EGEMM-TC: Accelerating Scientific Computing on Tensor Cores with Extended Precision

Boyuan Feng[†], Yuke Wang[†], Guoyang Chen*, Weifeng Zhang*, Yuan Xie[†], Yufei Ding[†]
[†]{boyuan, yuke_wang, yuanxie, yufeiding}@ucsb.edu, *{g.chen, weifeng.z}@alibaba-inc.com
[†]University of California, Santa Barbara, *Alibaba Group US Inc.

Abstract

Nvidia Tensor Cores achieve high performance with half-precision matrix inputs tailored towards deep learning workloads. However, this limits the application of Tensor Cores especially in the area of scientific computing with high precision requirements. In this paper, we build Emulated GEMM on Tensor Cores (EGEMM-TC) to extend the usage of Tensor Cores to accelerate scientific computing applications without compromising the precision requirements. First, EGEMM-TC employs an extendable workflow of hardware profiling and operation design to generate a lightweight emulation algorithm on Tensor Cores with extended-precision. Second, EGEMM-TC exploits a set of Tensor Core kernel optimizations to achieve high performance, including the highly-efficient tensorization to exploit the Tensor Core memory architecture and the instruction-level optimizations to coordinate the emulation computation and memory access. Third, EGEMM-TC incorporates a hardware-aware analytic model to offer large flexibility for automatic performance tuning across various scientific computing workloads and input datasets. Extensive evaluations show that EGEMM-TC can achieve on average 3.13× and 11.18× speedup over the cuBLAS kernels and the CUDA-SDK kernels on CUDA Cores, respectively. Our case study on several scientific computing applications further confirms that EGEMM-TC can generalize the usage of Tensor Cores and achieve about 1.8× speedup compared to the hand-tuned, highly-optimized implementations running on CUDA Cores.

CCS Concepts • **Theory of computation → Massively parallel algorithms**.

Keywords Emulation, GEMM, Tensor Core

PPoPP '21, February 27–March 3, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8294-6/21/02…$15.00
https://doi.org/10.1145/3437801.3441599

Table 1. Precision Specifications. Unit: Number of Bits.

Data Type	Sign	Exponent	Mantissa
Half-Precision [30]	1	5	10
Single-Precision [30]	1	8	23
Markidis-Precision [20]	1	5	20
Extended-Precision [7]	**1**	**5**	**21**

1 Introduction

Recently, many specialized cores and hardware accelerators have been built to speed up the general matrix multiply (GEMM) in deep learning applications. These specialized cores typically exploit low-precision matrix computation (*e.g.*, half-precision) to achieve high performance, based on the fact that deep learning workloads involve many matrix operations and are usually robust to low-precision computation [4, 11, 21]. One example is the Tensor Core on Nvidia Volta GPUs that conduct half-precision matrix-matrix computation, achieving 8× higher throughput over the CUDA Cores [22]. Since GEMM is also one essential building block of many scientific computing applications, we will bring this performance benefit to the scientific computing domain. For example, GEMM operations take 85% and 67% of the total time in popular implementations of kNN [9] and kMeans [2], respectively. We refer to these applications as *GEMM-based scientific computing*. However, many scientific computing applications (*e.g.*, kNN and kMeans in large-scale physical simulations [8] and mathematical computations [3]) are rather sensitive to computation precision to generate valid results. Such a restriction on precision prevents them from exploiting powerful Tensor Cores for performance enhancement.

Several approaches have been proposed for extended-precision computation [7, 14, 34, 36] on limited-precision hardware, which utilizes multiple low-precision computing instructions to emulate a single extended-precision computing instruction. Table 1 summarizes these precision types. For example, the popular extended-precision technique, Dekker [7], can utilize 16 half-precision instructions for an extended-precision instruction. One key problem is that these techniques are developed and optimized on CPUs. It requires a significant amount of manual effort to transfer them to Tensor Cores without hurting performance. In particular, Dekker [7] requires serialized execution of the 16 instructions, leading to high overhead. Considering that half-precision computation on Tensor Cores is only 8× faster than single-precision

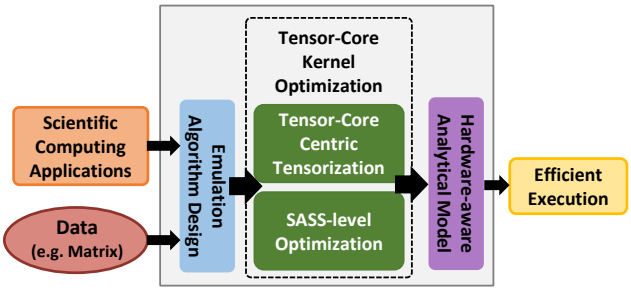


Figure 1. Overview of EGEMM-TC.

computation on CUDA Cores, this 16× overhead can easily make emulation inappropriate. Markidis [20] proposes a simple algorithm for emulation on Tensor Cores but utilizes a truncate-based strategy with 1-bit precision loss. It fails to achieve extended-precision and shows high overhead.

In this work, we design Emulated GEMM on Tensor Cores (EGEMM-TC) to accelerate GEMM-based scientific computing on Tensor Cores with both high performance and extended-precision computation. We identify several key challenges in the design and the development of EGEMM-TC. First, Tensor Cores require half-precision input matrices, leading to degraded computing precision. Naively borrowing existing emulation algorithms may lead to unsatisfactory performance. Second, the newly designed Tensor Cores bring new computing primitives and memory hierarchies, leading to unexplored optimizations. While Tensor Cores provide high computation performance, memory access speed remains the same as previous CUDA Cores and can easily become the bottleneck. Third, there is a large hyper-parameter design space on mapping scientific computing towards Tensor Cores. Experimenting with new hyper-parameters usually requires manual implementation [15, 17, 38], making the trial-and-error strategy not suitable.

To this end, we propose three techniques to tackle the above challenges, as shown in Figure 1. First, EGEMM-TC contains a lightweight emulation algorithm design with only 4 Tensor Core instructions. It achieves both extended-precision and low overhead by exploiting high-precision intermediate computation results. Second, EGEMM-TC utilizes a set of Tensor Core kernel optimizations that efficiently tensorize the emulation workload towards Tensor Cores with low memory overhead. EGEMM-TC also includes SASS-level optimizations for fully exploiting the instruction-level latency hiding opportunities and the register caching capability. Third, EGEMM-TC incorporates a hardware-aware analytic model to automatically explore the design space and reduce manual effort.

In summary, this paper makes the following contributions.

- We develop EGEMM-TC to accelerate scientific computing on Tensor Cores with extended-precision.
- We propose three novel techniques: a) a lightweight emulation algorithm to emulate extended-precision computation; b) a set of Tensor Core kernel optimizations to achieve high performance; c) a hardware-aware analytic model to facilitate the fast selection of hyper-parameters.
- We evaluate EGEMM-TC on Tesla T4 and Nvidia RTX 6000. It achieves 3.13× and 11.18× speedup on average over single-precision kernels on CUDA Cores from cuBLAS and CUDA-SDK, respectively. On a set of GEMM-based scientific computing applications, our approach achieves 1.8× speedup on average compared to hand-tuned code on CUDA Cores.

2 Background and Related Work

In this section, we discuss the background and the related work on Tensor Cores and emulation algorithms.

2.1 Tensor Cores

Tensor Core Computing and Memory Hierarchy. Different from scalar-scalar computation on CUDA Cores, Tensor Cores provide a matrix-matrix compute primitive. In particular, Tensor Cores support the compute primitive of $D = A \times B + C$, where A and B are required to be half-precision matrices, C and D can be configured to be half-precision or single-precision matrices. Before calling Tensor Cores, all registers in a warp need to collaboratively store these matrices into a new memory hierarchy *Fragment* [21], which allows data sharing across registers. This intra-warp sharing provides opportunities for fragment-based memory optimizations. Existing work [12, 13] reveals that Fragment is implemented as registers, from the perspective of hardware implementation.

Tensor Core Programming Interface. There are two popular programming interfaces for Tensor Cores — CUDA [27] and SASS [12, 13, 26, 29]. CUDA provides C-style APIs and enjoys the widest usage since it is easier to program. However, it provides only limited control over the hardware and cannot exploit the computing and memory capability. SASS provides assembly-style instructions that run natively on NVIDIA GPU hardware [13, 26, 29]. SASS is usually utilized by vendor experts in high-performance libraries (*e.g.*, Nvidia's cuBLAS [25]). In this paper, we will study the insight of Tensor Core related SASS instructions and propose a set of SASS-level optimizations to support high-performance GEMM-based scientific computing on Tensor Cores.

