Efficiency of Reproducible Level 1 BLAS

Chemseddine Chohra, Philippe Langlois, David Parello

Efficiency of Reproducible Level 1 BLAS

Chemseddine Chohra*, Philippe Langlois* and David Parello*

Univ. Perpignan Via Domitia, Digits, Architectures et Logiciels Informatiques, F-66860, Perpignan. Univ. Montpellier II, Laboratoire d'Informatique Robotique et de Microélectronique de Montpellier, UMR 5506, F-34095, Montpellier. CNRS.

Abstract. Numerical reproducibility failures appear in massively parallel floating-point computations. One way to guarantee the numerical reproducibility is to extend the IEEE-754 correct rounding to larger computing sequences, as for instance for the BLAS libraries. Is the overcost for numerical reproducibility acceptable in practice? We present solutions and experiments for the level 1 BLAS and we conclude about the efficiency of these reproducible routines.

1 Introduction

Numerical reproducibility is an open question for current high performance computing platforms. Dynamic scheduling and non-deterministic reduction on multithreaded systems affect the operation order. This leads to non-reproducible results because the floating-point addition is not associative. Numerical reproducibility is important for debugging and for validating results, particularly if legal agreements require the exact reproduction of the execution results. Failures have been reported in numerical simulation for energy science, dynamic weather forecasting, atomic or molecular dynamic, fluid dynamic — see entries in [7].

Solutions provided at the middleware level forbid the dynamic behavior and so impact the performances — see [10] for TBB, [13] for OpenMP or Intel MKL. Note that adding pragmas in the source code avoids memory alignment effects onto reproducibility. A first algorithmic solution has been recently proposed in [4]. Their summation algorithms, ReprodSum and FastReprodSum, guarantee the reproducibility independently from the computation order. They return about the same accuracy as the performance optimized algorithm only running a small constant times slower.

Correctly rounded results ensure numerical reproducibility. IEEE754-2008 floating-point arithmetic is correctly rounded in its four rounding modes [5]. We propose to extend this property to the level 1 routines of the BLAS that depend on the summation order: asum, dot and nrm2, respectively the sum of the absolute values, the dot product and the vectorial Euclidean norm. Recent algorithms that compute the correctly rounded sum of n floating-point values allow us to implement such reproducible parallel computation. The main issue is

* firstname.lastname@univ-perp.fr

to investigate whether the running-time overhead of these reproducible routines remains reasonable enough in practice. In this paper we present experimental answers to this question. Our experimental framework is significant of the current computing practice: it consists in a shared memory parallel system with several sockets of multicore x86 processing units. We apply standard optimization techniques to implement efficient sequential and parallel level 1 routines. We show that for large vectors, reproducible and accurate routines introduces almost no overhead compared to their original counterparts in a performance-optimized library (Intel MKL). For shorter ones, reasonable overheads are measured and presented in Table 4.1. Since Level 1 BLAS performance is mainly dominated by the memory transfers, additional computation does not significantly increase the running time, especially for large vectors.

The paper is organized as follows. In Section 2, we briefly present some accurate summation algorithms, their optimizations and an experimental performance analysis to decide how to efficiently implement the level 1 BLAS. The experimental framework used throughout the paper is also described in this part. Section 3 is devoted to the performance analysis of the sequential implementation of the level 1 BLAS routines. Section 4 describes their parallel implementations and the measure of their efficiency. We conclude describing the future developments of this ongoing project towards efficient and reproducible BLAS.

2 Choice of Optimized Floating-Point Summation

Level 1 BLAS subroutines mainly rely on floating-point sums. It exists several correctly rounded summation algorithms. Our first step aims to derive optimized implementations of such algorithms and to choose the most efficient ones. In the following, we briefly describe these accurate algorithms and then, how to optimize and to compare them.

All floating-point computations satisfy the IEEE754-2008. Let $fl(\sum p_i)$ be the computed sum of a length n floating-point vector p. The relative error of the classical accumulation is of the order of $u \cdot n \cdot cond(\sum p_i)$, where $cond(\sum p_i) = \sum |p_i| / |\sum p_i|$ is the condition number of the sum. u is the machine precision that equals 2^{-53} for IEEE754 binary64.

