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

Daichi Mukunoki, Katsuhisa Ozaki, Takeshi Ogita, Roman Iakymchuk

▶ **To cite this version:**

Daichi Mukunoki, Katsuhisa Ozaki, Takeshi Ogita, Roman Iakymchuk. Conjugate Gradient Solvers with High Accuracy and Bit-wise Reproducibility between CPU and GPU using Ozaki Scheme. 2020. hal-02986873

HAL Id: hal-02986873

https://hal.science/hal-02986873v1

Preprint submitted on 3 Nov 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

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), the number of iterations until convergence may increase due to the loss of computation accuracy caused by rounding errors in floating-point computations. Besides, as the order of operations is non-deterministic on parallel computations, the result and the behavior of the convergence may be non-identical in different environments, even for the same input. This paper presents a new approach for the CG method with high accuracy as well as bit-level reproducibility of computed solutions on many-core processors, including both x86 CPUs and NVIDIA GPUs. In our proposed approach, accurate and reproducible operations are installed into all the inner-product based operations such as matrix-vector multiplication and dot-product, which are the main sources that may disturb reproducibility in the CG method. The accurate and reproducible operations are performed using the Ozaki scheme, which is the error-free transformation for dot-product that can ensure the correct-rounding. As this method can be built upon vendor-provided linear algebra libraries such as Intel Math Kernel Library and NVIDIA cuBLAS/ cuSparse, it reduces the development cost. In this paper, showing some examples with the non-identical convergences and computed solutions on different platforms, we demonstrate the applicability and the effectiveness of the proposed approach as well as its performance on both CPUs and GPUs. Besides, we compare against an existing accurate and reproducible CG implementation based on the Exact BLAS (ExBLAS) on CPUs.

CCS CONCEPTS

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference'17, July 2017, Washington, DC, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/xxx

KEYWORDS

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

ACM Reference Format:
Daichi Mukunoki, Katsuhisa Ozaki, Takeshi Ogita, and Roman Iakymchuk. 2020. Conjugate Gradient Solvers with High Accuracy and Bit-wise Reproducibility between CPU and GPU using Ozaki scheme. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 10 pages. https://doi.org/xxx

1 INTRODUCTION

Floating-point computations with finite-precision involve rounding-errors at each operation, and their accumulation may result in the inaccuracy of the computation. At the same time, as floating-point computations are non-associative, hence the result is non-deterministic (i.e., non-reproducible) if the order of the computation is not identical, and it can be different at the rounding-error-level even for the same input. As the scale of the computation grows toward Exascale computing, and the rounding-error accumulation becomes large, those issues may become more serious. Besides, the recent trend in introducing low-precision hardware increases the magnitude of rounding-errors.

The reproducibility issue used to be out of focus until now. However, in addition to the previously mentioned background of increasing rounding error, with the advent of various processors such as GPUs and its heterogeneous use, the importance has become more clear and needed. In fact, recent high-performance computing environments contain many factors to vary the order of computations, which impact the reproducibility of the computational result. For example, those factors include parallel computation with different degrees of parallelism (change of the number of threads, processes, etc.), atomic operation on many-core architectures, the use or non-use of fused multiply-add (FMA) operation, and the introduction of auto-tuning and dynamic load balancing techniques. The reproducibility issue may disturb software debugging and cause problems for scientific activities that rely on the reproducibility of results. Besides, when a code is ported to a new system, and the results are different, it can be a problem from the viewpoint of reliability and

quality-control if it is not possible to distinguish whether it is 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. A particularly sensitive part of the CG methods is the computation of residual. Besides, 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 the 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 on the working-precision is computed with one rounding at the end. At first, we show some examples where the standard FP64 implementation of CG results in non-identical results among different CPUs and GPUs. Then, we demonstrate the applicability and the effectiveness of our implementations, in terms of the accuracy and reproducibility, and the 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 BLAS (ExBLAS)[1] [9, 10], which also performs correctly-rounded operations for inner-products as with our implementations.

