Dissecting Tensor Cores via Microbenchmarks: Latency, Throughput and Numeric Behaviors

Wei Sun, Ang Li, Tong Geng, Sander Stuijk, Henk Corporaal

Abstract—Tensor Cores have been an important unit to accelerate Fused Matrix Multiplication Accumulation (MMA) in all NVIDIA GPUs since Volta Architecture. To program Tensor Cores, users have to use either legacy wmma APIs or current mma APIs. Legacy wmma APIs are more easy-to-use but can only exploit limited features and power of Tensor Cores. Specifically, wmma APIs support fewer operand shapes and can not leverage the new sparse matrix multiplication feature of the newest Ampere Tensor Cores. However, the performance of current programming interface has not been well explored. Furthermore, the computation numeric behaviors of low-precision floating points (TF32, BF16, and FP16) supported by the newest Ampere Tensor Cores are also mysterious. In this paper, we explore the throughput and latency of current programming APIs. We also intuitively study the numeric behaviors of Tensor Cores MMA and profile the intermediate operations including multiplication, addition of inner product, and accumulation. All codes used in this work can be found in https://github.com/sunlex0717/DissectingTensorCores.

Index Terms—GPU, Tensor Cores, Numeric profiling, Ampere, Turing, Microbenchmark

1 Introduction

General Matrix Multiplication (GEMM) is a fundamental computation pattern in modern HPC applications and especially deep learning applications. To meet the increasing demands of high throughput GEMM, many commercial hardware accelerators have been designed such as NVIDIA Tensor Cores [28], Google TPUs [15], Intel Nervana [11] and Xilinx Versal AI Engines [9]. NVIDIA introduced the Tensor Cores Unit first in its Volta Architecture [28] and integrated this special computation unit in the GPGPU architecture. Tensor Cores can provide significant speed-up compared to traditional CUDA cores and more numeric precision choices to satisfy different demands [8, 10, 21, 26].

NVIDIA so far has released three generations of Tensor Cores - Volta [28], Turing [29], and Ampere [30]. Now Tensor Cores are becoming the standard components of all NVIDIA GPU products including high-end server GPUs (e.g. V100 and A100), Gaming GPUs (e.g. RTX20xx and RTX30xx) as well as Embedded GPUs (e.g. Jetson Xavier and Orin). However, the programming model and behaviors of the Tensor Cores are significantly different from previously well-explored **CUDA cores**. Furthermore, there are also noticeable differences between different Tensor Cores generations. For instance, Ampere Tensor Cores support fine-grained N: M sparse acceleration for Sparse Matrix Multiplication, which can only be programmed via the new mma instructions instead of previously studied legacy wmma instructions [23, 39]. There are also some incompatible instructions between Ampere and prior generations. For example, Ampere Tensor Cores do not support the HMMA.884 Assembly instruction; the corresponding mma.m8n8k4 PTX instruction is essentially compiled into a set of Floating point unit (FPU) Assembly instructions that are 10x slower than the expected Tensor Cores performance. In contrast, previous observations of the Volta Tensor Cores [39] suggest that this HMMA.884 is the fundamental machine code (SASS), and all legacy wmma.mma instructions will be compiled into a set of HMMA.884 SASS codes. Therefore, it is vital to understand the programming model and behavior of the Tensor Cores for achieving the best possible performance on different GPU architectures.

Despite the previous efforts on microbenchmarking Tensor Cores [6, 13, 14, 23, 24, 39, 45] from performance and numeric perspectives. There is no existing work that evaluates the current programming APIs with instruction-level microbenchmarking and numeric studies. Table 2 in Section 3 compares our work with previous Tensor Cores studies. Current programming APIs have significant advantages over legacy wmma APIs such as more flexible operand shapes, better performance, and new sparse acceleration on Ampere Tensor Cores. Therefore, a comprehensive and up-to-date study based on the new programming interface is necessary to complement the findings of existing literature. Our research fulfills this gap by making the following contributions:

- A new set of microbenchmarks to explore the three core PTX instruction sets closely related to Tensor Cores — ldmatrix, mma, and mma.sp.
- Benchmarking the Tensor Cores instruction latency and throughput. We provide in-depth analysis and provide programming guidelines which are needed to implement custom applications on Tensor Cores that are not well supported by vendor libraries.
- Profiling the numeric behavior of low-precision floating data types supported in newest Ampere Tensor Cores - TF32, BF16, and FP16.

- *Wei Sun, Sander Stuijk and Henk Corporaal are with Electronic System Group, Eindhoven University of Technology, the Netherlands. E-mail: {w.sun,s.stuijk,h.corporaal}@tue.nl*
- *Ang Li and Tong Geng are with the Physical and Computational Sciences Directorate, Pacific Northwest National Laboratory, WA, USA. E-mail: {ang.li,tong.geng}@pnnl.gov*

Manuscript received May 26, 2022; revised October 21, 2022.

Our work provides comprehensive and up-to-date information on the Tensor Cores. Our microbenchmarks cover the programming interface, latency, and throughput of Tensor Cores instructions as well as numeric behaviors of low-precision floating point operations. To the best of our knowledge, this paper is the first systematic study on recent Tensor Cores generations (Turing and Ampere) using the new programming interface.

The rest of the paper is organized as follows: Section 2 introduces the background knowledge of NVIDIA GPUs and Tensor Cores. Section 3 summarizes the related work and highlights the differences between our work. Section 4 introduces the microbenchmark methodology used in this paper. Section 5, 6 and 7 microbenchmark the performance of mma, mma.sp and ldmatrix instructions respectively. Section 8 presents our numeric studies. Section 9 concludes our findings.

2 Background of Tensor Cores GPUs

We first introduce the general architecture of modern Tensor Cores GPU in Section 2.1, and then we compare the three Tensor Cores generations and highlight their differences in Section 2.2.

2.1 Background of Modern GPU Architecture

Tensor Cores is a domain-specific computation unit integrated into NVIDIA GPUs for accelerating matrix multiplication (MM), which performs D = A×B + C, where A and B are input matrices with the shape of m×k and k×n, respectively; C is accumulator and D is the result.

Modern GPU architecture consists of a certain amount of Streaming Multiprocessor (SM) which works as the fundamental processing unit of the GPU. These SMs are further connected to the GPU device memory/global memory which is shared by all SMs. Each SM has its internal caches, Shared Memory, Register File, CUDA cores, Tensor Cores, Load Store units, and other special units. Figure 1 illustrates the abstract SM architecture of Tensor Cores integrated GPUs. Each SM consists of four warp schedulers or four sub-cores to issue four warp instructions simultaneously. Tensor Cores use the same memory hierarchy as CUDA cores. Both Tensor Cores and CUDA cores can fetch data through direct addressing (*ld* instruction) from the Shared Memory or global memory via the **per-thread**[1] scheme. Tensor Cores have two special load instructions – recent *ldmatrix* and legacy *wmma.load* via the **per-warp**[2] scheme.

In this paper, we exclude the microbenchmark experiments and discussions on global memory access because 1) Using Shared Memory as the buffer for global memory to increase data reuse has been a standard optimization technique for accelerating GEMM-like applications on GPUs. 2) The novel asynchronous global memory copy introduced in Ampere Architecture facilitates the software data pipeline by using Shared Memory as the staging storage to overlap the computation with the data transfer from global memory to Shared Memory [30]. 3) The ldmatrix PTX instruction can only load data from Shared Memory. Therefore, without losing generality, we assume the input data has already been fetched into the Shared Memory from global memory since our goal is to microbenchmark the Tensor Cores. We present ablation experiments in Appendix A.1 to further demonstrate the advantages of using shared memory and asynchronous copy.

1. Per-thread means the instruction is executed by a single thread independently; both behavior and result of the thread are deterministic.
2. Per-warp means the instruction is executed by the 32 threads within the same warp cooperatively; neither behavior nor the result of individual thread is deterministic.

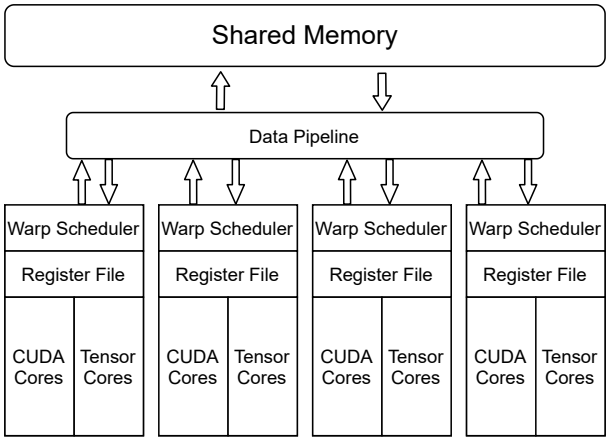


Fig. 1: Simplified architecture overview of Tensor Cores GPU SM [28, 29, 30]. Each SM consists of four sub-warps. Each sub-core has its warp scheduler, register file, CUDA cores, and Tensor Cores.

2.2 The Evolution of Tensor Cores

Up to date, there are three generations of Tensor Cores in the market — Volta, Turing, and Ampere. NVIDIA recently announced the fourth generation of Tensor Cores GPU Architecture known as Hopper [35]. However, Hopper GPUs are not publicly released yet.

Table 1 compares the three released Tensor Cores generations and preliminary features of the fourth-generation Hopper Tensor Cores. Volta Tensor Cores are the first generation and support only FP16 as the input data type. Turing Tensor Cores are the enhanced version of Volta Tensor Cores and support three extra data types – INT8, INT4, and Binary [21]. The third-generation Ampere Tensor Cores introduce acceleration for sparse matrix multiplication with fine-grained structured sparsity and a new machine learning data type called bfloat16 (BF16). Furthermore, Ampere Architecture redesigns the micro-architecture of Tensor Cores. Unlike Volta and Turing Architecture which have eight Tensor Cores per SM and each Tensor Core performs a 4×4×4 MM (i.e. m = n = k = 4), there are only four Tensor Cores per SM and each Tensor Core performs an 8×4×8 MM. The fourth-generation Hopper will bring a new 8-bit floating-point numeric precision (FP8) for accelerating certain machine learning applications that exhibit higher tolerance for training based on low-precision data types. Furthermore, it seems INT4 and Binary are no longer discussed in the Hopper Whitepaper [35]. Section 8 will give more discussions on the numeric behaviors of low-precision floating-point data types of Tensor Cores (i.e. TF32,BF16 and FP16).

TABLE 1: Comparisons of the properties of different generations of Tensor Cores. Note that Hopper GPU is not available on the market at this moment, the features are preliminary and could be adjusted according to the vendor's documentation [35]. **NA** means not being documented by the vendor yet.

