular GEMM kernel. As the problem sizes are relatively smaller, the GEMM operations are no longer compute-bound, and more attention should be paid to optimizing the memory traffic. Automatic performance tuning is even more important in batch routines, since it has been found that the performance is more sensitive to tuning parameters in small matrix problems [19].

Batched GEMM operations are crucial to machine learning applications in particular. For example, convolutional neural networks (CNNs) are a very popular class of deep neural networks. They were initially implemented using custom dense kernels, as originally done in Caffe [20] and other libraries, such as tensor convolutions and activation functions. These custom kernels were developed locally per package. And since they dominate the training time for CNNs, re-optimizations had to be done whenever the underlying architecture changed. This is why research efforts, such as cuDNN [18], MagmaDNN [21], and others, focused on providing optimized primitives for deep learning, similar to the way BLAS provides optimized primitives to LAPACK algorithms. The most important operation in CNNs is batched spatial convolution, which can be cast into batched matrix multiplication [22] [18]. In addition, the work done in [23] uses batched GEMMs of very small sizes (3×3) to implement fast CNN algorithms based on minimal filtering algorithms [24]. On another front, the batched GEMM operations in machine learning are not necessarily required to have the accuracy of single or double precisions. In fact, it has been shown that lower precisions are enough for training deep neural networks [4]. Furthermore, the need for extreme computational power in DNNs arises from their hyperparameter tuning – a process of training multiple DNNs to empirically find the best network in various applications [25]. With the popularity of GPUs in large scale AI applications, the latest architectures from NVIDIA, namely Volta and Turing, are equipped with Tensor Cores, which provide hardware acceleration for matrix-multiply-accumulate operations. The cuBLAS library provides high-level APIs for GEMM and batched GEMM in half precision (i.e., HGEMM and batched HGEMM, respectively). There are also low-level APIs that can be used to program the Tensor Cores inside a GPU kernel. While the high-level APIs have been used to accelerate mixed-precision iterative refinement dense linear solvers [26], [27], this is the first effort, to the best of the

authors' knowledge, to programmatically use the Tensor Cores in an open-source and general purpose batched HGEMM routine that is competitive with the vendor optimized library.

III. The FP16 Tensor Cores in GPUs

The CUDA Toolkit is one of the first programming models to provide half-precision (i.e., FP16) arithmetic. Early support was added in late 2015 for selected embedded GPU models that are based on the Maxwell architecture. The FP16 arithmetic has become mainstream in CUDA-enabled GPUs since the Pascal architecture. The data type (`__half`) implements the IEEE-754 standard specification of the *binary16* format [5]. The format uses a 16-bit storage, where 10 bits are used for the mantissa, 5 bits are assigned to the exponent, and one bit is used to denote the sign. In general, half precision has a dynamic range that is significantly smaller than single or double precisions. Incorporating such a reduced precision was mainly motivated by the disruptive emergence of machine learning applications.

The Pascal architecture has a theoretical peak FP16 performance that is twice as fast as the peak FP32 performance. While the following architectures (Volta and Turing) maintain such ratio, they introduce further hardware acceleration for matrix multiplication in FP16. The hardware acceleration units are called Tensor Cores. They can deliver a theoretical peak performance that is up to $8\times$ faster than the peak FP32 performance. For example, each Volta V100 GPU has 640 Tensor Cores, evenly distributed across 80 multiprocessors. Each Tensor Core possesses a mixed-precision $4\times4\times4$ matrix processing array which performs the operation $D = A\times B+C$, where A, B, C and D are 4×4 matrices. The inputs A and B must be represented in FP16 format, while C and D can be represented in FP16 or in FP32 formats. It is also possible that C and D point to the same matrix.

The vendor library (cuBLAS) provides various optimized routines, mostly GEMMs, that can take advantage of the Tensor Core acceleration by setting the proper flag. The programming model also provides a set of low-level APIs that can be used programmatically in a user's kernel. From a programmer's point of view, tensor cores can be programmed using opaque objects called *fragments*. Each fragment is used to store one matrix. Fragments can be loaded from shared memory or from global memory using the `load_matrix_sync()` API. A similar API is available for storing the contents of a fragment into shared/global memories. The `mma_sync()` is used to perform the multiplication.

The programming model imposes some restrictions to the programming of tensor cores. First, the GEMM dimensions (M, N, K), which also control the size of the fragments, are limited to three discrete combinations, namely (16, 16, 16), (32, 8, 16), and (8, 32, 16). Second, the operations of load, store, and multiply fragments must be performed by one full warp (32 threads). Finally, the load/store APIs require that the leading dimension of the corresponding matrix be multiple of 16-bytes. As an example, a standard GEMM operation of size (16, 16, 16) requires three `load_matrix_sync()` calls (for A, B, and C), one `mma_sync()` call, and then a final `store_matrix_sync()` call to write the result.

IV. Design Details

This section describes the design details of the proposed batched HGEMM kernel. We assume that the dimensions of the GEMM operations are unified across the batch (i.e., fixed-size batches). Recall that a GPU kernel consists of a grid of thread blocks (TBs). We define the number of GEMM operations as `batchCount`.

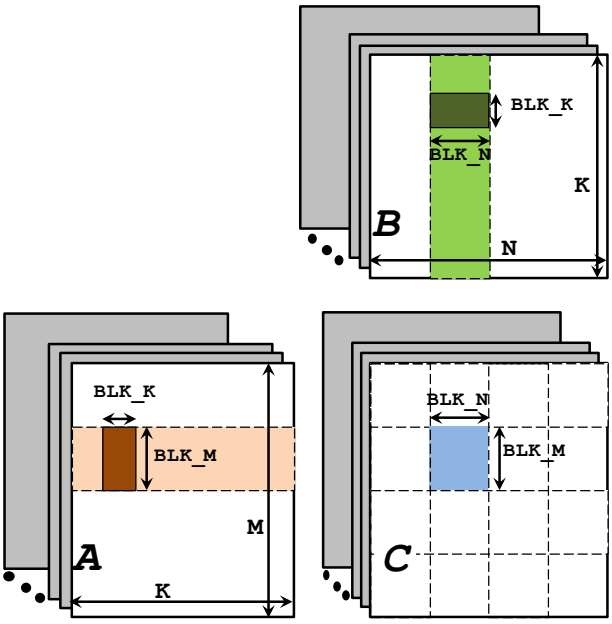

Fig. 1. Organization of the batched GEMM grid

The kernel is developed as part of the MAGMA library. [5] Performance comparisons are made against the highly optimized batched HGEMM routine provided by cuBLAS. In order to turn on Tensor Cores in cuBLAS, the flag `CUBLAS_TENSOR_OP_MATH` must be set. The CUDA version used in this paper is 9.2.

A. Grid Design

The grid design is similar to many previous contributions, such as [19]. The output matrices are subdivided into smaller blocks that can fit into a fast memory level (i.e., registers or shared memory). Such blocks can be square or rectangular, with their sizes denoted as (`BLK_M`×`BLK_N`). The grid configuration is a 3D thread block organization of size ($\left\lceil \frac{M}{\texttt{BLK\_M}} \right\rceil$, $\left\lceil \frac{N}{\texttt{BLK\_N}} \right\rceil$, `batchCount`). The grid is implicitly subdivided into `batchCount` subgrids, where each subgrid has a unique `batchid` (the $z-$dimension of the grid), and takes care of a single GEMM operation. Similarly, the input matrices A and B are subdivided into smaller blocks of sizes (`BLK_M`×`BLK_K`) and (`BLK_K`×`BLK_N`), respectively. Within every subgrid,

[5] https://icl.utk.edu/magma/

each TB is responsible for computing a block of the output matrix by reading a block row of A and a block column of B. Figure 1 illustrates the TB organization of the kernel. Unless otherwise mentioned, we use this grid organization across all the design variants discussed in this paper.