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 been adopted for an accurate and reproducible BLAS, OzBLAS[2] [13], and one of its greatest advantages is that it can be built upon standard BLAS implementations such as Intel Math Kernel Library (MKL)[3] and NVIDIA cuBLAS[4]: a good performance can be expected with low development cost. In fact, our proposed method can achieve comparable or better performance than the ExBLAS-based approach in many cases. Moreover, as far as we know, this study is the first to develop a CG solver that ensures reproducibility among 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 remaining 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 presents the numerical experiments. Section 6 presents a discussion on another way for reproducibility. Finally, the conclusion is presented 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

In the first place, accuracy and reproducibility issues are different ones that have different motivations and natures. However, since both are originated from the same cause (rounding-errors), we can see some common solutions and challenges for both. We note that, while reproducibility itself does not take part in the accuracy issue, improving accuracy may contribute to relaxing the magnitude of the reproducibility issue. Here, we introduce several examples in Basic Linear Algebra Subprograms (BLAS) and linear algebra computations.

If only reproducibility (on working-precision operations) is needed, a brute force approach is to fix the order of the computation. Although this approach can be costly on parallel computing, some vendors adopt this approach. For instance, Intel's Conditional Numerical Reproducible (CNR) mode [23] provides reproducibility on Intel MKL. NVIDIA cuBLAS also ensure the identical result if the computation is performed on the same number of cores except some routine implemented with atomic operations; but, it offers an alternative without atomic. However, their reproducibility is ensured only within each library (and under several restrictions), and thus both above cannot be a solution for the reproducibility between CPUs and GPUs.

A solution to provide full reproducibility on any environment is to perform the computation with the correctly-rounded operation. It can contribute to enhancing the accuracy as well. 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, and this study also utilizes the same scheme. This scheme enables one not only to return the correctly-rounded result but also to adjust the accuracy with a certain granularity, and reproducibility is ensured even at tunable accuracy. Moreover, unlike the other approaches, it has a great advantage that it can be built upon standard BLAS implementations: a good performance can be expected with low development cost. On the other hand, ReproBLAS[5] [5] delivers reproducibility without correctly-rounded operations. Their approach – that originates from the works by Rump, Ogita, and Oishi [19, 20, 22] – cuts (rounds) some lower bits that may cause rounding-errors and compute 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 work is the only example that addressed the reproducibility of computed solutions of CG methods. The other implementations do not provide sparse operations as of now. Although the ExBLAS based CG method has not supported GPUs yet, this study compares it with our proposed method on CPUs.

Apart from the above, there are many studies and software developed for improving the computation accuracy. Although most of them are mainly intended for improving accuracy, those can also be

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

Algorithm 2 Splitting of a 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: ρ = `ceil`$((\texttt{log2}(\mathbf{u}^{-1}) + \texttt{log2}(n))/2)$
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: τ = `ceil`$(\texttt{log2}(\mu))$
8: $\sigma = 2^{(\rho+\tau)}$
9: $\boldsymbol{x}_{\text{split}}[j]_i$ = `fl`$((\boldsymbol{x}_i + \sigma) - \sigma)$
10: $\boldsymbol{x}_i$ = `fl`$(\boldsymbol{x}_i - \boldsymbol{x}_{\text{split}}[j]_i)$
11: $\mu = \max_{1\le i\le n}(|\boldsymbol{x}_i|)$
12: **end while**
13: $s_x = j$
14: **end function**

used for the purpose of relaxing the reproducibility issue. In particular, the work of the ExBLAS based CG methods, which were previously mentioned, demonstrates some examples that can achieve reproducibility just through an accurate computation method (with only floating-point expansions and the FMA instruction). The other examples of accurate linear algebra computations include below. 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. Also, 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, we refer to 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 whole 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 $\mathbb{F}$ be 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 inner product results.

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

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

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

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

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

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

The former means that the accuracy of the final result can be controlled by omitting some lower split vectors. The key point is that the later: the inner products of the split vectors can be computed with the standard floating-point arithmetic: for FP64 data, the DDOT

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

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

routine provided in BLAS such as MKL and cuBLAS can be used[9]. Since $\texttt{fl}\left(\left({\boldsymbol{x}_{\texttt{split}}}^{(p)}\right)^T {\boldsymbol{y}_{\texttt{split}}}^{(q)}\right)$ has no round-off error, that is error-free. Even if it is computed with a non-reproducible operation, the result becomes reproducible.

