Demystifying Tensor Cores to Optimize Half-Precision Matrix Multiply

Da Yan*, Wei Wang*, Xiaowen Chu†
*Hong Kong University of Science and Technology
†Hong Kong Baptist University
*{dyanab,weiwa}@cse.ust.hk, †chxw@comp.hkbu.edu.hk

***Abstract*—Half-precision matrix multiply has played a key role in the training of deep learning models. The newly designed Nvidia Tensor Cores offer the native instructions for half-precision small matrix multiply, based on which Half-precision General Matrix Multiply (HGEMM) routines are developed and can be accessed through high-level APIs. In this paper, we, for the first time, demystify how Tensor Cores on NVIDIA Turing architecture work in great details, including the instructions used, the registers and data layout required, as well as the throughput and latency of Tensor Core operations. We further benchmark the memory system of Turing GPUs and conduct quantitative analysis of the performance. Our analysis shows that the bandwidth of DRAM, L2 cache and shared memory is the new bottleneck for HGEMM, whose performance is previously believed to be bound by computation. Based on our newly discovered features of Tensor Cores, we apply a series of optimization techniques on the Tensor Core-based HGEMM, including blocking size optimization, data layout redesign, data prefetching, and instruction scheduling. Extensive evaluation results show that our optimized HGEMM routine achieves an average of $1.73\times$ and $1.46\times$ speedup over the native implementation of cuBLAS 10.1 on NVIDIA Turing RTX2070 and T4 GPUs, respectively. The code of our implementation is written in native hardware assembly (SASS).**

***Index Terms*—GEMM, GPU, Tensor Core, Half-precision**

I. Introduction

GEMM (General Matrix Multiply) serves as a core building block for deep learning computations. For example, Tensor operations in the fully-connected layers can directly use GEMM. For convolutional layers in Convolutional Neural Networks (CNN), the convolution operations can be reduced to GEMM efficiently [1]. In the Long Short Term Memory (LSTM) model composed of multiple cells, it requires performing multiple GEMMs in each cell. In the state-of-the-art NLP model BERT [2], the basic transformer unit [3] also uses GEMM as the underlying operations.

Given the importance of GEMM, NVIDIA introduced dedicated hardware Tensor Core in 2017 with Volta V100 GPUs [4] to accelerate its execution. Tensor Cores are also available in the subsequent Turing generation. Each Tensor Core consumes two 4×4 half-precision (FP16) matrices and computes their multiplication result in one clock cycle. On devices like V100, T4, and RTX2070, Tensor Cores offer $4\times$ higher FLOPS than the FP16 units. The result provided by Tensor Core is also of higher accuracy [5] than that of the FP16 unit. Despite the significant performance advantage, how Tensor Cores work in detail and what throughput and latency of the Tensor Core operations can be achieved remain the important missing pieces in the literature.

This paper is the first to reveal the working details of Tensor Cores on NVIDIA Turing GPUs. Through extensive measurement studies, we illustrate the data layout required by the `HMMA.1688` instruction, which is used to manipulate Tensor Cores. We show that 8×8 matrix is the basic unit of half-precision Tensor Core programming. This observation can simplify Tensor Core programming. We also present a benchmark result of the throughput and latency performance of Tensor Core instructions.

We measure Turing GPUs' DRAM and L2 cache performance. Based on the bandwidth of DRAM and L2 cache we measured, we analyze the performance of Tensor Core-based HGEMM with the roofline model [6]. Our analysis shows that the performance of HGEMM—previously believed to be computation-bound—is actually bound by the global memory bandwidth. To mitigate the bottleneck of limited global memory bandwidth, we analyze and increase the blocking size [7] to maximize data reuse and reduce the data needed to be fetched from the global memory.

We then benchmark the throughput of Turing GPUs' shared memory in clock cycle per instruction (CPI). To our knowledge, this is the first work that benchmarks the CPI of shared memory access on GPUs. The CPI value helps us interleave the shared memory instructions with a proper number of Tensor Core instructions. Moreover, by analyzing the performance with the CPI value, we show that the shared memory bandwidth is also a performance bottleneck. To eliminate the bottleneck posed by the limited shared memory bandwidth, we redesign the data layout in shared memory to avoid bank conflict and increase the shared memory blocking size to increase computation intensity at the shared memory level.

Based on the identified hardware features, we implement our highly optimized Tensor Core-based HGEMM. We evaluate our HGEMM implementation on NVIDIA Turing RTX2070 and T4 GPUs and compare its performance with the native routine in cuBLAS 10.1. Evaluation results show that our optimized HGEMM routine achieves an average of $1.73\times$ and $1.46\times$ speedup over cuBLAS's implementation on RTX2070 and T4, respectively. For large matrices, the speedup is up to

This work was supported in part by Hong Kong RGC ECS grant 26213818 and GRF grant HKBU 12200418.

$3\times$, reaching the device peak.

We summarize our main contributions as follows:

- We demystify how Tensor Cores on Turing GPUs work.
- We benchmark the CPI of global memory instruction (`LDG`) and shared memory instructions (`LDS`, `STS`) on Turing GPUs.
- We analyze the performance of HGEMM with Tensor Cores and point out that the performance of HGEMM for large matrices is mainly bottlenecked by different memory components, including DRAM, L2 cache and shared memory.
- We implement Tensor Core-based HGEMM on NVIDIA Turing GPUs and evaluate its performance on RTX2070 and T4. Our HGEMM achieves up to $3\times$ speedup over cuBLAS 10.1's Tensor Core-based HGEMM for large matrices.

The remainder of this paper is organized as follows. We describe the notations and the basic of Tensor Core programming in Section II. We survey related work in Section III and present the details of Tensor Core on Turing GPUs in Section IV. In Section V, we benchmark DRAM, L2 cache and shared memory on GPUs. Section VI discusses the design and optimizations of HGEMM in detail, including the choosing of blocking size and instruction scheduling. Section VII evaluates our optimized HGEMM. We summarize our work in Section VIII.

II. Background

This section introduces the concept of GEMM and the notations we used in this work, as well as how we program the Tensor Cores.

A. GEMM and Notations

The standard form of GEneral Matrix Multiplication (GEMM) is $C = \alpha AB + \beta C$, where A, B and C are $m \times k$, $n \times k$ and $m \times n$ matrices, respectively, and α and β are scalar constants. In this work, we focus on the case where $\alpha = 1.0$ and $\beta = 0.0$, i.e., the $C = AB$ matrix multiplication.

Following the convention, we use $m \times n \times k$ to denote $C = AB$ where A, B and C are $m \times k$, $n \times k$ and $m \times n$ matrices, respectively.

B. Tensor Core Programming

As of this writing, programmers can access Tensor Cores in two ways. The first way is to program Tensor Core at CUDA C++ level with Warp Matrix Multiply and Accumulation (WMMA) API [8]. The second way is to call functions from NVIDIA's libraries like cuBLAS for linear algebra routines, or cuDNN for deep learning routines.

