Infinite-Precision Inner Product and Sparse Matrix-Vector Multiplication Using Ozaki Scheme with Dot2 on Manycore Processors

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

[1] RIKEN Center for Computational Science, Kobe, Hyogo, Japan
daichi.mukunoki@riken.jp
[2] Shibaura Institute of Technology, Saitama, Japan
[3] Tokyo Woman's Christian University, Tokyo, Japan

Abstract. Infinite-precision operations do not incur rounding errors except when rounding the computed result to a finite-precision value. This can be an effective solution for the accuracy and reproducibility concerns associated with floating-point operations. This research presents an infinite-precision inner product (IP-DOT) and sparse matrix-vector multiplication (IP-SpMV) on FP64 data for manycore processors. We propose using a 106-bit computation using Dot2 in the Ozaki scheme, which is an existing IP-DOT method. First, we discuss the theoretical performance of our method using the roofline model. Then, we demonstrate the actual performance as IP-DOT and reproducible conjugate gradient (CG) solvers, with IP-SpMV as their primary operation, using an Ice Lake CPU and an Ampere GPU. Although the benefits and performance are dependent on the input data, our experiments on IP-DOT demonstrated a speedup of approximately 1.9–3.4 times compared to the previous method, and an execution time overhead of approximately 10–25 times compared to the standard FP64 operation. On reproducible CG, a speedup of 1.1–1.7 times was achieved compared to the existing method, and an execution time overhead of approximately 3–19 times was observed compared to the non-reproducible standard solvers.

Keywords: Infinite-precision · Accurate · Reproducible · Inner product · Sparse matrix-vector multiplication (SpMV) · Conjugate gradient (CG)

1 Introduction

Floating-point operations are susceptible to rounding errors, which might lead to inaccurate computational result. Additionally, since a change in the order of operation causes different errors, the output may vary even when the same input is used on parallel computations where the order of operations is non-deterministic for each execution or in different hardware (e.g., CPUs and GPUs). This can be troublesome when debugging or porting codes to multiple environments [1]. Thus, computing methods that are both accurate and reproducible are being developed.

Infinite-precision operations do not incur rounding errors except when rounding the computed result to a finite-precision value, such as in FP64. This can be an effective solution for the accuracy and reproducibility concerns associated with floating-point operations[1]. Furthermore, infinite-precision operations can be utilized as a tool to analyze the mathematical behavior of numerical algorithms [27]. However, one of the major drawbacks of infinite-precision operations is their high runtime and program development costs, especially on modern manycore processors.

This research focuses on the infinite-precision inner product (IP-DOT) and sparse matrix-vector multiplication (IP-SpMV) for FP64 data on manycore processors. It proposes a fast computation method by combining an existing infinite-precision method with a 106-bit precision operation algorithm. IP-DOT and IP-SpMV are then implemented on an Ice Lake CPU and an Ampere GPU. The advantage of the proposed method is not only justified theoretically but also demonstrated as a speedup of IP-DOT separately and a speedup of reproducible sparse iterative solvers based on IP-DOT and IP-SpMV on matrices selected from a database collecting real-world problems.

2 Related Work

Several arithmetic tools, including iRRAM [20], RealLib [13], and Briggs's work [3], have been developed to enable infinite-precision computation. Its efficient implementation for vector and matrix operations (i.e., Basic Linear Algebra Subprograms (BLAS) operations) on parallel architectures can be investigated; for example, RARE-BLAS [4], ExBLAS [5], and OzBLAS [17] have been developed. OzBLAS adopts the same methodology that is referenced as an existing method in this paper.

Reproducible computation[2] does not necessarily require infinite-precision. The simplest way to ensure reproducibility is to fix the order of computation, although this is often inefficient in parallel computing. The Intel Math Kernel Library (MKL) supports conditional numerical reproducibility [26], but this is restricted to limited environments (with MKL on certain Intel processors) and execution conditions. ReproBLAS [7] is a reproducible BLAS implementation that uses a high-precision accumulator and pre-rounding technique but is not parallelized on manycore processors.

[1] Be aware, however, that infinite-precision operations do not necessarily improve the stability or accuracy of numerical algorithms.

[2] The concept of reproducibility is independent of accuracy. It is simply intended to be able to reproduce the same result.

Algorithm 1 Ozaki scheme with Dot2

```
1: function ($r$ = IP_DOT_Dot2($\boldsymbol{x}, \boldsymbol{y}$))
2:     $\underline{\boldsymbol{x}}[1 : s_x]$ = Split2($\boldsymbol{x}$)
3:     $\underline{\boldsymbol{y}}[1 : s_y]$ = Split2($\boldsymbol{y}$)
4:     $i = 1$
5:     for $q = 1 : s_y$ do
6:         for $p = 1 : s_x$ do
7:             $(u, v)[i]$ = Dot2($\underline{\boldsymbol{x}}[p]$, $\underline{\boldsymbol{y}}[q]$)
8:             $i = i + 1$
9:         end for
10:    end for
11:    $r$ = NearSum($(u, v)$)
12: end function
```