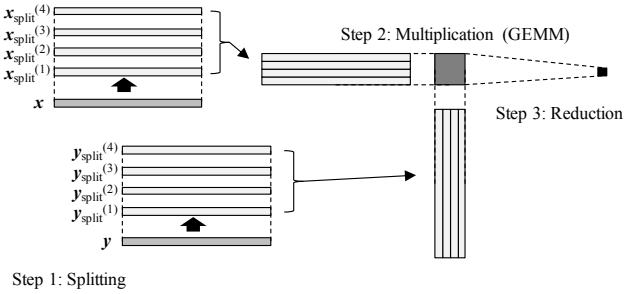

Figure 1: Dot-product with Ozaki scheme (when the number of split vectors is 4).

After that, the accurate and reproducible result is obtained by the summation of the all-to-all inner products of the split vectors. The summation can be computed by a correctly-rounded method such as NearSum [21]. Although it is also possible to compute using working-precision floating-point operations, the following points must care:

- The summation must be computed using a reproducible way. Since the summation is element-wise, fixing the computational order is not difficult and not costly.
- `log2` in Algorithm 2 must be computed by some reproducible way 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.

In this study, we use a correctly-rounded summation, NearSum, to observe the result obtained by completely eliminating the rounding-error introduced in inner products.

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

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 operation. Among them, the factor that may disturb the reproducibility of computed solutions on many-core processors is the operations that consist of inner-product operations: sparse matrix-vector multiplication (SpMV), dot-product (DOT), and 2-norm (NRM2). In this study, those 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 addition, we need to take care of the consistency for the use or non-use of the FMA operation on AXPY. In our implementation, FMA is explicitly used when it can be applied. Besides, we need to take care of the use of fast and less accurate computations for mathematical functions (e.g., `-fp-model fast` on ICC). Hence, 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, and those computations are performed 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 taken into account (i.e., symmetric matrices are given to the computation after expanded to general matrices). The coefficient sparse matrix is stored using the Compressed Sparse Row (CSR) format, which is one of the common formats. We note that the choice of the sparse format does not affect the performance discussion in this study as this study aims to see the relative difference. On the GPU implementations, whereas the BLAS routines are performed on GPUs, the scalar value computations are performed on CPUs.

4.2 FP64Oz-CR

The Ozaki scheme is installed into DOT, NRM2, and CSRMV in the FP64 implementation formerly mentioned. In the Ozaki scheme, the internal computations are performed using MKL on CPUs as well

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	5,080,961	726,713	7.0	electromagnetics problem
2	gridgena	48,962	512,084	10.5	optimization problem
3	cfd1	1,825,580	70,656	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

as cuSparse and cuBLAS on CPUs. 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 intrinsics for arithmetic on GPUs. 2^{τ} at lines 7–8 is computed using NextPowTwo [18]. AXPY is implemented using FMA (with the intrinsic) by ourselves because whether FMA is used or not is unknown.

In addition, we employed 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. It contributes to increasing the performance as the splitting becomes the major cost in the Ozaki scheme on memory-bound operations.
(2) The inner products of the split vectors can be performed using a dense matrix-multiplication (GEMM) by combining multiple split vectors into a 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 idea 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 cuSparse as cusparseDcsrmm. However, on CPUs, we do not use this idea because the performance 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 on each element in the split matrices/vectors, and 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 chosen to be as small as possible in the powers of 2. We can bias the σ to reduce the number of split data on either the matrix or vector by increasing σ on a side and decreasing σ on another side with the same amount. If SpMM is used in the computation, the reduction of the number of split matrices increases a chance to a speedup in general. Our GPU implementation decreases the σ on the matrix side with the minimum number that decreases the number of split matrices on the GPU implementation. On the other hand, the CPU implementation, which does not use SpMM in the computation, decreases the σ on the matrix side with the maximum number that does not change the number of split matrices. This strategy increases the chance to reduce the number of split vectors. The optimal σ is determined in a trial-and-error way by performing the matrix splitting with the σ reduced step by step. As this determination is performed once before starting the iterations of the CG method, the cost is not high.

[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.

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 is assigned, "`numactl --localalloc`" is 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 is assigned (64 cores are used for the computation[11]), memory-mode: flat, clustering-mode: quadrant, KMP_AFFINITY=scatter, "`numactl --preferred 1`" is used for the execution (MCDRAM preferred), on the Oakforest PACS system operated by 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 codes are compiled using the following options: for CPUs, "`-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 are disabled.

