Conjugate Gradient Solvers with High Accuracy and Bit-wise Reproducibility between CPU and GPU using Ozaki scheme

Daichi Mukunoki
daichi.mukunoki@riken.jp
RIKEN Center for Computational Science
Kobe, Hyogo

Katsuhisa Ozaki
ozaki@sic.shibaura-it.ac.jp
Shibaura Institute of Technology
Saitama, Japan

Takeshi Ogita
ogita@lab.twcu.ac.jp
Tokyo Woman's Christian University
Tokyo, Japan

Roman Iakymchuk
roman.iakymchuk@sorbonne-universite.fr
Sorbonne University
Paris, France
Fraunhofer ITWM
Kaiserslautern, Germany

ABSTRACT

On Krylov subspace methods such as the Conjugate Gradient (CG) method, the number of iterations until convergence may increase due to the loss of computational accuracy caused by rounding errors in floating-point computations. At the same time, because the order of the computation is nondeterministic on parallel computation, the result and the behavior of the convergence may be nonidentical in different computational environments, even for the same input. In this study, we present an accurate and reproducible implementation of the unpreconditioned CG method on x86 CPUs and NVIDIA GPUs. In our method, while all variables are stored on FP64, all inner product operations (including matrix-vector multiplications) are performed using the Ozaki scheme. The scheme delivers the correctly rounded computation as well as bit-level reproducibility among different computational environments. In this paper, we show some examples where the standard FP64 implementation of CG results in nonidentical results across different CPUs and GPUs. We then demonstrate the applicability and the effectiveness of our approach in terms of accuracy and reproducibility and their performance on both CPUs and GPUs. Furthermore, we compare the performance of our method against an existing accurate and reproducible CG implementation based on the Exact Basic Linear Algebra Subprograms (ExBLAS) on CPUs.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

KEYWORDS

Accuracy, reproducibility, Conjugate Gradient, heterogeneous computing, CPU, GPU

HPCAsia 2021, January 20–22, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8842-9/21/01.
https://doi.org/10.1145/3432261.3432270

ACM Reference Format:
Daichi Mukunoki, Katsuhisa Ozaki, Takeshi Ogita, and Roman Iakymchuk. 2021. Conjugate Gradient Solvers with High Accuracy and Bit-wise Reproducibility between CPU and GPU using Ozaki scheme. In *The International Conference on High Performance Computing in Asia-Pacific Region (HPCAsia 2021), January 20–22, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3432261.3432270

1 INTRODUCTION

Floating-point computations with finite precision introduce rounding errors with each operation; the accumulation of these errors may result in inaccuracy of the overall computation. At the same time, because floating-point computations are nonassociative, their results are nondeterministic (i.e., nonreproducible) if the order of the computation is not identical. As the scale of the computation grows toward Exascale computing and rounding errors accumulate, these issues may become more serious. In addition, the recent trend of introducing low-precision hardware increases the magnitude of rounding errors.

The reproducibility issue has until now received little focus. However, in view of the aforementioned rounding-error issues, along with the proliferation and heterogeneous of various processors, the importance of reproducibility has become more clear and needed. In fact, recent high-performance computing environments contain many factors that vary the order of computations, impacting the reproducibility of the computational result. These factors include parallel computation with different degrees of parallelism (various numbers of threads, processes, etc.), atomic operation on many-core architectures, the use or nonuse of the fused multiply-add (FMA) operation, and the introduction of auto-tuning and dynamic load balancing techniques. This issue of reproducibility may impact software debugging and cause problems for scientific activities that rely on the reproducibility of results. In addition, when code is ported to a new system, it can be a problem from the standpoint of reliability and quality control if it is not possible to distinguish whether differing results are being caused by a bug or just a rounding-error issue.

This paper focuses on the reproducibility issue of the Conjugate Gradient (CG) method, which is a Krylov subspace method and is often used for solving iteratively large sparse linear systems.

The CG method is a well-known example of computations that can be easily affected by rounding-errors; the number of iterations until convergence may increase due to the loss of computational accuracy, and the computation of residual is particularly sensitive. The computation result and the convergence behavior may be non-identical in different computational environments, even for the same input owing to rounding errors.

In this study, we present an accurate and reproducible implementation of the unpreconditioned CG method on x86 CPUs and NVIDIA GPUs. In our method, while all variables, including the coefficient matrix and all vectors, are stored on FP64 (IEEE binary64, a.k.a. double-precision), all inner product operations (including matrix-vector multiplications) are performed using the Ozaki scheme [17]. The scheme delivers the correctly rounded computation at each inner product operation as well as bit-level reproducibility among different computational environments – even between CPUs and GPUs. Here the "correctly rounded computation" means that for the inner product of two vectors stored on a working precision, the result is computed only with one rounding to the working precision at the end. First, we show some examples where the standard FP64 implementation of CG results in non-identical results across different CPUs and GPUs. We then demonstrate the applicability and the effectiveness of our implementations, in terms of accuracy and reproducibility, and their performance on both CPUs and GPUs. Furthermore, we compare the performance of our proposed method against an existing accurate and reproducible CG implementation based on the Exact Basic Linear Algebra Subprograms (ExBLAS)[1] [9, 10], which also performs correctly rounded operations for inner products.

The main contribution of this study is to show the adaptation of the Ozaki scheme to the CG method and discuss the behavior in terms of both performance and numerical results. The Ozaki scheme has already been adopted for an accurate and reproducible BLAS, OzBLAS[2] [13]; one of the chief advantages of this scheme is that it can be built upon standard BLAS implementations, such as Intel Math Kernel Library (MKL)[3] and NVIDIA cuBLAS[4]; good performance can be expected with low development cost. In fact, our proposed method can achieve comparable to or better performance than the ExBLAS approach in many cases. Moreover, as far as we know, this study is the first to develop a CG solver that ensures reproducibility across CPUs and GPUs. We note that our implementations are currently unpreconditioned solvers, but that our proposed approach can be used to construct preconditioned solvers.

The remainder of this paper is organized as follows. Section 2 introduces related work. Section 3 describes our methodology based on the Ozaki scheme. Section 4 presents our implementations on CPUs and GPUs. Section 5 details our numerical experiments and their results. Section 6 contains a discussion of other methods for ensuring reproducibility, and conclusions are drawn in Section 7.

[1]https://github.com/riakymch/exblas
[2]http://www.math.twcu.ac.jp/ogita/post-k/results.html
[3]https://software.intel.com/en-us/mkl
[4]https://developer.nvidia.com/cublas

2 RELATED WORK

First, issues of accuracy and reproducibility have different motivations and natures. However, because both, in this case, originate from the same cause (i.e., rounding errors), we can see solutions and challenges common to both. We note that, while reproducibility itself plays no part in the accuracy issue, improving accuracy may contribute to reducing the magnitude of the reproducibility issue. Here, we introduce several examples in BLAS and linear algebra computations.

