SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Targeting High-Performance Applications with NVIDIA Tensor Cores: Precision and Performance Insights

Rafael Piñán García

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Targeting High-Performance Applications with NVIDIA Tensor Cores: Precision and Performance Insights

Author:	Rafael Piñán García
Supervisor:	Prof. Dr. Michael Bader
Advisor:	Mario Wille, M.Sc.
Submission Date:	15.08.2024

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 15.08.2024 Rafael Piñán García

Acknowledgments

I want to thank Prof. Dr. Michael Bader for allowing me to contribute to the exciting research he and his colleagues conduct at the Leibniz Supercomputing Center. I am incredibly grateful to my advisor, M.Sc. Mario Wille. His motivation and commitment are inspiring; one can quickly get enlightened by his passion for science and research, as I did throughout the development of this thesis. I hope this work is helpful for their research.

Abstract

This work gives a complete overview of performing dense matrix multiplication and accumulating floating-point operations on NVIDIA Tensor Cores. In 2017, NVIDIA unveiled the first generation of Tensor Cores as part of the Volta architecture. Nowadays, Tensor Cores are an essential part of the computation hardware of data centers worldwide. As NVIDIA GPUs and these computation units developed, the possibilities expanded. This work reviews the current capabilities of Tensor Cores and how to leverage their performance. Tensor Cores are well-established in numerous applications that are not sensible to precision losses. This work proves that Tensor Cores are not limited to those applications and should be exploited for any application that performs dense matrix multiplication and accumulating floating-point operations. An API for carrying out Tensor Cores operations has been implemented as part of this work. The API shows real Tensor Cores programmability using two different approaches. A benchmark proving Tensor Core's performance and precision has been developed as part of this work as well.

Contents

1 Introduction and Related Work

Unveiling the Volta architecture in 2017, NVIDIA GPUs first incorporated a new specialized compute unit called Tensor Cores [15].

Tensor Cores, a unique hardware innovation designed to meet the demands of emerging data-intensive and deep-learning applications, are instrumental in facilitating dense matrix-matrix multiplication. This operation, that forms the foundation of learning and inference tasks in multi-layered neural networks, is where Tensor Cores truly shine [16].

Tensor Cores introduced reduced precision, which consists of floating point representations with less than the 32 bits traditionally used for single-precision representation.

This innovation significantly reduces the storage and computing resources required, making Tensor Cores a game-changer in the field of deep learning.

Many different deep learning models can be trained using Tensor Cores' reduced precision with no loss in accuracy, reducing their computation times notoriously. Research conducted in the literature achieved speed-ups from 2 to 6 times [7].

HPC applications relying on dense matrix-matrix multiplication could, in a first impression, not benefit from Tensor Cores. Precision is a critical aspect of HPC.

However, since the launch of Tensor Cores, researchers have conducted studies to leverage this module for high-performance applications. Previous work successfully incorporated emulation algorithms, tensorization, and instruction-level optimizations on Tensor Cores. These enhancements enabled Tensor Cores use in Scientific Computing with bearable precision loss [2]. The study achieved speedups over non-Tensor kernels ranging from 3 to 11 times in operation throughput and 1.3 to 1.8 times in specific applications.

Work on a comprehensive overview of Tensor Cores' programmability, performance, and precision was conducted previously [5]. The study reviews the available `APIs` and Tensor Cores targeting modes. The study benchmarks Tensor Cores, which deliver single-precision through reduced precision operations and precision refinement. The speedup reached was 5 times. The precision loss was also covered, achieving limited loss on matrices with large dimensions thanks to the refinement techniques.

The previously mentioned studies use CUDA 9 and first-generation Tensor Cores. Since NVIDIA revealed the first architecture implementing Tensor Cores, significant breakthroughs have occurred in AI and analytics, such as large language models or computer vision. As these types of applications evolve, the operation throughput they require also increases. New-generation Tensor Cores reduce energy requirements to meet rising costs and environmental compromises.

This thesis analyzes all floating point operations in NVIDIA Tensor Cores last generation and discusses the benefits of NVIDIA Tensor Cores vs. NVIDIA CUDA Cores. The evaluation is conducted using CUDA 12. Performance and accuracy are the features involved in the comparison.

Proving that last-generation Tensor Cores is the best hardware accelerator for high-performance applications is the motivation of this thesis.

2 Tensor Cores

2.1 NVIDIA Hardware Implementation

Streaming Multiprocessors (SMs) provide the base of the NVIDIA GPUs architecture. Multiprocessors are designed to execute hundreds of threads concurrently. To manage this vast number of threads, it employs the SIMT (Single-Instruction, Multiple-Thread) architecture [9].

In the SMIT architecture, the multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps.

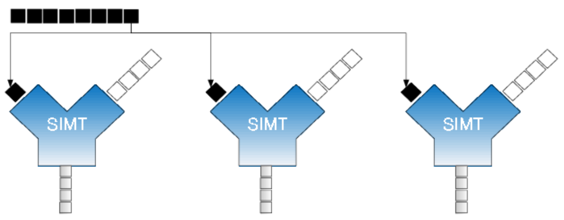

Figure 2.1: SIMT architecture [4].

In the figure 2.1, it can be observed that the threads receiving the same instruction (black) and different data (white) produce different results (grey).

2.2 Operation Description

Tensor Cores perform the operation:

$$\boldsymbol{D} = \alpha \boldsymbol{A}\boldsymbol{B} + \beta \boldsymbol{C}$$

In-place operation $\boldsymbol{C} = \boldsymbol{D}$ is supported as well.

The BLAS convention [1] describes the operation performed by Tensor Cores:

- The first character in the name denotes the matrix's data type.

- The second and third characters refer to the kind of matrix involved. Tensor Cores operate with matrices that are general rectangular, corresponding to letters GE.
- The characters in positions four and fifth denote the type of operation. Tensor Cores perform matrix-matrix product, corresponding to letters MM. The MM operation consists of the multiplication of two matrices A and B and the addition of a third one C.

Therefore, Tensor Cores perform BLAS GEMM operations (omitting the data type character).

2.3 Generations

Tensor Cores generations:

- First-generation Tensor Cores are integrated into the Volta architecture. Each SM contains eight mixed-precision matrix arithmetic Tensor Cores supporting half-precision.

  First-generation Tensor Cores operate with the input matrix elements represented in half-precision and the output matrix elements represented in half-precision or single-precision.
- Second-generation Tensor Cores are integrated into the Turing architecture [14]. Each SM contains eight mixed-precision matrix arithmetic Tensor Cores that support half-precision and new INT8 and INT4 precision modes.
- Third-generation Tensor Cores are integrated into the Ampere architecture. Each SM contains four mixed-precision Tensor Cores supporting half, alternate floating point, sub-byte, and double-precision matrix arithmetic.
- Fourth-generation Tensor Cores are integrated into the Ada Lovelace architecture. This unit supports the same configuration as the third generation, adding FP8.

2.4 Data Types

All NVIDIA compute devices follow the IEEE 754-2008 standard [3] for binary floating-point arithmetic with some minor deviations.

Throughput is directly related to precision. Precision increases at the expense of operation throughput. This trade-off is a core feature in the Tensor Cores architecture. This work only covers floating-point operations, although Tensor Cores also supports sub-byte operations. The data types supported are 4-bit, signed and unsigned GEMM (Integer GEMM), and 1-bit GEMM (Boolean GEMM).

