"Accurate Matrix Multiplication on Binary128 Format Accelerated by Ozaki Scheme"[1)]

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

[1]RIKEN Center for Computational Science
daichi.mukunoki@riken.jp

[2]Shibaura Institute of Technology

[3]Tokyo Woman's Christian University

Aug 12, 2021, ICPP 2021

[1)]This research was supported by the Japan Society for the Promotion of Science (JSPS) KAKENHI Grant #19K20286. This research used computational resources of the Cygnus supercomputer provided by Multidisciplinary Cooperative Research Program in Center for Computational Sciences, University of Tsukuba.

Introduction (1/3)

IEEE 754-2008 binary128 (15-bit exponent + 113-bit significand)

- Software implementations (emulations) are available (GCC, ICC, Berkeley SoftFloat[2)]), but extremely slow
- Hardware implementation is still rare (IBM Power9, FPGA)
- High-precision is needed in some apps and also to suppress the increasing rounding error in large-scale computations

Linear algebra library supporting binary128

- MPLAPACK[3)]: multi-precision BLAS & LAPACK using GCC's binary128 and high-precision arithmetic libraries (GMP, MPFR, QD)

---

2) J. Hauser, http://www.jhauser.us/arithmetic/SoftFloat.html.

3) M. Nakata, MPLAPACK, https://github.com/nakatamaho/mplapack

Introduction (2/3)

Double-double (DD) arithmetic[4] – a fast substitute for quadruple precision

- Double-word arithmetic built upon binary64 arithmetic; incompatible with binary128 (11-bit exponent + 106-bit significand) but faster
- QD[5] (DD & quad-double (QD)) is known as an implementation on x86
- $\approx$ 20x slower vs. binary64 on GEMM if well-optimized – hand-SIMDization is necessary on CPUs[6]

[4] T. J. Dekker, A Floating-Point Technique for Extending the Available Precision, Numer. Math. 18, 1971.
[5] Y. Hida, X.S. Li, D.H. Bailey, Quad-Double Arithmetic: Algorithms, Implementation, and Application, Lawrence Berkeley National Laboratory Technical Report, LBNL-46996, 2000.
[6] K. Tomonori, Acceleration of multiple precision matrix multiplication based on multi-component floating-point arithmetic using AVX2, arXiv:2101.06584, 2021.

Introduction (3/3)

Our proposal

- Fast & accurate implementation of matrix multiplication on binary128 matrices on x86 CPUs – faster than MPLAPACK's binary128- & DD-GEMM

Contributions

- An extension of DGEMM using Tensor Cores[7)] to binary128-GEMM using DGEMM – high-prec. GEMM is computed using low-prec. GEMM by Ozaki scheme[8)]
- Specific optimizations for binary128 with binary64
- More extensions: GPU acceleration, SGEMM-based implementation, reduced-precision performance, mat-vec, & distributed parallel implementation

[7)] D. Mukunoki, K. Ozaki, T. Ogita, T. Imamura, DGEMM using Tensor Cores, and Its Accurate and Reproducible Versions. ISC 2020.

[8)] 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 (1/2)

For inner product of $\boldsymbol{x}, \boldsymbol{y} \in {\mathbb{F}_{\mathtt{b128}}}^n$ ($\mathbb{F}_{\mathtt{b128}}$: the set of binary128 numbers)

(1) Splitting

Input vectors are split into several split-vectors, resp. (element-wise, from higher to lower bits)

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

- ${c_x}^{(p)}$ and $\underline{\boldsymbol{x}}^{(p)}$ correspond to the exponent and significand of $\boldsymbol{x}$, resp. (same for $\boldsymbol{y}$)
- Splitting is performed so that ${\underline{\boldsymbol{x}}^{(p)}}^T \underline{\boldsymbol{y}}^{(q)}$ at (2) is error-free in binary64
- Num. of splits increases depending on the absolute range of input elements & dimension

(2) All-to-all product, and (3) Summation

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

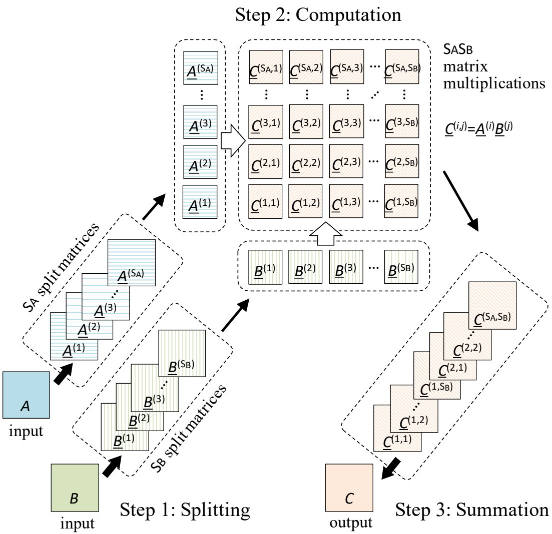
- $s_x s_y$ inner-products are computed – they can be computed in binary64 (i.e., DDOT)
- To obtain binary128-level accuracy, the summation can be computed in binary128 (infinite-prec. is achieved if summed in infinite-prec.)