B. Thread Block Design

1) A Simple Design: Before thinking of sophisticated optimizations, the following questions should be answered: is it enough to subdivide the matrices using the discrete combinations imposed by tensor cores? Will these sizes be enough to achieve high performance? To answer such questions, we tried a simple design where the values of `BLK_M`, `BLK_N`, and `BLK_K` are restricted to the combinations defined by the Tensor Core hardware. Each TB consists of one warp. In order to avoid the limitations imposed on the leading dimensions by the load/store APIs, the kernel always performs the load/store operations through shared memory storage that abides by the leading dimension rule, thus removing such limitations from the matrices stored in the global GPU memory.

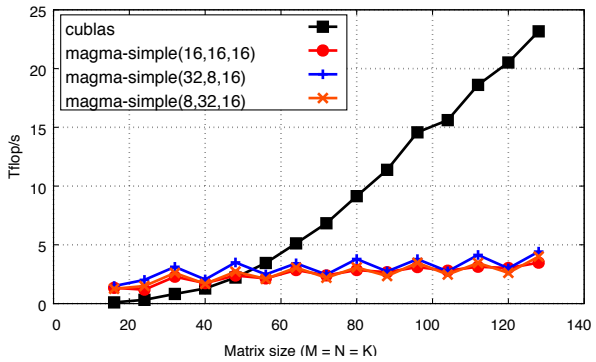

Fig. 2. Performance of the simple MAGMA batched HGEMM kernel that uses the same blocking parameters defined by Tensor Cores. Results are shown for square matrices using a Tesla V100 GPU, with `batchCount` = 50K.

Figure 2 shows the performance of such a kernel against cuBLAS. We observe that the performance of the MAGMA kernel is not competitive with the vendor routine, except for a slight advantage at sizes up to 50. Regardless of which Tensor Core sizes are used, the kernel performance does not scale as the matrix sizes get bigger. This means that the blocking values for the kernel must not be restricted to the Tensor Core sizes. The design should expand the parameter space for `BLK_M`, `BLK_N`, and `BLK_K` beyond the limited sizes imposed by the hardware.

2) Double-Sided Recursive Blocking: Recursive and hierarchical blocking are well-known techniques that have been used in previous GEMM designs [7]. In this paper, we use a similar technique. The matrices are blocked using the values of `BLK_M`, `BLK_N`, and `BLK_K`, which are typically larger than the Tensor Core sizes. The kernel main loop consists of $\left\lceil \frac{K}{\texttt{BLK\_K}} \right\rceil$ iterations. At each iteration, each TB holds a `BLK_M`×`BLK_K` block of A, a `BLK_K`×`BLK_N` block of B, and a `BLK_M`×`BLK_N` block of C. The latter is also cached for the lifetime of the TB, since it is accumulated across all iterations. This is the top level of blocking. Within each TB, the respective blocks of A, B, and C are recursively subdivided using one of the three Tensor Core sizes. We define the Tensor Core sizes by the parameters `TC_M`, `TC_N`, and `TC_K`. In order to fully benefit from the computational power of Tensor Cores, it is best that (`BLK_M`, `BLK_N`, `BLK_K`) are fully divisible by (`TC_M`, `TC_N`, `TC_K`), respectively.

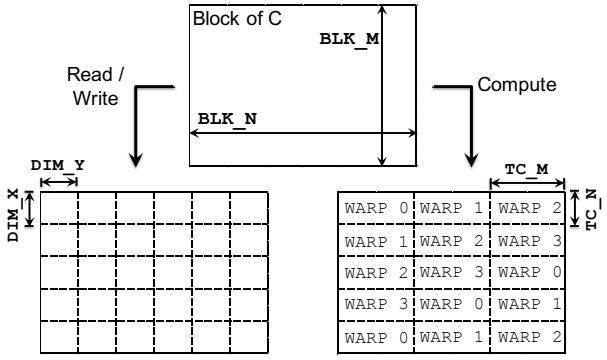

Fig. 3. An example for the double-sided recursive blocking applied to a block of C. The blocking dimensions for reads and writes are decoupled from those used for computations. The right side also shows how four warps can collaboratively work on a block og C.

Recursive blocking is usually single-sided, meaning that recursion is implemented in one specific way for both computations and memory operations. In this paper, however, we use a new double-sided recursive blocking technique, where recursion depends on which stage of the kernel is being executed. The recursion technique mentioned above uses the Tensor Core sizes for subdivision, and so it is used during the computation stage. However, the same subdivision may not be necessarily be good for memory operation. This is why we use different recursive subdivision sizes that come into play at loading and storing data blocks. Since the data are loaded into shared memory first (not into fragments directly), we are not obligated to use a single warp in a 32×1 configuration. We propose to use a generic 2D thread configuration at read and write stages. A warp reorganizes itself into a `DIM_X`×`DIM_Y` configuration. As an example, when loading a 16×16 block of data, it is much better to use a 16×2 configuration rather than the default 32×1 one. As a result, each block of A, B, and C is subdivided into sub-blocks of sizes `DIM_X`×`DIM_Y`, and the read/write operations are done using two nested `for` loops. In order to maximize the benefit of the design by having fully unrolled loops, it is required that each of `BLK_M`, `BLK_N`, and `BLK_K` is fully divisible by `DIM_X` and `DIM_Y`.

3) Collaborative Warps: As mentioned before, the flexibility of the developed kernel is a priority. Another generalization that helps in this direction is to allow a TB to have multiple collaborative warps. Such generalization allows TBs to have any number of threads that is a multiple of 32. During reading and writing of data blocks, threads reorganize themselves into a `DIM_X`×`DIM_Y` configuration, as mentioned before. Note that the parameter space for `DIM_X` and `DIM_Y` is now much

bigger, which serves flexibility. As an example, four warps can be used in many configurations, such as 8×16, 16×8, 32×4, 64×2, $\cdots$ etc. During computation, however, we must reorganize the threads in a $32 \times$ N_WARPS configuration in order to use the Tensor Cores. Since the data block of C is subdivided into many sub-blocks of size TC_M×TC_N, warps loop over these sub-blocks in a round-robin manner. For each sub-block, the respective warp loops over the corresponding sub-block rows of A and the sub-block columns of B, sends them in chunks to the Tensor Cores, and keeps accumulating the results in a dedicated fragment. The right side of Figure 3 shows collaboration among four warps.

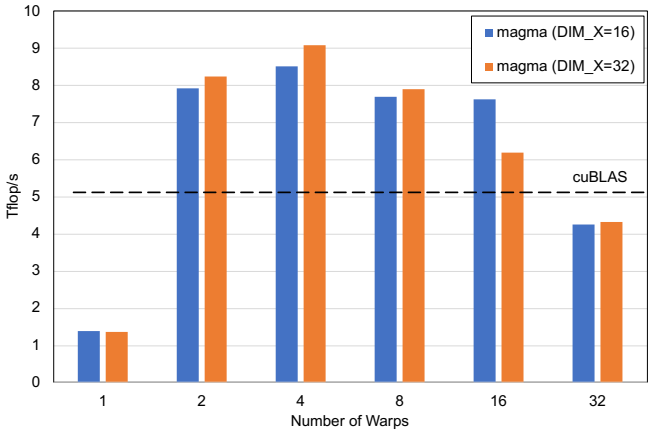

Fig. 4. The impact of the number of warps on performance for relatively large blocking sizes. Results are shown for square 64×64 matrices using a Tesla V100 GPU, with batchCount = 50K. All blocking sizes are set to 64, with all Tensor Core sizes set to 16. DIM_Y = (#warps×32)/DIM_X.