We collected eight symmetric positive definite matrices from the SuiteSparse Matrix Collection [4] as shown in Table 1. Those 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 is terminated when $||\boldsymbol{r}_i||/||\boldsymbol{b}|| \leq 10^{-16}$. Hereinafter, the residual plots show $||\boldsymbol{r}_i||/||\boldsymbol{b}||$.

[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) the number of iterations on different platforms. Note: in all cases, the true residual was computed on the correctly-rounded operations using the Ozaki scheme. FP64Oz-CR got the 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) The 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: The number of split matrices required to achieve correct-rounding, the total execution time until convergence, and the execution time overhead (FP64Oz-CR/FP64) on CPUs. † corresponds to the expected overhead of FP64Oz-CR.

#	Matrix	# of split mats†	CPU1			CPU2		
			FP64 (secs)	FP64Oz-CR (secs)	Overhead (times)	FP64 (secs)	FP64Oz-CR (secs)	Overhead (times)
1	tmt_sym	4	6.85E+00	1.81E+02	26.4	6.12E+00	2.53E+02	41.4
2	gridgena	3	6.12E-01	5.17E+00	8.4	9.93E-01	1.14E+01	11.5
3	cfd1	4	1.02E+00	1.17E+01	11.4	1.66E+00	2.07E+01	12.5
4	cbuckle	7	3.60E+00	4.63E+01	12.9	9.17E+00	1.08E+02	11.8
5	BenElechi1	4	7.97E+01	1.44E+03	18.0	6.04E+01	1.19E+03	19.7
6	gyro_k	7	9.85E+00	1.30E+02	13.2	2.44E+01	2.52E+02	10.3
7	pdb1HYS	4	4.24E+00	5.06E+01	11.9	7.72E+00	6.31E+01	8.2
8	nd24k	4	3.57E+01	4.86E+02	13.6	2.49E+01	3.09E+02	12.4

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

#	Matrix	# of split mats†	GPU1			GPU2		
			FP64 (secs)	FP64Oz-CR (secs)	Overhead (times)	FP64 (secs)	FP64Oz-CR (secs)	Overhead (times)
1	tmt_sym	3	2.47E+00	4.32E+01	17.4	3.29E+00	7.12E+01	21.6
2	gridgena	2	2.44E-01	3.04E+00	12.4	2.88E-01	3.50E+00	12.2
3	cfd1	3	4.46E-01	5.16E+00	11.6	5.30E-01	7.41E+00	14.0
4	cbuckle	6	2.90E+00	2.93E+01	10.1	3.58E+00	3.88E+01	10.8
5	BenElechi1	3	2.73E+01	3.47E+02	12.7	3.59E+01	5.96E+02	16.6
6	gyro_k	6	7.10E+00	5.60E+01	7.9	8.29E+00	7.66E+01	9.2
7	pdb1HYS	3	1.99E+00	1.20E+01	6.0	2.34E+00	1.64E+01	7.0
8	nd24k	3	8.60E+00	4.82E+01	5.6	1.13E+01	7.11E+01	6.3

5.1 Reproducibility, convergence, and accuracy

Table 2 shows the relative true residual ($||\boldsymbol{b} - \boldsymbol{Ax}_i||/||\boldsymbol{b}||$) when $||\boldsymbol{r}_i||/||\boldsymbol{b}|| \leq 10^{-16}$ and the number of iterations on 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 on four platforms became identical: that is, reproducibility was ensured by the Ozaki scheme. We can also see that some cases converge with fewer iterations by FP64Oz-CR with accurate computation when compared with FP64 implementations.

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]. We can see some significant differences in the convergence plots among five cases, but in all cases, the solution converges to a similar value on the same order (but most of them are non-identical as shown in Table 2). In terms of the residual $\boldsymbol{r}_i$, FP64Oz-CR often converged

[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.

