DGEMM Using Tensor Cores, and Its Accurate and Reproducible Versions

Daichi Mukunoki[1](✉), Katsuhisa Ozaki[2], Takeshi Ogita[3], and Toshiyuki Imamura[1]

[1] RIKEN Center for Computational Science, Hyogo, Japan
{daichi.mukunoki,imamura.toshiyuki}@riken.jp
[2] Shibaura Institute of Technology, Saitama, Japan
ozaki@sic.shibaura-it.ac.jp
[3] Tokyo Woman's Christian University, Tokyo, Japan
ogita@lab.twcu.ac.jp

Abstract. This paper proposes a method for implementing dense matrix multiplication on FP64 (DGEMM) and FP32 (SGEMM) using Tensor Cores on NVIDIA's graphics processing units (GPUs). Tensor Cores are special processing units that perform 4×4 matrix multiplications on FP16 inputs with FP32 precision, and return the result on FP32. The proposed method adopts the Ozaki scheme, an accurate matrix multiplication algorithm based on error-free transformation for matrix multiplication. The proposed method has three prominent advantages: first, it can be built upon the cublasGemmEx routine using Tensor Core operations; second, it can achieve higher accuracy than standard DGEMM, including the correctly-rounded result; third, it ensures bit-level reproducibility even for different numbers of cores and threads. The achievable performance of the method depends on the absolute-value range of each element of the input matrices. For example, when the matrices were initialized with random numbers over a dynamic range of 1E+9, our DGEMM-equivalent implementation achieved up to approximately 980 GFlops of FP64 operation on the Titan RTX GPU (with 130 TFlops on Tensor Cores), although cublasDgemm can achieve only 539 GFlops on FP64 floating-point units. Our results reveal the possibility of utilizing hardware with limited FP32/FP64 resources and fast low-precision processing units (such as AI-oriented processors) for general-purpose workloads.

Keywords: Tensor cores · FP16 · Half-precision · Low-precision · Matrix multiplication · GEMM · Linear algebra · Accuracy · Reproducibility

1 Introduction

The increasing number of deep learning applications has triggered the development of special processing units such as Tensor Cores on NVIDIA's graphics processing units (GPUs) and Google's Tensor Processing Units (TPUs) in

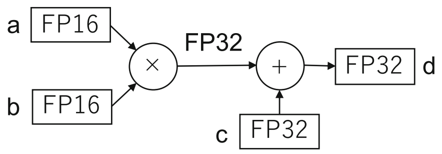


Fig. 1. Tensor Cores (FP16 computations with FP32 precision mode)

recent years. The kernel of such tasks is matrix multiplication, which does not require high-precision such as IEEE 754-2008 binary32 (known as single-precision or FP32, with an 8-bit exponent and a 23-bit fraction) and binary64 (known as double-precision or FP64, with an 11-bit exponent and a 52-bit fraction). The hardware instead supports fast, low-precision operations such as binary16 (known as half-precision or FP16, with a 5-bit exponent and a 10-bit fraction) and 8/16-bit integer operations.

One of the most widely used examples is Tensor Cores introduced in the Volta architecture, which computes a 4×4 matrix multiplication per clock with fused multiply-add operations. Although Tensor Cores support several data formats and computational precisions, the present paper focuses on FP16 computations with FP32 precision mode, which compute $d = a \times b + c$ with FP32 precision (Fig. 1). Here, a and b are FP16 values, and c and d are FP32. The Tensor Cores operate up to eight times faster than standard FP32 floating-point units (FPUs) on CUDA Cores. Many studies have exploited this tremendous performance of Tensor Cores in general tasks.

This paper presents a method for computing a general matrix multiply routine (GEMM) in level-3 basic linear algebra subprograms (BLAS) [4] on FP64 (DGEMM) and FP32 (SGEMM) using Tensor Cores. GEMM is one of the kernel operations of many scientific workloads, as well as high-performance Linpack. The proposed method is based on an accurate matrix multiplication algorithm based on error-free transformation for matrix multiplication, proposed by Ozaki et al. [13], also known as the Ozaki scheme. The advantages of this method are listed below.

- **Productive**: Being built upon the cublasGemmEx routine in cuBLAS[1] provided by NVIDIA, the method incurs a low development cost.
- **Accurate**: The method achieves higher accuracy than standard SGEMM and DGEMM even with correct-rounding.
- **Reproducible**: The method obtains the same (bitwise identical) result for the same input, even when the number of cores and threads differs in each execution.
- **Adaptable**: The concept is adaptable to other precisions.

Whereas some studies simply accelerate the computation not requiring high-precision by utilizing low-precision hardware, the present study attempts more

[1] http://developer.nvidia.com/cublas.

accurate computations by utilizing low-precision hardware. Our DGEMM implementations outperform cuBLAS DGEMM only on processors with limited FP64 support. However, the performance gain over FP64 FPUs is not necessarily important; rather, the intent is to increase the potential of low-precision hardware such as artificial intelligence (AI) oriented processors. Moreover, our method provides a new perspective on the efficient hardware design for both AI and traditional high-performance computing (HPC) workloads. For example, it may reduce the number of FP64 resources in exchange for massive low-precision support.

The remainder of this paper is organized as follows. Section 2 introduces related work, and Sect. 3 describes our methodology based on the Ozaki scheme. Section 4 implements the method, and Sect. 5 presents the accuracy and performance evaluations on Titan RTX and Tesla V100 GPUs. Section 6 discusses the perspective of future hardware design using our proposal. This paper concludes with Sect. 7.

2 Related Work

Several studies have attempted to utilize low-precision hardware designed for AI workloads for other purposes. For example, Haidar et al. [7] utilized standard FP16 and the Tensor Cores operation with FP32 precision in dense and sparse linear systems with iterative refinement. Energy improvement has also been studied [6]. Its error analysis was given by Carson and Higham [1]. Yang et al. [16] presented a Monte Carlo simulation of an Ising model using bfloat16 (BF16, with an 8-bit exponent and a 7-bit fraction) on Google's TPUs. These studies apply low-precision operations to the portions of code not requiring high accuracy, which can be computed at that precision level. Accordingly, their applicability is algorithm- or problem-dependent.

Similarly to the present study, several studies have attempted more accurate operations than those achieved by low-precision hardware. For example, Markidis et al. [11] proposed a method that improves the accuracy of matrix multiplication computed with Tensor Cores. Although their method is conceptually similar to ours, its capability is limited to the computation of matrices with dynamic ranges supported on FP16 with SGEMM-equivalent accuracy. Henry et al. [8] discussed the performance of high-precision operations with double-double arithmetic [2], a classic 2-fold precision arithmetic technique, on BF16 FPUs. Sorna et al. [15] proposed a method to improve the accuracy of 2D fast Fourier transform performed on Tensor Cores. We note that, in those studies, the performance gain over FP32 or FP64 FPUs was not necessarily important; rather, the intent was to increase the potential of low-precision hardware. Therefore, the hardware may need to be redesigned to balance the precisions supported on the FPUs. Our present discussion follows a similar direction.

