

# Reducing shared memory footprint to leverage high throughput on Tensor Cores and its flexible API extension library

Hiroyuki Ootomo  
Tokyo Institute of Technology  
Tokyo, Japan  
ootomo.h@rio.gsic.titech.ac.jp

Rio Yokota  
Tokyo Institute of Technology  
Tokyo, Japan

## Abstract

Matrix-matrix multiplication is used for various linear algebra algorithms such as matrix decomposition and tensor contraction. NVIDIA Tensor Core is a mixed-precision matrix-matrix multiplication and addition computing unit, where the theoretical peak performance is more than 300 TFlop/s on NVIDIA A100 GPU. NVIDIA provides WMMA API for using Tensor Cores in custom kernel functions. The most common way to use Tensor Core is to supply the input matrices from shared memory, which has higher bandwidth than global memory. However, the Bytes-per-Flops (B/F) ratio of the shared memory and Tensor Cores is small since the performance of Tensor Cores is high. Thus, it is important to reduce the shared memory footprint for efficient Tensor Cores usage. In this paper, we analyze the simple matrix-matrix multiplication on Tensor Cores by the roofline model and figure out that the bandwidth of shared memory might be a limitation of the performance when using WMMA API. To alleviate this issue, we provide a WMMA API extension library to boost the throughput of the computation, which has two components. The first one allows for manipulating the array of registers input to Tensor Cores flexibly. We evaluate the performance improvement of this library. The outcome of our evaluation shows that our library reduces the shared memory footprint and speeds up the computation using Tensor Cores. The second one is an API for the SGEMM emulation on Tensor Cores without additional shared memory usage. We have demonstrated that the single-precision emulating batch SGEMM implementation on Tensor Cores using this library achieves 54.2 TFlop/s on A100 GPU, which outperforms the theoretical peak performance of FP32 SIMT

Cores while achieving the same level of accuracy as cuBLAS. The achieved throughput can not be achieved without reducing the shared memory footprint done by our library with the same amount of register usage.

**CCS Concepts:** • Software and its engineering → Software libraries and repositories.

**Keywords:** Tensor Cores, WMMA API, GPU

## ACM Reference Format:

Hiroyuki Ootomo and Rio Yokota. 2023. Reducing shared memory footprint to leverage high throughput on Tensor Cores and its flexible API extension library. In *International Conference on High Performance Computing in Asia-Pacific Region (HPC ASIA 2023)*, February 27-March 2, 2023, Singapore, Singapore. ACM, New York, NY, USA, 9 pages. <https://doi.org/10.1145/3578178.3578238>

## 1 Introduction

NVIDIA Tensor Core is a mixed-precision matrix multiplication and addition computing unit with up to 312 TFlop/s on NVIDIA A100 GPU [2]. From the demand for high-throughput matrix multiplication from deep learning, several computing units specialized for matrix multiplication are developed, such as Google TPU [7], AMD Matrix Core, Intel Ponte Vecchio, and Preferred Networks MN-Core [9]. Tensor Core computes the multiplication of two matrices where the data type is low-precision in high throughput and high-precision. Although Tensor Core is developed for deep learning, especially fully-connected layer and convolution layer computations, it is applied to other fields of computations and fundamental linear algebra algorithms leveraging the low- and mixed-precision feature [3–5, 8, 10, 11]. NVIDIA provides highly optimized libraries for using Tensor Cores which can be called from a host, such as cuBLAS and cuDNN. We can leverage the high throughput of Tensor Core using these libraries without special knowledge of it. Furthermore, NVIDIA also provides an API for use inside a CUDA kernel function called WMMA (Warp Matrix Multiply Accumulate) API. This API provides basic functionalities such as loading matrix data from memory, multiplication and addition on Tensor Core, and storing the resulting matrix data in memory. Using this API, we load matrix data from the device memory or shared memory to an array of registers called “fragment” to input

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

HPC ASIA 2023, February 27-March 2, 2023, Singapore, Singapore

© 2023 Association for Computing Machinery.

ACM ISBN 978-1-4503-9805-3/23/02...\$15.00

<https://doi.org/10.1145/3578178.3578238>

Tensor Cores. On the other hand, there are some matrices where each element can be computed on the fly, for instance, the Householder matrix and Given’s rotation matrix. Even for these matrices, we have to store them in memory and load them since the API is too simple and lacks flexibility, and this can degrade the throughput. Thus, for instance, Dakkak *et al.* [3] use Tensor Cores for reduction and scan operation by generating a fragment of an upper triangular matrix and a lower triangular matrix without generating the matrices on the shared memory. Li *et al.* [8] use Tensor Cores in FFT operations by generating fragments directly. However, since NVIDIA does not provide information on fragment mapping, we need to analyze the structure of the fragment by ourselves to generate the fragment in these ways. For another example, the single-precision matrix-matrix multiplication emulation method on Tensor Cores [11] accesses the shared memory more than necessary if we only use the WMMA API. Therefore, the throughput of the emulation method can degrade if we only use the API.

