Parallel Computing 115 (2023) 102996

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Multi-level parallel multi-layer block reproducible summation algorithm ☆

Kuan Li [a,*], Kang He [b], Stef Graillat [c], Hao Jiang [d], Tongxiang Gu [e], Jie Liu [b]

[a] Dongguan University of Technology, No. 1, Daxue Rd., Songshan Lake, Dongguan, 523808, China
[b] Science and Technology on Parallel and Distributed Processing Laboratory, National University of Defense Technology, Changsha, 410000, China
[c] Sorbonne Université, CNRS, LIP6, Paris, F-75005, France
[d] College of Computer, National University of Defense Technology, Changsha, 410000, China
[e] Institute of Applied Physics and Computational Mathematics, Beijing, 100094, China

ARTICLE INFO

Keywords:
Reproducibility
1-Reduction
Summation
Multi-level parallel
Multi-layer block

ABSTRACT

Reproducibility means getting the bitwise identical floating point results from multiple runs of the same program, which plays an essential role in debugging and correctness checking in many codes (Villa et al., 2009). However, in parallel computing environments, the combination of dynamic scheduling of parallel computing resources. Moreover, floating point nonassociativity leads to non-reproducible results. Demmel and Nguyen proposed a floating-point summation algorithm that is reproducible independent of the order of summation (Demmel and Nguye, 2013; 2015) and accurate by using the 1-Reduction technique. Our work combines their work with the multi-layer block technology proposed by Castaldo et al. (2009), designs the multi-level parallel multi-layer block reproducible summation algorithm (MLP_rsum), including SIMD, OpenMP, and MPI based on each layer of blocks, and then attains reproducible and expected accurate results with high performance. Numerical experiments show that our algorithm is more efficient than the reproducible summation function in ReproBLAS (2018). With SIMD optimization, our algorithm is 2.41, 2.85, and 3.44 times faster than ReproBLAS on the three ARM platforms. With OpenMP optimization, our algorithm obtains linear speedup, showing that our method applies to multi-core processors. Finally, with reproducible MPI reduction, our algorithm's parallel efficiency is 76% at 32 nodes with 4 threads and 32 processes.

1. Introduction

Reproducibility is widely considered to be an essential requirement of the scientific process [1,2]. The concept of "numerical Reproducibility" was first proposed by He and Ding of Lawrence Berkeley National Laboratory in 2001 [3]. They used Kahan's self-compensated summation and Bailey's double-double precision summation to achieve the reproducibility of large-scale scientific simulations and provided an MPI operator. They also achieved a good result in climate modeling on distributed memory parallel computers and encountered severe difficulty calculating sea surface height in an ocean circulation model.

McNutt stressed the importance of reproducibility in *Science* in 2014, to demonstrate excellence in transparency [4]. In 2015, Taufer pointed out the numerical reproducibility challenges on extreme-scale multi-threading GPUs [5]. Taufer also presented a novel set of Reproducibility Enhancement Principles (REP) targeting disclosure challenges involving computation [6].

The non-reproducibility of floating-point computation arises because the associative law is no longer guaranteed in floating-point arithmetic [7]. However, for large-scale parallel computing, the number of processors, reduction tree shape, data assignment, and data partitioning have unfavorable effects on reproducibility. Therefore, there have been numerous recent scientific payoffs at conferences addressing the need for reproducibility [8–12]. Taufer improved reproducibility and stability in large-scale molecular dynamic simulations on GPUs and presented the development of a library of mathematical functions that use fast and efficient algorithms to fix the error produced by the equivalent operations performed by GPUs [13]. Chapp built containerized environments for reproducibility and traceability of scientific workflows [14]. Intel's Math Kernel Library (MKL) supports reproducibility under certain restrictive conditions [15]. NVIDIA's cuBLAS routines are reproducible under the same conditions [16].

☆ This research is partly supported by the National Key Research and Development Program of China under Grant 2020YFA0709803, 173 Program under Grant 2020-JCJQ-ZD-029, the Science Challenge Project under Grant TZ2016002. This work was also supported by the NuSCAP (ANR-20-CE48-0014) project of the French National Agency for Research (ANR).

* Corresponding author.
E-mail addresses: likuan@dgut.edu.cn (K. Li), hekang2019@nudt.edu.cn (K. He), stef.graillat@sorbonne-universite.fr (S. Graillat), haojiang@nudt.edu.cn (H. Jiang), txgu@iapcm.ac.cn (T. Gu), liujie@nudt.edu.cn (J. Liu).

https://doi.org/10.1016/j.parco.2023.102996
Received 21 March 2022; Received in revised form 4 December 2022; Accepted 12 January 2023
Available online 18 January 2023

In 2013, Demmel and Nguyen proposed fast and reproducible sequential summation algorithms [8], based on error-free vector transformation technology proposed by Rump [17–19]. They developed the 1-Reduction technique to attain local boundaries, which evaluates the local maximum absolute value instead of the global maximum absolute value. Thus one reduction can be removed. The main idea of the algorithms is to "pre-round" the input floating point numbers in K consecutive **bins** by precomputed boundaries with an interval of W bits, requires no communication. Subsequently, they proposed a series of reproducible algorithms [11,12] and the ReproBLAS software [10].

Based on investigating the reproducibility of HPC applications [20], Hoefler presented the challenges of reproducibility in HPC at the SC2015 [21]. Iakymchuk et al. developed the ExBLAS software [22–27], which is fast, accurate, and reproducible BLAS, which does not support ARM architecture. Mukunoki proposed DGEMM using tensor cores and its accurate and reproducible versions [28]. Chohra introduced the RARE-BLAS (Reproducible, Accurately Rounded, and Efficient BLAS) that benefits from recent accurate and efficient summation algorithms [29,30], which only provided Level 1 BLAS(asum, dot, and nrm2) and Level 2 BLAS(gemv) routines.