When programming with CUDA C++ level WMMA API, programmers need to load data with `load_matrix_sync`, do the computation with `mma_sync`, and finally store the result with `store_matrix_sync`. While programming in CUDA C++ involves less human efforts compared with programming in assembly, programmers lose the control of instruction scheduling. And as we will show in Section VI-A, the performance of Tensor Core-based HGEMM is sensitive to the memory bandwidth and it is hard for the compiler to generate an optimal instruction schedule. As reported in [5], a naive WMMA-based HGEMM is only able to achieve around 10% of device peak. Even highly optimized CUDA-level implementation like CUTLASS [9] can only achieve around 50% of device peak [5].

Another way to leverage Tensor Core is through libraries like cuBLAS. HGEMM routine in the cuBLAS library is believed to be written in native assembly, Streaming ASSembler (SASS). However, the detail of Tensor Cores at the SASS level is unpublished, making it challenging for users to program Tensor Cores at the SASS level. We reveal how Tensor Cores work in detail in the later section.

III. Related Work

In this section, we discusses related work in reverse engineer Tensor Cores, GPU memory benchmarking and GEMM optimization.

A. Dissecting Tensor Core

Jia et al. [10], [11] started the work of dissecting Tensor Cores. They showed that the Tensor Cores are controlled by instructions `HMMA.884` and `HMMA.1688`. They also showed the data layout required by the Tensor Core when computing $16 \times 16 \times 16$ matrix multiplication with `HMMA.884`.

Compared with their work, we make the following distinct contributions:

- We point out that the basic unit in Tensor Core programming with the `HMMA.1688` instruction is an 8×8 matrix.
- We show that the 8×8 matrix can be indexed by a 32-bit register, and we give the data layout (in registers) of the 8×8 matrix.
- We explicitly point out that the `.1688` infix stands for $16 \times 8 \times 8$ matrix multiplication.
- We benchmark the latency and throughput of the `HMMA.1688` instruction.
- We optimize the Tensor Core-based HGEMM based on our new observations.

B. GPU Memory Benchmark

Mei and Chu [12] introduced the fine-grained P-chase method to benchmark GPU memory hierarchy. Their method can detect cache size and latency, and was implemented in CUDA C++. Compared to their method, we implement our benchmark in SASS.

SASS-level benchmark enables us to discover patterns that are difficult to observe at the CUDA C++ level. For example, in CUDA C++, it is impossible to issue a long sequence (e.g., 128 continuous) of `LDG` instructions, because the compiler will think those codes have no effects and will optimize them away. Issuing a long sequence of load/store instructions is important to get the correct CPI value, which is important in our analysis as we will show later.

C. GEMM Tuning

There is a rich body of work on optimizing double and single precision GEMMs on GPUs. Lai et al. [13] optimized SGEMM on Fermi and Kepler GPUs. Zhang et al. [14] optimized SGEMM on Kepler K20. Gray [15] optimized SGEMM on Maxwell GPUs. All these works can achieve near-to-peak (more than 85% of limit) performance and are written in SASS. While HGEMM shares many similarities with SGEMM and DGEMM, it requires special data layout and higher computation intensity to keep the Tensor Core pipe busy. To solve the newly emerged problems, we propose an analytical model to choose blocking size (Section VI-A) and to schedule instructions (Section VI-C). In [16], Li et al. proposed a framework to strike a balance between ILP and TLP, targeting at batched small matrix multiplications of different sizes.

CUTLASS (CUDA Templates for Linear Algebra Subroutines) [9] is a template library that targets linear algebra, especially GEMM. The CUTLASS library supports different data types including double, single, half and int. It adopts many different optimizations like multi-level blocking to maximize data reuse and data prefetching to hide latency. The CUTLASS library is written in CUDA C++, making it portable across different GPU architectures. However, they lose the control of instruction scheduling at the CUDA C++ level. For Tensor Core-based HGEMM, CUTLASS usually cannot achieve the near-to-peak performance [5].

IV. Demystifying Tensor Cores

While Tensor Core provides higher throughput and more accurate results than FP16 unit, the underlying mechanism remains unclear, which makes it hard for users to understand and optimize the performance. In this section, we give an in-depth analysis of the Tensor Core. Details include the instruction to manipulate Tensor Core, the data layout required by the Tensor Core, the throughput and the latency of Tensor Core instructions.

A. Tensor Core Instruction

Turing GPUs use the `HMMA` instruction to manipulate the Tensor Core for float computation. As previous reverse engineering work [11] points out, there are two kinds of `HMMA` instructions with `.884` and `.1688` infix. In this work, we focus on the `.1688` version, because it is more succinct. Also, for the first time, we point out that the `.1688` infix stands for $16 \times 8 \times 8$ matrix multiplication. The `HMMA.1688` instruction has a type specifier as the suffix. It specifies whether the accumulator is single precision (`.F32`) or half precision (`.F16`). We limit our discussion to the `.F16` variant in this paper.

A typical `HMMA` instruction is given as follows:

```
HMMA.1688.F16 R0, R2, R6, R4;        (1)
```

which computes

$$D_{16\times 8} = A_{16\times 8} B_{8\times 8} + C_{16\times 8}, \qquad (2)$$

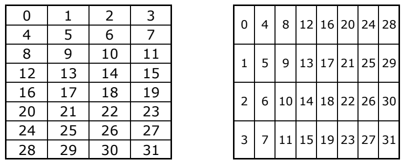


Fig. 1: Row-major order (left) and column-major order (right) to store one 8×8 half precision matrix. Number in the cell is the lane id. One 32-bit thread register stores two half elements. 32 threads within a warp can store 64 elements of the matrix.

where $D_{16\times 8}$ is stored in 64-bit registers, `R0` and `R1`, $A_{16\times 8}$ is stored in `R2` and `R3`, $B_{8\times 8}$ is stored in `R6`, and $C_{16\times 8}$ is stored in `R4` and `R5`. When the type specifier is `.F32`, matrix D and C are stored in 128-bit registers.

The basic building block in the half-precision Tensor Core regime is an 8×8 matrix. One 32-bit register within a warp (32 threads) can store $32 \times 4 = 128$ bytes. An 8×8 half-precision matrix also occupies 128 bytes space. Thus an 8×8 half-precision matrix can be represented by 32-bit registers within a warp with the same index. In this sense, the registers in the `HMMA.1688` instruction are not conventional thread registers (or scalar registers). Instead, they should be viewed as "warp registers" since this instruction requires data within a warp to cooperate to generate the correct result.

Note that the Tensor Core programming model breaks the original CUDA programming model. In the CUDA programming model, the data of each thread is private to itself. The only way to access data in other threads is to explicitly communicate with global/shared memory or shuffle instructions. In contrast, the Tensor Core instruction `HMMA` requires threads within a warp to cooperate, i.e., it allows data in different threads to communicate with each other implicitly.