In this paper, we first show that it is important to reduce the shared memory footprint to leverage the high Tensor Cores performance. We analyze a matrix-matrix multiplication on Tensor Cores using the roofline model [12]. As a result, it is difficult to leverage the high Tensor Cores performance without sufficient register blocking or reducing the shared memory footprint. However, the number of registers is limited. To reduce the shared memory footprint, we implement a WMMA API extension library, which flexibly manipulates the input register array of Tensor Cores by analyzing the memory and register array mappings. This library can generate an arbitrary input register array without an extra shared memory footprint. Furthermore, we provide an API for single-precision matrix-matrix multiplication emulation on Tensor Cores, which has the same interface as WMMA API. Our goals of this work are 1) to reveal that the shared memory bandwidth can degrade the utilization efficiency of Tensor Cores in some cases and 2) to provide a library to reduce such degradation by manipulating the fragment flexibly for reducing the shared memory footprint.

Our contributions are as follows:

- We show that the shared memory bandwidth might limit the matrix-matrix multiplication performance on Tensor Cores by roofline model analysis. Furthermore, we find it important to reduce the shared memory footprint on NVIDIA A100 compared to V100 since the Bytes-per-Flops (B/F) ratio of the Tensor Core performance and shared memory bandwidth on NVIDIA A100 is smaller than V100.
- We implement a general WMMA API extension library to reduce the shared memory footprint. By using this library, we can manipulate the fragment elements flexibly. And as a secondary effect, we can reduce the shared memory usage in some cases since some of the

	V100 (SXM2)	V100S (PCIe)	A100 (SXM4/PCIe)	
SMs	80		108	
Clock [MHz]	1,380	1,597	1,410	
<b>Device memory</b>				
Size [GB]	32/16	32	40	80
Bandwidth [GB/s]	900	1,134	1,555	2,039
<b>Shared memory</b>				
Size [KB/SM]	~96		~164	
Bandwidth [GB/s]	14,131	16,353	19,491	
<b>Performance</b>				
FP32 [TFlop/s]	15.7	16.4	19.5	
FP16 [TFlop/s]	31.4	32.8	39.0	
FP16-TC [TFlop/s]	112	125	312	
TF32-TC [TFlop/s]	-	-	156	

**Table 1.** Specifications of NVIDIA GPUs A100 and V100.

temporary shared memory areas for generating matrices that are loaded as fragments become unnecessary. We investigate the availability of this library and find the condition to speed up the fragment generation. The library is available on GitHub<sup>1</sup>.

- We figure out that by the inflexibility of WMMA API, shared memory bandwidth bounds the theoretical peak performance of single-precision matrix-matrix multiplication emulation on Tensor Cores. By using our extension library, we improve its theoretical peak performance. Furthermore, we provide functionality for that which has the same interface as WMMA API. To demonstrate the usability of the functionality, we implement batched matrix-matrix multiplication using the functionality. We show that our implementation outperforms the FP32 theoretical peak performance on NVIDIA A100 while the accuracy is the same level as cuBLAS SGEMM.

## 2 Background

### 2.1 Shared memory

**2.1.1 The bandwidth of shared memory.** The shared memory is a high bandwidth, low latency, and small size compared to the device memory. This memory is located on each Streaming Multiprocessor (SM) and shared by all threads in a thread block. The shared memory is divided into the same size memory modules called banks. In CUDA, a cluster of threads consisting of 32 threads is called a warp, and when multiple threads in a warp access the same bank and different addresses, it is called bank conflict. Since bank conflict degrades read/write performance, there are known

<sup>1</sup>[https://github.com/wmmae/wmma\\_extension](https://github.com/wmmae/wmma_extension)

workarounds, such as shifting the boundaries of shared memory. We show the specifications of NVIDIA Tesla V100 and A100 in Table 1. The shared memory bandwidth is calculated assuming it is accessed without bank conflict in all SMs in one clock. The shared memory has 12 ~ 15 times faster bandwidth than device memory.

### 2.1.2 The advantage of fewer shared memory usage.

The shared memory size that one thread block uses is one of the determining factors of occupancy, which is the max thread block size that one SM executes simultaneously. Fewer shared memory usage means higher occupancy, which effectively hides instruction latency. Furthermore, reducing shared memory usage can improve the L1 cache hit rate since shared memory and L1 cache resides in the same part of the chip.

