Numerical Reproducibility for the Parallel Reduction on Multi- and Many-Core Architectures

Caroline Collange, David Defour, Stef Graillat, Roman Iakymchuk

Numerical Reproducibility for the Parallel Reduction on Multi- and Many-Core Architectures

Sylvain Collange[1], David Defour[2], Stef Graillat[3], and Roman Iakymchuk[3,4]

[1] INRIA – Centre de recherche Rennes – Bretagne Atlantique
Campus de Beaulieu, F-35042 Rennes Cedex, France
sylvain.collange@inria.fr
[2] DALI–LIRMM, Université de Perpignan, 52 avenue Paul Alduy, F-66860 Perpignan, France
david.defour@univ-perp.fr
[3] Sorbonne Universités, UPMC Univ Paris 06, UMR 7606, LIP6, F-75005 Paris, France
CNRS, UMR 7606, LIP6, F-75005 Paris, France
{stef.graillat, roman.iakymchuk}@lip6.fr
[4] Sorbonne Universités, UPMC Univ Paris 06, ICS, F-75005 Paris, France

Abstract. On modern multi-core, many-core, and heterogeneous architectures, floating-point computations, especially reductions, may become non-deterministic and, therefore, non-reproducible mainly due to the non-associativity of floating-point operations. We introduce an approach to compute the correctly rounded sums of large floating-point vectors accurately and efficiently, achieving deterministic results by construction. Our multi-level algorithm consists of two main stages: a filtering stage that relies on fast vectorized floating-point expansions, and an accumulation stage based on superaccumulators in a high-radix carry-save representation. We present implementations on recent Intel desktop and server processors, Intel Xeon Phi accelerators, and both AMD and NVIDIA GPUs. We show that numerical reproducibility and bit-perfect accuracy can be achieved at no additional cost for large sums that have dynamic ranges of up to 90 orders of magnitude by leveraging arithmetic units that are left underused by standard reduction algorithms.

Keywords: Parallel floating-point summation, Reproducibility, Accuracy, Kulisch long accumulator, Error-free transformations, Multi- and many-core architectures.

1 Introduction

The increasing power of current computers enables one to solve more and more complex problems. That leads to a higher number of floating-point operations to be performed. Each of these operations potentially causes a round-off error. Because of the round-off error propagation, some problems must be solved with a wider floating-point format. This is especially the case for applications that carry out very complicated and enormous tasks in scientific fields such as quantum field theory, supernova simulation, semiconductor physics, or planetary orbit calculations [1]. Since Exascale computing (10^{18} operations per second) is likely to be reached within a decade, getting accurate results in floating-point arithmetic on such computers is an open challenge.

The reproducibility of parallel reductions involving floating-point addition is becoming a serious issue, as noted in the DARPA Exascale Report [2]. Large-scale summations typically appear within fundamental numerical blocks such as a dot product or a numerical integration. As floating-point addition is not associative, the result of a summation may vary from one parallel machine to another or even from one run to another. These discrepancies worsen on heterogeneous architectures – such as clusters composed of standard CPUs in conjunction with GPUs and/or accelerators like Intel Xeon Phi – which combine together different programming environments that may follow different floating-point models and offer different intermediate precision or different operators [3,4]. For instance, Intel acknowledges in [5] that *"there is no way to ensure bit-for-bit reproducibility between code executed on Intel® Xeon processors and code executed on Intel® Xeon Phi™ co-processors, even for fixed number of threads or for serial code"*. Non-determinism of floating-point calculations in parallel programs causes validation and debugging issues, and may even lead to deadlocks [6]. We expect these problems will get increasingly critical as the trend towards large-scale heterogeneous platforms continues.

In this work, we aim at addressing both accuracy and reproducibility in the context of parallel summation. We advocate to compute the correctly rounded result of the exact sum. The correct rounding criterion guarantees a unique, well-defined answer, ensuring bit-wise reproducibility. In addition, the correctly-rounded result is also the most accurate answer possible in the given floating-point format.

Without dedicated hardware support, large-scale correctly rounded sums have been considered impractical since computing the exact sum in software was deemed to be too costly [7]. The current paper revisits this assumption. We show that: 1. The computation of the exact sum can be carried out at the affordable cost using a large fixed-point accumulator, which is named a *superaccumulator*[5]; 2. The overhead can be made negligible on large sums with low to moderate dynamic ranges using vectorized floating-point expansions. Besides offering the best possible accuracy of the result, our approach guarantees the strict reproducibility by always returning the correctly-rounded value of the exact result.

