Calculation of Cross-correlation Function Accelerated by Tensor Cores with TensorFloat-32 precision on Ampere GPU

Yuma Kikuchi[1], Kohei Fujita[1,2], Tsuyoshi Ichimura[1,2,3], Muneo Hori[4], and Lalith Maddegedara[1]

[1] Earthquake Research Institute and Department of Civil Engineering, The University of Tokyo, Bunkyo, Tokyo, Japan
{kikuchi-y, fujita, ichimura, lalith}@eri.u-tokyo.ac.jp
[2] Center for Computational Science, RIKEN, Kobe, Japan
[3] Center for Advanced Intelligence Project, RIKEN, Tokyo, Japan
[4] Research Institute for Value-Added-Information Generation, Japan Agency for Marine-Earth Science and Technology, Yokohama, Japan
horimune@jamstec.go.jp

Abstract. The cross-correlation function appears in many fields with time-series data, and speeding up the computation is essential given the recent accumulation of significant amounts of data. The cross-correlation function can be calculated as a matrix-matrix product, and a significant speed-up can be expected utilizing Tensor Core, which is a matrix-matrix product acceleration unit of the latest NVIDIA Graphics Processing Units (GPUs). In this research, we target a new precision data type called the TensorFloat-32, which is available in the Ampere architecture. We develop a fast calculation method considering the characteristics of the cross-correlation function and TensorCore. Our method achieved a very high performance of 53.56 TFLOPS in the performance measurement assuming seismic interferometry using actual data, which is 5.97 times faster than cuBLAS, a widely used linear algebra library on NVIDIA GPUs. In addition, the accuracy of the calculation result is sufficiently high compared to the 64-bit floating-point calculation, indicating the applicability of Tensor Core operations using TensorFloat-32 for scientific calculations. Our proposed method is expected to make it possible to utilize a large amount of data more effectively in many fields.

Keywords: Cross-correlation function · GPU computing · Tensor Core.

1 Introduction

The cross-correlation function expresses the similarity or difference between two time-series data. This calculation is widely used in many fields dealing with time-series data, such as radar detection [2], the discovery of gravity waves in physics [3], and detecting earthquakes and volcanic events by matched filtering

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08754-7_37

[5][6]. A large amount of data has been amassed with the advancement of observation technology, and it is crucial to reduce the computational cost of the cross-correlation function to utilize these data more effectively.

In recent years, many computers equipped with Graphics Processing Units (GPUs) have appeared, and many scientific calculations have been accelerated utilizing GPUs. A GPU-based approach has been proposed to calculate cross-correlation functions [9], but further speed-up can be expected by developing methods based on the latest computer architecture.

NVIDIA's Volta architecture [10] and later GPUs have not only the usual arithmetic units but also matrix-matrix product acceleration units called Tensor Core [11]. Tensor Core can perform matrix-matrix products as a hardware function and maintains exceptionally high theoretical performance, which is one of the main features of the latest NVIDIA GPUs.

A study on accelerating the cross-correlation function calculation targeting the Tensor Core with the Volta architecture was performed by Yamaguchi et al. [12]. The Tensor Core with the Volta architecture supports only 16-bit floating-point arithmetic (FP16), making it challenging to use for scientific calculations that require high precision and a wide dynamic range. Yamaguchi et al. solved this problem by introducing local normalization considering the characteristics of the cross-correlation function and Tensor Core; they achieved a 4.47 fold speed-up while maintaining accuracy compared to the matrix-matrix product function of cuBLAS [13], a matrix arithmetic library on GPUs that also utilizes Tensor Core.

Many systems equipped with NVIDIA Ampere architecture GPU [14] have been established in the last few years. In addition to FP16, the Tensor Core with the Ampere architecture can handle various data precision types such as 8-bit integer (INT8), Brain Floating-Point (bfloat16), and TensorFloat-32 (TF32). Further computation speed-up can be expected utilizing these precision data types. TF32 is a new type of floating-point that supports the same range of values as the 32-bit floating-point (FP32) with the same precision as FP16, and in many cases, calculations that require scaling to avoid over/underflow in FP16 can be performed without scaling.

The calculation using TF32 with Tensor Core can be executed through cuBLAS, and some speed-up can be easily obtained. However, since the computation speed of Tensor Core is much faster than the memory access bandwidth of the GPU, the performance of Tensor Core may not be sufficiently high due to the memory access bandwidth limitation. This problem can be avoided by understanding the behavior of Tensor Core and implementing it in accordance with the characteristics of the cross-correlation function calculation, and further acceleration can be expected.

In this research, we develop a method to accelerate the calculation of the cross-correlation function using a TF32 Tensor Core with Ampere architecture based on the method of Yamaguchi et al. Since Ampere is the first architecture equipped with Tensor Core that can handle TF32 precision, we initially investigate its behavior, and then develop a method for calculating cross-correlation

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08754-7_37

functions that can exploit the high performance of the Ampere architecture. Our proposed method enables faster calculation of the cross-correlation function compared to the matrix-matrix product function of cuBLAS, which can also use Tensor Core with TF32.

Since the calculation of the cross-correlation function is mathematically a one-dimensional convolution operation, the method in this study is expected to be effective in many fields.