### 2.2 Blocking for matrix-matrix multiplication

The number of operations of matrix-matrix multiplication  $C \leftarrow A \cdot B$  for  $A \in \mathbb{R}^{m \times k}$ ,  $B \in \mathbb{R}^{k \times n}$  is  $2mnk$ . On the other hand, the sum of the number of elements in  $A$  and  $B$  is  $(m+n) \times k$ . It follows  $2mnk > (m+n) \times k$  in general ( $m \geq 2, n \geq 2, k \geq 1$ ), which means that the number of operations is larger than the number of data. Thus, data can be reused during the computation. When computing the matrix-matrix multiplication on device memory, we copy the sub-matrices of each input matrix from device memory to shared memory. Then compute the matrix-matrix multiplication of these sub-matrices on shared memory to reduce the device memory footprint using data reusability. This method of reducing the low-bandwidth memory footprint by utilizing the memory hierarchy is called “blocking”. The registers are also used for blocking the shared memory. In this paper, we denote the blocking size  $(m_b, n_b, k_b)$  as the size of blocking size for sub-matrices matrix-matrix multiplication  $A_b \cdot B_b$  where the sizes of matrix  $A_b$  and  $B_b$  are  $m_b \times k_b$  and  $k_b \times n_b$  respectively.

### 2.3 Tensor Cores

Tensor Cores are specialized computing units for mixed-precision matrix-matrix multiplication and addition, with higher computing performance than FP16 and FP32 computing units shown in Table 1. We show the supported input and output data types of Tensor Core in Figure 1. We can use the TF32 (Tensor Float) data type, 8 bits of exponent and 10 bits of mantissa, and Bfloat16, 8 bits of exponent and 7 bits of mantissa, as inputs to Tensor Cores in Ampere architectures. While TF32 has 19 bits in total, it occupies a 32-bit register and memory. Thus, it can not be used for data compression.

**2.3.1 Programming interface.** To use Tensor Cores in custom functions, NVIDIA provides WMMA API for C++ and Parallel Thread Execution (PTX). When computing matrix-matrix multiplication and addition  $D \leftarrow A \cdot B + C$  on Tensor Cores using WMMA API for C++, first, we copy the input matrices  $A, B$  and,  $C$  from memory to an array of registers

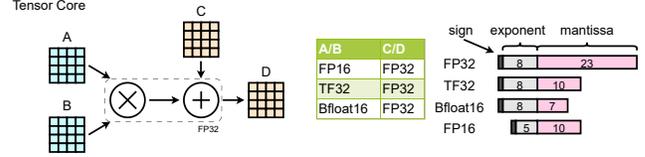


Figure 1. The input and output types of Tensor Cores on NVIDIA A100.

called “fragment”. Then, we compute Matrix-Multiplication-and-Add (MMA) on the Tensor Cores and obtain the resulting  $D$  fragment. The 32 threads in a warp cooperate to perform MMA operations on Tensor Cores. Finally, we store the  $D$  fragment in memory. The WMMA API provides the fragment and functions for these operations. The fragment is a C language structure that has an array of registers  $x[\text{num\_elements}]$  as a member. We show the pseudocode of simple matrix-matrix multiplication using WMMA API in Code 1.

```

1  __device__
2  void matmul(float* mem_c, half* mem_a, half* mem_b
3  ) {
4      using namespace nvcuda::wmma;
5      fragment<matrix_a, 16, 16, 16, half, col_major>
6      frag_a;
7      fragment<matrix_b, 16, 16, 16, half, col_major>
8      frag_b;
9      fragment<accumulator, 16, 16, 16, float> frag_c;
10     // Initialize an accumulator fragment
11     fill_fragment(frag_c, 0.f);
12     // Load matrices to fragments
13     load_matrix_sync(frag_a, mem_a, ...);
14     load_matrix_sync(frag_b, mem_b, ...);
15     // Compute matrix-matrix multiplication
16     // and accumulation on Tensor Cores
17     mma_sync(frag_c, frag_a, frag_b, frag_c);
18     // Store result to memory
19     store_matrix_sync(mem_c, frag_c, ...);
20 }

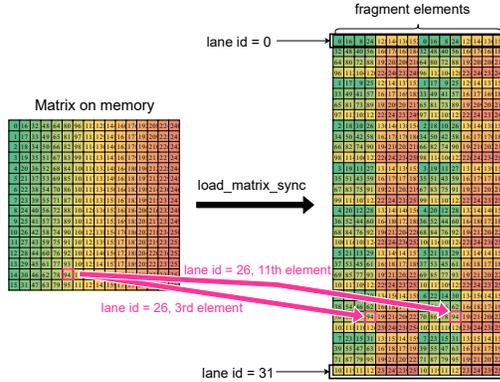
```

Code 1. A simple matrix-matrix multiplication on Tensor Cores using WMMA API.

Although the `load_matrix_sync` function in WMMA API can generate a fragment from the device and shared memory, we consider that the shared memory is used in most cases for the following reasons:

- The shared memory is used for memory blocking in matrix-matrix multiplication.
- The `load_matrix_sync` function has a 128-bit alignment restriction and leading dimension size restriction. It is difficult to satisfy the restriction on device memory.

