DGEMM using Tensor Cores[1)2)]

Daichi Mukunoki[1] Katsuhisa Ozaki[2] Takeshi Ogita[3] Toshiyuki Imamura[1]

[1]RIKEN Center for Computational Science, Hyogo, Japan

[2]Shibaura Institute of Technology, Saitama, Japan

[3]Tokyo Woman's Christian University, Tokyo, Japan

Mar. 4, 2021, SIAM CSE21
MS: Mixed Precision Algorithms for High Performance Scientific Computing

---

[1)]This work has been published as "D. Mukunoki, K. Ozaki, T. Ogita, T. Imamura: DGEMM using Tensor Cores, and Its Accurate and Reproducible Versions, ISC 2020, LNCS, Vol. 12151, pp. 230-248, 2020"

[2)]This research was supported by JSPS KAKENHI Grant #19K20286 and MEXT as "Exploratory Issue on Post-K computer" (Development of verified numerical computations and super high-performance computing environment for extreme researches)

Introduction (1/2)

Mixed-precision techniques try to utilize low-precision operations as much as possible

Two goals:

1. To improve the speed & energy on current hardware: software improvement
2. To improve the speed & energy on future hardware, by replacing FP64 FPUs with lower-precision FPUs: hardware improvement
   - GPUs (not for HPC) eliminated FP64 for graphics tasks
   - AI-oriented processors eliminated FP64 for AI tasks

Introduction (2/2)

Backgroud

- The increasing demand for AI / deep learning has triggered the development of special processing units for fast low-precision (16-bit floating-point) operations
  - E.g., FP16 on NVIDIA Tensor Cores, BF16 on Google TPUs
- However, as those numerical representation is tiny, it is difficult to utilize them for general tasks other than deep learning

Floating-point formats

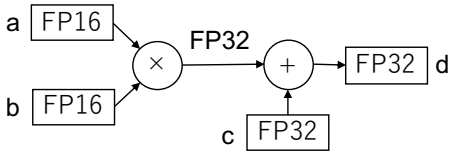
Format	Sign	Exponent	Fraction (hidden)	Decimal digits (approx.)
binary16 (FP16)	1	5	10 (1)	3.31
bfloat16 (BF16)	1	8	7 (1)	2.41
binary32 (FP32)	1	8	23 (1)	7.23
binary64 (FP64)	1	11	52 (1)	15.95

Related work

1. Applying low-precision to computations that do not require high accuracy to speed up the computation
   - Dense/sparse linear systems with iterative refinement using Tensor Cores (Haidar et al. 2018, Carson & Higham 2018)
   - Monte Carlo simulation of an Ising model using bfloat16 on TPUs (Yang et al. 2019)
2. Utilizing low-precision operations for more accurate computations
   - SGEMM using Tensor Cores (Markidis et al. 2018)
   - 2-fold precision arithmetic on bfloat16 (Henry et al. 2019)
   - 2D-FFT using Tensor Cores (Sorna et al. 2018)

- The latter ones do not necessarily aim to speed up the computation, but rather to extend the potential of low-precision hardware
- Our study also belongs to the latter

NVIDIA Tensor Cores



Tensor Core operation (FP16 with FP32 precision)

- Tensor Cores (Gen.2) can perform 4×4 matrix multiplication per clock with FMA that compute $d = a \times b + c$ with FP32 precision[3] (a, b: FP16, c, d: FP32)
- The Tensor Core operation is up to 8x faster than FP32 on CUDA Cores

[3] The other data formats and computational precisions are also supported

Overview

- **Proposal**
  - A method to compute DGEMM & SGEMM using the Tensor Core operation that computes FP16 inputs with FP32 precision
- **Performance**
  - On Titan RTX GPU (130 TFLOPS on Tensor Cores & 0.5 TFlops on FP64), up to 1 TFlops on DGEMM and 4.7 TFlops on SGEMM when the absolute range of the input values is 1e9
  - The performance depends on the absolute range of the input values

Ozaki scheme

Ozaki scheme (Ozaki et al. 2012[4]):