If only reproducibility (on working precision operations) is needed, a sufficient brute force approach is to fix the order of the computation. Although this approach can be costly on parallel computing, some vendors do go this route. For instance, Intel's Conditional Numerical Reproducible mode [23] provides reproducibility on Intel MKL. NVIDIA cuBLAS also ensure an identical result if the computation is performed on the same number of cores, except for some routines using atomic operations; but it offers alternatives without atomic. However, their reproducibility is ensured only within each library (and under several restrictions) and thus neither are appropriate solutions for the reproducibility between CPUs and GPUs.

A solution that provides full reproducibility in any environment is to perform the computation with the correctly rounded operation. This can additionally contribute to enhancing the accuracy of computation. This approach has been implemented in ExBLAS [3], RARE-BLAS [2], and OzBLAS. The OzBLAS is based on the Ozaki scheme, which is the error-free transformation for dot product/matrix multiplication; this study utilizes the same scheme. The scheme enables one not only to return the correctly rounded result, but also to adjust the accuracy with a certain granularity. Reproducibility is ensured even at tunable accuracy. Moreover, unlike the other approaches, it has a great advantage in that it can be built upon standard BLAS implementations: good performance can be expected with low development cost. ReproBLAS[5] [5] delivers reproducibility without correctly rounded operations. Their approach – originating from the works of Rump, Ogita, and Oishi [19, 20, 22] – cuts (rounds) some lower bits that may cause rounding errors and computes them using multiple bins to compensate for the accuracy.

The above ExBLAS approach has been extended to CG methods [8, 9]. They implemented the CG solver with the Jacobi preconditioner on distributed environments using the pure MPI as well as MPI + OpenMP tasks. To our knowledge, this is the only work to address the reproducibility of computed solutions of CG methods. The other implementations do not, as of now, provide sparse operations. Although the ExBLAS-based CG method has not added support for GPUs yet, this study compares it with our proposed method on CPUs.

There are many studies and software developed for improving accuracy (not for ensuring reproducibility) of CG solvers. Although most of these are mainly intended for improving accuracy, they can also be used to reduce the reproducibility issue. In particular, the aforementioned ExBLAS-based CG methods demonstrate examples that can achieve reproducibility using only an accurate computation method (with only floating-point expansions and the FMA instruction). Other examples of accurate linear algebra computations

[5]https://bebop.cs.berkeley.edu/reproblas/

Algorithm 1 The inner product: $r = \boldsymbol{x}^T\boldsymbol{y}$ ($\boldsymbol{x}, \boldsymbol{y} \in \mathbb{F}^n$) with the Ozaki scheme.

1: **function** (r = `Ozaki_DOT`$(n, \boldsymbol{x}, \boldsymbol{y})$)
2: $\quad \boldsymbol{x}_{\text{split}}[1 : s_x]$ = `Split`$(\boldsymbol{x}, n)$ // Algorithm 2
3: $\quad \boldsymbol{y}_{\text{split}}[1 : s_y]$ = `Split`$(\boldsymbol{y}, n)$ // Algorithm 2
4: $\quad r = 0$
5: $\quad$ **for** $q = 1 : s_y$ **do**
6: $\quad\quad$ **for** $p = 1 : s_x$ **do**
7: $\quad\quad\quad r = r +$ `fl`$((\boldsymbol{x}_{\text{split}}[p])^T \boldsymbol{y}_{\text{split}}[q])$ // DOT
8: $\quad\quad$ **end for**
9: $\quad$ **end for**
10: **end function**

Algorithm 2 Splitting of vector $\boldsymbol{x} \in \mathbb{F}^n$ in the Ozaki scheme, where $\mathbf{u}$ denotes the unit round-off of IEEE 754 ($\mathbf{u} = 2^{-53}$ for FP64). Lines 9 and 10 are computations of $\boldsymbol{x}_i$ and $\boldsymbol{x}_{\text{split}}[j]_i$ for $1 \le i \le n$.

1: **function** ($\boldsymbol{x}_{\text{split}}[1 : s_x]$ = `Split`$(\boldsymbol{x}, n)$)
2: $\quad \rho =$ `ceil`$((\texttt{log2}(\mathbf{u}^{-1}) + \texttt{log2}(n))/2)$
3: $\quad \mu = \max_{1 \le i \le n}(|\boldsymbol{x}_i|)$
4: $\quad j = 0$
5: $\quad$ **while** $\mu \neq 0$ **do**
6: $\quad\quad j = j + 1$
7: $\quad\quad \tau =$ `ceil`$(\texttt{log2}(\mu))$
8: $\quad\quad \sigma = 2^{(\rho+\tau)}$
9: $\quad\quad \boldsymbol{x}_{\text{split}}[j]_i =$ `fl`$((\boldsymbol{x}_i + \sigma) - \sigma)$
10: $\quad\quad \boldsymbol{x}_i =$ `fl`$(\boldsymbol{x}_i - \boldsymbol{x}_{\text{split}}[j]_i)$
11: $\quad\quad \mu = \max_{1 \le i \le n}(|\boldsymbol{x}_i|)$
12: $\quad$ **end while**
13: $\quad s_x = j$
14: **end function**

include the following. MPLAPACK [16] provides high-precision BLAS and Linear Algebra PACKage (LAPACK) routines. The high-precision operation is performed using existing high-precision arithmetic libraries such as the GNU Multiple Precision Floating-Point Reliable Library (MPFR)[6] [6] and QD[7] [7]. XBLAS[8] [12] provides computations with two-fold precision against the data precision. In addition, some studies have implemented CG solvers using IEEE 754 high-precision or an alternative arithmetic format – for example, NAS Parallel Benchmark (including CG) with IEEE 754 binary128 and Posit [1], as well as quadruple-precision CG [15] on GPUs using the double-double arithmetic.

3 METHODOLOGY

3.1 Ozaki scheme

The Ozaki scheme is the error-free transformation of dot product/matrix multiplication. This subsection presents a brief overview of the scheme. For further details of the Ozaki scheme, see the original paper [17].

Here, we explain the case of a dot product, but this scheme can be naturally extended to any dot product-based operations such as matrix-vector multiplication and matrix-matrix multiplication. Algorithm 1 shows the entire procedure of the Ozaki scheme for the dot product of two vectors $\boldsymbol{x} \in \mathbb{F}^n$ and $\boldsymbol{y} \in \mathbb{F}^n$, where $\mathbb{F}$ is the set of floating-point numbers (in this study, FP64). Briefly, this method consists of the following three steps:

(1) Element-wise splitting of the input vectors into several split vectors
(2) Computation of the all-to-all products of those split vectors
(3) Element-wise summation (reduction) of the above all-to-all-product results

This method can be understood as an extension of high-precision arithmetic with multiple components (e.g., double-double arithmetic [7]) into the vector level. Specifically, first, the input vectors are split element-wise into the summation of several vectors using Algorithm 2 as

$$\boldsymbol{x} = \sum_{p=1}^{s_x} \boldsymbol{x}_{\text{split}}{}^{(p)}, \quad \boldsymbol{x}_{\text{split}}{}^{(p)} \in \mathbb{F}^n \tag{1}$$

$$\boldsymbol{y} = \sum_{q=1}^{s_y} \boldsymbol{y}_{\text{split}}{}^{(q)}, \quad \boldsymbol{y}_{\text{split}}{}^{(q)} \in \mathbb{F}^n \tag{2}$$