Thus, this 1-bit operation is the most performant option regarding operations completed per time unit. In this option, 32 matrix elements are packed in 1 storage element. Increasing the

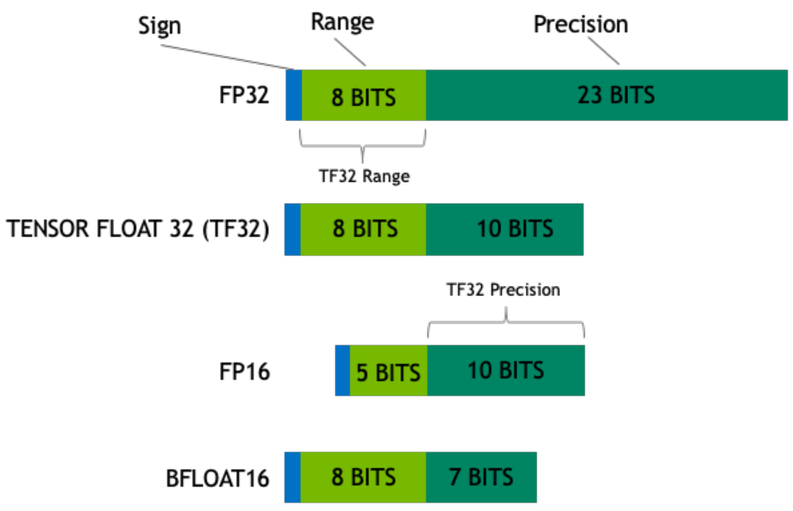

Figure 2.2: CUDA 12 floating-point data types.

number of bits available to describe the matrix elements makes broader representations possible. However, this implies dwindling performance.

For this reason, CUDA 12 offers intermediate representations to leverage reduced precision performance with limited precision losses. As shown in figure 2.2, CUDA 12 provides four floating-point representations for 32 bits or less.

- Extended precision:
  - FP64 or `double`.
- Single precision:
  - FP32 or `float`.
- Reduced precision:
  - FP16 or `half` precision provides half of the bits of FP32. It can represent a small subset of the values represented by FP32.
- Alternate floating point:
  - TF32 or tf32 provides the same range as FP32 and reduced precision.
  - BFLOAT16 or `nv_bfloat16` provides the same range as FP32 and reduced precision.

Double precision implies 64 bits for representation.

The DGEMM (Double GEMM) operation holds the potential to be a significant breakthrough. It allows for direct Tensor Cores usage without precision loss, which theoretically will deliver Tensor Cores throughput with extended precision. This advancement could remove the previous deterrent of precision loss, making the usage of Tensor Cores feasible in high-performance applications.

2.5 Tensor Cores Targeting

CUDA 12 provides three APIs for targeting Tensor Cores: `WMMA`, `cuBLASLt`, and `cuTENSOR`.

- `WMMA` is a set of Warp Matrix Functions that is part of the C/C++ CUDA Language extension. The API provides the lowest level of programmability of Tensor Cores GEMM operations. Cooperation from all threads in a warp is required. All threads in the warp must execute the same function calls and evaluate identically under control structures.

  ```
  template<typename Use, int m, int n, int k, typename T, typename Layout=void> class fragment;

  void load_matrix_sync(fragment<...> &a, const T* mptr, unsigned ldm);
  void load_matrix_sync(fragment<...> &a, const T* mptr, unsigned ldm, layout_t layout);
  void store_matrix_sync(T* mptr, const fragment<...> &a, unsigned ldm, layout_t layout);
  void fill_fragment(fragment<...> &a, const T& v);
  void mma_sync(fragment<...> &d, const fragment<...> &a, const fragment<...> &b, const
      fragment<...> &c, bool satf=false);
  ```

  Listing 2.1: WMMA namespace

  The first step is initializing an opaque CUDA data structure containing information relative to the operation configuration.

  As shown in listing 2.1, a `fragment` contains information relative to one of the matrices involved in the GEMM operation. It specifies the corresponding GEMM operand, the dimensions of all the operands, the type, and the data layout. The data is distributed across all threads in the warp. The API supplies functions to load, fill, and retrieve data from the structure.

  The `mma_sync` function performs the GEMM operation. It is a locking function. The warp-synchronous GEMM operation is performed after all warp threads have called it.

  This set of functions can carry out all GEMM operation modes currently supported in Tensor Cores; however, dimension constraints must be satisfied. Some precision options are experimental and might be subject to change.

- `cuBLASLt`: `cuBLAS` [8] is a GPU-accelerated Basic Linear Algebra Library. It contains several API extensions to provide standard BLAS and GEMM APIs. These are highly optimized for NVIDIA GPUs.

The `cuBLASLt` are highly expressive multi-stage APIs for GEMM operations. It provides customizable precision, algorithms, and heuristics with reduced API programmability complexity.

The library automatically uses Tensor Cores whenever possible.

The API executes the GEMM operation via the `cublasLtMatmul` function.

`cuBLASLt` operands must be in column-major storage, and the operation parameters are portable between different instances of the operations using specific structures and API calls.

- `cuTENSOR` [10] is a GPU-accelerated library for tensor contraction, reduction, and element-wise operations. The library contains just-in-time compiled kernels for tensor contraction.

`CUTLASS` [19] is a collection of CUDA C++ templates for implementing high-performance GEMM computations on all levels and scales in CUDA kernels. The aim is to create module-composable GEMM parts abstracted by C++ template classes, whose composition can express and perform GEMM operations subjected to the programmer's needs and preferences and specific CUDA kernels.

2.6 Third-Generation Tensor Cores Architecture

The launch of the NVIDIA Ampere architecture implemented third-generation Tensor Cores.

This new generation added comprehensive support for high-performance and deep-learning data types and a new sparsity feature. There are two classes of Ampere-architecture GPUs:

- GA100 [13].
- GA10x [12].

TF32, BFLOAT16, and IEE 754-compliant FP64 are the new data types that are supported. The BFLOAT16 operation delivers the same throughput as FP16.

However, GA10x GPUs do not include Tensor Cores acceleration for double-precision (FP64) operations, as provided in GA100. The new sparsity feature theoretically improves up to 2x the throughput of the predecessor generation.

A NVIDIA A100 Tensor Core GPU is based on GA100 and has 108 SMs. The A100 includes 432 third-generation Tensor Cores. Using Tensor Cores, each SM in a A100 computes 64 FP64 FMA operations/clock (or 128 FP64 operations/clock).

The GA102 is the most potent Ampere architecture GPU in the GA10x lineup. It has 84 SMs and 336 third-generation Tensor Cores.

The NVIDIA NVIDIA GeForce RTX 3080 Ti is based on the GA102.

Table 2.1 shows the A100 and GeForce RTX 3080 Ti theoretical performance.

Metric (TFLOPS)	**A100**	**GeForce RTX 3080 Ti**
Peak FP16 (non-Tensor)	78	35.6
Peak BF16 (non-Tensor)	39	35.6
Peak FP32 (non-Tensor)	19.5	35.6
Peak FP64 (non-Tensor)	9.7	-
Peak FP16 Tensor with FP16 Accumulate	312	142
Peak FP16 Tensor with FP32 Accumulate	312	71
Peak BF16 Tensor with FP32 Accumulate	312	71
Peak TF32 Tensor	156	35.6
Peak FP64 Tensor	19.5	-

Table 2.1: A100 and GeForce RTX 3080 Ti theoretical performance.