Architecture	Representative products	Numeric types	TCs/SM	m×n×k	Sparse Acceleration	Programmability
Volta	V100, Jetson Xavier	FP16	8	4×4×4	No	wmma
Turing	T4,RTX20x	FP16, INT8, INT4, Binary	8	4×4×4	No	wmma, ldmatrix, mma
Ampere	A100, RTX30x, Jetson Orin	FP16, BF16, TF32, FP64, INT8, INT4, Binary	4	8×4×8	fine-grained 50% sparsity	wmma, ldmatrix, mma, mma.sp
Hopper	H100	FP16, BF16, TF32, FP64, FP8, INT8	4	NA	fine-grained 50% sparsity	wmma, ldmatrix, mma, mma.sp

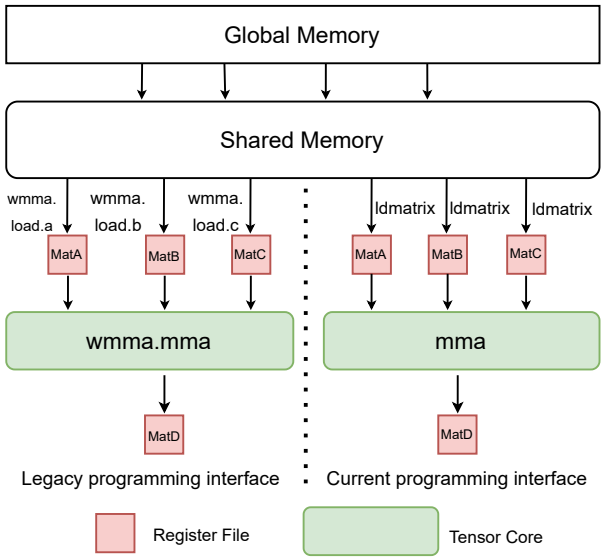


Fig. 2: Legacy and current programming interface for Tensor Cores.

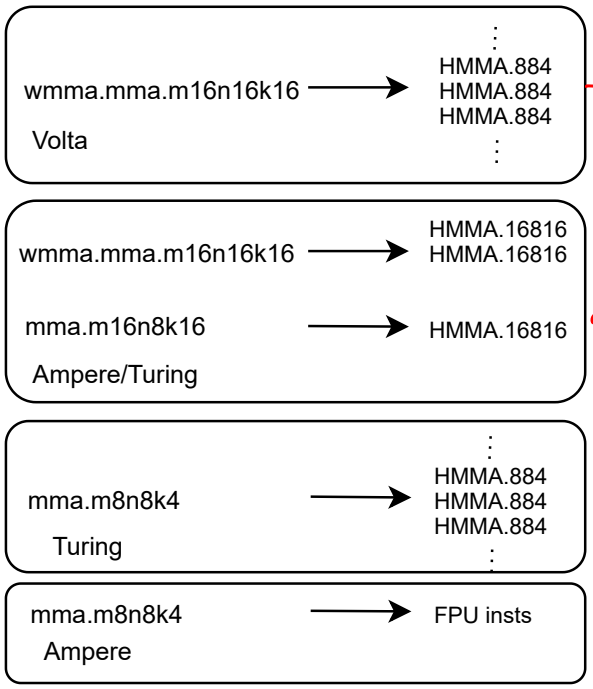


Fig. 3: The compilation from legacy and current PTX instructions to SASS assembly codes [13, 14, 39, 45].

The programming of Tensor Cores has also evolved over the generations. Figure 2 illustrates the legacy and current programming interface for Tensor Cores. The first step is to load data from Shared Memory to the Register File and then use Tensor Cores to accelerate the matrix computation. Legacy wmma interface was introduced with first-generation Volta Tensor Cores which have been extensively studied by previous work [23, 24, 39, 45]. In general, the legacy wmma programming interface can satisfy basic usage of Tensor Cores and requires less programming efforts, because wmma.load can help manage the special input operand storage layout in the Register File for Tensor Cores. However, wmma instructions can only leverage limited features of the Tensor Cores [22]. Specifically, it can not use sparse acceleration and has fewer choices of the matrix operand shape. Furthermore, wmma.load instructions have more strict requirements for the data layout in Shared Memory which will be introduced in Section 7. By contrast, the new programming interface provides access to all features of the Tensor Cores (i.e. more matrix operand shapes and novel sparse acceleration). The vendor's library CUTLASS [33] also chose the new programming interface as the underlying implementation for the best possible performance. Therefore, users should use the new programming interface when seeking for best possible performance or exploiting the new sparse acceleration feature.

Figure 3 uses an example to illustrate the differences between legacy wmma.mma and current mma PTX instructions. Previous studies [14, 39, 45] reveal that on Volta Tensor Cores, every wmma.mma instruction will be compiled to a set of HMMA.884 SASS instruction. By contrast, when using a mma instruction on Ampere or Turing Tensor Cores, it will be compiled to a single HMMA SASS instruction. On the other hand, when attempting to use the legacy wmma.mma on Turing/Ampere Tensor Cores, it will be compiled into several new HMMA instructions [13]. In the example of Fig. 3, one legacy wmma.mma.m16n16k16 is complied into two HMMA.16816 (corresponding to new mma.m16n8k16) instructions. Note that there is one special instruction mma.m8n8k4. On Turing Tensor Cores, it will be compiled into a couple of HMMA.884 instructions which are similar to Volta Tensor Cores. However, on Ampere Tensor Cores, it will be compiled into a set of FPU instructions which will eventually run on the CUDA cores and lead to inferior performance than expected for the Tensor Cores.

TABLE 2: Comparisons with previous Tensor Cores microbenchmark literature. Note that [13] and [14] are general microbenchmark studies and cover Tensor Cores as one Section. Others [6, 23, 24, 39, 45] are focusing on Tensor Cores. [6] only studies the numeric behaviors and performance is not evaluated. Our work is the first comprehensive study that evaluates current mma programming APIs and sparse Tensor Cores with instruction-level microbenchmarking and numeric studies.

Literature	GPU Arch	Dense FMA	Sparse FMA	Data movement	Performance evaluations	Numeric studies	Assembly studies
[23]	Volta	wmma.mma	–	wmma.load	Vendor libraries benchmark	FP16	–
[24]	Volta	wmma.mma	–	wmma.load	Vendor libraries benchmark	–	–
[14]	Volta	wmma.mma	–	wmma.load	Vendor libraries benchmark	–	✓
[13]	Turing	wmma.mma	–	wmma.load	Vendor libraries benchmark	–	✓
[39]	Turing Volta	wmma.mma	–	wmma.load	Instruction-level microbenchmark	–	✓
[45]	Turing Volta	wmma.mma	–	wmma.load	Instruction-level microbenchmark	–	✓
[6]	Ampere Turing Volta	wmma.mma	–	wmma.load	–	TF32,BF16, FP16	–
Ours	Ampere Turing	mma	mma.sp	ldmatrix	Instruction-level microbenchmark	TF32,BF16, FP16	✓

3 Related Work

GPU evaluations and microbenchmarks [6, 13, 14, 20, 23, 24, 25, 39, 43, 45] are important technique to explore the unknown characteristics of GPU architecture like memory hierarchy, throughput, latency and numeric behaviors.

Table 2 summaries the existing Tensor Cores microbenchmark studies and compares them with our work. [23] studies the Volta Tensor Cores with legacy wmma programming interface and benchmarks the vendor libraries [32, 33]. The authors also profile the numeric loss when using half (FP16) precision on Volta Tensor Cores. [24] also benchmarks the vendor libraries and further studies the software optimization techniques when using wmma APIs to program Volta Tensor Cores. However, the experiments and evaluations of these two work [23, 24] are only based on old Volta Tensor Cores. Furthermore, they do not provide instruction-level microbenchmarks (i.e. instruction latency and throughput) or discussions (e.g. Assembly codes studies).

[14] and [13] explore the characteristics of Volta and Turing GPUs respectively. They cover the vendor libraries benchmarking and Assembly code studies of legacy wmma APIs for Tensor Cores. However, these two works are generic GPU studies. The discussions on Tensor Cores are limited; the numeric behaviors and instruction-level performance are not treated.

[45] demystifies Volta and Turing Tensor Cores through Assembly code benchmarking and further optimizes the half-precision matrix multiplication on Tensor Cores. The authors focus on exploring the Assembly level optimizations for matrix multiplication performance on Tensor Cores and achieve better performance than the vendor's library released at that moment. However, programming Tensor Cores via Assembly codes (SASS) is miserable since SASS codes are not officially documented by NVIDIA and there is not an official Assembler. Although Assembly codes may expose more optimization opportunities for certain applications [18], users have to rely on third-party assemblers [46, 47] developed through reverse engineering, which may be not stable and can be error-prone. In this work, we focus our studies on the PTX level, since PTX instructions are fully documented and supported by NVIDIA.

[39] studies the legacy wmma instructions on Turing and Volta Tensor Cores and proposes a Tensor Core model in GPGPU-SIM [2], based on their findings that HMMA.884 is the fundamental execution pattern of the Tensor Cores. However, since there are significant differences between the newest Ampere and older Turing/Volta Tensor Cores as discussed in Section 2.2, the behaviors and performance of Ampere Tensor Cores may not be simulated accurately by the old model.

[6] focuses on the numeric behaviors of Tensor Cores and explores the underlying rounding modes and subnormal behaviors of TF32, BF16, and FP16 data types. However, the performance (i.e. latency and throughput) of Tensor Cores is not evaluated.

Compared with the aforementioned Tensor Cores studies, we make the following different contributions:

- We evaluate the current mma APIs (ldmatrix, mma, and mma.sp) instead of legacy wmma APIs. Current mma APIs provide more operand shapes and new sparse matrix multiplication acceleration, but have not been well studied by existing literature [6, 13, 14, 23, 24, 39, 45]. Our work fulfills this gap.
- We microbenchmark the instruction-level performance (i.e. throughput and latency). Compared

to benchmarking vendor libraries [32, 33], our instruction-level studies can give insights and programming guidelines that the latency of different mma instructions with different operand shapes as well as how maximum performance can be achieved. This information is important to users who need to implement their applications, which are not well supported by libraries, on Tensor Cores [8, 21, 22, 41].

- We intuitively study the numeric behaviors of three low-precision floating points (FP16, BF16, and TF32) w.r.t IEEE standard FP32 precision. We profile the numeric errors of three operations including multiplication, the addition of inner product, and accumulation. Our experiments complement findings in [6] by comparing the error level and behaviors of the three data types, which can help users to decide which data types should be considered for their applications.

4 Microbenchmark Methodology

The goal of our microbenchmark is to offer the following performance information for each instruction:

1) *Latency*: Latency is the duration (cycles) of starting to issue the instructions to the execution pipeline until the results become available for the next usage. Since GPU is a parallel architecture, there can be multiple instructions running simultaneously on each SM. When there is only one instruction per warp and one warp per SM, the measured latency is called *completion/issue latency*, which reveals the length of the pipeline.
2) *Throughput/bandwidth*: For data movement instruction, throughput is measured as bytes/clock cycle(clk)/SM. For computation instructions, throughput is FMA/clk/SM where FMA stands for fused multiplication accumulation. As the definition, d = a×b+c counts as one FMA, and m×n×k matrix multiplication counts as m×n×k FMAs. Throughput/bandwidth can be measured by dividing the workload (i.e. bytes/SM or FMA/SM) by latency (cycles). Workload / time

Figure 4 shows the core code piece of how to measure the latency of the instruction with different Instruction Level Parallelism (ILP). Tensor Cores instructions (e.g. mma in the example) are executed in a per-warp scheme so we use warp-level synchronization at the end of each iteration to avoid optimizations (i.e. parallelism) across different iterations. The latency is then computed by taking the average of ITERS iterations. If ILP = 1 and we only launch one CUDA thread block with one warp (32 threads), the measured latency will be the instruction completion/issue latency. By increasing ILP and launching more warps per SM, we can measure the throughput and latency under different parallel configurations which can then give the architectural and programming insights of the Tensor Cores.

In summary, we will conduct our microbenchmark experiments for each instruction as follows:

1) Measure the completion/issue latency by choosing ILP = 1 and launching one warp per SM.