Algorithm 2 Dot2

```
1: function ($(u, v)$ = Dot2($\boldsymbol{x}, \boldsymbol{y}$))
2:     $(u, v)$ = TwoProdFMA($\boldsymbol{x}_1, \boldsymbol{y}_1$)
3:     for $i = 2$ to $n$ do
4:         $(h, r)$ = TwoProdFMA($\boldsymbol{x}_i, \boldsymbol{y}_i$)
5:         $(u, q)$ = TwoSum($u, h$)
6:         $v = \mathrm{fl}(v + (q + r))$
7:     end for
8: end function
```

Algorithm 3 Splitting for Ozaki scheme with Dot2. Lines 9–10 are computations for $1 \le i \le n$.

```
1: function ($\underline{\boldsymbol{x}}[1 : s_x]$ = Split2($\boldsymbol{x}$))
2:     $\rho$ = ceil(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:         $\tau$ = ceil(log2($\mu$))
8:         $\sigma = 0.75 \times 2^{(\rho+\tau)}$
9:         $\underline{\boldsymbol{x}}[j]_i$ = fl($(\boldsymbol{x}_i + \sigma) - \sigma$)
10:        $\boldsymbol{x}_i$ = fl($\boldsymbol{x}_i - \underline{\boldsymbol{x}}[j]_i$)
11:        $\mu = \max_{1 \le i \le n}(|\boldsymbol{x}_i|)$
12:    end while
13:    $s_x = j$
14: end function
```

The use of high-precision arithmetic (in lower than infinite but better than FP64 precision) can be a lightweight solution for improving accuracy (without reproducibility). MPLAPACK [21] is an example of a linear algebra library that supports various high-precision operations with a backend of several high-precision arithmetic libraries such as the GNU Multiple Precision Floating-Point Reliable Library [9]. However, it is often difficult to determine the required level of precision for a specific objective.

3 Method

Hereafter, $\mathbb{F}_{\texttt{FP64}}$ will denote a set of FP64 floating-point numbers, and $\texttt{fl}(\cdot)$ will denote the FP64 floating-point operations. The objective is to compute $r = \boldsymbol{x}^T\boldsymbol{y}$ for $\boldsymbol{x}, \boldsymbol{y} \in {\mathbb{F}_{\texttt{FP64}}}^n$ with infinite precision.

Originally proposed as an accurate matrix multiplication technique, the Ozaki scheme [23] is employed in this research as an IP-DOT method. This scheme computes an IP-DOT as the sum of multiple inner products that can be calculated with some precision and without rounding errors using floating-point operations. Algorithm 1 shows the entire IP-DOT process. It consists of the following three steps:

Algorithm 4 TwoSum

```
1: function ((s, e) = TwoSum(a, b))
2:     $s = \mathrm{fl}(a + b)$
3:     $t = \mathrm{fl}(s - a)$
4:     $e = \mathrm{fl}((a - (s - t)) + (b - t))$
5: end function
```

Algorithm 5 TwoProdFMA

```
1: function ((p, e) = TwoProdFMA(a, b))
2:     $p = \mathrm{fl}(a \times b)$
3:     $e = \mathtt{FMA}(a \times b - p)$
4: end function
```

1. **Splitting**: In lines 2–3 of Algorithm 1, `Split2` (Algorithm 3) performs the element-wise splitting of the input vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ into $\underline{\boldsymbol{x}}$ and $\underline{\boldsymbol{y}}$ (FP64 vectors). `Split2` divides the input vectors so that the inner products of the split vectors ($\underline{\boldsymbol{x}}$ and $\underline{\boldsymbol{y}}$) can be computed with 106-bit precision without rounding errors. In line 8 of Algorithm 3, the constant 0.75 was introduced by [15]. Due to the possibility of overflow in this splitting technique, the inner product using the Ozaki scheme accepts a narrower input range than the standard inner product using FP64 arithmetic.
2. **Computation**: In line 7 of Algorithm 1, `Dot2` [22] (Algorithm 2) computes the inner products of the split vectors with at least 106-bit precision and returns the result in 106-bit as a pair of FP64 values[3]. `Dot2` is built utilizing `TwoSum` [12] (Algorithm 4) and `TwoProdFMA` [11] (Algorithm 5). $\mathtt{FMA}(a \times b - p)$ denotes the calculation of $a \times b - p$ using the fused multiply-add (FMA) operation. Note that although Dot2 is composed of FP64 arithmetic, the term “FP64” will henceforth refer to the absence of Dot2 usage. In lines 5–10 of Algorithm 1, several inner inner products can be computed using general matrix multiplication (GEMM) by combining multiple split vectors into a matrix. This is a key aspect of the implementation process. The use of GEMM is beneficial from a performance perspective because it permits data reuse.
3. **Summation**: In Algorithm 1, the infinite-precision result of IP-DOT is first obtained as an array of a pair of FP64 values ($[u, v]$) with a length of $s_x \times s_y$. Then, in line 11, the IP-DOT result in the FP64 format is obtained with NearSum [25], which is a correctly-rounded summation algorithm.