Most systems in high-performance computing feature a hierarchical hardware design, such as shared memory nodes with several multi-core CPUs. For some optimizations, it is necessary to block operations into chunks. Castaldo introduced a new and more general framework named superblock in 2009 [31].

We could use a multi-block algorithm to combine the various parallel designs, including MPI and OpenMP, as well as use intrinsic optimization in SIMD. As the parallel scale increases, the uncertainty of computation will also increase. We present and develop a 1-Reduction multi-level parallel multi-block reproducible summation algorithm, which can compute a rigorously reproducible sum of floating points in parallel computing environments. with only basic assumptions about the underlying arithmetic. We reproducibly attain results for the desired accuracy with the lowest absolute error bound

$$n \cdot 2^{(1-K)\cdot W-1} \cdot \max_i |v_i|,$$

where v is the input vector, n is the length of v, K is the number of bins, W is the width of a bin, and all operations are performed in rounding to the nearest mode. The error bound is the same as that in ReproBLAS, even though we add OpenMP and SIMD optimization.

We compare our algorithm with the reproducible summation function (**reproBLAS_dsum**) in ReproBLAS. The numerical tests illustrate our algorithm is efficient. We have make these codes available online.[1]

The remainder of this paper is organized in the following way: Section 2 presents some notation and existing algorithms that will be used throughout the paper. Section 3 presents our 1-Reduction multi-level parallel multi-layer block reproducible summation algorithms and analyses the error bounds. Section 4 contains some numerical experiments to check the reproducibility and the performance of our algorithm. Section 5 is the conclusion.

2. Notation and background

2.1. Notation

In this paper, we assume that the floating-point arithmetic in use complies with the IEEE-754 (2019) standard and only consider that all operations are performed in rounding to the nearest mode. We use $\mathbb{F}, \mathbb{Z}$ to denote the set of floating-point numbers and integers, respectively.

The floating-point number system is a proper subset of the real number system, a floating-point number f can be represented as $f = (-1)^s \times m \times 2^e \in \mathbb{F}$, where $s = \pm 1$ is the sign, m is the significand of f, e is the exponent and $e_{max} \geq e \geq e_{min}$. p denotes the precision. $m = m_0.m_1 m_2 \cdots m_{p-1}$ is also known as the mantissa, where $m_i \in \{0, 1\}$. We define machine epsilon $\epsilon = 2^{-p}$ as the spacing distance between 1 and the floating point number greater than and closest to 1.

We use $\mathrm{fl}(\cdot)$ to denote the evaluated result of an expression, and use $\mathrm{ulp}(f)$ to denote the unit in the last place, which is defined as the spacing between two consecutive floating-point numbers of the same exponent e. We denote the unit in the first place by $\mathrm{ufp}(f) = 2^e$. The unit roundoff $\mathbf{u}$ is the upper limit of the rounding error, it is closely related to the $\mathrm{ulp}(f)$ function and satisfied $\mathbf{u} = \epsilon$ in rounding to nearest mode. Let $x \in \mathbb{R}$, denote $x\mathbb{Z} = \{n \cdot x, n \in \mathbb{Z}\}$, we can deduce the following lemmas.

Lemma 1. *Let $f, x, y \in \mathbb{F}$, where $x, y \in \mathrm{ulp}(f)\mathbb{Z}$. If $|x + y| < \epsilon^{-1}\,\mathrm{ulp}(f)$ and no overflow occurs then $x + y \in \mathbb{F}$, i.e. $x + y$ can be computed exactly.*

2.2. Reproducible summation with 1-reduction technique

First, we introduce the error-free vector transformations ExtractVector3() [9, Algorithm 5], which is the kernel of reproducible algorithms in this section.

Second, we briefly review the precomputed values

$$M_{[k]} = 0.75 \cdot \epsilon^{-1} \cdot 2^{k\cdot W},\ W \in \mathbb{Z},\ W < \log_2(\epsilon^{-1}), k = 1, \ldots, K,$$

which is independent of input data and requires no communication. Input floating-point numbers will be split according to boundaries $\mathrm{ulp}(M_{[k]}) = 2^{k\cdot W}$. Each processor can use its own local boundary, which is the largest one to the right of the leading bit of local maximum absolute value.

The 1-Reduction technique can be described by means of binning. Each bin is characterized by an index i, which means that the bin covers the bits from position iW to position $(i+1)W-1$ in the exponent range. Each input floating-point value will be put in a number of contiguous bins. The values inside each bin will be summed respectively.

Before summation, it is necessary to adjust the index of bins by the local maximum absolute value. This adjustment is Algorithm 1, which is part of Algorithm 6 in [9].

Algorithm 1 [9] $[S, C] = \mathrm{update}(m, K, W, S, C)$

Require: m is the local maximum absolute value, W is the bin width, K is the number of bins to be kept. $S, C \in \mathbb{F}^K$ are already initialized.

```
if m ≥ 2^{W-1} * ulp(S_1) then
    g = 1 + ⌊log_2(m/(2^{W-1} * ulp(S_1)))/W⌋
    for k = K down to (g + 1) do
        S_k = S_{k-g}, C_k = C_{k-g}
    end for
    for k = 1 to min(g, K) do
        C_k = 0, S_k = 1.5 * 2^{g*W} * ufp(S_k)
    end for
end if
```

Ensure: S, C is updated by m.

We need to avoid overflow of bins, which means the maximum number of floating point numbers n that can be summed without overflow have to satisfies

$$n \leq 2^{-(k+1)\cdot W-1} \cdot \mathrm{ufp}(0.75 \cdot \epsilon^{-1} \cdot 2^{k\cdot W}) = \epsilon^{-1} \cdot 2^{-W-2}.$$

As for the longer input vector, it is required to adopt the method of renormalization and capture the carry-bit after every $NB \leq \epsilon^{-1} \cdot 2^{-W-2}$ addition. This process is Algorithm 2, which is part of Algorithm 6 in [9].

The Algorithm 1 and Algorithm 2 are both used in our parallel algorithm in Section 3.1.1.

