Journal of Parallel and Distributed Computing 145 (2020) 188–201

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Matrix multiplication on batches of small matrices in half and half-complex precisions

Ahmad Abdelfattah [a,*], Stanimire Tomov [a], Jack Dongarra [a,b,c]

[a] University of Tennessee, USA
[b] Oak Ridge National Laboratory, USA
[c] University of Manchester, UK

ARTICLE INFO

Article history:
Received 8 November 2019
Received in revised form 14 May 2020
Accepted 5 July 2020
Available online 15 July 2020

Keywords:
Matrix multiplication
Half precision
Batch linear algebra
GPU computing

ABSTRACT

Machine learning and artificial intelligence (AI) applications often rely on performing many small matrix operations—in particular general matrix–matrix multiplication (GEMM). These operations are usually performed in a reduced precision, such as the 16-bit floating-point format (i.e., half precision or FP16). The GEMM operation is also very important for dense linear algebra algorithms, and half-precision GEMM operations can be used in mixed-precision linear solvers. Therefore, high-performance batched GEMM operations in reduced precision are significantly important, not only for deep learning frameworks, but also for scientific applications that rely on batched linear algebra, such as tensor contractions and sparse direct solvers.

This paper presents optimized batched GEMM kernels for graphics processing units (GPUs) in FP16 arithmetic. The paper addresses both real and complex half-precision computations on the GPU. The proposed design takes advantage of the Tensor Core technology that was recently introduced in CUDA-enabled GPUs. With eight tuning parameters introduced in the design, the developed kernels have a high degree of flexibility that overcomes the limitations imposed by the hardware and software (in the form of discrete configurations for the Tensor Core APIs). For real FP16 arithmetic, performance speedups are observed against cuBLAS for sizes up to 128, and range between 1.5× and 2.5×. For the complex FP16 GEMM kernel, the speedups are between 1.7× and 7× thanks to a design that uses the standard interleaved matrix layout, in contrast with the planar layout required by the vendor's solution. The paper also discusses special optimizations for extremely small matrices, where even higher performance gains are achievable.

1. Introduction

High-performance linear algebra libraries enable scientific applications to run efficiently on massively parallel architectures. Considering dense linear algebra software, high performance is usually achievable 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 GEMM kernel is not only embarrassingly parallel; it also has a relatively high arithmetic intensity [30], 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. These two properties make the GEMM kernel extremely important for numerous computational domains. According to the standard interface of Basic Linear Algebra Subprograms (BLAS), a standard GEMM operation updates a matrix C, where $C_{M\times N} = \alpha A_{M\times K} \times B_{K\times N} + \beta C_{M\times N}$. Both α and β are scalars. The total number of FLOPs in a GEMM operation 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 real and complex half-precision arithmetic in this paper. For real FP16 arithmetic (HGEMM), P is equal to 2, while for the Half-Complex GEMM (i.e., HCGEMM), P is equal to 4. The name HCGEMM does not follow the standard Basic Linear Algebra Subprograms (BLAS) notation, which assigns a single letter prefix to denote the precision. In fact, there is no consensus to date for naming linear algebra operations in half-complex precision. The term HCGEMM is used only within this paper, and may not be used in the final API of the released software.

The GEMM kernel is also important in many scientific domains other than dense linear algebra, especially where the computational workload can be broken down into a large number of small matrix operations. The workload is often called a "batched workload", and dedicated routines have been optimized for such

* Corresponding author.
E-mail address: ahmad@icl.utk.edu (A. Abdelfattah).

https://doi.org/10.1016/j.jpdc.2020.07.001
0743-7315/

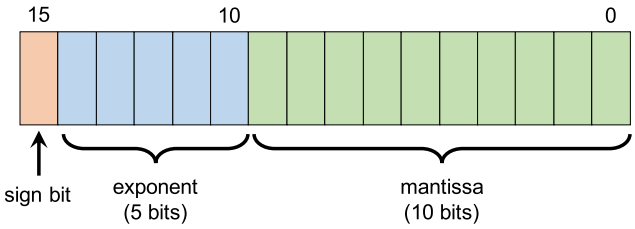


Fig. 1. Half precision format according to the IEEE-754 standard.

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 [32], and tensor contractions [1]. 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] The latter provides an optimized batched HGEMM kernel as well. However, for the complex case, the vendor advises the use of "split-complex" computations, which assumes that the real and the imaginary parts of the matrix are separated in a "planar layout".

The artificial intelligence (AI) and machine learning revolution has created a huge demand for 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 [10]. In terms of memory bandwidth and footprint, half precision theoretically offers a natural 2× improvement over single precision, and 4× over double precision. With respect to floating-point operations, the improvement factors over single and double precision depends on the architectural properties of the hardware. In this paper, we target NVIDIA's Volta GPUs, which offer native FP16 arithmetic. The theoretical peak performance for "general" FP16 computation is about 31.25 tera floating-point operations per second (teraFLOP/s). This is double the peak single precision performance of 15.6 teraFLOP/s, and four times the peak double precision performance of 7.8 teraFLOP/s. However, Volta GPUs offer even more performance for special operations in FP16 arithmetic. These operations are mostly variations of matrix multiplication, and are hardware-accelerated using special units called Tensor Cores (TCs). For such special operations, the performance of a single GPU can reach up to 125 teraFLOP/s. Although the Tensor Cores units first appeared in Volta GPUs, NVIDIA's first GPUs to ever support half precision were the Pascal GPUs. Half precision on NVIDIA GPUs implements the "binary16" format which is defined by the IEEE-754 standard [14]. As shown in Fig. 1, half precision uses one bit for the sign, five bits for the exponent, and ten bits for the fraction. However, the format assumes an implicit leading bit that is set to one unless the exponent is all zeros. This gives the FP16 format a total of eleven bits for the fraction, which accounts for an accuracy of about three decimal digits.

This paper investigates the use of the Tensor Cores to provide a general-purpose batched matrix multiplication in FP16 arithmetic. The paper extends the previous effort by the authors [4] by considering half-complex computation on the GPU, which is not natively supported to the best of our knowledge. While half-complex precision may be of limited use in the machine learning domain, it is of significant value in linear algebra, especially with respect to mixed-precision solvers. The paper addresses some challenges in using the Tensor Cores programmatically in a GPU kernel, such as discrete sizes and restricted thread configurations. The kernel design is expressed in terms of "building blocks", which are developed as device routines that perform specific tasks in the kernel. Some of these building blocks are shared across half and half-complex precisions, but some other functions had to be explicitly developed for each precision. An important goal was to provide highly flexible kernels that can withstand potential future changes to the Tensor Core technology. The developed kernels have at least eight tuning parameters that control different aspects of the kernel. An extensive tuning process has been applied to the developed kernels, with respect to typical use cases for batched GEMM operations. We also investigate the benefit of using Tensor Cores for extremely small problems, where full utilization of the Tensor Core units is not possible. While the vendor routine is very optimized for relatively large sizes, we observe that the batched HGEMM kernel outperforms cuBLAS for sizes $\leq$ 128 with speedups that range between 1.5× and 2.5×. We also show that the batched HCGEMM kernel outperforms the vendor solution (which uses split complex computation) by improvement factors between 1.7× and 7×. This work is part of the open-source MAGMA library [6].[3]

Below are the main highlights of the paper:

1. The use of FP16 arithmetic is proven to be useful for numerical linear algebra, and so important kernels like batched matrix multiplications must be optimized. This kernel is the focus of the paper.
2. The use of Tensor Cores is restricted by some limitations that are imposed by the programming model.
3. The proposed kernel design builds a flexible abstraction layer over the tensor cores. Such a layer hides the aforementioned restrictions.
4. The final kernel design has 8 tuning parameters. A comprehensive tuning sweep is conducted to record the best performance at different sizes.
5. For half-complex computations, we theoretically prove that the interleaved layout is better for performance than planar layouts (split-complex).
6. The performance gains for batch GEMM in half precision are between 1.5× and 2.5× for sizes up to 128.
7. The performance gains for batch GEMM in half-complex precision are between 1.7× and 7× for sizes up to 256.
8. For tiny matrices, we show that the use of Tensor Cores might be questionable due to the sub-optimal utilization of the Tensor Core units.

2. Related work

Matrix multiplication is an embarrassingly parallel operation with a relatively high operational intensity. These two properties enable GEMM operations to run asymptotically at 90+% of the GPU's peak performance. This good match between the properties of GEMM and GPUs being throughput-oriented processors has led to the emergence and success of GPU-accelerated dense linear algebra. Research efforts date back almost a decade, 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 [29]. Since then, the GEMM kernel has been subject to continuous improvements, like register and shared-memory blocking and prefetching [23]. Such developments sparked many efforts in providing fast high-level dense linear solvers on GPUs, such as the MAGMA library [27], ViennaCL [25], and Chameleon [5]. Performance portability of GEMM was achieved through performance-critical tuning parameters that control different properties of the GEMM design [18,

[1] https://software.intel.com/mkl
[2] https://developer.nvidia.com/cublas
[3] https://icl.cs.utk.edu/magma/

21]. 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 [26] in order to overcome some limitations imposed by the compiler and the hardware scheduler. Similarly, assembly implementations [9,19] 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.

All the aforementioned efforts address the problem of one GEMM operation that is relatively large enough to provide sufficient parallel work for the GPU. The recent application-driven interest in batched 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 [12]. 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 [32], tensor contractions [1,15], and machine learning [8]. 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 batched routines, since it has been found that the performance is more sensitive to tuning parameters in small matrix problems [2].