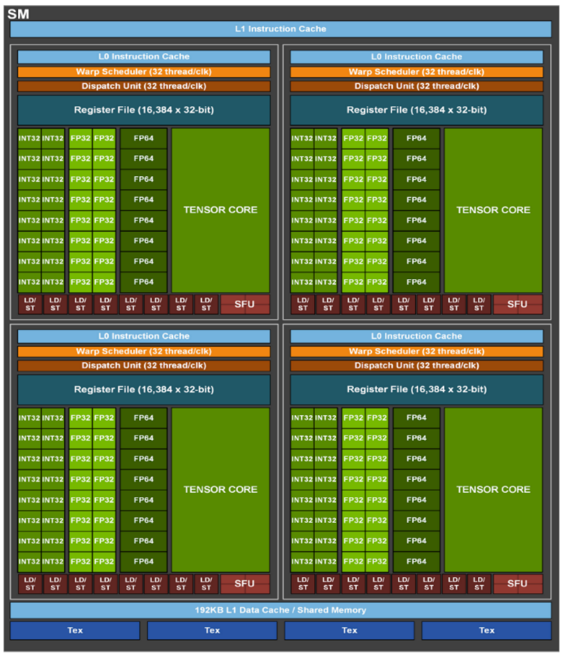
SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
FP64
TENSOR CORE
LD/ST
SFU
192KB L1 Data Cache / Shared Memory
Tex

Figure 2.3: GA100 SM.

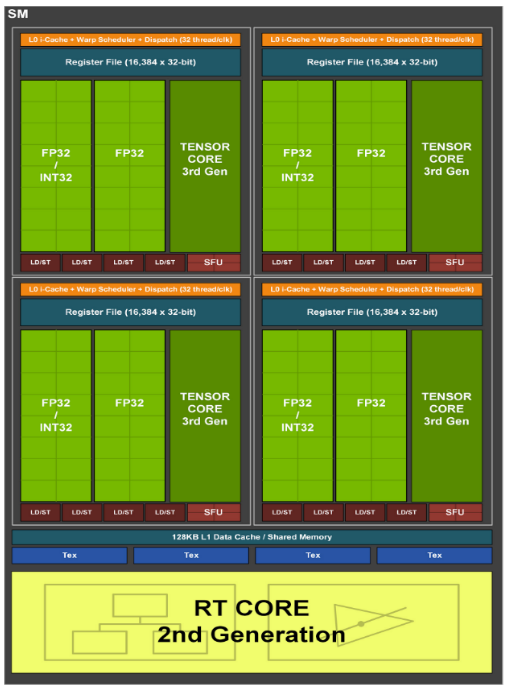
SM
L0 i-Cache + Warp Scheduler + Dispatch (32 thread/clk)
Register File (16,384 x 32-bit)
FP32 / INT32
FP32
TENSOR CORE 3rd Gen
LD/ST
SFU
128KB L1 Data Cache / Shared Memory
Tex
RT CORE 2nd Generation

Figure 2.4: GA10x SM.

3 Matrix Multiplication Complexity

In algorithmic complexity, matrix multiplication is an ongoing topic. Plenty of research has already been conducted.

Suppose the multiplication of two n × n matrices needs $O(n^{\alpha})$ operations. In that case, the least upper bound for α is called the exponent of matrix multiplication and is denoted by ω.

The naive way to perform matrix multiplication proceeds as follows:
If **A** is an $m \times n$ matrix and **B** is an $n \times p$ matrix,

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix},$$

the *matrix product* **C** = **AB** (denoted without multiplication signs or dots) is defined to be the $m \times p$ matrix

$$\mathbf{C} = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{pmatrix},$$

such that

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj},$$

for $i = 1, \ldots, m$ and $j = 1, \ldots, p$.

Thus, the naive approach needs $i * j * n$ operations, which represents a ω of 3.

In 1969, Volker Strassen [18] developed an algorithm with a lower complexity than the naive approach, which is based on a divide-and-conquer approach.

Strassen's algorithm achieves a complexity of $O(n^{\log_2(7)})$. This new $\omega = \log_2(7)$ remained the upper bound for matrix multiplication for a long time.

After that, a lower bound of $O(n^{2\log_2(n)})$ was proved by R. Raz [17].

Researchers conducted studies to improve Strassen's algorithm, achieving a new theoretical upper bound of $\omega < 2.371866$ [20].

Nevertheless, Strassen's algorithm is the only one with a practical application. Approaches with lower complexity require unachievable resources, such as infinite memory.

Multiple libraries implement Strassen's algorithm, for example, `cuBLASLt`.

4 Experimental Set-Up

I have developed an API for tracking NVIDIA Tensor Cores performance [1].

The API offers a simple set of functions to perform matrix-matrix multiplication using different accelerators. The API provides three execution modes:

- `naiveMatrixMultiply` employing a CUDA Kernel with a naive non-optimized GEMM algorithm.
- `wmmaMatrixMultiply` employing a CUDA Kernel executing the GEMM operation through `WMMA` namespace calls.
- `cublasMatrixMultiply` performing the GEMM operation using the `cuBLASLt` API.

The first execution mode uses regular CUDA Cores, and the last two leverage Tensor Cores.

The API offers a `Matrix_2D` object that encapsulates all the required functionality to perform the GEMM operation using the abovementioned functions.
`Matrix_2D` abstracts the kernel functionality and manages host and device storage.

The API offers further related functionality. It provides a member function `fillRandom` that fills the matrices using a pseudo-random generator contained in the C++ standard library: `std::mersenne_twister_engine`, which is a random number engine based on the Mersenne Twister algorithm. Mersenne Twister is a 623-dimensionally equidistributed uniform pseudo-random number generator [6].

The benchmark makes use of this API. Two different accelerators have executed the benchmark:

- RTX 3080 Ti accelerator connected to an AMD EPYC 7402 host. The operating system is Ubuntu 20.04.2 LTS, kernel 5.4.0-81. The accelerator uses CUDA 12.0 with compute capability 8.6. The GNU compiler for host code compile is 9.4.0. The `nvcc` compiler flags `-g -gencode arch=compute_86,code=sm_86 -x cu -c`.
- A100 accelerator connected to an Intel Xeon 8358 host. The operating system is Red Hat Enterprise Linux 8.7 (Ootpa). The accelerator uses CUDA 12.1 and driver version 530.30.02.

[1] `https://doi.org/10.5281/zenodo.13325208`

I measured the performance of NVIDIA Tensor Cores using the GEMM operation $D = \alpha AB + \beta C$ with $\alpha = 1.0$ and $\beta = 0$ and in-place operation $C = D$.

The benchmark tracks the TFlop/s using square matrices of size N for each dimension. It creates the A and B matrices and randomly fills them using the `fillRandom` member function. The function generates each value in double precision and then converts it to a reduced precision if required. Then, the benchmark calls the three execution modes. The execution functions initialize all C values with 0.

The main figure of merit for performance regarded is TFlop/s. The API performs the operation considering the following:

- Execution time is exclusively the execution time of the kernel. It is obtained through NVIDIA Nsight Compute CLI [11] with the flags `--csv --metrics=gpu__time_duration`.
- GEMM complexity is $O(n^3)$, and thus, the number of floating point operations performed on a complete GEMM operation is n^3. It is necessary to note that the CuBLAS API calls have been performed, providing the API with $4 * N * M$ bytes of available space, with NxM being the dimensions of the result matrix. This space is essential to store intermediate values when executing algorithms with complexity smaller than $O(n^3)$.