Algorithm 3 is the pseudo-code for sequential K-fold 1-Reduction reproducible summation, which is a building block for the parallel case. Denote by $v_{[l:r]} = [v_l, \ldots, v_r]$ a block of input vector v from index l to r. We set forth that the vector $(S_1, \ldots, S_k, C_1, \ldots, C_k)$ is denoted as (S, C), and C_k is stored in the same precision as S_k.

[1] https://github.com/qihaijun/MLP_rsum.git

Algorithm 2 [9] $[S, C] = \text{renormalize}(K, S, C)$

Require: K is the number of bins to be kept. $S, C \in \mathbb{F}^K$.

for $k = 1$ to K **do**
  if $S_k \geq 1.75 * \text{ufp}(S_k)$ **then**
    $S_k = S_k - 0.25 * \text{ufp}(S_k)$, $C_k = C_k + 1$
  else if $S_k < 1.25 * \text{ufp}(S_k)$ **then**
    $S_k = S_k + 0.5 * \text{ufp}(S_k)$, $C_k = C_k - 2$
  else if $S_k < 1.5 * \text{ufp}(S_k)$ **then**
    $S_k = S_k + 0.25 * \text{ufp}(S_k)$, $C_k = C_k - 1$
  end if
end for

Ensure: S_k, C_k are the trailing and leading parts of the aggregation of values in the k-th leftmost bins, respectively.

Algorithm 3 [9, Algorithm 6] Sequential Reproducible Summation: $[S, C] = \text{rsum}(v, K, W)$

Require: v is a vector of n floating-point numbers. W is the bin width satisfies $1 \leq W < -\log_2 \epsilon$. $M_i = 0.75 \cdot \epsilon^{-1} \cdot 2^{i \cdot W}, i_{min} \leq i \leq i_{max}$ are precomputed. K is the number of bins to be kept. $NB \leq \epsilon^{-1} \cdot 2^{-W-2}$ is the maximum unit size for renormalization. $S, C \in \mathbb{F}^K$.

for $k = 1$ to K **do** ▷ Initialization
  $S_k = M_{i_{min}+K-k}$, $C_k = 0$
end for
for $i = 1$ to n step NB **do**
  $lN = \min(i + NB - 1, n)$
  $m = \max(|v_{[i:lN]}|)$ ▷ Local maximum absolute
  $[S, C] = \text{update}(m, K, W, S, C)$
  for $k = 1$ to K **do** ▷ Deposit
    $[S_k, v_{[i:lN]}] = \text{ExtractVector3}(S_k, v_{[i:lN]})$
  end for
  $[S, C] = \text{renormalize}(K, S, C)$
end for

Ensure: S_k, C_k are the trailing and leading parts of the aggregation of values in the k-th leftmost bins, respectively.

3. 1-Reduction multi-level parallel multi-layer block reproducible algorithm

By analyzing Algorithm 3, the portion for computing maximum absolute value and depositing operator takes the largest proportion of costs and need to be optimized.

In this section, we will improve Algorithm 3 with optimization and parallelization, by means of introducing a special reduction operation RepReduce() [9, Algorithm 7] and designing the multi-level parallel structure with SIMD, OpenMP, and MPI. RepReduce() could avoid computing the global maximum absolute value in the parallel case. Using Algorithm 3 in each block, we can get a pair of arrays (S, C) as the local result, where S, $C \in \mathbb{F}^K$.

3.1. Multi-level parallel algorithm design

In this subsection, we introduce the SIMD, OpenMP, and MPI optimization, which could be used for the multi-layer block algorithm on the ARM architecture. Different types of multi-core CPUs have a large number of parallel computing cores and multi-level caches, which could provide our algorithms with multi-level parallel computing capabilities such as instruction-level, thread-level, and data-level, respectively.

3.1.1. SIMD optimization

To obtain higher efficiency of program execution and reduce the workload of the algorithm while maintaining performance, we optimize Algorithm 3 in terms of the perspective of intrinsic, prefetch operation, and blocking.

Since there has been a version of assembly optimization for Algorithm 3 on the X86 platform, the focus of our SIMD optimization will be on the ARM platforms. NEON is a SIMD extension structure for the ARM cortex-A processor, and we use intrinsic (internal functions) as the means to implement NEON. Intrinsic is simple to use and easy to maintain, making it suitable for cross-platform optimization.

We vectorize data in bins vs_0_0, arrays vx_0, and maximum absolute value. Then the arrays are added to the bins in batches. In addition, since accumulation is a memory-intensive calculation, each batch of array elements involved in the computation is prefetched into the cache by PRFM instruction to prefetch the data which is used in the calculation kernel part to effectively hit the cache and improve memory access efficiency. We test different data prefetch lengths to obtain the best performance. The algorithm performs best when the prefetch length is 1024.

The kernel code of SIMD parallel implementation of ExtractVector3() algorithm, called ExtractVectorSIMD(), is implemented as follows.

```
    ...
vs_0_0 = vdupq_n_f64(priY[0]);
    ...
vx_0 = vld1q_f64(X);
    ...
__asm__volatile("prfm PLDL1KEEP, [X, #1024]");
    ...
vq_0 = vs_0_0;
vs_0_0 = vaddq_f64(vs_0_0,
        vreinterpretq_f64_u64(vorrq_u64(
        vreinterpretq_u64_f64(vx_0), bmt)));
vq_0 = vsubq_f64(vq_0, vs_0_0);
vx_0 = vaddq_f64(vx_0, vq_0);
    ...
```

Finally, only one of vs_0_0 is returned after SIMD computation by converting vs_0_0 back to a scalar, called SIMD_reduction, shown as follows.

```
    ...
float64x2_t s_tmp = vdupq_n_f64(priY[0]);
s_tmp = vsetq_lane_f64(0, s_tmp, 1);
vs_0_0 = vsubq_f64(vs_0_0, s_tmp);
priY[0] = vgetq_lane_f64(vs_0_0, 0)
        + vgetq_lane_f64(vs_0_0, 1);
    ...
```

Consequently, Algorithm 4 is the pseudo-code for reproducible summation algorithm with SIMD parallel design.