The rest of this paper is organized as follows. Section 2 describes how to use Tensor Core with TF32 and implement it to achieve higher performance. Afterward, we describe the specific calculation method of the cross-correlation function. In Section 3, we apply our proposed method to the cross-correlation function calculation using real observed waveforms in the seismic interferometry and discuss its performance and accuracy. Finally, Section 4 summarizes the paper.

The code we developed can be available on GitHub repository [1].

2 Methodology

2.1 Usage of Tensor Cores with TF32

Tensor Core is a matrix-matrix product acceleration unit introduced in the Volta architecture, and can execute Fused Multiple-Add instruction $\mathbf{C} \leftarrow \mathbf{AB} + \mathbf{C}$ for small matrices as a hardware function. TF32 is a new precision data type available in the Tensor Core with Ampere architecture. It consists of a 1-bit sign part, an 8-bit exponent, and a 10-bit mantissa, for a total of 19 bits, and has the same dynamic range and precision as FP32 and FP16, respectively. Tensor Core operations specify the size of the matrices that can be executed; in the case of $\mathbf{A}$ and $\mathbf{B}$ being TF32 and $\mathbf{C}$ being FP32, the matrix sizes $\mathbf{A}^{m\times k}, \mathbf{B}^{k\times n}$, and $\mathbf{C}^{m\times n}$ must be $(m, n, k) = (16, 16, 8)$. Hereinafter, we use the notation $\mathbf{A}, \mathbf{B}, \mathbf{C}$ for the whole matrix, not the small submatrix. In CUDA C++, Tensor Core operations are available through Warp Matrix Multiply-Accumulate (WMMA) API. Fig. 2 shows the basic usage of WMMA API. The execution flow is as follows. Here, 32 threads (1 warp) work together to compute the matrix-matrix product of $(m, n, k) = (16, 16, 8)$.

(1) Define a fragment (i.e., a variable for each thread required for execution in Tensor Core) (`wmma::fragment`)
(2) Load data from shared memory to the fragment (`wmma::load_matrix_sync`)
(3) Perform matrix-matrix product using the fragment (`wmma::mma_sync`)
(4) Store the result in the fragment of the shared memory (`wmma::store_matrix_sync`)

In the case of a matrix-matrix product on a GPU, the shared memory can be used as a buffer for the global memory to perform the calculations more efficiently. When the kernel is executed, data required for computation is stored in the global memory. Global memory can be accessed by all threads and has a large capacity, but it requires long cycles for memory access. Therefore, hiding

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08754-7_37

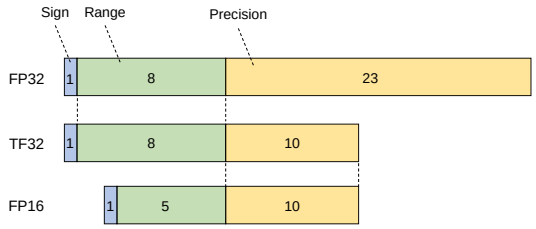

Fig. 1. Comparison of FP32, TF32, and FP16 format

```
// a_shmem, b_shmem and c_shmem are shared memory
__device__ void matmul_tensor_16_16_8 (float *a_shmem, float *b_shmem, float *c_shmem)
{
    wmma::fragment<wmma::matrix_a, 16, 16, 8, wmma::precision::tf32, wmma::row_major> a_frag;
    wmma::fragment<wmma::matrix_b, 16, 16, 8, wmma::precision::tf32, wmma::col_major> b_frag;
    wmma::fragment<wmma::accumulator, 16, 16, 8, float> c_frag;

    // load values from shared memory to fragment
    wmma::load_matrix_sync(a_frag, a_shmem, 8);
    wmma::load_matrix_sync(b_frag, b_shmem, 8);

    wmma::fill_fragment(c_frag, 0.0f);

    // convert to TF32
    for (int i = 0; i < a_frag.num_elements; ++i) a_frag.x[i] = wmma::__float_to_tf32(a_frag.x[i]);
    for (int i = 0; i < b_frag.num_elements; ++i) b_frag.x[i] = wmma::__float_to_tf32(b_frag.x[i]);

    // execute matrix Fused Multiply-Add
    wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);

    // store values from fragment to shared memory
    wmma::store_matrix_sync(c_shmem, c_frag, wmma::col_major);
}
```

Fig. 2. A basic usage of Tensor Core operations with TF32 by calling WMMA API in CUDA C++

the latency becomes difficult when access to global memory occurs frequently. On the other hand, shared memory can only be accessed from within the same thread block and has limited capacity, but the number of cycles associated with access is shorter than that of global memory, so performance improvement can be expected if it is well utilized.

As mentioned above, 32 threads (1 warp) performs matrix-matrix product of $(m, n, k) = (16, 16, 8)$ in the operation using Tensor Core with TF32. In this case, each matrix component must be distributed and stored in each of the 32 threads in the warp, which is called a fragment. The result of the matrix-matrix product is also distributed and stored in the fragment of each thread. The mapping pattern between this matrix and the fragment is complex. Therefore, in CUDA C++, `wmma::{load,store}_matrix_sync` is provided as an API to facilitate this mapping. These functions do not require us to consider complicated fragment mapping patterns, but they need shared memory for mapping to and from fragments. Therefore, when the functions are called frequently, the amount of access between the shared memory as a buffer and for this purpose increases.