High-performance Computing on Tensor Cores. Some research efforts have been devoted towards accelerating high-performance computing workloads with Tensor Cores. Yan [37] utilizes Tensor Cores to accelerate half-precision GEMM. Dakkak [6] accelerates half-precision scan on Tensor Cores by transforming *scan* to a GEMM workload. While showing performance improvement by utilizing Tensor Cores, these

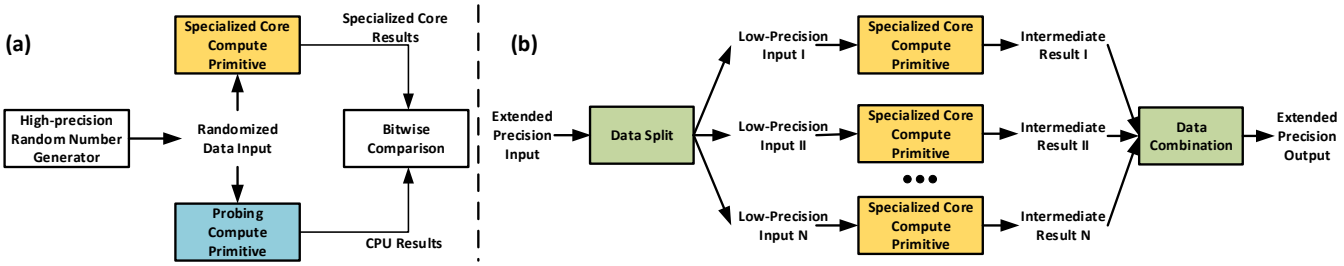


Figure 2. Illustration of the generalized emulation design workflow. It first uses (a) *precision profiling* to validate the precision of the intermediate results. Then, it uses (b) *emulation design* to generate a lightweight emulation algorithm based on the profiled precision from (a).

works focus on half-precision computation and fail to support extended-precision computation. By contrast, EGEMM-TC accelerates GEMM-based scientific computing on Tensor Cores with extended-precision and high performance.

2.2 Emulation Algorithms

There have been several emulation algorithms [7, 10, 14, 34, 36] that improve computation precision on low-precision hardware (e.g., IoT devices) and may be extended to Tensor Cores. One traditional emulation algorithm, Dekker [7], utilizes 16 half-precision instructions to emulate an extended-precision instruction. One recent work, Markidis [20], proposes a simple algorithm for emulation on Tensor Cores but utilizes a truncate-based strategy with 1-bit precision loss. By contrast, EGEMM-TC enjoys a lightweight emulation algorithm with 4 instructions, whose overhead is much reduced from Dekker. EGEMM-TC also employs a round-split algorithm that achieves higher precision by 1 extra mantissa bits, compared to Markidis [20]. In addition, EGEMM-TC achieves high performance by tailoring towards the Tensor Core architecture and incorporating a set of Tensor Core kernel optimizations.

3 Emulation Algorithm Design

As discussed in previous sections, existing emulation algorithms usually introduce high computation overhead. These algorithms assume that the hardware has the same input and output precision, thus utilizing a large number of low-precision instructions in the emulation. However, specialized cores usually have higher output precision than the input precision. For example, Tensor Cores require the input precision to be half-precision, while allowing the output precision to be single-precision. Moreover, modern specialized cores usually fuse multiplication and addition (*e.g.*, $D = A \times B + C$), where intermediate results $A \times B$ may also have higher precision than the input precision. Our key insight is that *exploiting high-precision intermediate results from hardware computation can effectively mitigate the emulation overhead.*

To this end, we first propose an extendable workflow to generate a lightweight emulation algorithm. Then, we showcase this workflow on the Tensor Cores and generate a Tensor-Core-specific emulation algorithm. Note that the workflow can be generally applied towards various accelerators and specialized cores. Here, we will focus on improving the precision of small matrices (*i.e.*, 16×16) that directly fit into the Tensor Core compute primitive and leave the performance consideration and large matrix tensorization to the next section.

3.1 Generalized Emulation Design Workflow

The emulation design workflow contains a precision profiling and an emulation design, as illustrated in Figure 2.

In precision profiling, the main idea is to simulate the computation results on the CPU and compare it bit-wisely with the specialized core results. In particular, we first generate a set of probing compute primitives with diverse intermediate precisions. Then, we evaluate the probing compute primitives on CPUs to get the corresponding results. Since current CPUs usually support a large range of precisions, we can get the ground-truth computation results of the probing compute primitives. Finally, we can get the computation results on the specialized cores as the ground truth, and compare it bitwisely with the CPU results. We repeat this procedure for a large number of randomized high-precision inputs. The "correct" probing compute primitive is identified if its value is bitwisely same with the specialized core results for all the tested inputs.

In emulation design, given the target-precision input, we first utilize a *data split* technique to split the target-precision input into several low-precision inputs following the hardware precision requirement and use each split input for specialized core computation. Then, we utilize a *data combination* technique to combine the intermediate results and generate the target-precision outputs. In the data combination technique, we will utilize the profiled intermediate precision to achieve the minimized overhead. We will provide concrete code snippets and emulation algorithms on Tensor Cores in the following sections.

3.2 Emulation Algorithm on Tensor Cores

In this section, we show our emulation algorithm on Tensor Cores. While it exploits the profiling on Nvidia Tensor Cores

to mitigate emulation overhead, its correctness can be easily verified on other specialized cores with our generalized emulation design workflow. When the precision is the same or higher, we can apply the same emulation algorithm as described below to achieve extended-precision computation. When the precision is lower (*e.g.*, half-precision), we may refer to Dekker [7], which assumes the hardware computation precision to be half-precision and emulates extended-precision computation at the cost of low performance.

Precision Profiling on Tensor Cores In this section, we showcase the precision profiling on Tensor Cores. Nvidia officially documents its *specialized core compute primitive* as $A \times B + C$, where the matrix A and B are half-precision, C and D can be either half-precision or single-precision. However, the operation precision during the matrix multiplication $A \times B$ is not officially documented. Without clear profiling, there are multiple *probing compute primitives*. One is that $A \times B$ is conducted in half-precision, which is the same as the data type of A and B. The other is that the half-precision matrices A and B are first converted to single-precision and $A \times B$ is conducted with single-precision or the extended-precision. Operation precision is important for the design and implementation of the emulation algorithm. Assuming both the operation and data are half-precision, Dekker shows that 16 instructions are required to emulate a single-precision instruction, which leads to high overhead.

```
half a1 = random_FP16();
half a2 = random_FP16();
half b1 = random_FP16();
half b2 = random_FP16();
initialize(FRAG_A, a1, i, k1, a2, i, k2);
initialize(FRAG_B, b1, k1, j, b2, k2, j);
zero_initialize(FRAG_C);
wmma::mma_sync(FRAG_D, FRAG_A, FRAG_B, FRAG_C);
float d_TC = access(FRAG_D, i, j);
float d_HALF = (float)(a1*b1+a2*b2);
float d_FLOAT = (float)a1 * (float)b1 + (float)a2 * (float)b2;
compare(d_TC, d_HALF, d_FLOAT);
```

Call Tensor Cores

Half-Precision Addition and Multiplication

Single-Precision Addition and Multiplication

Figure 3. Code Snippet for Tensor-Core Precision Profiling.

We use the following code (Figure 3) for profiling the operation precision in Tensor Cores. We randomly initialize the Tensor Core input matrices with half-precision data and use the *wmma::mma_sync()* CUDA API to call the specialized core compute primitive for computing *d_TC*. As reference values, we compute two probing compute primitives *d_HALF* and *d_FLOAT* of the above mentioned two possible operation precisions on CUDA Cores. Finally, we compare *d_TC* with *d_HALF* and *d_FLOAT* in a bit-wise manner. We randomly generate 10,000 groups of data and empirically observe that all *d_TC*s are identical to *d_FLOAT* bit-wisely up to 21 mantissa bits, which is required by the extended-precision computation. Thus we assume that the operation in Tensor Cores natively supports extended-precision and the only precision loss comes from the half-precision data type of A and B, enables our lightweight emulation algorithm.