The Ozaki scheme, which is the kernel of our proposed method, was originally proposed for accurate matrix multiplication by standard floating-point

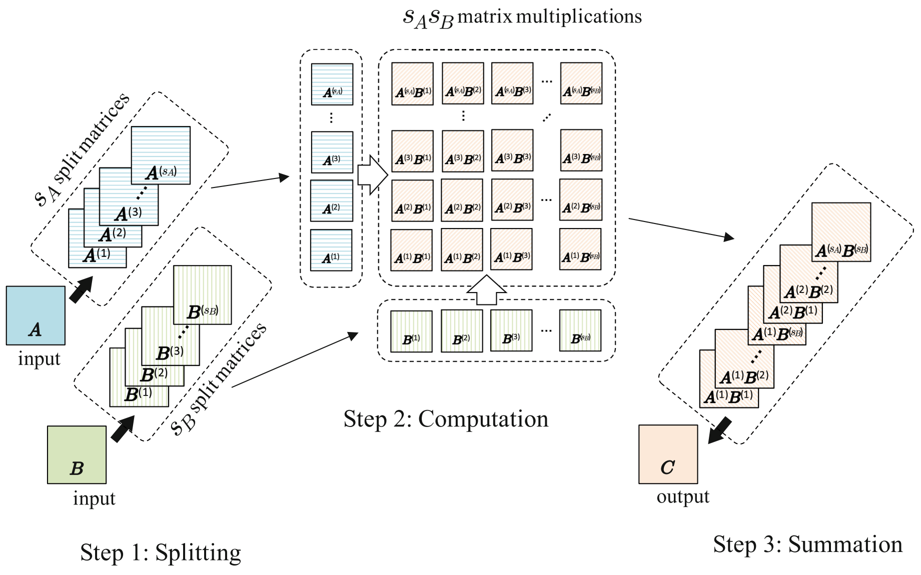


Fig. 2. Schematic of matrix multiplication ($\boldsymbol{C} = \boldsymbol{AB}$) by Ozaki scheme (in this figure, scaling is omitted).

operations. OzBLAS [12][2] implements accurate and reproducible BLAS routines on CPUs and GPUs based on the Ozaki scheme. Whereas OzBLAS was built on DGEMM performed on FP64 FPUs, the Ozaki scheme in the present study performs DGEMM/SGEMM operations using GEMM performed on Tensor Cores. Ichimura et al. [10] also reported a high-performance implementation of the Ozaki scheme based on FP64 operations on many-core CPUs.

3 Methodology

This section first describes the minimal scheme for computing DGEMM by the modified Ozaki scheme on Tensor Cores. Next, it presents additional techniques that accelerate the computations. In this paper, $\mathtt{fl}_{\mathtt{FP64}}(\cdots)$ and $\mathtt{fl}_{\mathtt{FP32}}(\cdots)$ denote the computations performed in FP64 and FP32 arithmetic, respectively, $\mathbf{u}_{\mathtt{FP64}}$ and $\mathbf{u}_{\mathtt{FP32}}$ denote the unit round-offs of FP64 ($\mathbf{u}_{\mathtt{FP64}} = \mathbf{2}^{-\mathbf{53}}$) and FP32 ($\mathbf{u}_{\mathtt{FP32}} = \mathbf{2}^{-\mathbf{24}}$), respectively, and $\mathbb{F}_{\mathtt{FP64}}$ and $\mathbb{F}_{\mathtt{FP16}}$ denote the sets of FP64 and FP16 floating-point numbers, respectively. $\mathbb{N}$ denotes the set of natural numbers including zero.

3.1 Ozaki Scheme for Tensor Cores

The Ozaki scheme performs an error-free transformation of matrix multiplication; specifically, the matrix multiplication is transformed into a summation of

[2] http://www.math.twcu.ac.jp/ogita/post-k/results.html.

Algorithm 1. Splitting of a vector $\boldsymbol{x} \in \mathbb{F}_{\texttt{FP64}}{}^n$ in Ozaki scheme for Tensor Cores. ($\boldsymbol{x}$, $\boldsymbol{x}_{\texttt{tmp}}$, and $\boldsymbol{x}_{\texttt{split}}[j]$ are vectors, and the others are scalar values. Lines 9–11 are computations of $\boldsymbol{x}_i$, $\boldsymbol{x}_{\texttt{tmp}_i}$, and $\boldsymbol{x}_{\texttt{split}}[j]_i$ for $1 \le i \le n$)

```
1: function $((\boldsymbol{x}_{\texttt{split}}[1:s_x], c[1:s_x]) = \texttt{Split}(n, \boldsymbol{x}))$
2:     $\rho = \texttt{ceil}(\texttt{log2}(\mathbf{u}_{\texttt{FP64}}{}^{-1}) - (\texttt{log2}(\mathbf{u}_{\texttt{FP32}}{}^{-1}) - \texttt{log2}(n))/2)$
3:     $\mu = \max_{1 \le i \le n}(|\boldsymbol{x}_i|)$
4:     $j = 0$
5:     while $(\mu \neq 0)$ do
6:         $j = j + 1$
7:         $c[j] = \tau = \texttt{ceil}(\texttt{log2}(\mu))$          // $\tau$ is hold on $c$ for upscaling later
8:         $\sigma = 2^{\rho+\tau}$
9:         $\boldsymbol{x}_{\texttt{tmp}_i} = \texttt{fl}_{\texttt{FP64}}((\boldsymbol{x}_i + \sigma) - \sigma)$          // $\boldsymbol{x}_{\texttt{tmp}}$ is the split vector on FP64
10:        $\boldsymbol{x}_i = \texttt{fl}_{\texttt{FP64}}(\boldsymbol{x}_i - \boldsymbol{x}_{\texttt{tmp}_i})$
11:        $\boldsymbol{x}_{\texttt{split}}[j]_i = \texttt{fl}_{\texttt{FP16}}(\texttt{fl}_{\texttt{FP64}}(2^{-\tau}\boldsymbol{x}_{\texttt{tmp}_i}))$
                                   // Downscaling and conversion from FP64 to FP16
12:        $\mu = \max_{1 \le i \le n}(|\boldsymbol{x}_i|)$
13:    end while
14:    $s_x = j$
15: end function
```

several matrix multiplications that can be performed on floating-point operations without rounding-errors. Figure 2 is a schematic of the whole Ozaki scheme. The method performs three major steps:

- **Step 1: Splitting** – element-wise splitting of the input matrices into several split matrices.
- **Step 2: Computation** – computation of all-to-all matrix products of the split matrices.
- **Step 3: Summation** – element-wise summation of the all-to-all matrix products.

We now describe each step in detail. For simplicity, we consider an inner product of two vectors $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{F}_{\texttt{FP64}}{}^n$, but the approach is naturally extendible to matrix multiplication as it consists of inner products. Also, although we describe the case for DGEMM only, the same concept applies to SGEMM.

Step 1: Splitting. Algorithm 1 splits the input vectors on FP64 into several vectors on FP16 as follows.

$$\begin{aligned}
\boldsymbol{x} &= 2^{c^{(x_1)}}\boldsymbol{x}^{(1)} + 2^{c^{(x_2)}}\boldsymbol{x}^{(2)} + \cdots + 2^{c^{(x_{s_x})}}\boldsymbol{x}^{(s_x)} \\
\boldsymbol{y} &= 2^{c^{(y_1)}}\boldsymbol{y}^{(1)} + 2^{c^{(y_2)}}\boldsymbol{y}^{(2)} + \cdots + 2^{c^{(y_{s_y})}}\boldsymbol{y}^{(s_y)} \\
&\quad x_p, y_q, s_x, s_y, c^{(p)}, c^{(q)} \in \mathbb{N}, \boldsymbol{x}^{(p)}, \boldsymbol{y}^{(q)} \in \mathbb{F}_{\texttt{FP16}}{}^n
\end{aligned}$$