The fragment is regarded as a register blocking. WMMA API specifies the blocking size of one fragment. For instance, in the case of FP16-Tensor Core, the blocking size  $(m_b, n_b, k_b)$



**Figure 2.** An example of memory-fragment mapping. The lane id is a thread number in a warp which is calculated by  $(\text{threadIdx.x} \& \text{0x1f})$ .

is one of the (16, 16, 16), (32, 8, 16) or (8, 32, 16). We can use the array of fragments to increase the blocking size.

**2.3.2 Mapping between memory and fragment.** Each matrix element in memory is stored as an element of a fragment of some thread. Although the mapping between memory and fragment elements is not public, we can investigate it [3, 6, 8]. This mapping depends on the type, memory layout, etc, of the matrix. We use Code 2 to investigate the mapping and show an example of the mapping in Figure 2.

```

1 template <class Use, class Layout, class T>
2 __global__ void investigate_mapping() {
3     __shared__ T smem[];
4     // initialize smem
5     for (i = 0; i < 16 * 16; i++) smem[i] = i;
6     fragment<Use, 16, 16, 16, T, Layout> frag;
7     load_matrix_sync(frag, smem, ...); // WMMA API
8     for (i = 0; i < 32; i++) {
9         if (threadIdx.x == i) {
10            for (j = 0; j < frag.num_elements; j++) {
11                // Print the mapping
12                printf("%d, ", (int)frag.x[j]);
13                printf("\n");
14            }
15        }
16    }
17 }

```

**Code 2.** A kernel function to investigate the memory-fragment mappings.

**2.3.3 WMMA API for PTX.** The WMMA API for PTX provides two types of instructions: 1) wmma instructions and 2) mma instruction. The WMMA API for C++ functions calls wmma instructions using inline assembly. The wmma instructions include functionality for loading and storing fragments and MMA operation. On the other hand, mma instruction only includes MMA operation. Thus, when using mma instruction, we must manually load fragments from memory. The mapping is available on CUDA developer

documentation. There is a difference between the wmma instructions and the mma instruction regarding register usage. When using wmma instructions, one element in a matrix is kept by two elements in a fragment in 32 threads in a warp. On the other hand, when using mma instruction, one element in a matrix is kept by only one element in a fragment in 32 threads in a warp without duplication. Thus, the mma instruction computes MMA operation using fewer registers than the wmma instructions.

### 3 The balance of Tensor Cores performance and shared memory bandwidth

Although the shared memory bandwidth is higher than device memory, the computing performance of the Tensor Cores is high, and its Bytes-per-Flops (B/F) ratio is calculated to be 0.06 ~ 0.12 from Table 1. This value is similar to the ratio between the FP32 computing unit and device memory (0.06 ~ 0.10). In the case of the FP32 computing unit and device memory, the memory blocking using shared memory reduces global memory access and alleviates the problem of this small B/F ratio. Similarly, in the case of shared memory and Tensor Cores, it is important to reduce shared memory accesses to take advantage of high computational performance.

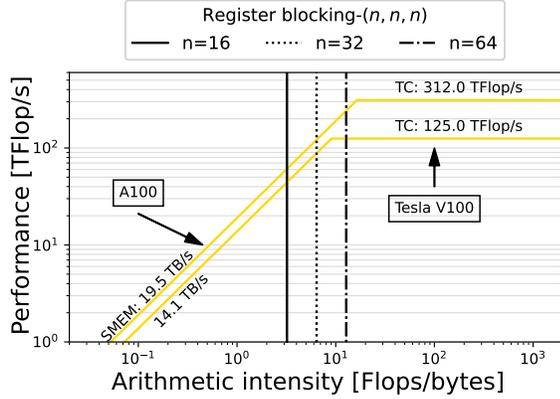
Now, we analyze a matrix-matrix multiplication on Tensor Cores using the roofline model. The input matrices **A** and **B** are FP16, **C** and **D** are FP32 stored in the shared memory. We load the sub-matrices of each input matrix as fragments  $\mathbf{A}_{\text{reg}}$  and  $\mathbf{B}_{\text{reg}}$  for register blocking. The register blocking size is  $(n, n, n)$ . We show the roofline model of computing  $\mathbf{D}_{\text{reg}} \leftarrow \mathbf{A}_{\text{reg}} \cdot \mathbf{B}_{\text{reg}} + \mathbf{C}_{\text{reg}}$  in Figure 1. The Arithmetic Intensity (AI) is calculated as follows:

$$\text{AI} = \frac{2n^3}{(n^2 + n^2)\text{sizeof}(\text{FP16}) + (n^2 + n^2)\text{sizeof}(\text{FP32})} = \frac{n}{5}. \quad (1)$$