We note that the number of split vectors (s_x and s_y) to achieve the correctly rounded result depends on the length of the input vectors and the range of the absolute values in the input vectors. Then, it computes the summation of the all-to-all inner products of the split vectors as

$$\boldsymbol{x}^T\boldsymbol{y} = \sum_{p=1}^{s_x}\sum_{q=1}^{s_y} \left(\boldsymbol{x}_{\text{split}}{}^{(p)}\right)^T \boldsymbol{y}_{\text{split}}{}^{(q)} \tag{3}$$

Let $\texttt{fl}(\cdot)$ denote a computation performed with floating-point arithmetic. Algorithm 2 performs the splitting to meet the following two properties for two vectors:

(1) If $\boldsymbol{x}_{\text{split}}{}^{(p)}{}_i$ and $\boldsymbol{y}_{\text{split}}{}^{(q)}{}_j$ are non-zero elements, $\left|\boldsymbol{x}_{\text{split}}{}^{(p)}{}_i\right| \ge \left|\boldsymbol{x}_{\text{split}}{}^{(p+1)}{}_i\right|$ and $\left|\boldsymbol{y}_{\text{split}}{}^{(q)}{}_j\right| \ge \left|\boldsymbol{y}_{\text{split}}{}^{(q+1)}{}_j\right|$.
(2) $\left(\boldsymbol{x}_{\text{split}}{}^{(p)}\right)^T \boldsymbol{y}_{\text{split}}{}^{(q)} = \texttt{fl}\left(\left(\boldsymbol{x}_{\text{split}}{}^{(p)}\right)^T \boldsymbol{y}_{\text{split}}{}^{(q)}\right)$, $1 \le p \le s_x, 1 \le q \le s_y$.

The former implies that the accuracy of the final result can be controlled by omitting some lower split vectors. The key point of the latter is that the inner products of the split vectors can be computed with the standard floating-point arithmetic: for FP64 data, the DDOT routine provided in BLAS such as MKL and cuBLAS can be used[9]. Since $\texttt{fl}\left(\left(\boldsymbol{x}_{\text{split}}{}^{(p)}\right)^T \boldsymbol{y}_{\text{split}}{}^{(q)}\right)$ has no round-off error, that is, it is error-free and reproducible, even if it is computed with a non-reproducible operation.

Subsequently, the accurate and reproducible result is obtained by the summation of the all-to-all inner products of the split vectors. In this study, we compute the summation by a correctly rounded method, NearSum [21], to observe the result obtained by completely

[6] https://www.mpfr.org
[7] https://www.davidhbailey.com/dhbsoftware/
[8] https://www.netlib.org/xblas/

[9] The computation can be performed using a dense matrix-multiplication as we mention later in Section 4. However, it must be implemented on the basis of the standard floating-point inner product. The use of the divide-and-conquer approach, such as Strassen's algorithm is not suitable.

Algorithm 3 CG method solving $\boldsymbol{Ax} = \boldsymbol{b}$ (Note: here, subscript 'i' means the number of iterations, unlike in Algorithm 2).

1: $\boldsymbol{p}_0 = \boldsymbol{r}_0 = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$ // SpMV
2: $\rho_0 = {\boldsymbol{r}_0}^T \boldsymbol{r}_0$ // DOT
3: $i = 0$
4: **while** 1 **do**
5: $\quad \boldsymbol{q}_i = \boldsymbol{A}\boldsymbol{p}_i$ // SpMV
6: $\quad \alpha_i = \rho_i / {\boldsymbol{p}_i}^T \boldsymbol{q}_i$ // DOT
7: $\quad \boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \alpha_i \boldsymbol{p}_i$ // AXPY
8: $\quad \boldsymbol{r}_{i+1} = \boldsymbol{r}_i - \alpha_i \boldsymbol{q}_i$ // AXPY
9: $\quad$ **if** $||\boldsymbol{r}_{i+1}||/||\boldsymbol{b}|| < \epsilon$ **then** // NRM2
10: $\quad\quad$ **break**
11: $\quad$ **end if**
12: $\quad \rho_{i+1} = {\boldsymbol{r}_{i+1}}^T \boldsymbol{r}_{i+1}$ // DOT
13: $\quad \beta_i = \rho_{i+1} / \rho_i$
14: $\quad \rho_i = \rho_{i+1}$
15: $\quad \boldsymbol{p}_{i+1} = \boldsymbol{r}_{i+1} + \beta_i \boldsymbol{p}_i$ // SCAL & AXPY
16: $\quad i = i + 1$
17: **end while**

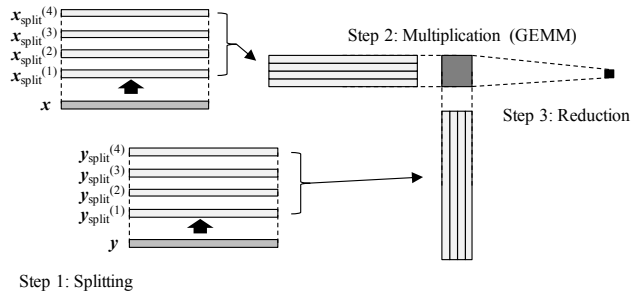

Figure 1: dot product with Ozaki scheme (when the number of split vectors is 4).

eliminating the rounding-error introduced in inner products. Although the summation can also be computed using working precision floating-point operations, the following points must care:

- The summation must be computed using a reproducible method. Since the summation is performed element-wise, fixing the computational order is neither difficult nor costly.
- `log2` in Algorithm 2 must be computed by a reproducible method on different platforms, because the accuracy is not standardized in IEEE and may differ on different platforms (e.g., the accuracy is different between x86 and NVIDIA GPUs). However, this is not a concern when using a correctly rounded summation.

3.2 Installation of reproducibility to the CG method

The CG method solves $\boldsymbol{Ax} = \boldsymbol{b}$ where $\boldsymbol{A}$ is a symmetric positive definite matrix. Algorithm 3 shows the typical algorithm and corresponding BLAS routines for each linear algebra computation. Among them, the factor that may disturb the reproducibility of computed solutions on many core processors is the operations that consist of the inner product operations, namely sparse matrix-vector multiplication (SpMV), dot product (DOT), and 2-norm (NRM2). In this study, these operations are computed using the Ozaki scheme. SpMV and DOT achieve the correctly rounded results. The NRM2 is implemented using DOT as $r = \sqrt{\texttt{DOT}(\boldsymbol{x}, \boldsymbol{x})}$, and the square root is performed on the standard FP64 operation.

In AXPY, we need to ensure consistency in whether or not the FMA operation is used. In this study, we use the AXPY implementation that uses the FMA operation. In addition, to prevent fast and less accurate computations for mathematical functions (e.g., `-fp-model fast` on ICC), we disable any less accurate options.

4 IMPLEMENTATION

We implement the following two versions for both CPUs and GPUs:

- **FP64**: the standard implementation on FP64
- **FP64Oz-CR**: the accurate and reproducible FP64 implementation using the Ozaki scheme

Below describes the details of the implementations.