5 Results

The benchmark tracks the performance of the GEMM operation for each of the floating-point real-number data types supported by the `cublasLtMatmul` function. Table 5.1 presents these combinations:

- The first column describes the possible values of the enumerator `cublasComputeType_t`. Table 5.2 describes them. This enumerator determines the data type of matrix C and D as the potential down-conversion used to speed up the calculations.
- The second and third columns describe the `cudaDataType_t` enumerator possible values. Table 5.3 describes them. This enumerator represents the data types displayed in section 2.4.
  - The second column determines the values of scalars α and β.
  - The third column determines the values of matrices A and B.

computeType	**scaleType**	**Atype/Btype**
CUBLAS_COMPUTE_16F	CUDA_R_16F	CUDA_R_16F
CUBLAS_COMPUTE_32F	CUDA_R_32F	CUDA_R_16F
		CUDA_R_16BF
		CUDA_R_32F
CUBLAS_COMPUTE_32F_FAST_16F or CUBLAS_COMPUTE_32F_FAST_16BF or CUBLAS_COMPUTE_32F_FAST_TF32	CUDA_R_32F	CUDA_R_32F
CUBLAS_COMPUTE_64F	CUDA_R_64F	CUDA_R_64F

Table 5.1: `cublasLtMatmul` operation modes supported.

Value	**Meaning**
CUBLAS_COMPUTE_16F	Default and highest-performance mode for 16-bit half precision floating point and all compute and intermediate storage precisions with at least 16-bit half precision. Tensor Cores will be used whenever possible.
CUBLAS_COMPUTE_32F	Default 32-bit single precision floating point and uses compute and intermediate storage precisions of at least 32-bits.
CUBLAS_COMPUTE_32F_FAST_16F	Allows the library to use Tensor Cores with automatic down-conversion and 16-bit half-precision compute for 32-bit input and output matrices.
CUBLAS_COMPUTE_32F_FAST_16BF	Allows the library to use Tensor Cores with automatic down-conversion and bfloat16 compute for 32-bit input and output matrices.
CUBLAS_COMPUTE_32F_FAST_TF32	Allows the library to use Tensor Cores with TF32 compute for 32-bit input and output matrices.
CUBLAS_COMPUTE_64F	Default 64-bit double precision floating point and uses compute and intermediate storage precisions of at least 64-bits.

Table 5.2: Enumerator `cublasComputeType_t`.

Value	**Meaning**
CUDA_R_16F	16-bit real half precision floating-point
CUDA_R_16BF	16-bit real bfloat16 floating-point
CUDA_R_32F	32-bit real single precision floating-point
CUDA_R_64F	64-bit real double precision floating-point

Table 5.3: Enumerator `cudaDataType_t`.

Five GEMMs are considered: HGEMM (`half` GEMM), `half` to `float` GEMM, `nv_bfloat16` to `float` GEMM, DGEMM (`double` GEMM) and FGEMM (`float` GEMM). The benchmark has tracked each one with matrix sizes from $N = 256$, doubling the size until $N = 16384$, and performs the procedure described in chapter 4.

HGEMM (`half` GEMM) operation performance is shown in figures 5.1 and 5.2. Compared with the `cublasMatrixMultiply`, the `wmmaMatrixMultiply` is 7 and 12 times slower, and the `naiveMatrixMultiply` is 188 and 300 times slower, respectively, for the RTX 3080 Ti and A100 accelerators. The A100 accelerator offers almost twice the TFlop/s of the RTX 3080 Ti.

`half` to `float` GEMM operation performance and the `nv_bfloat16` to `float` GEMM operation performance are shown in figures 5.3, 5.4, 5.5 and 5.6. In the two GEMMs, compared with the `cublasMatrixMultiply`, the `wmmaMatrixMultiply` is 4 and 12 times slower for the RTX 3080 Ti and A100 accelerators. The `naiveMatrixMultiply` performs HGEMM in the four figures.

The benchmark achieves the same peak of 151 TFlops/s for HGEMM, `half` to `float` GEMM and `nv_bfloat16` to `float`. `cublasMatrixMultiply` execution mode achieves this peak with a matrix size of $N = 16384$ using the A100 accelerator.

Every measured reduced precision Tensor Cores GEMM is equally performant. As stated in section 2.4, reduced precision representations `half` and `nv_bfloat16` occupy the same number of bits. This equality indicates that Tensor Cores data type width is directly related to performance.

All reduced precision Tensor Cores GEMM have the same theoretical 312 TFlop/s peak. Table 2.1 shows NVIDIA's theoretical Tensor Cores peak performance.

Anyhow, the measured performance is less than half the theoretical performance provided by NVIDIA.

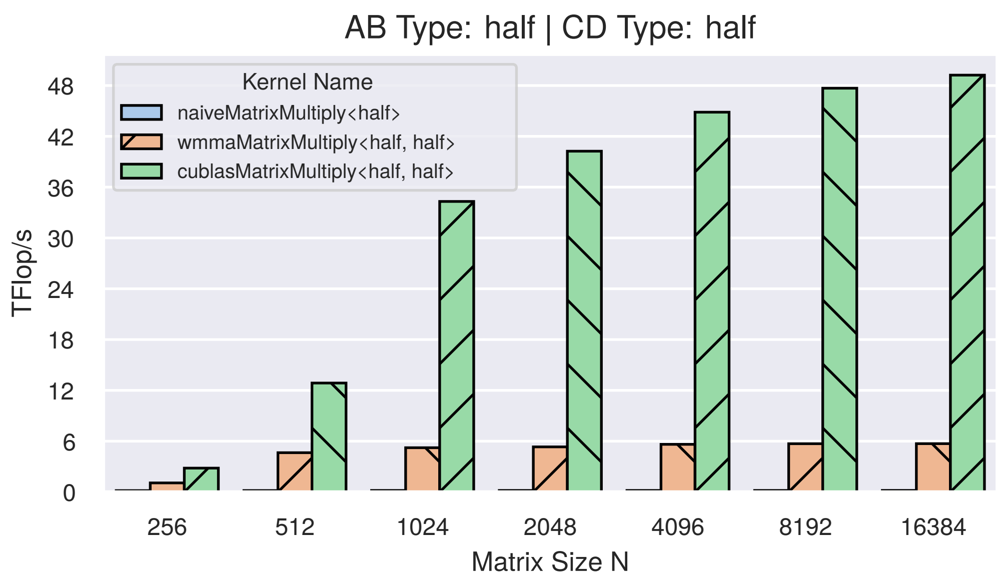

Figure 5.1: Half to Half on RTX 3080 Ti.