ILP = 1 → True latency

ILP ≥ 2 → Practical parallel throughput & latency.

```
1  //.....
2  // start timing
3  asm volatile("mov.u64 %0, %%clock64;" : "=l"(start)::"memory");
4  for (int j = 0; j < ITERS; ++j) {
5    // inline PTX Tensor Core mma instruction. D = A*B + D
6    asm volatile(
7        "mma.sync.aligned.m16n8k8.row.col.f32.bf16.bf16.f32
8        {%0,%1,%2,%3}, {%4,%5}, {%6}, {%7,%8,%9,%10};\n"
9        : "=f"(D[0]), "=f"(D[1]), "=f"(D[2]), "=f"(D[3])
10       : "r"(A[0]), "r"(A[1]), "r"(B[0]),
11         "f"(C[0]), "f"(C[1]), "f"(C[2]), "f"(C[3]));
12   #if (ILPconfig >= 2)
13   asm volatile(
14       "mma.sync.aligned.m16n8k8.row.col.f32.bf16.bf16.f32
15       {%0,%1,%2,%3}, {%4,%5}, {%6}, {%7,%8,%9,%10};\n"
16     : "=f"(D[4]), "=f"(D[5]), "=f"(D[6]), "=f"(D[7])
17     : "r"(A[2]), "r"(A[3]), "r"(B[1]),
18       "f"(C[4]), "f"(C[5]), "f"(C[6]), "f"(C[7]));
19   #endif
20   // more ILP codes not shown here
21   __syncwarp();
22 }
23 // stop timing
24 asm volatile("mov.u64 %0, %%clock64;" : "=l"(stop)::"memory");
```

Fig. 4: Code piece showing how to microbenchmark the performance of Tensor Cores. It contains a GPU timer (lines 3 and 24) and a for-loop to execute the body code mma instructions depending on the ILP configurations.

2) Measure the throughput and latency under different ILP and different warps per SM.

5 Dense FMA

As introduced in Section 2.2, there are two approaches to program dense Tensor Cores - legacy wmma.mma and current mma instructions. We focus on the new mma instructions microbenchmarks since the legacy wmma.mma has been extensively researched by previous work [6, 13, 14, 23, 24, 39, 45].

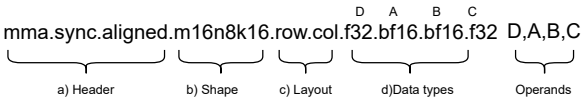
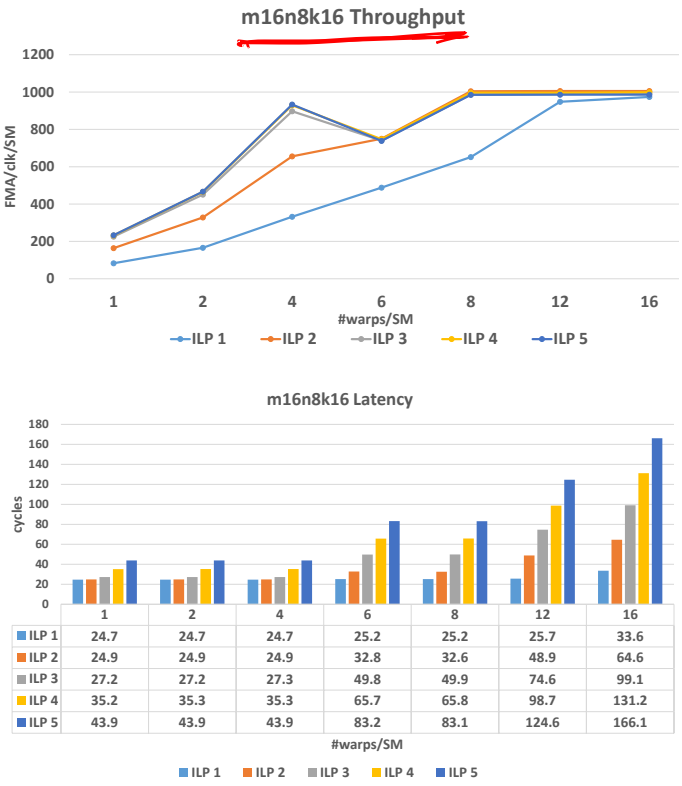


Fig. 5: The mma instruction performs a D = A×B + C MatMul. Segment m16n8k16 defines The shape of A(m×k), B(k×n), C(m×n), and D(m×n). The layout indicates the matrices are either row-major or col-major. The final instruction segment indicates the numeric types of D, A, B, and C respectively.

Figure 5 shows a mma instruction example. According to the vendor's documentation [37], for the BF16 data type, there are two choices of matrix shapes – m16n8k8 and m16n8k16. Depending on the shapes, the BF16 mma PTX instructions will be compiled to HMMA.1688.FP32.BF16 or to HMMA.16816.FP32.BF16 SASS assembly code. Since the Tensor Cores support many different data types as shown in Table 1, we present a detailed analysis for the BF16 data type on A100 Ampere Tensor Cores. The analysis for other data types is similar, so we provide the final results without losing generality.

Experimental results

Figure 6 shows the measured throughput and throughput latency of mma.m16n8k16 instruction on A100 Ampere Tensor Cores GPU under different ILPs and #warps/SM



	1	2	4	6	8	12	16
ILP 1	24.7	24.7	24.7	25.2	25.2	25.7	33.6
ILP 2	24.9	24.9	24.9	32.8	32.6	48.9	64.6
ILP 3	27.2	27.2	27.3	49.8	49.9	74.6	99.1
ILP 4	35.2	35.3	35.3	65.7	65.8	98.7	131.2
ILP 5	43.9	43.9	43.9	83.2	83.1	124.6	166.1

Fig. 6: Throughput and latency of mma.m16n8k16 instruction under different settings on A100 Tensor Cores.

(shortened as #warps). We start the experiments with ILP = 1 and #warps = 1 and then increase the ILP and #warps until convergence. Based on the results, we have findings as follows:

1) According to Section 4, the measured latency under ILP = 1 and #warp = 1 is completion latency. The instruction completion latency is therefore around 25 (cycles). The measured peak throughput is 1000 (FMA/clk/SM), almost matching the number of 1024 as claimed by the vendor [30].
2) By analyzing the latency and throughput under #warp = 1 and different ILPs, we observe that the peak throughput is 230 (FMA/clk/SM) and converges at ILP = 3. Increasing ILP further will not lead to higher throughput but significantly more latency cycles. This is because the instructions of a specific warp **can not** use idle hardware resources located in other sub-cores even if they are within the same SM. One warp can only claim the resources of one sub-core.
3) When #warps $\leq$ 4 (1, 2, 4), the throughput scales with #warps but the latency keeps unchanged. This observation confirms that the resources of the sub-core are isolated and each SM can issue four warps simultaneously. The maximum throughput can be achieved when #warps = 8 and ILP $\geq$ 2. Note that #warps = 4 with ILP $>$ 3 is also a convergence point, but it can not achieve the same performance as #warps = 8, ILP =2 (900 v.s. 1000). This observation indicates that allocating more than 1 warp per sub-core helps overlap the intra-warp synchronization stalls or other relevant overhead with starting issuing the next warp.
4) By analyzing the latency and throughput under ILP = 1 with different #warp, we observe that instructions from different warps can run in the same sub-core Tensor Cores computation pipeline. Specifically, increasing #warps from 4 to 8 (4×2) and 12 (4×3) will only lead to one extra cycle latency, but the latency increase significantly if #warp goes to 16. This is because that 12 warp with ILP = 1 is equivalent to four warps with ILP = 3 from the workload perspective. Based on our previous observation in **3)**, four warps with ILP = 3 can achieve the full Tensor Cores usage and increasing ILP will not lead to higher throughput but significantly longer latency.
5) #warps = 6 is a special case. First, the latency of #warps = 6 and #warps = 8 are always the same under different ILPs. Second, for ILP $\geq$ 3, the throughput drops if increasing #warps from 4 to 6. This is because when there is 6 warps resident in an SM, the first four warps will be issued to the Tensor Cores computation pipeline. Since the ILP = 3 is large enough to fulfill the whole Tensor Cores computation pipeline, the second two warps can not be issued until there are available resources freed by the first four warps. When the second two warps start to execute on two sub-cores, the other two sub-cores are idle which causes the drop in the overall throughput.
6) In general, to reach the peak performance when programming Tensor Cores using mma.m16n8k16 instruction, #warp should be at least four and ideally a multiple of 4. Although four warps with sufficient ILP (i.e. $\geq$ 3) can achieve near peak performance, eight warps with ILP $\geq$ 2 should be used whenever possible

Figure 7 presents the results of m16n8k8 instruction. Following the analysis of previous m16n8k16 instruction. We provide additional observations as follows:

7) The instruction completion latency is around 18 cycles and the peak throughput is around 1000 FMA/clk/SM.
8) Like the observations in **3)**, there are two convergence points – #warp = 4, ILP = 4 and #warp = 8, ILP = 3. However, the throughput gap between these two points is more significant (800 v.s. 1000). This suggests that the intra-warp synchronization stalls overhead of mma.m16n8k8 are significantly higher than previously discussed mma.m16n8k16 (i.e. 900 v.s. 1000). Therefore, when choosing mma.m16n8k8 to program Tensor Cores, at least eight warps should be allocated to achieve ideal performance.

Table 3 lists the rest of the data types supported in Ampere Tensor Cores. We summarize their completion latency and two key convergence points, one with 4 warps and the other one with 8 warps. Note that all instructions can achieve near peak performance except for the INT8 data type with m8n8k16 shape, which can only achieve near

TABLE 3: mma instructions with different data types on A100 Tensor Cores. The peak performance of FP16, TF32, INT8, INT4 and Binary are 1024, 512, 2048, 4096, and 16384 FMA/clk/SM respectively [30].

A/B	C/D	Shape	Completion Latency (cycles)	(#warp,ILP)	Latency (cycles)	Throughput (FMA/clk/SM)	(#warp,ILP)	Latency (cycles)	Throughput (FMA/clk/SM)
FP16	FP32	m16n8k16	24.7	4,3	27.4	897.6	8,2	32.6	1004.2
FP16	FP32	m16n8k8	17.7	4,4	20.5	800.2	8,3	25.3	974.1
FP16	FP16	m16n8k16	24.4	4,3	27.1	907.1	8,2	32.9	996.6
FP16	FP16	m16n8k8	17.7	4,4	19.1	860.9	8,3	24.5	1002.6
TF32	FP32	m16n8k8	25	4,3	28.2	435.9	8,2	33.3	492.4
TF32	FP32	m16n8k4	18.1	4,4	20.9	392.6	8,3	25.7	477.5
INT8	INT32	m8n8k16	15.9	4,4	20.1	813.2	8,2	16.4	998.3
INT8	INT32	m16n8k32	24.7	4,3	27.1	1812.4	8,2	32.9	1986.5
INT8	INT32	m16n8k16	17.6	4,4	20.9	1570.1	8,3	25.1	1965.1
INT4	INT32	m16n8k32	18.1	4,4	22.1	2971.1	8,3	27.1	3630.0
INT4	INT32	m16n8k64	26.1	4,3	28.1	3497.9	8,2	35.8	3660.8
Binary	INT32	m16n8k128	18.1	4,4	22.1	11884.3	8,3	27.1	14515.1
Binary	INT32	m16n8k256	26.0	4,3	28.1	13985.4	8,2	35.8	14643.4

TABLE 4: Tensor Cores performance of RTX3070Ti Ampere GPU [31]. Unlike A100 which is designed for high-end servers. RTX3070Ti is for graphical applications and there are noticeable differences between these two GPUs.