A split vector is first obtained on FP64 and then converted (downscaled) to FP16. The conversion moves only the exponent and causes no significand-bit loss. Here, $2^{c^{(p)}}$ and $2^{c^{(q)}}$ are the downscaling factors (from FP64 to FP16) of the exponents of $\boldsymbol{x}^{(p)}$ and $\boldsymbol{y}^{(q)}$, respectively. At line 7 in Algorithm 1, τ reaches 1024 when $\mu =$`DBL_MAX`, meaning that $c^{(p)}$ and $c^{(q)}$ can be stored as 2-byte short integers. The splitting algorithm must satisfy the following properties:

1. If ${\boldsymbol{x}^{(p)}}_i$ and ${\boldsymbol{y}^{(q)}}_j$ are non-zero elements,

$$|{\boldsymbol{x}^{(p)}}_i| \geq |{\boldsymbol{x}^{(p+1)}}_i|, |{\boldsymbol{y}^{(q)}}_j| \geq |{\boldsymbol{y}^{(q+1)}}_j|$$

2. $(\boldsymbol{x}^{(p)})^T\boldsymbol{y}^{(q)}$ must be error-free in the FP32 computation:

$$(\boldsymbol{x}^{(p)})^T\boldsymbol{y}^{(q)} = \mathtt{fl}_{\mathtt{FP32}}((\boldsymbol{x}^{(p)})^T\boldsymbol{y}^{(q)}), 1 \leq p \leq s_x, 1 \leq q \leq s_y$$

Splitting can be understood as a translation from a floating-point representation to a fixed-point representation. The former of the above two properties means that the accuracy of the final result can be controlled (to lower accuracy) by omitting some split vectors from the lowest term. The accuracy of the final result obtainable with a certain number of split vectors depends on the length of the inner product and the range of the absolute values in each element of the input vectors. Note that to replace Tensor Cores by other FPUs with different precisions, we need to modify parameter ρ in Algorithm 1, and the number of bits held in the split vectors ($\boldsymbol{x}^{(p)}$ and $\boldsymbol{y}^{(q)}$) depends on the precision of the FPUs.

Step 2: Computation. Next, the inner product $\boldsymbol{x}^T\boldsymbol{y}$ is computed as

$$\begin{aligned}
\boldsymbol{x}^T\boldsymbol{y} &= (2^{c^{(x_1)}}\boldsymbol{x}^{(1)} + 2^{c^{(x_2)}}\boldsymbol{x}^{(2)} + \cdots + 2^{c^{(x_{s_x})}}\boldsymbol{x}^{(s_x)})^T \\
&\quad (2^{c^{(y_1)}}\boldsymbol{y}^{(1)} + 2^{c^{(y_2)}}\boldsymbol{y}^{(2)} + \cdots + 2^{c^{(y_{s_y})}}\boldsymbol{y}^{(s_y)}) \\
&= 2^{c^{(x_1)}+c^{(y_1)}}\mathtt{fl}_{\mathtt{FP32}}((\boldsymbol{x}^{(1)})^T\boldsymbol{y}^{(1)}) + 2^{c^{(x_1)}+c^{(y_2)}}\mathtt{fl}_{\mathtt{FP32}}((\boldsymbol{x}^{(1)})^T\boldsymbol{y}^{(2)}) + \\
&\quad \cdots + 2^{c^{(x_1)}+c^{(y_{s_y})}}\mathtt{fl}_{\mathtt{FP32}}((\boldsymbol{x}^{(1)})^T\boldsymbol{y}^{(s_y)}) \\
&+ 2^{c^{(x_2)}+c^{(y_1)}}\mathtt{fl}_{\mathtt{FP32}}((\boldsymbol{x}^{(2)})^T\boldsymbol{y}^{(1)}) + 2^{c^{(x_2)}+c^{(y_2)}}\mathtt{fl}_{\mathtt{FP32}}((\boldsymbol{x}^{(2)})^T\boldsymbol{y}^{(2)}) + \\
&\quad \cdots + 2^{c^{(x_2)}+c^{(y_{s_y})}}\mathtt{fl}_{\mathtt{FP32}}((\boldsymbol{x}^{(2)})^T\boldsymbol{y}^{(s_y)}) \\
&+ \cdots \\
&+ 2^{c^{(x_{s_x})}+c^{(y_1)}}\mathtt{fl}_{\mathtt{FP32}}((\boldsymbol{x}^{(s_x)})^T\boldsymbol{y}^{(1)}) + 2^{c^{(x_{s_x})}+c^{(y_2)}}\mathtt{fl}_{\mathtt{FP32}}((\boldsymbol{x}^{(s_x)})^T\boldsymbol{y}^{(2)}) + \\
&\quad \cdots + 2^{c^{(x_{s_x})}+c^{(y_{s_y})}}\mathtt{fl}_{\mathtt{FP32}}((\boldsymbol{x}^{(s_x)})^T\boldsymbol{y}^{(s_y)})
\end{aligned}$$

Here, the computation of all-to-all inner products of the split vectors is performed: a total of $s_x s_y$ inner products are computed. $2^{c^{(x_p)}+c^{(y_q)}}$ is the upscaling factor that compensates the downscaling performed in the splitting process. By the second property of Algorithm 1, the inner products of the split vectors can be computed with Tensor Core operations because the inputs are stored in the FP16 format. When extending this example to matrix multiplication, the split matrices must be multiplied by the algorithm based on the standard inner product: divide-and-conquer approaches such as Strassen's algorithm are not permitted.

Algorithm 2. Matrix multiplication $\boldsymbol{C} = \boldsymbol{AB}$ ($\boldsymbol{A} \in \mathbb{F}_{\mathtt{FP64}}{}^{m\times k}$, $\boldsymbol{B} \in \mathbb{F}_{\mathtt{FP64}}{}^{k\times n}$, $\boldsymbol{C} \in \mathbb{F}_{\mathtt{FP64}}{}^{m\times n}$) with Ozaki scheme

```
 1: function ($\boldsymbol{C}$ = DGEMM-TC($m, n, k, \boldsymbol{A}, \boldsymbol{B}$))
 2:     ($\boldsymbol{A}_{\mathtt{split}}[1:s_A], \boldsymbol{c}_A[1:s_A]$) = SplitA($m, k, \boldsymbol{A}$)    // $\boldsymbol{A}_{\mathtt{split}}$ is obtained on FP16
 3:     ($\boldsymbol{B}_{\mathtt{split}}[1:s_B], \boldsymbol{c}_B[1:s_B]$) = SplitB($k, n, \boldsymbol{B}$)    // $\boldsymbol{B}_{\mathtt{split}}$ is obtained on FP16
 4:     $\boldsymbol{C}_{ij} = 0$
 5:     for ($q = 1 : s_B$) do
 6:         for ($p = 1 : s_A$) do
 7:             $\boldsymbol{C}_{\mathtt{tmp}}$ = GEMM$_{\mathtt{FP32}}$($m, n, k, \boldsymbol{A}_{\mathtt{split}}[p], \boldsymbol{B}_{\mathtt{split}}[q]$)
           // This can be performed using Tensor Cores as $\boldsymbol{A}_{\mathtt{split}}$ and $\boldsymbol{B}_{\mathtt{split}}$ are FP16
 8:             $\boldsymbol{C}_{ij} = \boldsymbol{C}_{ij} + 2^{\boldsymbol{c}_A[p]_i + \boldsymbol{c}_B[q]_j} {\boldsymbol{C}_{\mathtt{tmp}}}_{ij}$
                                         // Computations for $1 \le i \le m$ and $1 \le j \le n$
 9:         end for
10:     end for
11: end function
```

Step 3: Summation. Finally, the inner products of the split vectors are summed. The summation can be computed by FP64 arithmetic if the required accuracy is that of standard DGEMM. However, as $\mathtt{fl}_{\mathtt{FP32}}((\boldsymbol{x}^{(p)})^T \boldsymbol{y}^{(q)})$ in Step 2 has no rounding errors (being error-free), the correctly-rounded result of $\boldsymbol{x}^T\boldsymbol{y}$ can be obtained by summation with a correctly-rounded method such as NearSum [14]. The result is reproducible if the summation is performed by some reproducible method, even in FP64 arithmetic. As the summation is computed element-wise, the order of the computation is easily fixed.