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 (DNNs). They were initially implemented using custom dense kernels, as originally done in Caffe [16] 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 [8], MagmaDNN [24], 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 [7,8]. In addition, the work done in [20] uses batched GEMMs of very small sizes (3×3) to implement fast CNN algorithms based on minimal filtering algorithms [31]. 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 [10]. 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 [28]. 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 [11,13], 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 GEMM routine that is competitive with the vendor optimized library.

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

3. 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. 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 Volta and Turing architectures introduce 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. As an example, each Volta V100 GPU has 640 Tensor Cores, evenly distributed across 80 multiprocessors. Each Tensor Core possesses a mixed-precision $4 \times 4 \times 4$ 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. As an example, the routine `cublasHgemmBatched` implements the batched GEMM operation for real FP16 arithmetic. All matrices are assumed to have the same dimensions. Considering complex FP16 computations, there is no native support for half-complex precisions yet in the library. In fact, the only way to use cuBLAS is to use a "planar layout" for the matrices, where the real and the imaginary parts of the matrices are separated. The planar layout enables an easy solution using existing cuBLAS routines, but it lacks an important performance advantage, which will be discussed later in the paper. The same concept of split-complex computation applies to the cuBLASLt library,[5] as well as the open-source CUTLASS library.[6]

Taking advantage of the Tensor Cores in a custom kernel is possible through the use of low-level APIs that are provided by the programming model as well. As shown in Fig. 2, Tensor Cores deal with input and output data through opaque data structures 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 an output fragment into the shared/global memory of the GPU. The `mma_sync()` API is used to perform the multiplication. The user is responsible for declaring the fragments as required, and calling the APIs in the correct sequence.

The programming model imposes some restrictions to the programming of the 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 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. The latest CUDA version to date (10.1) provides direct access to the Tensor Cores through an instruction called `mma.sync`.

[5] https://docs.nvidia.com/cuda/cublas/index.html#using-the-cublasLt-api
[6] https://github.com/NVIDIA/cutlass

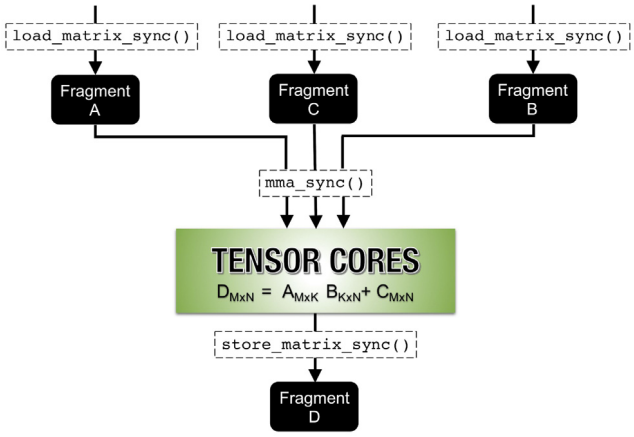


Fig. 2. Programmability of the Tensor Core units.

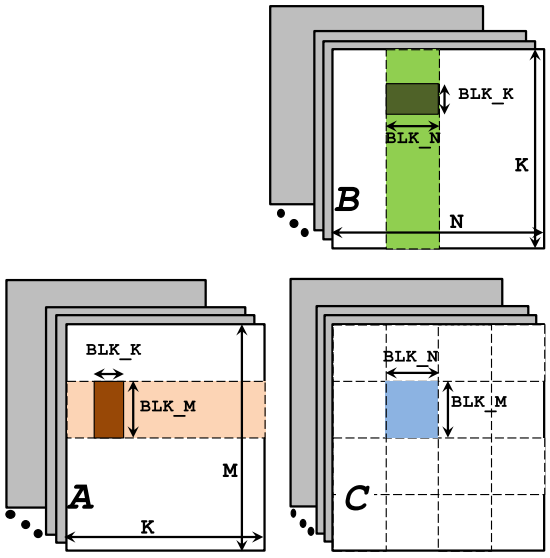


Fig. 3. Organization of the batched GEMM grid.

The instruction allows one warp to perform four independent GEMM operations of size (8, 8, 4). However, using the explicit instruction may lead to long-term compatibility issues as new architectures are released. This is why we decide not to consider using parallel thread execution (PTX) instructions, and focus only on the device-level CUDA APIs.

4. General design outlines

This section describes some general concepts about the design of the batched HGEMM and the batched HCGEMM kernels. The dimensions (M, N, K) of the GEMM operation are assumed to be unified across the batch. According to the CUDA programming model, a GPU kernel is, in general, a three-dimensional grid of three-dimensional thread blocks (TBs). The number of GEMM operations in the batch is referred to as `batchCount`. For real half-precision computation, the CUDA data type `__half` is used. For half-complex computations, we use the `__half2` vector type. The low 16-bits represent the real part, while the high 16-bits represent the imaginary part.

4.1. Grid design

The MAGMA library uses a common grid design for all of its batched kernels [2,3]. 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 first two dimensions of the grid are used to denote a two-dimensional $\lceil \frac{M}{\texttt{BLK\_M}} \rceil \times \lceil \frac{N}{\texttt{BLK\_N}} \rceil$ "subgrid" for each output matrix in the batch. The third grid dimension is used for batching across the problems, which yields a three-dimensional grid configuration of ($\lceil \frac{M}{\texttt{BLK\_M}} \rceil$, $\lceil \frac{N}{\texttt{BLK\_N}} \rceil$, `batchCount`). 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, 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. The block rows/columns are read in steps of `BLK_K`. At each iteration of the main loop, a TB multiplies a block of A (`BLK_M`×`BLK_K`) with a block of B (`BLK_K`×`BLK_N`). Since we are using the Tensor Core units, the accumulations of the partial results take place in the fragments rather than regular register buffers or shared memory. Fig. 3 illustrates the TB organization of the kernel.

The following sections describe the main design aspects of the kernel, which leverages some design concepts from existing kernels [2] while modifying or generalizing them to take advantage of the Tensor Cores.

5. Detailed thread block design

5.1. Abstracting tensor cores

As mentioned before, the use of the Tensor Cores programmatically must follow some constraints that are required by the device-level APIs. Our goal in this paper is to design GPU kernels with an abstraction layer over the Tensor Cores. The abstraction layer takes care of the device-API constraints and provides a general-purpose use of the cores. The three main constraints for using Tensor Cores are:

1. Tensor Cores can be used only with three discrete combinations of blocking sizes
2. Device-level APIs must be called by a single warp
3. Loading/storing fragments must be from/to memory spaces with specific leading dimensions (multiples of 16 bytes).

The abstraction layer proposed in this paper addresses each one of these constraints. Eventually, the GPU kernels will be able to use arbitrarily large blocking sizes, using any number of warps, with no constraints on the leading dimension of the matrices.

The developed solution must support arbitrary leading dimensions for A, B, and C in the global memory of the GPU. A straightforward solution is to read the matrices into shared-memory buffers, rather than reading them directly into the Tensor Core fragments. The shared-memory buffers are allocated with leading dimensions that abide by the 16-byte rule.

5.2. Double-Sided Recursive Blocking (DSRB)

This technique allows GPU kernels to use arbitrarily large blocking sizes (`BLK_M`, `BLK_N`, `BLK_K`) that are not necessarily restricted to the Tensor Core sizes (`TC_M`, `TC_N`, `TC_K`). Recursive blocking is a well-known technique that has been used for years in previous GEMM designs [23]. It transfers each block of A, B, and C to/from the global memory using a two-dimensional thread configuration `DIM_X`×`DIM_Y`. These blocks are subdivided into smaller `DIM_X`×`DIM_Y` tiles that can be stored in shared memory or in the register file. The same `DIM_X`×`DIM_Y` subdivision is used for computing the partial products of these blocks. We call this technique *single-sided recursive blocking (SSRB)*. Such a technique

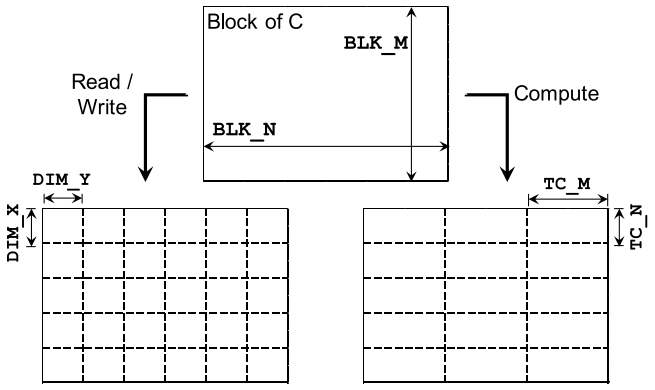


Fig. 4. An example of the double-sided recursive blocking (DSRB) applied to a block of C. The blocking dimensions for reads and writes are decoupled from those used for computations.

is not applicable to Tensor Cores, which have discrete blocking sizes for computation. This is why we propose a generalization over SSRB, which we call *double-sided recursive blocking (DSRB)*. As shown in Fig. 4, it simply decouples the way blocks are read/written from the way they are passed to/from the Tensor Core units. During the read/write of a block from/to the global memory, a matrix block is always subdivided into DIM_X×DIM_Y tiles. During the computation of a partial product, however, the loaded blocks are subdivided using the TC sizes (TC_M, TC_N, TC_K). Thread regrouping is also used to select the best configuration at each stage of the kernel. As an example, assume a single warp configuration in a kernel where all blocking sizes are equal to 16. When loading a 16 × 16 block of data, it is much better to use a 16 × 2 or 8 × 4 configuration rather than the default 32 × 1 one. This is why we allow a single warp to reorganize itself into a DIM_X×DIM_Y configuration when reading and writing blocks of data. During the computation, the regular 32 × 1 configuration is used.