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08754-7_37

Given the extremely high computation speed of Tensor Core, the performance is easily constrained by the shared memory access bandwidth. The fragment is internally represented as a set of registers, and the mapping pattern of the Volta architecture with FP16 has been analyzed in the previous study [15]. Therefore, by taking the same approach in this research, it is possible to directly map the corresponding values from the shared memory as a buffer to the registers without using `wmma::{load,store}_matrix_sync`. However, since TF32 is the first type introduced in Tensor Core with Ampere architecture and the supported matrix size is different from FP16, we cannot correctly calculate it by the mapping pattern described in previous work. Here, we explore the mapping pattern of the fragment with TF32 in the Ampere architecture and implement it in a way suitable for the Ampere architecture.

The fragment is a CUDA C++ structure that holds the number of elements it owns as `num_elements` and the values of the `num_elements` fragments as member variables `x[i]`. Thus, we can figure out the mapping pattern by actually outputting the thread number and the value of the fragment in the warp, as shown in Fig. 4, First, we present fragments of the matrix $\mathbf{A}, \mathbf{B}$ (called matrix_A and matrix_B in the WMMA API). These mapping patterns can be determined by generating a matrix with unique non-overlapping elements (e.g., 0,1,$\cdots$,127) executing `wmma::load_matrix_sync`, and then outputting the values of the fragment of each thread and comparing it with the original matrix. The mapping pattern of matrix $\mathbf{C}$ (called accumulator in the WMMA API) can be obtained by setting matrix $\mathbf{A}, \mathbf{B}$ so that the result of the matrix-matrix product is unique and without duplication, and then outputting the fragment after calling `wmma::mma_sync` in the same way as matrix_A and matrix_B.

Figs. 5 and 6 display the mapping patterns of matrix_A (row_major) and matrix_B (col_major), and the accumulator stored in col_major format, respectively. Here, {col, row}_major specifies whether the original two-dimensional matrix is column-first or row-first when it is stored in memory in one dimension. The number in the matrix signifies the element order, and the number in parentheses denotes the storage order in the register of each thread. For example, in matrix_A of row_major and matrix_B of col_major, each thread has four elements out of 16×8 (8×16) distributed elements, and the thread in which `threadIdx.x` % 32 is 0 has 0th, 64th, 4th, 68th, and 0th, 4th, 64th, 68th elements in the 0-index, respectively. In the accumulator stored as col_major, each thread has eight elements out of 16×16 distributed elements, and the thread in which `threadIdx.x` % 32 is 0 has the 0th, 1st, 128th, 129th, 8th, 9th, 136th, and 137th elements in the 0-index. These mappings allow the values to be stored directly in the registers without calling `wmma::load_matrix_sync`, but the matrix-matrix product execution function of Tensor Core `wmma::mma_sync` can only pass the values through the fragment and cannot execute using registers. We overcame this problem using CUDA PTX inline assembly similar to Yamaguchi et al. PTX is a pseudo-assembly language in CUDA, that allows us to write low-level code with higher flexibility than the standard API. Fig. 7 shows the PTX inline assembly of `wmma::mma_sync` for TF32. Based on these mapping

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08754-7_37

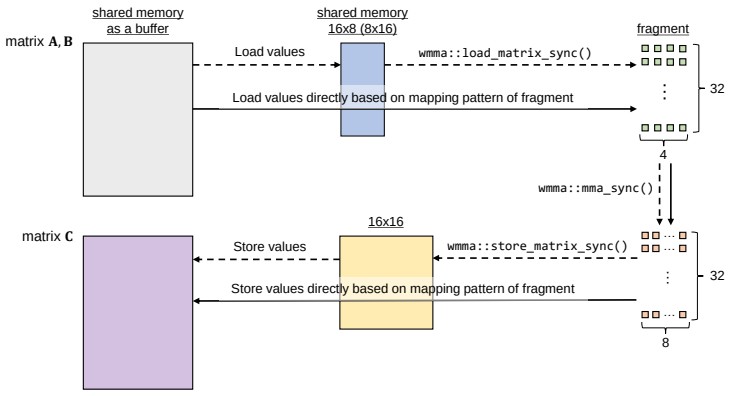

Fig. 3. A simplified data flow of the Tensor Core execution with TF32. Dashed line: use `wmma::load_matrix_sync` API to load/store values, Solid line: directly load/store values based on mapping patterns without using the API (proposed method).

```
…

int thid = threadIdx.x%32;
for (i = 0; i < a_frag.num_elements; ++i) printf("%d %f\n", thid, a_frag.x[i]);
for (i = 0; i < b_frag.num_elements; ++i) printf("%d %f\n", thid, b_frag.x[i]));

…

wmma::mma_sync(c_frag,a_frag,  b_frag, c_frag);
for (i = 0; i < c_frag.num_elements; ++i) printf("%d %f\n", thid, c_frag.x[i]);
```

Fig. 4. A way to examine the mapping pattern from shared memory to Tensor Core fragments and from Tensor Core fragments to shared memory with TF32

patterns, together with the inline PTX assembly, we can realize fast memory access and execution without `wmma::load_matrix_sync` even when TF32 is used in the Ampere architecture.