- Accurate inner product (or matrix multiplication) method
- Consist of 3 steps: for inner product $\boldsymbol{x}^T\boldsymbol{y}$,
  1. **Splitting** of vectors $\boldsymbol{x}, \boldsymbol{y}$ into split vectors respectively (element-wise)
  2. **Computation** of all-to-all product of the split vectors for $\boldsymbol{x}$ and the ones for $\boldsymbol{y}$
  3. **Summation** of the results of the above products (element-wise)
- The principle is similar to the one for other high-precision arithmetic methods (e.g., double-double): analogy to a product of multi-digit numbers:
  - $\boldsymbol{x} \cdot \boldsymbol{y} = (\boldsymbol{x}^{(1)} + \boldsymbol{x}^{(2)} + \boldsymbol{x}^{(3)}) \cdot (\boldsymbol{y}^{(1)} + \boldsymbol{y}^{(2)} + \boldsymbol{y}^{(3)})$
  - $123 \times 456 = (100 + 20 + 3) \times (400 + 50 + 6)$
- The performance depends on the absolute range of the input values

[4] K. Ozaki, T. Ogita, S. Oishi, S. M. Rump, Error-free transformations of matrix multiplication by using fast routines of matrix multiplication and its applications. Numer. Algorithms 59, 1, 2012

Ozaki scheme for Tensor Cores

For $r = \boldsymbol{x}^T\boldsymbol{y}$, ($\boldsymbol{x}$, $\boldsymbol{y} \in \mathbb{F}^n$, where $\mathbb{F}$ is the set of **FP64**)

$$\boldsymbol{x} = \sum_{p=1}^{s_x} 2^{c^{(p)}} \underline{\boldsymbol{x}}^{(p)}, \quad \boldsymbol{y} = \sum_{q=1}^{s_y} 2^{c^{(q)}} \underline{\boldsymbol{y}}^{(q)} \tag{1}$$

- $c^{(p)}$, $c^{(q)}$: exponent of $\boldsymbol{x}$, $\boldsymbol{y}$ (stored on 2byte-integer)
- $\underline{\boldsymbol{x}}^{(p)}$, $\underline{\boldsymbol{y}}^{(q)}$: significand of $\boldsymbol{x}$, $\boldsymbol{y}$ (stored on **FP16**)

Then, $\boldsymbol{x}^T\boldsymbol{y}$ can be computed accurately as follows:

$$\boldsymbol{x}^T\boldsymbol{y} = \sum_{p=1}^{s_x}\sum_{q=1}^{s_y} 2^{c_x{}^{(p)}+c_y{}^{(q)}} \underline{\boldsymbol{x}}^{(p)T}\underline{\boldsymbol{y}}^{(q)} \tag{2}$$

At (1), $\boldsymbol{x}, \boldsymbol{y}$ are split so that $\underline{\boldsymbol{x}}^{(p)T}\underline{\boldsymbol{y}}^{(q)}$ can be computed using **FP32** without rounding-errors ($\mathtt{fl}_{\mathtt{FP32}}$ denotes FP32 operations). Therefore,

$$\underline{\boldsymbol{x}}^{(p)T}\underline{\boldsymbol{y}}^{(q)} = \mathtt{fl}_{\mathtt{FP32}}\left(\underline{\boldsymbol{x}}^{(p)T}\underline{\boldsymbol{y}}^{(q)}\right) \tag{3}$$

- This can be computed using **Tensor Cores** as $\underline{\boldsymbol{x}}^{(p)}$, $\underline{\boldsymbol{y}}^{(q)}$ are stored on **FP16**

Ozaki scheme for Tensor Cores (cont'd)

Splitting of $\boldsymbol{x}, \boldsymbol{y}$ to $\underline{\boldsymbol{x}}^{(p)}$, $\underline{\boldsymbol{y}}^{(q)}$ are performed as (4)–(8) (recursively until $c^{(p)} = 0$. Note: in (4), we define $c^{(p)} \coloneqq 0$ when $\max_{1 \le i \le n} |\boldsymbol{x}^{(p)}{}_i| = 0$)

$$c^{(p)} \coloneqq \left\lceil \mathtt{fl}_{\mathtt{FP64}} \left(\log_2 \left(\max_{1 \le i \le n} \left| \boldsymbol{x}^{(p)}{}_i \right| \right) \right) \right\rceil \tag{4}$$