The paper is organized as follows. Section 2 describes main aspects of floating-point arithmetic and reviews floating-point expansions as well as superaccumulators. Section 3 presents our multi-level approach to superaccumulation. We expose in Section 4 various implementations and results on multi- and many-core architectures. Finally, we discuss related works and draw conclusions in Sections 5 and 6, respectively.

2 Floating-Point Arithmetic

Floating-point arithmetic consists in approximating real numbers with a significand, an exponent, and a sign :

$$x = \pm \underbrace{x_0.x_1 \dots x_{M-1}}_{mantissa} \times b^e, \quad 0 \leq x_i \leq b-1, \quad x_0 \neq 0$$

where b is the basis (2 in our case), M the precision, and e the exponent that is bounded ($e_{\min} \leq e \leq e_{\max}$).

The IEEE-754 standard [8], which was revised in 2008, specifies floating-point formats and operations. In this paper, we consider the `binary64` or double-precision format, although our strategy is applicable to the other formats as well. Floating-point representation of numbers allows to cover a wide *dynamic range*. Dynamic range is defined as the absolute ratio between the number with the largest magnitude and the number with the smallest non-zero magnitude in a set. For instance, `binary64` can represent positive numbers from 4.9×10^{-324} to 1.8×10^{308}, so it covers a dynamic range of 3.7×10^{631}.

Type	Size	Mantissa	Exponent	Unit rounding	Interval
`binary32`	32 bits	23+1 bits	8 bits	$u = 2^{1-24} \approx 1,92 \times 10^{-7}$	$\approx 10^{\pm 38}$
`binary64`	64 bits	52+1 bits	11 bits	$u = 2^{1-53} \approx 2,22 \times 10^{-16}$	$\approx 10^{\pm 308}$

Fig. 1: Main floating-point formats in IEEE-754.

The standard requires correctly rounded results for the basic arithmetic operations ($+, -, \times, /, \sqrt{}$). It means that the operations are performed as if the result was first computed with an infinite precision and then rounded to the floating-point format. Several rounding modes are provided. In this paper, we will assume the rounding-to-nearest mode. It means that an operation returns the closest floating-point number to the exact result, breaking ties by rounding to the floating-point number with the even significand.

The non-associativity of floating-point addition occurs due to rounding errors while performing addition of numbers with different exponents. It leads to the elimination of the lowest-order bits of the sum. For example, in `binary64`, $(-1 \oplus 1) \oplus 2^{-53} \neq -1 \oplus (1 \oplus 2^{-53})$ since $(-1 \oplus 1) \oplus 2^{-53} = 2^{-53}$ and $-1 \oplus (1 \oplus 2^{-53}) = 0$, where $\oplus$ stands for addition in floating-point arithmetic. In contrast, subtraction between numbers of the same sign and exponent is always exact. However, due to the cancellation, it amplifies the impact of previous errors. Thus, the accuracy of a floating-point summation depends on the order of evaluation. More detailed explanation can be found in the main references in floating-point arithmetic [9,10].

Two approaches exist to perform floating-point addition without incurring round-off errors. The first solution computes the error which occurred during rounding using floating-point expansions in conjunction with error-free transformations, see Section 2.1. The second solution exploits the finite range of representable floating-point numbers by storing every bit in a very long vector of bits, see Section 2.2.

[5] We use names Kulisch long accumulator, long accumulator, or superaccumulator interchangeably.

2.1 Floating-Point Expansion

In order to perform summations exactly, one has to recover errors which occurred while rounding and keep track of them. If a and b are two floating-point numbers, $r := a \oplus b$ is a floating-point number, but not $a + b$ as a general case. However, in round-to-nearest (which is the case in this paper), the rounding error $s := (a+b) - r = (a+b) - (a \oplus b)$ is a floating-point number and can be computed in floating-point arithmetic. Such an algorithm is called error-free transformation. The traditional error-free transformation for addition is `TwoSum` [11], depicted in Alg. 1. The `TwoSum` algorithm relies only on floating-point additions and subtractions.