2.2 Calculation of Cross-correlation Function Using Tensor Cores

The cross-correlation is a function calculated to check the similarity or deviation between two time-series data. The cross-correlation function between two waveforms X_i and X_j of length T can be expressed as a function of the time shift τ, as in Eq. (1).

$$CC_{i,j}(\tau) = \sum_{t=1}^{T} X_i(t) X_j(t+\tau) \tag{1}$$

The cross-correlation function can also be calculated in the frequency domain, but we only deal with it in the time domain. Focusing on a single τ, the calculation of the cross-correlation function is the inner product of two waveforms, which can be calculated as a matrix-matrix product operation for multiple wave-

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08754-7_37

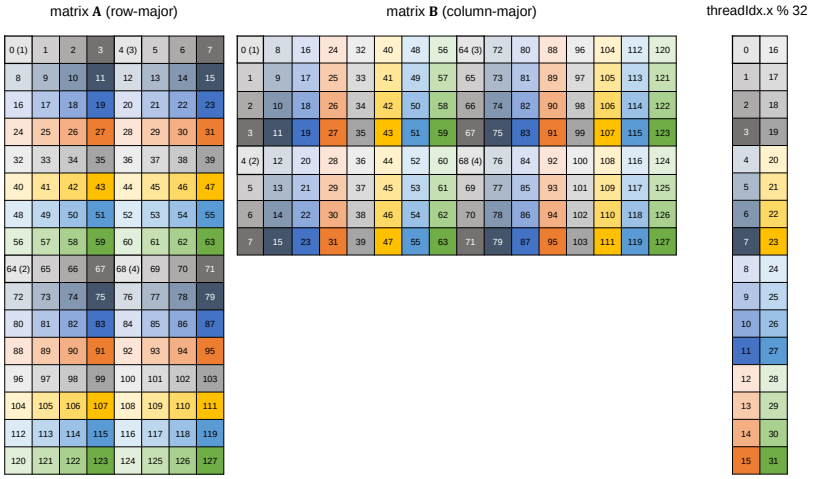

Fig. 5. Mappings of `wmma:fragment<wmma::matrix_a, 16, 16, 8, wmma::precision::tf32, wmma::row_major>` and `wmma:fragment<wmma::matrix_b, 16, 16, 8, wmma::precision::tf32, wmma::col_major>` with Ampere architecture. The number in parentheses shows the order of each value in a fragment.

form pairs with multiple time shifts simultaneously, as presented in Eq(2). We apply the Tensor Core operation to this matrix-matrix product calculation.

$$\begin{pmatrix} CC_{1,1}(0) & CC_{1,1}(1) & CC_{1,1}(2) & CC_{1,1}(3) \\ CC_{2,1}(0) & CC_{2,1}(1) & CC_{2,1}(2) & CC_{2,1}(3) \\ CC_{3,1}(0) & CC_{3,1}(1) & CC_{3,1}(2) & CC_{3,1}(3) \\ CC_{4,1}(0) & CC_{4,1}(1) & CC_{4,1}(2) & CC_{4,1}(3) \end{pmatrix} = \begin{pmatrix} X_1(1) & X_1(2) & X_1(3) & X_1(4) \\ X_2(1) & X_2(2) & X_2(3) & X_2(4) \\ X_3(1) & X_3(2) & X_3(3) & X_3(4) \\ X_4(1) & X_4(2) & X_4(3) & X_4(4) \end{pmatrix} \begin{pmatrix} X_1(1) & X_1(2) & X_1(3) & X_1(4) \\ X_1(2) & X_1(3) & X_1(4) & X_1(5) \\ X_1(3) & X_1(4) & X_1(5) & X_1(6) \\ X_1(4) & X_1(5) & X_1(6) & X_1(7) \end{pmatrix} \quad (2)$$

In Yamaguchi et al.'s method, cross-correlations are simultaneously computed for time shifts of N time steps for 16 waveforms of length 256 per 1 warp (32 threads). Thus, in each warp, the product of matrix $\mathbf{A}$ of size 16×256 and matrix $\mathbf{B}$ of size $256 \times N$ is calculated by executing the matrix-matrix product of size 16×16 and 16×16 multiple times. In this study, we target matrix-matrix product of these sizes, but the size of matrices supported by Tensor Core vary between FP16 and TF32. Thus, we calculate the matrix-matrix product of size 16×16 by performing the matrix product of size 16×8 and 8×16 twice.

As explained above, the matrix-matrix product calculation on the GPU can be performed more efficiently utilizing shared memory. In this research, shared