4.1 FP64

All computations are implemented using the standard FP64 arithmetic and the standard FP64 BLAS routines. Each linear algebra operation is performed through the corresponding BLAS routine shown in Algorithm 3 by using Intel MKL on CPUs and NVIDIA cuSparse (for SpMV) and cuBLAS (for the others) on GPUs. On the SpMV routines, the symmetrical structure of matrices is not considered (i.e., symmetric matrices are given to the computation after being expanded into general matrices). The coefficient sparse matrix is stored using the common Compressed Sparse Row (CSR) format. We note here that the choice of format does not affect the performance discussion in this study, as this study aims to examine the relative difference. In the GPU implementations, the BLAS routines are performed on GPUs, whereas the scalar value computations are performed on CPUs.

4.2 FP64Oz-CR

The Ozaki scheme is installed into DOT, NRM2, and CSRMV in the aforementioned FP64 implementation. In the Ozaki scheme, the internal computations are performed using MKL on CPUs and cuSparse and cuBLAS on GPUs. The splitting and summation portions are parallelized using OpenMP on CPUs and CUDA on GPUs. In Algorithm 2, to perform lines 9 and 10 correctly, the order of expression evaluation must be honored by using the compiler option "`-fprotect-parens`" on CPUs and the intrinsic for arithmetic on GPUs. 2^τ in lines 7 and 8 is computed using NextPowTwo [18]. AXPY is implemented using FMA.

In addition, we employ several techniques for speedup.

(1) For SpMV, as the coefficient matrix is not changed during the iterations, the splitting of the matrix is needed only once before the iteration starts. This contributes to a nontrivial performance increase, because matrix splitting becomes the major cost in the Ozaki scheme on memory-bound operations.

Table 1: Test matrices (the size is $n \times n$ with nnz non-zero elements, sorted in ascending order by nnz/n).

#	Matrix	n	nnz	nnz/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

(2) The inner products of the split vectors can be performed using a dense matrix multiplication (GEMM) by combining multiple split vectors into a single matrix, as shown in Figure 1. This strategy contributes to the speedup of memory-bound operations in the Ozaki scheme by reducing memory access. The same concept can be applied to SpMV: the computation can be performed using a sparse matrix - dense matrix multiplication routine (SpMM), which is available on MKL as mkl_dcsrmm and on cuSparse as cusparseDcsrmm. On CPUs, however, we do not use this technique because performance is degraded with mkl_dcsrmm[10].

(3) In SpMV, we use asymmetric splitting [18]. σ at line 8 in Algorithm 2 determines how many bits are stored in each element of the split matrices/vectors; smaller σ increases the number of bits that can be held. σ is determined not to cause an overflow in the computation of the product of the split matrices and vectors but is chosen to be as small as possible in the powers of 2. We can bias the ρ at line 2 to reduce the number of split data on either the matrix or vector by increasing ρ on one side and decreasing ρ on another side by the same amount. If SpMM is used in the computation, the reduction of the number of split matrices increases the chance of a speedup in general. Our GPU implementation decreases the ρ on the matrix side by the minimum amount that decreases the number of split matrices. On the other hand, the CPU implementation, which does not use SpMM in its computation, increases the ρ on the matrix side by the maximum amount that does not change the number of split matrices. This strategy increases the chance of reduction in the number of split vectors. The optimal ρ is determined by trial and error by performing the matrix splitting with the ρ reduced step by step. As this determination is performed only once before starting the iterations of the CG method, the cost is not high.

5 EVALUATION

We conducted evaluations using the following platforms:

- **CPU1**: Intel Xeon Gold 6126 (Skylake, 2.60–3.70 GHz, 12 cores) × 2 sockets, DDR4-2666 192 GB (255.9 GB/s), MKL 19.0.5, ICC 19.0.5.281, 1 thread/core was assigned, "`numactl --localalloc`" was used for the execution, on the Cygnus supercomputer in University of Tsukuba.
- **CPU2**: Intel Xeon Phi 7250 (Knights Landing, 1.40–1.60 GHz, 68 cores), MCDRAM 16GB (490 GB/s) + DDR4-2400 115.2 GB/s, MKL 19.0.5, ICC 19.0.5.281, 1 thread/core was assigned (64 cores were used for the computation[11]), memory-mode: flat, clustering-mode: quadrant, KMP_AFFINITY=scatter, "`numactl --preferred 1`" was used for the execution (MCDRAM preferred), on the Oakforest-PACS system operated by Joint Center for Advanced High Performance Computing (JCAHPC).
- **GPU1**: NVIDIA Tesla V100-PCIE-32GB (Volta, 1.370 GHz, 80 SMs, 2560 FP64 cores), HBM2 32GB 898.0 GB/s, CUDA 10.2, nvcc V10.2.89, on the Cygnus supercomputer in University of Tsukuba.
- **GPU2**: NVIDIA Tesla P100-PCIE-16GB (Pascal, 1.189–1.328 GHz, 56 SMs, 1792 FP64 cores), HBM2 16GB 720 GB/s, CUDA 10.2, nvcc V10.2.89.

The programs were compiled using the following options: for CPUs, "`-O3 -fma -fp-model source -fprotect-parens -qopenmp`" with "`-xCORE-AVX2 -mtune=skylake-avx512`" on CPU1 and "`-xMIC-AVX512`" on CPU2; for GPUs, "`-O3 -gencode arch=compute_60,code=sm_XX`" (XX=70 for Tesla V100 and XX=60 for Tesla P100). Any fast and less accurate computation options were disabled.

We collected eight symmetric positive definite matrices from the SuiteSparse Matrix Collection [4], as shown in Table 1. These matrices were chosen to be large enough to be computed on GPUs and used in different applications. The right-hand side vector $\boldsymbol{b}$ and the initial solution $\boldsymbol{x}_0$ were set as $\boldsymbol{b} = \boldsymbol{x}_0 = (1, 1, ..., 1)^T$. The iteration was terminated when $||\boldsymbol{r}_i||/||\boldsymbol{b}|| \leq 10^{-16}$. Hereafter, the residual plots showed $||\boldsymbol{r}_i||/||\boldsymbol{b}||$.

5.1 Reproducibility, convergence, and accuracy

Table 2 shows both the relative true residual ($||\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_i||/||\boldsymbol{b}||$) when $||\boldsymbol{r}_i||/||\boldsymbol{b}|| \leq 10^{-16}$ and the number of iterations across four platforms. In most cases, the results (both the solution and the number of iterations) of FP64 are non-identical among different platforms. However, the results of FP64Oz-CR across the four platforms were identical; that is, reproducibility was ensured by the Ozaki scheme. In addition, some cases converged with fewer iterations by FP64Oz-CR with accurate computation when compared with FP64 implementations.

[10] This routine has already been deprecated in the latest MKL, and the use of Inspector-executor Sparse BLAS interface is recommended instead. However, our implementation does not support it yet.

[11] The CPU has 68 cores but only 64 cores from the core number 2 were used to avoid OS jitter.

Table 2: (a) Relative true residual and (b) number of iterations on different platforms. Note: in all cases, true residual was computed on the correctly rounded operations using the Ozaki scheme. FP64Oz-CR got identical result on all platforms.