Whole Procedure on Matrix Multiplication. Algorithm 2 computes the whole Ozaki scheme for matrix multiplication on Tensor Cores. Here, `SplitA` and `SplitB` perform the splitting in the inner product direction (along k-dimension) of matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ respectively, using Algorithm 1. Note that as $\boldsymbol{A}_{\mathtt{split}}$ and $\boldsymbol{B}_{\mathtt{split}}$ can be stored on FP16, $\mathtt{GEMM}_{\mathtt{FP32}}$ can be performed by FP16 computations with FP32 precision on Tensor Cores through the cublasGemmEx routine in cuBLAS.

3.2 Fast Computation Techniques

To further improve the performance, the following methods modify Algorithm 1 or 2. Implementation-based speedup techniques that do not change the algorithm will be discussed in Sect. 4.

Fast Mode. As implemented in OzBLAS, we define a parameter $d \in \mathbb{N}, d \le \mathtt{max}(s_x, s_y)$ that determines the number of split matrices in the computation. With d specified, we can omit the computations $p + q > d + 1$ in $(\boldsymbol{x}^{(p)})^T \boldsymbol{y}^{(q)}$ in exchange for a small loss of accuracy. If the required accuracy is FP64 (equivalent to the standard DGEMM, as performed by the method that determines the

Algorithm 3. Determination of the number split matrices required to achieve the DGEMM equivalent accuracy with fast mode ($\boldsymbol{A} \in \mathbb{F}_{\texttt{FP64}}{}^{m\times k}$, $\boldsymbol{B} \in \mathbb{F}_{\texttt{FP64}}{}^{k\times n}$)

```
1: function ($d = \texttt{DetermineNumSplitMats}(m, n, k, \boldsymbol{A}, \boldsymbol{B})$)
2:     $(\boldsymbol{A}_{\texttt{split}}[1 : s_A], \boldsymbol{c}_A[1 : s_A]) = \texttt{SplitA}(m, k, \boldsymbol{A})$    // from line 2 in Algorithm 2
3:     $\boldsymbol{e} = (1, ..., 1)^T$
4:     $\boldsymbol{s} = \texttt{fl}_{\texttt{FP64}}(2\sqrt{k}\boldsymbol{u}_{\texttt{FP64}}(|\boldsymbol{A}|(|\boldsymbol{B}|\boldsymbol{e})))$
5:     $d = 2$
6:     while (1) do
7:         $\boldsymbol{t} = \texttt{fl}_{\texttt{FP64}}((d+1)(|2^{\boldsymbol{c}_A[d]_i}\boldsymbol{A}_{\texttt{split}}[d]_{ij}|(|\boldsymbol{B}|\boldsymbol{e})))$
8:         if ($\boldsymbol{s}_i > \boldsymbol{t}_i$ for $1 \le i \le m$) then
9:             break
10:        end if
11:        $d = d + 1$
12:    end while
13: end function
```

number of split matrices, described next), the accuracy loss is negligible. This technique reduces the number of matrix multiplications to $d(d+1)/2$ from d^2 at most.

Estimating the Number of Split Matrices that Achieves FP64-equivalent Accuracy. Splitting by Algorithm 1 automatically stops when $\mu = 0$; that is, when the accuracy of the final result is maximized. However, if the required accuracy is that of standard DGEMM performed on FP64 arithmetic (FP64-equivalent accuracy), we can estimate the minimum required number of splits by Algorithm 3 based on the probabilistic error bound [9] as

$$|\texttt{fl}_{\texttt{FP64}}(\boldsymbol{AB}) - \boldsymbol{AB}|\boldsymbol{e} \lesssim 2\sqrt{k}\mathbf{u}_{\texttt{FP64}}|\boldsymbol{A}||\boldsymbol{B}|\boldsymbol{e} \tag{1}$$

where $\boldsymbol{e} = (1, ..., 1)^T$ is introduced to avoid matrix multiplication in the estimation (note that at line 7 in Algorithm 3, $2^{\boldsymbol{c}_A[d]_i}\boldsymbol{A}_{\texttt{split}}[d]_{ij}$ is d-th non-downscaled split matrix stored on FP64. Hence, `SplitA` at line 2 does not necessarily need to perform until s_A, and `SplitA` and this algorithm can be integrated). This algorithm is designed to operate in fast mode. If the split number is determined such that Algorithm 1 executes until $\mu = 0$, the accuracy may be lower than that of standard DGEMM. In this case, we must disable the fast mode. Note that, a certain degree of difference between the desired (achieved by standard DGEMM) and obtained is expected in this method, because the number of split matrices is just estimated based on the probabilistic error bound, and will also be influenced by the vector $\boldsymbol{e}$.

Blocking Against Inner Product. This step is not implemented in the present study. As ρ in Algorithm 1 includes n, the dimension of the inner product, the number of splits required to achieve a certain accuracy depends on the inner-product-wise dimension of the matrix. Its increase can be avoided by employing a blocking strategy against the inner product-wise operations. The blocking size can be set to the minimum size that achieves the best performance. However, this strategy increases the summation cost; moreover, changing the block size may disturb the reproducibility except when the correctly-rounded computation is performed.

4 Implementation

4.1 Basic Design

Our DGEMM implementations, computing $\boldsymbol{C} = \alpha \boldsymbol{AB} + \beta \boldsymbol{C}$, using Tensor Cores are referred to as DGEMM-TC, and two versions are implemented as described below.

– **DP-mode**: This mode achieves FP64-equivalent accuracy. The number of split matrices is determined automatically by Algorithm 3. Fast mode is automatically applied if possible. The summation is performed in FP64 arithmetic.
– **CR-mode**: This mode achieves the correctly-rounded result when $\alpha = 1$ and $\beta = 0$. The splitting iterates until all elements of the split matrices are zero. Fast mode is disabled. The summation is performed with NearSum when $\alpha = 1$ and $\beta = 0$ or in FP64 arithmetic in other cases.

We also implemented SGEMM-TC in SP-mode, which corresponds to the FP32 version of DGEMM-TC in DP-mode.

Our implementations are interface-compatible with the standard DGEMM and SGEMM routines, except for an argument for the pointer to a handler that holds some parameters including the address pointing to the working memory of the Ozaki scheme. The working memory is wholly allocated outside the BLAS routine to avoid the allocation time. The allocation is performed through a BLAS initialization function, which must be called in advance, similar to cublasInit in cuBLAS. In our implementation of Algorithm 1, `max` (at line 12) is obtained on the register through the shared memory, whereas $\boldsymbol{x}$ is accessed at line 10. For downscaling (and upscaling in the summation), 2^n is computed by `scalbn` (double x, int n), a function that computes $2^n x$. In DP-mode, Algorithms 1 and 3 are performed simultaneously as the latter includes the former. The computation part is performed by cublasGemmEx, a matrix multiplication routine that uses Tensor Cores. This routine has several internal implementations[3], and can be selected as an argument. In this study, we used the default: CUBLAS_GEMM_DFALT_TENSOR_OP. α and β are computed in the summation process.