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08754-7_37

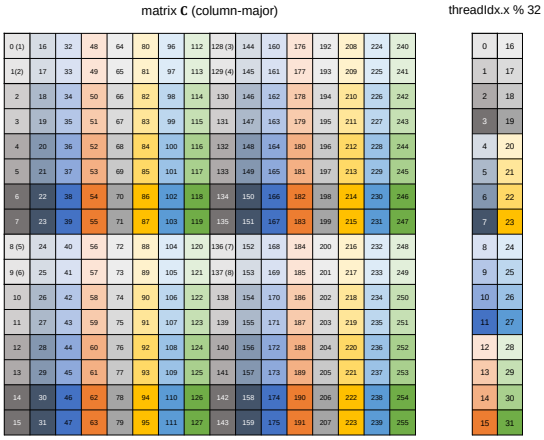

Fig. 6. A mapping of `wmma:fragment<wmma::accumulator, 16, 16, 8, float>` with Ampere architecture. The number in parentheses shows the order of each value in a fragment.

```
asm("{¥n¥t"
    "wmma.mma.sync.aligned.row.col.m16n16k8.f32.tf32.tf32.f32 {%0,%1,%2,%3,%4,%5,%6,%7},
     {%8,%9,%10,%11},{%12,%13,%14,%15},{%16,%17,%18,%19,%20,%21,%22,%23};¥n¥t"
    "}"
    : "=f"(cv[0]),"=f"(cv[1]),"=f"(cv[2]),"=f"(cv[3]),"=f"(cv[4]),"=f"(cv[5]),"=f"(cv[6]),"=f"(cv[7])
    : "r"(av[0]),"r"(av[1]),"r"(av[2]),"r"(av[3])
      "r"(bv[0]),"r"(bv[1]),"r"(bv[2]),"r"(bv[3])
      "f"(cv[0]),"f"(cv[1]),"f"(cv[2]),"f"(cv[3]),"f"(cv[4]),"f"(cv[5]),"f"(cv[6]),"f"(cv[7]));
```

Fig. 7. PTX inline assembly of Tensor Core operation with TF32

memory is also used as a buffer for global memory so that the access to global memory with long latency is minimized.

In the case of calculating the cross-correlation function, the matrix **B** can be constructed by sequentially shifting single time-series data, as shown on the right-hand side of Eq. (2). Since the size of matrix **B** is $256 \times N$, a typical calculation requires a total of $256 \times N$ elements to be loaded from the global memory. Conversely, by employing the characteristics of the cross-correlation function calculation, we only need to read a total of $256 + N - 1$ data, significantly reducing memory access cost and usage. As N increases, the amount of data read decreases in proportion to the amount of computation, which is expected to improve the performance. However, as memory usage increases, hiding the latency associated with operations becomes more difficult, resulting in performance degradation. Yamaguchi et al. stated that $N = 96$ is the equilibrium point in the performance measurement, and the same tendency was observed in our experiment, so the calculation is performed with $N = 96$ in the next section.

For the Tensor Core with Volta architecture, matrices **A** and **B** are supported only for FP16. FP16 has a much narrower dynamic range than the FP32/64-bit floating-point (FP64), which is generally used in scientific computing and

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08754-7_37

may cause over/underflow during matrix-matrix product operation. In Yamaguchi et al.'s method, local scaling is applied to matrices $\mathbf{A}, \mathbf{B}$ to prevent over/underflow during the computation. Although the calculation of the scaling value was quickly performed via the shuffle instructions in the warp, it was necessary to multiply the scaling value locally after every Tensor Core execution. Therefore, another set of registers was allocated and used as buffers for the calculation results, and the scaling values were multiplied when adding them to the buffers. In short, the matrix $\mathbf{C}$ was assigned zero for every calculation, and buffer $\mathbf{D}$ was assigned $\mathbf{C}$ multiplied by the local scaling value after performing $\mathbf{C} \leftarrow \mathbf{AB}+\mathbf{C}$. In contrast, TF32 can handle the same range of values as FP32, so the calculation can be performed correctly without scaling in many cases. Consequently, the scaling restriction mentioned above is eliminated, and computation can be performed only using one set of registers, meaning that the matrix $\mathbf{C}$ only needs to be initialized to zero once at the beginning. Afterward, the register containing the calculation result can be used directly for the Tensor Core execution in the subsequent corresponding execution, as in $\mathbf{C} \leftarrow \mathbf{AB}+\mathbf{C} \leftarrow \mathbf{AB}+\mathbf{C} \leftarrow \cdots$. This reduces the number of registers required for execution and enables more efficient computation.

TF32 is currently supported only for operations on Tensor Core and cannot be used for normal calculations on a CPU and GPU. Therefore, both matrices $\mathbf{A}$ and $\mathbf{B}$ are transferred to the GPU as FP32 at the kernel execution time and converted to TF32 by `wmma::__float_to_tf32` when they are stored in shared memory as buffers. The input and output of `wmma::__float_to_tf32` is FP32, but the output is numerically TF32. If the FP32 type, which is numerically TF32, is mixed with the usual FP32 type operations, the precision and range of the results are undefined [16], but this is not problematic in this time because there is no need to perform scaling or other operations on matrices $\mathbf{A}$ or $\mathbf{B}$ on the GPU.

An overview of the above computation procedure per thread block is shown in Fig. 8.

3 Application and Performance Measurement

3.1 Application Example

An example dealing with the calculation of cross-correlation functions in seismology is the seismic interferometry method [17]. In the seismic interferometry, the cross-correlation function of the waveforms observed at two different stations is calculated and stacked over a long period. In this way, a pseudo-response waveform (Green's function) in which one station is regarded as the source and the other as the observation point can be synthesized [18] (Fig. 9). The earth is constantly vibrating due to natural earthquakes and other microtremors such as pulsations caused by ocean waves and constant microtremors due to human activities. By applying the seismic interferometry to these wavefields, it is expected that Green's function can be obtained without the use of artificial seismic sources [19].