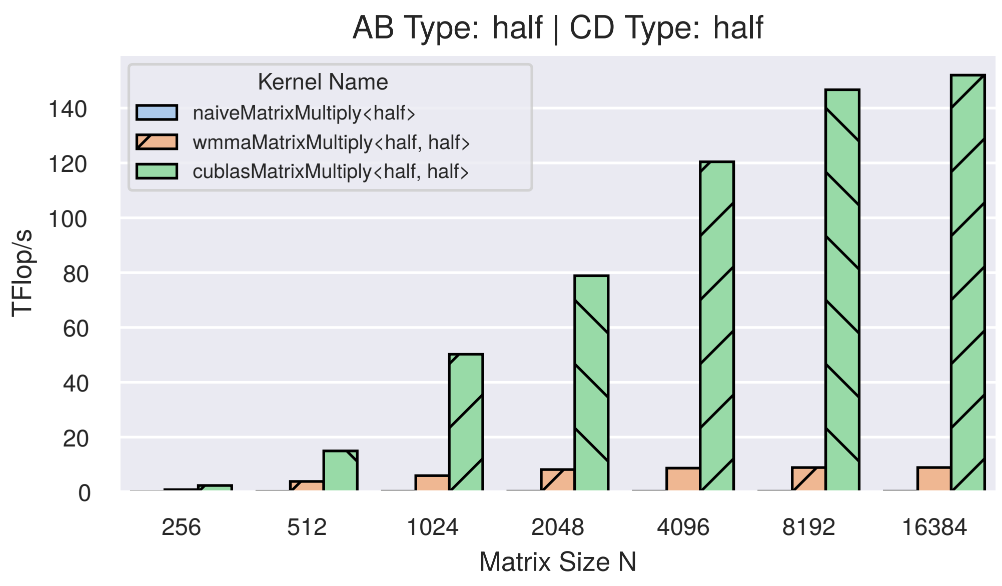

Figure 5.2: Half to Half on A100.

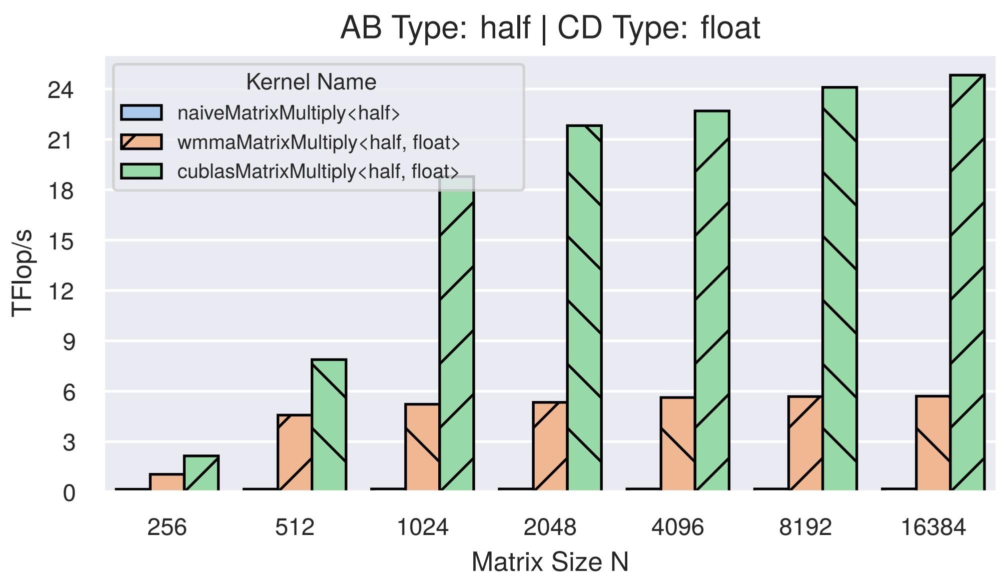

Figure 5.3: Half to Float on RTX 3080 Ti.

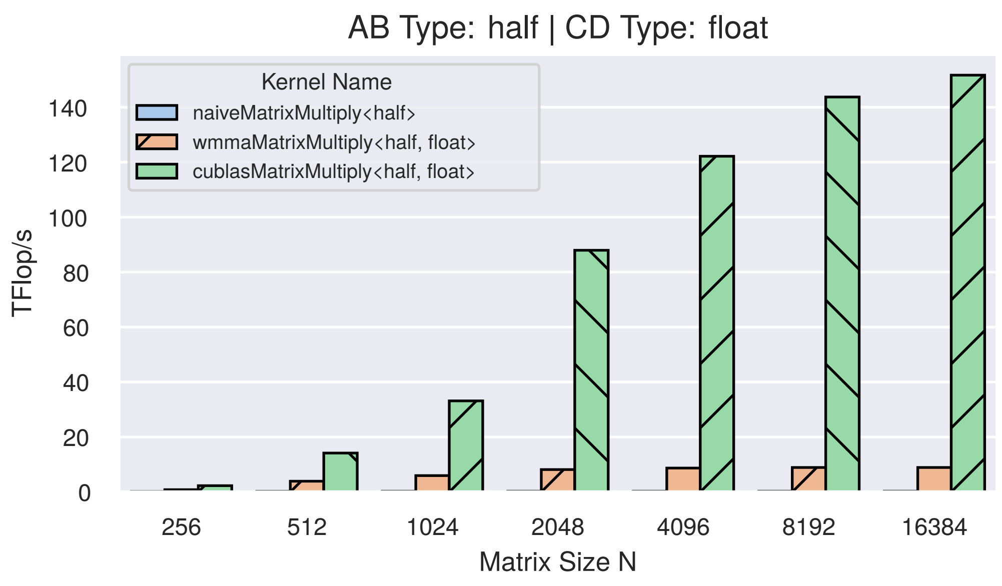

Figure 5.4: Half to Float on A100.

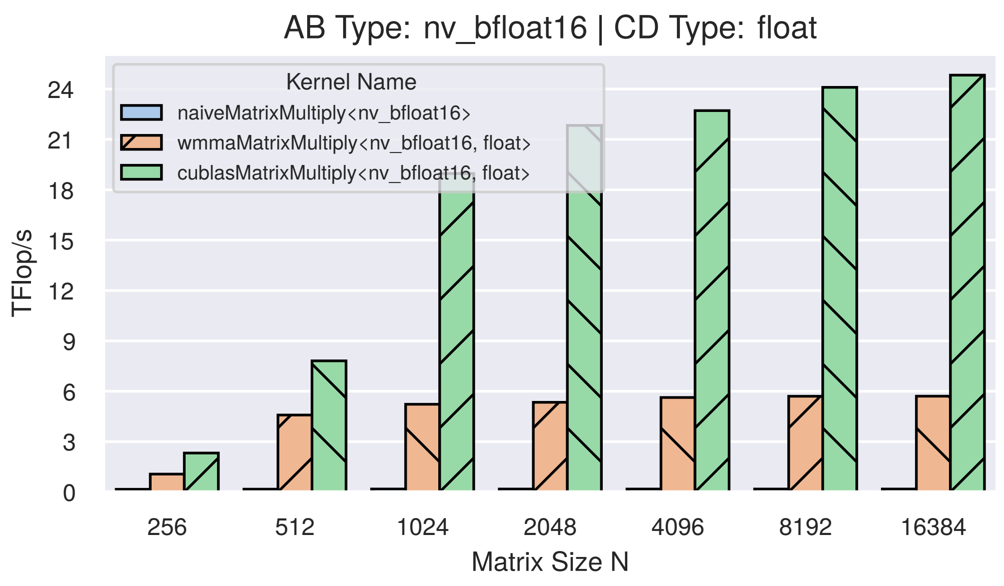

Figure 5.5: BFloat16 to Float on RTX 3080 Ti.