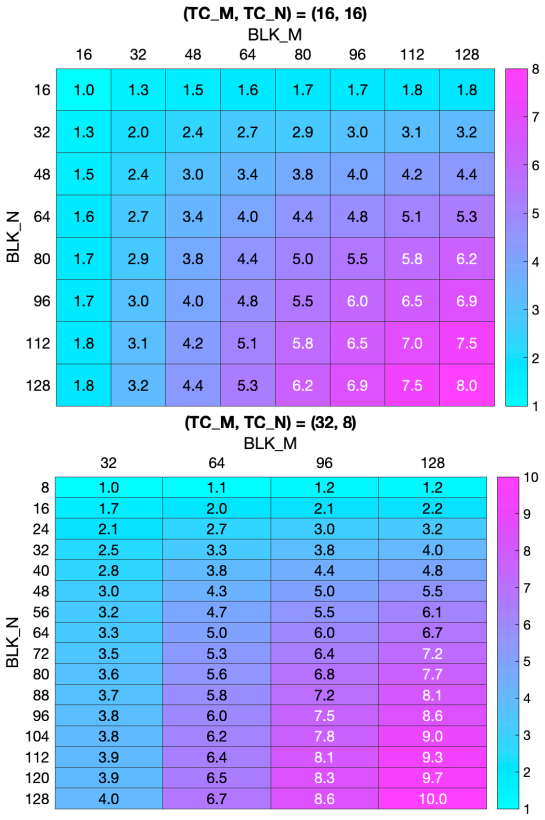

Fig. 5. The relative reduction in memory traffic for A and B using (BLK_M, BLK_N) $\geq$ (TC_M, TC_N). Results are shown for (TC_M, TC_N) = (16, 16) "top," and (32, 8) "bottom.." Note that BLK_x must always be multiple of TC_x.

The use of collaborative warps is beneficial when using relatively large values of BLK_M, BLK_N, or BLK_K. Reading in large blocks is usually required to achieve a high memory bandwidth and to increase data reuse. But since Tensor Core multiplications must be performed by a single warp, large blocks of data could mean too much work for one warp, and it becomes better to involve more warps in computation. To better quantify this concept, we performed an experiment on a batch of square multiplications of size 64×64. For simplicity, we fix BLK_M = BLK_N = BLK_K = 64, and TC_M = TC_N = TC_K = 16. We also show two possible thread configurations, one with DIM_X = 16, and the other with DIM_X = 32. Figure 4 shows that increasing the number of warps per TB leads to huge performance benefits when the blocks are relatively large (while fixing all other parameters). However, the performance slows down if too many warps are used per TB. One reason is occupancy, which impacts the number of live TBs that can be scheduled by the runtime on the same multiprocessor. Another reason is that some warps may be idle during the compute phase. This appears in Figure 4 when the number of warps is set to 32. With blocking sizes set to 64 and Tensor Core sizes set to 16, we have a 4×4 sub-block organization, which means that 16 warps out of the 32 are not assigned to any computational workload.

4) Summary of Design Contributions: We highlight two main design contributions in the proposed kernel. The first is the double-sided recursive blocking, which generalizes BLK_M, BLK_N, and BLK_K so that they are not bound to the Tensor Core sizes. The second is the use of collaborative warps inside a thread block. The latter improves the compute time inside a thread block versus a single warp configuration (Figure 4). On the other hand, generic block sizes help improve the memory traffic required by the kernel. To illustrate this point further, we simplify our analysis by assuming that BLK_M and BLK_N fully divide M and N, respectively. The proposed kernel would then require $\frac{M \times N}{\text{BLK\_M} \times \text{BLK\_N}}$ thread blocks. Ideally, all matrices (A, B, and C) are read once, while only C is written once. The proposed kernel performs an ideal memory traffic for C, since each thread block takes care of one block of C, and so it reads and writes such a block exactly once. However, since the parallel work is distributed across many independent thread blocks, there have to be redundant memory loads for A and B. The redundant loads are significantly affected by BLK_M and BLK_N. The total memory loads for A and B per thread block are given by $(K \times (\text{BLK\_M} + \text{BLK\_N}))$, since it has to read an entire block row of A and a block column of B. The total memory traffic (for A and B) for the whole kernel is given by $\frac{MNK(\text{BLK\_M}+\text{BLK\_N})}{\text{BLK\_M} \times \text{BLK\_N}}$. We can

now calculate the improvement (reduction) in memory traffic by comparing the previous formula against the case when (BLK_M, BLK_N, BLK_K) = (TC_M, TC_N, TC_K), which eventually yields:

$$\text{Memory traffic improvement} = \frac{\text{BLK\_M}\times\text{BLK\_N}\times(\text{TC\_M}+\text{TC\_N})}{\text{TC\_M}\times\text{TC\_N}\times(\text{BLK\_M}+\text{BLK\_N})}$$

Figure 5 shows the relative reduction in memory traffic for blocking sizes up to 128. As an example, using (BLK_M, BLK_N) = (64, 64) can reduce the memory traffic by a factor of $4\times$ when (TC_M, TC_N) = (16, 16), and by a factor of $5\times$ when (TC_M, TC_N) = (32, 8). Such a reduction usually leads to a better performance, since most of the memory requests are fulfilled from the main memory due to the relatively small caches in GPUs (compared to CPUs). It should be noted that very large blocks could worsen other aspects of the kernel, such as occupancy and register pressure. Therefore, a tuning process is required to search for the best blocking sizes on a specific GPU.

The pseudocode below shows the main loop of the kernel. At each iteration, a pair of data blocks, from A and B, is loaded from global memory to shared memory. The actual dimensions of these blocks (am, an, bm, bn) are computed at the beginning of the iteration so that the read_global() function accounts for partial blocks by means of zero-padding if required. Synchronization is required to make sure all data are visible to all warps before proceeding to the multiplication subroutine tc_multiply(). Another synchronization point is required to make sure all warps are done with the currently loaded data, and that it is safe to load another pair of data blocks.

```
for (int kk = 0; kk < K; kk += BLK_K) {
    (am, an) <- compute_block_size( A );
    (bm, bn) <- compute_block_size( B );

    sA <- read_global<...>(am, an, A, LDA);
    sB <- read_global<...>(bm, bn, B, LDB);

    sync<...>();
    tc_multiply<...>(sA, sB, fC);
    sync<...>();
    // advance A, B pointers
    A  += BLK_K * LDA;
    B  += BLK_K;
}
```

The tc_multiply() subroutine is a heavily templated inlined device function that performs the multiplication using Tensor Cores in a round-robin style. The routine uses a number of constants that are known at compile time (shown in all-caps, such as TC_BLOCKS, NWARPS, and NFRAG). TC_BLOCKS refers to the total number of multiplications a thread block performs, while NWARPS and NFRAG refer to the number of warps and the number of accumulator fragments per warp, respectively. The code distributes the TC_BLOCKS multiplications across warps. At each iteration of the outer loop, every warp independently calculates the coordinates of the C sub-block it should compute. It then proceeds to the innermost loop, where the corresponding sub-block row of A is multiplied by the corresponding sub-block column of B using the Tensor Core APIs. The code has a cleanup section that handles the situation when NWARPS does not fully divide TC_BLOCKS.

```
template<...>
__device__ __inline__ void
tc_multiply( half* sA, half* sB,
             wmma::fragment<...> fC[NFRAG] )
{
    // Declare A, B fragments
    wmma::fragment<...> fA;
    wmma::fragment<...> fB;

    int b = 0;
    #pragma unroll
    for(b = 0; b < TC_BLOCKS - NWARPS; b += NWARPS){
        (i, j, fid) <- get_next_frag_indices(warp_id);
        #pragma unroll
        for(int k = 0; k < BLK_K; k+=TC_K){
            half* ptrA = sA + k * BLK_M + i;
            half* ptrB = sB + j * BLK_K + k;
            wmma::load_matrix_sync(fA, ptrA, BLK_M);
            wmma::load_matrix_sync(fB, ptrB, BLK_K);
            wmma::mma_sync(fC[fid], fA, fB, fC[fid]);
        }
    }

    // cleanup code
    if(warp_id < TC_BLOCKS - b){
        (i, j, fid) <- get_next_frag_indices(warp_id);
        #pragma unroll
        for(int k = 0; k < BLK_K; k+=TC_K){
            half* ptrA = sA + k * BLK_M + i;
            half* ptrB = sB + j * BLK_K + k;
            wmma::load_matrix_sync(fA, ptrA, BLK_M);
            wmma::load_matrix_sync(fB, ptrB, BLK_K);
            wmma::mma_sync(fC[fid], fA, fB, fC[fid]);
        }
    }
}
```

V. Performance Tuning

The developed kernel is written using C++ templates, with eight main tuning parameters. These parameters are the configuration sizes of the Tensor Cores (TC_M, TC_N, TC_K), the blocking sizes for A, B, and C (BLK_M, BLK_N, BLK_K), and the thread configuration for reading and writing data blocks (DIM_X, DIM_Y). In this section, we describe a performance tuning experiment that was conducted on square matrices. The same steps apply to non-square test cases.