Algorithm 1 Lightweight GEMM Emulation Design.

```
function Emulation(D, A, B, C)
    Alo, Ahi = Round-Split(A)
    Blo, Bhi = Round-Split(B)
    ▷ Tensor Core natively supports single-precision C and D
    D = wmma::mma_sync(Alo, Blo, C)
    D = wmma::mma_sync(Alo, Bhi, D)
    D = wmma::mma_sync(Ahi, Blo, D)
    D = wmma::mma_sync(Ahi, Bhi, D)
end function
```

Figure 4. Illustration of Round Split Algorithms

Emulation Design on Tensor Cores In this section, we showcase the emulation design on Tensor Cores, especially the *data split* and the *data combination*. Based on the profiling results, we propose a 4-instruction emulation operation for enabling extended-precision computation on Tensor Cores with 21 mantissa bits. Algorithm 1 summarizes our emulation algorithm. For simplicity, we illustrate with small matrices that match with the Tensor Core computing primitives of shape 16×16 and leave the large-matrix computation to the following sections. Our emulation algorithm takes single-precision matrices A, B, C, and D as the inputs and generates the outputs as $D = A \times B + C$ with extended-precision. The key idea is to first split single-precision matrices A and B into half-precision matrix A_{lo}, A_{hi}, B_{lo} and B_{hi}. Then we can compute on Tensor Cores and accumulate the intermediate results for data combination. Since Tensor Cores natively supports the single-precision C and D, we do not need to conduct data split on these two matrices.

There are multiple approaches for data split. One approach is *truncate-split* from Markidis [20], as illustrated in Figure 4(a). It truncates the single-precision data x to be half-precision x_{hi} and uses the x_{lo} to store the remaining value $x - x_{hi}$. While this approach is simple to implement, it supports only 20-bit mantissa from the two 10-bit mantissa of the half-precision data.

Instead, we propose a *round-split* approach as illustrated in Figure 4(b). Besides the two 10-bit mantissa, we encode an additional s bit in the sign bit (1-bit) from x_{lo}. For a positive x, the sign-bit of x_{lo} from truncate-split is always 0, but it may be 0 or 1 when round-split is conducted. In particular, for a positive x, we check the 21-th mantissa bit s and,

if s is positive, we add 1 to the 10-th mantissa bits for x_{hi} and recompute the x_{lo}. While the round-split method introduces extra overhead, it only needs to be conducted once on every matrix element with time complexity $O(N^2)$. This overhead is significantly less than the matrix multiplication time complexity $O(N^3)$ and introduces negligible overhead in emulation. To fully exploit GPU capability, EGEMM-TC conducts data split on CUDA Cores and computes the GEMM on Tensor Cores.

Emulation Overhead. Our emulation algorithm introduces 4× computation overhead, which is significantly small than the Dekker [7] method with 16× overhead. A naive implementation may also introduce 4× memory overhead when independently reading the split matrices for each computation instruction. However, this memory overhead can be reduced to 2× when the data reuse is carefully designed. We leave this memory optimization to Section 4.

4 Tensor-Core-Centric Tensorization

EGEMM-TC has a carefully designed tensorization to efficiently map the GEMM-based scientific computing to Tensor Cores that require specialized matrix inputs. While our tensorization shares some similarities with existing ones, there are two challenges to be addressed before fully exploiting the Tensor Core computing capability. First, existing techniques usually independently assign tasks to individual warps, failing to exploit the collaboration cross warps and within warps. Second, Tensor Cores provide a new memory architecture of fragment (FRAG), which is composed of registers across threads within a warp. This FRAG provides intra-warp caching opportunities that have not been well explored. To this end, we provide two novel optimizations.

Tensorization and Warp Collaboration Different from previous CUDA Cores on the scalar level, Tensor Cores computes at the matrix level, requiring the tensorization design. Matching with the GPU hierarchy, our tensorization recursively divides the matrices into sub-matrices and assign them to GPU blocks, warps, and threads, in a hierarchy-style. Formally, given input matrices A, B, and C, of shape (m, k), (k, n), (m, n), respectively, we split these matrices into *block matrices* of size (b_m, b_k), (b_k, b_n), and (b_m, b_n). During execution, each GPU block computes a block matrix of C based on the corresponding block matrices of A and B. Here, the sizes of block matrices are typically larger than the Tensor Core compute primitive size, requiring further dividing the block matrices to *warp matrices* with size (w_m, w_k), (w_k, w_n), and (w_m, w_n). These warp matrices will be assigned to individual warps for Tensor Core execution, where warp matrices will be further divided to *TC matrices* for matching the Tensor Core compute primitives with size t_m,t_n and t_k.

EGEMM-TC split the workload into two phases and adopts a warp collaboration strategy as illustrated in Figure 5. The main difference from previous CUDA Cores is that Tensor

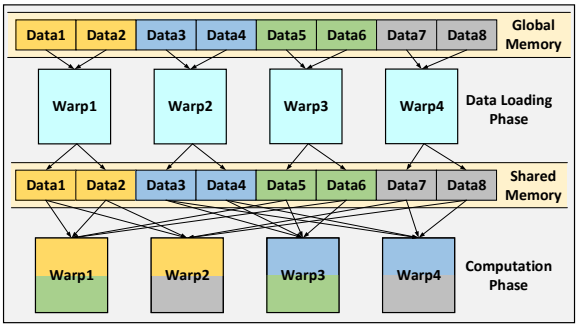


Figure 5. Warp Collaboration Illustration. During data loading phase, all warps collaboratively loads all data fragments. During computation phase, a data fragment may be used by multiple warps, indicated by the colors.

Table 2. Memory access on each GPU warp in GEMM workload. We skip the memory access of Ahi, Blo, and Bhi, since these matrices have similar memory access as the Alo.

Type	Size	w/o FRAG Caching	w/ FRAG Caching
Alo	$2w_m w_k$	$4w_k w_m \cdot w_k/t_k$	$2w_m w_k$
C	$4w_m w_n$	$4w_m w_n \cdot w_k/t_k$	$4w_m w_n$

Cores require 32 threads in a warp to collaboratively load matrices into the FRAG memory architecture and jointly compute the matrix multiplication and addition. Catering to the Tensor Core property, we assign different thread organization $(threadDim.x, threadDim.y)$ to the same warp during these two phases. During the computation phase, we utilize the default (32,1) thread layout for collaboratively calling Tensor Cores, as required by the CUDA programming guide [27]. During the data loading phase, we reorganize the warp threads to 2D layout for assigning non-overlapping memory access workload to each thread. For example, when loading a 16×16 block of data, it is much easier to program with the 16×2 thread configuration than with the default 32×1 one.

Intra-Warp FRAG Caching. Data caching is an effective strategy to reduce the memory overhead in the GEMM-based workload. Existing techniques usually utilize shared memory to cache a portion of the matrices A, B, and C but ignore the FRAG caching opportunity. We name it as the *w/o FRAG caching* strategy. With this strategy, data is still loaded multiple times from the shared memory to the register. Table 2 summarizes the memory access on a single GPU warp. When a warp matrix C of shape (w_m, w_n) are assigned to a GPU warp and stored in the shared memory, the memory access between shared memory and FRAG/register is

$$4t_n t_m \cdot \frac{w_k}{t_k} \cdot \frac{w_m}{t_m} \cdot \frac{w_n}{t_n} = 4w_m w_n \cdot \frac{w_k}{t_k} \quad (1)$$

where the warp matrix C is divided into $w_m/t_m \cdot w_n/t_n$ TC matrices, w_k/t_k times data loading when iterating over the

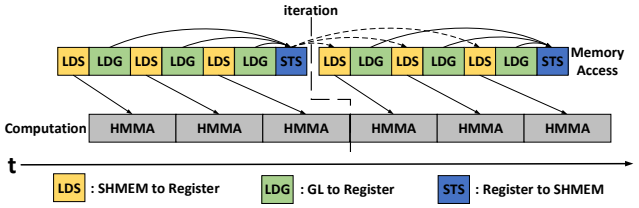


Figure 6. Illustration of Register-Enhanced Instruction Scheduling for Latency Hiding.

k-dimension, and each single-precision data requires 4 bytes. Similarly, we can compute the memory access of the warp matrix *Alo* as $2*2w_k w_m \cdot w_k/t_k$, where the first 2 comes from the emulation algorithm where *Alo* is used for two times. We observe that the memory access in the *w/o FRAG caching* strategy is significantly larger than the data size of *Alo* and *C*, leading to extra memory overhead.