4.2 Optimization

Blocking to Reduce Memory Consumption. Memory consumption is reduced by a blocking technique applied to the outer-product-wise direction

[3] The details are not presented.

(note that this blocking differs from the inner-product-wise blocking discussed in Subsect. 3.2). All procedures are blocked by dividing a matrix into a rectangle with block size b_k. In our implementation, the block size is determined as $b_k = \lceil n/\lceil n/b_{max}\rceil\rceil$. This blocking technique may reduce the performance, as the memory consumption shrinks towards $b_k = 1$, because each matrix multiplication more closely approaches the inner product.

Further Performance Improvement. Although not attempted in this study, the performance can be improved in several ways from an implementation technique perspective.

First, as implemented in OzBLAS, the computations of split matrices can be performed with batched BLAS (i.e., cublasGemmEx can be replaced with cublasGemmBatchedEx) because each matrix multiplication can be performed independently. We observed that the performance was improved when the matrix size was very small, or when the number of split matrices was relatively large, but was degraded in other cases.

Second, as discussed in the paper [13], a sufficiently sparse split matrix can be represented in sparse matrix form. Split matrices holding higher or lower bits of the input matrices may contain many zero elements. If a high-performance sparse matrix-matrix multiplication routine using Tensor Cores is provided, we might enhance the performance by switching the dense operation to a sparse operation.

4.3 Expected Performance and Memory Consumption

The most computationally complex part of matrix multiplication by this scheme is multiplying the split matrices using cublasGemmEx, which has $O(n^3)$ complexity. Ideally, the overall performance is thus determined by the number of GEMMs called in the computation and the GEMM throughput. For d split matrices, the number of GEMM is d^2 in the standard method and $d(d+1)/2$ in fast mode. These values show how the performance overheads (in time) compare with that of a one-time execution of cublasGemmEx. However, our implementations contain several operations executed using FP64 FPUs. Whereas those portions have a computational complexity of $O(n^2)$ at most, they may affect the performance, if the hardware has limited FP64 support.

The memory consumption when $\boldsymbol{A} \in \mathbb{F}_{\texttt{FP64}}{}^{m\times k}$ with s_A split matrices and $\boldsymbol{B} \in \mathbb{F}_{\texttt{FP64}}{}^{k\times n}$ with s_B split matrices in the naive implementation (i.e., without the blocking technique) is $(s_A m + s_B n)k$ on FP16 for storing the split matrices. As shown in Algorithm 2, if the summation is performed immediately after each GEMM execution, $\boldsymbol{C}_{\texttt{tmp}}$ requires mn storage on FP32; however, in our implementation, owing to the convenience of implementing NearSum in CR-mode, all computation results are retained, requiring $s_A s_B mn$ of storage. After applying the blocking technique with block size b_k in both the m and n dimensions, the memory consumption reduces to $(s_A + s_B)kb_k + s_A s_B {b_k}^2$ $(0 < b_k \leq m, n)$. On the other hand, as the s_A and s_B are unknown before execution and the working memory is allocated before execution to avoid the memory allocation time, a certain amount of memory must be allocated at initialization. We then determine the maximum possible block size under the memory constraint. In addition to the above, several working memory spaces are needed. The memory consumption of our implementation is not yet optimized and should be improved in future work.

5 Evaluation

5.1 Experimental Settings

The performance was mainly evaluated on NVIDIA Titan RTX, a Turing architecture GPU with a compute capability of 7.5. The theoretical peak performance (with a boost clock of 1.77 GHz[4]) is 509.76 GFlops on FP64, 16312.32 GFlops on FP32, and 130498.56 GFlops[5] on Tensor Cores with FP32 precision. This GPU has more limited FP64 support than the Tesla series targeting HPC workloads (1/32 of FP32 and 1/256 of Tensor Cores). The memory is 24 GB GDDR6 at 672.0 GB/s. The host machine was equipped with an Intel Core i7-5930K CPU running CentOS Linux release 8.1.1911 (4.18.0-147.3.1.el8_1.x86_64), CUDA 10.2, and CUDA driver version 440.44. The GPU codes were compiled by nvcc release 10.2, V10.2.89 with compiler options "-O3 -gencode arch=compute_60, code=sm_75".

Further evaluations were conducted on NVIDIA Tesla V100 (PCIe 32GB), which offers rich FP64 support (1/2 of FP32 and 1/16 of Tensor Cores). The Tesla V100 is a Volta architecture GPU with compute capability 7.0, and its theoretical peak performance (with a boost clock of 1.38 GHz) is 7065.6 GFlops on FP64, 14131.2 GFlops on FP32, and 113049.6 GFlops on Tensor Cores with FP32 precision. The memory is 32 GB HBM2 at 898.0 GB/s. The host machine was equipped with an Intel Xeon Gold 6126 CPU running Red Hat Enterprise Linux Server release 7.7 (3.10.0-1062.18.1.el7.x86_64), CUDA 10.2, and CUDA driver version 440.33.01. The codes for this GPU were compiled by nvcc release 10.2, V10.2.89 with "-O3 -gencode arch=compute_60, code=sm_70".

For accurate evaluation, we averaged the results of 10 executions after three warm-up executions. In our proposed method, the number of split matrices required to achieve a certain accuracy depends on the range of the absolute values in each element of the input matrices. To observe the performance degradation arising from this range, we initialized the input matrices with $(\texttt{rand}-0.5)\times\texttt{exp}(\phi\times\texttt{randn})$, where `rand` is a uniform random number $[0,1)$ and `randn` is a random number selected from the standard normal distribution. The range of the absolute value of the input can be controlled by ϕ, and is widened by increasing ϕ. For example, fixing $m = n = k = 10240$ and varying $\phi = 0.1$, 1, and 2, the ranges were obtained as 9.8E−10 – 8.9E−01, 1.4E−09 – 1.6E+02,

[4] The actual clock can exceed the boost clock, depending on the individual product and the execution environment.

[5] 576 (Tensor Cores) $\times$ 1.77 (GHz) $\times 2 \times 4^3$ (Flops) = 130498.56 (GFlops).

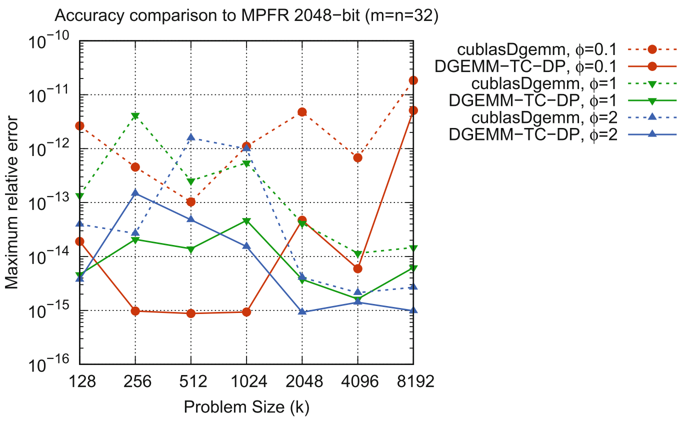


Fig. 3. Accuracy of cublasDgemm and DGEMM-TC in DP mode on Titan RTX. The maximum relative error is plotted against the results of MPFR 2048-bit. ϕ varies the range of the input values.

and 4.4E−10 – 4.8E+04, respectively. In all experiments, we allocated 20 GB to the working memory, and set the maximum block size to $b_k = 3584$. The scalar parameters were set as $\alpha = 1$ and $\beta = 0$.