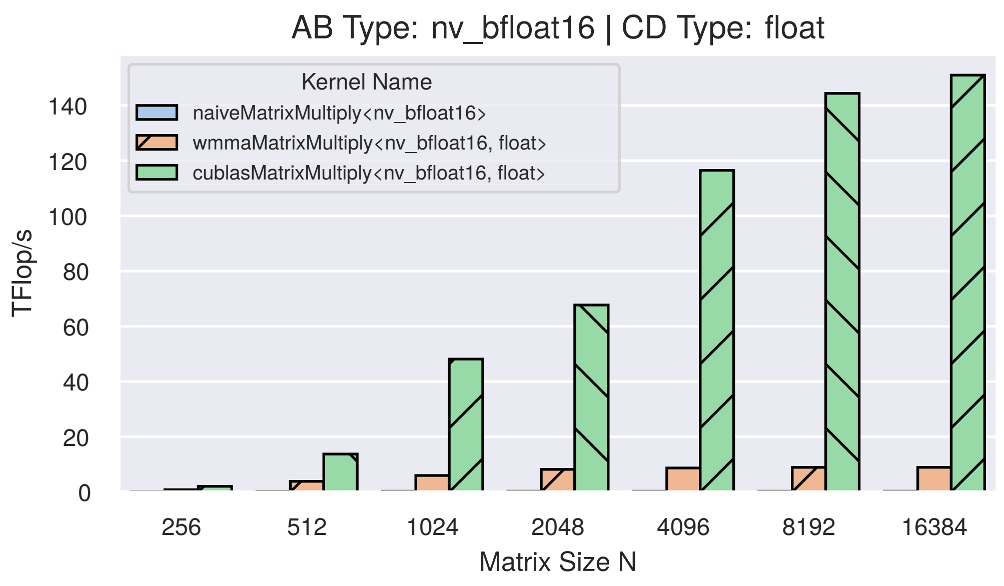

Figure 5.6: BFloat16 to Float on A100.

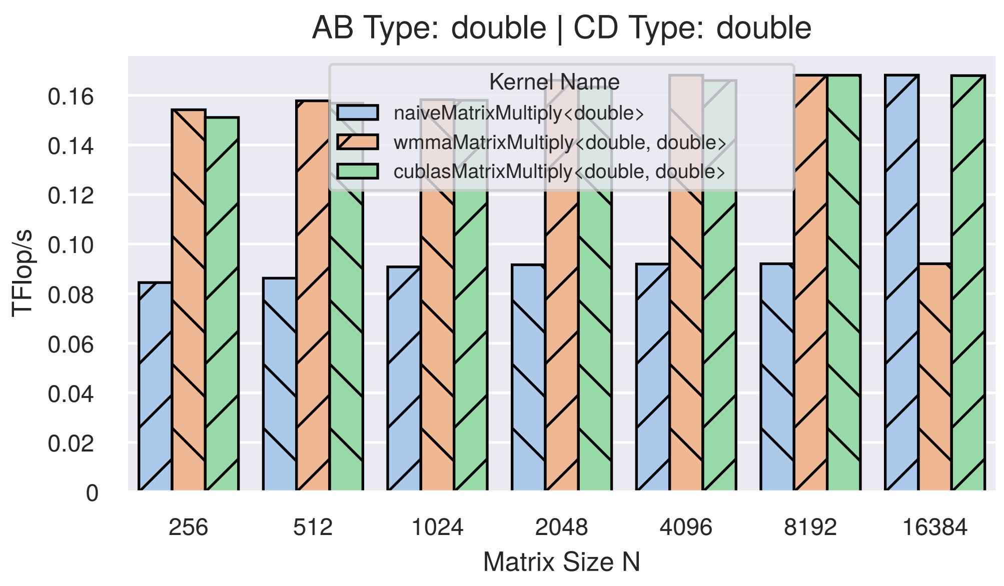

Figure 5.7: Double to Double on RTX 3080 Ti.

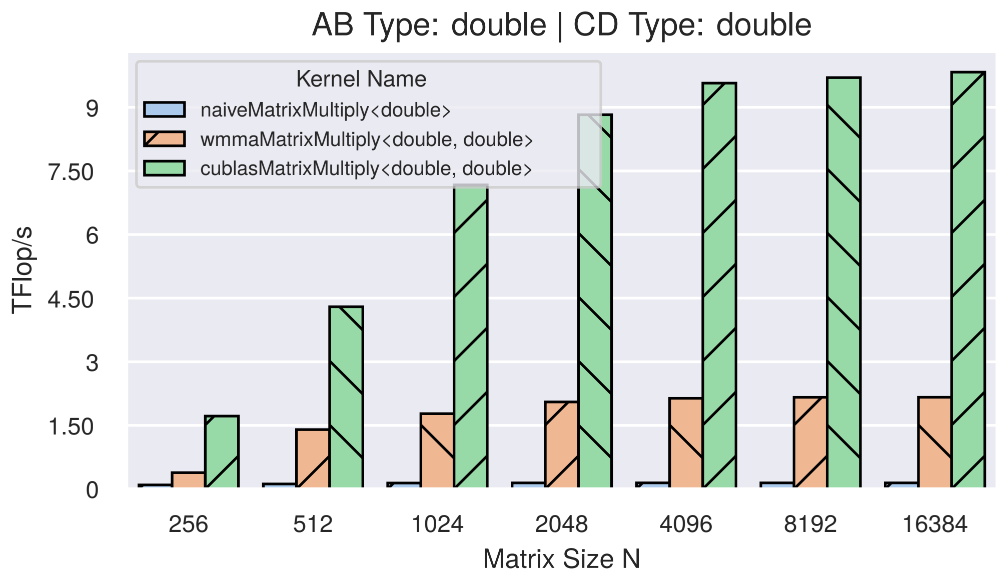

Figure 5.8: Double to Double on A100.

DGEMM (`double` GEMM) operation performance is shown in figures 5.7 and 5.8. The `naiveMatrixMultiply`, `wmmaMatrixMultiply`, and `cublasMatrixMultiply` are equally performant on the RTX 3080 Ti accelerator. While on the A100 accelerator, compared with the `cublasMatrixMultiply`, the `naiveMatrixMultiply` and `wmmaMatrixMultiply` are respectively 56 and 4 times slower. The benchmark achieves a peak of 9.8 TFlop/s Tensor Cores for DGEMM with `cublasMatrixMultiply` execution mode and a matrix size of $N = 16384$ using the A100 accelerator.

For DGEMM, the number of bits of the data type increases 4 times, dwindling the TFlop/s peak by 15 times. The decrease factor of performance is 4 times the bit width increase factor.

The measured performance is less than half the theoretical performance provided by NVIDIA.

FGEMM (`float` GEMM) operation performance is shown in figures 5.9, 5.10.

Five execution modes are represented in the figures and depicted in the legend. In the legend, the first entry corresponds to the `naiveMatrixMultiply` and the last four to `cublasMatrixMultiply` with different down-conversion strategies as described in table 5.2. The last four legend entries omit the name of the execution mode and show only its parameterization for presentation purposes. There are two types of parameterizations:

- Without down-conversion: all compute, and intermediate storage precisions are 32-bits. The `naiveMatrixMultiply` execution mode is 41 and 28 times slower than `cublasMatrixMultiply` on the RTX 3080 Ti accelerator and on the A100 accelerator, respectively. The implementation of optimizations and Strassen's algorithm in the `cuBLAS` library are responsible for these speed-ups as, in this operation mode, the library is not using Tensor Cores.