Instead, we propose an *intra-warp FRAG caching* strategy, that effectively mitigates the memory overhead. The key observations are: 1) FRAG allows registers from multiple threads within a warp to collaboratively access a TC matrix, making it possible to reuse matrix data within warps; 2) FRAG has 256 KB which is much larger than the $64KB$ shared memory. EGEMM-TC will track whether a TC matrix has been stored in the FRAG and skip the data loading when possible. In particular, the TC matrix C is cached in FRAG during the whole computation and the *Alo* is read once to the FRAG. In total, this strategy leads to $4w_m w_n + 4 * 2w_m w_n$ bytes consumption in register/FRAG. While this strategy may increase the register pressure, we will carefully select hyperparameters to avoid register spilling in Section 6.

5 Instruction-Level Optimizations

In this section, we propose two instruction-level optimizations to fully exploit Tensor Core computing capability and memory hierarchies.

5.1 Register-Enhanced Instruction Scheduling for Latency Hiding

The first optimization at the SASS level is a register-enhanced instruction scheduling for latency hiding. While latency hiding has been discussed in existing works [5, 16] and can be implemented at the CUDA-level (*e.g.*, with streams), EGEMM-TC has two distinguishing designs. First, to mitigate the limitation on the shared memory size (*e.g.*, 64 KB per SM on Tesla T4), EGEMM-TC intentionally utilizes registers (*e.g.*, with 256 KB per SM on Tesla T4) to exploiting more latency hiding opportunities. Second, EGEMM-TC supports fine-grained data access latency hiding at the instruction-level by breaking down the KB-level data access into a sequence of Byte-level data access and interleaving with individual Tensor Core computation instructions.

On the SASS instructions, we utilize 4 instructions that are widely used in many generations of Nvidia GPUs [12, 13, 26, 29]. In particular, we use the *LDS* instruction to load data from shared memory to registers, the *LDG* instruction for loading data from global memory to registers, the *STS* instruction for storing data from registers to shared memory, and the *HMMA* instruction for computation on Tensor Cores. Note that existing works [15, 39] demystify that the memory instructions (*e.g.*, *LDS*, *LDG*, and *STS*) are executed sequentially and cannot be further paralleled.

Figure 6 illustrates the register-enhanced instruction scheduling for latency hiding. At a high level, EGEMM-TC tensorizes the input matrices into several sub-matrices and processes one sub-matrix at each iteration. Before the first iteration, EGEMM-TC has a "cold-start" that loads the data for the first iteration from global memory to shared memory. On the following iterations, EGEMM-TC simultaneously conducts the computation for the current iteration and the data loading for the next iteration. Assuming that the data for the current iteration has been stored in the shared memory, EGEMM-TC uses *LDS* to load data from shared memory to registers for computation. Meanwhile, EGEMM-TC loads the data for the next iteration. Noting that Nvidia GPUs usually do not support loading data directly from global memory to shared memory, we first load data from global memory to registers and then store to shared memory with *LDG* and *STS*, respectively. Considering that the shared memory stores the data for the current iteration, we delay *STS* to the end of the current iteration to avoid undesired data overwriting. This design enables caching large matrices in registers and provides more latency hiding opportunities for improving performance.

5.2 Register Allocation Design

The second optimization at the SASS level is a manual register allocation to avoid register spilling [33, 35]. To fully exploit the fast register access, we heavily utilize registers for a set of memory-related optimizations. While this register-caching can improve performance theoretically, it also increases the register pressure. Indeed, implementing these optimizations at the CUDA level can easily introduce register spilling, leading to heavy slow down.

While the optimal register allocation has been shown as an NP-problem [32], we propose a heuristic register allocation design for the Tensor-Core centric workload. Our key observation is that these workloads usually contain four stages with different register usage. During the first stage, a large number of registers are utilized on the context information (*e.g.*, threadIdx, blockIdx, and block matrix size) to locate the block matrix for computation. During the following three stages, registers are utilized to load the C matrix from global memory, conducting computation, and saving the C matrix to the global memory. Register allocations across these stages are usually non-overlapping and only utilized in a single

stage. Based on this observation, we manually reuse most registers across stages to reduce the register pressure. In total, we utilize 232 out of 256 registers on each thread for all optimizations mentioned above.

6 Hardware-aware Analytic Model

In this section, we propose an analytic model to facilitate the hyper-parameter selection for achieving high performance. There are 6 hyper-parameters $(b_m, b_n, b_k, w_m, w_n, w_k)$ that have a significant influence on the performance. Selecting larger hyper-parameters generally leads to higher data reuse and lower memory overhead. However, larger hyper-parameters also increase the pressure on the shared memory and the register/FRAG. Moreover, when the value exceeds the capability of FRAG, register spilling will happen, leading to degraded performance.

Existing works [17, 38, 39] usually utilize a trial-and-error strategy to select these hyper-parameters. There are two drawbacks of this strategy. First, experimenting with new tiling sizes usually requires extra manual effort, making it a time-consuming task. Second, there is a large design space of 6 parameters, making it infeasible to enumerate all settings. To this end, we propose a hardware-aware analytic model that takes the small set of hardware resource budgets and selects the parameters without trial-and-error.

6.1 Resource Consumption

At each iteration, each GPU block needs to do two tasks. First, it reads 2 matrices (Alo, Ahi) of size (b_m, b_k) and 2 matrices (Blo, Bhi) of size (b_k, b_n). This step introduces global memory access of

$$(b_m + b_m + b_n + b_n) \times b_k \times 2 = 4(b_m + b_n)b_k \tag{2}$$

Here, the last two comes from half data type (2 bytes). We skip the memory access for block matrices of C since it is only loaded once for every k/b_k times reading of the split matrices and accounts for a negligible portion of memory overhead. Second, EGEMM-TC conducts the computation with FLOPs of

$$2 \times b_m \times b_n \times b_k \times 4 = 8b_m b_n b_k \tag{3}$$

There is a constant 4 since EGEMM takes 4 Tensor Core calls for one extended-precision computation. To this end, the ratio of computation to global memory access is

$$\frac{8 \times b_m \times b_n \times b_k}{4 \times (b_m + b_n) \times b_k} = \frac{2b_m \times b_n}{b_m + b_n} \tag{4}$$

We want to improve this ratio to fully exploit GPU compute capability and achieve compute-bound. Noting that the numerator uses multiplication and the denominator uses addition, we can improve the ratio by choosing a larger b_m and b_n. We surprisingly observe that the ratio is independent of b_k, indicating that we can select a smaller b_k to leave space for storing larger b_m and b_n.

Table 3. Resource Budget on T4 GPU.

Shared Memory Size	64 KB
FRAG/Register Size	256 KB
Peak Computation	2^6 TFLOPS
L2 Cache Speed	750 GB/s

Table 4. Design Choice on T4 GPU

(b_m, b_n, b_k)	(128, 128, 32)
(w_m, w_n, w_k)	(64, 32, 8)
Shared memory/block	36 KB
Active Blocks/SM	1
Active Warps / Block	8

On the memory space, we store a block matrix C of size (b_m, b_n) in the FRAG following the *intra-warp FRAG caching* design. This would consume $b_m \times b_n \times 4 + 2 \times (b_m + b_n) \times b_k \times 2$ bytes in registers. For reducing register pressure, we store the Alo, Ahi, Blo, and Bhi blocks in the shared memory, leading to $2 \times (b_m + b_n) \times b_k \times 2$ bytes shared memory usage.

Inside each warp, we also have the computation and the memory access, determined by the warp tiling size (w_m, w_n, w_k) and the block tiling size (b_m, b_n, b_k). Our design goal is to adjust warp tiling size such that the computation time is larger than the memory access time to achieve the compute-bound. Assuming that each Tensor Core execution takes T_{HMMA} time, the computation time for a block matrix is

$$T_{Comp} = \frac{2b_m b_n b_k \times 4}{2 \times 16 \times 8 \times 8 \times 4} \times T_{HMMA} \tag{5}$$

where 4 in the numerator represents the 4 Tensor Core calls in the emulation, $2 \times 16 \times 8 \times 8$ is the computation done with a *HMMA.1688.F32* Tensor Core instruction, and each block can call 4 tensor cores simultaneously from the hardware perspective [12, 13]. On the memory access time, there are two steps, as described in the *register-caching-based warp collaboration*. First, all warps collaboratively load data from global memory to the shared memory. Denoting $T_{LDG.128}$ as the time for reading 128-bit data from the global memory and $T_{STS.128}$ the time for writing 128-bit data to the shared memory, the memory access time is

$$T_{Mem1} = \frac{(b_m + b_m + b_n + b_n)b_k \times 2}{32 \times 16} \times (T_{LDG.128} + T_{STS.128}) \tag{6}$$

where b_m and b_n are repeated for reading both Alo, Ahi, and Blo, Bhi, 32 is the warp size, and 16 stands for 16 bytes (128 bits). The second step is to load the Alo, Ahi, Blo, and Bhi from shared memory to the FRAG for computing. Denoting $T_{LDS.32}$ as the time for reading 32-bit data from the shared memory, the memory access time is

$$T_{Mem2} = \frac{b_m b_n b_k}{w_m w_n w_k} \times \left(\frac{w_m}{8} + \frac{w_m}{8} + \frac{w_n}{8} + \frac{w_n}{8}\right) \times T_{LDS.32} \tag{7}$$

6.2 Analytic Solver

Our analytic model matches the theoretical resource consumption with the resource budget and transforms the design

Table 5. Baseline Kernels.