Floating-point expansions represent the result as an unevaluated sum of floating-point numbers, whose components are ordered by decreasing magnitude with minimal overlap to cover a wide range of exponents. Floating-point expansions of size 2 and 4 are described in [12] and [13], accordingly. Adding one floating-point number to an expansion is an iterative operation: the floating-point number is first added to the head of the expansion and the rounding error is recovered as a floating-point number using an error-free transformation such as `TwoSum`. The error is then recursively accumulated to the remaining elements of the expansion.

Algorithm 1 Error-free transformation of the sum of two floating-point numbers.

Function$[r, s]$ = TwoSum(a, b)

1: $r \leftarrow a \oplus b$
2: $z \leftarrow r \ominus a$
3: $s \leftarrow (a \ominus (r \ominus z)) \oplus (b \ominus z)$

With expansions of size n – that correspond to the unevaluated sum of n floating-point numbers – it is possible to accumulate floating-point numbers without losing the accuracy as long as every intermediate result can be represented exactly as a sum of n floating-point numbers. This situation happens when the dynamic range of the sum is lower than $2^{53 \cdot n}$ in case of binary64.

The main advantage of this solution is that expansions only involve basic floating-point operations and no memory access, which allows very efficient implementations on modern architectures thanks to pipelining and SIMD units. On the one hand, the accuracy of small-size expansions is insufficient for the summation of numerous floating-point numbers or sums with a large dynamic range. On the other hand, large-size expansions are computationally inefficient as the complexity of the accumulation algorithm grows linearly with the size of the expansion.

2.2 Superaccumulator

An alternative algorithm to floating-point expansions is based on very long fixed-point accumulators. A fixed-point representation store numbers using an integral part and a fractional part of fixed size, or equivalently as a scaled integer. The length of the fixed-point accumulator is selected in such way that it can represent every bit of information of the input format (`binary64` in our case); this covers the range from the minimum representable floating-point value to the maximum value, independently of the sign. For instance, Kulisch [14] proposed to utilize an accumulator of 4288 bits to handle dot products of two vectors composed of `binary64` values. The summation is performed without loss of information by accumulating every floating-point input number in the long accumulator, see Fig. 2. The superaccumulator is the perfect solution to produce the exact result of a very large amount of floating-point numbers of arbitrary magnitude. However, for a long period this approach was considered impractical as it induces a very large memory overhead. Furthermore, without dedicated hardware support, its performance is limited by indirect memory accesses that makes vectorization challenging.

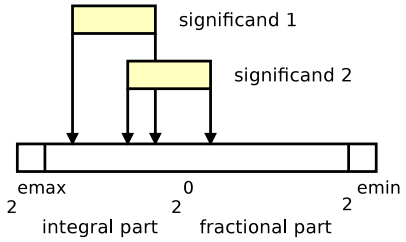


Fig. 2: Kulisch long accumulator.

3 Hierarchical Superaccumulation Scheme

Most prior works on exact summation have been targeting severely ill-conditioned problems that could not be satisfied using the accuracy provided by the conventional floating-point addition. Our objective is different – we are concerned with computing a deterministic and accuracy-guaranteed result for sums that are typically evaluated using today's floating-point arithmetic. We expect such sums to be well-conditioned or mildly ill-conditioned and cover a commensurate dynamic range (up to 10^{50}).

We follow a multi-level approach shown in Fig. 3. The accumulation of floating-point numbers is split into five levels. This decomposition suits well the memory hierarchy as well as the nested parallelism of modern architectures. In addition, such approach makes full use of SIMD, multi-thread, and multi-core parallelism.

