Implementation and Efficiency of Reproducible Level 1 BLAS

Chemseddine Chohra, Philippe Langlois, David Parello

Implementation and 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 this reproducibility is to extend the IEEE-754 correct rounding to larger computing sequences, *e.g.* to the BLAS. Is the extra cost for numerical reproducibility acceptable in practice? We present solutions and experiments for the level 1 BLAS. We detail optimized implementations and we conclude about their efficiency.

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 [6].

Solutions provided at the middleware level forbid the dynamic behavior and so impact the performances — see [9] 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 [3]. 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 [4]. 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

* firstname.lastname@univ-perp.fr

allow us to implement such reproducible parallel computation. The main issue is 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 [5]). 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 [8] 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 [7]. *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 [3] 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.

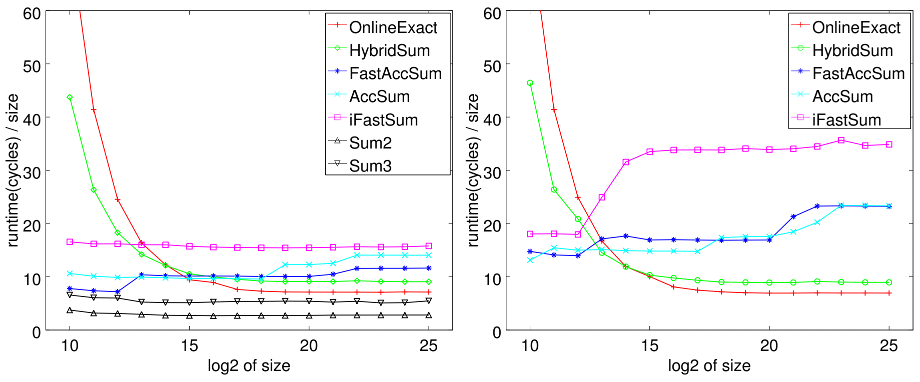
Rounding intermediate results to the binary64 format (53 bit mantissa) and value safe optimizations are provided with `-fp-model double` and `-fp-model strict` options. Runtimes are measured in cycles with the hardware counters thanks to the `RDTSC` assembly instruction. We display the minimum cycle measures over more than fifty runs for each data. Condition dependant data are computed with the dot product generator from [8]. We compare our solutions to the well optimized but non-reproducible MKL BLAS implementation [5].

Table 2.1: Experimental framework

Software	
Compiler (language)	ICC 14.0.2 (C)
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 $\times$ 8 cores (hyper-threading disabled)

2.3 Implementation and Test

For a fair comparison, all algorithms are manually optimized by a best effort process. All optimization details are presented in Appendix A. The source code for all presented algorithms is downladable on [10]. AVX vectorization, data prefetching and loop unrolling are carefully combined to pull out the best implementation of each algorithm.



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

Fig. 2.1: Runtime/size for optimized summation algorithms

Figures 2.1a and 2.1b 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 conditioning of the entry vector. Shorter size vectors benefit from the other algorithms, especially from *FastAccSum* while their conditioning 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 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 detailing 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.

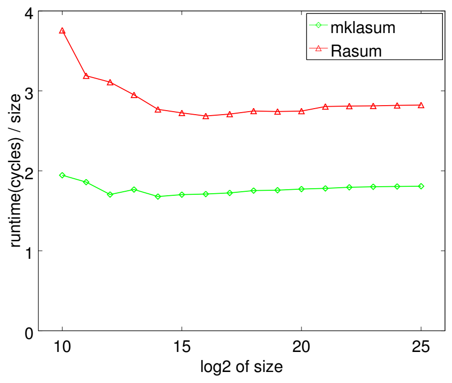
3.2 Dot Product

The dot product of two n-vectors is transformed into a sum of a $2n$-vector with Dekker’s TwoProd [2]. 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.