5.2 DGEMM-TC

Figure 3 shows the accuracies of cublasDgemm and DGEMM-TC in DP-mode (DGEMM-TC-DP) for various input ranges (collected with different ϕ values) on Titan RTX. The maximum relative error is compared with the result of 2048-bit MPFR[6] [5] on FP64 (the results of MPFR are rounded to FP64). As the CR-mode with NearSum always obtained "zero," meaning that all the results were correctly-rounded, its results are omitted from Fig. 3. The accuracy of our implementation (solid lines) was equivalent to that of cublasDgemm (dotted lines), but some differences were observed, because our method (Algorithm 3) simply estimates the minimum number of split matrices that ensure similar accuracy to the classic DGEMM based on a probabilistic error bound of GEMM. The estimation further roughened by the $\boldsymbol{e} = (1, ..., 1)^T$ term that avoids matrix multiplications in the estimation.

Figure 4 shows the performance of DGEMM-TC in DP-mode (with FP64-equivalent accuracy) for the ϕ values. "Flops (on DP)" is the number of floating-point operations on FP64 per second when viewed as the standard DGEMM (i.e., it is computed as $2mnk/t$, where t denotes the execution time in seconds). Although the theoretical peak performance of the GPU on FP64 was only 510 GFlops (539 GFlops was observed on cublasDgemm with GPU boost), our implementation achieved up to approximately 980 GFlops (when $n = 7168$), outperforming cublasDgemm.

[6] http://www.mpfr.org.

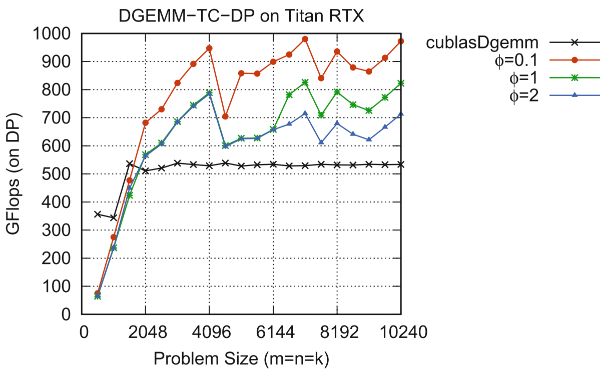
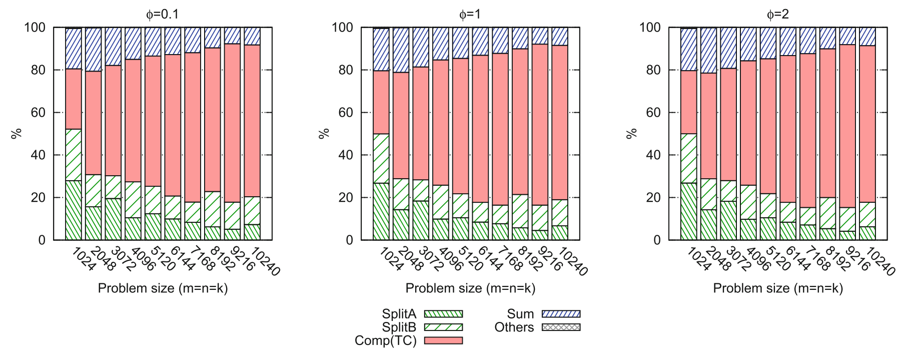
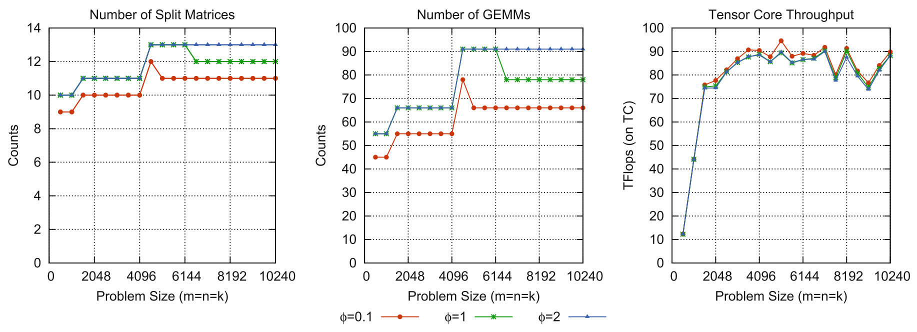


Fig. 4. Performance of DGEMM-TC in DP-mode (with FP64-equivalent accuracy) on Titan RTX. "Flops (on DP)" is the number of FP64 floating-point operations corresponding to the standard DGEMM. ϕ varies the range of the input values.



(a) Execution time breakdown



(b) The number of split matrices^{†1} and GEMMs^{†2} and Tensor Core throughput

Fig. 5. Details of DGEMM-TC in DP-mode (with FP64-equivalent accuracy) on Titan RTX. ϕ varies the range of the input values. †1: The same line plots the values of matrices $\boldsymbol{A}$ and $\boldsymbol{B}$. †1†2: Average over all blocks.

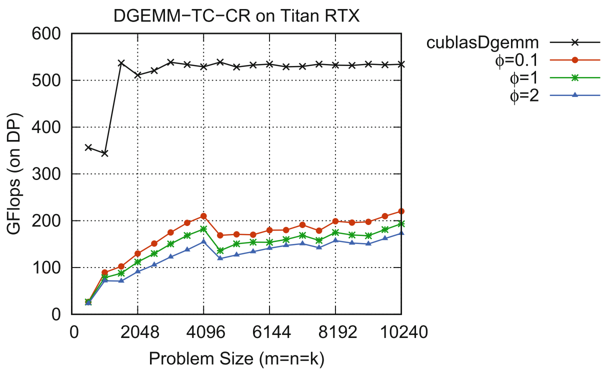
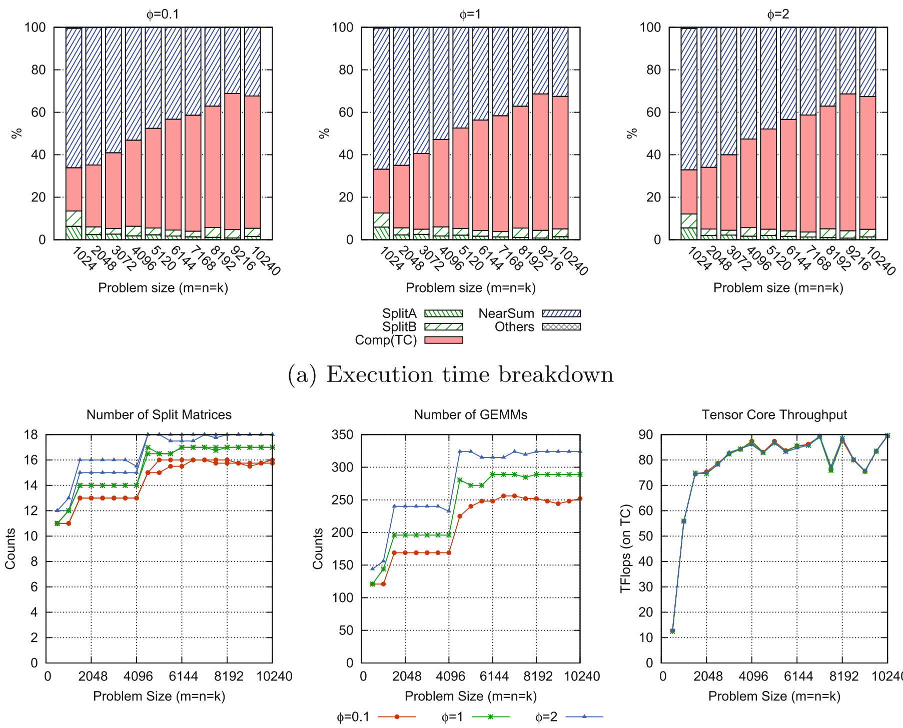


Fig. 6. Performance of DGEMM-TC in CR-mode (correctly-rounded) on Titan RTX. "Flops (on DP)" is the number of FP64 floating-point operations corresponding to the standard DGEMM. ϕ varies the range of the input values.