A. Parameter Space Generation

The kernel parameters must satisfy a number of conditions in order for the kernel to perform correctly. As mentioned before, a Tensor Core dimensions TC_x must fully divide BLK_x, where x$\in${M, N, K}. Each of DIM_X and DIM_Y must fully divide every blocking size BLK_x. The product DIM_X$\times$DIM_Y must also be multiple of 32, in order to have full warps. There is another set of restrictions imposed by the hardware resources, such as the maximum allowed shared memory space per TB. Despite such conditions and restrictions, the parameter space for the kernel remains very large. Using an automated script, the initial number of eligible kernels was larger than $15,000$. In order to prune the search space, we have applied some "soft constraints." Such constraints are based on gained experience while tuning the batched GEMM kernel for other precisions [19], where the learned lesson was to use lightweight thread blocks in order to help the CUDA runtime schedule as many thread

blocks per multiprocessor as possible. Recall that Figure 5 suggests large blocking sizes, which could use too much of the resources. Therefore, the purpose of the soft constraints is to eliminate potentially bad candidates, which reduces the amount of time required to perform the tuning sweep. Because we are considering small sizes up to 128, the constraints aim for a "sweet spot" where thread blocks can handle large blocks of data while being relatively lightweight. As an example, each blocking size BLK_x is given the range [16:128] in steps of 16. This caps the kernel's shared memory requirement to 32KB. The maximum number of warps per TB is also capped at four. This is the minimum number of warps required to hide the execution latencies of core math operations on a multiprocessor.[6] Despite such constraints, the automated generation script was able to find $4,948$ eligible kernel instances. Each kernel instance has a unique ID (version number) that is used for testing and reporting results.

B. Testing and Evaluation

All kernel instances were compiled and tested on square sizes up to 128. The total compilation and execution time was around four days, due to the limited access to one V100 GPU. All the collected data were then sent to an automatic analysis program that truncates the results into a smaller subset. This is done by extracting, for each size, the best performing BK "winning kernels," where BK is a tunable value. The program also takes an acceptable performance tolerance TOL. We define TOL as the percentage of the best observed performance that can be sacrificed in order to reduce the number of compiled kernels. This helps avoid very large binaries when the routine is finally released into a software library. By investigating the winning kernel IDs across all the test points, we can reduce the number of compiled kernels by prioritizing the most frequent winning kernels that are within the acceptable tolerance.

The tuning experiment was conducted using BK = 10, while varying TOL between 0% and 15%. Table I shows the number of compiled kernels against different values of TOL. At zero tolerance, there are eleven kernel instances required for the fifteen test points used in the experiment. Such a finding emphasizes the performance sensitivity to tuning parameters, since a small change in the size leads to a switch of kernels. This is not the case for relatively large sizes, where usually one kernel instance can cover a larger range of sizes. In general, we can reduce the number of compiled kernels by 36% if the tolerance is increased from 0% to 15%. We also observe that increasing TOL does not always result in reducing the number of kernels. In fact, the outcome depends on the performance variations at each test point, and whether more "acceptable" kernels can be found within the larger tolerance.

Figure 6 shows the best performance of the developed kernel when tolerance is set to 0% and 15%. The figure also shows the percentage drop going from zero tolerance to 15% tolerance. Ideally, no negative percentages should be observed.

[6]https://docs.nvidia.com/cuda/volta-tuning-guide/index.html#sm-scheduling

TOL	Number of kernels (reduction %)
0%	11 (0%)
5%	8 (27.2%)
10%	8 (27.2%)
15%	7 (36.3%)

TABLE I
THE REQUIRED NUMBER OF COMPILED KERNELS FOR A GIVEN PERFORMANCE TOLERANCE (TOL). THE VALUE OF BK IS FIXED AT 10. THE COLLECTED DATA ARE GENERATED ON A TESLA V100 GPU. TEST POINTS ARE FOR SQUARE SIZES UP TO 128, WITH BATCHCOUNT = 3,000.

However, our measurements show a different behavior. While, for the most part, the performance of zero tolerance is higher, there are some measurements where the performance at 15% tolerance is actually better. Such behavior can occur when there is some noise in the original data collected, due to warm-up runs or insufficient number of runs at each test point, especially when two kernel instances are performing very closely to each other. In addition, the negative percentages are in the range of $\approx 1\%$ or less, which means that the tuning experiment conducted here delivers the best performance with a error margin of $\approx$ 1%. The maximum performance drop observed is 11.92%, which is in the range of the accepted tolerance.

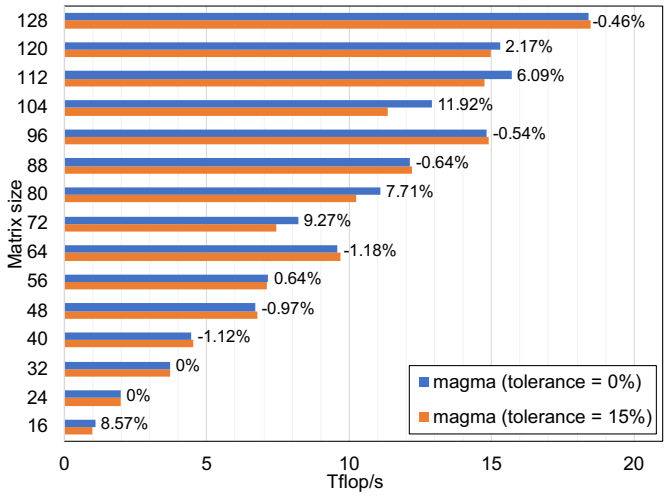

Fig. 6. Impact of the tolerance setting on the outcome of the tuning experiment. The performance is recorded for square sizes up to 128, with batchCount = 3,000, on a Tesla V100 GPU.

VI. PERFORMANCE AGAINST THE VENDOR LIBRARY

Figure 7 shows the final performance of the tuned MAGMA kernel against cuBLAS. Recall that the cuBLAS kernel is written in a low-level language to utilize some optimization techniques that are not available in CUDA C or PTX instructions. And so, its asymptotic performance is faster than MAGMA by factors up to $3\times$ for large matrices. However, the developed kernel has a significant advantage against cuBLAS for small sizes below 100, scoring speedups that range between $1.2\times$ and $23.8\times$. We observe that, in general, the smaller the sizes, the larger the speedup. We also observe performance

drops at some sizes (e.g., 72, 104 and 112). This is a known behavior in most GPU kernels, which occurs for problem sizes that are not fully divisible by the blocking values.

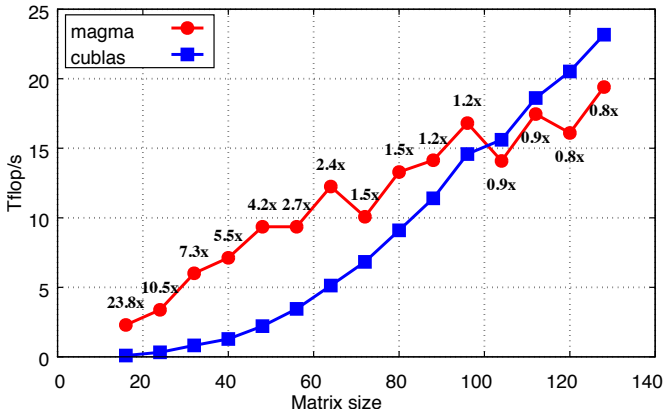

Fig. 7. Performance of the batched HGEMM kernel against cuBLAS. Results are for square sizes up to 128, with `batchCount` = 50,000, on a Tesla V100 GPU.