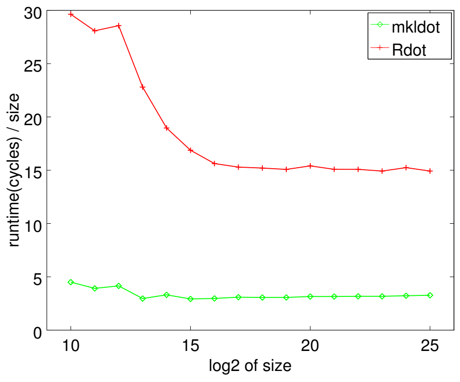
Figure 3.1b shows this runtime divided by the input vector size — the condition number is 10^{32}. Despite the previous optimizations, the extra cost ratio compared to MKL dot is between 3 and 6. This is essentially justified by the additional computations (memory transfers are unchanged). If a fused-multiply-and-add unit (FMA) is available, the 2MultFMA algorithm [7] that only costs 2 FMA (compared to the TwoProd’s 17 flop) certainly improves these values.

3.3 Euclidean Norm

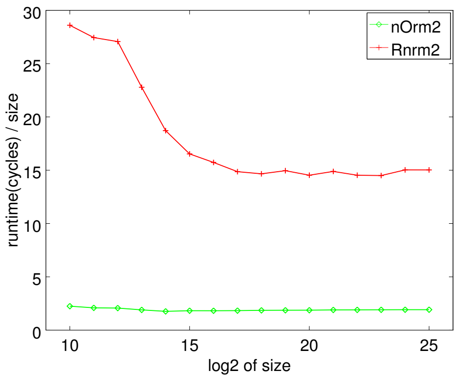
It is not difficult to implement an efficient and reproducible Euclidean norm. Reproducibility is ensured by the correct rounding of the sum of the squares



(a) asum



(b) dot



(c) nrm2

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

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 disappointing 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 transfer volume, performs the same number of floating-point operations and runs twice faster, see Figures 3.1b and 3.1c. As previously mentioned, 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 split 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 distilled (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*.

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

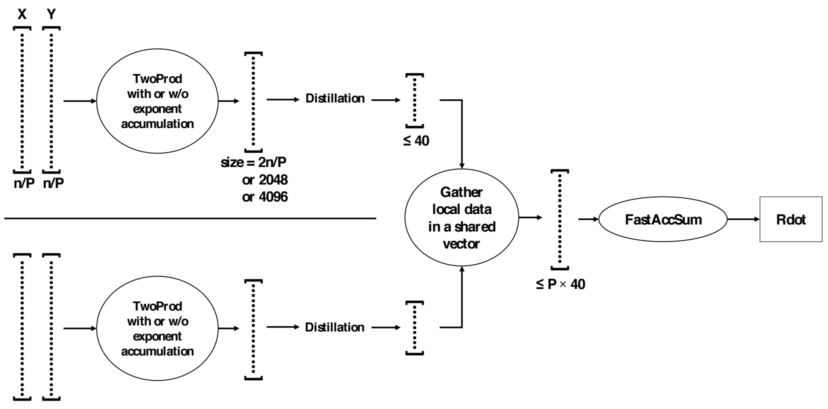

X
Y
n/P
n/P
TwoProd with or w/o exponent accumulation
size = 2n/P or 2048 or 4096
Distillation
≤ 40
Gather local data in a shared vector
≤ P × 40
FastAccSum
Rdot
TwoProd with or w/o exponent accumulation
Distillation

Fig. 4.1: Parallel dot "mixed solution"

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 resources (16). Intel MKL's (hidden) choice is denoted with a $\star$.

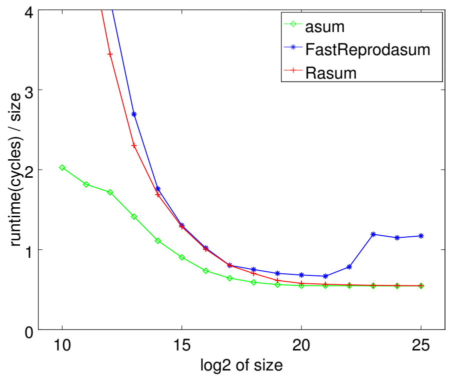
Table 4.1: Runtime extra cost for the reproducibility of parallel level 1 BLAS

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/$\star$)	3.8	(8/$\star$)	1.6	1.1	1
Rnrm2/nOrm2	9.1	(8/$\star$)	7.1	(8/$\star$)	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