Ozaki scheme (2/2)

On mat-mul

- All-to-all product of split matrices are computed using DGEMM
- Exec. time is DGEMM dominant
- Exec. time increases with the square of the num. of split matrices – it increases depending on the absolute range of input elements & the inner-product dimension

Performance is input-dependent



Optimization (1/2)

Reducing binary128 operations (for improving performance)

- Binary128 operations used in splitting & summation can degrade performance
- **Split3**: input binary128 vector $\boldsymbol{x}$ is split into three binary64 vectors such that $\boldsymbol{x} = \underline{\boldsymbol{x}} + \underline{\boldsymbol{x}}_2 + \underline{\boldsymbol{x}}_3$ with $|\underline{\boldsymbol{x}}_i| \geq |\underline{\boldsymbol{x}}_{2i}| \geq |\underline{\boldsymbol{x}}_{3i}|$ (113 bits $\rightarrow$ 53 + 53 + 7 bits). Then, we use the splitting algorithm for binary64 in the original Ozaki scheme[9)]
- **Sum3**: accumulating binary64 values to a binary128 value using three binary64 bins (i.e., 159-bit precision) – an adaptation of VecSum[10)]
- Note: both can only be used when the input is in the exponent range of binary64

[9)] K. Ozaki et al., 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.

[10)] D. M. Priest, Algorithms for arbitrary precision floating point arithmetic, ARITH 1991, 1991.

Optimization (2/2)

Summation order (for preventing accuracy loss)

- In the Ozaki scheme (below), the smaller p and q are, the digits with larger absolute values are held in $\underline{\boldsymbol{x}}^{(p)}$ and $\underline{\boldsymbol{y}}^{(q)}$, respectively

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

- Large accuracy loss may occur when a large cancellation occurs during the summation; better to sum the data in decreasing order of $p + q$

Blocking (for saving memory)

- Performing the entire procedure (i.e., split, comp, & sum) in a block manner by dividing a matrix into a rectangle along with the inner product direction

Implementation

Implementation on x86

- Splitting & summation are parallelized using OpenMP (`parallel for` to the outermost loop where memory accesses are discontinuous)
- DGEMM is computed using Intel MKL

Since the execution time is DGEMM-dominant, good performance can be expected by utilizing highly-optimized BLAS without elaborate optimizations 😎

Experimental setup (1/2)

Experimental setup

- Intel Xeon Gold 6126 (Skylake, 2.6–3.7 GHz, 12 cores) $\times$ 2 sockets with 192 GB DDR4-2666 RAM (255.9 GB/s)
- Executed with 1 thread/core (24 threads in total) with "`numactl --localalloc`"
- g++ 8.3.1 with -O3
- Intel MKL 19.1.3
- 64GB work memory (this can be reduced by blocking – blocking size is automatically determined)

Experimental setup (2/2)

Comparison

- **Oz-b128**: Proposed implementation using Ozaki scheme with Split3 & Sum3
- **MP-b128**: MPLAPACK's GEMM using binary128 (with GCC's emulation)
- **MP-dd**: MPLAPACK's GEMM using DD arithmetic (with QD v2.3.22)

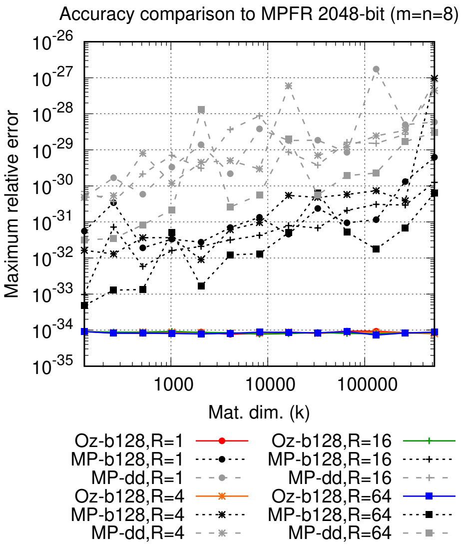
– GEMMs of MPLAPACK (v0.9.3): based on the naive mat-mul algorithm with triple loops parallelized using OpenMP (but not SIMDized)