Name	Source	Precision	Description
cuBLAS-CUDA-FP32	cuBLAS	single	*cublasSgemm* on CUDA Cores
cuBLAS-TC-Half	cuBLAS	half	*cublasGemmEx* on Tensor Cores
cuBLAS-TC-Emulation	cuBLAS	extended	implement with *cublasGemmEx* on Tensor Cores
SDK-CUDA-FP32	SDK	single	*matrixMul* on CUDA Cores
Markidis	[20]	extended*	implemented Markidis method on Tensor Cores
kMeans	[9]	single	open-source implementation with cublasSgemm on CUDA Cores
kNN	[2]	single	open-source implementation with cublasSgemm on CUDA Cores

space exploration to an optimization problem, which can be solved analytically with existing optimization solvers [1]. To support different GPUs, the user only needs to provide a small set of resource budgets. Table 3 shows the budget on Tesla T4 GPU.

Formally, we have the following optimization problem

$$\begin{aligned} max \quad & \frac{2b_m \times b_n}{b_m + b_n} \\ s.t. \quad & 4b_m b_n + 4(b_m + b_n)b_k \le Size_{Register} \\ & 2 \times (b_m + b_n) \times (b_k + 8) \times 2 \le Size_{SHMEM} \\ & T_{Mem1} + T_{Mem2} \le T_{Comp} \end{aligned} \tag{8}$$

Our goal is to maximize the ratio of computation to global memory access (Equation 4) to fully exploit the computing capability. Meanwhile, we need to make sure that the usage of registers and shared memory does not exceed the corresponding resource budget. In addition, we aim to increase (w_m, w_n) for ensuring that each warp spends more time on computation than memory access, leaving space for latency hiding. Table 4 details our design choice for Tesla T4.

7 Evaluation

In this section, we compare EGEMM-TC with various GEMM kernels and show the benefit of accelerating GEMM-based scientific computing on Tensor Cores.

7.1 Experiment Setup

Baseline Kernels. We compare EGEMM-TC with a diverse set of GEMM kernels and GEMM-based scientific computing benchmarks shown in Table 5. These kernels include cuBLAS kernels running on CUDA Cores and Tensor Cores. We utilize cuBLAS kernel *cublasGemmEx* to implement Algorithm 1 on Tensor Cores, namely *cuBLAS-TC-Emulation*, and compare with EGEMM-TC on the performance benefit of EGEMM-TC optimizations. We also compare the performance with open-source code from CUDA-SDK. Besides, we compare against Markidis [20], the most recent emulation work on Tensor Cores. Note that Markidis has 1-bit lower precision than EGEMM-TC due to the truncate-split, as detailed previously

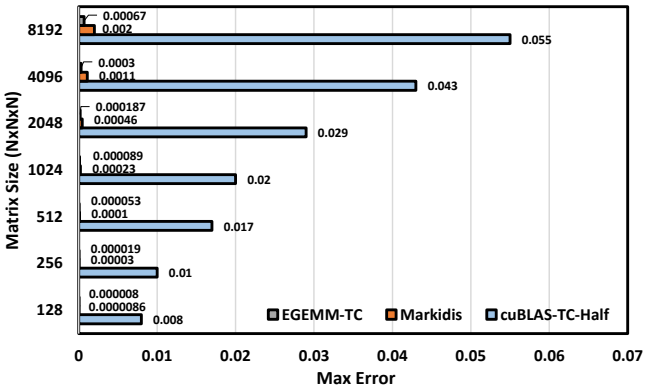


Figure 7. Emulation Precision.

in Table 1. We evaluate on diverse matrix sizes from 1024 to 16384 and report the performance averaged over 10 runs, measured with Trillion Floating Point Operations per Second

$$TFLOPS = 2 \times M \times N \times K/(T \times 10^9) \tag{9}$$

T is the time in milliseconds measured by *cuda event* [28].

We also experiment on two popular scientific computing workloads, kMeans and kNN, that have wide applications in diverse domains (*e.g.*, gene analysis [31], environmental science [19], and astronomy [18]). In particular, we compare with two open-source kernels (kNN [9] and kMeans [2]) on CUDA Cores that implement with *cuBLAS-CUDA-FP32*.

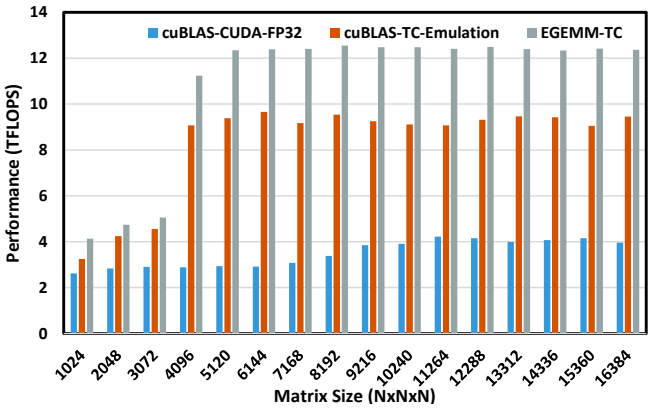
Environments. We evaluate on both Nvidia T4 and Nvidia RTX6000. T4 [24] has 320 Tensor Cores and 16 GB GDDR6 memory. RTX6000 [23] has 576 Tensor Cores and 24 GB GDDR6 memory. The host server has a 32-core Intel Xeon CPU E5-2620 processor and 126 GB memory and runs Ubuntu 18.04 with CUDA 10.1 and cuBLAS 10.1.

7.2 Precision Improvement

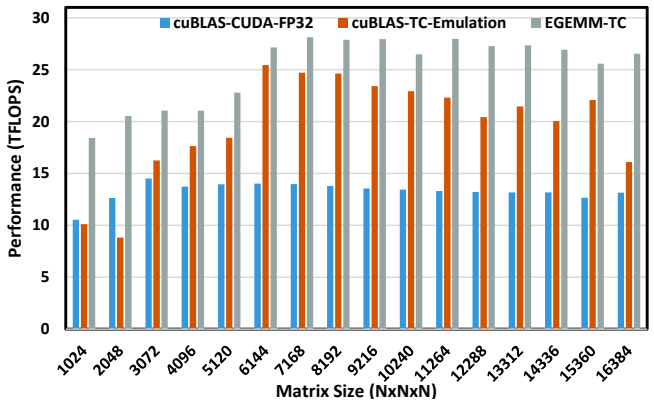
Figure 7 compares the precision of EGEMM-TC and baseline GEMM kernels. We present the max error relative to the single-precision computation

$$MaxError(p) = |V_p - V_{Single}| \tag{10}$$

Here, V_p is the computation results under the precision p, which could be one of the extended-precision, the half-precision, and the single-precision. During the computation, we generate square matrices of size $N \times N \times N$ with values sampled from [-1,+1]. On average, EGEMM-TC effectively reduces the max error by 350× compared to cuBLAS-TC-Half. This result shows the effectiveness of our emulation algorithm in improving the computation precision on Tensor Cores. As the matrix size increases, we observe a slow increase in the max error. The reason is that, a single element in the output matrix involves N additions and N multiplications and the emulation error accumulates as N increases. However, EGEMM-TC still achieves 82× reduction in max

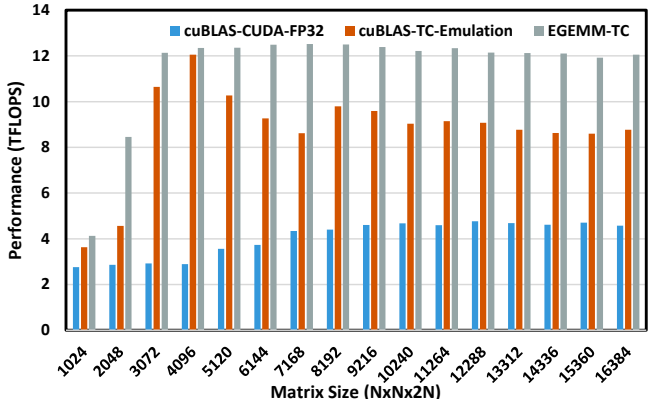


(a) on T4.