As the size of register blocking size increases, we can utilize the performance of Tensor Cores more. However, the number of registers is finite, and the registers spill to local memory when using more than 256 registers per thread. The number of 32-bit registers required for the blocking is calculated as follows assuming the mma instruction is used and each element in a matrix is stored by only one element of a fragment without duplication.

$$\text{nRegs} = \left( \underbrace{(n^2)}_{\mathbf{A}_{\text{reg}}} + \underbrace{(n^2)}_{\mathbf{B}_{\text{reg}}} \right) \times \frac{1}{2} + \underbrace{(n^2)}_{\mathbf{C}_{\text{reg}}} / \text{warpSize} = \frac{1}{16}n^2. \quad (2)$$

For instance, in the case of  $n = 64$ , the number of required registers is 256, and the registers spill to local memory. Therefore, we need to reduce the shared memory access not by increasing the register blocking size. Furthermore, the Tensor Cores performance has been improved more than the shared memory bandwidth on NVIDIA A100 compared to



**Figure 3.** The arithmetic intensity of matrix-matrix multiplication for each size of register blocking  $(n, n, n)$ .

V100. This can be seen from the fact that the AI value at the boundary between the memory bandwidth and the computational performance bound is smaller for A100 than for V100.

## 4 WMMA API extension library

To leverage the high Tensor Cores performance, it is necessary to supply matrix data to Tensor Core with sufficient throughput. However, due to the limited functionality of the WMMA API, the throughput improvements that can be made using only the WMMA API are limited. Therefore, we implement a WMMA API Extension library (WMMAE) to reinforce the functionality of WMMA API. The WMMAE consists of the following two components:

1. Primitive functions
2. SGEMM emulation on Tensor Cores using Error Correction method (WMMAE-TCEC)

In this section, we show the functionality of these components and evaluate the performance improvement compared to only using WMMA API. We use NVIDIA A100 40GB SXM4 and NVIDIA V100 16GB PCIe GPUs for the evaluations.

### 4.1 Primitive functions

We can generate a fragment of a matrix in which all elements are the same value without shared memory access using `fill_fragment` function in WMMA API. On the other hand, to generate fragments of other matrices, it is necessary to explicitly store the matrix in shared memory and load it using `load_matrix_sync` function in WMMA API. Now, we consider the matrices that have some structural rules. For instance, when performing scan operations using matrix-vector multiplication, we need an upper triangular matrix  $U$  in which all non-zero elements are one. Then, we perform a scan operation to an array  $[a_0 \ a_1 \ \dots \ a_{n-1}]$  using  $n \times n$  matrix  $U$  as follows:

$$\begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}^T \cdot U = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}^T \cdot \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 0 & 1 & 1 & \dots & 1 \\ 0 & 0 & 1 & \dots & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix} = \begin{bmatrix} a_0 \\ \sum_{i=0}^1 a_i \\ \vdots \\ \sum_{i=0}^{n-1} a_i \end{bmatrix}^T.$$

The structural rule for the  $(i, j)$  element of the matrix  $U$  is as follows:

$$u_{i,j} = \begin{cases} 1 & i \leq j \\ 0 & \text{Otherwise} \end{cases} \quad (3)$$

Dokkak *et al.* utilize the rule for generating the fragment of the matrix without storing it explicitly in shared memory. We generalize the functionality and provide functions for generating a fragment of any matrix from its structural rule: `foreach_ij` and `map`.

### 4.2 Primitive function : foreach\_ij

The `foreach_ij` function calculates the mapping between matrix element position  $(i, j)$  and fragment indices and gives them to a given lambda function. In the lambda function, we calculate the value of the  $(i, j)$  element of the matrix and set it to the fragment using the given mapping information. For instance, we show a pseudocode for generating the matrix  $U$  fragment by the rule in Eq (3) in Code 3. Strictly speaking, since one element in a matrix is kept by two fragment elements when using WMMA API for C++, `foreach_ij` function gives the list of fragment element indices to the lambda function. However, in this pseudocode, we simplify the argument of the lambda function as only one fragment index is given. By using this function, we can generate a fragment of any matrix from its structural rule without storing it in shared memory.

```

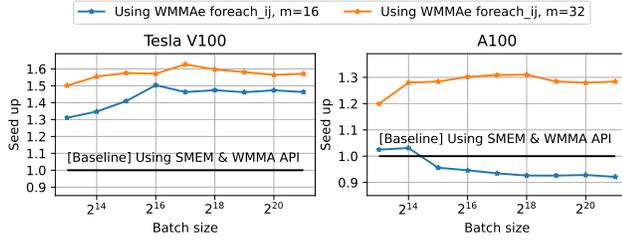
1 fragment <16, 16, 16> frag;
2 foreach_ij <decltype(frag)>(<
3 // The lambda function to set each fragment
4   elements
5   [&](fid, i, j) {
6     if (i <= j) frag.x[fid] = 1;
7     else frag.x[fid] = 0;
8   });