5.2.1. How does DSRB improve memory traffic?

The DSRB technique generalizes BLK_M, BLK_N, and BLK_K so that they are not bound to the TC sizes. Generic block sizes help improve the memory traffic required by the GEMM kernel. To illustrate this point further, we simplify our analysis by assuming that BLK_M and BLK_N fully divide M and N, respectively. According to our grid configuration in Section 4.1, the proposed kernels would require $\frac{M \times N}{\text{BLK\_M} \times \text{BLK\_N}}$ TBs. Theoretically, an ideal implementation would read A, B, and C from the global memory exactly once, and write C once. The proposed kernels perform an ideal memory traffic for C, since each TB 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 TBs, 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 TB 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:

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

Fig. 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 theoretically reduce the memory traffic by a factor of 4× when (TC_M, TC_N) = (16, 16), and by a factor of 5× 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). However, this theoretical analysis assumes infinite resources available for each TB. Using too many resources per TB could actually worsen other aspects of the kernel, such as the occupancy and the register pressure. Therefore, a tuning process is usually required to search for the best blocking sizes on a specific GPU.

5.3. Two-stage loading of input data

Instead of loading the blocks of A and B directly into the shared-memory buffers, we use a two-stage process where the data is first read in register buffers, then offloaded to the shared memory. The use of double buffers would allow us incorporate a prefetching mechanism into the register file while the data are being processed through the buffers. Two device-level functions have been developed for this purpose. The first one reads a block of data from the global memory into the register file.

```
template<typename T,
const int DIM_X, const int DIM_Y,
const int BLK_R, const int BLK_C>
static __device__ __inline__ void
read_global2reg(
    const int blk_m, const int blk_n,
    const T* __restrict__ A, int LDA,
    T reg[BLK_C/DIM_Y][BLK_R/DIM_X],
    const int tx, const int ty)
{
  int m, n;
  #pragma unroll
  for(n = 0; n < BLK_C; n+=DIM_Y) {
    #pragma unroll
    for(m = 0; m < BLK_R; m+=DIM_X) {
      reg[n/DIM_Y][m/DIM_X]
       = fetch<T>(/* address info */);
    }
  }
}
```

The function is templated for type, block sizes (BLK_R × BLK_C), as well as the thread configuration (DIM_X×DIM_Y). These parameters are known at compile time, which enables fully unrolled loops, and avoids register spilling into the local memory. The `fetch<T>()` function returns either the required element, or zero if the passed address is out-of-bound. This function is one of the shared building blocks across half and half-complex kernels.

The second device function offloads the content of a register buffer into a shared-memory space. A similar structure of two nested loops is used.

```
template<const int DIM_X, const int DIM_Y,
         const int BLK_R, const int BLK_C>
static __device__ __inline__ void
store_reg2smem(
  __half rA[BLK_C/DIM_Y][BLK_R/DIM_X],
  __half* sA,
  const int tx, const int ty)
{
  int m, n;
  #pragma unroll
  for (n = 0; n < BLK_C; n+=DIM_Y) {
    #pragma unroll
    for (m = 0; m < BLK_R; m+=DIM_X) {
      sA[(n+ty) * BLK_R + (m+tx)] = rA[n/DIM_Y][m/DIM_X];
    }
  }
}
```

Unlike the previous device function, the `store_reg2smem` function cannot be used as it is for half-complex buffers. Since the TCs support only real arithmetic, we need to split the contents of the register file into two separate memory spaces; one for the real part, and the other for the imaginary part. The function below shows the corresponding `stroe_reg2smem_complex` function, which uses low-level intrinsics for splitting the values.

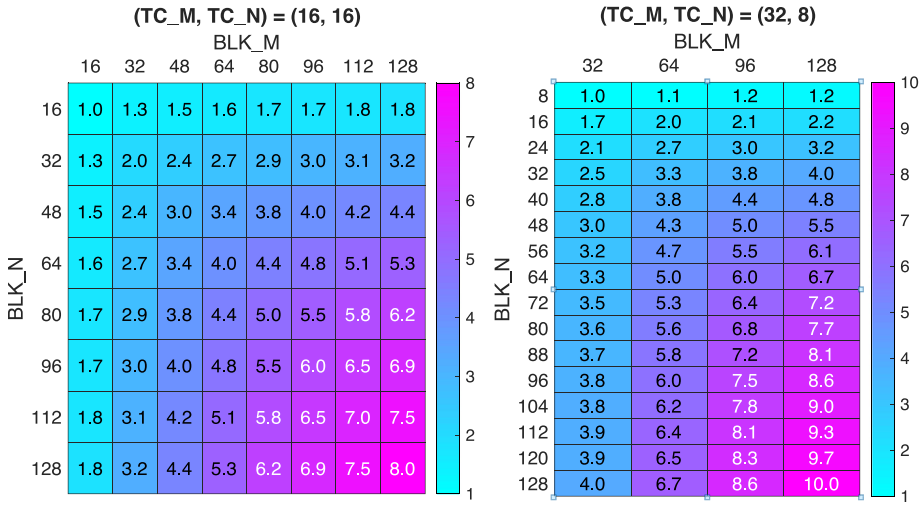


Fig. 5. The relative reduction in memory traffic for A and B using (BLK_M, BLK_N) $\geq$ (TC_M, TC_N). Results are shown for (TC_M, TC_N) = (16, 16) and (32, 8). Note that BLK_x must always be multiple of TC_x.

```
template<const int DIM_X, const int DIM_Y,
         const int BLK_R, const int BLK_C>
static __device__ __inline__ void
store_reg2smem_complex(
  __half2 reg[BLK_C/DIM_Y][BLK_R/DIM_X],
  __half* sAr, __half* sAi,
  const int tx, const int ty)
{
  int m, n;
  #pragma unroll
  for (n = 0; n < BLK_C; n+=DIM_Y) {
    #pragma unroll
    for (m = 0; m < BLK_R; m+=DIM_X) {
      sAr[(n+ty) * BLK_R + (m+tx)]
       = __low2half ( reg[n/DIM_Y][m/DIM_X] );
      sAi[(n+ty) * BLK_R + (m+tx)]
       = __high2half( reg[n/DIM_Y][m/DIM_X] );
    }
  }
}
```

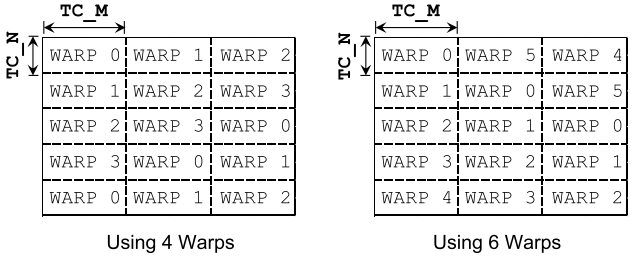


Fig. 6. Multi-warp configuration with round-robin assignment for the output C block.

5.4. Multi-warp configuration

All TC device functions must be invoked using one warp. However, this does not necessarily mean that the TB configuration should be restricted to a single warp. In fact, it is sometimes beneficial to use multiple warps per TB, especially when the amount of work per TB is relatively large, or when a warp is stalled. We generalize our thread configuration to support any number of warps. As mentioned before, threads reorganize themselves into a DIM_X×DIM_Y configuration during memory operations. During computation, however, we must reorganize the threads in a 32 × N_WARPS configuration in order to use the TC. Note that the parameter space for DIM_X and DIM_Y is now much bigger, which serves the design flexibility. As an example, four warps can be used in many configurations, such as 8 × 16, 16 × 8, 32 × 4, 64 × 2, ··· etc. When a partial product is being computed, the block accumulator 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 row of A and the sub-block column of B, sends them in chunks to the Tensor Cores, and keeps accumulating the results in its respective fragment. Fig. 6 shows the workload distribution for two different configuration on a block accumulator that has 15 sub-blocks.

The use of multiple warps enables controlling the amount of work per warp, especially for large blocks of data. Reading in large blocks is usually required to achieve a high memory bandwidth and to increase data reuse. But since TC 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 batched HGEMM on square 64 × 64 matrices. 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. Fig. 7 shows that increasing the number of warps per TB leads to significant performance gains when the blocks are relatively large (while fixing all other parameters). However, the performance drops if too many warps are used per TB. One reason is the 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 Fig. 7 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.