2.1 Some Accurate or Reproducible Summation Algorithms

Algorithm SumK [9] reduces the previous relative error bound as if the classical accumulation is performed in K times the working precision:

$$\frac{|SumK(p) - \sum p_i|}{|\sum p_i|} \leq \frac{(n \cdot u)^K}{1 - (n \cdot u)^K} \cdot cond(\sum P_i) + u. \tag{2.1}$$

SumK replaces the floating-point add by Knuth's TwoSum algorithm that computes both the sum and its rounding error [8]. SumK iteratively accumulates

these rounding errors to enhance the final result accuracy. The correct rounding could be achieved by choosing a large enough K to vanish the effect of the condition number in (2.1) — but in practice this latter is usually unknown.

Algorithm iFastSum [16] repeats SumK to error-free transform the entry vector. This distillation process terminates returning a correctly rounded result thanks to a dynamic control of the error.

Algorithms AccSum [12] and FastAccSum [11] also rely on error-free transformations of the entry vector. They split the summands, relatively to $\max|p_i|$ and n, such that their higher order parts are then exactly accumulated. This split-and-accumulate steps are iterated to enhance the accuracy up to return a faithfully rounded sum. These algorithms return the correctly rounded sum and FastAccSum requires 25% less floating-point operations than AccSum.

HybridSum [16] and OnlineExact sum [17] exploit the short range of the floating-point number exponents. These algorithms accumulate the summands with a same exponent in a specific way to produce a short vector with no rounding error. The length of the output vector of this error-free transform step is the exponent range. HybridSum splits the summands such that floating-point numbers can be used as error-free accumulators. OnlineExact uses two floating-point numbers to simulate a double length accumulator. These algorithms then apply iFastSum to evaluate the correctly rounded sum of the error-free short vector(s).

ReprodSum and FastReprodSum [4] respectively rely on AccSum and FastAccSum to compute not fully accurate but reproducible sums independently of the summation order. So numerical reproducibility of parallel sums is ensured for every number of computing units.

2.2 Experimental Framework

Table 2.1 describes our experimental framework. Aggressive compiler options as `-ffast-math` are disabled to prevent the modification of the sensitive floating-point properties of these algorithms. Rounding intermediate results to the binary64 format (53 bit mantissa) and value safe optimizations are provided with `-fp-model double` and `-fp-model strict` options. Runtimes are measured in cycles with the hardware counters thanks to the `RDTSC` assembly instruction. We display the minimum cycle measures over more than fifty runs for each data. Condition dependant data are computed with the dot product generator from [9]. We compare our solutions to the well optimized but non-reproducible MKL BLAS implementation [6].

2.3 Implementation, Optimization and Test

For a fair comparison, all algorithms are manually optimized by a best effort process. AVX vectorization, data prefetching and loop unrolling are carefully

Table 2.1: Experimental framework

Software	
Compiler	ICC 14.0.2
Options	`-O3 -axCORE-AVX-I -fp-model double -fp-model strict -funroll-all-loops`
Parallel library	OpenMP 4.0
BLAS library	Intel MKL 11
Hardware	
Processor	Xeon E5 2660 (Sandy Bridge) at 2.2 GHz
Cache	L1: 32KB, L2: 256KB, shared L3 for each socket: 20MB
Bandwidth	51.2 GB/s
#cores	2×8 cores (hyper-threading disabled)

combined to pull out the best implementation of each algorithm. Moreover, some tricks improve the exponent based table access in HybridSum and OnlineExact, see for instance Alg. 2.1 for the latter and [1] for details.

1: Declare arrays $C1$ and $C2$
2: **for** i in 1:n **do**
3: $exp = \text{exponent}(p_i)$
4: $\text{FastTwoSum}(C1_{exp}, p_i, C1_{exp}, error)$
5: $C2_{exp} = C2_{exp} + error$
6: **end for**
7: $S = \text{iFastSum}(C1 \cup C2)$
8: **return** S

(a) Before optimization

1: Declare array C
2: Declare *distance* of prefetch
3: **for** i in 1:n (Manually unrolled) **do**
4: $\text{prefetch}(p_{i+distance})$;
5: $exp = \text{exponent}(p_i)$
6: $\text{FastTwoSum}(C_{2exp}, p_i, C_{2exp}, error)$
7: $C_{2exp+1} = C_{2exp+1} + error$
8: **end for**
9: $S = \text{iFastSum}(C)$
10: **return** S