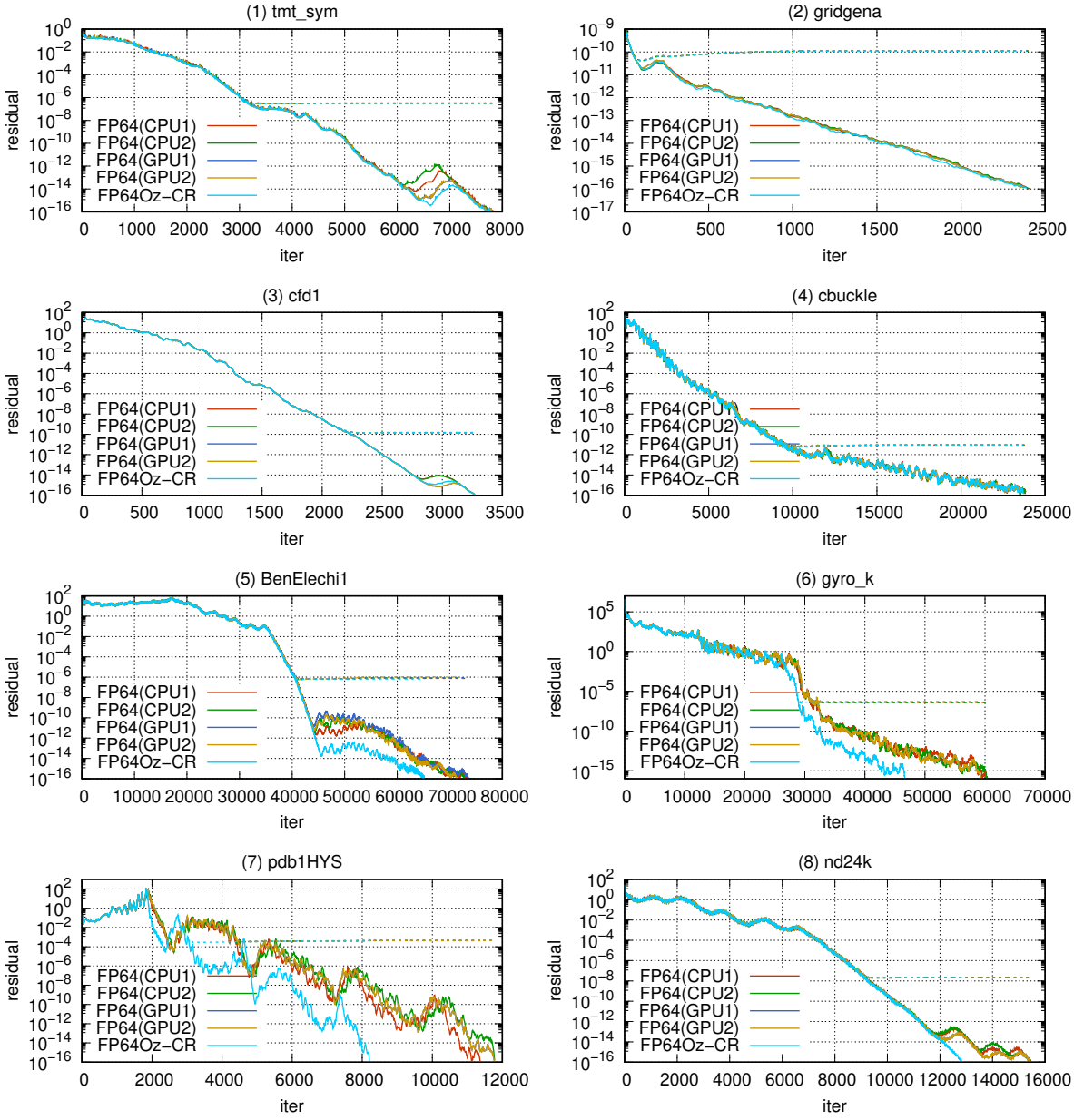

Figure 2: Convergence plots (at every 10 iterations). The results of FP64Oz-CR on four platforms are shown with one line as they are identical. Solid lines show the relative residual $||r_i||/||b||$ and dotted lines show the relative true residual $||b-Ax_i||/||b||$.

with fewer iterations than FP64, but in those cases, the stopping criterion of 10^{-16} was too small.

5.2 Performance (overhead)

Before showing the experimental results, we discuss the expected performance. When SpMV with the Ozaki scheme is computed using SpMM (i.e., our GPU implementation. The CPU implementation currently does not use it.), the execution time overhead of FP64Oz-CR against FP64 per iteration can be roughly estimated. Ideally, the overhead becomes close to d times if the matrix is dense enough, where d is the number of split matrices – it 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 the 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 (the accesses to vector $\boldsymbol{x}$ at lines 9 and 10 in Algorithm 3), but it