A/B	C/D	Shape	Completion Latency (cycles)	(#warp,ILP)	Latency (cycles)	Throughput (FMA/clk/SM)	(#warp,ILP)	Latency (cycles)	Throughput (FMA/clk/SM)
FP16	FP32	m16n8k16	33	4,1	33	248.2	8,1	64.8	252.7
FP16	FP32	m16n8k8	18.8	4,2	32.3	253.9	8,1	32.4	253.2
FP16	FP16	m16n8k16	24	4,2	32.2	509.4	8,1	32.3	506.9
FP16	FP16	m16n8k8	17.7	4,3	24	511.8	8,2	32.3	507.8
TF32	FP32	m16n8k8	33.3	4,1	33.4	122.6	8,1	64.6	126.8
TF32	FP32	m16n8k4	19.1	4,2	32.7	125.3	8,1	32.6	125.7
INT8	INT32	m8n8k16	15.9	4,4	19.3	848.9	8,2	16.2	1008.5
INT8	INT32	m16n8k32	24.3	4,2	32.2	1017.2	8,1	32.1	1023.2
INT8	INT32	m16n8k16	17.7	4,3	24.1	1018.2	8,2	32.6	1005.4
INT4	INT32	m16n8k32	17.3	4,3	24.9	1967.9	8,2	32.3	2031.7
INT4	INT32	m16n8k64	24.5	4,2	33.3	1967.9	8,1	32.5	2013.5
Binary	INT32	m16n8k128	17.3	4,3	24.8	7908.3	8,2	32.3	8127.2
Binary	INT32	m16n8k256	24.6	4,2	33.3	7871.9	8,1	32.5	8053.9

TABLE 5: Tensor Cores performance of RTX2080Ti Turing GPU [29].

A/B	C/D	Shape	Completion Latency (cycles)	(#warp,ILP)	Latency (cycles)	Throughput (FMA/clk/SM)	(#warp,ILP)	Latency (cycles)	Throughput (FMA/clk/SM)
FP16	FP32	m16n8k8	17.3	4,2	32.5	252.4	8,1	32.1	255.1
FP16	FP16	m16n8k8	14.7	4,2	17.5	467.9	8,1	16.1	509.4
INT8	INT32	m8n8k16	11	4,3	14.5	846.1	8,2	16.2	1012.6

m8n8k16 → Full for Turing

INT8 on A100 only achieves half peak throughput ⇒ Old & Unprepared for A100

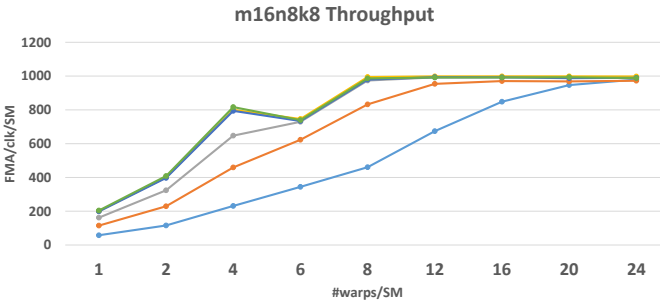
half peak performance. By comparing the performance of this instruction on Turing Tensor Cores (Table 5), which can achieve near peak performance. We find the reason can be that m8n8k16 is an old shape optimized for Turing Tensor Cores, the Ampere Tensor Cores prefer new shapes with m16n8 (i.e. m16n8k16 and m16n8k32).

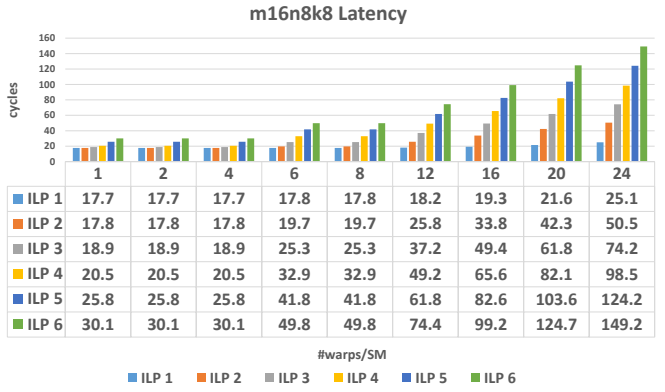
Table 4 summarizes the dense Tensor Cores performance on an RTX3070Ti Ampere GPU which is designed for video gaming applications. Compared to high-end data center A100 GPU, there are two key differences: First, the Tensor Cores in RTX3070Ti are less powerful and offer less peak throughput for all data types. Second, when using FP32 as the data type of matrix C and D, the performance is half of that of using FP16 as the data type. On the other hand, the data type of matrix C/D will not affect the peak performance of the A100 Tensor Cores. The differences suggest a careful check and adjustment is necessary when using Tensor Cores to accelerate gaming or video applications, especially if the original program is developed on a data center A100 GPU, since the difference may lead to undesired performance.

Table 5 presents the Tensor Cores performance on a RTX2080 Turing GPU. The Turing Tensor Cores is the predecessor of the Ampere Tensor Cores, it supports fewer instruction shapes and data types. It is interesting to note that the Dense FMA latency of Ampere Tensor Cores does not improve compared to Turing Tensor Cores. For instance, the latency of mma.m16n8k8 is 17.3 cycles on RTX2080Ti Turing Tensor Cores, which is close to the number 17.7 cycles in A100 Ampere Tensor Cores.

And Turing also has higher clock rate

To summarize, in this Section, we experimented on different mma PTX instructions on Ampere and Turing Tensor Cores. Since Volta Tensor Cores only supports legacy programming interfaces and has been extensively studied by previous work as explained in Section 3, we do not include the experiments of Volta GPU Tensor Cores. Based on our experiments, we indicate how peak performance can be achieved (i.e. i.e. how to choose proper #warps and ILP).





	1	2	4	6	8	12	16	20	24
ILP 1	17.7	17.7	17.7	17.8	17.8	18.2	19.3	21.6	25.1
ILP 2	17.8	17.8	17.8	19.7	19.7	25.8	33.8	42.3	50.5
ILP 3	18.9	18.9	18.9	25.3	25.3	37.2	49.4	61.8	74.2
ILP 4	20.5	20.5	20.5	32.9	32.9	49.2	65.6	82.1	98.5
ILP 5	25.8	25.8	25.8	41.8	41.8	61.8	82.6	103.6	124.2
ILP 6	30.1	30.1	30.1	49.8	49.8	74.4	99.2	124.7	149.2

Fig. 7: Throughput and latency of mma.m16n8k8 instruction under different settings on A100 Tensor Cores.

6 Sparse FMA

Sparse matrix multiplication (SpMM) is an important computation pattern in Deep Learning domains. Plenty of efforts have been invested in optimizing SpMM software on GPUs over the past years [1, 3, 4, 5, 17, 19, 27, 38, 42, 44]. Although significant progress has been made, the performance of SpMM software on GPUs is not satisfied in terms of the hardware throughput bound. On the other hand, due to the irregularity of non-zeros in sparse matrix, it is extremely challenging to adjust GPGPU architecture to accelerate SpMM via hardware approach with acceptable overhead. Recently, fine-grained structured sparsity, especially N:M sparsity (i.e. there are N non-zeros for every M consecutive elements), has shown a promising path to compress Deep Learning Models while maintaining a hardware-friendly sparse pattern [12, 40, 48]. Ampere Tensor Cores are the first general purpose hardware empowered with this kind of fine-grained N:M sparse hardware acceleration.

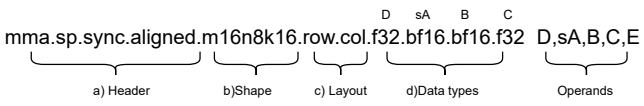


Fig. 8: The mma.sp instruction performs D = sA×B + C sparse-dense matrix multiplication. sA contains the non-zero values of fine-grained 2:4 sparse matrix A (m×k) and the shape of sA, therefore, is m×k/2. Matrix B is still a dense matrix with the shape k×n. The final operand e is the sparse metadata containing the indexes of Matrix A.

Unlike dense Tensor Cores which can be programmed by both modern mma instruction and legacy wmma.mma instruction, the sparse computation can only be exploited by modern mma.sp instruction. Figure 8 shows an example of sparse instruction for BF16 computation. Figure 9 illustrates 2:4 sparse Tensor Cores computation. To use the sparse Tensor Cores correctly, programmers need to first compress the sparse matrix A into the non-zeros matrix (denoted as sA) with the index metadata. The sparse pattern of matrix A should strictly follow the 2:4 fine-grained sparsity which means for every four consecutive elements along the K dimension, there are two non-zero values. Therefore, after compression, the shape of matrix A will become m×k/2. The index metadata encodes the relative location of every non-zero in 2 bits because 2 bits can represent all possibilities of the four consecutive elements (00,01,10,11). On the other hand, matrix B is still stored in the original dense format, the hardware selector determines which values should participate in the computation on the fly based on the indexes provided by metadata.

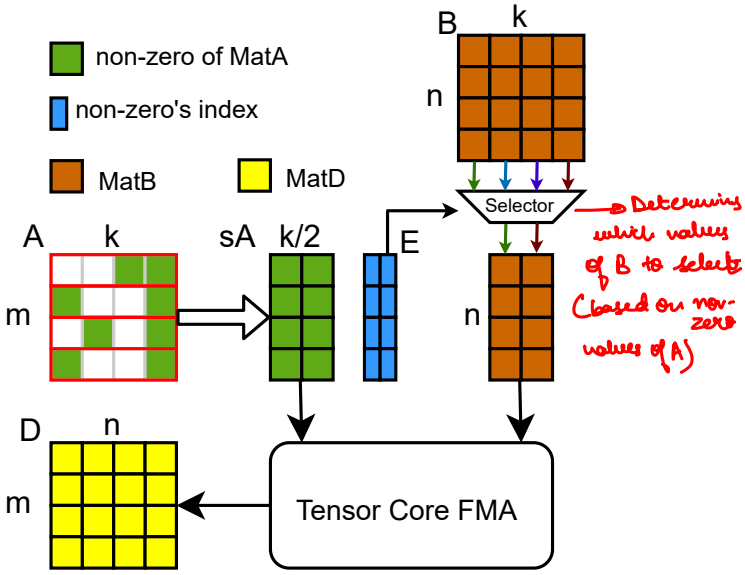
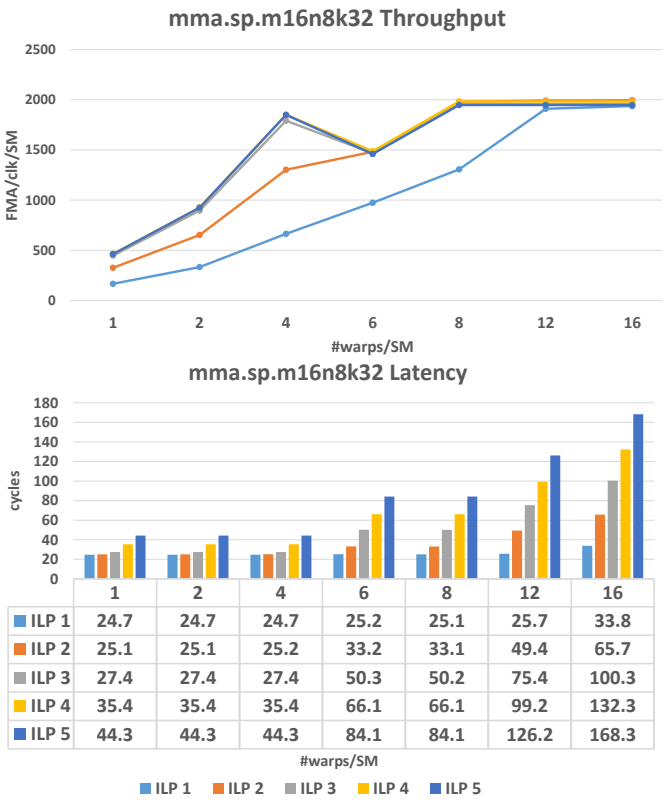


Fig. 9: Illustration of fine-grained 2:4 SpMM [30]. Matrix C is not shown for simplification.