VII. Optimization for Extremely Small Matrices

This section focuses on performance optimization for small sizes that cannot fully occupy the Tensor Cores. In particular, we are looking into small matrices, both square and rectangular, whose dimensions are ≤ 16. This range of sizes has been subject to many research efforts recently, due to its popularity in many applications [3] [28]. We present three different approches for such tiny sizes, and assess the advantages and disadvantages of each. Since the matrices are extremely small, performance tests are conducted using `batchCount`$=$ 1M in order to saturate the GPU as much as possible. The significant slowdown of the cuBLAS performance was not observed on sizes smaller than 16 when the batch contains more than 65K operations.

A. Improving Grid Design of the Existing Kernel

We begin with the same kernel developed in Section IV. For very small sizes, it is enough to use a single warp per TB (recall that we cannot use sub-warps in order to use Tensor Cores). However, the original grid design in Section IV-A assigns one subgrid to one TB, which means exactly one warp per TB. This means that if the warp gets stalled—due to memory operations, for example—the whole TB stays idle in a waiting state. While it is possible for the runtime to execute other TBs on the same multiprocessor, we propose to alter the design of the kernel grid to allow multiple subgrids (problems) in the same TB, and so warp latency hiding is possible not only across TBs, but also across warps in the same TB. Such a modification has been added to the same kernel with the introduction of a ninth tuning parameter `NSG` that controls the number of subgrids per TB. The `NSG` is set to 1 by default for sizes larger than 16, since the TB is usually configured with more than one warp, and so setting `NSG`>1 has a limited benefit. On the contrary, it is recommended to increase `NSG` for small sizes less than 16. We tested the performance for `NSG`$\in\{1, 2, 3, 4\}$. Figure 8 shows the performance benefit with a tuned `NSG` versus fixing it at 1. We observe that a tunable `NSG` has an impact only on sizes up to 10, with speedups up to 23% at size 2×2. The performance for sizes 11 through 16 remains the same despite increasing `NSG`. We point out that such a finding does not contradict the results of Figure 4, which shows a drop in performance with increasing warps. The experiment shown in Figure 4 increases the number of warps while fixing the amount of work, while the tuning experiment of `NSG` increases both the number of warps and the workload (each warp is assigned a different GEMM), in an effort to hide the potential latencies of stalled warps. We refer to this kernel with the modified grid as *magma-small-v1*.

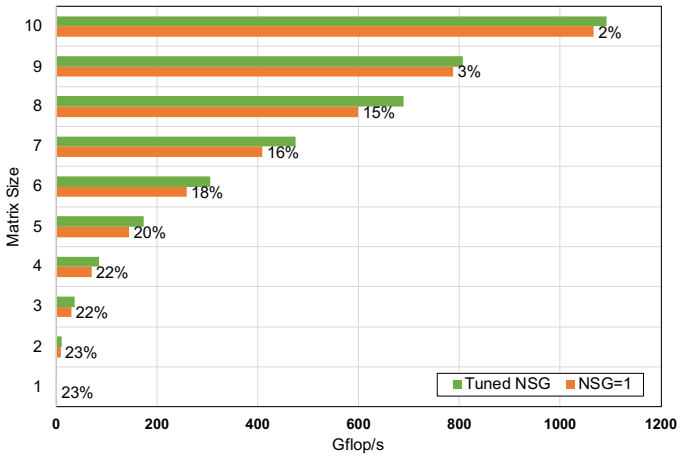

Fig. 8. Performance of the MAGMA kernel against against different values of `NSG`. Results are shown for square sizes up to 10, with `batchCount` = 1M, on a Tesla V100 GPU.

B. A New Kernel: Improving Tensor Core Utilization

The modified grid design of the existing kernel does improve the performance for small sizes. However, such modification still assigns a single multiplication to the Tensor Cores at a time. For simplicity, consider the Tensor Core sizes (16, 16, 16). If there is a multiplication where both M and N are ≤ 8, then we can use a single Tensor Core operation to perform multiple multiplications at the same time. Figures 9 and 10 show how the Tensor Cores can be utilized to perform multiple independent GEMMs concurrently. The figures show both square and rectangular cases. The general idea is to store the A matrices along a block column of width K, while the B matrices are stored along a block row of height K. The output matrices are stored in a diagonal-like shape in the output fragment. Note that the grey cells in the output fragment correspond to non-zero results that are not needed.

The number of simultaneous GEMMs that can be done is a function of M, N, `TC_M`, and `TC_N`. From Figures 9 and 10, we can deduce that the maximum number of simultaneous GEMMs is $\texttt{min}(\lfloor \frac{\texttt{TC\_M}}{M} \rfloor, \lfloor \frac{\texttt{TC\_N}}{N} \rfloor)$. Note that this formula is independent of K, for which the only requirement is $K \leq$`TC_K`.

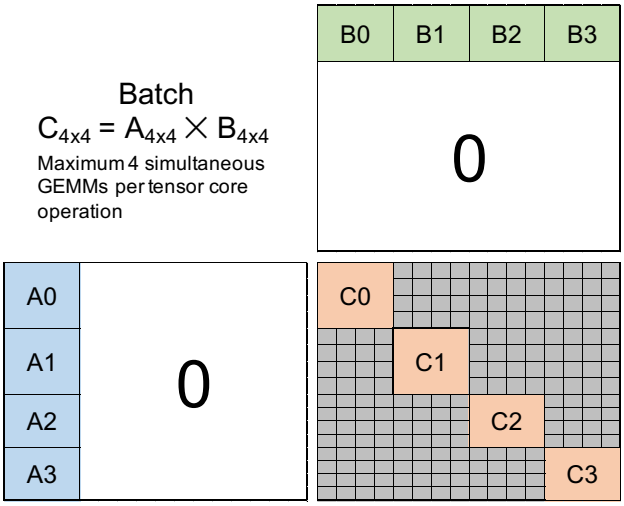

Fig. 9. Improving Tensor Core utilization by assigning multiple GEMMs at a time. Example for square matrices of size 4, with all Tensor Core sizes set to 16.

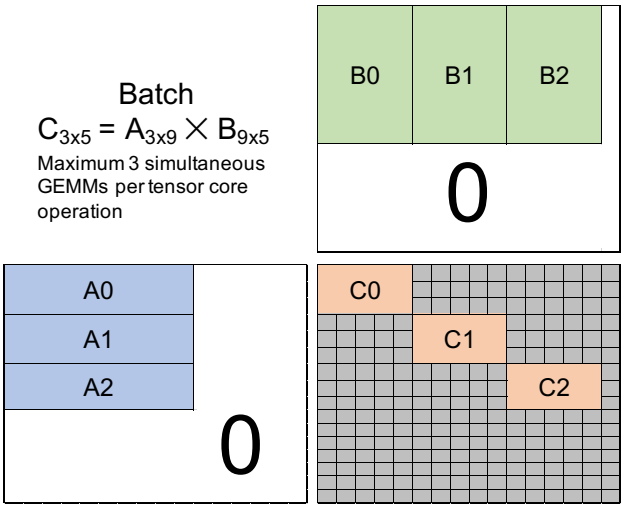

Fig. 10. Improving Tensor Core utilization by assigning multiple GEMMs at a time. Example for (M, N, K) = (3, 5, 9), with all tensor core sizes set to 16.