Problem setting

- Input matrices are initialized with pseudo uniform random numbers $[1, 10^{R})$ with random sign and evaluated the performance at different R
  – because the performance of Oz-b128 is input-dependent: num. of split matrices increases depending on the absolute range of inputs

Maximum relative error of Oz-b128, MP-b128, & MP-dd vs. 2048-bit MPFR on different input ranges (varied by R)

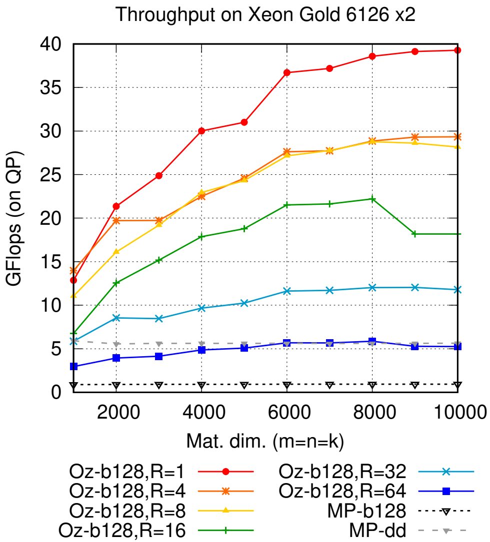
- All the Oz-b128 results overlap
- Oz-b128 achieves higher accuracy as most computations are performed with error-free (except summation)



Evaluation – throughput

Throughput of Oz-b128 on different input ranges (varied by R), MP-b128, and MP-dd on R=1

- "Flops (on QP)": the value obtained by $2n^3/t$, where t is execution time
- Oz-b128 can outperform MP-b128 & MP-dd, while the performance is input-dependent
  – But MP-dd may have room for performance improvement with SIMD optimization
- Note: this environment can achieve approx. 1600 GFlops on DGEMM (Oz-b128 has 40x overhead at best)

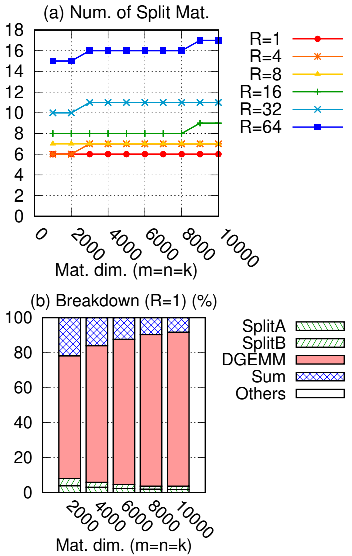


Num. of split mat.

- Depends on the absolute range of the input elements (varied by R) & inner product dimension (k-dim)

Execution time breakdown (R=1)

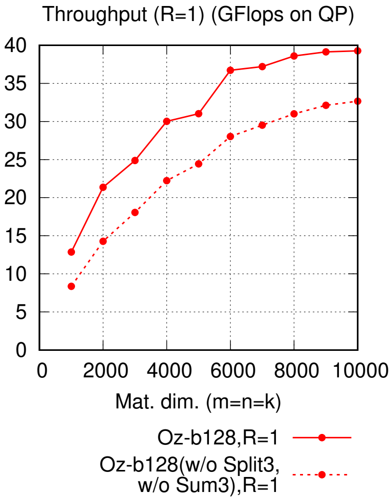
- DGEMM dominant (the larger R, the more so)



Evaluation – throughput w/o Split3 & Sum3

Throughput w/o Split3 & Sum3 (R=1) (dotted line)

- Throughput was reduced to 83% (at $n = 10000$) as the Split and Sum costs increase 2.7x and 2.1x, resp.



Extensions

GPU acceleration

- Easy to accelerate by offloading DGEMMs

SGEMM-based implementation

- SGEMM can be used instead of DGEMM (but no performance merit on this CPU)

Performance on reduced-precision inputs

- Performance increases as num. of split mat. decreases

Memory-bound operation (mat-vec)

- Oz-b128 is faster than MM-b128 but slightly slower than MM-DD

Distributed parallel implementation

- Two ways (discussion only)

Details and demonstrations are available in the paper

Summary

Fast & accurate matrix multiplication on binary128 matrices using Ozaki scheme

- Faster than MPLAPACK's binary128- and DD-GEMM
  - but DD-GEMM has room for performance improvement with SIMD optimization

Advantages

- High-performance and low development cost (can be built upon DGEMM)
  - also, easy to accelerate using GPUs
- Accurate – most computations are performed with error-free

Disadvantages

- Throughput is input dependent
- Large memory consumption (but can be relaxed by blocking)

Code is available as part of OzBLAS[11)]

[11)] https://www.r-ccs.riken.jp/labs/lpnctrt/projects/ozblas/