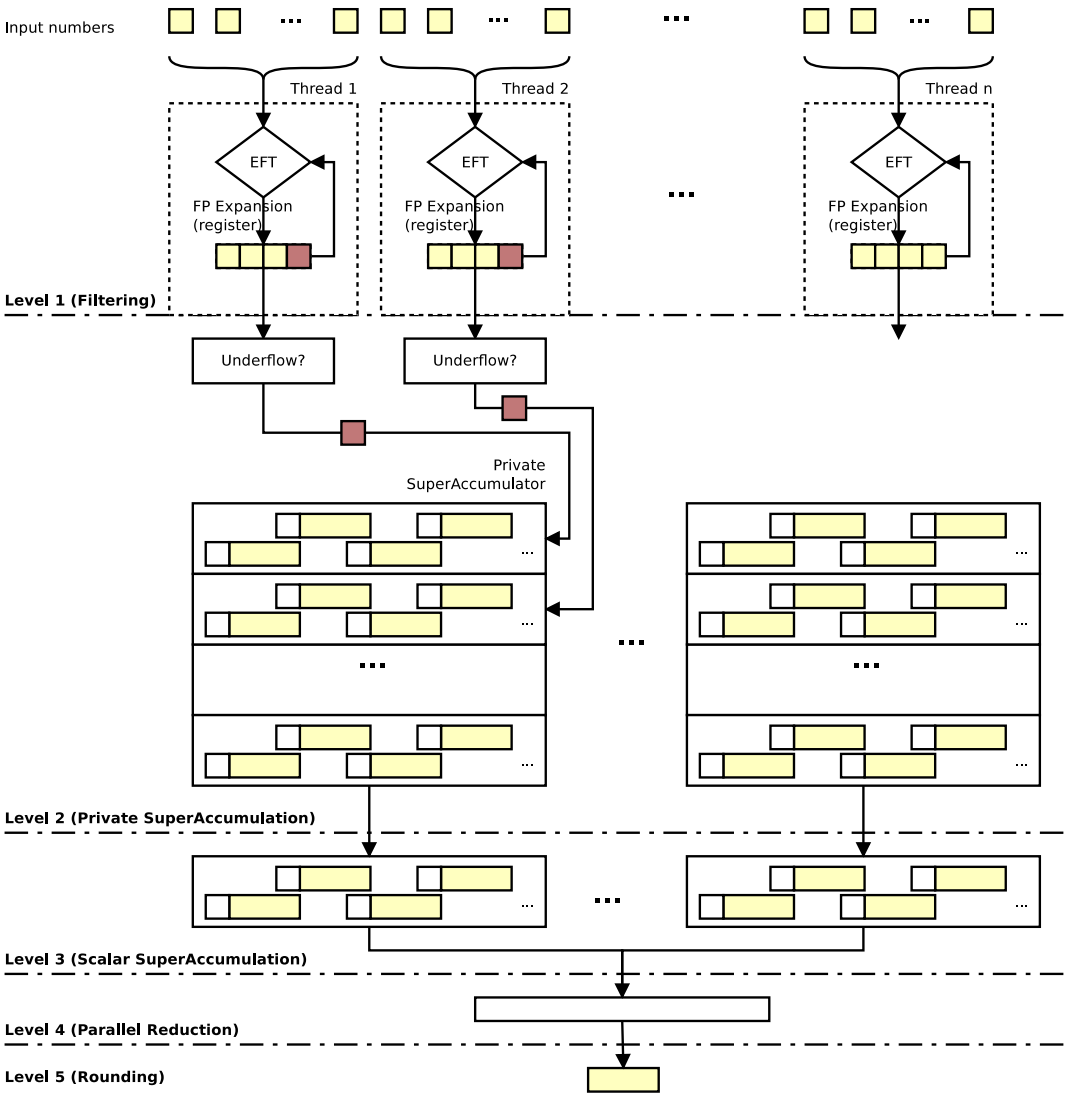


Fig. 3: Multi-level summation approach.

The first level consists of floating-point expansions in order to perform fast accumulation of consecutive floating-point numbers with similar magnitude. On CPUs and Xeon Phi accelerators, the expansions are vectorized to take advantage of SIMD units: a vector of input numbers is accumulated in parallel to a vector of expansions. On GPUs, each thread maintains its own expansion of size n. As tens to thousands of threads running in parallel using their private registers to store floating-point expansions, this solution ensures the high speed of computations due to the minimized memory accesses.

The second level is invoked only whenever the accuracy provided by expansions is not enough. Threads transmit to private superaccumulators values that are too small to be accumulated exactly in the expansions as well as the results of partial sums accumulated during the first step. Private superaccumulators are stored in a fast local memories like cache (CPU and Xeon Phi) or local memory (GPU). They contain exact partial sums for one thread or a group of threads, respectively.

In the third level, private superaccumulators within a group of threads are merged into a single scalar superaccumulator. In the fourth level, all scalar superaccumulators within one processor (GPU or MPI process) are combined together into one single superaccumulator accessible by all threads. This step is performed with the standard parallel reduction. It is followed by the MPI reduction among nodes. Finally, the fifth level consists in rounding the global superaccumulator back to the target floating-point format in order to produce the correctly rounded result. One can notice that levels 2 to 4 of the proposed algorithm have several concepts in common with the histogram problem [15].