3.1.2. OpenMP optimization

OpenMP [32] is a multi-thread programming scheme for shared memory parallel systems, provides a high-level abstract syntax expressing parallelism. OpenMP is suitable for parallel programming on multi-core CPU machines. Application requirements or system restrictions may limit the number of MPI processes that can be used. In this case, using OpenMP in addition to MPI can increase the amount of parallelism. Some applications show an unbalanced workload at the MPI level that might be hard to overcome. In this case, OpenMP provides a convenient way to address the imbalance by exploiting extra parallelism on a finer granularity and by assigning a different number of threads to different MPI processes, depending on the workload. Programmers specify their intentions by adding special pragma to the source code so that the compiler can automatically parallelize the program.

In this section, we design a multi-thread reproducible summation algorithm based on OpenMP. OpenMP uses the fork-join mode, i.e., only

Algorithm 4 Sequential Reproducible Summation: $[S, C] = \text{rsumSIMD}(v, K, W)$

Require: v is a vector of n floating-point numbers. W is the bin width satisfies $1 \le W < -\log_2 \epsilon$. $M_i = 0.75 \cdot \epsilon^{-1} \cdot 2^{i \cdot W}, i_{min} \le i \le i_{max}$ are precomputed. K is the number of bins to be kept. $NB \le \epsilon^{-1} \cdot 2^{-W-2}$ is the maximum unit size for renormalization. $VS, VC \in \mathbb{F}^{b \cdot K}$ where b is the number of blocking. $S, C \in \mathbb{F}^K$.

1: **for** $k = 1$ to K **do** ▷ Initialization
2: $S_k = M_{i_{min}+K-k},\ C_k = 0$
3: **end for**
4: **for** $i = 1$ to n step NB **do**
5: $lN = \min(i + 2 * NB - 1, n)$
6: $m = \max(|v_{[i:lN]}|)$
7: ▷ Vectorize maximum absolute value
8: $[S, C] = \text{update}(m, K, W, S, C)$
9: **for** $j = i$ to lN step b **do** ▷ Vectorize Deposit
10: **for** $k = 1$ to K **do**
11: $[V_k, v_{[j:j+b]}] = \text{ExtractVectorSIMD}(V_k, v_{[j:j+b]})$
12: **end for**
13: $S = \text{SIMD\_reduction}(V)$ ▷ SIMD Reduce
14: **end for**
15: $[S, C] = \text{renormalize}(K, S, C)$
16: **end for**

Ensure: S_k, C_k are the trailing and leading parts of the aggregation of values in the k-th leftmost bins, respectively.

Algorithm 5 1-Reduction Multi-level Parallel Multi-layer Block Reproducible Summation Algorithm

Require: v is a vector of n floating-point numbers. W is the bin width satisfies $1 \le W < -\log_2 \epsilon$. K is the number of bins to be kept. $S, C \in \mathbb{F}^K$. $thread\_count$ and $process\_count$ are the numbers of available threads and nodes separately.

1: $mpin = n/process\_count$
2: **for** $f = 1$ to $process\_count$ **do** ▷ MPI parallel
3: **if** $f != process\_count$ **then** $p = mpin$
4: **else** $p = n - mpin * (process\_count - 1)$
5: **end if**
6: $ompn = p/thread\_count$
7: *#pragma omp parallel*
8: **for** $i = 1$ to $thread\_count$ **do** ▷ OpenMP parallel
9: **if** $i != thread\_count$ **then** $t = ompn$
10: **else** $t = p - ompn * (thread\_count - 1)$
11: **end if**
12: ▷ SIMD parallel
13: $l = mpin * (f - 1) + ompn * (i - 1)$
14: $[s, c] = \text{rsumSIMD}(v_{[l:l+t]}, K, W)$
15: $[S_{[f]}, C_{[f]}] = \text{RepReduce}(S_{[f]}, C_{[f]}, s, c)$
16: **end for**
17: **end for**
18: $Final\_sum = \text{MPI\_RepReduce}(S, C)$

Ensure: $Final\_sum$ is the reproducible summation of vector v.

one master thread at the beginning. When parallel computation is needed, several child threads are forked to perform parallel tasks. Each thread takes data from a different location on the same segment of memory, then computes its local maximum absolute value and uses Algorithm 4 to attain a local result. When the threads finish their work, the child threads will join and hand over control to the master thread, i.e., synchronize and terminate, leaving only the master thread, which is achieved by RepReduce() to aggregate all threads, gets only one pair of mantissa S and carries C reproducibility in the master thread. Therefore, we obtain the data according to the ID of threads, then use the following pragma instruction to parallelize the algorithm:

```
#pragma omp parallel
{
    [s, c] = rsumSIMD(v, K, W)
}
[S, C] = RepReduce(S, C, s, c)
/* on master thread only */
```

3.1.3. MPI optimization

After we get the pairs from each thread, we need to aggregate all pairs into one final pair by RepReduce(). Then, we compute $Final\_sum$ in a certain order in Algorithm 5.11 in [11] to guarantee reproducibility. We denote the whole action above as MPI_RepReduce().

3.2. MLP_rsum

Castaldo proposed a t-layer block algorithm for dot product named SuperBlock [31]: Given t temporaries, we could accumulate a sum in t levels, which permits a practitioner to make trade-offs between computational performance, memory usage, and error behavior. We combine the t-layer block technology with the 1-Reduction technique to implement a parallel reproducible summation algorithm.

We perform a parallel design, including SIMD instruction-level parallelization, OpenMP thread-level parallelization, and MPI data-level parallelization, to obtain a 1-Reduction multi-level parallelization multi-layer block algorithm, that is Algorithm 5, with user-defined number of processes $process\_count$ and of threads $thread\_count$.