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08754-7_37

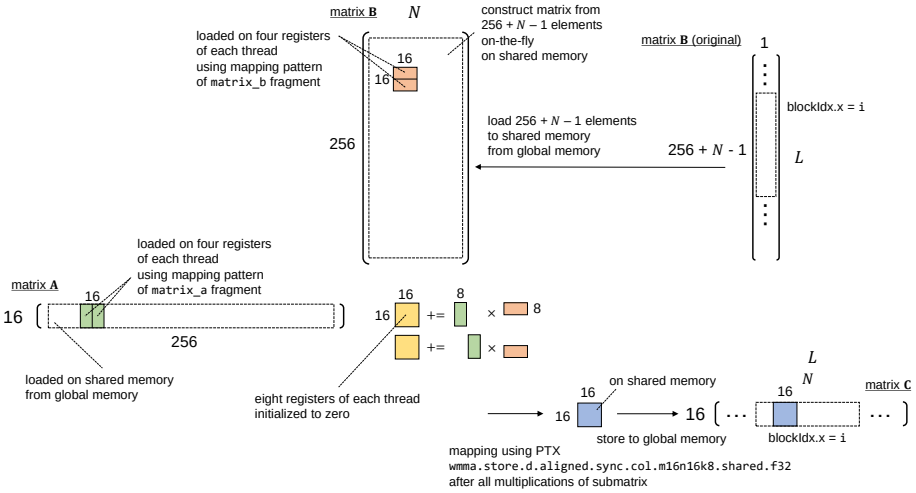

Fig. 8. An overview of the cross-correlation function calculation using Tensor Core with TF32 per thread block. L is the total time shift of the cross-correlation function.

In recent years, many seismic observation networks, such as the MOWLAS [20], have been operating in Japan, and a large amount of data has been accumulated. By continuously applying seismic interferometry to these observation data, it is expected to lead to the realization of the continuous monitoring of underground structures without the need for artificial seismic sources. However, since the number of calculations of the cross-correlation function increases proportionately to the square of the number of observation points, we are currently able to handle only a portion of the data (e.g., by narrowing down the number of observation points).

In this measurement, we calculate the cross-correlation function for 256 time steps of 16 channels and 1 channel for a total of 4.32×10^6 time shifts, as in Yamaguchi et al. In other words, we calculate the cross-correlation function as a matrix-matrix product of size 16×256 and size $256 \times (4.32 \times 10^6)$ and measure the performance of our proposed method. The matrices are constructed using the actual observed data of K-net [23] which is one of the MOWLAS.

3.2 Performance Measurement

At this time, we measure the elapsed time and accuracy of three types of kernels, including the proposed method. The first is the kernel of Yamaguchi et al. that performs computations on FP16 with local scaling and runs on a NVIDIA V100 GPU with the Volta architecture. The other is a kernel using `cublasGemmEx`, which is a dense matrix-dense matrix product function in cuBLAS, a linear algebra library provided by NVIDIA, and runs on a NVIDIA A100 GPU with the Ampere architecture. In `cublasGemmEx`, we can select the data precision type

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08754-7_37

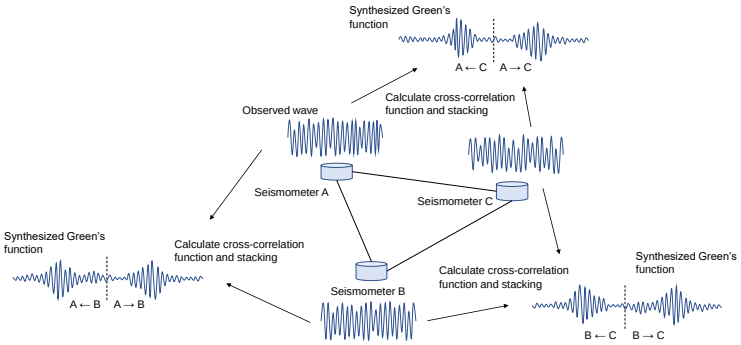

Fig. 9. A basic concept of seismic interferometry. Green's function can be synthesized by calculating the cross-correlation function of the observed waves at two observation points and stacking them over a long period of time.

and execution mode through arguments and options. We set the calculation to be performed using Tensor Core with TF32 precision. Unlike the other kernels, matrix $\mathbf{B}$ is constructed explicitly in advance, and the computation is performed. Lastly, we analyze our proposed method using TF32 with Tensor Core, executed on A100 GPU.

Table 1 displays the peak hardware performance of the NVIDIA V100 and A100 GPUs used in this measurement. We use nvcc V11.2.67 in nvhpc 21.2 as the compiler and `--generate-code arch=compute_{70,80},code=sm_{70,80} -O3 --use_fast_math` as the compile option for V100 and A100, respectively. The elapsed time is measured using nsys nvprof, and the FLOP is counted manually.