B. Matrix in Register

After knowing that one "warp register" can store one 8×8 half-precision matrix, the remaining important question is: how is the matrix scattered over registers in different lanes?

By looking into the SASS code generated by compiling `load_matrix_sync` and `mma_sync`, we have the following observations:

- An 8×8 half precision matrix can be stored in either row-major or column-major order. We depict both row-major and column-major layout in Fig. 1.
- `HMMA.1688.F16` requires the destination matrix in row-major order, the first source matrix in row-major order, and the second matrix in column-major order. For example, in Eq. (2), $D_{16\times 8}$, $C_{16\times 8}$ and $A_{16\times 8}$ are stored in row-major order whereas $B_{8\times 8}$ is stored in column-major order.

We summarize our observations and show how matrix are distributed in registers for `HMMA.1688.F16 R0, R6, R4` in Fig. 2.

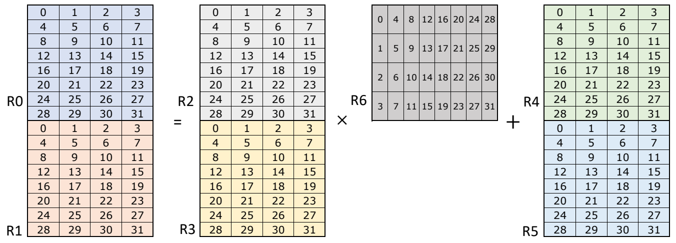


Fig. 2: Distribution of matrix elements in resigers for `HMMA.1688.F16 R0, R2, R6, R4`. Each colored 8×8 square is an 8×8 matrix. We label the register index next to the coresponding matrix. The second source matrix (`R6`) needed to be stored in column-major order.

TABLE I: Throughput and latency of `HMMA.1688.F16`.

Metric	Value
CPI theoretical	8.00
CPI measured	8.06
Latency for the first half of $D_{16\times8}$	10
Latency for the second half of $D_{16\times8}$	14

C. Performance Metrics

There are two critical performance metrics of `HMMA.1688.F16`, namely throughput and latency. We perform a SASS-level benchmark to obtain those two metrics, where we use *clock cycles per instruction* (CPI) to measure the throughput.

Specifically, we measure the CPI by issuing thousands of `HMMA` instructions and record the clock cycles it takes. To eliminate the effect of cold instruction cache miss, we reconstruct the long `HMMA` sequence into a loop, which can be fit into the L0 instruction cache. Since the $16 \times 8 \times 8$ matrix multiplication is composed of 16 $4 \times 4 \times 4$ matrix multiplications and each processing block has 2 Tensor Cores [17], the expected CPI is $16/2 = 8$. Our measured CPI is 8.06, close to the theoretical analysis.

We measure the latency of `HMMA.1688.F16` by varying the stall cycles and check if the output result is correct. It takes 10 cycles for the first half of $D_{16\times8}$ (`R0` in Eq. (1)) and 14 cycles for the second half of $D_{16\times8}$ (`R1` in Eq. (1)) to get correct result. No register bank conflict has been observed for `HMMA.1688`. The register reuse flag has no impact on performance. Those metrics are the same on RTX2070 and T4 since they are of the same architecture and have the same Streaming Multiprocessor (SM).

We summarize the important performance metrics in Table I and highlight our key observations on Tensor Cores as follows:

- The basic element in half-precision Tensor Core programming is an 8×8 matrix.
- An 8×8 half-precision matrix is scattered over different threads within a warp and can be indexed by one register. The layout of the matrix is shown in Fig. 1.
- One `HMMA.1688` instruction computes $16 \times 8 \times 8$ matrix multiplication and accumulation.

TABLE II: Measured DRAM and L2 cache bandwidth of RTX2070 and T4 GPUs

	RTX2070	T4
DRAM theoretical	448GB/s	320GB/s
DRAM measured	380GB/s	238GB/s
L2 measured	750GB/s	910GB/s
Tensor Core throughput	59.7 TFLOPS	65 TFLOPS

- We observe no register bank conflict for `HMMA.1688`. Register reuse flag has no impact on the performance of the `HMMA` sequence.

V. Memory Microbenchmark

Since the Tensor Core provides $4\times$ higher throughput, the bottleneck may shift to memory. However, important performance metrics, such as L2 cache throughput and shared memory throughput, remain unpublished. To support subsequent performance analysis, we perform detailed benchmarks on Turing GPUs' memory system. We benchmark the throughput of DRAM and L2 cache in GB/s to analyze the performance with the roofline model to choose the right blocking size. We then benchmark the CPI of global and shared memory access to help us interleave memory instructions with a proper number of `HMMA` instructions.

A. DRAM and L2 Cache

a) Methodology: We measure DRAM's throughput by launching a kernel with multiple thread blocks and letting each thread load a 512KB of data. To ensure data are loaded from DRAM rather than L2 cache, we let each thread block load data at different locations. The kernel running time is recorded using *cuda event* [18].

We measure L2 cache's throughput by letting each thread block load a 512KB of data at the same location to ensure data are loaded from L2 cache rather than DRAM. For both DRAM and L2 cache measurement, we enforce the `LDG` instruction to bypass the L1 cache, which is accomplished by using the `.ca` flag in the PTX `ld` instruction [19].

We measure the CPI of `LDG` by launching a kernel with 32 threads and letting the kernel issue thousands of `LDG` instructions continuously. We record the cycles needed by reading the value in the clock register (equivalent to calling `clock()` in CUDA C++) at the beginning and the end of the long `LDG` sequence. To eliminate the effect of instruction cache miss, we reconstruct the long sequence into a loop, which is small enough (128 instructions) to fit into the L0 instruction cache.

Note that this kind of CPI benchmark is only possible at SASS-level. If the code is written in CUDA C++ or PTX level, the compiler will optimize the long sequence of load instructions away, as they have no effect. While we can use a sink to prevent the compiler from optimizing away the `LDG`s, the instruction sequence will be messed up, making the result inaccurate.

TABLE III: CPI of LDG on Turing GPUs.

Type	Width		
	32	64	128
LDG (data in L1 cache)	4.04	4.04	8.00
LDG (data in L2 cache)	4.19	8.38	15.95

b) Benchmark result: We list the measured throughput in GB/s in Table II, and show the measured throughput in CPI of LDG with different width in Table III. We summarize important observations about DRAM and L2 cache as follows:

- While T4 GPU has higher FLOPS than RTX2070, its DRAM bandwidth is smaller. As we will show in Section VI-A and Section VII, the performance of Tensor Core-based HGEMM on T4 is bound by DRAM bandwidth.
- The measured DRAM bandwidth is 85% of the peak on RTX2070 and 75% of the peak on T4.
- From the SM's perspective, when data are stored in L2 cache, LDG.32 has the same throughput as LDG.64 ($32/CPI_{\texttt{LDG.32}} = 64/CPI_{\texttt{LDG.64}}$). LDG.128 has 5.1% higher throughput than the other two.