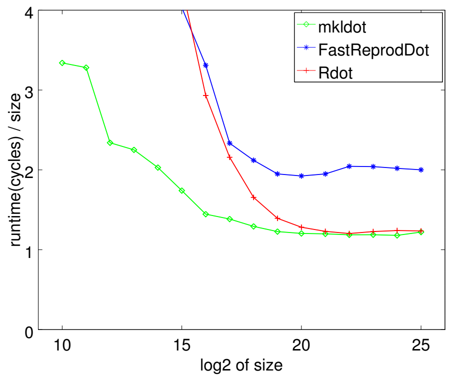
For the next performance comparisons, optimized parallel routines are necessary as references. We use the MKL parallel dot and we implement asum and nrm2 parallel versions. Our parallel asum runs up to 16 MKL dasum and performs a final reduction. Our parallel nOrm2 derives similarly from the sequential 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.

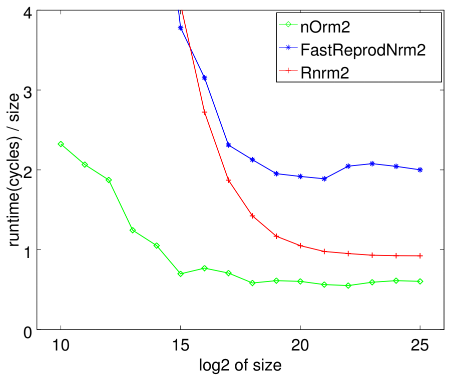
Our reproducible Rasum compares very well to the optimized asum: the initial $2\times$ extra cost tends to 1 for n about 10^6, see Fig. 4.2a. Compared to the sequential cases and since it operates now on $16\times$ smaller local vectors, our reproducible Rdot and Rnrm2 reach their optimal linear performance for larger entry sizes. Nevertheless the reproducible Rdot runs less than $2\times$ slower 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.



(a) asum



(b) dot



(c) nrm2

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

Finally our fully accurate reproducible level 1 routines compare quite favorably to those derived from the reproducible *FastReprodSum*, especially for large vectors: see Fig. 4.2. Those latest 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 extra cost 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 [1].

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].

Appendix A Optimization of summation algorithms

To guarantee best efficiency, all the algorithms that we have presented are optimized manually. For prefetching and vectorization we use assembly like intrinsics. Loops have been manually unrolled to reduce dependencies on loop incremental. We show next the different optimizations in details for each algorithm.

A.1 OnlineExact

We show in Alg. A.1 the difference between optimized and not optimized implementation of algorithm OnlineExact. On the optimized version side there are three changes. (1) Loops are manually unrolled. (2) Prefetching in the 4^{th} line reduce the cost of memory latency. The distance of prefetching has been selected according to experimental work. It could change on another test environment. Distance of prefetch depends mainly on memory latency and bandwidth. (3) The third difference is about data locality. Since the algorithm OnlineExact simulate a large accumulator using two floating point numbers. The original idea was to use two vectors, one for either part of accumulators. Those two vectors are indexed easily using the exponent of the entry. What we have done is to put the accumulator two floating point numbers successively in the same vector. That costs two extra integer operations to compute "2exp" and "2exp + 1". Although it guarantees that the accumulator holds always on the same cache-line. such optimization is very efficient in practice because memory latency has much more

```
1: Declare arrays $C1$ and $C2$
2: for i in 1:n do
3:     $exp$ = exponent($p_i$)
4:     FastTwoSum($C1_{exp}, p_i, C1_{exp}, error$)
5:     $C2_{exp} = C2_{exp} + error$
6: end for
7: $S$ = 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:     prefetch($p_{i+distance}$)
5:     $exp$ = exponent($p_i$)
6:     FastTwoSum($C_{2exp}, p_i, C_{2exp}, error$)
7:     $C_{2exp+1} = C_{2exp+1} + error$
8: end for
9: $S$ = iFastSum($C$)
10: return $S$
```

(b) After optimization

Alg. A.1: Optimization of algorithm OnlineExact(p, n)

extra cost compared to integer operations. Note also that even prefetching can not avoid latency cost in this case. Because we do not know the location of data before the previous instruction that gets exponent.