Experimental results

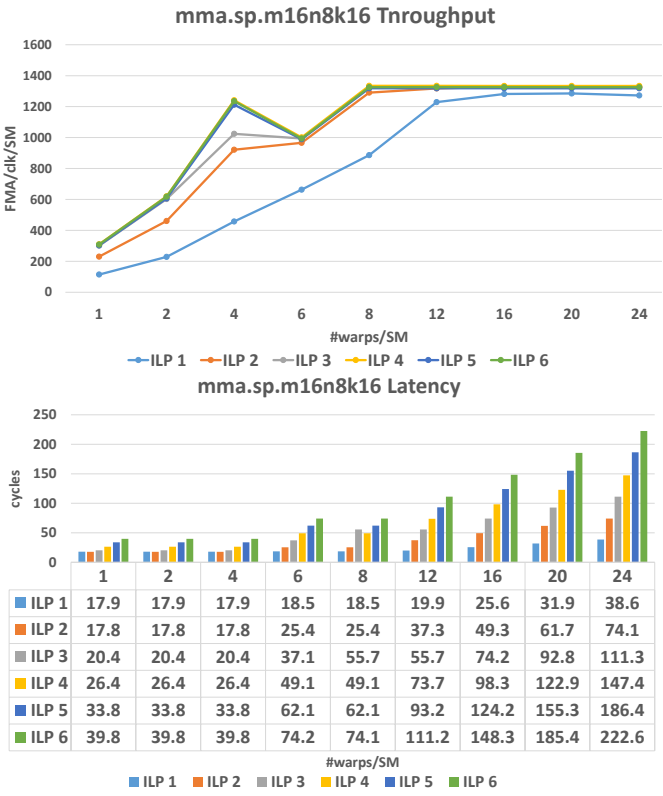
Figure 10 presents the latency and throughput of sparse instruction with shape m16n8k32 instruction. Note there are some interesting similarities between sparse m16n8k32 and dense m16n8k16 (Fig. 6). We provide the following observations based on the experimental results:

1) The completion latency is 24.7 cycles which is the same as its dense mma counterpart (mma.m16n8k16). Furthermore, the latency results under different settings are also almost the same as the results of dense mma.m16n8k16 if we consider the experimental deviations. This observation indicates that the selector for matrix B is integrated with the Tensor Cores hardware pipeline. Although dense FMA operation does not need the selector, it will go through the selector as sparse FMA operation which therefore results in the same execution cycles.



	1	2	4	6	8	12	16
ILP 1	24.7	24.7	24.7	25.2	25.1	25.7	33.8
ILP 2	25.1	25.1	25.2	33.2	33.1	49.4	65.7
ILP 3	27.4	27.4	27.4	50.3	50.2	75.4	100.3
ILP 4	35.4	35.4	35.4	66.1	66.1	99.2	132.3
ILP 5	44.3	44.3	44.3	84.1	84.1	126.2	168.3

Fig. 10: Throughput and latency of mma.sp.m16n8k32 instruction under different settings on A100 Tensor Cores.



	1	2	4	6	8	12	16	20	24
ILP 1	17.9	17.9	17.9	18.5	18.5	19.9	25.6	31.9	38.6
ILP 2	17.8	17.8	17.8	25.4	25.4	37.3	49.3	61.7	74.1
ILP 3	20.4	20.4	20.4	37.1	55.7	55.7	74.2	92.8	111.3
ILP 4	26.4	26.4	26.4	49.1	49.1	73.7	98.3	122.9	147.4
ILP 5	33.8	33.8	33.8	62.1	62.1	93.2	124.2	155.3	186.4
ILP 6	39.8	39.8	39.8	74.2	74.1	111.2	148.3	185.4	222.6

Fig. 11: Throughput and latency of mma.sp.m16n8k16 instruction under different settings on A100 Tensor Cores.

2) The peak throughput achieves at 2000 which is 2x higher than the dense one. The throughput graph shows the same trend as the dense one, the only difference is that the numbers on Y-axis double. This indicates that the improvement comes from skipping the zero multiplication. It confirms that a sparse feature can improve the throughput but can not reduce the latency.

Figure 11 presents the results of mma.sp.m16n8k16. According to the discussions of mma.sp.m16n8k32, the behaviors of mma.sp.m16n8k16 should show similarities with dense mma.m16n8k8. However, we notice that the experimental results are not as expected:

3) The instruction completion latency is 17.9 cycles which are close to the latency of dense m16n8k8 (i.e., 17.7 cycles).
4) The throughput behaviors are not as expected. The peak throughput can only reach 1300. There is a significant gap compared to theoretical peak performance 2000. Furthermore, measured throughput under different settings is also significantly lower than the expected values (i.e. twice the corresponding dense throughput).

Table 6 lists sparse FMA instructions for the rest of the data types and shapes on A100 Tensor Cores. For each data type, there are two supported shapes with different k sizes. Like the behavior of the BF16 data type in Fig. 11, sparse FMA instructions with the smaller k can not reach the theoretical peak performance. To study whether this is a special issue for A100 GPU or a general problem for Ampere Tensor Cores, we check the results on an RTX3070Ti Ampere GPU as presented in Table 7. Interestingly, the sparse Tensor Cores performance of RTX3070Ti does not show the same issue and the instruction with a smaller k can also reach the same throughput as the instruction with a larger k.

To summarize, the sparse Tensor Cores in general can provide twice throughput improvement compared to the dense Tensor Cores but can not reduce the completion latency. However, programmers and users must be careful when deciding which PTX instruction should be used especially when the targeted platform is the data-center A100 GPUs. The instruction with a smaller k can not provide the expected peak throughput. However, the vendor does not document the reason for this issue.

7 Data Movement

In this Section, we study the data movement instructions related to Tensor Cores. As introduced in Section 2, before using Tensor Cores instructions (mma and mma.sp), the input data should be first loaded from Shared Memory to the Register File via data movement instructions. This can be done by special Tensor Cores instructions wmma.load and ldmatrix or generic ld.shared instruction.

Figure 12 illustrates the differences of these three instructions. The generic per-thread ld.shared instruction receives a pointer (p) of an element in Shared Memory and loads it to the destination register (r). For each warp, 32 elements will be loaded from the Shared Memory and each thread holds one register to store the element. On the other hand,

TABLE 6: Sparse Tensor Cores performance of A100 Ampere GPU.

A/B	C/D	Shape	Completion Latency (cycles)	(#warp,ILP)	Latency (cycles)	Throughput (FMA/clk/SM)	(#warp,ILP)	Latency (cycles)	Throughput (FMA/clk/SM)
FP16	FP32	m16n8k32	24.7	4,3	27.4	1791.9	8,2	33.1	1979.1
FP16	FP32	m16n8k16	17.8	4,3	20.4	1024.5	8,2	25.4	1290.5
FP16	FP16	m16n8k32	24.3	4,3	26.6	1850.9	8,2	32.4	2019.8
FP16	FP16	m16n8k16	17.6	4,3	19.8	1242.9	8,2	24.9	1318.2
TF32	FP32	m16n8k16	24.9	4,3	28.3	868.2	8,2	33.9	981.2
TF32	FP32	m16n8k8	18.2	4,3	20.6	597.8	8,2	25.5	643.6
INT8	INT32	m16n8k64	24.7	4,3	27.7	3544.7	8,2	33.1	3961.5
INT8	INT32	m16n8k32	17.9	4,3	20.4	2403.9	8,2	25.4	2665.2

TABLE 7: Sparse Tensor Cores performance of RTX3070Ti Ampere GPU.

A/B	C/D	Shape	Completion Latency (cycles)	(#warp,ILP)	Latency (cycles)	Throughput (FMA/clk/SM)	(#warp,ILP)	Latency (cycles)	Throughput (FMA/clk/SM)
FP16	FP32	m16n8k32	33	4,1	33	496.5	8,1	64.1	511.2
FP16	FP32	m16n8k16	18.8	4,2	32.3	507.8	8,1	32.4	506.2
FP16	FP16	m16n8k32	24.3	4,2	32	1022.2	8,1	32.1	1022.3
FP16	FP16	m16n8k16	17.7	4,3	24.2	1013.4	8,2	32	1023.1
TF32	FP32	m16n8k16	33.2	4,1	33.2	247	8,1	64.2	255.1
TF32	FP32	m16n8k8	19	4,2	32.5	252.5	8,1	32.4	253.2
INT8	INT32	m16n8k64	24.3	4,2	64.2	2040.2	8,1	32.1	2039.5
INT8	INT32	m16n8k32	17.7	4,3	24.2	2028.8	8,2	32.3	2031.8

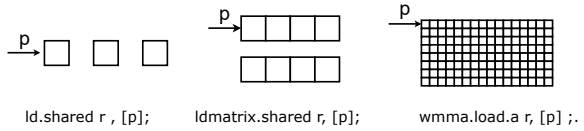


Fig. 12: Differences of three data movement instructions.

wmma.load and ldmatrix run in the per-warp scheme so the 32 threads of a specific warp will cooperatively complete the job. For the legacy wmma.load instruction, there is only one address pointer p to the starting element of the matrix. The elements of the matrix will then be loaded to one or more registers of each thread within the warp depending on the loading shape specified by the instruction. For example, when using wmma.load.a.m16n16k16.FP16, a warp loads matrix A with 16×16 FP16 elements from the Shared Memory. Then each thread will use 16×16/32 = 8 registers to store the elements.

The new ldmatrix instruction is a bit tricky. It runs in a per-warp scheme like wmma.load, but it offers more flexibility. Figure 13 shows the functionality of ldmatrix instruction. It loads N×128 bytes per warp cooperatively from the Shared Memory and stores it in the Register File where each thread stores N×4 bytes. Compared to generic per-thread ld.shared instruction, ldmatrix is less flexible because it requires a group of four consecutive threads to load 16 consecutive bytes (as a group). However, ldmatrix requires fewer source operands (each p will be broadcast to four threads), and the data layout eventually fetched to the Register File matches the arrangement of the inputs to the Tensor Cores mma/mma.sp instructions [37]. Compared to legacy per-warp wmma.load instructions, ldmatrix is more flexible because the 16-bytes groups do not need to be consecutive, while wmma.load requires the whole matrix stored consecutively. By contrast, additional index computations are necessary if using generic ld.shared instruction, which will cause extra overhead. The flexibility of ldmatrix enables using some special data layouts to avoid bank conflict such as permuted data layout used in CUTLASS [33]. Furthermore, ldmatrix can be used to load matrix A, matrix B, and matrix C, while wmma.load has dedicated instructions for different operands as shown in Figure 14. We present ablation experiments in Appendix A.2 to demonstrate how the flexibility of ldmatrix can be exploited to improve performance.