This scheme applies naturally to other inner-product-based operations, including SpMV. There are two observations in SpMV. First, in Algorithm 3, the number of non-zero elements in each row can be used instead of n. Second, just as GEMM was used for DOT, the computation can be performed using sparse-matrix dense-matrix multiplication (SpMM) by combining the split vectors into a matrix.

[3] The original Dot2 algorithm is designed to obtain the output as an FP64 value with $\mathtt{fl}(u + v)$ at the end.

The performance of this scheme is input-dependent; it is determined by the numbers of split vectors (s_x and s_y)[4]. Each of them depends on the absolute range, the number of significant digits of the elements in the input vector (lines 3 and 11 of Algorithm 3), and the vector length n (line 2 of Algorithm 3). As demonstrated in Sect. 5, it is often expected to be around 2 to 3 for real problems. Thus, the GEMM utilized in the computation is usually very skinny. Additionally, the summation cost using NearSum is expected to be relatively small in terms of overall execution time, as the summed elements are $s_x \times s_y \times 2$ (2 is the pair of FP64 values), which is typically small enough compared to n.

Existing studies, such as [17], use FP64 (or lower precision [18]) for computation, but our proposal in this research is to use 106-bit operations using Dot2 for the computation (i.e., GEMM in DOT and SpMM in SpMV) and the corresponding modification at line 2 in Algorithm 3. This permits the packing of more bits into the split vectors ($\underline{\boldsymbol{x}}$, $\underline{\boldsymbol{y}}$), thereby reducing the number of split vectors. In contrast, there are concerns regarding the increase in execution time due to the additional computational cost required by Dot2. In practice, however, the cost of Dot2 can be ignored in memory-intensive operations, as discussed in [16]. Our method yields skinny-shaped GEMM and SpMM that are sufficiently memory-intensive, and operate in Dot2 with memory-bound performance. As a result, the throughput is unaffected when using Dot2 instead of FP64. We provide a theoretical explanation of this in the next section.

4 Performance Estimation

4.1 Throughput of GEMM and SpMM Using Dot2

To demonstrate that the use of Dot2 does not reduce the throughput of GEMM and SpMM relative to FP64, we first estimate the throughput of them computed using Dot2 and FP64. We intend to use Xeon Platinum 8360Y (Ice Lake, 36 cores) later in the evaluation. Note that this discussion almost reaches the same conclusion also for the GPU (A100-SXM4-40) used in this research. The SpMV uses the compressed sparse row (CSR) format with 32-bit indices.

The roofline model [28] estimates the achievable throughput of the target kernel in bytes/s (B)

$$B = \mathtt{min}(B_{\mathtt{CPU}}, O_{\mathtt{CPU}} \times Q/W) \quad (1)$$

using the following parameters:

- $B_{\mathtt{CPU}}$: the memory throughput of the CPU in bytes/s
- $O_{\mathtt{CPU}}$: the computation throughput of the CPU in Ops/s
- Q: the target kernel's memory traffic in bytes
- W: the number of operations of the target kernel in Ops.

[4] In fact, it is even possible to adjust the accuracy of the result by varying the number of split vectors. The result will no longer be infinite precision, but reproducibility can still be preserved. See [17] for details.

Note that we use "Ops" as the number of operations per second to represent the throughput of Dot2 and FP64 on the same scale (i.e., an inner product for $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{F}_{\mathtt{FP64}}{}^n$ performs $2n$ (Ops) in both Dot2 and FP64).

For Q and W in the GEMM and SpMM, we assume the following parameters:

- d: number of split vectors/matrices
- n: dimensions of vectors/matrices ($n \times n$)
- n_{nz}: number of non-zero elements of the sparse matrix in SpMM.

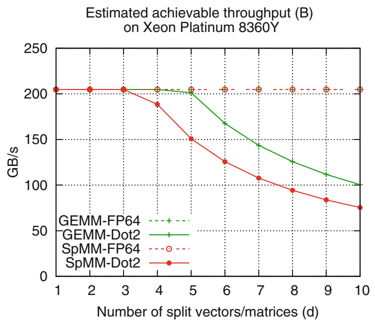

Fig. 1. Estimated achievable throughput (B) of GEMM and SpMM.

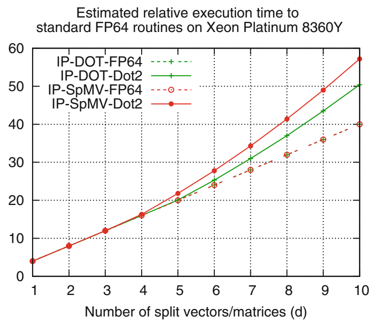

Fig. 2. Estimated relative execution times compared to the standard FP64 routines.

The GEMM computes $C_{d\times d} = A_{n\times d}{}^T B_{n\times d}$ and the SpMM computes $C_{n\times d} = A_{n\times n} B_{n\times d}$. Thus, assuming that data reusability is fully considered, the Q and W are as follows:

- GEMM: $Q = 16dn$ (bytes), $W = 2d^2n$ (Ops)
- SpMM: $Q = 12n_{nz}$ (bytes), $W = 2dn_{nz}$ (Ops) (assuming $n_{nz} \gg n$).