(a) Relative true residual ($||\boldsymbol{b} - \boldsymbol{Ax}_i||/||\boldsymbol{b}||$)

#	Matrix	FP64				FP64Oz
		CPU1	CPU2	GPU1	GPU2	-CR
1	tmt_sym	3.30E-07	3.29E-07	3.29E-07	3.30E-07	3.29E-07
2	gridgena	1.11E-10	1.10E-10	1.09E-10	1.09E-10	1.08E-10
3	cfd1	1.48E-10	1.48E-10	1.50E-10	1.50E-10	1.48E-10
4	cbuckle	8.97E-12	9.01E-12	8.85E-12	8.85E-12	9.08E-12
5	BenElechi1	7.66E-07	8.37E-07	8.50E-07	1.04E-06	6.68E-07
6	gyro_k	4.00E-07	3.77E-07	4.70E-07	4.70E-07	4.30E-07
7	pdb1HYS	4.27E-04	4.35E-04	4.36E-04	4.36E-04	3.82E-04
8	nd24k	2.09E-08	2.10E-08	2.09E-08	2.09E-08	2.10E-08

(b) Number of iterations to $||\boldsymbol{r}_i||/||\boldsymbol{b}|| \leq 10^{-16}$

#	FP64				FP64Oz
	CPU1	CPU2	GPU1	GPU2	-CR
1	7859	7831	7828	7825	7793
2	2400	2413	2368	2368	2393
3	3279	3277	3278	3278	3279
4	23834	23856	23515	23515	23724
5	73327	72701	73515	71302	65161
6	60387	60247	59341	59341	46641
7	11378	11788	11775	11775	8214
8	15461	15445	15438	15438	12837

Table 3: Number of split matrices required to achieve correct-rounding, total execution time until convergence, and execution time overhead (FP64Oz-CR/FP64) on CPUs.

#	Matrix	# of split mats	CPU1			CPU2		
			FP64 (secs)	FP64Oz-CR (secs)	Overhead (times)	FP64 (secs)	FP64Oz-CR (secs)	Overhead (times)
1	tmt_sym	4	6.14E+00	1.73E+02	28.2	6.12E+00	2.24E+02	36.6
2	gridgena	3	4.72E-01	5.48E+00	11.6	9.98E-01	1.08E+01	10.8
3	cfd1	4	7.14E-01	1.08E+01	15.1	1.65E+00	1.93E+01	11.7
4	cbuckle	7	2.92E+00	4.68E+01	16.0	9.24E+00	1.06E+02	11.5
5	BenElechi1	4	8.12E+01	1.36E+03	16.7	6.05E+01	1.11E+03	18.3
6	gyro_k	7	1.02E+01	1.30E+02	12.7	2.44E+01	2.46E+02	10.1
7	pdb1HYS	4	4.04E+00	5.28E+01	13.1	7.63E+00	6.18E+01	8.1
8	nd24k	4	3.30E+01	4.54E+02	13.8	2.49E+01	3.02E+02	12.1

Table 4: Number of split matrices required to achieve correct-rounding, total execution time until convergence, and execution time overhead (FP64Oz-CR/FP64) on GPUs.

#	Matrix	# of split mats	GPU1			GPU2		
			FP64 (secs)	FP64Oz-CR (secs)	Overhead (times)	FP64 (secs)	FP64Oz-CR (secs)	Overhead (times)
1	tmt_sym	3	2.49E+00	4.26E+01	17.1	3.26E+00	7.09E+01	21.7
2	gridgena	2	2.42E-01	3.05E+00	12.6	2.67E-01	3.28E+00	12.3
3	cfd1	3	4.52E-01	5.58E+00	12.3	4.89E-01	7.28E+00	14.9
4	cbuckle	6	3.05E+00	2.96E+01	9.7	3.62E+00	3.73E+01	10.3
5	BenElechi1	3	2.71E+01	3.47E+02	12.8	3.53E+01	5.96E+02	16.9
6	gyro_k	6	7.36E+00	5.61E+01	7.6	8.35E+00	7.12E+01	8.5
7	pdb1HYS	3	2.00E+00	1.20E+01	6.0	2.26E+00	1.57E+01	6.9
8	nd24k	3	8.71E+00	4.81E+01	5.5	1.12E+01	7.12E+01	6.4

Figure 2 shows the residual plots for every 10 iterations on four platforms. Solid lines show the relative residual $||\boldsymbol{r}_i||/||\boldsymbol{b}||$ and dotted lines show the relative true residual $||\boldsymbol{b} - \boldsymbol{Ax}_i||/||\boldsymbol{b}||$[12]. Significant differences can be observed in the convergence plots among the five cases, but in all cases, the solution converges to a similar value on the same order (however, most of them are non-identical, as shown in Table 2). In terms of the residual $\boldsymbol{r}_i$, FP64Oz-CR often converged with fewer iterations than FP64, but in those cases, the stopping criterion of 10^{-16} was too small.

[12]Here, unlike Table 2, the true residual was computed using the standard FP64 operation, but it does not cause visible difference in the plot.

5.2 Performance (overhead)

Before presenting the experimental results, we discuss the expected performance. When SpMV with the Ozaki scheme is computed using SpMM (i.e., our GPU implementation), the execution time overhead of FP64Oz-CR against FP64 per iteration can be roughly estimated. Ideally, if the matrix is sufficiently dense, the overhead becomes close to d times, where d is the number of split matrices,