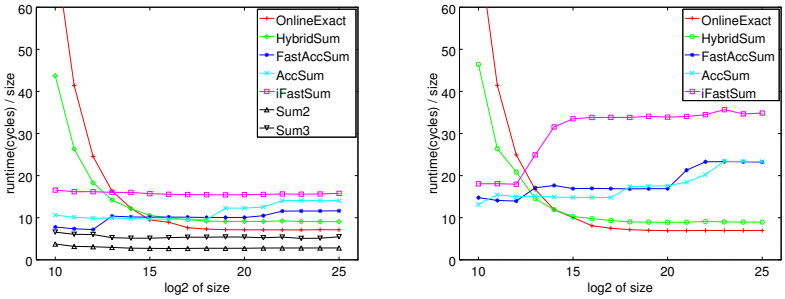
(b) After optimization

Alg. 2.1: Optimization of algorithm $\text{OnlineExact}(p, n)$

Figures 2.2a and 2.2b present the runtime measured in cycles divided by the vector size (y-axis). Vector lengths vary between 2^{10} and 2^{25} (x-axis) and two condition numbers are considered : 10^8 and 10^{32}.

It is not a surprise that HybridSum and OnlineExact are interesting for larger size vectors. These algorithms produce one or two short vectors (length = 2048 in binary64) whose distillation is of constant time compared to the linear times of the data preprocessing step (exponent extraction) or also, of the successive error free transformations in the other algorithms. Moreover they are very less sensitive to the conditionning of the entry vector. Shorter size vectors benefit from the other algorithms, especially from FastAccSum while their conditionning remains small.

In the following we take advantage of these different behaviors according to the size of the entry vector. We call it a “mixed solution”. In practice for the



(a) Condition number = 10^8 (b) Condition number = 10^{32}

Fig. 2.2: Runtime/size for optimized summation algorithms

level 1 BLAS routines, FastAccSum or iFastSum are useful for short vectors while larger ones benefit from HybridSum or OnlineExact as we will explain it.

3 Sequential Level 1 BLAS

Now we focus on the sum of the absolute value vector (asum), the dot product (dot) and the 2-norm (nrm2). Note that other level 1 BLAS subroutines do not suffer neither of accuracy nor of reproducibility failures. In this section, we start with sequential algorithms detailling our implementations and their efficiency.

3.1 Sum of Absolute Values

The condition number of asum equals 1. So SumK is enough to efficiently get a correctly rounded result. According to (2.1), K is chosen such that $n \leq u^{1/K-1}$.

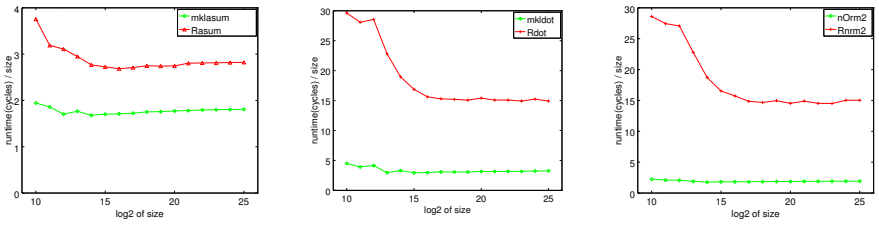
Figure 3.1a exhibits that the correctly rounded asum costs less than $2\times$ the optimized MKL dasum. Indeed $K = 2$ applies for the considered sizes. Note that $K = 3$ is enough until $n \leq 2^{35}$, *i.e.* until 256 Terabyte of data.

3.2 Dot Product

The dot product of two n-vectors is transformed into a sum of a $2n$-vector with Dekker's TwoProd [3]. This sum is correctly rounded using a "mixed solution". Short vectors are correctly rounded with FastAccSum. For large n, we avoid to build and read this intermediate $2n$-vector: the two TwoProd results are directly exponent-driven accumulated into the short vectors of OnlineExact. This explains why this latter is interesting for shorter dot products than what we can expect from Section 2.3.

Figure 3.1b shows this runtime divided by the input vector size — the condition number is 10^{32}. Despite the previous optimizations, the overcost ratio compared to MKL dot is between 3 and 6. This is essentially justified by the

additional computations (memory transfers are unchanged). If a fused-multiply-and-add unit (FMA) is available, the 2MultFMA algorithm [8] that only costs 2 FMA (compared to the TwoProd's 17 flop) certainly improves these values.



(a) asum (b) dot (c) nrm2

Fig. 3.1: Runtime/size for sequential asum, dot and nrm2.