5.5. Performing the multiplication using the tensor cores

After loading the input data blocks in the shared-memory buffers, another device function (tc_multiply()) is invoked to perform the TC multiplication using the round-robin style illustrated before. The function is heavily templated with a number of constants that are known at compile time, such as TC_BLOCKS, NWARPS, and NFRAG. The parameter 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 pseudocode

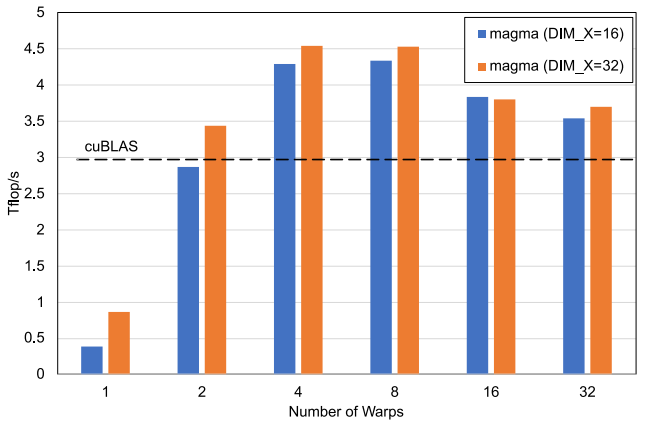


Fig. 7. The impact of the number of warps on performance for relatively large blocking sizes. Results are shown for square 64 × 64 matrices using a Tesla V100 GPU, with `batchCount` = 500. CUDA version is 10.1. All blocking sizes are set to 64, with all Tensor Core sizes set to 16. `DIM_Y` = (#warps×32)/`DIM_X`.

below 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 TC 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]);
        }
    }
}
```

The `tc_multiply()` function does not work for half-complex arithmetic, and a dedicated function is required to do a split-complex multiplication on the TB level, which we call `tc_multiply_complex()`. The function accepts two shared-memory pointers (real and imaginary) for each of the A and B blocks. The innermost loop performs four multiplications instead of one. The partial results are accumulated in separate output fragments.

```
template<...>
__device__ __inline__ void
tc_multiply_complex(
    half* sAr, half* sAi, half* sBr, half* sBi,
    wmma::fragment<...> fCr[NFRAG],
    wmma::fragment<...> fCi[NFRAG] )
{
    // Declare A, B fragments
    wmma::fragment<...> fA;
    wmma::fragment<...> fB;

    #pragma unroll
    for(int b = 0; b < nblks; b += NWARPS){
        (i, j, fid ) <- get_next_frag_indices ( warp_id );
        #pragma unroll
        for(int k = 0; k < BLK_K; k+=TC_K){
            half* ptrAr = sAr + k * BLK_M + i;
            half* ptrAi = sAi + k * BLK_M + i;

            half* ptrBr = sBr + j * BLK_K + k;
            half* ptrBi = sBi + j * BLK_K + k;

            // sAr * sBr -> real
            wmma::load_matrix_sync(fA, ptrAr, BLK_M);
            wmma::load_matrix_sync(fB, ptrBr, BLK_K);
            wmma::mma_sync(fCr[fid], fA, fB, fCr[fid]);

            // sAr * sBi -> complex
            wmma::load_matrix_sync(fB, ptrBi, BLK_K);
            wmma::mma_sync(fCi[fid], fA, fB, fCi[fid]);

            // sAi * sBr -> complex
            wmma::load_matrix_sync(fA, ptrAi, BLK_M);
            wmma::load_matrix_sync(fB, ptrBr, BLK_K);
            wmma::mma_sync(fCi[fid], fA, fB, fCi[fid]);

            // -sAi * sBi -> real
            wmma::load_matrix_sync(fB, ptrBi, BLK_K);
            #pragma unroll
            for(int t=0; t<fA.num_elements; t++)
                fA.x[t] *= (half)(-1.0);
            wmma::mma_sync(fCr[fid], fA, fB, fCr[fid]);
        }
    }
    /* cleanup code similar to the real case */
}
```

5.6. Post processing

Recall that the GEMM operation is defined as $C_{M\times N} = \alpha A_{M\times K} \times B_{K\times N} + \beta C_{M\times N}$. So far, all of the different computational stages serve for computing the $A \times B$ product. The scaling operations by α and β are performed at a post-processing stage. For the batched HGEMM kernel, the post-processing stage loads the respective block of C into a register buffer and scales it by β. In order to save memory traffic, this step takes place only if β is a non-zero. The product $A \times B$ resides in shared memory. It is scaled by α and added to the register buffer of C before finally writing it to the global memory. As for the half-complex case (batched HCGEMM), recall that the `tc_multiply_complex()` function has the real and imaginary parts of the product separated. The product $A \times B$ is therefore merged first in the shared memory before any scaling takes place. The rest of the post-processing step is similar to the batched HGEMM kernel.

5.7. The main kernel structure

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 (in two stages). The actual dimensions of these blocks (`am, an, bm, bn`) are computed at the beginning of the iteration so that the `read_global2reg()` 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()`. The multiplication takes place simultaneously while reading a new pair for data blocks. Another synchronization point is required to make sure all warps are done with the currently loaded data, and that it is safe to overwrite the contents of the shared memory.

```
(am, an) <- compute_block_size( A, kk );
(bm, bn) <- compute_block_size( B, kk );

rA[][] <- read_global2reg<...>(am, an, A, LDA);
rB[][] <- read_global2reg<...>(bm, bn, B, LDB);

for (int kk = 0; kk < K; kk += BLK_K) {
  sA[] <- store_reg2smem( rA[][] );
  sB[] <- store_reg2smem( rB[][] );

  sync<...>();
  if(/* not last iteration*/) {
    // prefetch
    A  += BLK_K * LDA;
    B  += BLK_K;
    (am, an) <- compute_block_size( A, kk );
    (bm, bn) <- compute_block_size( B, kk );
    rA[][] <- read_global2reg<...>(am, an, A, LDA);
    rB[][] <- read_global2reg<...>(bm, bn, B, LDB);
}

  tc_multiply<...>(sA[], sB[], fC[]);
  sync<...>();
}

// post processing step
rC[][] <- post_process( fC[], sC[]);