We also evaluate the calculation result accuracy of each kernel in comparison with the calculation results of FP64 on the CPU. The accuracy is calculated using Eq. (3), which is defined as the largest absolute error between the result of each kernel and FP64. In order to compute the cross-correlation function on the CPU, we use `cblas_dgemm`, a dense matrix-dense matrix product function in the Intel Math Kernel Library (MKL). We use the Xeon Platinum 8360Y (36 cores), and the peak FP64 performance of this CPU is 2.765 TFLOPS. Additionally, We use icc 19.1.3.304 as the compiler and set `-O3 -mkl=parallel` as the compile option and `KMP_AFFINITY=compact` as the environment variable. For reference, we also measure the MKL kernel performance.

$$ERR = \max_{i,j} \left| CC_{i,j} - CC_{i,j}^{FP64} \right| \tag{3}$$

Table 2 lists the elapsed time, execution performance, and ERR of each kernel. First, Yamaguchi et al.'s kernel using FP16 demonstrates a performance of 25.90 TFLOPS. This is higher than the theoretical performance ratio of FP32 in V100, which is 20.72% of the theoretical performance of FP16 with Tensor Core in V100. In addition, local scaling keeps the error as small as 5.6×10^{-4}.

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08754-7_37

Table 1. Comparison of peak performance between the NVIDIA V100 and A100 GPUs.

	V100 16GB SXM	A100 40GB SXM
FP64 performance	7.8 TFLOPS	9.7 TFLOPS
FP32 performance	15.7 TFLOPS	19.5 TFLOPS
FP16 Tensor Core performance	125 TFLOPS	312 TFLOPS
TF32 Tensor Core performance	N/A	156 TFLOPS
Memory bandwidth	900 GB/s	1555 GB/s

Table 2. Performance and precision of each kernel.

Kernel	Elapsed time	TFLOPS (ratio to peak)	ERR
Yamaguchi (2019) FP16 (V100)	1.366 ms	25.90 (20.72%)	5.6×10^{-4}
cuBLAS TF32 (A100)	3.947 ms	8.97 (5.75%)	2.0×10^{-4}
Proposed TF32 (A100)	0.661 ms	53.56 (34.34%)	2.0×10^{-4}
MKL FP64 (Xeon Platinum)	84.91 ms	0.42 (15.07%)	0.0

On the other hand, our method using TF32 with Tensor Core achieves a very high execution performance of 53.56 TFLOPS, which is 34.34% of the theoretical performance of TF32 with Tensor Core on A100 and higher than the peak performance ratio of Yamaguchi et al.'s kernel. While the theoretical performance of A100's TF32 with Tensor Core is 1.25 times higher than that of V100's FP16 with Tensor Core, the obtained execution performance is 2.07 times higher, likely because the wide dynamic range of TF32 eliminates the need for scaling and allows for more efficient use of Tensor Core. In terms of accuracy, the maximum error is 2.03×10^{-4}, which is approximately half that of the FP16 calculation. In addition, compared with the result on cuBLAS with TF32 with Tensor Core, our proposed method achieves a speed-up of 5.97 times, indicating the proposed method's effectiveness. Finally, we compare the elapsed time on the CPU for reference. In MKL, for matrices of the sizes targeted in this measurement, the overhead of copying the matrices to the buffer becomes relatively large, and the execution performance is as low as 0.42 TFLOPS (15.07% of the peak performance). On the other hand, the proposed method achieves high performance even for matrices of such sizes, resulting in a speed-up of 128.46 times when the theoretical performance ratio is about 56.42 times. By implementing the calculation accounting for the characteristics of Tensor Core and the cross-correlation function, we developed a method that can take advantage of the high performance of Tensor Core with Ampere architecture.

4 Closing Remarks

In this study, we developed a fast computation method for cross-correlation functions in the time domain utilizing Tensor Core with a NVIDIA Ampere architecture GPU. Tensor Core with the Ampere architecture supports various precision data types. We can expect a significant speed-up in many computations

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08754-7_37

involving the matrix-matrix product by utilizing these data types. Tensor Core has very high theoretical performance, but its computational performance is easily limited by the global and shared memory bandwidth when ordinary APIs are used because of its extremely high speed. We first investigated the mapping pattern between shared memory and the fragment for Tensor Core operation using TensorFloat-32 precision on the Ampere architecture. By combining this with a low-level description using PTX inline assembly, we have achieved an implementation that does not require a standard API, and our method is less constrained by memory bandwidth. In the performance measurement of the cross-correlation function calculation in seismic interferometry using actual seismic data, our proposed method achieved a significant performance level of 53.56 TFLOPS, which is 34.34% of the theoretical performance of TF32 with Tensor Core. Moreover, our proposed method is 5.97 times faster than cuBLAS using TF32 with Tensor, a linear algebra library commonly used on NVIDIA GPUs. The cross-correlation function appears in many fields that deal with time-series data, and speeding up the computation has become an important issue in light of the accumulation of observation data, especially in recent years. It is expected that our proposed method can enable analysis that makes fuller and more effective use of big data.

Acknowledgments

We acknowledge support from the Japan Society for the Promotion of Science (18H05239).