  The `cublasMatrixMultiply` execution mode without down-conversion executes non-Tensor FGEMM and is equally performant in the RTX 3080 Ti as in the A100.

- With down-conversion: this operation mode allows the library to use `half`, `nv_bfloat16` or `tf32` Tensor Cores.

  `cublasMatrixMultiply` execution mode without down-conversion is, on average, 6 and 8 times slower than the `cublasMatrixMultiply` execution mode with down-conversion on the RTX 3080 Ti accelerator and on the A100 accelerator, respectively. The benchmark achieves a FGEMM with down-conversion 81 peak TFlop/s. `half` down conversion execution mode achieves this peak with a matrix size of $N = 16384$ using the A100 accelerator.

Figures 5.11 and 5.12 represent the precision loss of the execution modes that allow down-conversion. The benchmark performs the epsilon calculation considering as ground truth the result of the execution mode without down-conversion. The down-conversion data type that achieves the lowest epsilon is `half` for both accelerators. The RTX 3080 Ti accelerator shows more precision loss than the A100 accelerator for every execution mode with down-conversion.

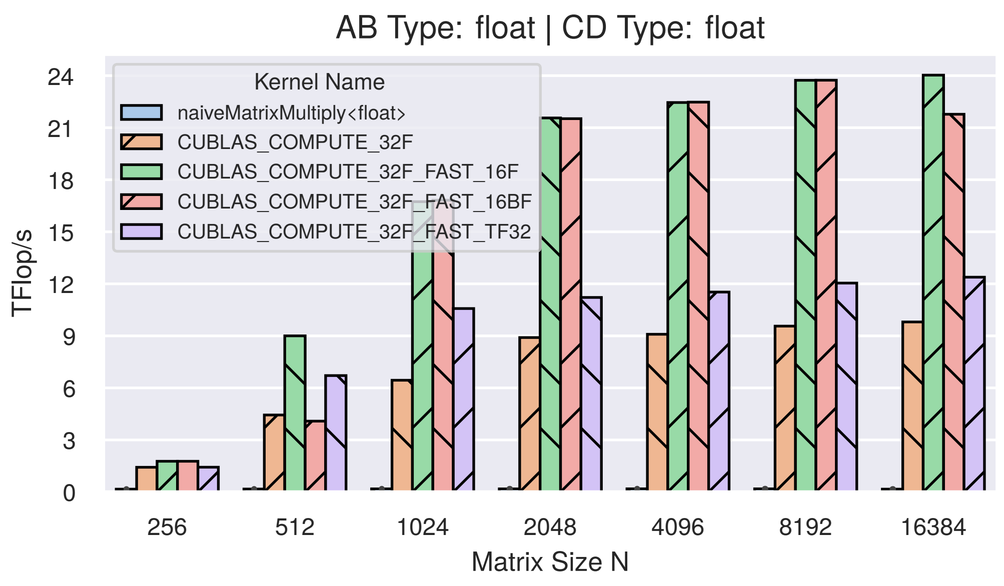

Figure 5.9: Float to Float on RTX 3080 Ti.

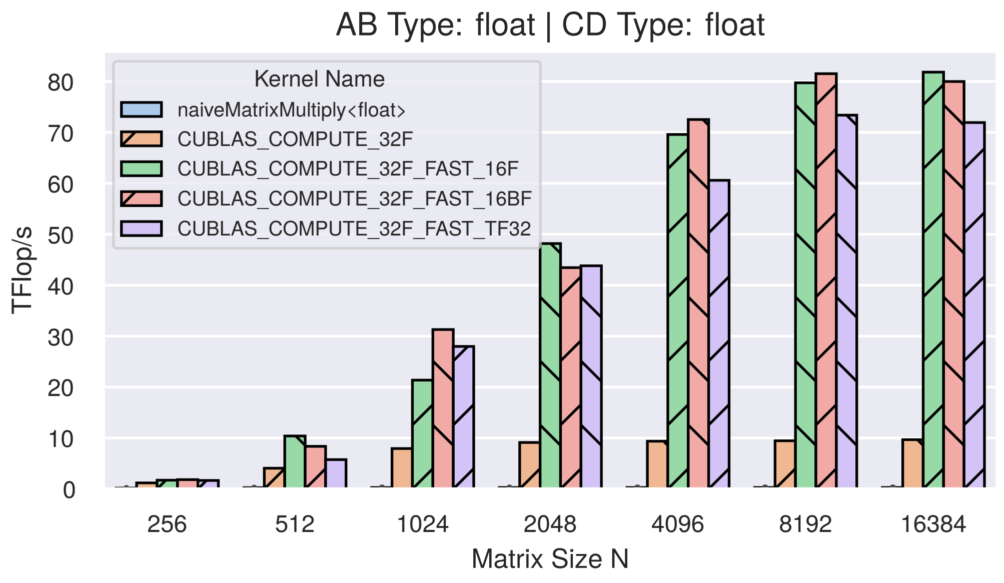

Figure 5.10: Float to Float on A100.

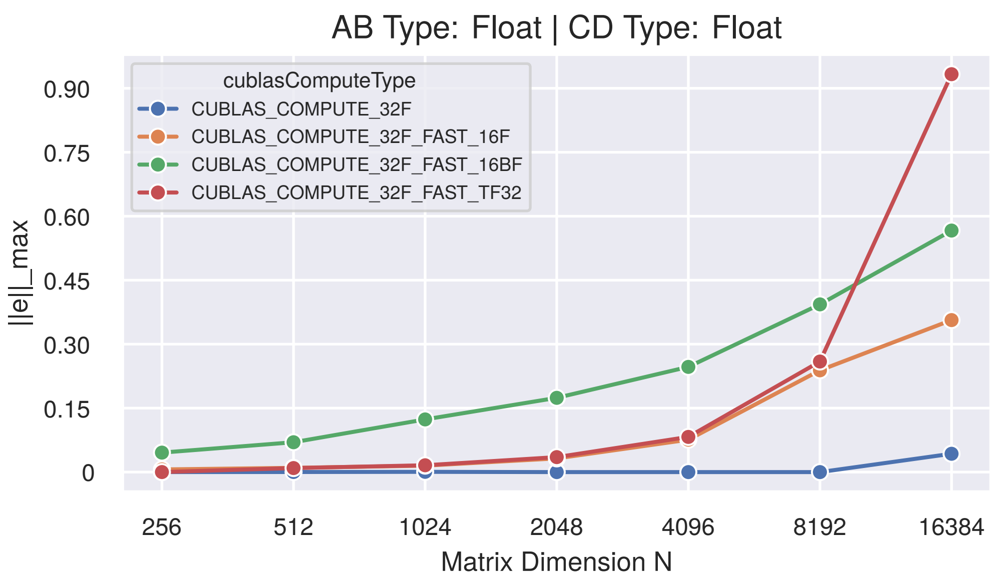

Figure 5.11: Float to Float epsilon on RTX 3080 Ti.