We present below a more detailed level-by-level explanation of the proposed approach, illustraded in Fig. 3.

3.1 Level 1: Filtering

We propose to extend the classical approach described in Section 2.1 in order to maintain floating-point expansions of size n. We maintain a vector of n floating-point numbers as the accumulator. We update this expansion using error-free transformations (Alg. 2). This new algorithm is amenable to parallelization, vectorization and pipelining by maintaining and updating multiple expansions in parallel.

Algorithm 2 Floating-point expansion of size n with error-free transformations.

Function = ExpansionAccumulate(x)

1: **for** $i = 0 \rightarrow n - 1$ **do**
2: $\quad (a_i, x) \leftarrow \mathrm{TwoSum}(a_i, x)$
3: **end for**
4: **if** $x \neq 0$ **then**
5: $\quad$ Superaccumulate(x)
6: **end if**

Alg. 2 works as an iterative process: the input number is first added to the head of the expansion. Then, the rounding error is extracted by the `TwoSum` algorithm and propagated to the next slot of the expansion. This step is repeated until the n-th slot of the expansion. Finally, underflows can be detected by checking if the last rounding error x is non-zero. If this happens, the result of the accumulation cannot be represented with a floating-point expansion of size n. In this case, the non-zero residue x is forwarded to the scalar superaccumulator (described in Section 3.2). We also propose a version of Alg. 2 that consists in stopping the accumulation loop as soon as the residue is zero ($x \equiv 0$). This technique is called *early-exit*. Its performance depends on two factors: the distribution of input numbers and the ability of the architecture to handle irregular branches.

The challenge of optimizing the first step is to select a suitable value of n. A large value of n will lead to large expansions that eventually reduce the number of calls to the second level (superaccumulator), however this will increase register usage. A small value of n will impose a limited representation range for the expansion that will reduce the pressure on registers, but will cause more transfers towards superaccumulators. In both cases, all components of the floating-point expansion are accumulated into the superaccumulator at the end of the summation.

3.2 Levels 2 and 3: Private and Scalar Superaccumulators

We follow the general approach of the Complete Register implementation proposed by Bohlender and Kulisch [16] and adapt it to a software implementation. This approach allows to accumulate floating-point numbers in amortized $O(1)$ time regardless of the length of the long accumulator. The algorithm behind consists of aligning the floating-point significand and accessing the corresponding words in the accumulator.

We implement the superaccumulator as a vector of 64-bit signed integers that partially overlap by following a high-radix carry-save scheme [17]. Thanks to this representation, the M leading bits of each digit store the carry information, delaying the carry propagation to every 2^M accumulation steps. Unlike Bohlender's and Kulisch's approach, we use multiple carry-save bits per digit to reduce the frequency of carry propagation. Each digit is

responsible for accumulating $w = 64 - M$ bits of the sum; the digit i has a weight of $2^{w \cdot i}$. The exact sum s is expressed as

$$s = \sum_{i=l}^{h} 2^{wi} a_i, \quad -2^{63} \leq a_i < 2^{63}, \quad l = \left\lfloor \frac{e_{min}}{w} \right\rfloor, \quad h = \left\lceil \frac{e_{max}}{w} \right\rceil,$$

where e_{min} and e_{max} are the smallest and the largest binary floating-point exponents, respectively. In our implementation, we choose $w = 56$ and $M = 8$.

The use of 64-bit integer arithmetic allows the digit size w to be greater or equal than the floating-point significand size ($m = 52$), so a mantissa may span at most two digits. In order to split the mantissa of the input number x and accumulate each part to the corresponding digits of the superaccumulator, we use the algorithm depicted in Fig. 4. This algorithm is based on rounding scaled floating-point numbers. First, the exponent of the input number x is extracted and used to compute the indexes k and $k - 1$ of the two consecutive slots in the superaccumulator. Then, x is scaled by multiplying it by 2^{-kw} and rounded to the nearest integer. This integer corresponds to the upper bits of the mantissa that are accumulated to the digit at the index k of the superaccumulator. It is also used to compute the lower part of the mantissa of x that is added to the digit at the position $k - 1$ within the superaccumulator. When the result of the accumulation exceeds the range of the superaccumulator slot, a partial normalization is performed along the propagation of the carry or the borrow. In this case, the upper M bits of the current digit are transferred to the next digit until the carry propagation chain stops. The carry or borrow propagation is only needed at most once per every 2^{M-1} accumulation steps and typically requires only a single propagation step. At the end of the summation, a complete normalization pass propagates all carries.