Note that Algorithm 5 uses only one reduction for aggregating all the partial sums. Although RepReduce for OpenMP and MPI_RepReduce are slightly different, they still are treated as the same reduction operation in the whole algorithm.

Therefore, Algorithm 5 could make full use of multiple parallel computing cores and multi-level caches of modern processors, combining the features of the processor architecture with the multi-level parallel design, which is well suited to highly parallel environments.

3.3. Accuracy

For Algorithm 5, the error is produced by pre-rounding for each input data and executing the "Update" portion, (other portions are performed exactly). Suppose the gap between two consecutive pre-computed boundaries is W bits, K is given, usually $W = 40$, $K = 3$.

s, $S_{[f]}$, $S \in \mathbb{F}^K$. The size of each process and thread are denoted by

$$p = [mpin, \dots, mpin,\ n - mpin * (process\_count - 1)],$$

$$t = [ompn, \dots, ompn,\ p - ompn * (thread\_count - 1)].$$

Size of p is $process\_count$, Size of t is $thread\_count$.

According to Lemma 1, we could deduce the error from different level of parallelism.

At the SIMD level, the error generated by summation in ith thread is

$$a_i \le t_i \cdot \frac{1}{2}\,\text{ulp}(s_K)\ ,$$

where s_K is from the local result $(s,\ c)$. At the OpenMP level, the error generated by aggregating all threads in fth process is

$$b_f \le \sum_{i=1}^{thread\_count} a_i \le p_f \cdot \frac{1}{2}\,\text{ulp}(S_{[f]K})\ .$$

where $S_{[f]K}$ is from the local result $(S_{[f]},\ C_{[f]})$. At the MPI level, the error generated by aggregating all processes is

$$c \le \sum_{f=1}^{process\_count} b_f \le n \cdot \frac{1}{2}\,\text{ulp}(S_K)\ ,$$

Table 1
Non-reproducibility of conventional summation using different compilers and optimization options on ARM platforms.

Compiler	Optimization		FT2000+	Kunpeng920	ThunderX2
	(1)	(2)			
gcc	-fno-fast-math	-O0	−7.74515613378954650e−11	−7.74515613378954650e−11	−7.74515613378954650e−11
		-Ofast	−7.74515613378954650e−11	−7.74515613378954650e−11	−7.74515613378954650e−11
	-ffast-math	-OX	−7.74515613378954650e−11	−1.64574758883120149e−10	−7.74515613378954650e−11
		-Ofast	−1.64574758883120149e−10	−1.64574758883120149e−10	−1.64574758883120149e−10
clang	-fno-fast-math	-O0	−7.74515613378954650e−11	−7.74515613378954650e−11	−7.74515613378954650e−11
		-Ofast	−2.22971173884867961e−11	4.71949378195161322e−12	4.71949378195161322e−12
	-ffast-math	-OX	−7.74515613378954650e−11	−7.74515613378954650e−11	−7.74515613378954650e−11
		-Ofast	−2.22971173884867961e−11	4.71949378195161322e−12	4.71949378195161322e−12

Remark: -OX in Optimization(2) is -O0/-O1/-O2/-O3/-Os/-Og, and the result is the same as the control group in both compilers.

where S_K is from pair the final result $(S,\ C)$. Therefore, the error bound of the K-fold 1-Reduction multi-level parallel multi-layer block reproducible summation algorithm can be estimated to be:

$$\begin{aligned} absolute\_error &\le n \cdot \frac{1}{2}\,\mathrm{ulp}(S_1) \cdot 2^{(1-K)\cdot W} \\ &\le n \cdot 2^{(1-K)\cdot W-1} \cdot \max_i |v_i|. \end{aligned} \tag{1}$$

The error bound is the same as that in ReproBLAS, even though we add OpenMP and SIMD optimization. Based on the above analysis, we can get the following conclusion: when the input vector is given, then the accuracy of Algorithm 5 is tuned by choosing big enough K to attain the accuracy required in each application.

4. Experimental results

In this section, we execute different algorithms on some ARM platforms and X86 platforms. Firstly, we compare some experimental results of conventional summation algorithms and our algorithm on different compilers and optimization options to check the reproducibility. Secondly, the accuracy is shown by computing absolute errors. Last but not least, we compare the performance of the proposed reproducible summation method with ReproBLAS in double precision.

4.1. Reproducibility experiment

The experiments in Table 1 and Table 2 respectively show the results of conventional summation, using different compilers and optimization options on three ARM platforms (including FT2000+, Kunpeng920, and ThunderX2) and three X86 platforms (including Hygon C86 7185, Intel(R) Xeon(R) CPU E5-2620, and Intel(R)Xeon(R) Platinum 8180M). We used the function in ReproBLAS [9]:

$$\sin\left(2\pi \cdot \left(\frac{i}{n} - \frac{1}{2}\right)\right) \tag{2}$$

to generate vector v of size $n = 10^6$, and $i = 1, \ldots, n$. We could see 18 different results, account for non-reproducible.

For example, the results in Hygon are different from the results of the other five platforms with the same compiler and the same option. On Intel E5-2620, there are 9 different results with three kinds of compiler and serval options. On Intel 8180M, there are 5 different results with the icc compiler and five kinds of options.

However, we used function (2) to generate vectors with random size n, and shuffle the vector randomly inside the vectors. Then we calculated them by Algorithm 5 using different compilers and optimization options on different platforms, which are the same as those in Tables 1 and 2. We test 5 vectors, and the results are listed in Table 3. Experimental results show that our algorithms are not affected by the computation order and other factors, obtaining the same results by bit for the same data set, satisfying the reproducibility.

4.2. Accuracy experiment

We use the results calculated by MPFR [33] as accurate results. They were compared with the results computed by Algorithm 5 to obtain absolute errors to verify the accuracy. The results are listed in Table 4.

The condition number $\sum_i |v_i| / |\sum_i v_i|$ of vectors in our experiment is huge. The results in Table 4 can verify the correctness of our numerical analysis (1).