For $B_{\mathtt{CPU}}$ and $O_{\mathtt{CPU}}$, the target CPU has the following theoretical peak hardware performance parameters:

- $B_{\mathtt{CPU}} = 204.8$ GB/s
- FP64: $O_{\mathtt{CPU}} = 1382.4$ GOps/s
- Dot2: $O_{\mathtt{CPU}} = 125.7$ GOps/s (1/11 of the case in FP64 as it requires 11 times the number of floating-point instructions).

Using the above parameters with Eq. (1), the throughput of GEMM and SpMM in bytes/s (B) is estimated, as shown in Fig. 1. In this figure, we denote "-FP64" and "-Dot2" for operations computed by FP64 and Dot2, respectively (the same hereinafter). When d is small, both FP64 and Dot2 can be executed in the same amount of time as they are memory-bound. However, when d is large, Dot2 becomes computational-bound, and the memory throughput decreases. Here, d serves as a parameter that controls the arithmetic intensity for the roofline model.

4.2 Performance of IP-DOT and IP-SpMV

Next, we discuss the total execution time of IP-DOT and IP-SpMV. We first estimate the relative execution time compared with the standard DOT and SpMV using FP64 arithmetic (DOT-FP64 and SpMV-FP64, respectively). As discussed in [17], based on the number of memory read/written to vectors and matrices, the relative execution time is estimated to increase by a factor of $4d$, depending on d. The splitting process accounts for $3d$ of the $4d$, and the remaining d is attributable to the computation utilizing GEMM-FP64 (for DOT) or SpMM-FP64 (for SpMV), with the assumption that their performance is memory-bound and achieves $B_{\texttt{CPU}}$. However, the estimated achievable throughput B is depicted in Fig. 1 as discussed in Sect. 4.1. Accordingly, as shown in Fig. 2, the relative execution times of IP-DOT and IP-SpMV are projected to be $(3 + B_{\texttt{CPU}}/B)d$ times slower compared to DOT-FP64 and SpMV-FP64. The required d is problem-dependent; however, if the situation is similar to that demonstrated in the next section, d is no more than 7 with FP64, and using Dot2 can reduce d by half or less.

We then discuss a practical rather than a theoretical outlook on performance. Although up to three-quarters of the execution time is attributable to the splitting process (Algorithm 3), it is a straightforward memory-bound operation that poses no implementation challenges for manycore processors. The remaining one-fourth, which results from matrix multiplications (GEMM or SpMM), can be problematic. There are two issues present. First, since the highly-optimized implementation of GEMM-Dot2 and SpMM-Dot2 are not readily available, one must create it themselves. Second, which concerns not only in Dot2 but also in FP64, is that GEMM for very skinny matrices, performed in our scheme, may require a different optimization strategy than GEMM for square matrices to achieve adequate performance. This problem is discussed in [8][5]. The aforementioned issues are certainly challenges in software development. However, GEMM for skinny matrices with FP64 and Dot2 have their independent uses and should be discussed independently from our method[6].

5 Demonstration on CPU and GPU

We demonstrate our method on DOT and conjugate gradient (CG) solvers, where SpMV is the primary operation, using a CPU and GPU of a node (Wisteria-A node) of the Wisteria/BDEC-01 system at the University of Tokyo. The specifics of the CPU and GPU environments are as follows:

- **CPU**: Intel Xeon Platinum 8360Y (Ice Lake, 36 cores, 1382.4 GFlops in FP64, 204.8 GB/s), Intel oneAPI 2022.1.2 (with ICC 2021.5.0 and MKL 2022.0.0), compiled with `-O3 -fma -fp-model source -fprotect-parens -qopenmp -march=icelake-server`, executed with `numactl --localalloc` using the same number of threads as the number of physical cores.

[5] This problem is not encountered in SpMM.
[6] For example, XBLAS [14] supports 106-bit operations.

– **GPU**: NVIDIA A100-SXM4-40GB (Ampere, 9.7 TFlops in FP64[7], 1555 GB/s), CUDA 11.4 (driver: 470.57.02), nvcc V11.4.152, compiled with "`-O3 -gencode arch= compute_60, code=sm_80`".

The codes are implemented in C++ with OpenMP and CUDA. They extend the existing implementations (the Ozaki scheme with FP64 operations) for CPUs and GPUs in [19]; however, there have been some improvements.

Table 1. Results of DOT ($n = 2^{25}$). Overhead is the relative execution time compared to the standard DOT with FP64 arithmetic (DOT-FP64).

	Abs. range of input	d	Theor. overhead	CPU		GPU	
				GB/s	Overhead	GB/s	Overhead
DOT-FP64	–	–	–	142.7	1	1314.0	1
IP-DOT-FP64	[1e0,1e1)	4	16	67.7	33.7	1105.3	19.0
	[1e0,1e4)	5	20	65.2	43.8	1022.6	25.7
	[1e0,1e8)	6	24	61.4	55.8	1126.8	28.0
	[1e0,1e16)	7	28	59.1	67.7	993.4	37.0
IP-DOT-Dot2	[1e0,1e1)	2	8	68.6	16.6	1043.2	10.1
	[1e0,1e4)	2	8	76.1	15.0	1039.2	10.1
	[1e0,1e8)	2	8	69.9	16.3	1038.4	10.1
	[1e0,1e16)	3	12	69.3	24.7	1055.8	14.9

5.1 DOT

As discussed in Sect. 4.2, the skinny GEMM employed in the computation represents a potential challenge in DOT. We developed not only GEMM-Dot2 but also GEMM-FP64 ourselves for comparison, which outperformed GEMM-FP64 of MKL and cuBLAS in the Ozaki scheme. They are implemented using the Advanced Vector Extensions 2 (AVX2) intrinsic and are parallelized along the long axis of the matrix; this can be described as an extension of the typical parallel implementation of DOT to compute multiple vectors.