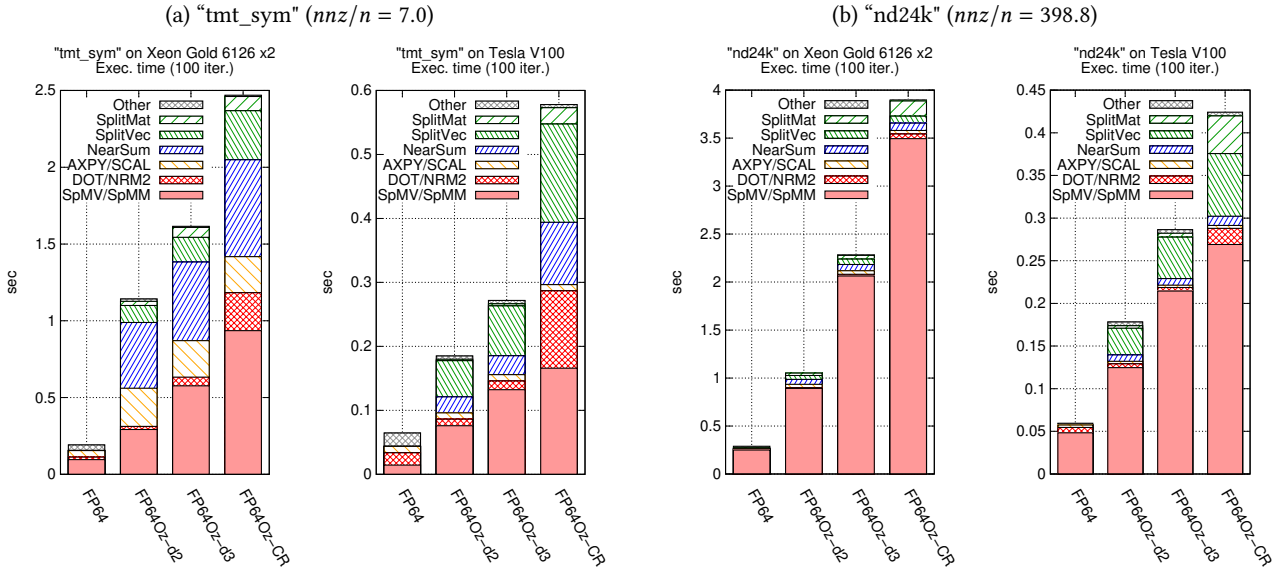

Figure 3: Execution time breakdown of FP64, FP64Oz-dn, and FP64Oz-CR for 100 iterations on CPU1 and GPU1. FP64Oz-dn shows the result of FP64Oz-CR executed with a specified d (the number of split matrices/vectors).

is eliminated on CG methods since the splitting is performed only once before iterations. 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, as SpMV closes to O(n) and or the execution efficiency becomes low owing to the frequent random memory access, the other operations cost than SpMV becomes non-negligible, and the overhead becomes non-predictable. As DOT requires $4d$ times overhead assuming cache hits all split vectors, the cost for DOT (and NRM2) may become dominant instead of SpMV. Moreover, the number of d is unpredictable in CG methods because the vectors are updated during the iterations. Besides, as d becomes large, the chance to cache-miss 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, accordingly. Note that the required number of splits becomes minus-one on GPUs when compared with that on CPUs as we used the asymmetric-splitting technique as 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 can be understood as the expected overhead, as we explained before. When compared with them, approximately 1.3 – 7.2 times additional overhead was observed on GPUs, and we can see the tendency that the smaller nnz/n it has, the greater the overhead takes, 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 when FP64Oz-CR, the matrix splitting cost (SplitMat) includes the tuning cost for the optimal asymmetric-splitting. However, it is not so large within 100 iterations.

5.3 Comparison with the ExBLAS-based CG

Iakymchuk et al. has 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 the proposed method. As with the proposed method, the ExBLAS-based implementation installs correct-rounding in all dot-product operations in CG through the ExBLAS scheme. The ExBLAS approach efficiently combines the Kulisch long accumulator [11] and floating-point expansions (FPEs). While 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., not severe dynamic ranges and/ or condition numbers. Hence, ExBLAS aims to use as much as possible FPEs due to their speed and only occasionally long accumulators. For instance, long accumulators are used when the accuracy of FPEs is not sufficient or at the final rounding to FP64. One clear advantage of the ExBLAS-based implementation against our proposed approach is the low memory consumption: it uses 2097 bits for a long accumulator and 192-512 bits for an FPE per MPI process, while our proposed method consumes a large amount of memory for storing split matrices (i.e., in proportion to the number of split data d).