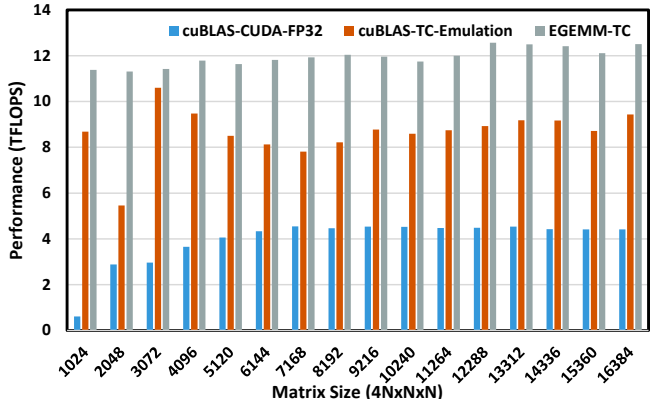


(b) on RTX6000.

Figure 8. Comparison with Vendor Kernels on Square Matrices.



(a) Set matrix shape (M, N, K) as $(N, N, 2N)$.



(b) Set matrix shape (M, N, K) as $(4N, N, N)$.

Figure 9. Comparison with Vendor Kernels on Skewed Matrices.

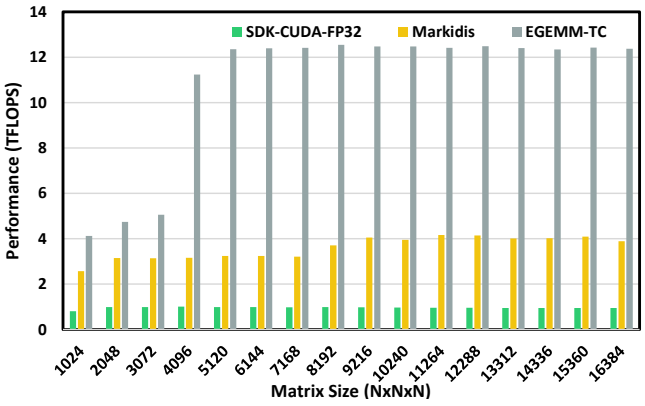


Figure 10. Comparison with Open-Source Kernels.

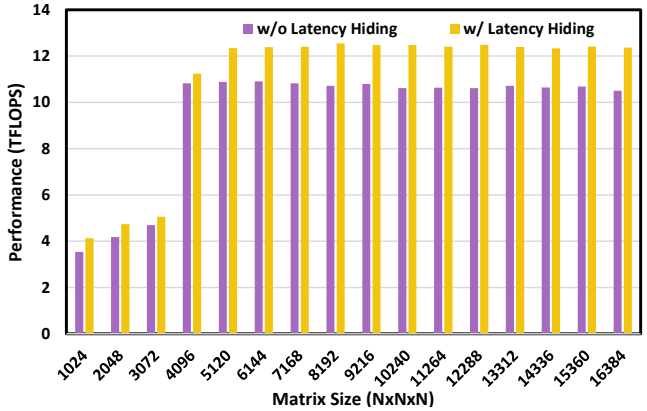
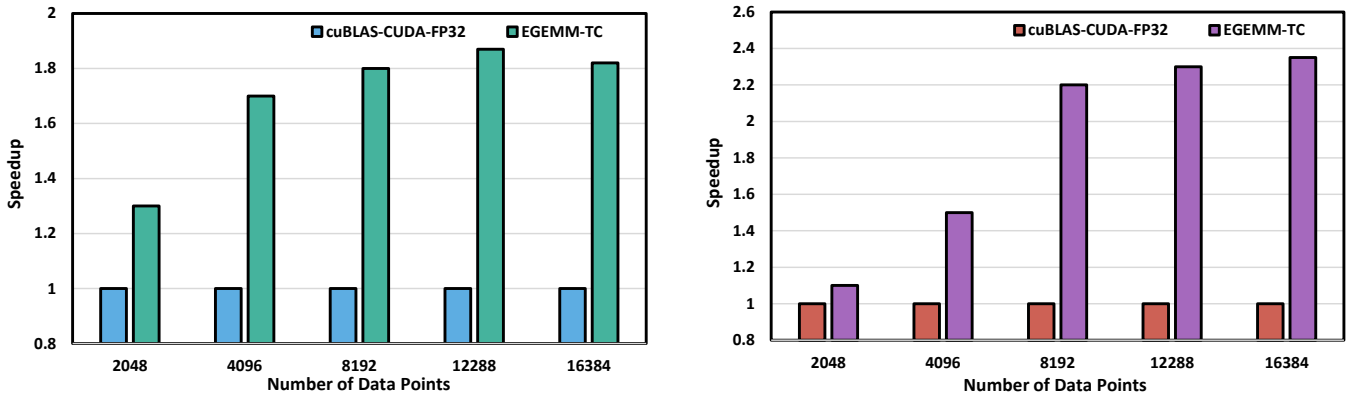


Figure 11. Benefit of Latency Hiding.

error, when computing a large matrix of 8192×8192 with extended-precision. In addition, EGEMM-TC reduces the max error by 2.33× over Markidis, thanks to the round-split algorithm.

7.3 Overall Speedup

Comparison with Vendor Kernels. Figure 8a shows the performance comparison with vendor kernels on Tesla T4. Comparing with *cuBLAS-CUDA-FP32*, EGEMM-TC is faster



(a) kMeans **(b)** kNN

Figure 12. GEMM-based Scientific Computing Acceleration with EGEMM-TC.

by 3.13× on average. This result shows that EGEMM-TC on Tensor Cores can effectively outperform the single-precision GEMM on CUDA Cores by a large margin. This benefit comes from the high-performance half-precision computation on Tensor Cores and our kernel optimizations. Comparing with *cuBLAS-TC-Emulation*, we still observe 1.35× speedup on average. This result shows the effectiveness of our kernel optimizations, considering that cuBLAS provides highly-optimized vendor GEMM kernel. Comparing across matrix sizes, we can see that EGEMM-TC shows a larger speedup at large matrix sizes. The reason is that the GPU capability is not fully utilized at small matrix sizes and the compute-bound has not been achieved. As the matrix size increases, the Tensor Core occupancy also increases and optimizations for reducing memory movement start to show benefit. We show the performance comparison on Nvidia RTX6000 in Figure 8b, where EGEMM-TC has similar benefits as the case on Tesla T4. Since similar patterns show on Telta T4 and RTX6000, we will only show the results on Tesla T4 in the following experiments.

Figure 9 shows the performance comparison on skewed matrices, where dimensions K and M are larger than the remaining dimensions by 2× and 4×, respectively. We skip dimension N since it can be viewed as dimension M under matrix transpose. When dimension K is enlarged, we observe that the *cuBLAS-TC-Emulation* exhibits significant slowdown when the matrix size exceeds $4096 \times 4096 \times 8192$. Instead, EGEMM-TC consistently provides high performance across different matrix sizes. In this case, EGEMM-TC provides 1.33× speedup over *cuBLAS-TC-Emulation* and 2.89× speedup over *cuBLAS-CUDA-FP32*. When dimension M is enlarged, *cuBLAS-TC-Emulation* achieves higher performance but is still much slower than EGEMM-TC. Under this setting, our GEMM are 1.40× faster than *cuBLAS-TC-Emulation* and 2.9× faster than *cuBLAS-CUDA-FP32* on average.