4.3. Performance experiment

The testing platforms include three ARM processors, as shown in Table 5. This subsection includes a single-core SIMD optimization experiment, single-chip multi-core experiment and large-scale node optimization experiment, showing the performance of Algorithm 5.

4.3.1. Single-core SIMD optimization experiment

ReproBLAS does not support SIMD optimization on ARM architecture. We adopt the intrinsic function supporting NEON to parallelize the algorithm and enhance prefetch instructions. Then, the reproducibility summative with SIMD optimization, i.e., Algorithm 4, is realized. The length of arrays that we choose is 5×10^7, and the average value of 20 results is taken as the final result. In Fig. 1, Algorithm 4 runs 2.41, 2.85 and 3,44 times faster than Algorithm 3(Algorithm 6 in [9]), respectively on three platforms. The experiment verifies the efficiency of our SIMD optimization in Section 3.1.1.

4.3.2. Single-chip multi-core parallel optimization experiment

Due to page limitation, only one ARM platform(ThunderX2) is selected to demonstrate the actual effect of Algorithm 5. The length of arrays that we choose is 5×10^7, and the average value of 30 results is taken as the final result. Then the speedup is calculated and shown in Table 6.

As shown in Table 6, when the number of processes (nP) is 1, and the number of threads (nT) is less than 16, Algorithm 5 is scalable. The effect of OpenMP parallel optimization in Section 3.1.2 is illustrated. In addition, we find that the speedup of nT = 8 and nP = 1 is better than that of nT = 1 and nP = 8, indicating the necessity of OpenMP parallel optimization under multi-core architecture. We find the Algorithm runs fastest when nT = 4, nP = 16 and nT = 16, nP = 4 on ThunderX2 with 64 cores. According to Table 5, ThunderX2 has a shared L3 Cache of 32 MB. Therefore setting nT*nP = 16 ∗ 4 is better. According to the NUMA feature, since it has 16 memory channels, each of which corresponds to 4 processor cores, setting nT*nP = 4 ∗ 16 is a good option.

4.3.3. Large-scale node optimization experiment

The experimental platform is the latest cluster of National Supercomputing Center in ChangSha, China. The results are shown in Table 7.

Each compute node is FT2000+(16 cores), with 8 channels of memory and 64 GB in total. In large-scale experiments, Algorithm 5 has good parallel scalability in both multi-process and multi-thread. We

Table 2
Non-reproducibility of conventional summation using different compilers and optimization options on X86 platforms.

Compiler	Optimization		Hygon C86 7185	Intel(R) Xeon(R)	Intel(R)Xeon(R)
	(1)	(2)		CPU E5–2620	Platinum 8180M
gcc	-fno-fast-math	-O0	−1.74665353909333272e−10	−7.74515613378954650e−11	−7.74515613378954650e−11
		-Ofast	−7.74515613378954650e−11	−7.74515613378954650e−11	−7.74515613378954650e−11
	-ffast-math	-OX	−1.74665353909333272e−10	−7.74515613378954650e−11	−7.74515613378954650e−11
		-Ofast	−1.74665353909333272e−10	−1.74665353909333272e−10	−1.01892451682742931e−10
Remark: -OX in Optimization(2) is -O0/-O1/-O2/-O3/-Os/-Og, and the result is the same as the control group.					
icc	–	–	−7.74515613378954650e−11	−7.74515613378954650e−11	−7.74515613378954650e−11
	-fp-model extended	–	1.9104921582955089 3e−10	1.76497300601184042e−10	1.76497300601184042e−10
	–	-O2/-O3	−7.68340646995938981e−12	−7.68340646995938981e−12	5.31814340608711402e−12
	–	-Os	−3.92753789001343235e−12	−3.92753789001343235e−12	−3.92753789001343235e−12
	–	-Ofast	6.18194683997926519e−12	6.18194683997926519e−12	2.54780171764405565e−11
Remark: When the Optimization(1) is -fp-model fast=1/-fp-model 2/-fp-model precise/-fp-model strict/-fp-model source /-fp-model double, fast=or Optimization(2) is -O0/-O1, and the result is the same as the control group.					
icx	–	–	−7.74515613378954650e−11	−7.74515613378954650e−11	−7.74515613378954650e−11
	–	-O1	−8.87225454838900542e−11	−8.87225454838900542e−11	−8.87225454838900542e−11
	–	-O2/-O3	3.52986972732165553e−11	3.16607185000061602e−11	3.16607185000061602e−11
	–	-Os	−1.73673255207761718e−10	−1.73673255153551610e−10	−1.73673255153551610e−10
	–	-Ofast	3.52986972732165553e−11	3.16607185000061602e−11	3.16607185000061602e−11
Remark: When the Optimization(1) is -fp-model fast/-fp-model fast/-fp-model precise/-fp-model strict, or Optimization(2) is -O0, and the result is the same as the control group.					

Table 3
Reproducibility of Algorithm 5 using different compilers and optimization options on different platforms.

Size n	MLP_rsum	MLP_rsum after shuffle	error
36101600	8.89003676494079E−15	8.89003676494079E−15	0
15414000	6.64174217604646E−14	6.64174217604646E−14	0
10779808	7.36835249154126E−14	7.36835249154126E−14	0
37772000	4.51502674927232E−14	4.51502674927232E−14	0
55527408	1.35797156754243E−13	1.35797156754243E−13	0

Table 4
Accuracy of Algorithm 5 using different compilers and optimization options on different platforms.

Size n	MLP_rsum	MPRF	Theoretical error(1)	Experiment error
43162624	1.493127180107219e−17	1.785164287985800e−17	1.6694E−15	1.9722E−31
31782624	5.34832577991188E−14	5.34832577991188E−14	1.3145e−17	6.3109E−30
35504224	−3.05667390488188E−14	−3.05667390488188E−14	1.4684e−17	6.3109E−30
78838528	5.49285697840650E−14	5.49285697840650E−14	3.2607e−17	6.3109E−30
47467200	9.47310100821296E−14	9.47310100821295E−14	1.9632e−17	1.2622E−29

Table 5
Parameters of platforms.