```

**Code 3.** Generating the matrix  $U$  fragment from the structural rule in Eq. (3) using WMMAE `foreach_ij` function.

**4.2.1 Performance evaluation.** We use a batched Householder transformation benchmark for evaluating the performance improvement by `foreach_ij` function. The Householder transformation is one of the orthogonal transformations used for QR factorization etc. This transformation is calculated as follows for a  $n \times n$  Householder matrix  $H$ ,  $m \times k$  input matrix  $A$ :

$$H \cdot A = (I_m - 2v^T v) \cdot A, \quad (4)$$



**Figure 4.** The performance evaluation of `foreach_ij` function using batched Householder benchmark, where we multiply a  $m \times m$  Householder matrix  $H$  with an input matrix  $A$  using Tensor Cores.

where  $\mathbf{v}$  is a  $m$ -dimensional identity vector and  $\mathbf{I}_m$  is a  $m \times m$  identity matrix. In this benchmark, we explicitly compute the Householder matrix  $H$  from  $\mathbf{v}$  and multiply it by  $A$ . This computation is performed for  $b$  (batch size) FP16 input matrices  $A_i$  and FP16 vectors  $\mathbf{v}_i$ . To obtain the baseline performance, we implemented the batched Householder transformation, which stores the Householder matrix in shared memory and loads it using the WMMA API function. Then the multiplication of  $A$  and  $H$  is performed on Tensor Cores. We show a speed-up ratio using WMMAe in Figure 4. We can see that the performance is improved using `foreach_ij` on V100 GPU in both cases. On the other hand, for  $m = 16$  on A100, the implementation using `foreach_ij` has a lower performance compared to the baseline. In this case, the pseudocode of the implementation is shown in Code 4.

```

1 fragment <16, 16, 16> frag;
2 foreach_ij <decltype (frag)> (
3   [&](fid, i, j) {
4     auto elm = v[i]*v[j]*(-2);
5     if (i==j) elm += 1;
6     frag.x[fid] = elm;
7   });

```

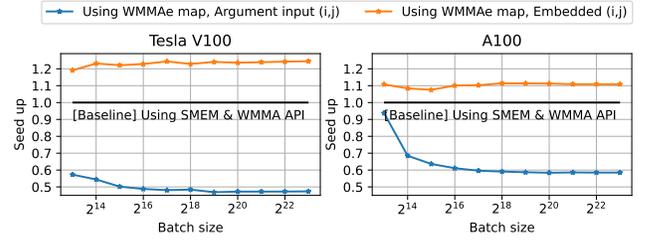
**Code 4.** Generating a  $16 \times 16$  Householder matrix fragment using `foreach_ij`.

In this code, the cost of the mapping calculation is higher than the cost of storing the matrix explicitly in shared memory, which might be the reason for the low performance. Whereas, for  $m = 32$  on A100, the implementation using `foreach_ij` has higher performance than the baseline. In this case, the pseudocode of the implementation is shown in Code 5.

```

1 fragment <16, 16, 16> frag[2 * 2]; // 32x32 matrix
2 foreach_ij <decltype (*frag)> (
3   [&](fid, i, j) {
4     for (unsigned bi = 0; bi < 2; bi++) {
5       for (unsigned bj = 0; bj < 2; bj++) {
6         auto elm = v[i+bi*16] * v[j+bj*16]*(-2);
7         if (i==j) elm += 1;
8         frag[bi+bj*2].x[fid] = elm;

```



**Figure 5.** The performance evaluation of `map` function using batched Given's rotation benchmark. The "Argument input ( $i, j$ )" means that the parameter ( $i, j$ ) for Given's rotation matrix is set through kernel function arguments, and "Embedded ( $i, j$ )" means that these parameters are set in compile-time.

```

9   } } } );

```

**Code 5.** Generating an array of fragments for  $32 \times 32$  Householder matrix using `foreach_ij`.

For the  $32 \times 32$  matrix fragment, we used a  $2 \times 2$  array of fragments holding matrices of size  $16 \times 16$ . The elements of all the fragments are set in a single `foreach_ij` function. This means that four fragments are generated in one mapping calculation, and the cost of the mapping calculation is relatively lower than that of the  $m = 16$  case. Thus, we consider that reusing the mapping calculation among several fragments is important to speed up the use of the `foreach_ij` function.