Such a design could not be incorporated into the original kernel, because it contradicts one of its design principles. The general kernel uses multiple warps per a partial GEMM operation, while this kernel allows a single warp to span multiple operations at the same time. This is why a separate kernel is developed, which we call *magma-small-v2*. Figure 11 shows the performance improvement obtained by *magma-small-v2* against *magma-small-v1*. The speedups are much more significant than the ones observed in Figure 8. The speedups decrease as the problem sizes increase, since fewer concurrent GEMMs can be executed per Tensor Core operation. The performance improvements become insignificant after size 8×8, since exactly one GEMM can be executed at a time starting size 9×9 and up.

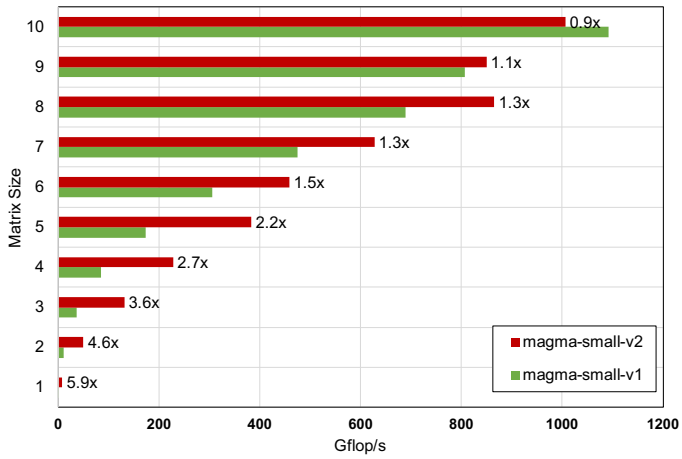

Fig. 11. Performance improvement due to increased Tensor Core utilization. Results are shown for square sizes up to 10, with `batchCount` = 1M, on a Tesla V100 GPU.

C. Are Tensor Cores Necessary for Small Sizes?

Although the *magma-small-v2* kernel has been successful in improving the performance against *magma-small-v1*, the use of Tensor Cores for such small sizes may still be questionable. A single Tensor Core operation can perform 8192 FLOPs using any of the (`TC_M`, `TC_N`, `TC_K`) combinations. Considering Figure 9, we perform four multiplications, each having $2 \times 4^3 = 128$ FLOPs, for a total of 512 FLOPs. This means that *magma-small-v2* uses only 6.25% of the available compute power, while the configuration shown in Figure 10 uses 9.89% of it. Such low ratios raise questions about using Tensor Cores, and whether conventional methods (i.e., without Tensor Cores) can perform the multiplications more efficiently. In this regard, we refer to a kernel that has been developed before for very small matrices [29]. The kernel addresses very small square multiplication of sizes up to 32. The main design idea is to use an $N \times N$ thread configuration for each GEMM, such that each thread is responsible for a single element in the output matrix. The code is fully unrolled for every size using C++ templates. In this paper, we use the same kernel, but rather generalize it to work for both square and rectangular multiplications. The generalization uses a 1D thread configuration of $\texttt{max}(M \times K, K \times N, M \times N)$ for each subgrid. The configuration is remapped to $M \times K$, $K \times N$, or $M \times N$ configurations when transferring A, B, or C respectively. For each problem, A and B are loaded in shared memory, while C is kept in registers for accumulation. We call the generalized kernel *magma-small-v3*.

We tested the performance of *magma-small-v3* for square sizes against *magma-small-v2*, for which the results are summarized in Figure 12. We observe that significant improvements for small sizes can yet be achieved by a kernel that does not use the Tensor Cores. Similar to Figure 11, the speedups are observed only for size up to 10. The smaller the sizes, the larger the speedup.

To put all results into perspective, Figure 13 summarizes the

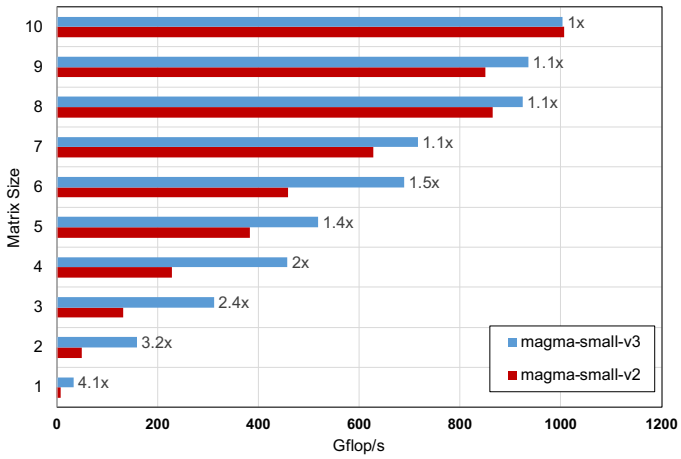

Fig. 12. Performance improvement of *magma-small-v3* against *magma-small-v2*. Results are shown for square sizes up to 10, with `batchCount` = 1M, on a Tesla V100 GPU.

performance of all three versions against cuBLAS. The data labels show the speedup scored by the best performing version against the vendor routine. The figure shows a minimum speedup of 80% at size 16. The speedup grows as the sizes become smaller, reaching $25.8\times$ at sizes 2×2.

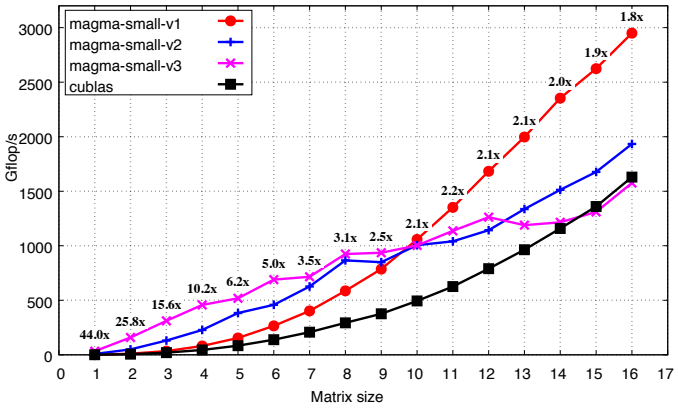

Fig. 13. Performance of all kernel versions against cuBLAS. Results are shown for square sizes up to 16, with `batchCount` = 1M, on a Tesla V100 GPU

The *magma-small-v1* is the best performing kernel for sizes 10 and up. For smaller sizes, the *magma-small-v3* is the best solution, though no Tensor Cores are used. This is an interesting observation that nicely aligns with the percentage Tensor Core utilization in a given multiplication. Recall that the performance is memory bound at such small sizes, and so the compute time of the kernel should ideally be negligible in order to have the best performance. The V100 GPU has a theoretical peak FP16 performance of 125 teraFLOP/s using Tensor Cores. The theoretical peak performance without Tensor Cores is 31.7 teraFLOP/s, which is 25.12% of the Tensor Core performance. Now, depending on the utilization of the Tensor Core compute power, we can tell whether it is recommended to use Tensor Cores for memory bound kernels that operate on very small problems. Figure 14 shows the percentage utilization for square matrix multiplications, based on a (16, 16, 16) configuration of Tensor Cores. The utilization is computed as $\left\lfloor \frac{16}{N} \right\rfloor \times \frac{2 \times N^3}{8192}$, where N is the size of the square matrices. The figure shows matching results with Figure 13. Using Tensor is recommended only if the utilization is above 25.12%, which is the case for sizes larger than 10. Below this size, the useful compute power provided by the Tensor Cores becomes less than the non-accelerated FP16 performance. This is why *magma-small-v3* is the best solution for sizes less than 10.

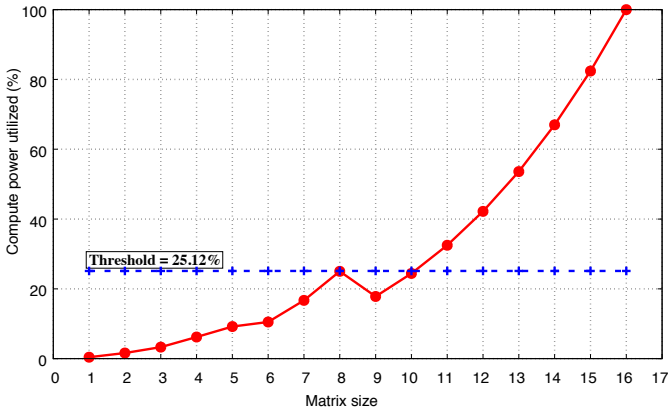

Fig. 14. The percentage of available Tensor Core compute power for square multiplications of small matrices.