		FT2000+	Kunpeng920	ThunderX2
Clock Frequency		2.2 GHz	2.6 GHz	2.5 GHz
Number of Core		64(1CPU)	64(1CPU)	32(2CPU)
Cache	L1I	48 KB	64 KB	32 KB
	L1D	32 KB	64 KB	32 KB
	L2	32 MB	512 KB	256 KB
	L3	–	64 MB	32 MB
Size of Memory		16*8 GB	16*32 GB	16*32 GB
Channel of Memory		DDR4 2400 MHz	DDR4 2933 MHz	DDR4 2666 MHz

Table 6
Speedup of MLP_rsum on ThunderX2 platform.

nT	nP						
	1	4	8	12	16	32	64
1	1.0000	3.8407	6.6930	9.2094	12.4398	19.4527	11.7829
2	1.9995	7.2414	12.8435	13.1553	14.8194	30.0535	7.5110
4	3.9534	13.8712	13.9892	18.8877	32.2438	22.7265	5.0684
8	7.6936	22.8247	21.2572	25.3891	19.0683	15.1905	3.8574
12	11.1447	25.5128	24.1698	15.6007	19.9494	12.0491	3.5295
16	14.3464	33.0389	17.5050	17.7051	12.5788	7.7170	3.2625
32	19.9920	21.1391	11.3428	11.7331	7.6456	5.1237	2.0058
64	12.4437	12.8403	6.3675	5.9216	4.4646	3.2423	0.9915

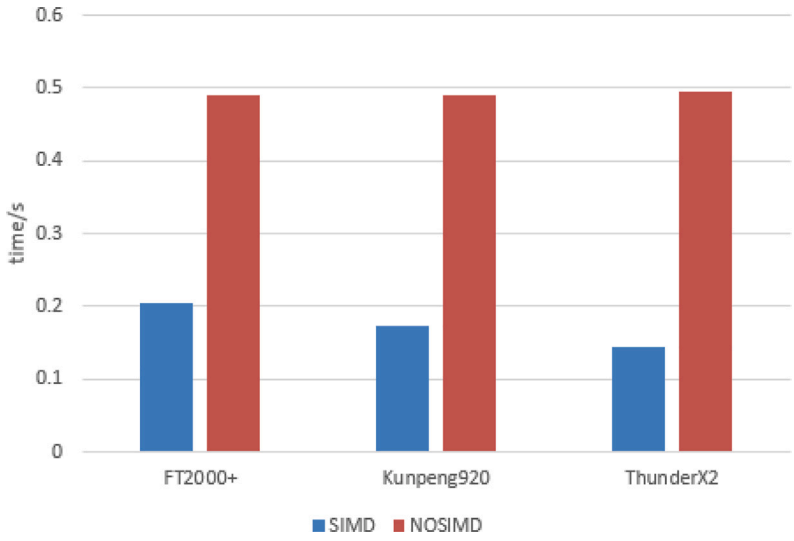

Fig. 1. Performance with SIMD optimization.

Table 7
Parallel efficiency of MLP_rsum in large-scale test.

nT	nP					
	1	8	16	32	64	128
1	1.0000	0.8483	0.8529	0.9579	0.7827	0.7595
2	0.9855	0.7941	0.7034	0.8406	0.6843	0.5854
4	0.9832	0.6196	0.8387	0.7608	0.5921	0.5138
8	0.7973	0.3725	0.5652	0.3127	0.2983	0.2171

test the strong scalability of the algorithm. When $nT < 4$, we find that the scalability of multi-thread is good, and the parallel efficiency does not change significantly. Since Algorithm 5 is bandwidth limited, when $nT > 4$, the competition of bandwidth is obvious, and the scalability is inconspicuous. The parallel efficiency is 51% at 128 nodes, which shows the effectiveness of the multi-level parallel design. As we know, in the existing literature, our experiment is the first one that runs a reproducible algorithm on a multi-node cluster.

5. Conclusions

In this paper, we propose a multi-level parallel multi-layer block reproducible summation algorithm, which is the extension of RepeoBLAS. Firstly, we adopt the intrinsic function supporting NEON to parallelize Demmel's algorithm on three ARM platforms, and enhance the spatial locality of data by embedding prefetch instructions. The experimental results testify that our algorithm is 2.41, 2.85, and 3.44 times faster than RepeoBLAS on the three platforms, respectively. Secondly, we achieve OpenMP parallel optimization under multi-core architecture. Specifically, we add special pragma to the source code so that the compiler can automatically parallelize the program. Single-chip multi-core parallel optimization experiment verifies the efficiency and linear speedup of our algorithm. As for multi-process parallel optimization, we use MPI for reduction and implement a large-scale node optimization experiment, which indicates that the parallel efficiency is 51% at 128 nodes. To sum up, the multi-level parallel design based on a multi-layer block structure is beneficial.

Declaration of competing interest

No author associated with this paper has disclosed any potential or pertinent conflicts which may be perceived to have impending conflict with this work. For full disclosure statements refer to https://doi.org/10.1016/j.parco.2023.102996.

Data availability

No data was used for the research described in the article.