Table 1 illustrates the performance for $n = 2^{25}$, which is sufficient to exceed the cache size. Since the performance depends on the absolute range of the elements of the input vectors, we demonstrate the performance for different inputs using a random number within the specified absolute value range. The number of split vectors (d) increases proportionally, and the theoretical overhead (relative execution time) multiplies by a factor of $4d$ compared with DOT-FP64, which is performed using the DOT routines of MKL and cuBLAS. In these cases, Dot2 decreased d by half or less compared to IP-DOT-FP64. On the

[7] 9.7 TFlops is the performance without Tensor Cores. 19.5 TFlops with Tensor Cores but cannot be used for Dot2.

CPU, the observed overhead is larger than the theoretical overhead because IP-DOT-FP64/Dot2 has a lower throughput (GB/s) than DOT-FP64 because of the insufficient performance optimization of GEMM-FP64/Dot2.

Table 2. Test matrices ($n \times n$ with n_{nz} non-zeros, sorted by n_{nz}/n).

#	name	n	n_{nz}	n_{nz}/n	kind
1	tmt_sym	726,713	5,080,961	7.0	electromagnetics problem
2	gridgena	48,962	512,084	10.5	optimization problem
3	cfd1	70,656	1,825,580	25.8	computational fluid dynamics problem
4	cbuckle	13,681	676,515	49.4	structural problem
5	BenElechi1	245,874	13,150,496	53.5	2D/3D problem
6	gyro_k	17,361	1,021,159	58.8	duplicate model reduction problem
7	pdb1HYS	36,417	4,344,765	119.3	weighted undirected graph
8	nd24k	72,000	28,715,634	398.8	2D/3D problem

5.2 Reproducible CG Solvers

IP-DOT and IP-SpMV are used to ensure reproducibility in CG solvers [10] [19]. These are simply intended to ensure reproducibility but not to improve the numerical stability or accuracy of the solution. We demonstrate the proposed method on existing reproducible CG solvers based on the Ozaki scheme [19]. Our implementations used in this evaluation are based on the codes of previous studies, with a few improvements[8]. The implementation overview of the reproducible CG solvers can be summarized as follows.

– The unpreconditioned CG algorithm is implemented. All data are stored in the FP64 format.
– All inner-product-based operations, including DOT, NRM2, and SpMV, are performed with infinite precision using the Ozaki scheme with NearSum. The implementations in Sect. 5.1 are used for DOT. NRM2 is implemented using DOT.
– For SpMV, the CSR format is used, and the symmetry of the matrix is not considered. The computation of SpMV was performed using SpMM. The GPU implementation of SpMM extends the vector-CSR [2] SpMV implementation to compute multiple vectors. The CPU implementation computes the output vector in parallel in threads, and the inner product computed in each thread is parallelized with AVX2.
– AXPY is implemented by explicitly using FMA.

[8] Major improvements: (1) use of [15], (2) use of in-house GEMM and SpMM with asymmetric splitting on CPUs, (3) use of more recent vendor libraries.

– The matrix splitting is required and performed only once before the iterations begin.
– The number of split matrices is reduced by using the asymmetric splitting technique [24], which shifts ρ at line 2 in Algorithm 3 for the matrix and vector (it contributes to reducing the number of SpMM computed, see [19] for details).

Eight matrices from [6] are used (Table 2) (those are the same ones used in [19]). For $\boldsymbol{Ax} = \boldsymbol{b}$, $\boldsymbol{b}$ and the initial solution $\boldsymbol{x}_0$ are $\boldsymbol{b} = \boldsymbol{x}_0 = (1, 1, ..., 1)^T$. The iteration is terminated when $||\boldsymbol{r}_i||/||\boldsymbol{b}|| \leq 10^{-16}$. Since the focus of this research is the speedup with Dot2, we do not present the numerical behavior (it is available in [19]), but the use of Dot2 does not affect the numerical behavior at the bit level. Hereafter, the reproducible CG solvers will be referred to as ReproCG-FP64 (existing method using FP64) and ReproCG-Dot2 (proposed method using Dot2), and the standard non-reproducible solvers implemented

Table 3. Execution time in seconds and the relative execution time compared to the standard CG (CG-FP64) (in parentheses).