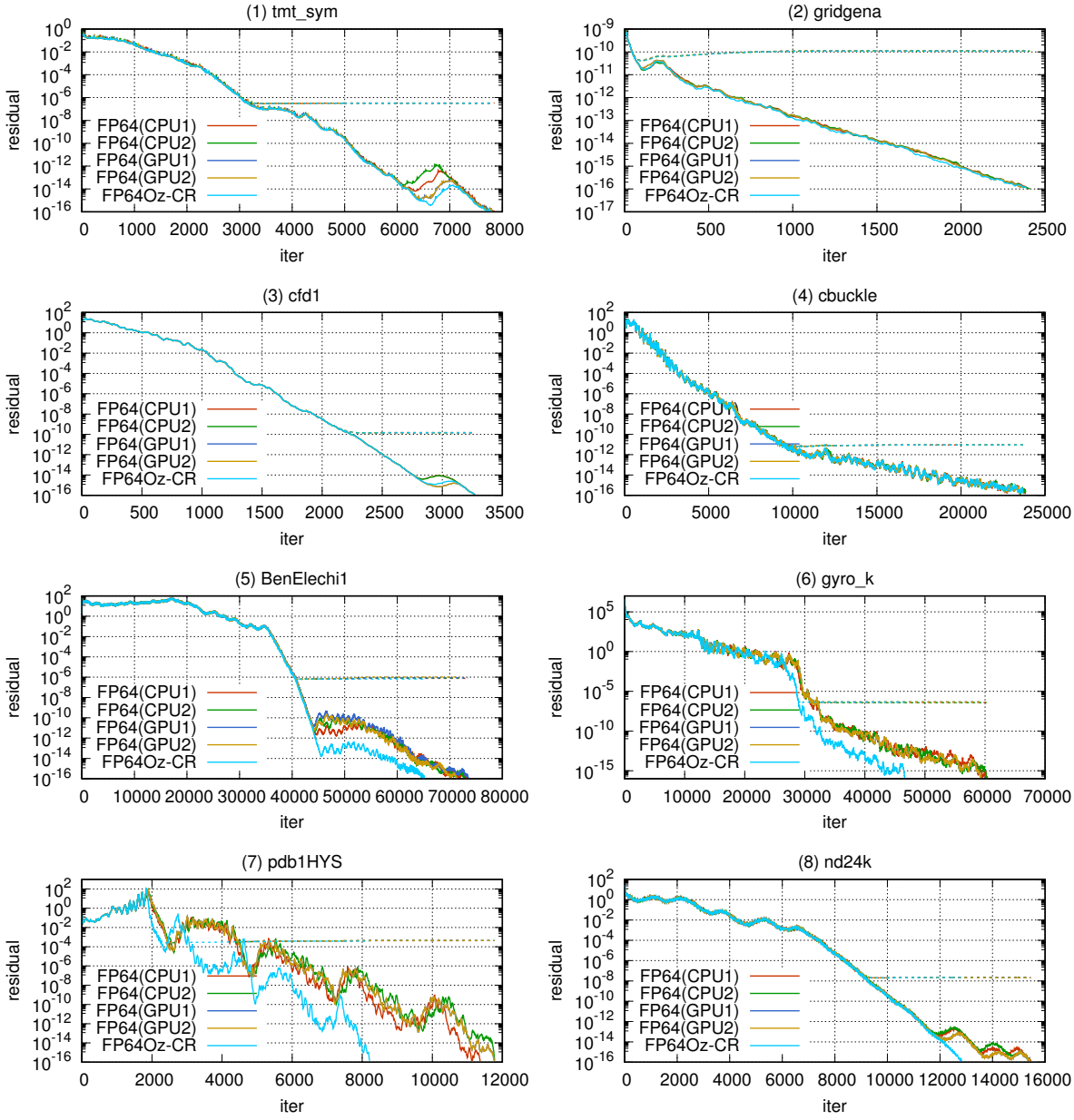

Figure 2: Convergence plots (at every 10 iterations). Results of FP64Oz-CR on four platforms are shown with one line as they are identical. Solid lines show relative residual $||r_i||/||b||$ and dotted lines show relative true residual $||b - Ax_i||/||b||$.

which can be obtained by performing the splitting of the coefficient matrix once. When SpMV is computed using SpMM, it requires $4d$ times execution time overhead against the standard floating-point operation in terms of memory access to the matrix (we assume that the cost to the vector can be ignored as it is small enough when compared with the matrix). The $3/4$ of the $4d$ times overhead arises in the splitting process (the accesses to vector $\boldsymbol{x}$ at lines 9 and 10 in Algorithm 3); however, it is eliminated on CG methods since the splitting is performed only once before iteration. In CG methods, the SpMV cost is usually dominant, as matrix-vector multiplication is an $O(n^2)$ operation, whereas the others are $O(n)$. Therefore, only the d times overhead appears. However, if the matrix is highly sparse, and the execution efficiency becomes low owing to the frequent random memory access, the cost of other operations than SpMV becomes nonnegligible, and the overhead becomes unpredictable. As DOT requires $4d$ times overhead (assuming cache hits all split vectors), the cost of DOT (and NRM2) may become dominant instead of SpMV. Moreover, the value of d is unpredictable in CG methods

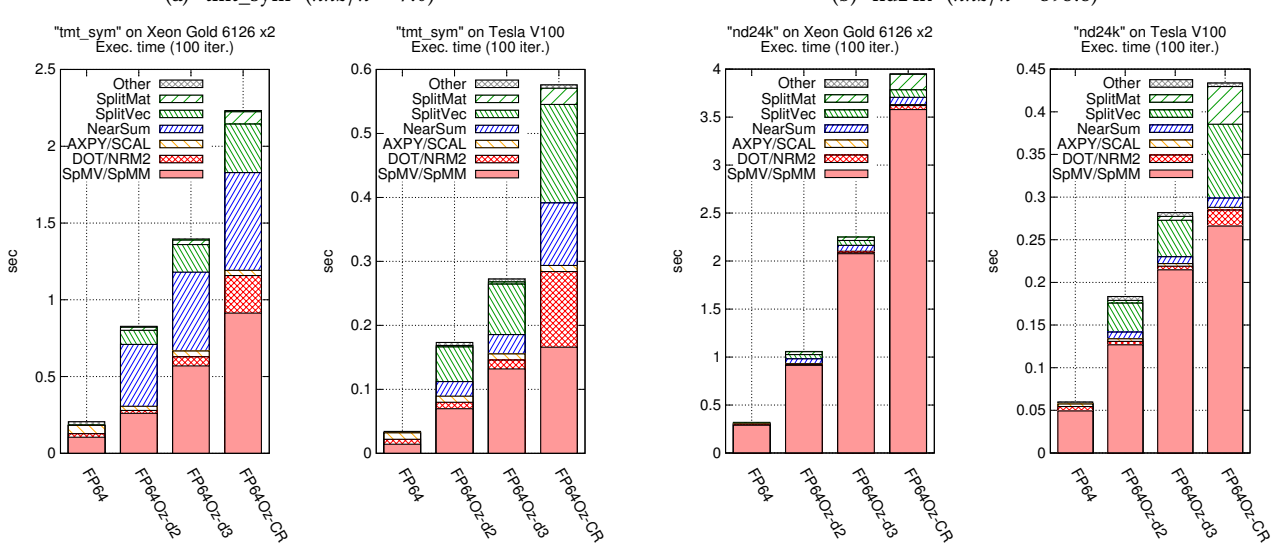

Figure 3: Execution time breakdown of FP64, FP64Oz-dn and FP64Oz-CR for 100 iterations on CPU1 and GPU1. FP64Oz-dn shows result of FP64Oz-CR executed with specified d (the number of split matrices/vectors).

because the vectors are updated during iteration. In addition, as d increases, the chance of cache-missing increases.

Tables 3 and 4 show the number of split matrices required to achieve correct-rounding (d), the execution time until convergence, and the execution time overhead (times) of FP64Oz-CR against FP64 on CPUs and GPUs, respectively. Note that the required number of splits becomes minus-one on GPUs when compared with that on CPUs as we use the asymmetric-splitting technique described in Section 4. The FP64 implementations, which are the baselines for the comparison, can be seen as the ideal performance as they are constructed on vendor-implemented routines. As the CPU implementation does not use SpMM, we consider that the GPU implementation shows more desirable results. The number of split matrices corresponds to the expected minimum overhead, as explained before. When compared with the number of split matrices, approximately 1.3 – 7.2 times additional overhead was observed on GPUs, and we can see that the smaller nnz/n it has, the greater the overhead tends to take, following the previous discussion.