// write output
wrire_reg2global( rC[][], C[] );
```

5.8. Tuning Parameters

The developed kernels are written using CUDA C++ templates, with eight main tuning parameters. These parameters are:

- The configuration sizes of the Tensor Cores (`TC_M`, `TC_N`, `TC_K`). These sizes are discrete configurations that are enforced by the programming model.
- The blocking sizes for A, B, and C (`BLK_M`, `BLK_N`, `BLK_K`). These sizes control the amount of data reuse by improving the memory traffic, as explained in Section 5.2.1. They also influence the shared memory requirement for the kernel.
- The thread configuration for reading and writing data blocks (`DIM_X`, `DIM_Y`). These parameter control the number of threads processing the data blocks of A, B, and C. They also influence the register pressure per thread. Assuming that the blocking sizes (`BLK_M`, `BLK_N`, `BLK_K`) are not changed, increasing the number of threads leads to fewer registers per thread.

The tuning parameters must satisfy a number of conditions in order for the kernel to perform correctly. The 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. We chose to run a comprehensive brute-force tuning sweep for all eligible kernel instances. Our reasoning behind that decision was to get the best performance for small sizes, where the performance is more sensitive to the tuning parameters than for large sizes [2].

6. Half-complex batched GEMM: Planar vs. interleaved layouts

In order to perform half-complex computations using the cuBLAS library, a user must use "split-complex" computation. The real and the imaginary parts of A, B, and C must be stored separately in a planar layout. Considering dense linear algebra algorithms, planar layouts do not help achieve good performance, especially for relatively small sizes. **First**, all the existing linear algebra numerical libraries assume an *interleaved layout*, meaning that the real and the imaginary parts of each element are contiguous in memory. It is not practical to rewrite entire algorithms using split-complex computations to make use of the TCs. **Second**, while it is relatively easy to develop the GEMM kernel in planar layouts using the existing GEMM kernels, other linear algebra components might not be as straightforward. Examples are triangular solve and the pivoting stage in the LU factorization. For such operations, it is not possible to use the existing triangular solve or pivoting kernels, which leads to new developments at the BLAS level. Therefore, it is more convenient to use the standard interleaved layout. **Third**, complex compute-bound kernels normally reach the peak performance of the underlying hardware earlier than their counterparts that use real arithmetic. This is because complex kernels have more *operational intensity* than real arithmetic kernels. The operational intensity of an operation is the ratio between the number of FLOPs and the number of bytes transferred for that operation. As an example, the real FP16 scalar operation ($c = c + a \times b$) costs 2 FLOPs (1 addition and 1 multiplication), which results in $\frac{2}{8} = 0.25$ FLOP/byte ratio. Using complex FP16 arithmetic, the same operation would cost 8 FLOPs. The product $a \times b$ costs 6 FLOPs (4 multiplications and 2 additions), and the update of c costs 2 more additions. This results in $\frac{8}{16} = 0.5$ FLOP/byte ratio (double the operational intensity of the real scalar operation). Such a difference in the operational intensity leads to a "slowly growing" theoretical peak performance (i.e. roofline) for the HCGEMM kernel, thus requiring relatively large sizes to approach the hardware peak performance. In other words, the planar layout makes the HCGEMM kernel operate at the same roofline as the real arithmetic HGEMM kernel, since it calls HGEMM four times.

We theoretically investigate the half-complex GEMM operation based on the Roofline model [30]. Our focus is on batched square multiplications ($M = N = K$), as well as batched rank-k updates ($M = N$, K is a relatively small constant). The former use case demonstrates a test for the peak performance of the kernel, while the latter is an important use case in batched linear algebra, especially in the batched LU factorization.

6.1. Roofline for HCGEMM (standard interleaved layout)

For simplicity of our analysis, we can safely ignore the scaling by α and β. According to the LAPACK Working Note #41,[7] a standard GEMM operation ($C_{M\times N} = C_{M\times N} + A_{M\times K}B_{K\times N}$) involves ($M \times N \times K$) additions, and ($M \times N \times K$) multiplications. This can be explained as follows. The output C matrix has $M \times N$ elements. Each element c_{ij} is computed as $c_{ij} = c_{ij} + \sum_{k=1}^{K} a_{ik}b_{kj}$, which accounts for K multiplications and K additions. Therefore, the HCGEMM-interleaved operation requires (MNK) additions and (MNK) multiplications. In order to estimate the number of FLOPs, we recall that one multiplication of complex numbers requires 6 FLOPs and one addition requires 2 flops. The total number of flops is, therefore, ($8MNK$).

In order to derive a roofline (i.e. a performance upper-bound), we derive the ideal (minimum) amount of memory traffic for the operation. The ideal amount of data required for HCGEMM is

1. One read and one write for C, which equals ($2 \times M \times N$)
2. One read for A ($M \times K$)
3. One read for B ($K \times N$)

Considering half-complex precision, each element is stored in 4-bytes. Therefore, the ideal amount of bytes transferred is $4 \times [2MN + K(M + N)]$. Now we can write:

$$\texttt{Operational intensity of HCGEMM-interleaved} = \frac{2MNK}{2MN + K(M + N)} \quad (1)$$

[7] http://www.netlib.org/lapack/lawns/lawn41.ps

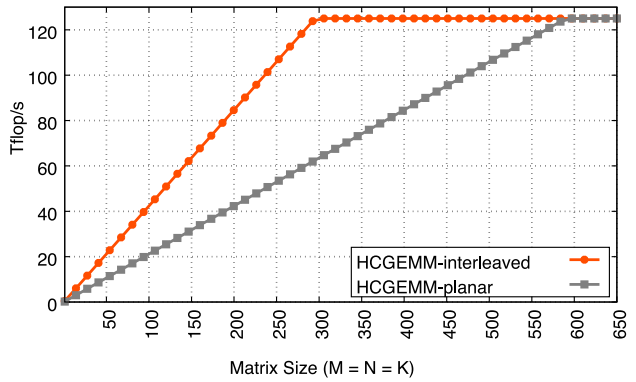


Fig. 8. Performance upper bound for batched HCGEMM on square sizes. The analysis is based on a 847 GB/s peak bandwidth (STREAM benchmark on a Tesla V100-PCIe GPU).

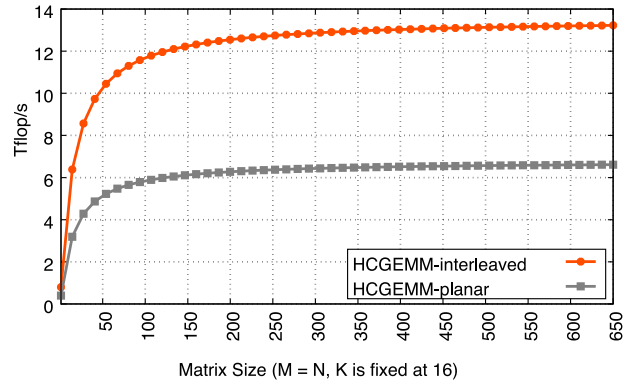


Fig. 9. Performance upper bound for batched HCGEMM for rank-16 updates. The analysis is based on a 847 GB/s peak bandwidth (STREAM benchmark on a Tesla V100-PCIe GPU).

6.2. Roofline for HCGEMM (planar layout)

We now estimate the roofline for the HCGEMM-planar operation similar to Section 6.1. For a split-complex computation, the operation can be done via four calls to the standard HGEMM operation. Assuming that we have two separate output matrices C_r (real) and Ci (imaginary), the four HGEMM calls are:

1. First call: $C_r = C_r + A_r \times B_r$
2. Second call: $C_r = C_r - A_i \times B_i$
3. Third call: $C_i = C_i + A_r \times B_i$
4. Fourth call: $C_i = C_i + A_i \times B_r$

Each call involves (MNK) additions and (MNK) multiplications. Since these calls perform real half-precision arithmetic, each addition/multiplication accounts for one FLOPs. Therefore, one call performs $(2MNK)$ FLOPs, leading to a total of $(8MNK)$ FLOPs. From an operation count point of view, these four calls have the same complexity as HCGEMM-interleaved. However, the collective memory traffic for these calls is different. Each call to HGEMM involves $(2MN)$ traffic for C, an (MK) traffic for A, and an (KN) traffic for B. The memory traffic for one HGEMM call is 2-bytes$\times[2MN + K(M + N)]$, leading to an aggregate traffic of $8 \times [2MN + K(M + N)]$. Similarly, we can write

$$\texttt{Operational intensity of HCGEMM-planar} = \frac{MNK}{2MN + K(M + N)} \quad (2)$$