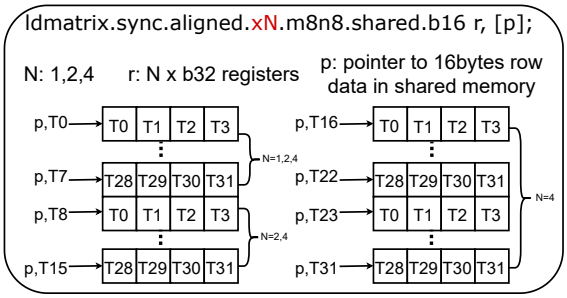


Fig. 13: Functionality overview. p is the source operand and is provided by threads. If N = 4, then every thread will provide a valid p. if N < 4, partial threads (e.g. T0 - T7 for N = 1) will provide a valid p, and other threads' p will be ignored.

```
wmma.load.a.sync.layout.shape.type r, [p] {stride};.
wmma.load.b.sync.layout.shape.type r, [p] {stride};.
wmma.load.c.sync.layout.shape.type r, [p] {stride};.
ldmatrix.sync.aligned.m8n8.num{.trans}.shared.b16 r, [p];
```

Fig. 14: wmma.load and ldmatrix [37]. ldmatrix can be used for loading a,b, and c operands but wmma has dedicated instruction for each of the operands.

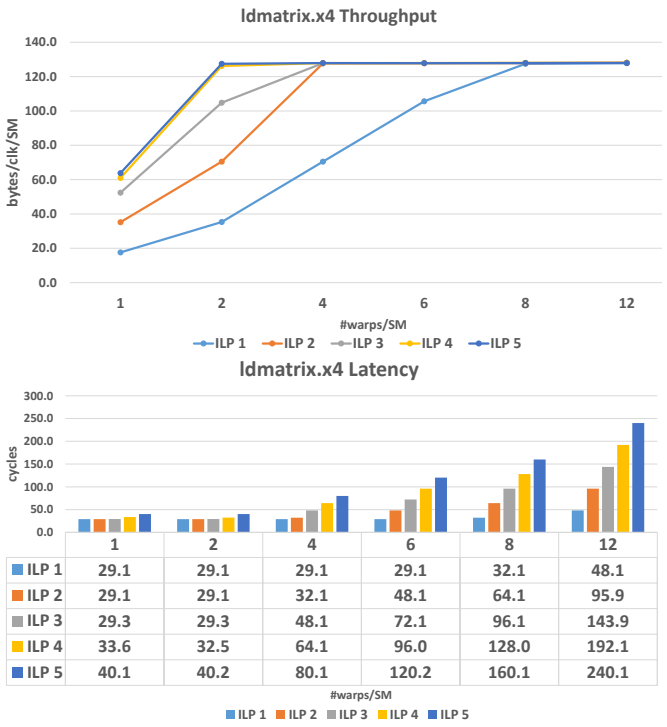
Unlike mma and mma.sp instructions which are designed for the novel Tensor Cores hardware, ldmatrix instruction is an alternative instruction for using the existing hardware (i.e. loading data from the Shared Memory to the Register File). Therefore, the previous study of GPU Shared Memory [25] also applies to the ldmatrix instruction. For instance, the Shared Memory of modern NVIDIA GPUs (e.g. Ampere, Turing, Volta) has 32 banks and the width of each bank is 4 bytes which give 32×4 = 128 bytes/clk as the theoretical bandwidth. This number is also the bandwidth bound of ldmatrix instruction.

TABLE 8: The loading bytes per instruction of ldmatrix and ld.shared.

	bytes/warp	bytes/thread
ldmatrix.x1	128	4
ldmatrix.x2	256	8
ldmatrix.x4	512	16
ld.shared.u32	128	4
ld.shared.u64	256	8

Table 8 lists the instruction workload of ldmatrix and ld.shared. Since traditional ld.shared offers better granularity, we microbenchmark both ldmatrix and ld.shared and make comparisons for better understanding.

Experimental results



	1	2	4	6	8	12
ILP 1	29.1	29.1	29.1	29.1	32.1	48.1
ILP 2	29.1	29.1	32.1	48.1	64.1	95.9
ILP 3	29.3	29.3	48.1	72.1	96.1	143.9
ILP 4	33.6	32.5	64.1	96.0	128.0	192.1
ILP 5	40.1	40.2	80.1	120.2	160.1	240.1

Fig. 15: Throughput and latency of ldmatrix.x4 instruction under different settings on A100 Tensor Cores.

Figure 15 presents the results of ldmatrix.x4 on A100 GPU. Note ldmatrix has three options - ldmatrix.x1, ldmatrix.x2 and ldmatrix.x4 as introduced in Fig. 13. We use ldmatrix.x4 as the example to provide in-depth analyses:

1) The completion latency of ldmatrix.x4 is 29 cycles. The measured peak throughput of ldmatrix.x4 can achieve the peak theoretical performance offered by hardware (128 bytes/clk/SM).
2) If there is only one warp, the peak measured throughput is only around 64 and can be achieved with ILP = 4. To reach the maximum throughput offered by hardware, at least two warps should be used. This indicates there could be two data movement units for ldmatrix.x4 instruction between Shared Memory and Register File.
3) Allocating 6 warps will not cause the same problem observed in the experiments of Tensor Cores computation instructions mma and mma.sp. This also confirms that the data movement units are not located in the sub-core.

Table 9 lists the completion latency and performance of two convergence points of three ldmatrix instructions. If choosing ldmatrix.x1 as the data movement instruction, then at least 8 warps should be launched, while for ldmatrix.x2 and ldmatrix.x4, 4 warps are sufficient.

According to the definition of Shared Memory bank conflict, Shared Memory can serve at most 32 threads with a 4 byte/thread access request, which gives 128 byte/clk/warp. Therefore, as shown in Table 8, ldmatrix.x2 and ldmatrix.x4 should intrinsically cause 2-way and 4-way bank conflicts respectively (i.e. two/four memory transactions are needed). However, the NVIDIA official profiler [36] does not detect Shared Memory conflicts for both ldmatrix.x2 and ldmatrix.x4 instructions. We further profile the effect of bank conflicts for ld.shared instruction as listed in Table 10. By comparing results to the completion latency of ldmatrix in Table 9, we have the following observations:

1) The completion latency of ldmatrix.x1, ldmatrix.x2, and ldmatrix.x4 are close to the latency of ld.shared.u32 with bank conflict-free, two-way, and four-way bank conflicts respectively. This observation indicates that ldmatrix.x2 and ldmatrix.x4 intrinsically causes bank conflicts and requires more Shared Memory transactions to sequentially serve the memory access.
2) The bank conflict penalty for modern GPUs is around 2 cycles/way, which matches the previous study on older GPUs like Maxwell Architecture [25].

In summary, ldmatrix instructions perform the function of loading data from the Shared Memory to the Register File for Tensor Cores. ldmatrix and ld.shared instructions have similar intrinsic behaviors and performance. The major difference is the layout of destination registers which store the fetched data from Shared Memory. ldmatrix is designed for Tensor Cores computations while ld.shared is preferable for CUDA cores programming.

8 Numeric Behaviors

Mixed-precision FMA and low-precision float point data types are becoming increasingly important for Deep Learning applications and some HPC applications. The third generation Ampere Tensor Cores enable tensor computations with three low-precision float point data types: TF32 (19bit), BF16 (16bit), and FP16 (16bit). Table 11 shows different precision formats. Compared to IEEE standard FP32, TF32

TABLE 9: Performance of three ldmatrix instructions on A100 GPU.

	Bytes/warp	Completion Latency (cycles)	(#warp,ILP)	Latency (cycles)	Throughput (bytes/clk/SM)	(#warp,ILP)	Latency (cycles)	Throughput (bytes/clk/SM)
ldmatrix.x1	128	23.1	4,5	26.8	95.4	8,4	32.1	127.7
ldmatrix.x2	256	25.1	4,4	32.1	127.8	8,2	32.1	127.7
ldmatrix.x4	512	29.3	4,2	32.2	127.3	8,1	32.6	125.9

TABLE 10: Latency of ld.shared instructions under different bank conflicts.

	no-conflict	2-way	4-way	8-way
ld.shared.u32	23.0	25.0	29.0	37.0
ld.shared.u64	NA	25.1	29.1	37.0

and BF16 have the same 8bit exponent but fewer mantissa bits, so they have the same range as FP32 but lower accuracy (fewer mantissa bits). Note that although TF32 only has 19 bits (1+8+10), it stores in the 32-bit register in GPU, which means using TF32 to replace FP32 will not lead to a reduced memory footprint.

Note that we exclude the discussions and experimental results of Integer data type of Tensor Cores, since our experiments show that Integer computations on Tensor Core give 0 errors compared to CPU implementation (i.e. The results of GPU and CPU baseline are the same) as long as the initialization values are within the data type range (e.g. for INT8, the input data should be initialized within [-128,127]). If the initialization data is out of range, then type casting has to be performed by either a user-defined casting function or C++ default type casting. As long as the casting is the same for the GPU (i.e. Tensor Cores) and CPU baseline, the numeric results are still the same.

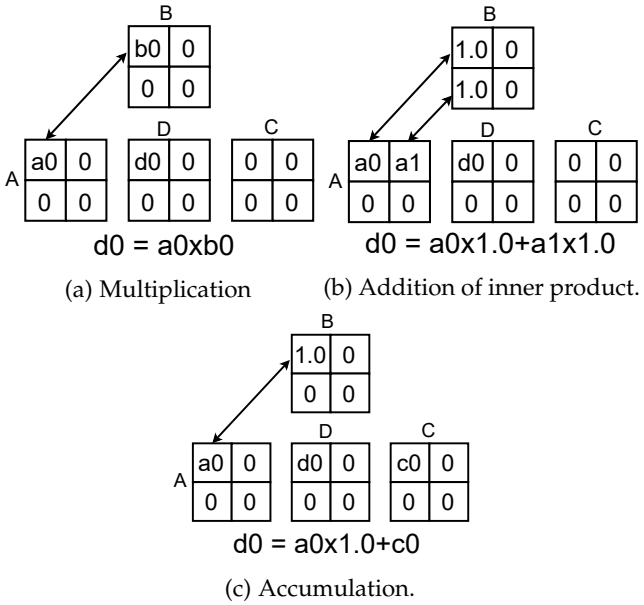
TABLE 11: Different precision formats and storage in GPUs. Note that TF32 has 19bits (1+8+10), but it is stored in the 32-bit registers. Two FP8 types will be supported by the forthcoming Hopper Tensor Cores.

data types	sign	exponent	mantissa	register
FP32	1	8	23	32b
TF32	1	8	10	32b
FP16/half	1	5	10	16b
BF16	1	8	7	16b
FP8-E4M3	1	4	3	8b
FP8-E5M2	1	5	2	8b

Although low-precision computation offers significant performance gains as discussed in Section 5 and Section 6. Low precision will cause numeric errors compared to FP32 precision. When using low-precision computation to accelerate applications, it is important to understand the numeric behaviors of Tensor Cores. In this Section, we present a set of numeric benchmarks to reveal the properties of low-precision float points operations of Tensor Cores.