B. Shared Memory

a) Methodology: We measure the CPI of shared memory access in the same way as we measure the CPI of LDG. That is, we issue a loop of shared memory access instructions and record the cycles needed. We pick the shared memory access offset to make sure that all accesses are bank conflict-free.

b) Benchmark result: We summarize the CPI of shared memory instructions in Table IV and the corresponding throughput (bytes/cycle) in Table V. The CPI and throughput are the same on RTX2070 and T4.

We summarize the important shared memory behaviors as follows:

- LDS.128 has the same throughput as LDS.64, and has 5.5% higher throughput than LDS.32.
- LDS.64 and LDS.128 can achieve theoretical peak bandwidth. In comparison, [11] reports LDS bandwidth of 58.8 bytes/cycle. This demonstrates the power of benchmarking at the SASS-level.
- STS.128 has 20% higher throughput than STS.64 and 62.4% higher than STS.32. Meaning, using a narrow version of STS results in a high penalty on throughput.

TABLE IV: CPI of shared memory load/store instructions on Turing GPUs.

Type	Width		
	32	64	128
LDS	2.11	4.00	8.00
STS	4.06	6.00	10.00

TABLE V: Throughput (bytes/cycle) of shared memory load/store instructions on Turing GPUs.

Type	Width		
	32	64	128
LDS	60.66	64.00	64.00
STS	31.53	42.67	51.20

VI. Optimizing Matrix Multiplication Routine

The state-of-the-art GEMM implementations on GPUs [9], [13], [14] all use two-level blocking to maximize data reuse. In this section, we first analyze how the blocking size affects the performance of HGEMM and how to choose the proper blocking size. Other than blocking sizes, the instruction order and the data layout in shared memory can also impact the performance of HGEMM routines. To this end, we further discuss how we use the CPI value to schedule instructions and how we design the data layout in shared memory. We evaluate the effects of these optimizations and show that they provide us with significantly higher throughput. Our results are obtained on RTX2070, where we use the *cuda event* to record the running time. The reported throughput is averaged over 10 measurements.

A. Blocking Size Analysis

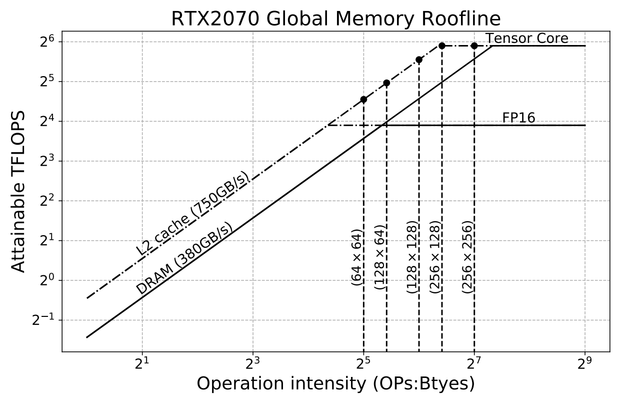
1) Blocked HGEMM: We show how two-level blocking works in Algorithm 1. We omit the details including data prefeching, barrier synchronization and index calculation for brevity. We refer to CUTLASS [9] for the idea of two-level blocking.

Algorithm 1: Blocked HGEMM. Fragments (A_frag, B_frag, C_frag) reside in registers

```
__shared__ A_smem[$b_m$][$b_k$];
__shared__ B_smem[$b_n$][$b_k$];
A_frag[$w_m$][$w_k$];
B_frag[$w_n$][$w_k$];
C_frag[$w_m$][$w_n$];
for iter←0 to $k$ by $b_k$ do
    A_smem ← $b_m \times b_k$ of A;
    B_smem ← $b_n \times b_k$ of B;
    for i←0 to $b_k$ by $w_k$ do
        foreach $16 \times 8$ matrix in C_frag do
            C_frag[][] ← C_frag[][] + A_frag[][$i$]×
              B_frag[][$i$];
        end
    end
end
```

2) Blocking Size of Thread Block: Thread block is at the first level of blocking, also known as shared memory blocking in the literature. We divide the $m \times n \times k$ matrix multiplication into multiple blocks. Each thread block computes $b_m \times b_n$ of matrix multiplication result. Thus $\lfloor (m + b_m - 1)/b_m \rfloor \times \lfloor (n + b_n - 1)/b_n \rfloor$ thread blocks will be launched.

Each thread block performs $2b_m b_n b_k$ operations and loads $(b_m + b_n)b_k$ half-precision elements. The computation intensity, therefore, is $\frac{2b_m b_k b_n}{2(b_m+b_n)b_k} = \frac{b_m b_n}{b_m+b_n}$. The blocking size



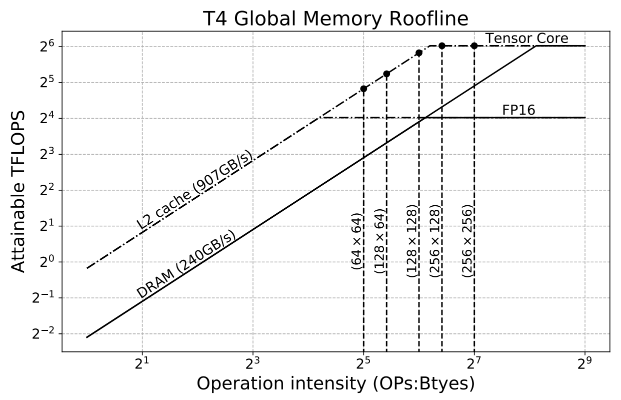


Fig. 3: Global memory roofline model on RTX2070 and T4. Compared with the solution using FP16 units, the high throughput of Tensor Core makes dense GEMM kind of memory-bounded.

will affect the computation intensity. Typical blocking sizes $(b_m \times b_n)$ for half precision are (256×256), (256×128), (128×128), (128×64), (64×64).

Larger blocking size increases the computation intensity but may lower the occupancy, hence harming the performance. To choose a proper thread block blocking size, we use the bandwidth we collected (see Table II) to depict the roofline models for RTX2070 and T4 GPUs in Fig. 3. We also plot the roof of FP16 unit in Fig. 3 for comparison. When using FP16 units, (128×128) is good enough to keep the arithmetic pipe busy. But for Tensor Cores, (128×128) makes the DRAM bandwidth a new bottleneck.

While increasing the blocking size of thread block can increase the computation intensity, we cannot adopt a larger blocking size like (512×256). The bounding factor here is the number of registers. On Turing GPUs, each SM has 64K 32-bit registers. 512×256 half-precision elements occupy all of them and leave no place for the other data. Thus, the (512×256) version does not work. Other large blocking sizes like (256×320) may work, but we prefer blocking sizes that are power of 2 since it is natural for GPU programming and easier to reason about.