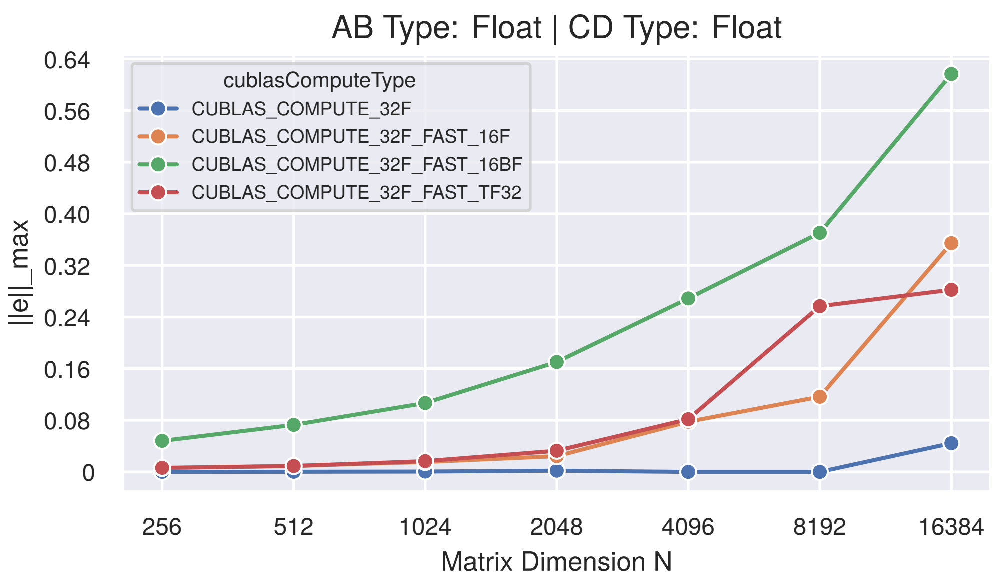

Figure 5.12: Float to Float epsilon on A100.

6 Conclusion and Further Work

NVIDIA GPUs are currently the most valued and demanded computer hardware and will also be an essential asset in the future. Existing applications, such as artificial intelligence, high-performance applications, and computer vision, have established GPUs as the best hardware accelerator for computing-intensive tasks. This work proves that Tensor Cores GEMM kernels provide more excellent acceleration than CUDA non-Tensor GEMM kernels.

NVIDIA's cuBLAS library, the CUDA Basic Linear Algebra Subroutine library, is hailed as the optimal method for executing any linear algebra subroutines in NVIDIA GPUs. However, this research goes a step further to prove that `cuBLASLt` outperforms any benchmarked implementation for Tensor and non-Tensor GEMM. It also demonstrates that `cuBLASLt` offers all the functionality of Tensor Cores with enhanced performance, while maintaining a lower programmability complexity than the `WMMA` namespace.

While NVIDIA initially launched the first-generation Tensor Cores to accelerate artificial intelligence applications, the unveiling of subsequent generations has broadened their potential applications. This research highlights high-performance computing as one of the many fields that could benefit from Tensor Cores' acceleration. It also demonstrates that the GA100 GPU offers IEE-754-compliant extended-precision GEMM with excellent acceleration, a feat that the GA10x GPU cannot match.

This work also shows how the GA10x GPU and GA100 GPU can accelerate single-precision GEMMs using the `cuBLASLt` library. Moreover, that `cuBLASLt` with down-conversion Tensor Cores can accelerate the operation with reduced precision losses. The best benchmarked down-conversion data type is `half` in terms of performance (peak TFlop/s) and precision loss (maximum epsilon).

This work studies NVIDIA Tensor Cores precision and performance for targeting high-performance applications, although further work is required to apply this novel technique to real applications.

The Tensor Cores acceleration for IEE-754-compliant extended-precision GEMM is a breakthrough. NVIDIA Tensor Cores improve up to two times the performance of non-Tensor CUDA Cores. High-performance applications should benefit from this performance improvement and adopt the fastest available GPU cores for time-consuming extended-precision operations. Further study is needed to benchmark the performance improvement of Tensor Cores in these applications.

List of Figures

List of Tables

Bibliography

[1] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. “A set of level 3 basic linear algebra subprograms.” In: vol. 16. 1. New York, NY, USA: Association for Computing Machinery, Mar. 1990, pp. 1–17. DOI: 10.1145/77626.79170.

[2] B. Feng, Y. Wang, G. Chen, W. Zhang, Y. Xie, and Y. Ding. “EGEMM-TC: accelerating scientific computing on tensor cores with extended precision.” In: *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2021, pp. 278–291. DOI: 10.1145/3437801.3441599.

[3] “IEEE Standard for Floating-Point Arithmetic.” In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), pp. 1–84. DOI: 10.1109/IEEESTD.2019.8766229.

[4] G. Kyung, C. Jung, and K. Lee. “An implementation of a SIMT architecture-based stream processor.” In: *TENCON 2014 - 2014 IEEE Region 10 Conference*. 2014, pp. 1–5. DOI: 10.1109/TENCON.2014.7022313.

[5] S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng, and J. S. Vetter. “NVIDIA Tensor Core Programmability, Performance Precision.” In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2018, pp. 522–531. DOI: 10.1109/IPDPSW.2018.00091.

[6] M. Matsumoto and T. Nishimura. “Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator.” In: vol. 8. 1. New York, NY, USA: Association for Computing Machinery, Jan. 1998, pp. 3–30. DOI: 10.1145/272991.272995.

[7] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu. “Mixed Precision Training.” In: 2018.

[8] NVIDIA Corporation. *cuBLAS Library Documentation*. 2024.

[9] NVIDIA Corporation. *CUDA C Programming Guide*. 2024.

[10] NVIDIA Corporation. *cuTensor Library Documentation*. 2024.

[11] NVIDIA Corporation. *Nsight Compute CLI*. 2024.

[12] NVIDIA Corporation. *NVIDIA A100 Tensor Core GPU Architecture Whitepaper*. Tech. rep. NVIDIA Corporation, 2024.

[13] NVIDIA Corporation. *NVIDIA Ampere GA-102 GPU Architecture Whitepaper*. Tech. rep. NVIDIA Corporation, 2024.

[14] NVIDIA Corporation. *NVIDIA Turing Architecture Whitepaper*. Tech. rep. NVIDIA Corporation, 2018.

[15] NVIDIA Corporation. *NVIDIA Volta Architecture*. Tech. rep. NVIDIA Corporation, 2017.

[16] M. A. Raihan, N. Goli, and T. M. Aamodt. "Modeling Deep Learning Accelerator Enabled GPUs." In: *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2019, pp. 79–92. DOI: `10.1109/ISPASS.2019.00016`.

[17] R. Raz. "On the complexity of matrix product." In: *Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing*. STOC '02. Montreal, Quebec, Canada: Association for Computing Machinery, 2002, pp. 144–151. ISBN: 1581134959. DOI: `10.1145/509907.509932`.

[18] V. Strassen. "Gaussian Elimination is Not Optimal." In: vol. 13. 1969, pp. 354–356. DOI: `10.1007/BF02165411`.

[19] V. Thakkar, P. Ramani, C. Cecka, A. Shivam, H. Lu, E. Yan, J. Kosaian, M. Hoemmen, H. Wu, A. Kerr, M. Nicely, D. Merrill, D. Blasig, F. Qiao, P. Majcher, P. Springer, M. Hohnerbach, J. Wang, and M. Gupta. *CUTLASS*. Version 3.0.0. Jan. 2023.

[20] V. V. Williams, Y. Xu, Z. Xu, and R. Zhou. "New Bounds for Matrix Multiplication: from Alpha to Omega." In: 2023. arXiv: `2307.07970 [cs.DS]`.