References

1. TC-enhanced Cross-correlation Function, https://github.com/nlnxfkl/TC-enhanced_Cross-correlation_Function, Accessed 10 Apr 2022
2. Venu, D., Rao, N.V.K.: A cross-correlation approach to determine target range in passive radar using FM Broadcast Signals, http://dx.doi.org/10.1109/WiSPNET.2016.7566190, (2016). https://doi.org/10.1109/wispnet.2016.7566190.
3. Alonso, D., Cusin, G., Ferreira, P.G., Pitrou, C.: Detecting the anisotropic astrophysical gravitational wave background in the presence of shot noise through cross-correlations, http://dx.doi.org/10.1103/PhysRevD.102.023002, (2020). https://doi.org/10.1103/physrevd.102.023002.
4. Turin, G.: An introduction to matched filters, http://dx.doi.org/10.1109/TIT.1960.1057571, (1960). https://doi.org/10.1109/tit.1960.1057571.
5. Shearer, P.M.: Global seismic event detection using a matched filter on long-period seismograms, http://dx.doi.org/10.1029/94JB00498, (1994). https://doi.org/10.1029/94jb00498.
6. Aso, N., Ohta, K., Ide, S.: Volcanic-like low-frequency earthquakes beneath Osaka Bay in the absence of a volcano, http://dx.doi.org/10.1029/2011GL046935, (2011). https://doi.org/10.1029/2011gl046935.
7. Norman, M.R., Bader, D.A., Eldred, C., Hannah, W.M., Hillman, B.R., Jones, C.R., Lee, J.M., Leung, L., Lyngaas, I., Pressel, K.G., Sreepathi, S., Taylor, M.A., Yuan, X.: Unprecedented cloud resolution in a GPU-enabled full-physics atmospheric climate simulation on OLCF's summit supercomputer, http://dx.doi.org/10.1177/10943420211027539, (2021). https://doi.org/10.1177/10943420211027539.

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08754-7_37

8. Ichimura, T., Fujita, K., Yamaguchi, T., Naruse, A., Wells, J.C., Schulthess, T.C., Straatsma, T.P., Zimmer, C.J., Martinasso, M., Nakajima, K., Hori, M., Maddegedara, L.: A Fast Scalable Implicit Solver for Nonlinear Time-Evolution Earthquake City Problem on Low-Ordered Unstructured Finite Elements with Artificial Intelligence and Transprecision Computing, http://dx.doi.org/10.1109/SC.2018.00052, (2018). https://doi.org/10.1109/sc.2018.00052.
9. Beaucé, E., Frank, W.B., Romanenko, A.: Fast Matched Filter (FMF): An Efficient Seismic Matched - Filter Search for Both CPU and GPU Architectures, http://dx.doi.org/10.1785/0220170181, (2017). https://doi.org/10.1785/0220170181.
10. NVIDIA TESLA V100 GPU ARCHITECTURE, https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, Accessed 04 Feb 2022
11. Markidis, S., Chien, S.W.D., Laure, E., Peng, I.B., Vetter, J.S.: NVIDIA Tensor Core Programmability, Performance & Precision, http://dx.doi.org/10.1109/IPDPSW.2018.00091, (2018). https://doi.org/10.1109/ipdpsw.2018.00091.
12. Yamaguchi, T., Ichimura, T., Fujita, K., Kato, A., Nakagawa, S.: Matched Filtering Accelerated by Tensor Cores on Volta GPUs With Improved Accuracy Using Half-Precision Variables, http://dx.doi.org/10.1109/LSP.2019.2951305, (2019). https://doi.org/10.1109/lsp.2019.2951305.
13. cuBLAS, https://docs.nvidia.com/cuda/cublas/index.html, Accessed 04 Feb 2022
14. NVIDIA A100 Tensor Core GPU Architecture, https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf, last accessed 2022/01/17
15. Raihan, M.A., Goli, N., Aamodt, T.M.: Modeling Deep Learning Accelerator Enabled GPUs, http://dx.doi.org/10.1109/ISPASS.2019.00016, (2019). https://doi.org/10.1109/ispass.2019.00016.
16. CUDA C++ Programming Guide, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, Accessed 04 Feb 2022
17. Curtis, A., Gerstoft, P., Sato, H., Snieder, R., Wapenaar, K.: Seismic interferometry —turning noise into signal, http://dx.doi.org/10.1190/1.2349814, (2006). https://doi.org/10.1190/1.2349814.
18. Wapenaar, K., Fokkema, J.: Green ' s function representations for seismic interferometry, http://dx.doi.org/10.1190/1.2213955, (2006). https://doi.org/10.1190/1.2213955.
19. Chen, Y., Saygin, E.: Empirical Green ' s Function Retrieval Using Ambient Noise Source - Receiver Interferometry, http://dx.doi.org/10.1029/2019JB018261, (2020). https://doi.org/10.1029/2019jb018261.
20. National Research Institute For Earth Science And Disaster Resilience: NIED MOWLAS (Monitoring of Waves on Land and Seafloor), https://nied-ir.bosai.go.jp/?action=repository_uri&item_id=2151&lang=english, (2019). https://doi.org/10.17598/NIED.0009.
21. Dales, P., Audet, P., Olivier, G.: Seismic Interferometry Using Persistent Noise Sources for Temporal Subsurface Monitoring, http://dx.doi.org/10.1002/2017GL075342, (2017). https://doi.org/10.1002/2017gl075342.
22. Voisin, C., Guzmán, M.A.R., Réfloch, A., Taruselli, M., Garambois, S.: Groundwater Monitoring with Passive Seismic Interferometry, http://dx.doi.org/10.4236/jwarp.2017.912091, (2017). https://doi.org/10.4236/jwarp.2017.912091.
23. National Research Institute For Earth Science And Disaster Resilience: NIED K-NET, KiK-net, https://nied-ir.bosai.go.jp/?action=repository_uri&item_id=2146&lang=english, (2019). https://doi.org/10.17598/NIED.0004.

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08754-7_37