Comparison with Open-Source Kernels. Figure 10 shows the performance comparison with the open-source kernels. Comparing with *SDK-CUDA-FP32*, EGEMM-TC is faster by 11.18× on average. This result shows the significant performance improvement from EGEMM-TC on Tensor Cores. EGEMM-TC is also faster than *Markidis* by 3.0× on average. We manually tune *Markidis* performance with our optimizations on the hand-written CUDA code, but the performance remains similar. The reason is that the CUDA programming interface provides limited control over the GPU hardware while our implementation-level optimizations with the SASS programming interface can utilize GPU capability to much larger extent (*e.g.*, register-enhanced instruction scheduling).

7.4 Benefit of Instruction Scheduling

Figure 11 shows the performance benefits of instruction scheduling in latency hiding. In this optimization, we focus on the SASS programming interface and switch orders of computation and memory access instructions for latency hiding. The instruction scheduling can achieve 1.14× speedup on average. Comparing with the latency hiding on the CUDA programming interface, we can achieve more fine-grained latency hiding with the SASS programming interface. For example, loading data from global memory to shared memory is a single instruction with the CUDA programming interface but two instructions with the SASS programming interface (*i.e.*, loading to register from global memory and storing from registers to shared memory). This provides more opportunities to interleave the memory access instructions with the compute instructions.

7.5 Scientific Computing Acceleration

Figure 12 shows the speedup of scientific computing based on EGEMM-TC over cuBLAS-CUDA-FP32. We observe an average speedup of 1.9× on kMeans and an average speedup of 1.7× on kNN. These speedups show that EGEMM-TC can be effectively utilized to accelerate GEMM-based scientific computing. Comparing across data sizes, EGEMM-TC accelerates kMeans by 1.3× when there are 2048 data points and accelerates kMeans by 1.82× when there are 16384 data points, as shown in Figure 12a. There are two reasons. First,

EGEMM-TC shows a larger speedup than cuBLAS-CUDA-FP32 when the data size increases, as shown previously in Figure 8. Second, when data size increases, GEMM accounts for more running time and the acceleration on GEMM shows more benefits. We also observe similar trends on the kNN workload (Figure 12b).

8 Conclusion

In this paper, we design and implement EGEMM-TC that accelerates general-purpose scientific computing on Tensor Cores with extended-precision. Specifically, EGEMM-TC contains a lightweight emulation algorithm on Tensor Cores to achieve the extended-precision computation, a set of Tensor Core kernel optimizations to efficiently map the workloads to Tensor Cores, and a hardware-aware analytic model to facilitate the selection of performance-related hyper-parameters. Overall, EGEMM-TC achieves 3.13× and 11.18× speedup on average over the single-precision kernels on CUDA Cores from cuBLAS and CUDA-SDK, respectively. EGEMM-TC also achieves 1.8× speedup on a set of popular GEMM-based scientific computing workloads and diverse input sizes.

References

[1] Martin S. Andersen, Joachim Dahl, and Lieven Vandenberghe. 2020. Convex Optimization Solver. https://cvxopt.org/.

[2] angelhof. 2017. Hipeac GPUs K-means. https://github.com/angelhof/gpus-kmeans.git.

[3] D. H. Bailey. 2005. High-precision floating-point arithmetic in scientific computation. *Computing in Science Engineering* 7, 3 (2005), 54–61.

[4] Ron Banner, Yury Nahshan, and Daniel Soudry. 2019. Post training 4-bit quantization of convolutional networks for rapid-deployment. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc., Vancouver, 7950–7958.

[5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*. USENIX Association, USA, 579–594.

[6] Abdul Dakkak, Cheng Li, Jinjun Xiong, Isaac Gelado, and Wen-mei Hwu. 2019. Accelerating Reduction and Scan Using Tensor Core Units. In *Proceedings of the ACM International Conference on Supercomputing (ICS '19)*. Association for Computing Machinery, New York, NY, USA, 46–57. https://doi.org/10.1145/3330345.3331057

[7] T.J. Dekker. 1971/72. A Floating-Point Technique for Extending the Available Precision. *Numer. Math.* 18 (1971/72), 224–242. http://eudml.org/doc/132105

[8] F. D. Dinechin and G. Villard. 2006. High precision numerical accuracy in physics research. *Nuclear Instruments and Methods in Physics Research Section A-accelerators Spectrometers Detectors and Associated Equipment* 559 (2006), 207–210.

[9] Vincent Garcia, Eric Debreuve, Frank Nielsen, and Michel Barlaud. 2010. K-nearest neighbor search: Fast GPU-based implementations and application to high-dimensional feature matching. In *Proceedings of the International Conference on Image Processing 2010*. IEEE, Hong Kong, China, 3757–3760. https://doi.org/10.1109/ICIP.2010.5654017

[10] Sridhar Gopinath, Nikhil Ghanathe, Vivek Seshadri, and Rahul Sharma. 2019. Compiling KB-Sized Machine Learning Models to Tiny IoT Devices. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 79–95. https://doi.org/10.1145/3314221.3314597

[11] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2018. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. CVPR, Salt Lake City, Utah, 2704–2713.

[12] Zhe Jia, Marco Maggioni, Jeffrey Smith, and Daniele Paolo Scarpazza. 2019. Dissecting the NVidia Turing T4 GPU via Microbenchmarking. arXiv:1903.07486 http://arxiv.org/abs/1903.07486

[13] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele Scarpazza. 2018. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking.

[14] Donald E. Knuth. 1997. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms* (third ed.). Addison-Wesley, Boston.

[15] Junjie Lai and André Seznec. 2013. Performance upper bound analysis and optimization of SGEMM on Fermi and Kepler GPUs. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society, Shenzhen, China, 4:1–4:10. https://doi.org/10.1109/CGO.2013.6494986

[16] Shin-Ying Lee and Carole-Jean Wu. 2014. Characterizing the latency hiding ability of GPUs. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2014*. IEEE Computer Society, Monterey, CA, USA, 145–146. https://doi.org/10.1109/ISPASS.2014.6844477

[17] Xiuhong Li, Yun Liang, Shengen Yan, Liancheng Jia, and Yinghan Li. 2019. A Coordinated Tiling and Batching Framework for Efficient GEMM on GPUs. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming (PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 229–241. https://doi.org/10.1145/3293883.3295734

[18] Yanxia Zhang LiLi Li and YongHeng Zhao. 2008. k-Nearest Neighbors for automated classification of celestial objects. In *Sci. China Ser. G-Phys. Mech. Astron.*, Vol. 51. Springer, China, 916–922.

[19] K. Lin, L. Jing, M. Wang, M. Qiu, and Z. Ji. 2017. A novel long-term air quality forecasting algorithm based on kNN and NARX. In *2017 12th International Conference on Computer Science and Education (ICCSE)*. IEEE, Beijing, China, 343–348.

[20] S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng, and J. S. Vetter. 2018. NVIDIA Tensor Core Programmability, Performance Precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE Computer Society, Vancouver, British Columbia, CANADA, 522–531.

[21] NVIDIA. 2017. Programming Tensor Cores in CUDA 9. https://devblogs.nvidia.com/programming-tensor-cores-cuda-9/.

[22] NVIDIA. 2017. Tensor Core Performance. https://www.nvidia.com/en-us/data-center/volta-gpu-architecture/.

[23] NVIDIA. 2018. Nvidia RTX 6000. https://www.nvidia.com/en-us/design-visualization/quadro/rtx-6000/.

[24] NVIDIA. 2018. Nvidia T4. https://www.nvidia.com/en-us/data-center/tesla-t4/.

[25] NVIDIA. 2020. cuBLAS: CUDA Toolkit Documentation. https://docs.nvidia.com/cuda/cublas/index.html.

[26] NVIDIA. 2020. CUDA Binary Utilities. https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html#instruction-set-ref.

[27] NVIDIA. 2020. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[28] NVIDIA. 2020. CUDA Event. https://devblogs.nvidia.com/how-implement-performance-metrics-cuda-cc/.

[29] NVIDIA. 2020. PTX and SASS Assembly Debugging. https://docs.nvidia.com/gameworks/content/developertools/desktop/ptx_sass_assembly_debugging.htm.

[30] Institute of Electrical and Electronics Engineers. 1985. IEEE Standard for Binary Floating Point Arithmetic.

[31] et al. Parry, R.M. 2010. k-Nearest neighbor models for microarray gene expression analysis and clinical outcome prediction. *The Pharmacogenomics Journal* 10, 4 (2010), 292.