$$\sigma \coloneqq 2^{\rho + c^{(p)}} \tag{5}$$

$$\underline{\boldsymbol{x}}'{}_i \coloneqq \mathtt{fl}_{\mathtt{FP64}} \left(\left(\boldsymbol{x}^{(p)}{}_i + \sigma \right) - \sigma \right) \tag{6}$$

$$\boldsymbol{x}^{(p+1)}{}_i \coloneqq \mathtt{fl}_{\mathtt{FP64}} \left(\boldsymbol{x}^{(p)}{}_i - \underline{\boldsymbol{x}}'{}_i \right) \tag{7}$$

$$\underline{\boldsymbol{x}}^{(p)}{}_i \coloneqq \mathtt{fl}_{\mathtt{FP64}} \left(2^{-c^{(p)}} \underline{\boldsymbol{x}}'{}_i \right) \tag{8}$$

In (5), ρ is defined as following ($\mathtt{u}_{\mathtt{FPx}}$ denotes the unit round off of FPx)

$$\rho \coloneqq \left\lceil -\log_2 \mathtt{u}_{\mathtt{FP64}} + \frac{\log_2 \mathtt{u}_{\mathtt{FP32}} + \log_2 n}{2} \right\rceil \tag{9}$$

Ozaki scheme for Tensor Cores (on matrix multiplication)

Algorithm 1 $\boldsymbol{C} = \boldsymbol{AB}$ ($\boldsymbol{A} \in \mathbb{F}_{\texttt{FP64}}{}^{m\times k}$, $\boldsymbol{B} \in \mathbb{F}_{\texttt{FP64}}{}^{k\times n}$, $\boldsymbol{C} \in \mathbb{F}_{\texttt{FP64}}{}^{m\times n}$) using Ozaki scheme

function ($\boldsymbol{C} = \texttt{DGEMM-TC}(m, n, k, \boldsymbol{A}, \boldsymbol{B})$)
  $(\boldsymbol{A}_{\texttt{split}}[1 : s_A], \boldsymbol{c}_A[1 : s_A]) = \texttt{SplitA}(m, k, \boldsymbol{A})$ // $\boldsymbol{A}_{\texttt{split}}$ is obtained on FP16
  $(\boldsymbol{B}_{\texttt{split}}[1 : s_B], \boldsymbol{c}_B[1 : s_B]) = \texttt{SplitB}(k, n, \boldsymbol{B})$ // $\boldsymbol{B}_{\texttt{split}}$ is obtained on FP16
  $\boldsymbol{C}_{ij} = 0$
  for $(q = 1 : s_B)$ **do**
    for $(p = 1 : s_A)$ **do**
      $\boldsymbol{C}_{\texttt{tmp}} = \texttt{GEMM}_{\texttt{FP32}}(m, n, k, \boldsymbol{A}_{\texttt{split}}[p], \boldsymbol{B}_{\texttt{split}}[q])$
      $\boldsymbol{C}_{ij} = \boldsymbol{C}_{ij} + 2^{\boldsymbol{c}_A[p]_i + \boldsymbol{c}_B[q]_j} \boldsymbol{C}_{\texttt{tmp}_{ij}}$ // Computations for $1 \le i \le m$ and $1 \le j \le n$
    end for
  end for
end function

- $\texttt{GEMM}_{\texttt{FP32}}$[5] can be performed using cublasGemmEx using Tensor Cores as the inputs are FP16 (but cublasSgemm can also be used)

[5] This must be computed by the algorithm based on the standard inner product: divide-and-conquer approaches such as Strassen's algorithm are not permitted

Speedup techniques

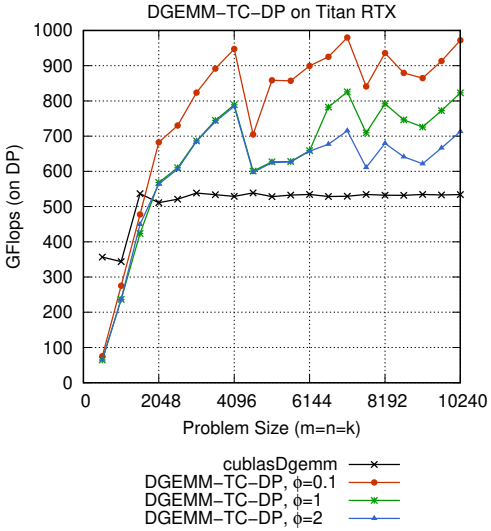
The basic Ozaki scheme is too accurate (as it performs infinite-precision operation) compared with the standard DGEMM