3.3 Euclidean Norm

It is not difficult to implement an efficient and reproducible Euclidean norm. Reproducibility is ensured by the correct rounding of the sum of the squares and then by the correct rounding of the IEEE-754 square root. Of course this reproducible 2-norm is only faithfully rounded. Hence a "mixed solution" is similar to the dot one.

Here the MKL nrm2 is not used as the comparison reference since we measure very desapointing runtimes for it. We implement a non-reproducible simple and efficient 2-norm with the optimized MKL dot (cblas_ddot). We named it nOrm2.

The memory transfer cost dominates the computing one for dot and nOrm2: compared to dot, nOrm2 halves the memory tranfer volume, performs the same number of floating-point operations and runs twice faster, see Figures 3.1b and 3.1c. As previously mentionned, the "mixed solution" dot product is still computation-dominated. This justifies that the previous dot ratios prohibitively double for our sequential nrm2.

4 Reproducible Parallel Level 1 BLAS

Now we consider the parallel implementations. As in the previous section, parallel asum relies on parallel SumK while parallel dot and nrm2 derive from a parallel version of a "mixed solution" for the dot product. We start introducing these two parallel algorithms. Then we derive the parallel reproducible level 1 BLAS and perform its performance analysis.

4.1 From Parallel Sums to Reproducible Level 1 BLAS

Parallel SumK. It derives from the sequential version and has already been introduced in [15]. It consists in 2 steps. Step 1 applies the SumK algorithm on the local data without the final error compensation for every K iterations. Hence it returns a K-length vector S such that $(S_j)_{j=1,K}$ is the sum of the j^{th} layer in SumK applied to the local subvector. Step 2 gathers these K-length vectors to the master unit and applies the sequential SumK.

Parallel dot "mixed solution". Every n-length entry vector is splitted within P threads (or computing units) and N denotes the length of these local subvectors. The key point is to perform efficient error-free transformations of these N-vectors until the last reduction step. This consists in a 4 step process presented with Fig. 4.1 for $P = 2$. Steps 1 and 2 are processed by the P threads with local private vectors. Step 1 is similar to the sequential case and produces one vector of $size = 2N$ or 2048 or 4096: TwoProd transforms short N-vectors into a $2N$-one while this latter is not built for larger entries but directly exponent-driven accumulated into the $size$-length vector as for HybridSum or OnlineExact. Step 2: the $size$-length vector is distillated (as for iFastSum) into a smaller vector of non overlapping floating-point numbers. Step 3: every thread fuses this small vector into a global shared one. Step 4 is performed by the master thread that computes the correctly rounded result of the global vector with FastAccSum.

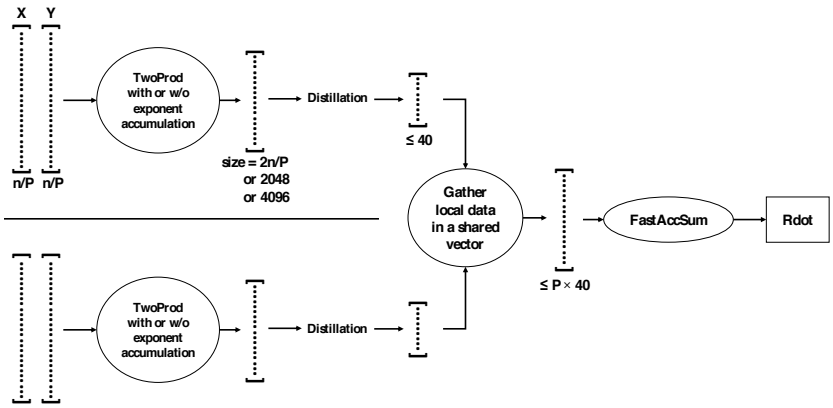


Fig. 4.1: Parallel dot "mixed solution"

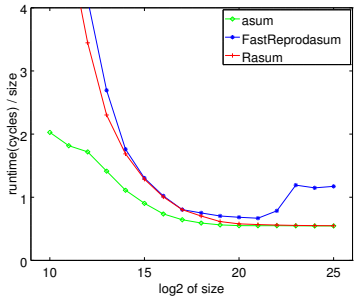
Let us remark that the small vector issued from Step 2 is at most of length 40 in binary64. Hence the distillation certainly benefits from cache effect. The next fusing step moves accross the computing units these vectors of length 40 in the worst case. This induces a communication over-cost especially for distributed memory environments. Nevertheless it introduces no more reduction step than a classic parallel summation.