This algorithm has a very annoying drawback. Since we do not access to accumulator vector in regular order, hardware vectorization is impossible to employ. The 5th instruction which gets exponent value with a mask and shift operations can be easily vectorized. Anyways, according to experimental results this vectorization was not efficient, and have had even negative effect on performance. The reason of that is that those vector instruction are followed by scaler operations on the same data. Since vector operations use 256 bits YMM registers and scaler operations work only on "the first" 64 bits of the same registers. Compiler has to add shuffle or store and load operations to be able to apply scaler operations on these data. Those operations cost more cycles, and also create some data dependencies that give rise to poor instruction level parallelism.

If we have had more operations to vectorize, performance will be better. For instance when we implement an exact dot product which is based on OnlineExact, we vectorize the TwoProd operation. In case of two prod, even if there is scaler operations after vector operations, vectorization gain worth it. The reason is that it overlays the additional cost to apply scaler operations.

A.2 HybridSum

The optimization process for algorithm HybridSum is quite similar of that of OnlineExact. In 2nd, 3rd and 4th line in Algorithm A.2b, we unroll the loop and prefetch the data like for OnlineExact and for the same reasons. We see also in the 7th instruction that p_l is not accumulated according to its exponent. Although since the last non-zero bit of p_l should always be greater than or equal the last non-zero bit of C_{h-27}, we guarantee that the proof in the Section 3.2 of [16] still valid.

```
1: Declare array $C$
2: for i in 1:n do
3:     split($p_i, p_h, p_l$)
4:     $exp_h$ = exponent($p_h$)
5:     $exp_l$ = exponent($p_l$)
6:     $C_{exp_h} = C_{exp_h} + p_h$
7:     $C_{exp_l} = C_{exp_l} + p_l$
8: end for
9: $S$ = iFastSum($C$)
10: return $S$
```

(a) Before optimization

```
1: Declare array $C$
2: Declare $distance$ of prefetch
3: for i in 1:n (Manually unrolled) do
4:     prefetch($p_{i+distance}$)
5:     split($p_i, p_h, p_l$)
6:     $exp_h$ = exponent($p_h$)
7:     $exp_l$ = $exp_h$ - 27
8:     $C_{exp_h} = C_{exp_h} + p_h$
9:     $C_{exp_l} = C_{exp_l} + p_l$
10: end for
11: $S$ = iFastSum($C$)
12: return $S$
```

(b) After optimization

Alg. A.2: Optimization of algorithm HybridSum(p, n)

The major disadvantage of both algorithms HybridSum and OnlineExact is that we can not vectorize most operations. This will reduce the scalability of algorithms performance on new microarchitectures.

A.3 AccSum

```
M = Max(p)
$\sigma$ = NextPower2(n + 2) × NextPower2(M)
S = 0
repeat
    S = S + ExtractVector($\sigma$, p)
    $\sigma = \sigma$ × u × NextPower2(M)
until "Stop cretiria"
return $S$
```

Alg. A.3: Algorithm AccSum(p, n)

In Algorithm A.3 we show a simplified version of the algorithm AccSum (see [12] for full details). As we have explained in Section 2.1 AccSum split summands according to σ. Then the higher order parts are accumulated exactly. This work is done with the algorithm ExtractVector as we will show in Algorithm A.4.

The advantage of the algorithm ExtractVector is that all its operations can be vectorized. The operations that are performed to generate the vector q are fully independent. The accumulation of q_i can be vectorized since the summation of these high order parts is exact.