Two techniques to speedup and achieve the DGEMM-equivalent accuracy[6)]

- Fast-mode (reducing the number of GEMMs in the computation)
- Determination of the optimal number of split matrices to achieve the DGEMM-equivalent accuracy

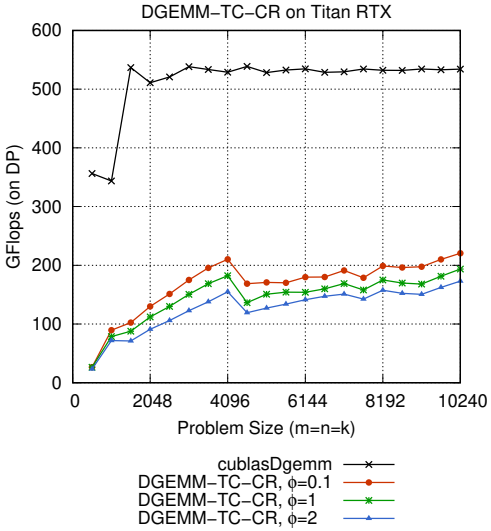
[6)] For details, see "D. Mukunoki, K. Ozaki, T. Ogita, T. Imamura: DGEMM using Tensor Cores, and Its Accurate and Reproducible Versions, ISC 2020, LNCS, Vol. 12151, pp. 230-248, 2020"

DGEMM-TC-DP (DGEMM-equivalent accuracy)



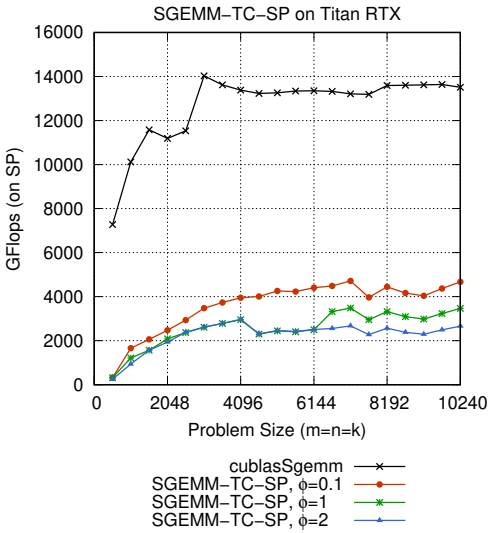
- GPU: **Titan RTX**
  - TC: 130 TFlops (FP16/FP32-mixed)
  - FP32: 16 TFlops
  - FP64: 0.5 TFlops
- Input: ϕ controls the input range:
  - $\phi = 0.1$: 9.3E-10 – 5.0E-01
  - $\phi = 1$: 1.4E-09 – 1.6E+02
  - $\phi = 2$: 4.4E-10 – 4.8E+04
  - Performance depends on the absolute range of the input values
- Performance
  - Up to 1TFlops when ϕ=0.1 (input-range: 1e9)
  - Faster than cubasDgemm as the GPU has only 0.5TFlops on FP64

DGEMM-TC-CR (infinite-precision with correct-rounding)



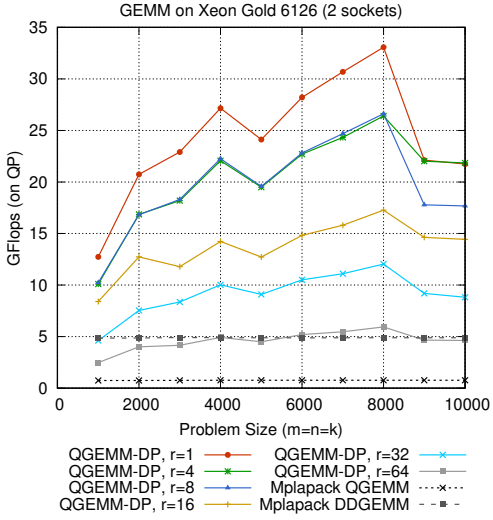
- Infinite-precision with correct-rounding using a correctly-rounded summation algorithm, NearSum (Rump et al. 2009)
- Performance decreased because the number of split matrices increased and NearSum was costly