#	CPU					GPU				
	CG-FP64	ReproCG -FP64		ReproCG -Dot2		CG-FP64	ReproCG -FP64		ReproCG -Dot2	
1	2.3e+0	7.5e+1	(32.5)	4.5e+1	(19.4)	1.6e+0	2.0e+1	(12.6)	1.8e+1	(11.2)
2	4.3e-1	3.9e+0	(9.1)	2.9e+0	(6.6)	3.3e-1	2.2e+0	(6.6)	1.8e+0	(5.3)
3	9.4e-1	7.8e+0	(8.3)	5.2e+0	(5.5)	4.7e-1	3.2e+0	(6.8)	2.5e+0	(5.4)
4	2.9e+0	3.7e+1	(12.7)	2.4e+1	(8.3)	2.9e+0	2.2e+1	(7.7)	1.5e+1	(5.4)
5	3.5e+1	3.9e+2	(10.9)	2.3e+2	(6.5)	1.8e+1	1.1e+2	(6.1)	9.4e+1	(5.2)
6	7.5e+0	8.3e+1	(11.0)	5.2e+1	(6.9)	7.2e+0	4.4e+1	(6.0)	3.1e+1	(4.2)
7	2.3e+0	1.8e+1	(7.6)	1.3e+1	(5.5)	1.7e+0	8.9e+0	(5.1)	6.0e+0	(3.5)
8	2.4e+1	8.9e+1	(3.7)	7.0e+1	(2.9)	5.4e+0	2.5e+1	(4.5)	1.7e+1	(3.2)

Table 4. Number of split matrices/vectors.

#	ReproCG-FP64					ReproCG-Dot2				
	matrix	vectors				matrix	vectors			
		min	max	med	avg		min	max	med	avg
1	3	3	7	5	4.9	2	2	3	2	2.0
2	2	3	5	4	4.0	2	2	2	2	2.0
3	3	3	6	4	4.1	3	2	3	2	2.0
4	5	4	7	4	4.0	3	2	3	2	2.0
5	3	4	6	5	4.8	2	2	3	2	2.0
6	5	4	7	4	4.0	3	2	3	2	2.0
7	3	3	5	4	4.0	2	2	2	2	2.0
8	3	3	5	4	4.2	2	2	3	2	2.0

using the BLAS routines in MKL and cuBLAS/cuSparse will be referred to as CG-FP64.

Table 3 illustrates the execution and relative execution times compared to CG-FP64. First, when compared to ReproCG-FP64, ReproCG-Dot2 achieved a speedup of 1.3–1.7 times on the CPU and a speedup of 1.1–1.5 times on the GPU. This range of performance improvement is supported by the reduction in the number of split matrices/vectors used in the computation, as depicted in Table 4. Dot2 reduced the number of split vectors, which varies during iterations, by about half, while the number of split matrices remained the same or decreased by no more than three-fifths. Next, ReproCG-Dot2 requires 2.9–19.4 times more execution time on the CPU and 3.2–11.2 times more execution time on the GPU than CG-FP64. These overheads are, in most cases, lower than those reported in [19] for reproducible CG performed using ExBLAS [10] for identical problems and conditions. As discussed in Sect. 4, in DOT, the Ozaki scheme incurs a $4d$-fold relative execution time overhead compared to the standard operation with

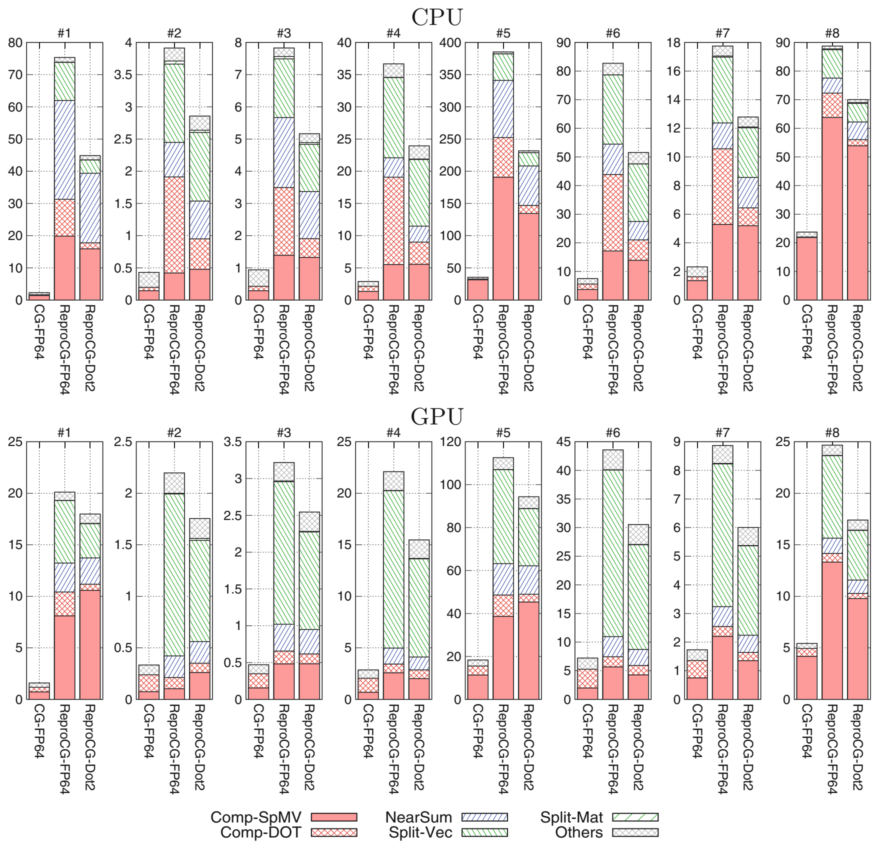

Fig. 3. Execution time breakdown (in seconds).