To summarize, HCGEMM-interleaved has double the operational intensity of HCGEMM-planar. For a square multiplication $(M = N = K)$, the operational intensity is simplified to $0.25N$ for HCGEMM-planar, and to $0.5N$ for HCGEMM-interleaved. The roofline model [30] estimates the performance upper bound as `operational intensity` $\times$ `the peak memory bandwidth`. Using a GPU STREAM benchmark on the Tesla V100 GPU, we got up to 847 GB/s of peak memory bandwidth. Fig. 8 shows the performance upper bounds for HCGEMM-planar and HCGEMM interleaved for square matrices. Both graphs grow linearly with the matrix size until they hit the GPU peak performance. However, due to the increased operational intensity, HCGEMM-interleaved reaches the peak earlier than HCGEMM-planar. This is very crucial for relatively small sizes. As an example, HCGEMM-planar is bandwidth-limited at size 300, while HCGEMM-interleaved becomes compute bound. For large sizes (larger than 550 in Fig. 8), we may not see a difference between the two kernels (if properly optimized), since HCGEMM-planar will be able to saturate the GPU anyway.

Another useful test case for GEMM is the rank-k updates. This is a very important use case in dense linear algebra. Fig. 9 shows a similar roofline analysis for updating a square matrix $(M = N)$, with a rank-16 update $(K = 16)$. As per the roofline model, this is a use case that always remains bandwidth-limited on the V100 GPU, regardless of the size. The operational intensity of HCGEMM-planar will be given by $\frac{8N^2}{N^2+16N}$, while the HCGEMM-interleaved has its own at $\frac{16N^2}{N^2+16N}$. This means that HCGEMM-interleaved is theoretically $2\times$ faster than HCGEMM-planar, no matter how large M and N are.

7. Performance results

This section shows the performance results of the proposed MAGMA kernels against the equivalent implementations by the vendor library (cuBLAS).

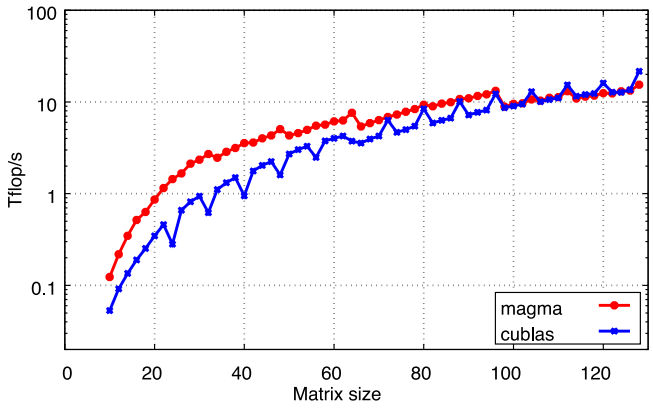
7.1. Experimental setup

The performance tests are conducted on a system equipped with a two-socket CPU (Intel Xeon E5-2650 v3 @ 2.30 GHz), with 10 cores per socket, and a Tesla V100-PCIe GPU. The GPU has 16 GB of memory, and 80 streaming multiprocessors, which are clocked at 1.38 GHz. We use CUDA Toolkit 10.1 for the compilation of the MAGMA kernels, as well as for testing cuBLAS. The tests are done for relatively small square sizes and for relatively small rank-16 updates.

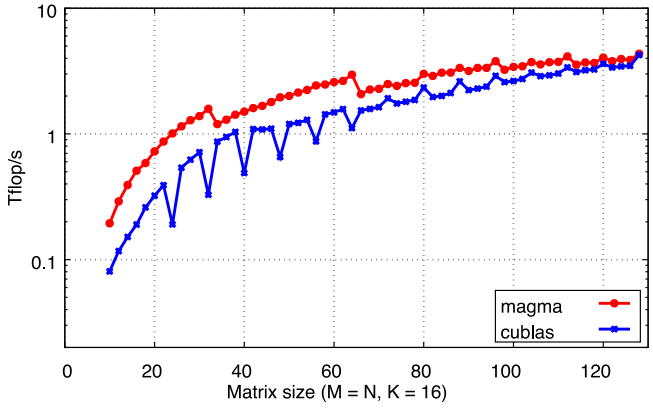
7.2. Performance of the batched-HGEMM kernel

Fig. 10 shows the performance of the tuned MAGMA kernel against cuBLAS for the batched HGEMM operation. Performance speedups are observed for the MAGMA kernel for sizes up to 100 on square sizes and for sizes up to 128 for the rank-16 updates. This behavior shows the importance of auto-tuning. For small problem sizes, we notice that the performance of a given kernel is sensitive to tuning parameters in the sense that more kernel instances are required for relatively small problems. This is unlike the situation for larger sizes, where usually one or two kernel instances can deliver the best performance.

Fig. 11 summarizes the performance speedup for every size between 10 and 128. On average, the speedup numbers range between $1.5\times$ and $2.8\times$, except for few spikes or drops. The spikes/drops in speedup are mainly due to the cuBLAS performance behavior, which seem to have periodic drops for some sizes ≤ 64, and some other performance spikes after that.



(a) Square sizes



(b) Rank-16 updates

Fig. 10. Performance of the batched HGEMM kernel against cuBLAS. Results are for square sizes up to 128, with `batchCount` = 1000, on a Tesla V100-PCIe GPU.

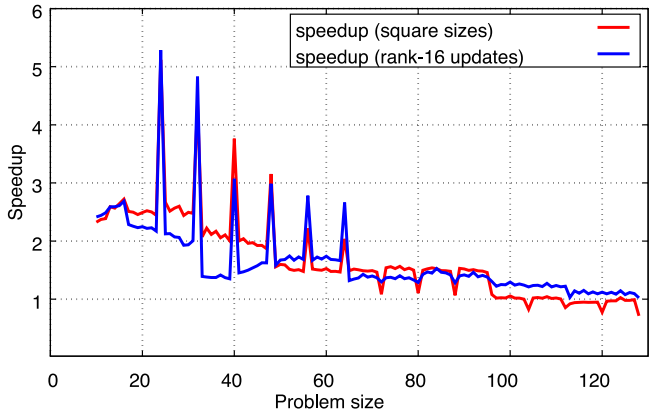


Fig. 11. Performance speedup of the batched HGEMM kernel against cuBLAS. Results are for sizes up to 128, with `batchCount` = 1000, on a Tesla V100-PCIe GPU.

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 greater than 2× for large matrices. However, the significant cuBLAS advantage is observable only for sizes that are multiples of 8. As an example, for a batch of 100 square problems of size 2000, the cuBLAS kernel is 2.3× faster than MAGMA (68.8 teraFLOP/s for cuBLAS vs. 29.5 teraFLOP/s for MAGMA). However, the same batch for sizes of 2100 × 2100 sees a significant drop for cuBLAS vs. a slight one for MAGMA. In fact, MAGMA has a slight 3% advantage in this case (26.5 teraFLOP/s for cuBLAS vs. 27.4 teraFLOP/s for MAGMA). In general, large sizes that are not multiples of 8 witness competitive performance numbers from both libraries.

7.3. Performance of the batched-HCGEMM kernel

Fig. 12 shows the performance of the batched HCGEMM kernels, while Fig. 13 shows the respective relative speedups. Recall that the cuBLAS library uses a planar layout where the real and the imaginary parts of the matrices are stored in separate memory spaces. However, the reported results do not include the timing for separating and merging these components. We only compare the execution time of the computational kernels with no overheads. As per the theoretical analysis in Section 6, there is an advantage to using interleaved layouts over planar layouts. The increased operational intensity in the former gives a performance advantage for the MAGMA kernel on a wider range of sizes. In fact, the speedup numbers reported in Fig. 13 are much more significant than those reported in Fig. 11. The reported speedups for the MAGMA batched HCGEMM kernel are in the range between 1.7× and 7×.

7.4. Impact of batch size on performance

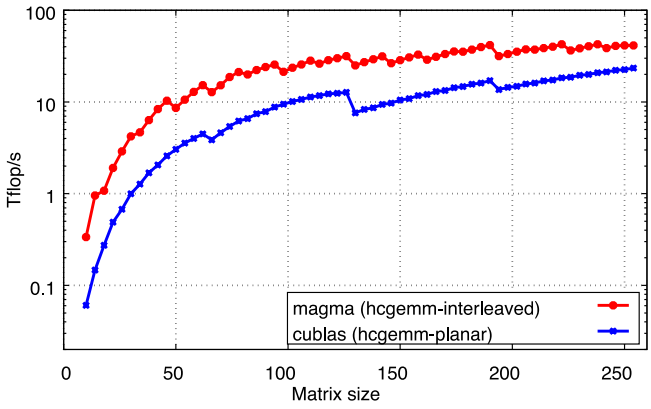
The performance of batched kernels often depends on the batch size. So far, we have tested the batched kernels on relatively large batches, which saturate the GPU with enough parallel work. Such experiments are conducted to test how much of sustained peak performance is achievable by the kernels. However, applications may not necessarily use large batches, and the typical sizes vary from one application to another. This is why we show the performance of the batched HGEMM/HCGEMM kernels on relatively small batches that may not necessarily occupy all of the GPU resources.

Figs. 14 and 15 show the performance for two different batch sizes (10 and 100), for the batched HGEMM and HCGEMM kernels, respectively. We highlight square sizes only, since the behavior for rank-k updates is the same, and hence was omitted to avoid redundancy.

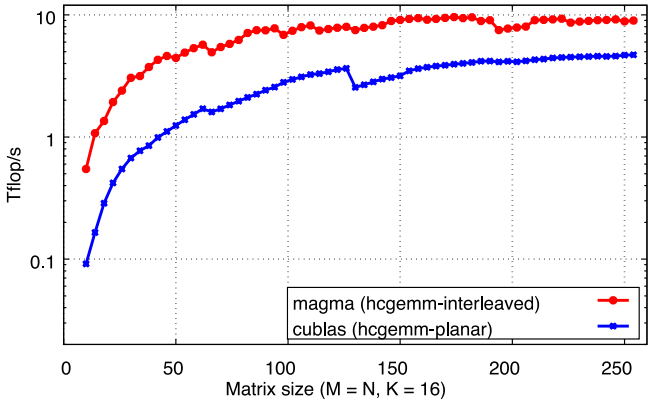
As expected, a small batch leads to a slower performance, since less parallel work is available for the GPU. However, the relative performances between MAGMA and cuBLAS in both figures are very similar to the asymptotic behaviors observed in Figs. 10 and 12. As an example, the behavior of the batched HGEMM kernel in Fig. 14 shows two similar pairs of graphs. Increasing `batchCount` from 10 to 100 results in shifting up the performance graphs without a significant change in the relative speedups. The exact same behavior is observable in Fig. 15 for the batched HCGEMM kernel. However, since the MAGMA HCGEMM kernel has a better arithmetic intensity, we can observe that the MAGMA performance on 10 operations is sometimes equivalent to the cuBLAS performance on 100 operations. Such a behavior emphasizes the advantage of the interleaved layouts for dense matrix computations, especially for batch workloads.

8. Optimization for extremely small matrices

The Tensor Core APIs have discrete configurations (`TC_M`, `TC_N`, `TC_K`). For batched GEMM problems with sizes smaller than these configurations, the TC utilization is below 100%, and depending on the problem size, the use of the TCs might be questionable. This section focuses on performance optimization



(a) Square sizes



(b) Rank-16 updates

Fig. 12. Performance of the batched HCGEMM kernel (interleaved layout) against cuBLAS (planar layout). Results are for square sizes up to 256, with `batchCount` = 1000, on a Tesla V100-PCIe GPU.

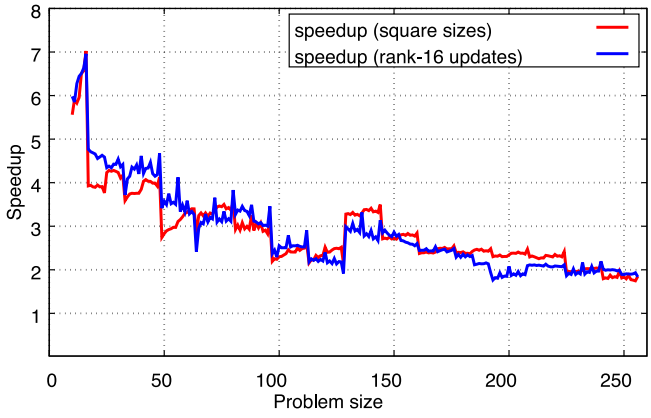