In this work, we focus on the (256×256) version. The third dimension of the blocking b_k is limited by the shared memory size. Turing GPUs have up to 64KB shared memory per SM, thus b_k must be smaller than $64KB/(256 + 256)/2 = 64$. While choosing b_k as 64 works, the fetched tiles of A and B will take up all 64KB shared memory and leave no room for padding to avoid bank conflict, which will hurt performance greatly. Our final choice for b_k is 32.

3) Warp Level Blocking Size: The second level of blocking is the warp level. This level of blocking is also called register blocking in the literature. The problem is how to distribute the (256×256) of work to proper number of warps. Currently, most of the exiting works [9], [13] choose the warp level blocking size via trial-and-error, and some [14] compute the computation intensity and use the roofline model at warp level to choose the warp level blocking size. We propose a new simple yet effective method to determine the warp level blocking size before implementation. Our method is different from the previous methods in several ways:

- We use CPI as the metric to guide the selection of warp level blocking size, which has been largely ignored in the previous work.
- Compared with the trial-and-error method, our approach gives the proper blocking size before implementation, saving lots of engineering efforts.

HGEMM in cuBLAS 10.1 uses (64×64) blocking size at this level. We will show that (64×64) is not enough and makes the memory pipe a new bottleneck. (128×64) can solve this problem.

The cycles used to do computation per thread block in each iteration with `HMMA` can be computed as:

$$\frac{2 b_m b_n b_k}{2 \times 16 \times 8 \times 8 \times 4} \times CPI_{\texttt{HMMA.1688.F16}}, \tag{3}$$

where $2 \times 16 \times 8 \times 8$ is the computation done with `HMMA.1688`, and there are 4 processing blocks in an SM.

`LDG`, `STS` and `LDS` instructions all occupy memory I/O pipe [17]. The cycles used to load $(b_m + b_n) \times b_k$ of data from global memory and store to shared memory with 128-bit width instructions are:

$$\frac{(b_m + b_n) b_k \times 2}{32 \times 16} \times (CPI_{\texttt{LDG.128}} + CPI_{\texttt{STS.128}}), \tag{4}$$

where 2 stands for 2 bytes of a half-precision element, 32 is the warp size, and 16 stands for 16 bytes (128 bits).

We compute the cycles used to load data from shared memory with 32-bit width instructions as follows:

$$\frac{b_m b_n}{w_m w_n} \times \left(\frac{w_m}{8} + \frac{w_n}{8}\right) \times \frac{b_k}{w_k} \times CPI_{\texttt{LDS.32}}. \tag{5}$$

While changing the two-level blocking size will not change the total computation, it will change the data needed to be transferred. Thus, the goal is to keep Tensor Cores busy and not to let the memory I/O be the performance bottleneck. In other words, we need to keep the clock cycles needed for memory I/O in each iteration smaller than the clock cycles needed for Tensor Core processing. Based on the

TABLE VI: Clock cycles needed by Tensor Core pipe and memory IO pipe under different blocking sizes.

		cycles	
$(b_m \times b_n \times b_k)$	$(w_m \times w_n \times w_k)$	`HMMA`	Memory IO
$(128 \times 128 \times 32)$	$(64 \times 64 \times 8)$	1031	1370
$(128 \times 128 \times 32)$	$(128 \times 64 \times 8)$	1031	1235
$(256 \times 128 \times 32)$	$(64 \times 64 \times 8)$	2063	2325
$(256 \times 128 \times 32)$	$(128 \times 64 \times 8)$	2063	2055
$(256 \times 256 \times 32)$	$(64 \times 64 \times 8)$	4126	3821
$(256 \times 256 \times 32)$	$(128 \times 64 \times 8)$	4126	3281

TABLE VII: Details of our HGEMM and cuBLAS 10.1's HGEMM.

	Ours	cuBLAS 10.1
$(b_m \times b_n \times b_k)$	$(256 \times 256 \times 32)$	$(128 \times 128 \times 64)$
$(w_m \times w_n \times w_k)$	$(128 \times 64 \times 8)$	$(64 \times 64 \times 8)$
Shared memory/CTA	36KB	32KB
Active CTAs/SM	1	2
Active warps/SM	8	8

aforementioned analysis and CPI values we collected (see Tables I, III and IV), we list cycles needed by `HMMA` and memory instructions of blocking sizes of interests in Table VI.

Table VI shows that when the thread block's blocking size is (128×128), the HGEMM is bound by memory I/O. When the blocking size is (256×128), the HGEMM will be bound by memory I/O if the warp blocking size is (64×64) and Tensor Core bound if the warp blocking size is (128×64). The best case is to set the thread block's blocking size as (256×256) and the warp blocking size as (128×64). In this case, the clock cycles required by `HMMA` is significantly greater than the clock cycles required by memory I/O. This blocking size setting makes the HGEMM robust to L2 cache miss and leaves enough room for latency hiding.

Note that we cannot set the warp-level blocking size to (128×128), as each thread can only use up to 256 registers and (128×128) will occupy all of them within a warp.

4) Comparison with cuBLAS: Table VII compares our implementation of HGEMM with the stock implementation in cuBLAS 10.1. The benefit of the smaller blocking size (128×128) as compared to what cuBLAS 10.1 adopts is that it allows 2 CTAs to reside on one SM, which makes the warps run in an asynchronous manner. The asynchronous behavior gives the warp scheduler more opportunities to switch to the other warps for increased throughput. This is align with the conventional wisdom of GPGPU programming.

However, as our previous analysis in this section shows, when the blocking size is $(128 \times 128 \times 64)$, both DRAM and shared memory will throttle the program. It is interesting to notice that cuBLAS' HGEMM only uses 32KB of shared memory. This suggests that cuBLAS's HGEMM does not use padding to avoid bank conflict. We believe such an economical data layout deserves an in-depth analysis.

To summarize, the important lessons we learn from our analysis are:

- We should choose a larger thread block blocking size. Otherwise the performance of HGEMM will be bound by DRAM and L2 cache bandwidth. (256×256) is an ideal choice at the thread block level. While (256×256) is the largest blocking size we can choose, the performance can still be bound by DRAM bandwidth.
- We also need to enlarge the blocking size at the warp level; if not, the shared memory bandwidth will be a new bottleneck. (128×64) is our choice at the warp level.

B. Data Prefetching

Similar to the previous work, we hide the global and shared memory accesses latency by data prefetching (some works call it software pipelining), i.e., we load the data needed by the next iteration at the beginning of the current iteration. While this adds pressure to the registers, there are enough free registers to hold prefetched data. In our implementation, we have at least $(64 + 32) \times 8 = 768$ cycles to hide the `LDG` latency.