[32] Fernando Magno Quintão Pereira and Jens Palsberg. 2006. Register Allocation after Classical SSA Elimination is NP-Complete. In *Proceedings of the 9th European Joint Conference on Foundations of Software Science and Computation Structures (FOSSACS'06)*. Springer-Verlag, Berlin, Heidelberg, 79–93. https://doi.org/10.1007/11690634_6

[33] Shlomit S. Pinter. 1993. Register Allocation with Instruction Scheduling. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation (PLDI '93)*. Association for Computing Machinery, New York, NY, USA, 248–257. https://doi.org/10.1145/155090.155114

[34] Douglas M. Priest. 1992. *On Properties of Floating Point Arithmetics: Numerical Stability and the Cost of Accurate Computations*. Technical Report. University of California, Berkeley.

[35] Fernando Magno Quintão Pereira and Jens Palsberg. 2008. Register Allocation by Puzzle Solving. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 216–226. https://doi.org/10.1145/1375581.1375609

[36] Jonathan Richard Shewchuk. 1997. Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. *Discrete & Computational Geometry* 18, 3 (Oct. 1997), 305–363.

[37] Da Yan, Wei Wang, and Xiaowen Chu. 2020. Demystifying Tensor Cores to Optimize Half-Precision Matrix Multiply. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, New Orleans, LA, USA, 634–643. https://doi.org/10.1109/IPDPS47924.2020.00071

[38] Xiuxia Zhang, Guangming Tan, Shuangbai Xue, Jiajia Li, Keren Zhou, and Mingyu Chen. 2017. Understanding the GPU Microarchitecture to Achieve Bare-Metal Performance Tuning. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '17)*. Association for Computing Machinery, New York, NY, USA, 31–43. https://doi.org/10.1145/3018743.3018755

[39] Xiuxia Zhang, Guangming Tan, Shuangbai Xue, Jiajia Li, Keren Zhou, and Mingyu Chen. 2017. Understanding the GPU Microarchitecture to Achieve Bare-Metal Performance Tuning. *SIGPLAN Not.* 52, 8 (Jan. 2017), 31–43. https://doi.org/10.1145/3155284.3018755

A Artifact Evaluation Appendix

A.1 Abstract

EGEMM-TC extends the usage of Tensor Cores to accelerate scientific computing applications without compromising the precision requirement. We provide artifacts to support following claims:

- **Profiling:** We have designed an extendable workflow of hardware profiling. It shows that the intermediate results of Tensor Core computation are identical to floating-point results bit-wisely up to 21 mantissa bits, which is required by the extended-precision computation.
- **Precision:** We have designed a lightweight emulation algorithm to achieve both extended-precision and low overhead by exploiting high-precision intermediate computation results.
- **Performance:** We achieve high performance with a set of Tensor Core kernel optimizations and a hardware-aware analytic model, compared with the cuBLAS kernel and the CUDA-SDK kernel on CUDA Cores.

A.2 Getting Started Guide

Our artifacts include the related CUDA code, SASS code, python code for SASS compilation, and a Makefile.

GPU Requirement. Due to the setting of the SASS compiler, currently Nvidia GPUs with Turing architecture are required to compile and evaluate the SASS code. We suggest the T4 GPU following our evaluation setting. While the code can also run on other Nvidia GPUs with Turing architecture, there would be slightly different performance due to the different peak performance across GPUs. However, we can still expect similar precision improvement and performance improvement compared with the baseline methods.

Environment Setting. We compile the artifacts on Ubuntu 16.04.6 LTS machine with *nvcc 10.1* and Python 3.7.3. To compile the SASS code, a SASS assembler, namely *TuringAs* is required. TuringAs is a python package that takes SASS code and generates the compiled executable code. We suggest to use Anaconda to install TuringAs and related python packages. All the installations can be conducted at the user level and do not need root privilege. The first step is to install Anaconda. Please download Anaconda3 with the following command "wget https://repo.anaconda.com/archive/Anaconda3-2019.03-Linux-x86_64.sh". Anaconda3 can be installed with

$$sh\ Anaconda3\text{-}2019.03\text{-}Linux\text{-}x86\_64.sh$$

It also needs to be specified in the bash shell

$$eval\ "\$(\$ANACONDA\_PATH/bin/conda\ shell.bash\ hook)"$$

The second step is to install the python package *setuptools* with the following command

$$conda\ install\ setuptools$$

Finally, we can install the TuringAs. To install the TuringAs, please git clone from the following repository

$$https://github.com/daadaada/turingas.git$$

Then please change into the turingas directory.

$$cd\ turingas$$

The TuringAs can be installed with the following command

$$python3\ setup.py\ install$$

Compilation and Evaluation. We provide a *Makefile* to simplify the compilation procedure. The *Makefile* contains five options to support our claims.

- *precision_profiling*: compiling the code for profiling.
- *precision_test*: compiling the code for precision improvement with the emulation algorithm. The *TuringAs* is required.
- *main_emulation*: compiling the code for SASS emulation. The *TuringAs* is required.
- *cublas_CUDA_FP32*: compiling the baseline for the single-precision cuBLAS kernel on CUDA Cores.
- *SDK_CUDA_FP32*: compiling the baseline for the single-precision CUDA-SDK kernel on CUDA Cores.
- *clean*: remove compiled programs.

A simple test for the environment setting and the SASS compilation is to run the following command:

$$make\ main\_emulation;\ ./emulation$$

A successful compilation and run should provide the throughput of the SASS emulation code on $8192 \times 8192 \times 8192$ square matrices. We will detail the commands and expected outputs for individual claims in the following section.

Typical Errors One typical error comes from compilation. There would be an error message during compilation

$$/usr/bin/python:\ No\ module\ named\ turingas$$

This error indicates the TuringAS is not installed successfully. Please refer to the *Environment Setting* paragraph for more details.

If the SASS code can be compiled correctly, one possible error is

$$Segmentation\ fault\ (core\ dumped)$$

This error may be encountered when compiling and running the SASS code on GPUs without Turing architecture (*e.g.*, V100 and Titan Xp). Please refer to the *GPU Requirement* paragraph for more details.

A.3 Step-by-Step Instructions

On **Profiling**, please run the following commands:

make precision_profiling; *./precision_profiling*

This program provides a precision profiling on Tensor Cores to support our claim on the precision of intermediate results. Typical outputs are

```
half_result: 926.00000000, 0x00806744
single_result: 934.40637207, 0x029a6944
Tensor Core : 934.40631104, 0x019a6944
```

This program takes randomly generated half-precision data and prints the intermediate results and their hex representation. "half_result" indicates the computation results with half precision. "single_result" indicates the computation results with single precision. "Tensor Core" indicates the computation results on Tensor Cores. In this example, the "Tensor Core" and "single_result" differ with only 1 bit. By repeating the profiling for 10,000 times, we empirically observe that the "Tensor Core" and "single_result" are bitwisely identical for upto 21 mantissa bits.

On **Precision**, please run the following commands:

make precision_test; *./precision_test*

This program shows the emulation error and the half-precision cuBLAS error. In particular, we the max error relative to the single precision computation Typical outputs are

```
m*n*k: 1024.
max Emulation Error: 0.00025177
max Half cuBLAS Error: 0.13489914
Ratio (Max_Emulation_Error/Max_Half_cuBLAS_Error): 0.00186636
```

In this example, we evaluate a matrix of size 1024. The half-precision cuBLAS kernel shows an error of 0.13489914. Our kernel shows a significantly reduced error of 0.00025177. The ratio shows that the error is reduced by more than 500×. For the same matrix size, the absolute error may be different across runs due to the randomness in matrix initialization while the ratio should be similar. For different matrix sizes, we observe a slightly larger error for large matrix sizes due to the error accumulation, as discussed in Section 7.2.

On **Performance**, please run the following commands:

make main_emulation; *./emulation*

This program runs our SASS emulation code on a square matrix of size 8192×8192×8192. Then, please run the baseline for the single precision (FP32) cuBLAS kernel on CUDA Cores with the following commands:

make cublas_CUDA_FP32; *./cublas_CUDA_FP32*

Finally, please run the baseline for the single precision (FP32) CUDA-SDK kernel on CUDA Cores with the following commands:

make SDK_CUDA_FP32; *./SDK_CUDA_FP32*

On T4 GPU, the performance of the emulation code, the cublas_CUDA_FP32 code, the SDK_CUDA_FP32 code are around 12 TFLOPs, 4 TFLOPs, and 1 TFLOPs, repectively. This results show the significant performance improvement compared with the single precision cuBLAS kernel and the single-precision CUDA SDK kernel on CUDA Cores, supporting our claim on the performance.