Figure 3 shows the breakdown of the execution time of 100 iterations on the tmt_sym and nd24k matrices on CPU1 and GPU1. Both matrices have the smallest and biggest sparsities, respectively. tmt_sym shows a high cost for level-1 BLAS operations, while nd24k shows a high cost for SpMV (SpMM). FP64Oz-dn shows the result executed with a specified d (the number of split matrices/vectors). When d is specified, the asymmetric-splitting technique is not used; only with FP64Oz-CR does the matrix splitting cost (SplitMat) include the tuning cost for the asymmetric splitting. However, it is not so large within 100 iterations.

5.3 Comparison with ExBLAS-based CG

Iakymchuk et al. have already proposed accurate and reproducible CG solvers [8, 9] based on the ExBLAS approach [10]. These CG solvers are parallelized with the flat MPI as well as MPI and OpenMP tasks but support only CPUs. We evaluate their performance and compare it against that of the proposed method. As with the proposed method, the ExBLAS-based implementation ensures correct rounding in all dot product operations in CG. The ExBLAS approach efficiently combines the Kulisch long accumulator [11] and floating-point expansions (FPEs). While the long accumulator is robust and designed for severe (ill-conditioned) cases, keeping every bit of information until the final rounding, FPEs are unevaluated sums (arrays of FP64) to target a limited range of numbers (e.g., nonsevere dynamic ranges and/or condition numbers). Hence, ExBLAS aims to use FPEs as much as possible owing to their speed and only occasionally long accumulators (e.g., when the accuracy of FPEs is insufficient, or at the final rounding to FP64). One clear advantage of the ExBLAS-based implementation against our proposed approach is its low memory consumption: it uses only 2097 bits for a long accumulator and 192-512 bits for an FPE per MPI process, while the proposed method consumes a large amount of memory for storing split matrices (i.e., in proportion to the number of split data d).

The ExBLAS-based implementations have two versions: the MPI-OpenMP hybrid parallel [8] and the flat MPI version [9]. However, the flat MPI version was faster than the hybrid version on both CPU1 and CPU2. Therefore, the following evaluation was conducted using only the flat MPI version. We conducted these experiments with the same conditions as the other evaluations except for the following points: The code[13] was compiled using GCC 8.3.1 on CPU1 and GCC 7.5.0 on CPU2 with `-mavx -fabi-version=0 -fopenmp`. On both CPUs, we executed the code by mapping one MPI process per core. The ExBLAS-based implementation supports preconditioning, but it was disabled in this evaluation.

[13] https://github.com/riakymch/ReproCG

Table 5: The results of the ExBLAS-based implementation (FP64Ex-CR): number of iterations, execution time, and overhead against FP64 shown in Table 3.

#	Matrix	Num. Iter.	CPU1		CPU2	
			Time (secs)	Overhead (times)	Time (secs)	Overhead (times)
1	tmt_sym	7812	3.37E+02	54.9	5.48E+02	89.5
2	gridgena	2467	7.97E+00	16.9	1.37E+01	13.8
3	cfd1	3278	1.59E+01	22.3	2.81E+01	17.0
4	cbuckle	23828	3.24E+01	11.1	5.95E+01	6.4
5	BenElechi1	73838	1.28E+03	15.8	2.49E+03	41.2
6	gyro_k	60103	9.74E+01	9.5	1.82E+02	7.5
7	pdb1HYS	11839	3.77E+01	9.3	9.61E+01	12.6
8	nd24k	15415	1.66E+02	5.0	5.49E+02	22.1

Table 5 shows the results obtained by the ExBLAS-based implementation (FP64Ex-CR) on CPU1 and CPU2. We note that the results of FP64Oz-CR and FP64Ex-CR may differ because the implementations of the CG algorithm itself are different and the strategies for reproducibility are clearly not the same. Since FP64Oz-CR and FP64Ex-CR adopt different strategies for accurate and reproducible computations, their overhead compared to the baseline FP64-based implementation depends on both the matrix at hand and the processors used. The FP64Ex-CR achieved better (lower) overhead on five out of eight matrices on CPU1. On the other hand, the proposed method FP64Oz-CR performs better on six out of eight matrices on CPU2. Even though FP64Ex-CR does not support GPUs, the best performance of FP64Oz-CR is on GPUs, where the potential of FP64Oz-CR flourishes with the use of SpMM and the reduction of the number of split matrices.

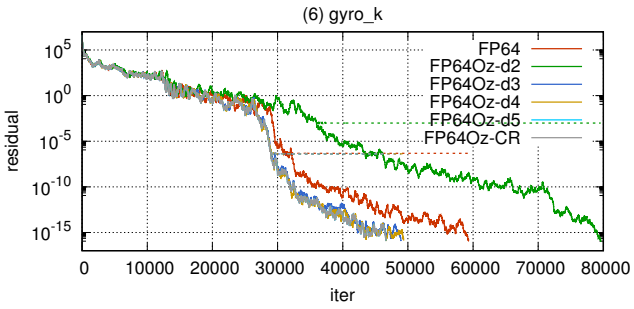

Figure 4: Convergence plot of "gyro_k" (at every 10 iterations) on CPU1. Solid lines show the relative residual $||\boldsymbol{r}_i||/||\boldsymbol{b}||$ and dotted lines show the relative true residual $||\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_i||/||\boldsymbol{b}||$.

6 REPRODUCIBILITY WITHOUT CORRECTLY ROUNDED OPERATIONS, AND ACCURATE COMPUTATION FOR REPRODUCIBILITY

In this study, we installed reproducibility via correctly rounded operations, but the Ozaki scheme can adjust the accuracy to a given granularity and still ensure reproducibility (as we described in Section 3.1, the summation and log2 must care for ensuring reproducibility). The accuracy can be adjusted by the number of split matrices/vectors, and reducing it improves performance, as shown in Figure 3.

Figure 4 demonstrates, as an example, the convergence of "gyro_k" with FP64, FP64Oz-CR, and FP64Oz-d with a specified number of split data d. This matrix requires 6 split matrices ($d = 6$) to achieve the correctly rounded operation, but $d = 3$ is sufficient to achieve higher accuracy than FP64. However, there is currently no lightweight way to determine the optimal number in advance (i.e., the number needed to achieve the FP64 equivalent accuracy at least or to get the solution at the shortest time). There is a risk that the iteration will be unconverged or require a lot of iterations if an inadequate choice is made. To determine the optimal number, we have to try it once. However, this approach can be useful at least when the aim is to reproduce the results obtained on one system, together with the necessary split number, on another system.

On the other hand, reproducibility may also be achieved simply by using accurate (not necessarily reproducible) computation methods. Accurate computations can increase the possibility of reproducibility (i.e., the number of bits that can be reproduced, through reducing rounding errors). However, the level of accuracy needed to ensure a certain level of reproducibility is problem (input) dependent. Iakymchuk et al. [8, 9] demonstrated the achievement of reproducibility with only an accurate computation method (only with FPE, i.e., the ExBLAS scheme without a long accumulator) by focusing on certain problems. It contributes more toward improving performance than the ExBLAS approach does.