C. Instruction Scheduling

Once the blocking size is determined, the number of `HMMA` instructions and memory I/O instructions are fixed. The next question is how to place those instructions in a proper order (also called the instruction scheduling problem). Specifically, we care about how to insert memory I/O instructions into the `HMMA` sequence. Currently, there is no canonical way to determine the space between continuous memory I/O instructions. In [14], [15], the authors propose a trial-and-error strategy to find the space. We give a simple yet effective principle based on the CPI values to determine the space between continuous memory I/O instructions.

Take the `STS.128` instruction as an example. In cuBLAS 10.1's HGEMM, two `HMMA`s are used to interleave continuous `STS.128`. We argue that this space is not enough. The minimum number of `HMMA`s (#`HMMA`) required to interleave `STS.128` is computed as follows:

$$\#\texttt{HMMA} \times CPI_{\texttt{HMMA}} \geq 4 \times CPI_{\texttt{STS.128}}, \qquad (6)$$

where 4 stands for 4 processing blocks within an SM. The idea behind this expression is that *the time used to do computation should not be shorter than the time to do memory I/O*. Based on our measured CPI values, we need at least 5 `HMMA`s to interleave `STS.128`.

Fig. 4 shows the throughput of interleaving `STS.128` with 2 `HMMA`s (STS2) and interleaving `STS.128` with 5 `HMMA`s (STS5) respectively. The average speedup of STS5 over STS2 is $1.13\times$, and the maximum speedup is $1.26\times$.

D. Shared Memory Data Layout

Based on the discussion in Section VI-A, the thread block blocking size is chosen as $(256 \times 256 \times 32)$. We need to store $A_{256\times32}$ (row-major) and $B_{32\times256}$ (column-major) in shared memory. The naive strategy to store $A_{256\times32}$ and $B_{32\times256}$ in shared memory is to allocate two 256×32 arrays and store all the elements one after the other. In CUDA C++, this can be written as `A[256][32]` and `B[256][32]` (column-major).

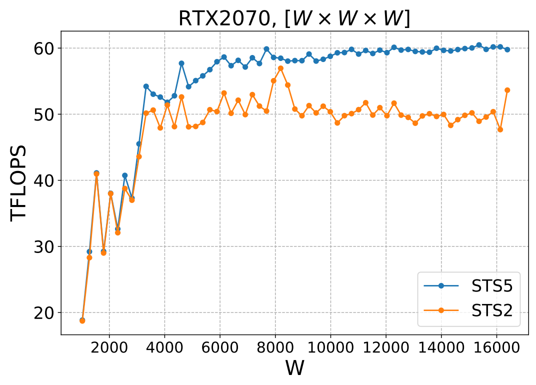


Fig. 4: Our HGEMM's throughput on RTX2070 when interleaving `STS.128` with 2 `HMMA`s (STS2) and with 5 `HMMA`s (STS5) respectively.

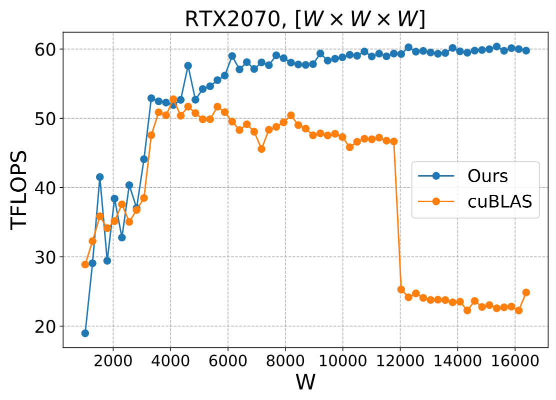


Fig. 6: Throughput of our and cuBLAS's HGEMM with square matrices on RTX2070.

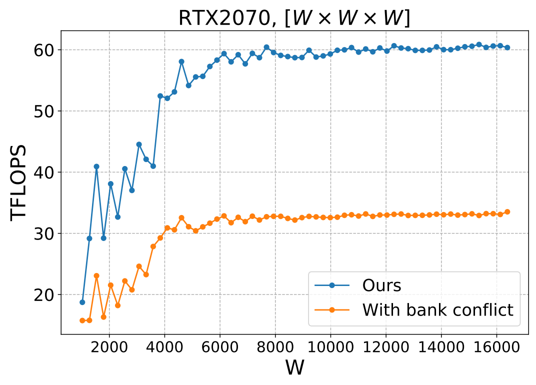


Fig. 5: Our HGEMM's throughput on RTX2070 when padding to avoid shared memory bank conflict and no padding with bank conflict.

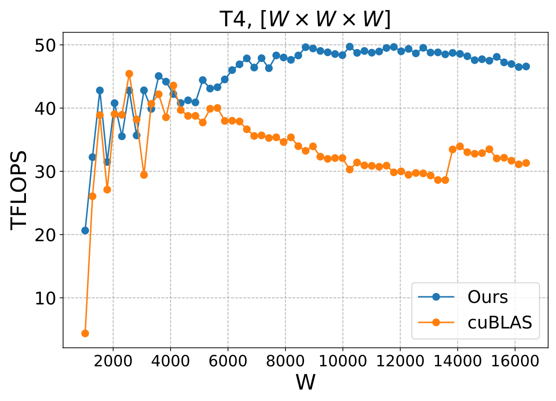


Fig. 7: Throughput of our and cuBLAS's HGEMM with square matrices on T4.

We evaluate this naive layout and show its throughput in Fig. 5. As the figure shows, the naive layout slows the HGEMM by half compared with our optimized data layout.

Instead, our proposed data layout is to pad 8 half elements every other row, i.e., the offset of element `A[row][col]` can be computed as $\texttt{row} \times 32 + \texttt{row}\%2 \times 8 + \texttt{col}$. This data layout makes both the store-to and load-from shared memory bank conflict-free.

VII. Evaluation

In this section, we compare the performance of our optimized HGEMM to cuBLAS' HGEMM. All results in this section are collected on Ubuntu 18.04 with CUDA 10.1 and cuBLAS 10.1. All input and output are in half precision. Input and output data are stored in the GPU memory, which is the case for many deep learning applications. The cost of the memory allocation is not included in the timing.

We compare the performance of our optimized HGEMM with cuBLAS 10.1 on two NVIDIA Turing GPUs, namely RTX2070 and T4. We set the memory clock rate and GPU clock rate on T4 to 5001GHz and 1590GHz respectively, while RTX2070 does not support setting clock rate. We record the running time of each kernel with *cuda event* [18]. The reported throughput is averaged over 10 measurements.