8.1 Element-wise numeric profiling

Tensor Cores instruction performs D = A×B + C, which consists of three types of operation: 1) multiplication, 2) addition of inner product, and 3) accumulation. To understand the numeric behavior at the lowest possible level, we use three experiments to profile the three operations separately.



(a) Multiplication (b) Addition of inner product.

(c) Accumulation.

Fig. 16: Profiling of three operations.

Figure 16a, 16b and 16c illustrate the ideas of profiling multiplication, addition of inner product, and accumulation, respectively. When profiling multiplication numeric precision, we randomly generate the first element in the first row of matrix A and the first element in the first column of matrix B and set other values as 0.0. Then the matrix FMA D = A×B+C becomes d0 = a0×b0 as shown in Fig. 16a. By comparing the result to the FP32 result on CPU as the baseline, we can get the numeric errors of the low-precision floating-point data types used in Tensor Cores. Following the same ideas, we can further profile the addition operation of inner product and addition operation of accumulation. Note that we use a random generator of the normal distribution with $\mu = 0.0$ and $\sigma = 1.0$, and use the same random seed for all experiments so the sequences of random values are the same. This setting helps to compare the precision between different numeric types since they will get the same random initialization values.

8.1.1 BF16

For BF16 Tensor Core operations, the data type for A/B and C/D on Tensor Cores are BF16 and FP32 respectively. We have two initialization strategies, the first is to initialize the data with FP32 precision and convert it to BF16 when copying to GPU. However, this will introduce conversion precision loss. The second approach is to initialize the data with FP16 precision to eliminate the conversion precision loss. After eliminating the conversion loss, we can discover the computation precision used in Tensor Cores by compar-

ing it to the result of FP32 computation precision on CPU [7].

TABLE 12: The numeric profiling results of BF16 on Tensor Cores w.r.t FP32 on CPU.

	init_BF16	init_FP32
multiplication	0.0	1.29E-03
add - Inner product	0.0	1.72E-03
accumulation	1.89E-08	1.13E-03

Table 12 shows the average errors under different initialization types. When initializing the data with FP32, all operations have numeric errors. By contrast, when choosing BF16 as the initialization type, multiplication and inner product addition will not cause any numeric errors. This observation indicates that the computation of A×B inside the Tensor Cores is using high-precision [6] otherwise it can not give the same result as FP32 computation on the CPU. On the other hand, accumulation error indicates that the accumulation of [A×B] + C is performed with relative low precision.

8.1.2 FP16

Half/FP16 Tensor Cores operation offers two data type choices - FP32 and FP16 for accumulation matrix C and result matrix D. Instead of just comparing to the FP32 result of CPU, we also compare to the CPU FP16 result converted from CPU FP32 result when using FP16 as the data type for matrix C and matrix D.

TABLE 13: The numeric profiling results of FP16 on Tensor Cores w.r.t FP32 on CPU. Matrix C and D are in FP32 data type.

FP32 as C/D	init_FP16	init_FP32
multiplication	0.00	1.59E-04
add - Inner Product	0.00	2.18E-04
accumulation	0.00	1.36E-04

TABLE 14: The numeric profiling results of FP16 on Tensor Cores w.r.t FP32 on CPU. Matrix C and D are in FP16 data type.

	CPU_FP32		CPU_FP32cvtFP16	
FP16 as C/D	init_FP16	init_FP32	init_FP16	init_FP32
multiplication	1.22E-04	1.94E-04	0.00	1.67E-04
add - inner product	1.81E-04	2.99E-04	0.00	2.21E-04
accumulation	1.81E-04	2.99E-04	0.00	2.21E-04

Table 13 shows the profiling results of using FP32 as the data type of matrices C and D. When initializing with FP16, there are no numeric errors for all three operations. When initializing with FP32, numeric errors occur due to type conversion like the observations in BF16 profiling. However, the error level(E-04) of FP16 is lower than the error level of BF16 (E-03) since FP16 has more mantissa bits and is more accurate as long as the values are within the FP16 range.

Table 14 shows the profiling results of using FP16 as the data type of matrix C and D. Since the result matrix D is in FP16, there are always numeric errors when compared to the CPU FP32 baseline. Interestingly, if we compare the CPU FP16 results, the errors are zero when using FP16 as the initialization data type. This observation suggests that even though the matrix D is in FP16, the hardware conducts the computation in high-precision and only converts the final result to low-precision FP16 in the end.

8.1.3 TF32

TABLE 15: The numeric profiling results of TF32 on Tensor Cores w.r.t FP32 on CPU.

	init_TF32	init_FP32
multiplication	0.0	1.59E-04
add - Inner Product	0.0	2.17E-04
accumulation	0.0	1.36E-04

The profiling code for TF32 Tensor Cores is similar to the one for the BF16 data type. Table 15 presents the results of TF32 Tensor Cores operations. Note that if we compare FP16 results in Table 13, we can find that TF32 and FP16 give the same level of error because they have the same number of mantissa bits (10 bits).

8.2 Chain Matrix Multiplication

We use a simple application - chain matrix multiplication, to evaluate the effect of low-precision FMA. Chain matrix multiplication is a simplified computation pattern of modern deep learning applications which consists of many layers and the results of the previous layer will be the input of the next layer. In this computation pattern, both range (exponent bits) and precision (mantissa bits) are important and will affect the final accuracy. Fewer mantissa bits will result in larger accumulative errors for the final result of the chain. Fewer exponent bits give a smaller valid range and will result in overflow (infinity) earlier.

The initialization data type can be either FP32 or low-precision type (BF16/FP16). For each node of the chain, we first compute D = A×B. The result matrix D will be assigned to matrix A for the next computation, and matrix B will be assigned new random values. Unlike the element-wise profiling in Section 8.1 where we only evaluate the errors of one element, we evaluate the numeric errors of the whole matrix D with shape m×n. Based on our study in Section 5, we choose mma instructions with shape m16b8k8 since this common shape is supported by BF16, FP16, and TF32. After completion the chain computations, we evaluate the l2 relative errors between CPU results D^{FP32} and Tensor Cores low-precision results D^{l} with equation 1.

$$RelativeErr = \frac{\sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|D_{ij}^{l} - D_{ij}^{FP32}|^2}}{\sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|D_{ij}^{l}|^2}} \quad (1)$$

Figure 17 shows the results of three data types of chain matrix multiplication with different chain lengths (N). We have the following observations:

1) In general the errors increase with the length of the chain. This increase is more significant for the BF16 data type compared to TF32 and FP16, this is because BF16 has fewer mantissa bits and will suffer more from accumulative errors.
2) The results of FP16 give the same error levels as using TF32 because FP16 and TF32 have the same

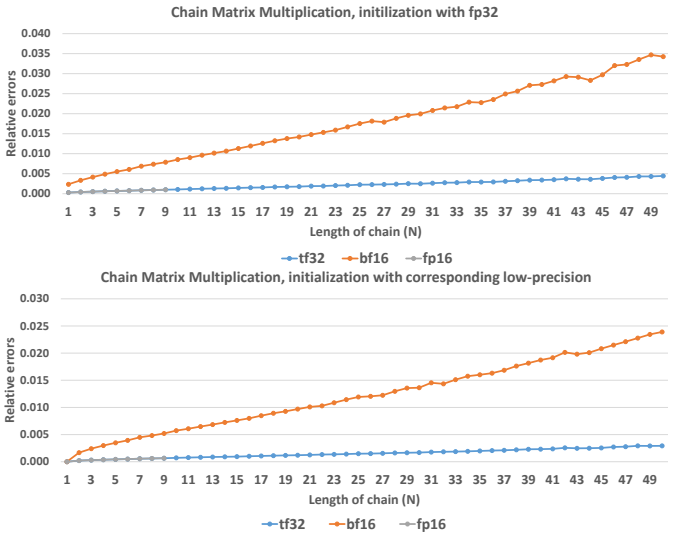


Fig. 17: Numeric profiling of chain matrix multiplication with different data types -TF32, BF16, and FP16. All values are randomly generated by normal distribution with $\mu = 0$ and $\sigma = 1$. The errors are taken on average with 1000 measurements. Note that the line of FP16 stops at N = 10 due to overflow (infinity).

number of mantissa bits which gives the same level of precision, but it will run into overflow (infinity) after N $\geq$ 10 since it has fewer exponent bits which result in a less range of numbers can be represented.

3) Using FP32 as an initialization strategy always causes higher errors since it introduces type conversion precision loss. When using a low-precision type for initialization, the error is almost zero when chain length is one because the internal computations are conducted in high-precision as discovered in Section 8.1 and there is no conversion precision loss.

In summary, FP16 offers the same level of precision as TF32 as long as the numbers are within the valid range that can be represented by FP16. BF16 has the same valid range as TF32 but it suffers more seriously from accumulative precision loss. The users should consider both numeric behaviors and the performance benefits studied in Section 5 and 6 when choosing the target data type for accelerating their applications on Tensor Cores.

9 Conclusion

We dissect NVIDIA Tensor Cores, especially the newest Ampere Tensor Cores through a set of microbenchmarks and numeric profiling experiments. We summarize our main findings as follows:

- When programming Ampere Tensor Cores. Using the new ldmatrix + mma instructions is favorable compared to the legacy wmma.load + wmma.mma instructions. The ablation experiments in the Appendix A demonstrate the advantages of using the new instructions. Specifically, by levering the flexibility of the new ldmatrix instruction, it can reduce the number of corresponding GPU clock cycles by more than 60%.
- Sparse acceleration introduced in Ampere Tensor Cores can only be programmed through mma APIs. Sparse operation doubles the throughput by accepting larger input matrices than dense ones while using the same number of execution cycles.
- It is well known that four warps should be allocated on a GPU SM since there are four warp schedulers [30, 39]. Our experiments show that for some instructions (e.g. Fig. 7), peak performance can only be achieved when there are at least eight warps .
- For each data type, there are two mma.sp instructions with different k sizes. In general, the instructions with larger k can achieve the expected performance on both data-center A100 and Gaming RTX3070Ti Tensor Cores. However, the instructions with smaller k give an undesired performance on A100 Tensor Cores (i.e. can not achieve peak throughput).
- The performance of different GPUs with the same Architecture generation may not be consistent. For instance, RTX3070Ti Tensor Cores favor FP16 as an accumulation data type from a performance perspective, but there is no difference no matter whether using FP16 or FP32 on A100 Tensor Cores.
- The performance of BF16 and FP16 instructions on Tensor Cores are the same, which confirms the observations in previous work [6]. On the other hand, our experiments intuitively show that FP16 suffers from a smaller range and BF16 suffers from higher numeric errors. For machine learning applications that have a relatively high tolerance for numeric precision loss, BF16 is favorable since it has the same range as F32 and therefore the same overflow behavior.

In conclusion, our work provides comprehensive and up-to-date information on the Tensor Cores, especially the instruction-level evaluations. To the best of our knowledge, this paper is the first systematic study on recent Tensor Cores generations (Turing and Ampere) using the new programming interface. We believe our contributions can help users to implement their own Tensor Cores applications and facilitate accurate Tensor Cores modeling.

Acknowledgments

This work is funded by the Dutch Research Council (NWO) Perspectief Program ZERO-ARM P3. This work is also supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, ComPort: Rigorous Testing Methods to Safeguard Software Porting, under Award Number 78284. The Pacific Northwest National Laboratory is operated by Battelle for the U.S. Department of Energy under Contract DE-AC05-76RL01830.