Theoretically, vectorization can give a multiplicative boost to the computation performance. Unfortunately for this algorithm the memory bandwidth limits performance. Especially that it pass through the vector multiple times to

```
1: t = 0
2: for i in 1:n do
3:     $q_i$ = ($\sigma$ + $p_i$) - $\sigma$
4:     $p_i$ = $p_i$ - $q_i$
5:     t = t + $q_i$
6: end for
7: return $t$
```

(a) Before optimization

```
1: t = 0
2: $t_v$[size] = {0}
3: $\sigma_v$[size] = {$\sigma$}
4: for i in 1:n do (unrolled by size)
5:     $q_{i:i+size}$ = ($\sigma_v$ + $p_{i:i+size}$) - $\sigma_v$
6:     $p_{i:i+size}$ = $p_{i:i+size}$ - $q_{i:i+size}$
7:     $t_v$ = $t_v$ + $q_{i:i+size}$
8: end for
9: t = ReduceSum($t_v$)
10: return t
```

(b) After optimization

Alg. A.4: Optimization of algorithm ExtractVector(σ, p)

enhance precision until the stop cretiria is verified. Prefetching was not efficient on the algorithm ExtractVector. A possible reason is that we read and write on the vector p in each iteration, while the software prefetching should be either for read, or for write.

Since algorithms FastAccSum, ReprodSum and FastReprodSum have similar properties, they are optimized in the same way. The difference is that prefetching for read was efficient on ReprodSum and FastReprodSum because there is no overwrite of the input vector.

A.4 SumK

```
1: S[K] = {0}
2: for k in 1:K-1 do
3:     for i in 1:n do
4:         (S[k], $p_i$) = TwoSum(S[k], $p_i$)
5:     end for
6: end for
7: for i in 1:n do
8:     S[K] = S[K] + $p_i$
9: end for
10: return $\sum_{k=1}^{K} S_k$
```

(a) Before optimization

```
1: S[K] = {0}
2: Declare *distance* of prefetch
3: for i in 1:n (manually unrolled) do
4:     prefetch($p_{i+distance}$);
5:     for k in 1:K-1 do
6:         (S[k], $p_i$) = TwoSum(S[k], $p_i$)
7:     end for
8:     S[K] = S[K] + $p_i$
9: end for
10: return $\sum_{k=1}^{K} S_k$
```

(b) After optimization

Alg. A.5: Optimization of algorithm SumK(p, n)

For the algorithm SumK, the most important optimization is the inversion of loops. Since we know the value of K a priori, we can accumulate summands for all K passes in one iteration. Also we can keep the original values of vector p

by using intermediate variable to write instead of p_i. This should make prefetching more efficient and reduce memory cost, because there is no need to write values on p. Similar optimization is applied for the algorithms ReprodSum and FastReprodSum.

Unfortunately we could not do the same with AccSum and FastAccSum because the number of iterations is not predefined.

Vectorization of SumK We vectorize the algorithm SumK using the same idea of the parallel version of SumK [15]. A brief description of this algorithm is given in Section 4.1. So in the end of vectorized part, we will get K vectors of size s (such that s is the size of hardware registers). Then we concatenate those vectors, and we apply non vectorized SumK on the result vector of size $K \times s$.

References

1. Collange, S., Defour, D., Graillat, S., Iakimchuk, R.: Reproducible and Accurate Matrix Multiplication in ExBLAS for High-Performance Computing. In: SCAN'2014. Würzburg, Germany (2014)
2. Dekker, T.J.: A floating-point technique for extending the available precision. Numer. Math. 18, 224–242 (1971)
3. Demmel, J.W., Nguyen, H.D.: Fast reproducible floating-point summation. In: Proc. 21th IEEE Symposium on Computer Arithmetic. Austin, Texas, USA (2013)
4. IEEE Task P754: IEEE 754-2008, Standard for Floating-Point Arithmetic. Institute of Electrical and Electronics Engineers, New York (Aug 2008)
5. Intel Math Kernel Library, `http://www.intel.com/software/products/mkl/`
6. Jézéquel, F., Langlois, P., Revol, N.: First steps towards more numerical reproducibility. ESAIM: Proceedings 45, 229–238 (2013), `http://hal-lirmm.ccsd.cnrs.fr/lirmm-00872562`
7. 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)
8. Ogita, T., Rump, S.M., Oishi, S.: Accurate sum and dot product. SIAM J. Sci. Comput. 26(6), 1955–1988 (2005)
9. Reinders, J.: Intel Threading Building Blocks. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edn. (2007)
10. `http://webdali.univ-perp.fr/ReproducibleSoftware`
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 instantiating BLAS functionality. ACM Transactions on Mathematical Software 41(3)
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)