(b) The number of split matrices[†1] and GEMMs[†2] and Tensor Core throughput

Fig. 7. Details of DGEMM-TC in CR-mode (correctly-rounded) on Titan RTX. ϕ varies the range of the input values. †1: The same line plots the values of matrices $\boldsymbol{A}$ and $\boldsymbol{B}$. †1†2: Average over all blocks.

Table 1. Performance comparison ($m = n = k = 10240$) on Titan RTX and Tesla V100 (GFlops on DP). ϕ varies the range of the input values.

	Titan RTX			Tesla V100		
	$\phi = 0.1$	$\phi = 1$	$\phi = 2$	$\phi = 0.1$	$\phi = 1$	$\phi = 2$
cublasDgemm	534.1			6761		
DGEMM-TC-DP	972.4	823.0	713.1	1064	914.3	790.8
DGEMM-TC-CR	220.4	193.5	173.1	255.0	222.5	198.5
DGEMM-DP-CR	24.75	21.17	21.17	293.1	250.7	250.7

Additional performance analyses are shown in Fig. 5. Panel (a) shows the execution time breakdown for $\phi = 0.1$–2, observed in the tenth (final) execution. The execution time was dominated by the Tensor Cores computations. The splitting execution SplitA was slightly slower than SplitB because it included the cost of determining the number of splits, but SplitB was more costly than SplitA overall, because it was performed several times on the same portions of matrix $\boldsymbol{B}$. Such multiple executions were required by the blocking strategy. The left part of Fig. 5 (b) shows the number of split matrices (d) of $\boldsymbol{A}$ and $\boldsymbol{B}$ (plotted by the same line). The central part of Fig. 5 (b) plots the number of GEMMs called in the computation (d^2 or $d(d+1)/2$ in fast mode against the number of split matrices d). Finally, the right part of Fig. 5 shows the computational throughput on Tensor Cores (i.e., cublasGemmEx). Unlike the case in Fig. 4, the Flops value directly represents the number of floating-point operations performed on the Tensor Cores, and excludes all other computations. The actual performance can be understood through the following example: when $n = 7168$ and $\phi = 0.1$, we observed 980 GFlops. In this case, the number of GEMM calls was 66. The throughput of cublasGemmEx was approximately 92 TFlops on TC, and consumed approximately 70% of the total execution time. Hence, there were $0.7 \times 92/66 \approx 0.98$ TFlops on DP.

Figure 6 shows the performance of DGEMM-TC in CR-mode (DGEMM-TC-CR) (correctly-rounded) on Titan RTX. Figure 7 analyzes the performance in detail. The number of split matrices and GEMMs called in the computation can be decimals because the results were averaged over several blocks processed by the blocking technique. This mode degraded the performance because it increased the number of split matrices (and GEMMs), disabled the fast mode (i.e., affected the number of GEMMs), and increased the summation cost (NearSum is much costly than the standard FP64 summation).

Finally, Table 1 summarizes the performances of DGEMM-TC with $m = n = k = 10240$ on Titan RTX and on Tesla V100, which has rich FP64 support. For comparison, we also show the performance of a correctly-rounded DGEMM implementation (DGEMM-DP-CR), which is based on the Ozaki scheme but uses cublasDgemm instead of cublasGemmEx (hence, the computation was performed using FP64 instead of Tensor Cores). On Tesla V100, DGEMM-TC in DP-mode could not accelerate cublasDgemm.

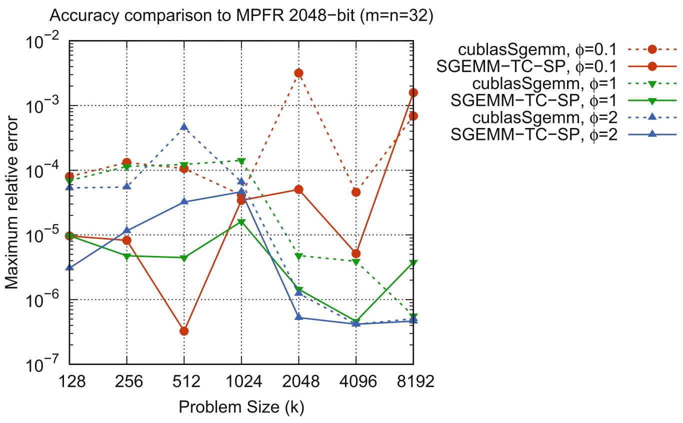


Fig. 8. Accuracy of cublasSgemm and SGEMM-TC in SP mode on Titan RTX. The maximum relative error is plotted against the results of MPFR 2048-bit. ϕ varies the range of the input values.

5.3 SGEMM-TC

Figure 8 shows the accuracy of cublasSgemm and SGEMM-TC in SP-mode for various input ranges (controlled by varying ϕ) on Titan RTX. The results are compared on FP32 (the results of MPFR are rounded to FP32).

Figure 9 shows the performance of SGEMM-TC in SP-mode. Similarly to DGEMM-TC on Tesla V100, our proposed method was useless for accelerating SGEMM on this GPU with fast FP32 support, but outperformed DGEMM-TC. The reason for the superior performance is not discernible from Fig. 9; however, the number of split matrices decreased, and the execution time of the splitting and summation parts was reduced in SGEMM-TC.

6 Discussion

This section discusses perspectives for introducing our proposed approach into hardware design. Although our method is limited to inner product based computations, it extends the application range of hardware with limited (or no) FP32/FP64 resources and fast low-precision processing units for general purpose workloads. Consequently, we can consider reducing the number of FP64 (or even FP32) FPUs, as discussed by Domke et al. [3], by exchanging them with low-precision FPUs such as Tensor Cores. Our rationale is supported by the following situations.

- The demand for AI workloads not requiring FP64 is increasing, and such work is becoming a significant part of the total workloads of HPC systems.
- The performance of large-scale computations is becoming communication-bound as the degree of parallelism of HPC systems increases.

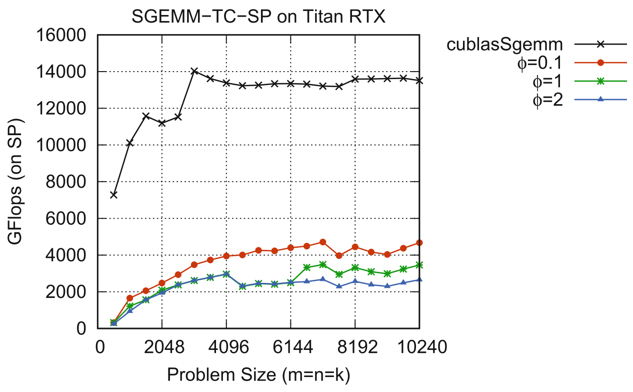


Fig. 9. Performance of SGEMM-TC in SP-mode (with FP32-equivalent accuracy) on Titan RTX. "Flops (on SP)" is the number of FP32 floating-point operations corresponding to the standard SGEMM. ϕ varies the range of the input values.