References

[1] Roger D. Peng, Reproducible research in computational science, Science 334 (6060) (2011) 1226–1227.
[2] P. Ivie, T. Douglas, Reproducibility in scientific computing, ACM Comput. Surv. 51 (3) (2018) 1–36.
[3] Y. He, C.H.Q. Ding, Using accurate arithmetics to improve numerical reproducibility and stability in parallel applications, J. Supercomput. 18 (3) (2001) 259–277.
[4] M. McNutt, Reproducibility, Science 343 (6168) (2014) 229.
[5] D. Chapp, T. Johnston, M. Becchi, M. Taufer, Numerical reproducibility challenges on extreme multi-threading gpus.
[6] V. Stodden, M. McNutt, D.H. Bailey, E. Deelman, Y. Gil, B. Hanson, M.A. Heroux, J.P.A. Ioannidis, M. Taufer, Enhancing reproducibility for computational methods, Science 354 (6317) (2016) 1240–1241.
[7] O. Villa, D. Chavarria-Miranda, V. Gurumoorthi, A. Márquez, S. Krishnamoorthy, Effects of floating-point non-associativity on numerical computations on massively multithreaded systems, in: Proceedings of Cray User Group Meeting, CUG, 2009, p. 3.
[8] J. Demmel, H.D. Nguyen, Fast reproducible floating-point summation, in: 2013 IEEE 21st Symposium on Computer Arithmetic, IEEE, 2013.
[9] J. Demmel, H.D. Nguyen, Parallel reproducible summation, IEEE Trans. Comput. 64 (7) (2015) 2060–2070.
[10] ReproBLAS: reproducible BLAS, 2018, [Online]. Available: http://bebop.cs.berkeley.edu/reproblas/.
[11] P. Ahrens, J. Demmel, H.D. Nguyen, Algorithms for efficient reproducible floating-point summation, ACM Trans. Math. Software 46 (2020) 49, 3 (2020) Article 22.
[12] J. Demmel, G. Gopalakrishnan, M. Heroux, W. Keyrouz, K. Sato, Reproducibility of high-performance codes and simulations: Tools, techniques, debugging, in: Proceedings of the SC 2015 Birds of a Feather Sessions, 2015.
[13] M. Taufer, O. Padron, P. Saponaro, S. Patel, Improving numerical reproducibility and stability in large-scale numerical simulations on GPUs, in: IEEE International Symposium on Parallel & Distributed Processing, IPDPS, 2010, pp. 1–9.
[14] D. Chapp, V. Stodden, M. Taufer, Building a vision for reproducibility in the cyberinfrastructure ecosystem: Leveraging community efforts, Supercomput. Front. Innov. 7 (1) (2020) 112–129.

[15] Intel, Intel oneAPI Math Kernel Library reference manual, 2020, [Online]. Available: https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onemkl.html.
[16] NVIDIA, NVIDIA cuBLAS, 2021, [Online]. Available:https://developer.nvidia.com/cublas.
[17] T. Ogita, S.M. Rump, S. Oishi, Accurate sum and dot product, SIAM J. Sci. Comput. 26 (6) (2005) 1955–1988.
[18] S.M. Rump, Ultimately fast accurate summation, SIAM J. Sci. Comput. 31 (5) (2009) 3466–3502.
[19] S.M. Rump, T. Ogita, S. Oishi, Fast high precision summation, Nonlinear Theory Appl. IEICE 1 (1) (2010) 2–24.
[20] A. Arteaga, O. Fuhrer, T. Hoefler, Designing bit-reproducible portable high-performance applications, in: 2014 IEEE 28th International Parallel and Distributed Processing Symposium, IEEE, 2014.
[21] T. Hoefler, Performance reproducibility in HPC - challenges and state-of-the-art, Invited talk, SC, 2015.
[22] ExBLAS. http://exblas.lib6.fr/index.php.
[23] R. Iakymchuk, S. Collange, D. Defour, S. Graillat, ExBLAS: Reproducible and accurate BLAS library, in: Proceedings of the SC 2015 Numerical Reproducibility at Exascale Workshops, NRE'15, 2015.
[24] R. Iakymchuk, S. Collange, D. Defour, S. Graillat, Reproducible and accurate matrix multiplication, in: Proceedings of the Conference on Scientific Computing, Computer Arithmetic, and Validated Numerics, SCAN'15, Springer, Cham, 2015, pp. 126–137.
[25] R. Iakymchuk, S. Collange, D. Defour, S. Graillat, Reproducible triangular solvers for high-performance computing, in: Proceedings of the International Conference on Information Technology - New Generations, ITNG'15, 2015, pp. 353–358.
[26] S. Collange, D. Defour, S. Graillat, R. Iakymchuk, Numerical reproducibility for the parallel reduction on multi-and many-core architectures, Parallel Comput. 49 (Nov. 2015) (2015) 83–97.
[27] S. Collange, D. Defour, S. Graillat, R. Iakymchuk, A reproducible accurate summation algorithm for high-performance computing, in: Proceedings of the SIAM Workshop on Exascale Applied Mathematics Challenges and Opportunities (EX14) held as part of the 2014 SIAM Annual Meeting. Chicago, Il, USA, July 6–11, 2014.
[28] D. Mukunoki, K. Ozaki, T. Ogita, T. Imamura, DGEMM using tensor cores, and its accurate and reproducible versions, in: P. Sadayappan, B.L. Chamberlain, G. Juckeland, H. Ltaief (Eds.), High Performance Computing - 35th International Conference, ISC 2020, Proceedings, in: Lecture Notes in Computer Science, Springer, 2020, pp. 230–248.
[29] C. Chohra, P. Langlois, D. Parello, Efficiency of reproducible level 1 BLAS, in: Proceedings of the Conference on Scientific Computing, Computer Arithmetic, and Validated Numerics, SCAN'15, Springer, Cham, 2015, pp. 99–108.
[30] C. Chohra, P. Langlois, D. Parello, Reproducible, accurately rounded and efficient BLAS, in: Proceedings of the Euro-Par Parallel Processing Workshops, Springer, Cham, 2016, pp. 609–620.
[31] A.M. Castaldo, R.C. Whaley, A.T. Chronopoulos, Reducing floating point error in dot product using the superblock family of algorithms, SIAM J. Sci. Comput. 31 (2) (2009) 1156–1174.
[32] L. Dagum, R. Menon, OpenMP: an industry standard API for shared-memory programming, IEEE Comput. Sci. Eng. 5 (1) (1998) 46–55.
[33] P. Zimmermann, Reliable computing with GNU MPFR, in: Mathematical Software –ICMS 2010, Springer Berlin Heidelberg, 2010, pp. 42–45.