FP64 arithmetic, whereas in the CG method, the matrix is split only once before iterations. Thus, if SpMV is dominant in execution time, the optimum overhead would be d-fold. However, SpMV's influence on execution time diminishes as the matrix becomes more sparse (matrices are numbered in ascending order starting with the most sparse in Table 2). This explains why the overhead for highly sparse matrices is significant.

Figure 3 illustrates the execution time breakdowns to elaborate on the preceding results. Examining the computational cost for SpMV (Comp-SpMV, which is computed by SpMM), there are cases where the execution time has increased despite the decrease in the number of split matrices by Dot2. ReproCG-FP64 employs SpMM in MKL/cuSparse, while ReproCG-Dot2 uses in-house implementations. Since the kernel design has a large impact on the performance of SpMM, factors other than the Dot2 overhead may also be affected. Also, the observed NearSum overhead, particularly on the CPU, maybe a future concern.

6 Conclusion

This study presents an IP-DOT and IP-SpMV on FP64 data on CPU and GPU. We propose using 106-bit precision arithmetic (Dot2) rather than working precision (FP64) to compute the Ozaki scheme, which is an existing infinite-precision method. Although the performance depends on various conditions, including the input data, we demonstrate a theoretical and practical performance improvement of more than twofold in IP-DOT compared with the existing method using the Ozaki scheme with FP64 arithmetic, and the effectiveness of our approach increases as the input range increases. As a result, our IP-DOT requires approximately 10–25 times more execution time in reality (8–12 times in theory) than the standard DOT with FP64 arithmetic in MKL and cuBLAS. On CG solvers, a speedup of approximately 1.1–1.7 times is achieved compared to the existing method, and the overhead required to ensure reproducibility is approximately 3–19 times compared to the standard non-reproducible solvers.

Although this research successfully improves the performance of IP-DOT and IP-SpMV using the Ozaki scheme, the relative execution time compared to the standard FP64 operations is still significant. Furthermore, the Ozaki scheme is somewhat vulnerable to overflow. The superiority of this method, based on the Ozaki scheme, over other methods (ExBLAS and RARE-BLAS) is debatable. They have claimed lower overhead than our IP-DOT (e.g., RARE-BLAS [4] reported an overhead of 1–2 times at most on CPUs). However, our method offers the advantage of low development cost. It can be built upon matrix multiplication, enabling hierarchical software development and easy implementation on manycore processors, and it can be easily extended from DOT to other BLAS routines or tunable-accuracy operations with reproducible results, as demonstrated in [17]. We expect that, as a means to rapidly developing infinite-precision (accurate and reproducible) BLAS, our method is still an attractive option along with other faster methods. Also, it is a practical achievement to realize the lowest level of overhead for reproducible CG on both CPU and GPU.

This research utilized Dot2 as a swift quadruple-precision operation. However, a better alternative would be a hardware-implemented fast FP128 (with 113-bit mantissa), which would be capable of accelerating the infinite-precision operation of computationally intensive operations on FP64 data, such as matrix multiplication. Our research demonstrates that quadruple-precision arithmetic, such as FP128 and Dot2, is beneficial not only for accurate computations but also for reproducible computations in FP64 through infinite-precision operations.

Acknowledgment. This research was supported by the Japan Society for the Promotion of Science (JSPS) KAKENHI Grant #19K20286. This research was conducted using the FUJITSU Server PRIMERGY GX2570 (Wisteria/BDEC-01) at the Information Technology Center, The University of Tokyo (project #jh220022).

References

1. Arteaga, A., Fuhrer, O., Hoefler, T.: Designing bit-reproducible portable high-performance applications. In: Proceedings of IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS 2014), pp. 1235–1244 (2014). https://doi.org/10.1109/IPDPS.2014.127
2. Bell, N., Garland, M.: Implementing sparse matrix-vector multiplication on throughput-oriented processors. In: Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2009), pp. 1–11. No. 18 (2009). https://doi.org/10.1145/1654059.1654078
3. Briggs, K.: Implementing exact real arithmetic in python, c++ and c. Theoret. Comput. Sci. **351**(1), 74–81 (2006). https://doi.org/10.1016/j.tcs.2005.09.058
4. Chohra, C., Langlois, P., Parello, D.: Reproducible, accurately rounded and efficient BLAS. In: 22nd International European Conference on Parallel and Distributed Computing (Euro-Par 2016), pp. 609–620 (2016). https://doi.org/10.1007/978-3-319-58943-5_49
5. Collange, S., Defour, D., Graillat, S., Iakymchuk, R.: Numerical reproducibility for the parallel reduction on multi- and many-core architectures. Parallel Comput. **49**, 83–97 (2015). https://doi.org/10.1016/j.parco.2015.09.001
6. Davis, T.A., Hu, Y.: The university of Florida sparse matrix collection. ACM Trans. Math. Softw. **38**(1), 1:1–1:25 (2011). https://doi.org/10.1145/2049662.2049663
7. Demmel, J., Ahrens, P., Nguyen, H.D.: Efficient Reproducible Floating Point Summation and BLAS. Technical report. UCB/EECS-2016-121, EECS Department, University of California, Berkeley (2016)
8. Demmel, J., Eliahu, D., Fox, A., Kamil, S., Lipshitz, B., Schwartz, O., Spillinger, O.: Communication-optimal parallel recursive rectangular matrix multiplication. In: 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, pp. 261–272 (2013). https://doi.org/10.1109/IPDPS.2013.80
9. 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). https://doi.org/10.1145/1236463.1236468