- The need for FP64 performance has reduced under the advance of mixed-precision techniques such as iterative refinement and precision-tuning.
- Low-precision hardware is easier to implement than high-precision hardware. In general, the complexity of p-bit precision hardware is $O(p^2)$. Currently, most processors only exploit the $O(p)$ benefit of instruction-level parallelism in single-instruction-multiple-data (SIMD) vectorization.
- Field-programmable gate arrays (FPGAs) are becoming a promising platform for HPC. Computations that do not fit into general processors can be accommodated by FPGAs. For instance, FPGAs can cover any "niche" demands for FP64 in future.

Accurate and high-precision computational methods, such as the proposed method and the other methods introduced in Sect. 2, may satisfy the "averaged" demand for the workloads requiring FP64 operations on an HPC system. Particularly in memory-bound operations, sufficient performance may be delivered by limited FP64 performance on hardware, or by software emulation of FP64 through multi-precision techniques; for example, "double-float" arithmetic as a float version of double-double arithmetic.

We now propose some hardware designs based on the Ozaki scheme. As described in Subsect. 3.1, the core concept of the Ozaki scheme is error-free transformation, which is similar to conversion from a matrix represented by floating-point numbers to matrices represented by fixed-point numbers. Accordingly, the length of the significand bit is important, and the exponent can be computed separately. Fast integer matrix multiplication (i.e., fast Arithmetic Logic Units) is desired for such a scheme because it requires fewer split matrices than the floating-point format for the same bit length. Moreover, this study effectively utilizes the Tensor Core design that computes FP16 data with FP32 precision and returns the result on FP32. Although the same idea can be implemented on standard FP16 FPUs, which adopt FP16 for both data format and computation, this implementation would increase the number of split matrices that achieve a given accuracy. This situation is worsened on BF16 FPUs, which have fewer significand bits. From this perspective, FPUs like the FP64 version of Tensor Cores are desired; as it computes $d = a \times b + c$ with FP64 accuracy, where a and b are FP32 and c and d are FP64. Such FPUs can adequately substitute full FP64 FPUs with the Ozaki scheme on DGEMM.

7 Conclusion

This paper presented an implementation technique for DGEMM and SGEMM using Tensor Cores that compute FP16 inputs with FP32 precision. Our method is based on the Ozaki scheme and is built upon cublasGemmEx, a GEMM implementation in cuBLAS performed on Tensor Cores. Besides providing a DGEMM and SGEMM compatible interface with equivalent accuracy, our technique can support accurate (correctly-rounded) and reproducible computations. The performance of our method depends on the range of the absolute values in each element of the input matrices. For instance, when matrices were initialized with random numbers over a dynamic range of 1E+9, and our DGEMM implementation with FP64-equivalent accuracy was run on Titan RTX with 130 TFlops on Tensor Cores, the highest achievement was approximately 980 GFlops of FP64 operation, although cublasDgemm can achieve only 539 GFlops on FP64 FPUs. The proposed method enhances the possibility of utilizing hardware with limited (or no) FP32/FP64 resources and fast low-precision processing units (such as AI-oriented processors) for general-purpose workloads. Furthermore, because the proposed method reduces the demand for FP64 FPUs in exchange for lower-precision FPUs, it will contribute to new perspectives of future hardware designs. Our code is available on our webpage[7].

Acknowledgment. This research was partially supported by the Japan Society for the Promotion of Science (JSPS) KAKENHI Grant Number 19K20286 and MEXT as "Exploratory Issue on Post-K computer" (Development of verified numerical computations and super high-performance computing environment for extreme researches). We thank Takeshi Terao (Shibaura Institute of Technology) for his helpful suggestion for the idea of the blocking toward inner product. This research used computational resources of Cygnus (for Tesla V100) provided by Multidisciplinary Cooperative Research Program in Center for Computational Sciences, University of Tsukuba.

References

1. Carson, E., Higham, N.: Accelerating the solution of linear systems by iterative refinement in three precisions. SIAM J. Sci. Comput. **40**(2), A817–A847 (2018)
2. Dekker, T.J.: A floating-point technique for extending the available precision. Numerische Mathematik **18**, 224–242 (1971)

[7] http://www.math.twcu.ac.jp/ogita/post-k/results.html.

3. Domke, J., et al.: Double-precision FPUs in high-performance computing: an embarrassment of riches? In: Proceedings 33rd IEEE International Parallel and Distributed Processing Symposium (IPDPS 2019), pp. 78–88 (2019)
4. Dongarra, J.J., Du Croz, J., Hammarling, S., Duff, I.S.: A set of level 3 basic linear algebra subprograms. ACM Trans. Math. Softw. **16**(1), 1–17 (1990)
5. Fousse, L., Hanrot, G., Lefèvre, V., Pélissier, P., Zimmermann, P.: MPFR: a multiple-precision binary floating-point library with correct rounding. ACM Trans. Math. Softw. **33**(2), 13:1–13:15 (2007)
6. Haider, A., et al.: The design of fast and energy-efficient linear solvers: on the potential of half-precision arithmetic and iterative refinement techniques. In: Shi, Y., et al. (eds.) ICCS 2018. LNCS, vol. 10860, pp. 586–600. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-93698-7_45
7. Haidar, A., Tomov, S., Dongarra, J., Higham, N.J.: Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers. In: Proceedings International Conference for High Performance Computing, Networking, Storage, and Analysis (SC 2018), pp. 47:1–47:11 (2018)
8. Henry, G., Tang, P.T.P., Heinecke, A.: Leveraging the bfloat16 artificial intelligence datatype for higher-precision computations. In: Proceedings 26th IEEE Symposium on Computer Arithmetic (ARITH-26), pp. 69–76 (2019)
9. Higham, N.J., Mary, T.: A new approach to probabilistic rounding error analysis. SIAM J. Sci. Comput. **41**(5), A2815–A2835 (2019)
10. Ichimura, S., Katagiri, T., Ozaki, K., Ogita, T., Nagai, T.: Threaded accurate matrix-matrix multiplications with sparse matrix-vector multiplications. In: Proceedings 32nd IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). pp. 1093–1102 (2018)
11. Markidis, S., Chien, S.W.D., Laure, E., Peng, I.B., Vetter, J.S.: NVIDIA tensor core programmability, performance precision. In: Proceedings 32nd IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 522–531 (2018)
12. Mukunoki, D., Ogita, T., Ozaki, K.: Reproducible BLAS routines with tunable accuracy using ozaki scheme for many-core architectures. In: Proceedings 13th International Conference on Parallel Processing and Applied Mathematics (PPAM2019), Lecture Notes in Computer Science, vol. 12043, pp. 516–527 (2020)
13. Ozaki, K., Ogita, T., Oishi, S., Rump, S.M.: Error-free transformations of matrix multiplication by using fast routines of matrix multiplication and its applications. Numer. Algorithms **59**(1), 95–118 (2012)
14. Rump, S., Ogita, T., Oishi, S.: Accurate floating-point summation part ii: Sign, k-fold faithful and rounding to nearest. SIAM J. Sci. Comput. **31**(2), 1269–1302 (2009)
15. Sorna, A., Cheng, X., D'Azevedo, E., Won, K., Tomov, S.: Optimizing the fast fourier transform using mixed precision on tensor core hardware. In: Proceedings 25th IEEE International Conference on High Performance Computing Workshops (HiPCW), pp. 3–7 (2018)
16. Yang, K., Chen, Y.F., Roumpos, G., Colby, C., Anderson, J.: High performance monte carlo simulation of ising model on TPU clusters. In: Proceedings International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2019), pp. 83:1–83:15 (2019)