The Reproducible Parallel Level 1 BLAS. The reproducible parallel Rasum derives from parallel SumK as in Sect. 3.1. The parallel dot "mixed solution" gives reproducible parallel Rdot and Rnrm2. In practice, the parallel implementation of the Step 1 differs from the sequential one as follows.

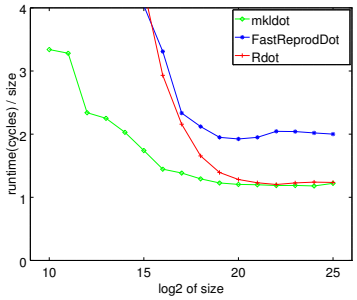
For shorter vectors, iFastSum is prefered to FastAccSum to minimize the Step 3 communications. For medium sized vectors, HybridSum is prefered to OnlineExact for Rdot to minimize the Step 2 distillation cost. Otherwise OnlineExact is chosen to minimize the exponent extraction cost.

4.2 Test and Results

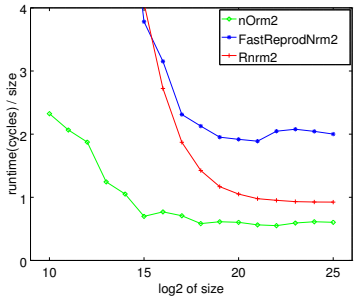
The experimental framework is unchanged. Each physical core runs at most one thread thanks to the `KMP_AFFINITY` variable. For every routine, we run from 1 to 16 threads on 16 cores to select the most efficient configurations with respect to the vector size. This optimal number of threads is given in parentheses in Table 4.1 except when it corresponds to the maximum possible ressources (16). Intel MKL's (hidden) choice is denoted with a $\star$.



(a) asum



(b) dot



(c) nrm2

Fig. 4.2: Runtime/size of parallel level 1 BLAS (up to 16 threads, cond=10^{32})

For the next performance comparisons, optimized parallel routines are necessary as references. We use the MKL parallel dot and we implement asum and

Table 4.1: Runtime overcost for the reproducibility of parallel level 1 BLAS

Vector size	10^3		10^4		10^5	10^6	10^7
Rasum/asum	2.0	(1/1)	1.5	(4/2)	1.3	1.1	1
Rdot/mkldot	6.4	(8/⋆)	3.8	(8/⋆)	1.6	1.1	1
Rnrm2/nOrm2	9.1	(8/⋆)	7.1	(8/⋆)	3.4	1.6	1.5
Rasum/FastReprodasum	0.9	(1/1)	0.9	(4/4)	1.0	0.8	0.5
Rdot/FastReprodDot	1.5	(8/1)	1.5	(8/8)	0.9	0.7	0.6
Rnrm2/FastReprodNrm2	1.7	(8/1)	1.5	(8/8)	0.9	0.5	0.4

nrm2 parallel versions. Our parallel asum runs up to 16 MKL dasum and performs a final reduction. Our parallel nOrm2 derives similarly from the sequential nOrm2 introduced in Sect. 3.3. These implementations exhibit the best performances in Fig. 4.2. As in Section 2.3, our implementations of ReprodSum and FastReprodSum are optimized in a fair way using again AVX vectorization, data prefetching and loop unrolling. The latter one is selected for the sequel.

We compare our reproducible Rasum, Rdot and Rnrm2, to the optimized but non-reproducible reference implementations, and to the one derived from FastReprodSum. Fig. 4.2 and Table 4.1 present these results.

Our reproducible Rasum compares very well to the optimized asum: the initial 2× overcost tends to 1 for n about 10^6, see Fig. 4.2a. Compared to the sequential cases and since it operates now on 16× smaller local vectors, our reproducible Rdot and Rnrm2 reach their optimal linear performance for larger entry sizes. Nevertheless the reproducible Rdot runs less than 2× slowler than the MKL reference for vector size up to 10^5, see Fig. 4.2b. For the same reasons as in the sequential case (Sect. 3.3), our reproducible Rnrm2 is not enough efficient to exhibit the same optimal tendency. Nevertheless the Rnmr2 overhead now reduces to the more convincing ratios compared to nOrm2, see Fig. 4.2c.