Once all input numbers are accumulated into private superaccumulators, these superaccumulators are merged together within a group of threads into a single scalar superaccumulator. We use the classic reduction algorithm, albeit with a larger payload than in the customary floating-point case. Reduction is a common parallel primitive available in highly optimized versions on various systems which we rely on. Each reduction step involves the summation of two superaccumulators. All digits of the superaccumulators are summed in parallel to take advantage of vectorization and pipelining. We ensure that no carry can propagate by counting the number of reduction steps that occurred since the last normalization, and trigger normalizations as required. In practice, 8 carry-save bits enable a normalization-free reduction across up to 256 threads.

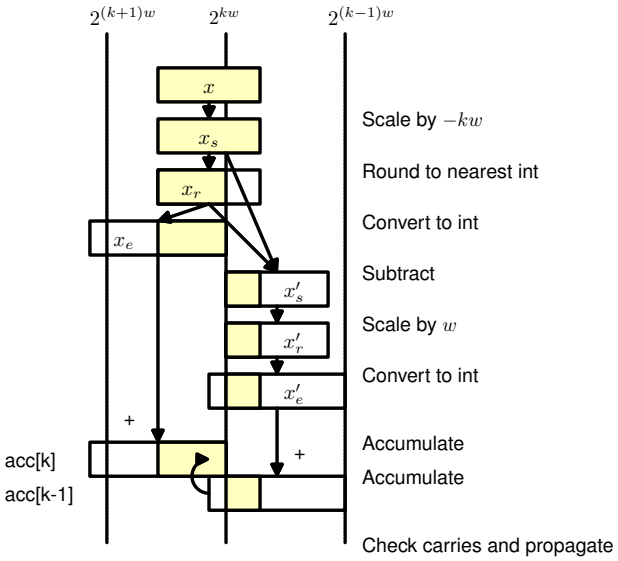


Fig. 4: Scalar accumulation scheme.

3.3 Levels 4 and 5: Parallel Reduction and Rounding

As for private superaccumulators, we perform the same reduction procedure in order to combine together all scalar superaccumulators of one processor (MPI process) into one single global superaccumulator. In turn, the superaccumulators of each node of the cluster are accumulated into a single superaccumulator using the MPI reduction.

To perform the final rounding to the target floating-point format, we first normalize the superaccumulator. Then, digits are scanned for the most-significant digit that is non-zero (positive) or not minus-one (negative); this most-significant digit has the weight k. We compute the sticky bit as a logical OR of digits $k-2$ to 0, and add it to the word $k-2$. This sticky bit is used to discriminate between a number that falls exactly between two floating-point numbers (rounded to the even mantissa) and a number that is slightly higher (rounded up). Finally, words k, $k-1$, and $k-2$ are added together by following the `Add3` algorithm [18], which is based on round-to-odd rounding.

4 Implementations and Experimental Results

This section details our implementations of the multi-level scheme and presents their evaluation on the whole range of parallel platforms, which are listed in Tab. 1: desktop and server CPUs, the Intel Xeon Phi many-core accelerator, and both NVIDIA and AMD GPUs. We verified the accuracy of our implementation by comparing the computed results with the ones produced by the multiple precision library MPFR [19,20], which is, unfortunately, not multi-threaded. In case of binary64, we used 2098 bits (emin + emax + mantissa $= 1022 + 1023 + 53$) within the MPFR library in order to guarantee the bit-wise accuracy as well as reproducibility of the deterministic sums independently of both the round-off errors and dynamic ranges.

Table 1: Hardware platforms employed in the experimental evaluation.