Consider the matrix multiplication problem $C = AB$, where matrix A is in row-major layout and B is in column-major layout. To leverage Tensor Cores when calling cuBLAS' routine, we set the cuBLAS math mode to `CUBLAS_TENSOR_OP_MATH` and then invoke `cublasGemmEx()`. Since the matrices are relatively large in deep learning applications and Tensor Cores are targeting large matrices, we evaluate matrix size from 1024 to 16384 with step size 256. Our evaluation starts with square matrices. To evaluate the performance on rectangular matrices, we vary one dimension of HGEMM at a time. We test the performance of HGEMM with shape $[2W \times W \times W]$, $[W \times 2W \times W]$, $[W \times W \times 2W]$, $[4W \times W \times W]$, $[W \times 4W \times W]$ and $[W \times W \times 4W]$.

A. Performance on Square Matrices

a) RTX2070: Fig. 6 shows the TFLOPS sustained by our HGEMM and cuBLAS's HGEMM on RTX2070. When the matrix size is small ($W < 4096$), our implementation achieves comparable performance with cuBLAS. In this case, the performance is prone to the kernel launch overhead and the

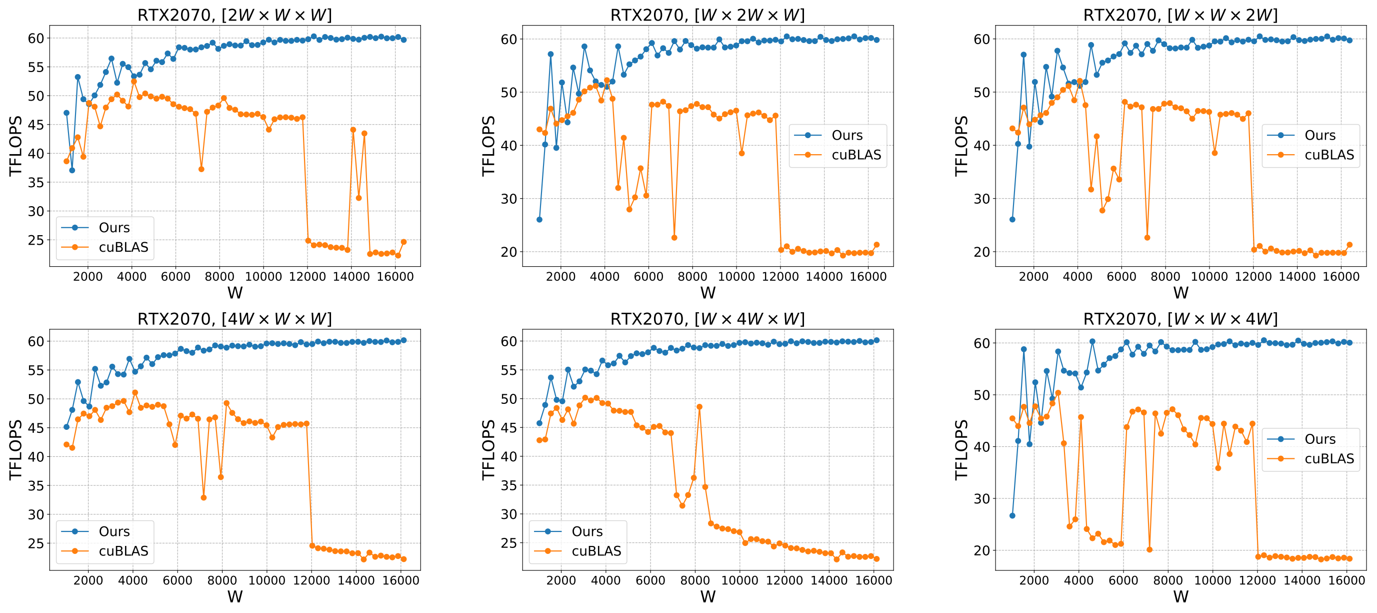


Fig. 8: Throughput of our implementation and cuBLAS's under different matrix shape and size on RTX2070.

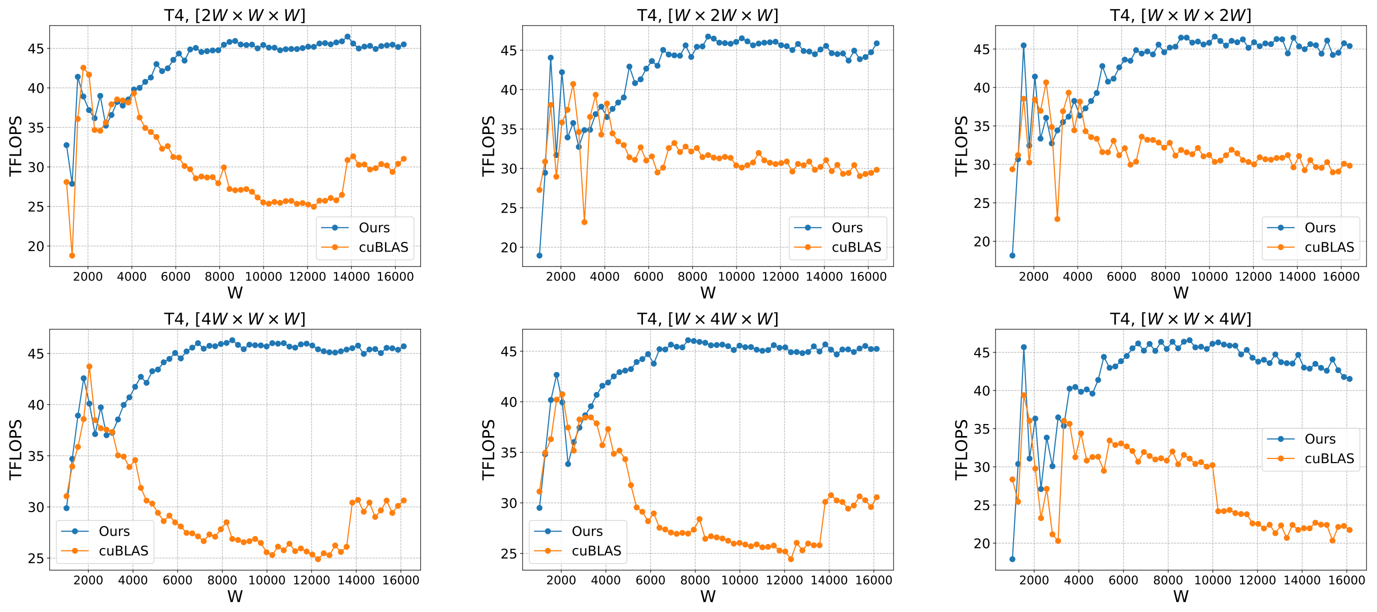


Fig. 9: Throughput of our implementation and cuBLAS's under different matrix shape and size on T4.

small number of thread blocks. As the matrix size grows, the throughput of our HGEMM increases steadily until it reaches the device peak. The highest throughput of our implementation is 60.37 TFLOPS, higher than the device peak (59.7 TFLOPS). This may be due to the noise in time measurement and the rounding error when calculating throughput. In comparison, the throughput of cuBLAS's HGEMM drops slightly when the matrix size is greater than 4096. We observe a sharp drop in performance when W reaches 12032. We suspect that this is because the L2 cache blocking strategy of cuBLAS fails at that size. The maximum throughput of cuBLAS is 52.75 TFLOPS when $W = 4096$. The maximum speedup over cuBLAS is $2.7\times$ when $W = 16128$. The average speedup over cuBLAS is $1.55\times$.

b) T4: Fig. 7 shows the TFLOPS sustained by our HGEMM and cuBLAS's implementation on T4. Similar to the case on RTX2070 when matrix size is smaller than or equal to 4096, our implementation achieves comparable performance with cuBLAS.