Finally our fully accurate reproducible level 1 routines compare quite favourably to those derived from the reproducible FastReprodSum, especially for large vectors: see Fig. 4.2. Those lastest algorithms read twice the entry vector and thus suffer from cache effects for large vectors. It is not the case for our algorithms. On the other hand, the additional computation required by OnlineExact or HybridSum benefit from the floating-point unit availability.

5 Conclusion and Future Developments

This experimental work illustrates that reproducible level 1 BLAS can be implemented with a reasonable overcost compare to the performance-optimized non-reproducible routines. Moreover our implementations offer full accuracy almost for free compared to the existing reproducible solutions.

Indeed the floating-point peak performance of current machines is far to be exploited by level 1 BLAS. So the additional floating-point operations required by our accuracy enhancement do not significantly increase their execution time.

Of course these results are quantitatively linked to the experimental framework. Nevertheless the same tendencies should be observed in other current computing contexts. Work is ongoing to benefit from FMA within dot and nrm2, to validate an hybrid OpenMP+MPI implementation on larger HPC cluster, to port and optimize this approach to accelerators (as Intel Xeon Phi) and to compare it to the expansions and software long accumulator of [2].

Finally there is alas no reason to be optimistic for the BLAS level 3 where the floating-point units have no space left for extra computation. Reproducible solutions need to be implemented from scratch, for example following [14].

References

1. Chohra, C., Langlois, P., Parello, D.: Level 1 Parallel RTN-BLAS: Implementation and Efficiency Analysis. In: SCAN'2014. Wurzburg, Germany (Sep 2014), `http://hal-lirmm.ccsd.cnrs.fr/lirmm-01095172`
2. Collange, S., Defour, D., Graillat, S., Iakimchuk, R.: Reproducible and Accurate Matrix Multiplication in ExBLAS for High-Performance Computing. In: SCAN'2014. Wurzburg, Germany (Sep 2014)
3. Dekker, T.J.: A floating-point technique for extending the available precision. Numer. Math. 18, 224–242 (1971)
4. Demmel, J.W., Nguyen, H.D.: Fast reproducible floating-point summation. In: Proc. 21th IEEE Symposium on Computer Arithmetic. Austin, Texas, USA (2013)
5. IEEE Task P754: IEEE 754-2008, Standard for Floating-Point Arithmetic. Institute of Electrical and Electronics Engineers, New York (Aug 2008)
6. Intel Math Kernel Library, `http://www.intel.com/software/products/mkl/`
7. Jézéquel, F., Langlois, P., Revol, N.: First steps towards more numerical reproducibility. ESAIM: Proceedings 45, 229–238 (Oct 2013), `http://hal-lirmm.ccsd.cnrs.fr/lirmm-00872562`
8. Muller, J.M., Brisebarre, N., de Dinechin, F., Jeannerod, C.P., Lefèvre, V., Melquiond, G., Revol, N., Stehlé, D., Torres, S.: Handbook of Floating-Point Arithmetic. Birkhäuser Boston (2010)
9. Ogita, T., Rump, S.M., Oishi, S.: Accurate sum and dot product. SIAM J. Sci. Comput. 26(6), 1955–1988 (2005)
10. Reinders, J.: Intel Threading Building Blocks. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edn. (2007)
11. Rump, S.M.: Ultimately fast accurate summation. SIAM J. Sci. Comput. 31(5), 3466–3502 (2009)
12. Rump, S.M., Ogita, T., Oishi, S.: Accurate floating-point summation – part I: Faithful rounding. SIAM J. Sci. Comput. 31(1), 189–224 (2008)
13. Story, S.: Numerical reproducibility in the Intel Math Kernel Library. Salt Lake City (Nov 2012)
14. Van Zee, F.G., Van De Geijn, R.A.: BLIS: A Framework for Rapidly Instanciating BLAS Functionality. ACM Trans. Math. Software (2015), to appear
15. Yamanaka, N., Ogita, T., Rump, S., Oishi, S.: A parallel algorithm for accurate dot product. Parallel Comput. 34(6–8), 392 – 410 (2008)
16. Zhu, Y.K., Hayes, W.B.: Correct rounding and hybrid approach to exact floating-point summation. SIAM J. Sci. Comput. 31(4), 2981–3001 (2009)
17. Zhu, Y.K., Hayes, W.B.: Algorithm 908: Online exact summation of floating-point streams. ACM Trans. Math. Software 37(3), 37:1–37:13 (2010)