While Figure 13 shows that there is no winning scenario for the *magma-small-v2* kernel, the same is not true for rectangular multiplications. In order for *magma-small-v2* to outperform *magma-small-v1*, it has to perform multiple GEMMs using one Tensor Core operation, which implies that M and N must be ≤ 8. In addition, the value of K has to be sufficiently larger than M and N in order to enhance the Tensor Core utilization. As an example, Figure 15 shows the performance results for $M = 4$ and $N = 3$, while varying K between 1 and 16. We observe that *magma-small-v2* has a clear advantage for $K \geq 8$. The overall improvement against cuBLAS is also substantial, scoring speedups between $5.5\times$ and $9.8\times$.

VIII. Impact of `batchCount` on Performance

The reported results so far discuss only the "asymptotic performance" by making `batchCount` large enough to saturate the GPU with enough work. However, the actual value of `batchCount` varies significantly from one application to another. As an example, the original work that introduced the CUDA Deep Neural Network (cuDNN) library [18] shows results for "mini-batches" up to 128 only. Tensor-formulated, high-order finite element method (FEM) simulations [3] can be performed using a sequence of GEMM operations executed independently on each element, thus making `batchCount` equal to the number of elements. The latter typically ranges from a few thousands to an order of a million, depending

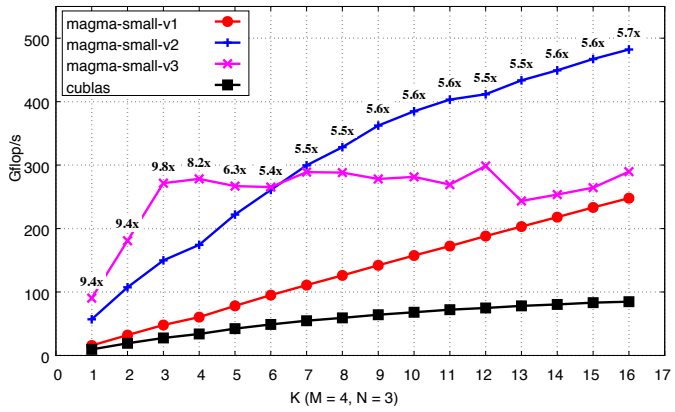

Fig. 15. Performance of all kernel versions against cuBLAS, with $M = 4$ and $N = 3$. Results are shown for `batchCount` = 1M, on a Tesla V100 GPU

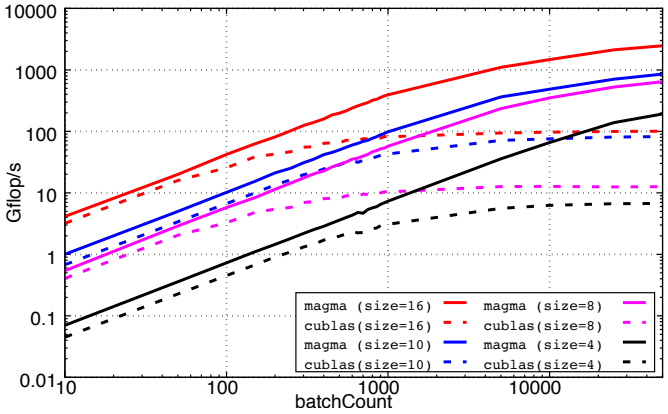

Fig. 17. Impact of `batchCount` on performance. Results are shown for very small selected sizes a Tesla V100 GPU

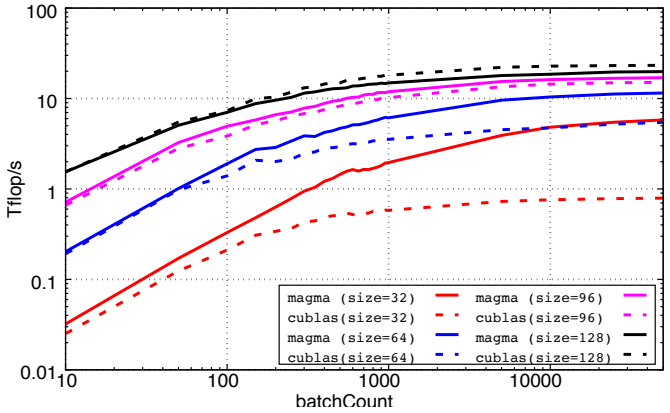

Fig. 16. Impact of `batchCount` on performance. Results are shown for selected square sizes on a Tesla V100 GPU

on the discretization. Performing batch computations on hierarchical matrices, as presented in [30], can use batches that are order of $100K$, depending on the blocking size and the order of the original matrix. Other work related to block-Jacobi preconditioning [31] uses a maximum block size of 16×16, with the number of blocks being dependent again on the size of the matrix. Generally speaking, there is no typical range for `batchCount`, since such information is very application-specific.

Figures 16 and 17 show the performance plotted against a wide range of `batchCount` values for selected matrix sizes. For the very small sizes in Figure 17, we run *magma-small-v1* for sizes $\geq$ 10, and *magma-small-v3* otherwise. In general, the performance of both MAGMA and cuBLAS steadily increases as the batch becomes larger. For any given matrix size, the faster of the two routines maintains such an advantage across all batch sizes. We observe that, for sizes less than 96, the performance of cuBLAS stagnates much earlier than MAGMA, thus conceding the advantage to the latter. We also observe that, for most sizes, the advantage of MAGMA becomes apparent when the batch contains more than 200 operations.

IX. Conclusion and Future Work

This paper introduced an optimized batched matrix multiplication kernel using FP16 arithmetic on GPUs. The paper represents one of the first efforts to programmatically use the Tensor Core technology in an open-source implementation. The first part of the paper presents a highly flexible design that uses a double-sided blocking technique in order to overcome the restricted block sizes of the Tensor Cores. The developed kernel outperforms the vendor routine for sizes up to 100, scoring speedups that range between $1.2\times$ and $10\times$. The second part of the paper addresses optimization techniques that are specific for very small problem sizes that cannot entirely occupy the Tensor Core units. Three different designs have been discussed in this regard. The best performance observed across those designs is significantly higher than cuBLAS. The observed speedups are in the range of $1.8\times$ to $26\times$.

Future directions include investigating other optimization techniques to improve the asymptotic performance for larger sizes, automatic performance tuning for various shapes and settings, supporting variable size problems in the same batch, and studying the impact of such work on real applications from different fields.

Acknowledgment

This work is partially supported by NSF Grant No. OAC 1740250 and CSR 1514286, NVIDIA, and the Department of Energy under the Exascale Computing Project (17-SC-20-SC and LLNL subcontract under DOE contract DE-AC52-07NA27344).