References

[1] P. N. Q. Anh, R. Fan, and Y. Wen, "Balanced hashing and efficient gpu sparse general matrix-matrix multiplication," in *Proceedings of the 2016 International Conference on Supercomputing*, 2016, pp. 1–12.

[2] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *2009 IEEE international symposium on performance analysis of systems and software*, 2009.

[3] Y. Chen, K. Li, W. Yang, G. Xiao, X. Xie, and T. Li, "Performance-aware model for sparse matrix-matrix multiplication on the sunway taihulight supercomputer," *IEEE transactions on parallel and distributed systems*, 2018.

[4] S. Dalton, L. Olson, and N. Bell, "Optimizing sparse matrix—matrix multiplication for the gpu," *ACM Transactions on Mathematical Software (TOMS)*, 2015.

[5] M. Deveci, C. Trott, and S. Rajamanickam, "Multithreaded sparse matrix-matrix multiplication for many-core and gpu architectures," *Parallel Computing*, 2018.

[6] M. Fasi, N. J. Higham, M. Mikaitis, and S. Pranesh, "Numerical behavior of nvidia tensor cores," *PeerJ Computer Science*, 2021.

[7] B. Feng, Y. Wang, G. Chen, W. Zhang, Y. Xie, and Y. Ding, "Egemm-tc: Accelerating scientific computing on tensor cores with extended precision," ser. PPoPP '21, 2021.

[8] B. Feng, Y. Wang, T. Geng, A. Li, and Y. Ding, "Apnn-tc: Accelerating arbitrary precision neural networks on ampere gpu tensor cores," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021.

[9] B. Gaide, D. Gaitonde, C. Ravishankar, and T. Bauer, "Xilinx adaptive compute acceleration platform: Versaltm architecture," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2019.

[10] A. Haidar, S. Tomov, J. Dongarra, and N. J. Higham, "Harnessing gpu tensor cores for fast fp16 arithmetic to speed up mixed-precision iterative refinement solvers," in *SC18*, 2018.

[11] B. Hickmann, J. Chen, M. Rotzin, A. Yang, M. Urbanski, and S. Avancha, "Intel nervana neural network processor-t (nnp-t) fused floating point many-term dot product," in *2020 IEEE 27th Symposium on Computer Arithmetic (ARITH)*, 2020.

[12] I. Hubara, B. Chmiel, M. Island, R. Banner, J. Naor, and D. Soudry, "Accelerated sparse neural training: A provable and efficient method to find n: m transposable masks," *Advances in Neural Information Processing Systems*, vol. 34, 2021.

[13] Z. Jia, M. Maggioni, J. Smith, and D. P. Scarpazza, "Dissecting the nvidia turing t4 gpu via microbenchmarking," *arXiv preprint arXiv:1903.07486*, 2019.

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

[15] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," ser. ISCA '17, 2017.

[16] D. B. Kirk and W. H. Wen-Mei, *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.

[17] R. Kunchum, A. Chaudhry, A. Sukumaran-Rajam, Q. Niu, I. Nisa, and P. Sadayappan, "On improving performance of sparse matrix-matrix multiplication on gpus," in *Proceedings of the International Conference on Supercomputing*, 2017, pp. 1–11.

[18] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016.

[19] J. Lee, S. Kang, Y. Yu, Y.-Y. Jo, S.-W. Kim, and Y. Park, "Optimization of gpu-based sparse matrix multiplication for large sparse networks," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, 2020.

[20] A. Li, S. L. Song, J. Chen, J. Li, X. Liu, N. R. Tallent, and K. J. Barker, "Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect," *IEEE Transactions on Parallel and Distributed Systems*, 2019.

[21] A. Li and S. Su, "Accelerating binarized neural networks via bit-tensor-cores in turing gpus," *IEEE Transactions on Parallel and Distributed Systems*, 2020.

[22] J. Liu, D. Yang, and J. Lai, "Optimizing winograd-based convolution with tensor cores," in *50th International Conference on Parallel Processing*, 2021, pp. 1–10.

[23] S. Markidis, S. W. Der Chien, E. Laure, I. B. Peng, and J. S. Vetter, "Nvidia tensor core programmability, performance & precision," in *IPDPSW*, 2018.

[24] M. Martineau, P. Atkinson, and S. McIntosh-Smith, "Benchmarking the nvidia v100 gpu and tensor cores," in *European Conference on Parallel Processing*, 2018.

[25] X. Mei and X. Chu, "Dissecting gpu memory hierarchy through microbenchmarking," *IEEE Transactions on Parallel and Distributed Systems*, 2017.

[26] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh *et al.*, "Mixed precision training," *arXiv preprint arXiv:1710.03740*, 2017.

[27] Y. Niu, Z. Lu, H. Ji, S. Song, Z. Jin, and W. Liu, "Tilespgemm: a tiled algorithm for parallel sparse general matrix-matrix multiplication on gpus," in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2022, pp. 90–106.

[28] Nvidia, "Nvidia volta architecture white paper," 2017. [Online]. Available: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[29] ——, "Nvidia turing architecture white paper," 2018. [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

[30] ——, "Nvidia ampere architecture white paper," 2020. [Online]. Available: https://resources.nvidia.com/en-us-genomics-ep/ampere-architecture-white-paper

[31] ——, "Nvidia ampere ga102 gpu architecture," https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf, 2021.

[32] ——, "cublas," 2022. [Online]. Available: https://docs.nvidia.com/cuda/cublas/index.html

[33] ——, "Cutlass," 2022. [Online]. Available: https://github.com/NVIDIA/cutlass

[34] ——, "Developing cuda kernels to push tensor cores to the absolute limit on nvidia a100," https://www.nvidia.com/en-us/on-demand/session/gtcsj20-s21745/, 2022.

[35] ——, "Nvidia hopper architecture white paper," 2022. [Online]. Available: https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper

[36] ——, "Nvidia nsight," 2022. [Online]. Available: https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html

[37] ——, "Ptx isa," 2022. [Online]. Available: https://docs.nvidia.com/cuda/parallel-thread-execution/index.html

[38] M. Parger, M. Winter, D. Mlakar, and M. Steinberger, "speck: accelerating gpu sparse matrix-matrix multiplication through lightweight analysis," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2020.

[39] M. A. Raihan, N. Goli, and T. M. Aamodt, "Modeling deep learning accelerator enabled gpus," in *ISPASS*, 2019.

[40] W. Sun, A. Zhou, S. Stuijk, R. Wijnhoven, A. O. Nelson, H. Corporaal *et al.*, "Dominosearch: Find layer-wise fine-grained n: M sparse schemes from dense neural networks," *Advances in Neural Information Processing Systems*, vol. 34, 2021.

[41] Y. Wang, B. Feng, and Y. Ding, "Tc-gnn: Accelerating sparse graph neural network computation via dense tensor core on gpus," *arXiv preprint arXiv:2112.02052*, 2021.

[42] M. Winter, D. Mlakar, R. Zayer, H.-P. Seidel, and M. Steinberger, "Adaptive sparse matrix-matrix multiplication on the gpu," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, 2019.

[43] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying gpu microarchitecture through microbenchmarking," in *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, 2010.

[44] Z. Xie, G. Tan, W. Liu, and N. Sun, "A pattern-based spgemm library for multi-core and many-core architectures," *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[45] D. Yan, W. Wang, and X. Chu, "Demystifying tensor cores to optimize half-precision matrix multiply," in *IPDPS*, 2020.

[46] ——, "Optimizing batched winograd convolution on gpus," in *25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '20)*, 2020.

[47] X. Zhang, G. Tan, S. Xue, J. Li, K. Zhou, and M. Chen, "Understanding the gpu microarchitecture to achieve bare-metal perfor-

mance tuning," in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2017.
[48] A. Zhou, Y. Ma, J. Zhu, J. Liu, Z. Zhang, K. Yuan, W. Sun, and H. Li, "Learning n: m fine-grained structured sparse neural networks from scratch," *arXiv preprint arXiv:2102.04010*, 2021.

APPENDIX A
ABLATION EXPERIMENTS

In this section, we use three matrix multiplication CUDA kernels to demonstrate the advantages of using the new instructions studied in this paper.

A.1 Benefits of shared memory usage

In section 7, we excluded the experiments for data movement from global memory to shared memory and assume the data are ready in shared memory for ldmatrix instructions which can only fetch data from shared memory to registers. We explained in section 2 with two reasons:

1 Using shared memory as the buffer to reduce global memory traffic and increase data reuse.
2 The novel asynchronous memory copy introduced in Ampere Architecture facilitates the software data pipeline by using shared memory as the staging storage to overlap the data movement with computation.

The first point - using shared memory to increase data reuse has been widely used and been a fundamental optimization technique. A detailed study of this technique can be found in textbook [16] so we exclude further discussions here and emphasize the second point - new asynchronous memory copy.

Asynchronous memory copy acceleration was introduced in Ampere Architecture [30]. It allows asynchronous data movement from off-chip global memory to on-chip shared memory. Compared to the old synchronous copy fashion, asynchronous copy can be leveraged to hide the data copy latency by overlapping it with computation. To demonstrate the advantages of asynchronous copy, we implemented two matrix multiplication CUDA kernels.

- mma_baseline.cu
  a Copy the data tile (among inner dimension K) from global memory to shared memory.
  b Synchronization. Wait until all copy ready to avoid data hazards.
  c Copy data from shared memory to register with ldmatrix.
  d Use Tensor Core (mma) to do the computations.
  e Repeat a-d until the end of the K dimension.
- mma_pipeline.cu
  a Copy the first data tile from global memory to shared memory with the asynchronous copy.
  b Copy the next data tile from global memory to shared memory with the asynchronous copy.
  c Wait until the first tile copy is complete.
  d Copy data from shared memory to register with ldmatrix.
  e Use Tensor Core (mma) to do the computation.
  f Repeat a-e until the end of the K dimension.

We profile the number of cycles (using clock64() CUDA API) of the above two implementations with large matrix multiplication (2048 x 2048 with bf16 data type).

TABLE 16: Compare baseline with asynchronous copy pipeline. Tested on an Ampere A100 GPU.

Implementation	Number of GPU cycles
mma_baseline.cu	913363
mma_pipeline.cu	451560

The results demonstrate the advantages of the asynchronous copy pipeline. Therefore we can conclude that shared memory should be used for the best possible performance.

A.2 Flexibility of ldmatrix instruction

As introduced in Figure 12 of section 7. ldmatrix instruction has more flexibility than the legacy wmma.load instruction, which helps reduce bank conflicts by using some special shared data layout. For instance, CUTLASS introduced permuted shared memory layout [34] to reduce bank conflicts. We followed the ideas and re-implemented our third CUDA kernel mma_permuted.cu by adjusting the mma_baseline.cu implementation. Table 17 shows the profiling results.

TABLE 17: Compare baseline with permuted shared memory layout. Tested on an Ampere A100 GPU.

Implementation	Number of GPU cycles
mma_baseline.cu	913363
mma_permuted.cu	303227

The results show the advantages of leveraging the flexibility of ldmatrix to reduce bank conflicts, which can therefore reduce the number of cycles significantly.

Note that ldmatrix should always be used with mma instructions since ldmatrix will store the fetched data in agreement with the requirements of mma instructions as specified by vendor's documentations [37]. In another word, the correct execution chain should be either ldmatrix + mma or the legacy wmma.load + wmma.mma.