Fig. 13. Performance speedup of the batched HCGEMM kernel (interleaved layout) against cuBLAS (planar layout). Results are for sizes up to 256, with `batchCount` = 1000, on a Tesla V100-PCIe GPU.

for small sizes that cannot fully occupy the TCs when using the vendor-provided APIs. Without loss of generality, we are looking into small square matrices whose dimensions are $\leq$ 16. This range of sizes has been subject to many research efforts recently, due to its popularity in many applications [1,17].

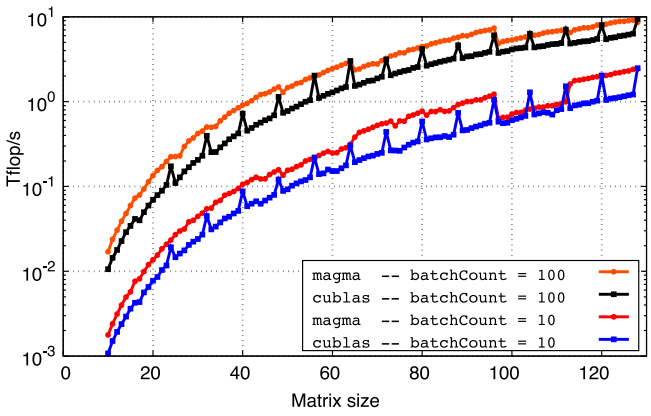


Fig. 14. The impact of batch size on the batched HGEMM performance. Results are for sizes up to 128, with `batchCount`$\in$ {10, 100}, on a Tesla V100-PCIe GPU.

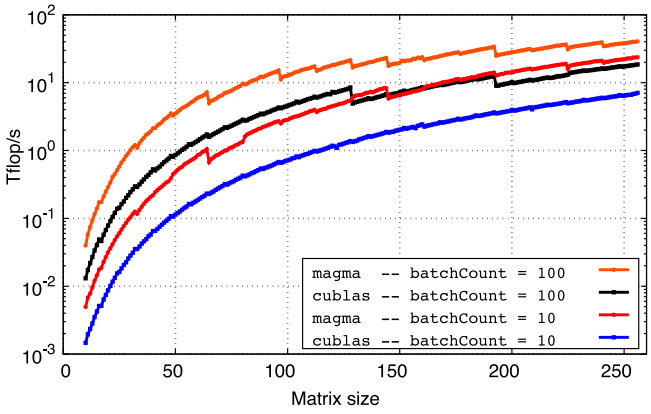


Fig. 15. The impact of batch size on the batched HCGEMM performance. Results are for sizes up to 256, with `batchCount`$\in$ {10, 100}, on a Tesla V100-PCIe GPU.

8.1. Multiple GEMMs in one tensor core multiplication

Recall that the permissible sizes for the TC multiplications are (16, 16, 16), (32, 8, 16), and (8, 32, 16). A single TC multiplication can therefore perform 8192 FLOPs using any of these combinations. Considering a square multiplication of size 4, the operation count is $2 \times 4^3 = 128$ FLOPs, which is less than 1.5% of a full TC multiplication. A previous proposition by the authors [4] showed how to improve the utilization by performing multiple GEMM operations in a single call to the TC APIs. An example is shown in Fig. 16, which improves the utilization for a (4, 4, 4) GEMM from 1.5% to about 6%. While this is a 4$\times$ improvement, the TC utilization is still very low. Fig. 17 shows the maximum achievable utilization for the TCs when performing square multiplications using the (16, 16, 16) configuration. The utilization in this figure is computed as $\lfloor \frac{16}{N} \rfloor \times \frac{2 \times N^3}{8192}$. The figure also shows the peak half-precision performance (as a percentage) without using the TC multiplications. Theoretically, a sufficiently optimized kernel that does not use the TC APIs can outperform the TC kernels for small sizes, $\leq$ 10.

The low utilization percentages shown in Fig. 17 raise questions about using TCs for tiny matrices, and whether conventional methods (i.e., without Tensor Cores) can perform the multiplications more efficiently. In this regard, we refer to a kernel developed for very small matrices specifically [22]. The kernel addresses very small square multiplications 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

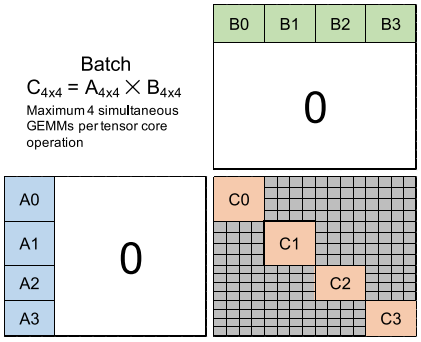


Fig. 16. Improving Tensor Core utilization by assigning multiple GEMMs at a time. Example for square matrices of size 4, with all Tensor Core sizes set to 16.

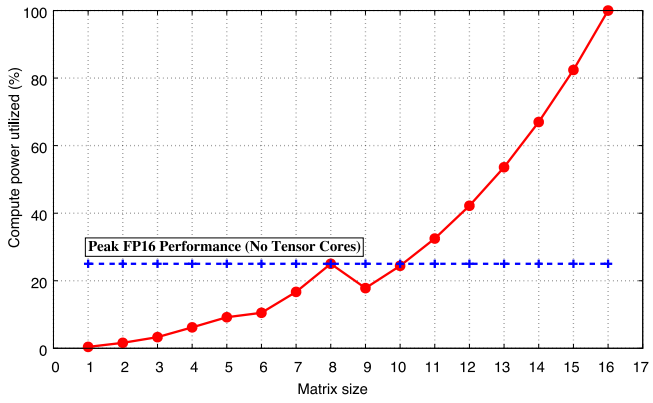


Fig. 17. The percentage of available Tensor Core compute power for square multiplications of small matrices.

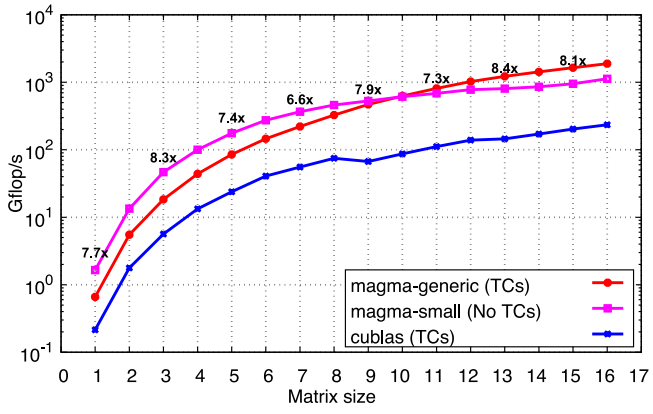


Fig. 18. Performance of the batched HGEMM kernels. Results are for small square sizes up to 16, with `batchCount` = 10k, on a Tesla V100-PCIe GPU.

in the output matrix. The code is fully unrolled for every size using C++ templates. In this paper, we use a similar kernel design that supports both half and half-complex arithmetic. We call this kernel *magma-small*.

We tested the performance of the *magma-small* kernel against the general MAGMA kernel as well as against cuBLAS, the results of which are summarized in Figs. 18 and 19 for the batched HGEMM and HCGEMM kernels, respectively. The generic MAGMA kernel is the best performing kernel for sizes $\approx$ 10 and up. For sizes smaller than 10, the *magma-small* kernel is the best solution, though no Tensor Cores are used. This is an interesting observation that nicely aligns with Fig. 17. Suboptimal TC utilization

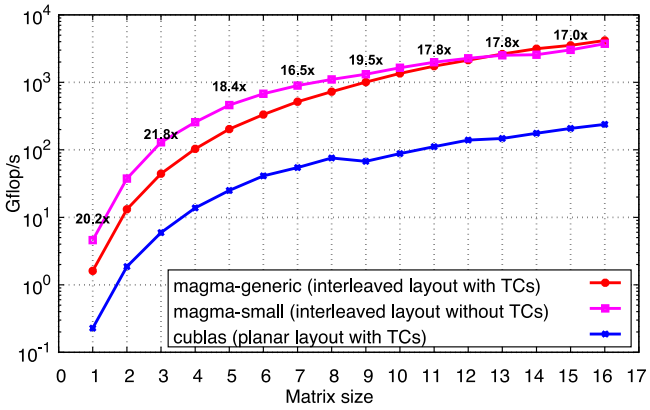


Fig. 19. Performance of the batched HCGEMM kernels. Results are for small square sizes up to 16, with `batchCount` = 10k, on a Tesla V100-PCIe GPU.

does not give access to the full 125 teraFLOP/s of compute power. Depending on the utilization (which is impacted by the problem size), the available compute power through the TCs may be lower than the available compute power without them. The threshold of 25% shown in Fig. 17 is the ratio between the full FP16 compute power in both situations. A utilization below this threshold may give the advantage to a kernel that does not use the TCs. This is realized in Figs. 18 and 19, where the magma-small kernel outperforms any kernel that uses the TCs as long as the problem size is below $\approx 10 \times 10$. For sizes larger than that, the use of TCs begins to pay off. For the batched HGEMM kernel, the observed speedups against cuBLAS are between 6.6× and 8.4×. For the batched HCGEMM, the use of the interleaved layout adds an extra advantage for the MAGMA kernels, which score speedups up to 21.8×.

9. Conclusion and future work

This paper introduced optimized batched matrix multiplication kernels using FP16 arithmetic on GPUs. The developed kernels address both half and half-complex precisions, and take advantage of the Tensor Core accelerator units in NVIDIA GPUs. The kernels share a common abstraction layer that encapsulates several constraints when calling the vendor-supplied APIs for programming the TC units. For half-precision matrices (batched HGEMM), the developed kernel outperforms cuBLAS for sizes up to 128, with speedups ranging between 1.5× and 2.5×. For sizes larger than 128, the developed kernel is still very competitive with cuBLAS—except for sizes multiple of 8, where cuBLAS has a clear advantage. For half-complex matrices, the paper shows that the standard interleaved layout provides a better solution than using planar layouts. This is mainly due to the increased operational intensity. The developed batched GEMM for complex matrices (batched HCGEMM) is between 1.7× and 7× faster than the cuBLAS solution using planar layouts. The paper also discusses special optimizations for extremely small problems, where the use of the TC APIs is questionable. The overall solution for tiny matrices can be up to 8.4×/21.8× faster than cuBLAS for the batched HGEMM and HCGEMM kernels, respectively.

The development of optimized matrix multiplication kernels is usually the most important step in developing higher-level dense linear algebra algorithms. The developed kernels can be used in reduced precision factorizations as well as mixed-precision solvers for linear systems of equations. The role of auto-tuning is critical in maintaining performance portability across different architectures. Some applications also require batches to have

problems of different sizes. All of these directions are promising for future work based on the developed kernels.

CRediT authorship contribution statement

Ahmad Abdelfattah: Conceptualization, Methodology, Software, Validation, Investigation, Writing - original draft, Writing - review & editing, Visualization. **Stanimire Tomov:** Investigation, Resources, Writing - original draft, Writing - review & editing, Supervision, Project administration, Funding acquisition. **Jack Dongarra:** Resources, Supervision, Project administration, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

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] A. Abdelfattah, M. Baboulin, V. Dobrev, J.J. Dongarra, C.W. Earl, J. Falcou, A. Haidar, I. Karlin, T.V. Kolev, I. Masliah, 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, http://dx.doi.org/10.1016/j.procs.2016.05.302.

[2] A. Abdelfattah, A. Haidar, S. Tomov, 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, http://dx.doi.org/10.1007/978-3-319-41321-1_2.