References

[1] S. Williams, A. Waterman, and D. Patterson, "Roofline: An Insightful Visual Performance Model for Multicore Architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009. [Online]. Available: http://doi.acm.org/10.1145/1498765.1498785

[2] S. N. Yeralan, T. A. Davis, W. M. Sid-Lakhdar, and S. Ranka, "Algorithm 980: Sparse QR Factorization on the GPU," *ACM Trans. Math. Softw.*, vol. 44, no. 2, pp. 17:1–17:29, Aug. 2017. [Online]. Available: http://doi.acm.org/10.1145/3065870

[3] A. Abdelfattah, M. Baboulin, V. Dobrev, J. J. Dongarra, C. W. Earl, J. Falcou, A. Haidar, I. Karlin, T. V. Kolev, I. Masliah, and S. Tomov, "High-Performance Tensor Contractions for GPUs," in *International Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA*, 2016, pp. 108–118. [Online]. Available: https://doi.org/10.1016/j.procs.2016.05.302

[4] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep Learning with Limited Numerical Precision," in *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ser. ICML'15. JMLR.org, 2015, pp. 1737–1746. [Online]. Available: http://dl.acm.org/citation.cfm?id=3045118.3045303

[5] "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2008*, pp. 1–70, Aug 2008. [Online]. Available: https://ieeexplore.ieee.org/document/4610935

[6] V. Volkov and J. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2008, November 15-21, 2008, Austin, Texas, USA*, 2008, p. 31. [Online]. Available: http://doi.acm.org/10.1145/1413370.1413402

[7] R. Nath, S. Tomov, and J. Dongarra, "An Improved Magma Gemm For Fermi Graphics Processing Units," *The International Journal of High Performance Computing Applications*, vol. 24, no. 4, pp. 511–515, 2010. [Online]. Available: https://doi.org/10.1177/1094342010385729

[8] S. Tomov, J. J. Dongarra, and M. Baboulin, "Towards dense linear algebra for hybrid GPU accelerated manycore systems," *Parallel Computing*, vol. 36, no. 5-6, pp. 232–240, 2010. [Online]. Available: https://doi.org/10.1016/j.parco.2009.12.005

[9] K. Rupp, P. Tillet, F. Rudolf, J. Weinbub, A. Morhammer, T. Grasser, A. Jngel, and S. Selberherr, "ViennaCL—Linear Algebra Library for Multi- and Many-Core Architectures," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, pp. S412–S439, 2016. [Online]. Available: https://doi.org/10.1137/15M1026419

[10] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, and S. Tomov, "Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs," in *GPU Computing Gems*, W. mei W. Hwu, Ed. Morgan Kaufmann, Sep. 2010, vol. 2. [Online]. Available: https://hal.inria.fr/inria-00547847

[11] Y. Li, J. Dongarra, and S. Tomov, "A Note on Auto-tuning GEMM for GPUs," in *Computational Science – ICCS 2009*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 884–892.

[12] J. Kurzak, S. Tomov, and J. Dongarra, "Autotuning GEMM Kernels for the Fermi GPU," *IEEE Transactions on Parallel and Distributed Systems*, vol. 23, no. 11, pp. 2045–2057, Nov 2012.

[13] G. Tan, L. Li, S. Triechle, E. H. Phillips, Y. Bao, and N. Sun, "Fast implementation of DGEMM on Fermi GPU," in *Conference on High Performance Computing Networking, Storage and Analysis, SC 2011, Seattle, WA, USA, November 12-18, 2011*, 2011, pp. 35:1–35:11. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063431

[14] J. Lai and A. Seznec, "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)*, ser. CGO '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/CGO.2013.6494986

[15] S. Gray, "A full walk through of the SGEMM implementation," https://github.com/NervanaSystems/maxas/wiki/SGEMM, 2015.

[16] A. Haidar, T. Dong, P. Luszczek, S. Tomov, and J. J. Dongarra, "Batched matrix computations on hardware accelerators based on GPUs," *IJHPCA*, vol. 29, no. 2, pp. 193–208, 2015. [Online]. Available: https://doi.org/10.1177/1094342014567546

[17] C. Jhurani and P. Mullowney, "A GEMM interface and implementation on NVIDIA GPUs for multiple small matrices," *CoRR*, vol. abs/1304.7053, 2013. [Online]. Available: http://arxiv.org/abs/1304.7053

[18] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cuDNN: Efficient Primitives for Deep Learning," *CoRR*, vol. abs/1410.0759, 2014. [Online]. Available: http://arxiv.org/abs/1410.0759

[19] A. Abdelfattah, A. Haidar, S. Tomov, and J. J. Dongarra, "Performance, design, and autotuning of batched GEMM for gpus," in *High Performance Computing - 31st International Conference, ISC High Performance 2016, Frankfurt, Germany, June 19-23, 2016, Proceedings*, 2016, pp. 21–38. [Online]. Available: https://doi.org/10.1007/978-3-319-41321-1_2

[20] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional Architecture for Fast Feature Embedding," in *Proceedings of the 22Nd ACM International Conference on Multimedia*, ser. MM '14. New York, NY, USA: ACM, 2014, pp. 675–678. [Online]. Available: http://doi.acm.org/10.1145/2647868.2654889

[21] L. Ng, K. Wong, A. Haidar, S. Tomov, and J. Dongarra, "MagmaDNN High-Performance Data Analytics for Manycore GPUs and CPUs," December 2017, Magma-DNN, 2017 Summer Research Experiences for Undergraduate (REU), Knoxville, TN. [Online]. Available: http://icl.cs.utk.edu/magma/software/

[22] K. Chellapilla, S. Puri, and P. Simard, "High Performance Convolutional Neural Networks for Document Processing," in *Tenth International Workshop on Frontiers in Handwriting Recognition*, G. Lorette, Ed., Université de Rennes 1. La Baule (France): Suvisoft, Oct. 2006, http://www.suvisoft.com. [Online]. Available: https://hal.inria.fr/inria-00112631

[23] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[24] S. Winograd, S. for Industrial, A. Mathematics, C. B. of the Mathematical Sciences, and N. S. F. E. U. d'Amèrica), *Arithmetic Complexity of Computations*, ser. CBMS-NSF Regional Conference Series in Applied Mathematics. Society for Industrial and Applied Mathematics, 1980. [Online]. Available: https://books.google.com/books?id=GU1NQJBcWIsC

[25] N. Tomov and S. Tomov, "On deep neural networks for detecting heart disease," *CoRR*, vol. abs/1808.07168, 2018. [Online]. Available: http://arxiv.org/abs/1808.07168

[26] A. Haidar, A. Abdelfattah, M. Zounon, P. Wu, S. Pranesh, S. Tomov, and J. Dongarra, "The Design of Fast and Energy-Efficient Linear Solvers: On the Potential of Half-Precision Arithmetic and Iterative Refinement Techniques," in *Computational Science – ICCS 2018*. Springer International Publishing, 2018, pp. 586–600.

[27] A. Haidar, S. Tomov, J. Dongarra, and N. Higham, "Harnessing GPU Tensor Cores for Fast FP16 Arithmetic to Speed up Mixed-Precision Iterative Refinement Solvers," in *Proceedings of the International ACM Conference on International Conference on Supercomputing (SC'18)*, 2018, (In press).

[28] K. Kim, T. B. Costa, M. Deveci, A. M. Bradley, S. D. Hammond, M. E. Guney, S. Knepper, S. Story, and S. Rajamanickam, "Designing Vector-friendly Compact BLAS and LAPACK Kernels," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '17. New York, NY, USA: ACM, 2017, pp. 55:1–55:12. [Online]. Available: http://doi.acm.org/10.1145/3126908.3126941

[29] I. Masliah, A. Abdelfattah, A. Haidar, S. Tomov, M. Baboulin, J. Falcou, and J. J. Dongarra, "High-Performance Matrix-Matrix Multiplications of Very Small Matrices," in *Euro-Par 2016: Parallel Processing - 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings*, 2016, pp. 659–671. [Online]. Available: https://doi.org/10.1007/978-3-319-43659-3_48

[30] I. Yamazaki, A. Abdelfattah, A. Ida, S. Ohshima, S. Tomov, R. Yokota, and J. J. Dongarra, "Performance of Hierarchical-matrix BiCGStab Solver on GPU Clusters," in *2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2018, Vancouver, BC, Canada, May 21-25, 2018*, 2018, pp. 930–939. [Online]. Available: https://doi.org/10.1109/IPDPS.2018.00102

[31] H. Anzt, J. Dongarra, G. Flegar, and E. S. Quintana-Ort, "Variable-size batched GaussJordan elimination for block-Jacobi preconditioning on graphics processors," *Parallel Computing*, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819117302107