### 4.3 Primitive function : `map`

The `map` function takes the position ( $i, j$ ) of an element of the matrix as an argument and returns a pair (`lid`, `fid`) of the thread number (lane id; `lid`) in a warp and the element number of the fragment holding this element. Using this function, we can manipulate any ( $i, j$ ) element of the matrix as a fragment. For instance, Code 6 sets the ( $i, j$ ) element of a matrix  $A$ , which is held as a fragment, to 1.

```

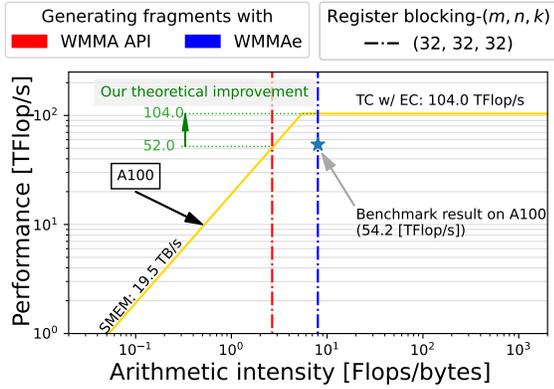
1 fragment frag_a;
2 unsigned lid, fid;
3 // Calculate lid and fid from matrix position (i,
4   j)
5 map<decltype (frag)>(lid, fid /*=2*/, i, j);
6 // Set 1
7 if ((threadIdx.x & 0x1f) == lid) {
8   frag_a.x[fid] = 1;
9 }

```

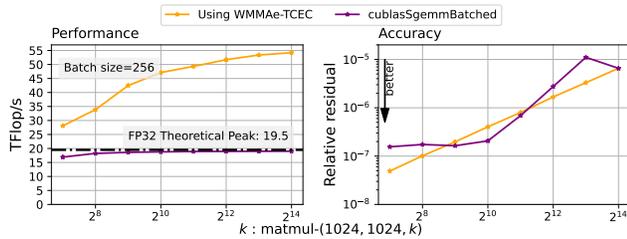
**Code 6.** Setting ( $i, j$ )-element of a matrix held as fragment using WMMAe `map` function.

**4.3.1 Performance evaluation.** We define a batched Given's rotation benchmark to evaluate the performance improvement by the `map` function. The Given's rotation is a rotation operation for a vector and matrix and is used for QR factorization etc. The definition of Given's rotation for a matrix  $A$





**Figure 7.** The arithmetic intensity of SGEMM emulation on Tensor Cores using error correction method. The peak performance is calculated by dividing the theoretical peak performance of FP16-TC in Table 1 by 3 since we need 3 times matrix-matrix multiplication in Eq. (8).



**Figure 8.** The throughput and accuracy evaluation of batched SGEMM using WMMAe-TCEC.

memory bandwidth. Although we can increase the AI by increasing the size of register blocking, the number of registers that one thread can use is limited by the hardware. For instance, in the case of  $(m, n, k) = (32, 32, 32)$ , which is used in our benchmark evaluation, we need 128 32-bit registers to keep the fragments, which amounts to 50% of registers that one thread can use. The registers are used not only for fragments but also for memory access offset calculations and other floating-point value operations such as eq. (7). Reducing the number of required registers can improve the throughput since it can improve occupancy. And when the number of required registers exceeds the hardware limitation, the device memory is used instead, which results in performance degradation. Therefore, increasing the AI without increasing the register blocking size is advantageous.

**4.4.2 Performance evaluation.** We use a batched matrix-matrix multiplication benchmark to evaluate the performance and accuracy of the WMMAe-TCEC. In this benchmark, we compute 256 matrix-matrix multiplications  $A_i \cdot B_i$  where each  $A_i$  and  $B_i$  are  $1024 \times k$  and  $k \times 1024$  FP32 matrices. Then, we calculate the computing performance from

the computing time  $t$  [s] as  $(2 \times 1024 \times 1024 \times k/t)$  [Flop/s], and a max relative error for the accuracy. We show the performance and accuracy comparison between our implementation using WMMAe-TCEC and cuBLAS batched SGEMM function in Figure 8. In our implementation, we use the mma instruction, and the shared memory and register blocking sizes are  $(128, 128, 32)$  and  $(32, 32, 32)$ , respectively. We found this blocking size using a grid search that experimentally maximizes the throughput on NVIDIA A100 (40GB, SXM4) GPU. The outcome of our evaluation shows that our implementation achieves 54.2 [TFlop/s], which outperforms the theoretical peak performance of FP32 on NVIDIA A100, while the accuracy remains the same with cuBLAS SGEMM. The achieved throughput is larger than the throughput of SGEMM emulation that we have achieved using the NVIDIA CUTLASS library (51 TFlop/s) in our previous paper [11]. According to the roofline model, when we only use WMMA API, the theoretical peak performance for our chosen register blocking size is limited to 52.0 TFlop/s bounded by the shared memory bandwidth. Therefore, the achieved throughput can not be achieved without reducing the shared memory footprint that our library does. However, by using WMMAe, we improved the theoretical peak performance of this method to 104.0 TFlop/s by reducing the shared memory footprint. Since the achieved efficiency is only 52% of the theoretical peak performance, we believe there is room for improving the throughput.

