Accurate DGEMM using Tensor Cores

Daichi Mukunoki
RIKEN Center for Computational Science, Kobe, Japan

Katsuhisa Ozaki
Shibaura Institute of Technology, Saitama, Japan

Takeshi Ogita
Tokyo Woman's Christian University, Tokyo, Japan

1 INTRODUCTION

As the demand for deep-learning increases, specialized hardware dedicated to low-precision matrix-multiplication, which is the kernel of those tasks, has been developing. Tensor Cores, which are equipped on recent NVIDIA GPUs, are capable of computing a matrix-multiplication on FP16 inputs with FP32 accuracy and return the result on FP16 or FP32 format. As it is 4 or 8 times faster than FP32, many studies have been trying to utilize it on general tasks as well. However, the applicable tasks are still limited, owing to the tiny bit-length. This paper presents a novel way to utilize Tensor Cores for matrix-multiplication (aka GEMM) with higher accuracy than FP16. With that method, we can even create DGEMM (GEMM on FP64), which is a kernel operation of many HPC tasks as well as high-performance Linpack (HPL).

2 METHOD AND IMPLEMENTATION

Our approach, the Ozaki scheme [2], computes an accurate matrix-multiplication as the summation of several matrix-multiplications which can be computed without rounding-errors using the standard floating-point operations. The input matrices are element-wisely split into several matrices. It can achieve the correctly-rounded result (computed with only one-rounding), but the accuracy can be tunable. The number of split matrices (d) required to achieve a certain accuracy depends on the range of the absolute values in the input matrices, the inner-product wise dimension, and the precisions of the computation and input data. It requires d^2 matrix-multiplications, and its cost becomes dominant in the total execution time. Our implementation is based on our previous work, OzBLAS [1], using FP64 (DGEMM) on GPUs. It is designed to achieve the correctly-rounded result with NearSum [3] in the computation of $C = AB$. However, to utilize Tensor Cores, we modify it as follows: (1) For splitting the input matrices on FP64 into FP16, we need to change a parameter determining the number of split matrices from the case for splitting into FP64, used in [1]. As a result, the number of split matrices increases compared to the case for FP64. Here, we also need to scale down the exponent on FP64 to fit in FP16. The shift value is determined for each row on matrix A or column on B and must be kept to recover later. (2) The computations of the split matrices are performed with cublasGemmEx, which performs a matrix-multiplication of FP16 matrices with the FP32 accuracy and return the result on FP32 using Tensor Cores. (3) The computed results, obtained on FP32, need to be scaled up by shifting the exponent to compensate the scale down done previously.

3 PERFORMANCE

Table 1 shows the performance on NVIDIA Titan V, a Volta architecture GPU, on CUDA 10.0. The theoretical peak performance is 110 TFlops on Tensor Cores and 6.9 TFlops on FP64. The cublasGemmEx was called with CUBLAS_GEMM_DFALT_TENSOR_OP.

Table 1: DGEMM ($C = AB$, $m = n = k = 10240$) with correct-rounding using Tensor Cores on Titan V

Input[†1]	Performance	# of split	Tensor Core usage	
(exp. range)	GFlops[†2]	matrices	TFlops[†3]	Ratio[†4]
ϕ=0 (-10/-01)	1264	7	89.7	69%
ϕ=1 (-09/+02)	167.6	16	53.2	78%
ϕ=2 (-10/+04)	126.4	18	52.4	78%
ϕ=3 (-10/+07)	105.3	20	53.8	78%
ϕ=4 (-11/+09)	85.73	22	52.9	79%

†1: initialized as (rand − 0.5) × exp(ϕ × randn), where rand is an uniform random number [0, 1) and randn is a random number from the standard normal distribution. †2: obtained as $2n^3/t$, where t is the execution time in sec (as with standard DGEMM). †3: performance of cublasGemmEx alone. †4: the ratio of cublasGemmEx in the total execution time.

We note that our implementation achieves the correctly-rounded result, which means that it is more accurate than cublasDgemm. When the number of split matrices is 7, 49 cublasGemmEx are performed. While $89.7e3/49 = 1830$ GFlops was expected, 1264 GFlops was obtained owing to the overhead for the other costs from the splitting, scaling, and summation processes. When the number of split matrices was greater than 7, the cublasGemmEx performance was decreased because the computed matrix size became small with memory-blocking due to the memory constraint.

4 CONCLUSION

We proposed an implementation of a DGEMM compatible routine using Tensor Cores. Although our approach brings no performance advantage on Titan V, which supports fast FP64, it can be beneficial on hardware with limited FP64 support such as NVIDIA Tesla T4, whose FP64 performance is 1/256 of the Tensor Cores. Therefore, our approach opens the way to utilize AI-oriented hardware for more general purposes and may impact the system co-design. The same idea can be applied to create SGEMM or to create DGEMM using FP32 GEMM as well as even Bfloat16. More detailed discussions, including some techniques to improve the performance more, are presented in our poster.

Acknowledgements. This work was supported by MEXT as Exploratory Issue on Post-K computer (Development of verified numerical computations and super high-performance computing environment for extreme researches) and JSPS KAKENHI #19K20286.

REFERENCES

[1] D. Mukunoki, T. Ogita, and K. Ozaki. 2019. Accurate and Reproducible BLAS Routines with Ozaki Scheme for Many-core Architectures. In *Proc. International Conference on Parallel Processing and Applied Mathematics (PPAM2019).* (to appear).
[2] K. Ozaki, T. Ogita, S. Oishi, and S.M. Rump. 2012. Error-free transformations of matrix multiplication by using fast routines of matrix multiplication and its applications. *Numer. Algorithms* 59, 1 (2012), 95–118.
[3] S. Rump, T. Ogita, and S. Oishi. 2009. Accurate Floating-Point Summation Part II: Sign, K-Fold Faithful and Rounding to Nearest. *SIAM Journal on Scientific Computing* 31, 2 (2009), 1269–1302.