10. Iakymchuk, R., Barreda, M., Graillat, S., Aliaga, J.I., Quintana-Ortí, E.S.: Reproducibility of parallel preconditioned conjugate gradient in hybrid programming environments. IJHPCA (2020). https://doi.org/10.1177/1094342020932650
11. Karp, A.H., Markstein, P.: High-precision division and square root. ACM Trans. Math. Softw. **23**, 561–589 (1997). https://doi.org/10.1145/279232.279237
12. Knuth, D.E.: The Art of Computer Programming. Seminumerical Algorithms, vol. 2. Addison-Wesley, Boston (1969)
13. Lambov, B.: Reallib: an efficient implementation of exact real arithmetic. Math. Struct. Comp. Sci. **17**(1), 81–98 (2007). https://doi.org/10.1017/S0960129506005822
14. Li, X.S., et al.: Design, implementation and testing of extended and mixed precision BLAS. ACM Trans. Math. Softw. **28**(2), 152–205 (2000). https://doi.org/10.1145/567806.567808
15. Minamihata, A., Ozaki, K., Ogita, T., Oishi, S.: Preconditioner for ill-conditioned tall and skinny matrices. In: The 40th JSST Annual International Conference on Simulation Technology (JSST2016) (2016)
16. Mukunoki, D., Ogita, T.: Performance and energy consumption of accurate and mixed-precision linear algebra kernels on GPUs. J. Comput. Appl. Math. **372**, 112701 (2020). https://doi.org/10.1016/j.cam.2019.112701
17. Mukunoki, D., Ogita, T., Ozaki, K.: Reproducible BLAS routines with tunable accuracy using Ozaki scheme for many-core architectures. In: Wyrzykowski, R., Deelman, E., Dongarra, J., Karczewski, K. (eds.) PPAM 2019. LNCS, vol. 12043, pp. 516–527. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-43229-4_44
18. Mukunoki, D., Ozaki, K., Ogita, T., Imamura, T.: DGEMM using tensor cores, and its accurate and reproducible versions. In: Sadayappan, P., Chamberlain, B.L., Juckeland, G., Ltaief, H. (eds.) ISC High Performance 2020. LNCS, vol. 12151, pp. 230–248. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-50743-5_12
19. Mukunoki, D., Ozaki, K., Ogita, T., Iakymchuk, R.: Conjugate gradient solvers with high accuracy and bit-wise reproducibility between CPU and GPU using Ozaki scheme. In: Proceedings of The International Conference on High Performance Computing in Asia-Pacific Region (HPC Asia 2021), pp. 100–109 (2021). https://doi.org/10.1145/3432261.3432270
20. Müller, N.T.: The irram: Exact arithmetic in c++. In: Computability and Complexity in Analysis. pp. 222–252. Springer, Berlin Heidelberg (2001). DOI: 10.1007/3-540-45335-0_14
21. Nakata, M.: Mplapack version 1.0.0 user manual (2021)
22. Ogita, T., Rump, S.M., Oishi, S.: Accurate sum and dot product. SIAM J. Sci. Comput. **26**, 1955–1988 (2005). https://doi.org/10.1137/030601818
23. 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). https://doi.org/10.1007/s11075-011-9478-1
24. Ozaki, K., Ogita, T., Oishi, S., Rump, S.M.: Generalization of error-free transformation for matrix multiplication and its application. Nonlinear Theory Appl. IEICE **4**, 2–11 (2013). https://doi.org/10.1587/nolta.4.2
25. Rump, S.M., 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). https://doi.org/10.1137/07068816X

26. Todd, R.: Introduction to Conditional Numerical Reproducibility (CNR) (2012). https://software.intel.com/en-us/articles/introduction-to-the-conditional-numerical-reproducibility-cnr
27. Wei, S., Tang, E., Liu, T., Müller, N.T., Chen, Z.: Automatic numerical analysis based on infinite-precision arithmetic. In: 2014 Eighth International Conference on Software Security and Reliability (SERE), pp. 216–224 (2014). https://doi.org/10.1109/SERE.2014.35
28. Williams, S., Waterman, A., Patterson, D.: Roofline: an insightful visual performance model for multicore architectures. Commun. ACM **52**(4), 65–76 (2009). https://doi.org/10.1145/1498765.1498785