We summarize the advantages of WMMAe-TCEC as follows:

- It provides an interface for the single-precision emulation method on Tensor Cores, which has the same interface as NVIDIA WMMA API.
- It improves the theoretical peak performance of matrix-matrix multiplication with error correction by reducing shared memory footprint without increasing register usage.
- It reduces the shared memory usage required to store the fragments of FP16 matrices when using only WMMA API.
- It is proved to outperform the FP32 theoretical peak performance on NVIDIA A100 experimentally while the accuracy remains the same with FP32 computation.

## 5 Conclusion

We have investigated a simple matrix-matrix multiplication on Tensor Cores by roofline model and found that reducing the shared memory footprint is necessary to fully exploit the high throughput of Tensor Cores. To reduce the footprint, we implement a WMMA API extension library which allows us to generate fragments flexibly. This library is open-source and available on GitHub. We show that this library can improve the computing throughput on Tensor Cores. Furthermore, we improve the theoretical peak performance

of single precision matrix-matrix multiplication emulation on Tensor Cores, which is bounded by the shared memory bandwidth when using only WMMA API. Then, we provide this functionality with the same interface as WMMA API. We also show that this functionality can outperform the FP32 theoretical peak performance on NVIDIA A100 GPU. We believe such a faster data supply is necessary to maximize the use of high-speed matrix multiplication units in future architectures.

## Acknowledgments

This work was partially supported by JSPS KAKENHI JP22H03598 and JP21J14694. This work was partially supported by "Joint Usage/Research Center for Interdisciplinary Large-scale Information Infrastructures" in Japan (Project ID: jh220022-NAHI)

## References

- [1] Peng Chen, Mohamed Wahib, Shinichiro Takizawa, Ryousei Takano, and Satoshi Matsuoka. 2019. A versatile software systolic execution model for GPU memory-bound kernels. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '19)*. Association for Computing Machinery, New York, NY, USA, 1–81. <https://doi.org/10.1145/3295500.3356162>
- [2] NVIDIA Corporation. 2022. NVIDIA H100 TENSOR CORE GPU. <https://resources.nvidia.com/en-us-tensor-core/nvidia-h100-datasheet>
- [3] 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>
- [4] Joshua Finkelstein, Emanuel H. Rubensson, Susan M. Mniszewski, Christian F. A. Negre, and Anders M. N. Niklasson. 2022. Quantum Perturbation Theory Using Tensor Cores and a Deep Neural Network. *Journal of Chemical Theory and Computation* 18, 7 (July 2022), 4255–4268. <https://doi.org/10.1021/acs.jctc.2c00274> Publisher: American Chemical Society.
- [5] A. Haidar, S. Tomov, J. Dongarra, and N. J. Higham. 2018. Harnessing GPU Tensor Cores for Fast FP16 Arithmetic to Speed up Mixed-Precision Iterative Refinement Solvers. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 603–613. <https://doi.org/10.1109/SC.2018.00050>
- [6] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P. Scarpazza. 2018. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. *arXiv:1804.06826 [cs]* (April 2018). <http://arxiv.org/abs/1804.06826> arXiv: 1804.06826.
- [7] Norman P. Jouppi, Cliff Young, Nishant Patil, and et al. 2017. In-Datcenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 1–12. <https://doi.org/10.1145/3079856.3080246>
- [8] Binrui Li, Shenggan Cheng, and James Lin. 2021. tcFFT: Accelerating Half-Precision FFT through Tensor Cores. *arXiv:2104.11471 [cs]* (April 2021). <http://arxiv.org/abs/2104.11471> arXiv: 2104.11471 version: 1.
- [9] Junichiro Makino. 2021. "Near-Optimal" Designs. In *Principles of High-Performance Processor Design: For High Performance Computing, Deep Neural Networks and Data Science*, Junichiro Makino (Ed.). Springer International Publishing, Cham, 95–134. [https://doi.org/10.1007/978-3-030-76871-3\\_5](https://doi.org/10.1007/978-3-030-76871-3_5)
- [10] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. 2018. NVIDIA Tensor Core Programmability, Performance & Precision. *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (May 2018), 522–531. <https://doi.org/10.1109/IPDPSW.2018.00091> arXiv: 1803.04014.
- [11] Hiroyuki Ootomo and Rio Yokota. 2022. Recovering single precision accuracy from Tensor Cores while surpassing the FP32 theoretical peak performance:. *The International Journal of High Performance Computing Applications* (June 2022). <https://doi.org/10.1177/10943420221090256> Publisher: SAGE PublicationsSage UK: London, England.
- [12] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (April 2009), 65–76. <https://doi.org/10.1145/1498765.1498785>

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009