Table 5: The results of the ExBLAS-based implementation (FP64Ex-CR): the number of iterations, the execution time, and the 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	49.3	5.48E+02	89.4
2	gridgena	2467	7.97E+00	13.0	1.37E+01	13.8
3	cfd1	3278	1.59E+01	15.6	2.81E+01	17.0
4	cbuckle	23828	3.24E+01	9.0	5.95E+01	6.5
5	BenElechi1	73838	1.28E+03	16.1	2.49E+03	41.2
6	gyro_k	60103	9.74E+01	9.9	1.82E+02	7.5
7	pdb1HYS	11839	3.77E+01	8.9	9.61E+01	12.5
8	nd24k	15415	1.66E+02	4.7	5.49E+02	22.1

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 the flat MPI version. We have conducted the 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.

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. Despite that, the obtained solutions and the number of iterations do not have a significant difference. Since both FP64Oz-CR and FP64Ex-CR take different strategies for accurate and reproducible computations, their overhead compared to the baseline FP64-based implementation depends on the matrix at hand and the processors. The FP64Ex-CR achieved better (lower) overhead on 5 out of 8 matrices on CPU1. On the other hand, the proposed method FP64Oz-CR performs better on 6 out of 8 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.

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

In this study, we installed reproducibility by correctly-rounded operations, but the Ozaki scheme can adjust the accuracy at a certain 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 splittings, and reducing it improves the performance, as shown in Figure 3.

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

Figure 4 demonstrates an example as 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 actually, $d = 3$ is enough to achieve higher accuracy than FP64. However, there is currently no light and easy way to determine the optimal number in advance (i.e., the number needed to achieve double-precision accuracy at least or to get the solution at the shortest time). There is a risk to be unconverged or take a lot of iterations by using inadequate choice. To find out the optimal number, we have to try it once. Nevertheless, this approach can be useful when we aim 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 the reproducibility, i.e., the number of bits that can be reproduced, through reducing rounding errors. However, how much accuracy is needed to ensure, for example, bit-level reproducibility is a problem (input) dependent. Iakymchuk et al. [8, 9] demonstrated to achieve reproducibility only with an accurate computation method (only with FPE, i.e., the ExBLAS scheme without long accumulator) by focusing on certain problems. It contributes to improving the performance more than the ExBLAS approach.

7 CONCLUSION

This paper presented 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 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 comparable or better performance than an existing work based on the ExBLAS approach in many cases. Also, our proposed approach has an advantage due its low development cost as it relies on vendor-provided BLAS implementations. Moreover, we demonstrated some cases that accurate computation through the Ozaki scheme improved

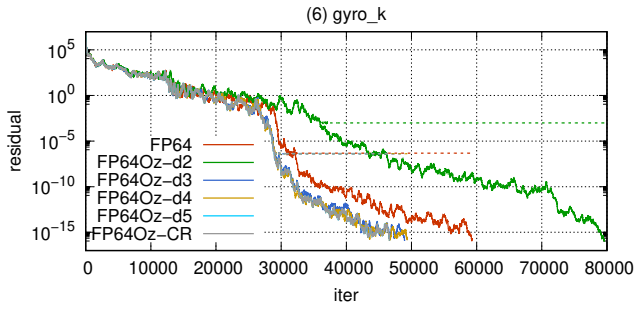

Figure 4: Convergence plot for the gyro_k matrix (at every 10 iterations) on CPU1. Solid lines show the relative residual $||r_i||/||b||$ and dotted lines show the relative true residual $||b-Ax_i||/||b||$.

the convergence, even though it did not contribute to reducing the total execution time to get the solution. The source code of our implementations is available together with OzBLAS[14].

As future work, we can adopt the mixed-precision approach to our proposed method. While this study used FP64 for the 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, for example on dot-product, instead of using DGEMM for the computation as we did in this study, one can use SGEMM or Tensor Core GEMM (gemmEx) to compute FP64 input/ output. It may contribute to improving the performance on hardware with a limited FP64 support.

ACKNOWLEDGMENT

This research was partially supported by the Japan Society for the Promotion of Science (JSPS) KAKENHI Grant #19K20286 and the EU H2020 research, innovation programme 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 Joint Center for Advanced High Performance Computing (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