7 CONCLUSIONS

This paper presents an accurate and reproducible implementation of the CG method on CPUs and GPUs. The accurate and reproducible operations were introduced into all the inner product-based operations in the CG method through the Ozaki scheme, which performs the correctly rounded operation. We conducted numerical experiments on different platforms, including CPUs and GPUs. While the standard FP64 CG implementations using existing vendor libraries might return non-identical results, our implementations always returned a bit-level identical result. The cost of the Ozaki scheme depends on the problem, but our implementations

achieve performance comparable to an existing work based on the ExBLAS approach in many cases. The proposed approach has an additional advantage in its low development cost, as it relies on vendor-provided BLAS implementations. Moreover, we demonstrated certain cases for which accurate computation through the Ozaki scheme improved the convergence, even though it did not contribute to reducing the total execution time. The source code of our implementations is available together with OzBLAS[14].

As future work, we can adopt the mixed-precision approach into our proposed method. While this study used FP64 for computation, the Ozaki scheme can also be built upon low-precision operations such as FP32 and the mixed-precision operation of FP32 and FP16 on Tensor Cores available on NVIDIA GPUs, as we have shown in [14]. That is, on dot product, for example, instead of using DGEMM for the computation as we did in this study, SGEMM or Tensor Core GEMM (gemmEx) can also be used to compute FP64 input/output. This may contribute to improving the proposed method's performance on hardware with limited FP64 support.

ACKNOWLEDGMENTS

This research was partially supported by the Japan Society for the Promotion of Science (JSPS) KAKENHI Grant #19K20286 and the EU H2020 research, innovation program under the Marie Skłodowska-Curie grant agreement via the Robust project No. 842528. This research used computational resources of the Cygnus supercomputer provided by Multidisciplinary Cooperative Research Program in Center for Computational Sciences, University of Tsukuba, and the Oakforest-PACS system operated by JCAHPC.

REFERENCES

[1] S. W. D. Chien, I. B. Peng, and S. Markidis. 2019. Posit NPB: Assessing the Precision Improvement in HPC Scientific Applications. (to appear).
[2] C. Chohra, P. Langlois, and D. Parello. 2016. Reproducible, Accurately Rounded and Efficient BLAS. In *22nd International European Conference on Parallel and Distributed Computing (Euro-Par 2016)*. 609–620.
[3] Sylvain Collange, David Defour, Stef Graillat, and Roman Iakymchuk. 2015. Numerical Reproducibility for the Parallel Reduction on Multi- and Many-Core Architectures. *Parallel Computing* 49 (2015), 83–97. https://doi.org/10.1016/j.parco.2015.09.001
[4] T. A. Davis and Y. Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Software* 38, 1 (2011), 1:1–1:25.
[5] J. Demmel, P. Ahrens, and H. D. Nguyen. 2016. *Efficient Reproducible Floating Point Summation and BLAS.* Technical Report UCB/EECS-2016-121. EECS Department, University of California, Berkeley.
[6] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. 2007. MPFR: A Multiple-precision Binary Floating-point Library with Correct Rounding. *ACM Trans. Math. Software* 33, 2 (2007), 13:1–13:15.
[7] Y. Hida, X. S. Li, and D. H. Bailey. 2007. *Library for Double-Double and Quad-Double Arithmetic.* Technical Report. NERSC Division, Lawrence Berkeley National Laboratory.
[8] R. Iakymchuk, M. Barreda, S. Graillat, J. I. Aliaga, and E. S. Quintana-Ortí. 2020. Reproducibility of Parallel Preconditioned Conjugate Gradient in Hybrid Programming Environments. *IJHPCA* (2020). Available OnlineFirst 17 June 2020. https://doi.org/10.1177/1094342020932650.
[9] R. Iakymchuk, M. Barreda, M. Wiesenberger, J. I. Aliaga, and E. S. Quintana-Ortí. 2020. Reproducibility strategies for parallel Preconditioned Conjugate Gradient. *J. Comput. Appl. Math.* 371 (2020), 112697. https://doi.org/10.1016/j.cam.2019.112697
[10] R. Iakymchuk, S. Collange, D. Defour, and S. Graillat. 2015. ExBLAS: Reproducible and Accurate BLAS Library. In *Proc. Numerical Reproducibility at Exascale (NRE2015) at SC'15.*
[11] U. W. Kulisch. 2013. *Computer arithmetic and validity* (2nd ed.). de Gruyter Studies in Mathematics, Vol. 33. Walter de Gruyter & Co., Berlin. xxii+434 pages. Theory, implementation, and applications.
[12] X. S. Li, J. W. Demmel, D. H. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, A. Kapur, M. C. Martin, T. Tung, and D. J. Yoo. 2000. Design, Implementation and Testing of Extended and Mixed Precision BLAS. *ACM Trans. Math. Software* 28, 2 (2000), 152–205.
[13] D. Mukunoki, T. Ogita, and K. Ozaki. 2020. Reproducible BLAS Routines with Tunable Accuracy Using Ozaki Scheme for Many-core Architectures. In *Proc. 13th International Conference on Parallel Processing and Applied Mathematics (PPAM2019), Lecture Notes in Computer Science*, Vol. 12043. Springer Berlin Heidelberg, 516–527. https://doi.org/10.1007/978-3-030-43229-4_44
[14] D. Mukunoki, K. Ozaki, T. Ogita, and T. Imamura. 2020. DGEMM using Tensor Cores, and Its Accurate and Reproducible Versions. In *ISC High Performance 2020, Lecture Notes in Computer Science*, Vol. 12151. Springer International Publishing, 230–248. https://doi.org/10.1007/978-3-030-50743-5_12
[15] D. Mukunoki and D. Takahashi. 2014. Using Quadruple Precision Arithmetic to Accelerate Krylov Subspace Methods on GPUs. In *10th International Conference on Parallel Processing and Applied Mathematics (PPAM2013).* 632–642.
[16] M. Nakata. [n.d.]. The MPACK; Multiple precision arithmetic BLAS (MBLAS) and LAPACK (MLAPACK). http://mplapack.sourceforge.net.
[17] 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.
[18] K. Ozaki, T. Ogita, S. Oishi, and S. M. Rump. 2013. Generalization of error-free transformation for matrix multiplication and its application. *Nonlinear Theory and Its Applications, IEICE* 4 (2013), 2–11.
[19] S. M. Rump, T. Ogita, and S. Oishi. 2008. Accurate Floating-Point Summation Part I: Faithful Rounding. *SIAM J. Sci. Comput.* 31, 1 (2008), 189–224. https://doi.org/10.1137/050645671
[20] S. M. Rump, T. Ogita, and S. Oishi. 2008. Accurate floating-point summation part II: Sign, K-fold faithful and rounding to nearest. *SIAM J. Sci. Comput.* 31, 2 (2008), 1269–1302.
[21] S. M. 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.
[22] S. M. Rump, T. Ogita, and S. Oishi. 2010. Fast high precision summation. *Nonlinear Theory and Its Applications, IEICE* 1, 1 (2010), 2–24.
[23] R. Todd. 2012. Introduction to Conditional Numerical Reproducibility (CNR). https://software.intel.com/en-us/articles/introduction-to-the-conditional-numerical-reproducibility-cnr.

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