As the matrix size grows, the throughput of our HGEMM increases steadily until it reaches around 50 TFLOPS. The highest throughput of our implementation is 49.71 TFLOPS, which is 76% of the device peak (65 TFLOPS). The difference between the performance on RTX2070 and T4 clearly shows the importance of DRAM bandwidth. In comparison, the throughput of cuBLAS's HGEMM drops slightly when the matrix size is greater than 4096. The maximum throughput of cuBLAS is 45.43 TFLOPS when $W = 2560$. The maximum speedup over cuBLAS is $1.7\times$ when $W = 13312$. The average speedup over cuBLAS on T4 is $1.53\times$.

We also notice a trend that the throughput of our implementation starts to fall when W exceeds 12800. We believe that this indicates that the performance of our implementation is harmed by the low L2 cache hit rate and relatively low DRAM bandwidth on T4.

B. Performance on Rectangle Matrices

a) RTX2070: Fig. 8 shows the TFLOPS sustained by our HGEMM and cuBLAS's implementation on RTX2070 on rectangular matrices. The trend is similar to the case of the square matrices. Our implementation consistently achieves near-to-peak throughput. The maximum speedup over cuBLAS is $3.23\times$ when $W = 14848$ and shape is $[W \times W \times 4W]$. The average speedup for rectangular matrices on RTX2070 is $1.77\times$.

b) T4: Fig. 9 compares the TFLOPS sustained by our implementation with cuBLAS on T4 on rectangular matrices. The trend is similar to the case of the square matrices. And our implementation consistently achieves high throughput. The maximum speedup over cuBLAS is $2.17\times$ when $W = 15360$ and shape is $[W \times W \times 4W]$. The average speedup for rectangular matrices on T4 is $1.45\times$.

C. Summary

The average speedup over cuBLAS 10.1 for all settings on two devices is $1.61\times$. For both our implementation and cuBLAS's HGEMM, the throughput is higher on RTX2070 than on T4 while the device peak of T4 is higher. The major difference between RTX2070 and T4 is that RTX2070's DRAM bandwidth (380GB/s) is 58% higher than T4's DRAM bandwidth (240GB/s). We argue that this is a strong evidence that the performance of HGEMM on T4 is bound by the bandwidth of DRAM.

VIII. Conclusion and Future Work

We demystify, for the first time, how half-precision Tensor Cores on Turing GPUs work. Specifically, we present how the `HMMA.1688.F16` instruction is used to manipulate Tensor Core. We point out the basic element in half-precision is an 8×8 matrix, which can be stored in registers with the same index within a warp, and depict the row-major and column-major order to store the 8×8 half-precision matrix. We also benchmark the throughput and latency of the `HMMA.1688.F16` instruction.

We present detailed benchmarks of the GPU memory system. In addition to the benchmark techniques already known to the community, we measure the CPI values of memory instructions at the SASS level with higher accuracy. We use the CPI value, which is not accessible with previous techniques, to guide the process of selecting blocking size and scheduling instructions.

We have presented an optimized Tensor Core-based HGEMM on Turing GPUs and our implementation achieves on average $1.6\times$ and up to $3\times$ speedup over cuBLAS 10.1's HGEMM. The speedup mainly comes from the larger blocking size together with optimizations like smarter load/store instruction scheduling and novel data layout in shared memory.

Future work includes a deeper look into the L2 cache-friendly thread block launch order, demystifying Tensor Cores with single-precision accumulators and integer data type, automatic tools to simplify programming while achieving near to peak performance.

References

[1] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cuDNN: Efficient primitives for deep learning," *arXiv preprint arXiv:1419.0759*, vol. abs/1410.0759, 2014.

[2] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[3] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *arXiv preprint arXiv:1706.03762*, 2017.

[4] NVIDIA. (2018) NVIDIA Tesla V100 GPU architecture. [Online]. Available: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[5] S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng, and J. S. Vetter, "NVIDIA tensor core programmability, performance & precision," *arXiv preprint arXiv:1803.04014*, 2018.

[6] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for floating-point programs and multicore architectures," Lawrence Berkeley National Lab., Tech. Rep., 2009.

[7] M. D. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," in *ASPLOS*, 1991.

[8] J. Appleyard and S. Yokim. (2017) Programming tensor cores in CUDA 9. [Online]. Available: https://devblogs.nvidia.com/programming-tensor-cores-cuda-9

[9] K. Andrew, M. Duane, D. Julien, and T. John. (2018) Cutlass. [Online]. Available: https://github.com/NVIDIA/cutlass

[10] Z. Jia, M. Maggioni, B. Staiger, and D. P. Scarpazza, "Dissecting the NVIDIA volta GPU architecture via microbenchmarking," *arXiv preprint arXiv:1804.06826*, 2018.

[11] Z. Jia, M. Maggioni, J. Smith, and D. P. Scarpazza, "Dissecting the NVidia Turing T4 GPU via microbenchmarking," *arXiv preprint arXiv:1903.07486*, 2019.

[12] X. Mei and X. Chu, "Dissecting GPU memory hierarchy through microbenchmarking," *IEEE TPDS*, 2017.

[13] J. Lai and A. Seznec, "Performance upper bound analysis and optimization of SGEMM on Fermi and Kepler GPUs," in *CGO*, 2013.

[14] X. Zhang, G. Tan, S. Xue, J. Li, K. Zhou, and M. Chen, "Understanding the GPU microarchitecture to achieve bare-metal performance tuning," ser. PPoPP, 2017.

[15] NervanaSystems. (2016) Maxas. [Online]. Available: https://github.com/NervanaSystems/maxas

[16] X. Li, Y. Liang, S. Yan, L. Jia, and Y. Li, "A coordinated tiling and batching framework for efficient GEMM on GPUs," in *PPoPP*. ACM, 2019, pp. 229–241.

[17] NVIDIA. (2018) NVIDIA Turing GPU architecture. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

[18] M. Harris. (2019) How to implement performance metrics in CUDA C/C++. [Online]. Available: https://devblogs.nvidia.com/how-implement-performance-metrics-cuda-cc/

[19] NVIDIA. (2019) Parallel thread execution ISA. [Online]. Available: https://docs.nvidia.com/cuda/parallel-thread-execution/