SGEMM-TC-SP (SGEMM-equivalent)



- Up to 4.7 TFlops, but slower than cublasSgemm on this GPU that has fast FP32

FP128-GEMM (QGEMM) using DGEMM



- **QGEMM-DP**: FP128-GEMM using DGEMM + ICC's FP128 emulation
  - FP128 (IEEE binary128), 15-bit exponent + 113-bit significand
  - ICC's FP128 emulation is used for splitting & summation in Ozaki scheme
- Preliminary result
  - Some speedup techniques used in DGEMM-TC are not applied
  - Input range is controlled by r to be $[1, 10^r)$
  - This environment can achieve 1300 GFlops (on DP) on MKL's DGEMM

Applications

Using Ozaki scheme,

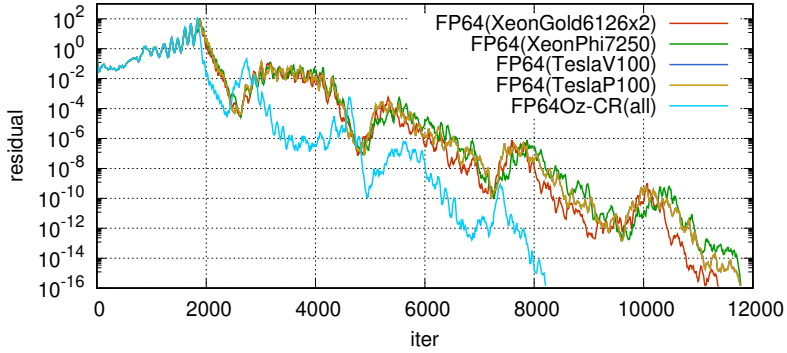
- SGEMM using TC-GEMM
- DGEMM using TC-GEMM or SGEMM
- QGEMM using TC-GEMM or SGEMM or DGEMM
- ...

Ozaki scheme can deliver reproducibility.

- Accurate and reproducible BLAS "OzBLAS" (DOT, GEMV, GEMM, SpMV)
  - Currently, OzBLAS[7)] is built on FP64 but also can be built on FP32, TC, etc.
- Accurate and reproducible CG based on OzBLAS

[7)] D. Mukunoki, T. Ogita, K. Ozaki: Reproducible BLAS Routines with Tunable Accuracy Using Ozaki Scheme for Many-core Architectures, PPAM2019, LNCS, Vol. 12043, pp. 516-527, 2020.

Reproducible CG with Ozaki scheme[8)]



Convergence plot of matrix "pdb1HYS" ($||\boldsymbol{r}_i||/||\boldsymbol{b}||$, at every 10 iters)

- FP64Oz-CR (using Ozaki scheme) is identical on 4 platforms

[8)] D. Mukunoki, K. Ozaki, T. Ogita, R. Iakymchuk, Conjugate Gradient Solvers with High Accuracy and Bit-wise Reproducibility between CPU and GPU using Ozaki scheme, HPCAsia 2021

Conclusion

- **Proposal**
  - DGEMM / SGEMM using Tensor Cores (FP32/FP16-mixed) based on the Ozaki scheme, which can be built upon cublasGemmEx
  - And, the accurate (infinite-precision) version and FP128-GEMM using DGEMM
- **Performance**
  - On Titan RTX (130 TFLOPS on TC, 0.5 TFlops on FP64), up to 1 TFlops on DGEMM and 4.7 TFlops on SGEMM when the input value range is 1e9
  - The performance depends on the input range
- **Expectation**
  - Extends the potential of Tensor Cores and stimulate discussion on the precision that hardware should support
- GEMM-TC and OzBLAS: http://www.math.twcu.ac.jp/ogita/post-k/results.html