[3] A. Abdelfattah, A. Haidar, S. Tomov, J.J. Dongarra, Novel HPC techniques to batch execution of many variable size BLAS computations on GPUs, in: Proceedings of the International Conference on Supercomputing, ICS 2017, Chicago, IL, USA, June 14–16, 2017, 2017, pp. 5:1–5:10, http://dx.doi.org/10.1145/3079079.3079103.

[4] A. Abdelfattah, S. Tomov, J.J. Dongarra, Fast batched matrix multiplication for small sizes using half-precision arithmetic on GPUs, in: 2019 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2019, Rio de Janeiro, Brazil, May 20–24, 2019, 2019, pp. 111–122, http://dx.doi.org/10.1109/IPDPS.2019.00022.

[5] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, S. Tomov, Faster, cheaper, better – a hybridization methodology to develop linear algebra software for GPUs, in: W. mei W. Hwu (Ed.), GPU Computing Gems, Vol. 2, Morgan Kaufmann, 2010, URL https://hal.inria.fr/inria-00547847.

[6] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, S. Tomov, Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects, J. Phys. Conf. Ser. 180 (2009) 012037, http://dx.doi.org/10.1088/1742-6596/180/1/012037.

[7] K. Chellapilla, S. Puri, P. Simard, High performance convolutional neural networks for document processing, in: G. Lorette (Ed.), Tenth International Workshop on Frontiers in Handwriting Recognition, Suvisoft, Université de Rennes 1, La Baule (France), 2006, http://www.suvisoft.com, URL https://hal.inria.fr/inria-00112631.

[8] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, E. Shelhamer, CuDNN: Efficient primitives for deep learning, 2014, CoRR abs/1410.0759. arXiv:1410.0759.

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

[10] S. Gupta, A. Agrawal, K. Gopalakrishnan, P. Narayanan, Deep learning with limited numerical precision, in: Proceedings of the 32nd International Conference on International Conference on Machine Learning - Vol. 37, ICML'15, JMLR.org, 2015, pp. 1737–1746, URL http://dl.acm.org/citation.cfm?id=3045118.3045303.

[11] A. Haidar, A. Abdelfattah, M. Zounon, P. Wu, S. Pranesh, S. Tomov, 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.

[12] A. Haidar, T. Dong, P. Luszczek, S. Tomov, J.J. Dongarra, Batched matrix computations on hardware accelerators based on GPUs, Int. J. High Perform. Comput. Appl. 29 (2) (2015) 193–208, http://dx.doi.org/10.1177/1094342014567546.

[13] A. Haidar, S. Tomov, J. Dongarra, N. Higham, Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, Dallas, TX, USA, November 11-16, 2018, IEEE / ACM, 2018, pp. 47:1–47:11, http://dl.acm.org/citation.cfm?id=3291719.

[14] IEEE Standard for Floating-Point Arithmetic, IEEE Std 754-2008, 2008, pp. 1–70, http://dx.doi.org/10.1109/IEEESTD.2008.4610935. URL https://ieeexplore.ieee.org/document/4610935.

[15] C. Jhurani, P. Mullowney, A GEMM interface and implementation on NVIDIA GPUs for multiple small matrices, 2013, CoRR abs/1304.7053. URL http://arxiv.org/abs/1304.7053.

[16] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, T. Darrell, Caffe: Convolutional architecture for fast feature embedding, in: Proceedings of the 22nd ACM International Conference on Multimedia, MM '14, ACM, New York, NY, USA, 2014, pp. 675–678, http://dx.doi.org/10.1145/2647868.2654889, URL http://doi.acm.org/10.1145/2647868.2654889.

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

[18] J. Kurzak, S. Tomov, J. Dongarra, Autotuning GEMM kernels for the fermi GPU, IEEE Trans. Parallel Distrib. Syst. 23 (11) (2012) 2045–2057, http://dx.doi.org/10.1109/TPDS.2011.311.

[19] J. Lai, 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), CGO '13, IEEE Computer Society, Washington, DC, USA, 2013, pp. 1–10, http://dx.doi.org/10.1109/CGO.2013.6494986.

[20] A. Lavin, S. Gray, Fast algorithms for convolutional neural networks, in: The IEEE Conference on Computer Vision and Pattern Recognition, CVPR, 2016.

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

[22] I. Masliah, A. Abdelfattah, A. Haidar, S. Tomov, M. Baboulin, J. Falcou, 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, http://dx.doi.org/10.1007/978-3-319-43659-3_48.

[23] R. Nath, S. Tomov, J. Dongarra, An improved magma gemm for fermi graphics processing units, Int. J. High Perform. Comput. Appl. 24 (4) (2010) 511–515, http://dx.doi.org/10.1177/1094342010385729.

[24] L. Ng, K. Wong, A. Haidar, S. Tomov, J. Dongarra, MagmaDNN – High-performance data analytics for manycore GPUs and CPUs, 2017, Magma-DNN, 2017 Summer Research Experiences for Undergraduate (REU), Knoxville, TN. http://icl.cs.utk.edu/projectsfiles/magma/pubs/71-MagmaDNN.pdf. http://icl.cs.utk.edu/magma/software/.

[25] K. Rupp, P. Tillet, F. Rudolf, J. Weinbub, A. Morhammer, T. Grasser, A. Jüngel, S. Selberherr, ViennaCL—Linear algebra library for multi- and many-core architectures, SIAM J. Sci. Comput. 38 (5) (2016) S412–S439, http://dx.doi.org/10.1137/15M1026419.

[26] G. Tan, L. Li, S. Triechle, E.H. Phillips, Y. Bao, 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, http://dx.doi.org/10.1145/2063384.2063431, URL http://doi.acm.org/10.1145/2063384.2063431.

[27] S. Tomov, J.J. Dongarra, M. Baboulin, Towards dense linear algebra for hybrid GPU accelerated manycore systems, Parallel Comput. 36 (5–6) (2010) 232–240, http://dx.doi.org/10.1016/j.parco.2009.12.005.

[28] N. Tomov, S. Tomov, On deep neural networks for detecting heart disease, 2018, CoRR abs/1808.07168. arXiv:1808.07168.

[29] V. Volkov, 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, http://dx.doi.org/10.1145/1413370.1413402, URL http://doi.acm.org/10.1145/1413370.1413402.

[30] S. Williams, A. Waterman, D. Patterson, Roofline: An insightful visual performance model for multicore architectures, Commun. ACM 52 (4) (2009) 65–76, http://dx.doi.org/10.1145/1498765.1498785, URL http://doi.acm.org/10.1145/1498765.1498785.

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

[32] S.N. Yeralan, T.A. Davis, W.M. Sid-Lakhdar, S. Ranka, Algorithm 980: Sparse QR factorization on the GPU, ACM Trans. Math. Software 44 (2) (2017) 17:1–17:29, http://dx.doi.org/10.1145/3065870, URL http://doi.acm.org/10.1145/3065870.

Ahmad Abdelfattah received his Ph.D. in computer science from King Abdullah University of Science and Technology (KAUST) in 2015, where he was a member of the Extreme Computing Research Center (ECRC). He is currently a research scientist in the Innovative Computing Laboratory at the University of Tennessee. He works on optimization techniques for different linear algebra workloads on GPUs. Ahmad has B.Sc. and M.Sc. degrees in computer engineering from Ain Shams University, Egypt.

Stanimire Tomov received a M.S. degree in Computer Science from Sofia University, Bulgaria, and Ph.D. in Mathematics from Texas A&M University. He is a Research Assistant Professor at the Innovative Computing Laboratory (ICL), University of Tennessee. Tomov's research interests are in parallel algorithms, numerical analysis, and high-performance scientific computing (HPC). Currently, his work is concentrated on the development of numerical linear algebra software for emerging architectures for HPC.

Jack Dongarra received a Bachelor of Science in Mathematics from Chicago State University in 1972 and a Master of Science in Computer Science from the Illinois Institute of Technology in 1973. He received his Ph.D. in Applied Mathematics from the University of New Mexico in 1980. He worked at the Argonne National Laboratory until 1989, becoming a Senior Scientist. He now holds an appointment as University Distinguished Professor of Computer Science in the Department of Electrical Engineering and Computer Science at the University of Tennessee, has the position of a Distinguished Research Staff member in the Computer Science and Mathematics Division at Oak Ridge National Laboratory (ORNL), Turing Fellow in the Computer Science and Mathematics Schools at the University of Manchester, and an Adjunct Professor in the Computer Science Department at Rice University.