A Intel Core i7-4770 (Haswell)	4 cores with Hyper-Threading	3.400 GHz
B Mesu cluster Intel Xeon E5-4650L (Sandy Bridge)	64 nodes with 2×8 cores	2.600 GHz
C Intel Xeon Phi 5110P	60 cores $\times$ 4-way Multi-Threading	1.053 GHz
D NVIDIA Tesla K20c	13 SMs $\times$ 192 CUDA cores	0.705 GHz
E AMD Radeon HD 7970	32 CUs $\times$ 64 units	0.925 GHz

4.1 Implementations

Our implementations strive to use all resources of modern processors: SIMD instructions, fused-multiply-and-add (FMA) instruction, and multi-threading on multi-core CPUs and Xeon Phi; local memory and atomic instructions on GPUs. For example, on the Intel Haswell architecture, we use AVX intrinsics to benefit from the $4-$way SIMD and implement `TwoSum` by replacing half of the 6 additions/subtractions by FMAs (multiplying by one). The latter optimization allows to use both the floating-point adder and FMA units that can run in parallel on the Haswell micro-architecture. Thread-level parallelism is exposed using OpenMP in order to take advantage of multi-core and hardware multi-threading. The final inter-node reduction within a cluster is performed using MPI.

On the Intel Xeon Phi 5110P many-core accelerator, we benefit from $8-$way SIMD by using $512-$bit vector intrinsics. We perform explicit memory prefetching in order to maximize memory throughput. For the performance experiments, we use 236 threads with the compact affinity.

We develop separate hand-tuned OpenCL implementation for NVIDIA and AMD GPUs. They use 16 superaccumulators per work group, which are stored in local memory. In order to avoid bank conflicts, superaccumulators are interleaved together to spread their digits among different memory banks. Concurrency between 16 threads that share one superaccumulator is resolved through atomic operations.

All provided implementations are parallel multi-threaded implementations.

4.2 Performance scaling

As baselines, we consider several algorithms:

1. A vectorized and parallelized non-deterministic reduction (indicated as "Parallel FP sum" in all figures) that is computed using vector intrinsics and OpenMP reduction on CPUs and Xeon Phi, and using the standard parallel reduction algorithm like the one introduced in [15] for histograms on GPUs;
2. The deterministic floating-point reduction provided by the Intel TBB library [21] (referred as "TBB deterministic");

3. The Fast Deterministic Parallel Sum algorithm, *Alg. 9 Parallel K-fold Reproducible Sum* [7], proposed by Demmel and Nguyen (cited as "Demmel fast"). We have implemented the Fast Deterministic Parallel Sum algorithm on the range of architectures including Intel Xeon Phi co-processors and both NVIDIA and AMD GPUs. This algorithm requires to conduct two reductions: one to find the maximum element and the other for summation. On CPUs, the reduction is performed using Intel TBB, as OpenMP does not natively support reductions with the maximum operator. On GPUs, due to the impossibility to perform synchronization among work-groups within one kernel in OpenCL 1.2, two kernels have to be invoked.

Figs. 5a, 5c, 5e, 6a and 6c present the throughput, which is measured in billions of accumulations per second (Gacc/s), achieved by the summation algorithms as a function of the input dataset size N of double-precision floating-point numbers. For these experiments, we use numbers with a dynamic range of one (numbers with identical exponents), which corresponds to the best case for our method. In the keys of Figs. 5 to 7, "Superacc" corresponds to our algorithm for computing deterministic sums that is solely based on superaccumulators (Section 3.2); "FPEn + Superacc" stands for our algorithm with floating-point expansions of size n ($n = 2 : 8$) in conjunction with error-free transformations and superaccumulators when needed; "FPE8EE + Superacc" represents our algorithm with the expansion of size 8 and the early-exit optimization technique as described in Section 3.1.

We can observe from these plots that on large datasets the performance of the baseline unordered sum as well as the expansion cache is constrained by the memory bandwidth on all platforms. For instance, both algorithms achieve 23.3 GB/s on Platform A over the theoretical peak bandwidth of 25.6 GB/s. Indeed, the peak double-precision performance of this CPU is 218 GFlops. This means that for every double-precision number loaded from memory, 68 floating-point operations could be performed without affecting the total throughput. With only one operation per input number, the conventional floating-point sum or reduction leaves most computational resources idle waiting for data from the memory.

When the input dataset fits within caches, the non-reproducible parallel sum outperforms our expansion cache implementation by a factor of 2 (NVIDIA GPU) to 5 (Intel Haswell). However, the deterministic reduction of TBB runs two orders of magnitude slower (75 Macc/s on Platform A and 136 Macc/s on Platform C) than our version. The general-purpose multi-precision MPFR library (not shown) runs one order of magnitude slower at 9.3 Macc/s on Platform A.

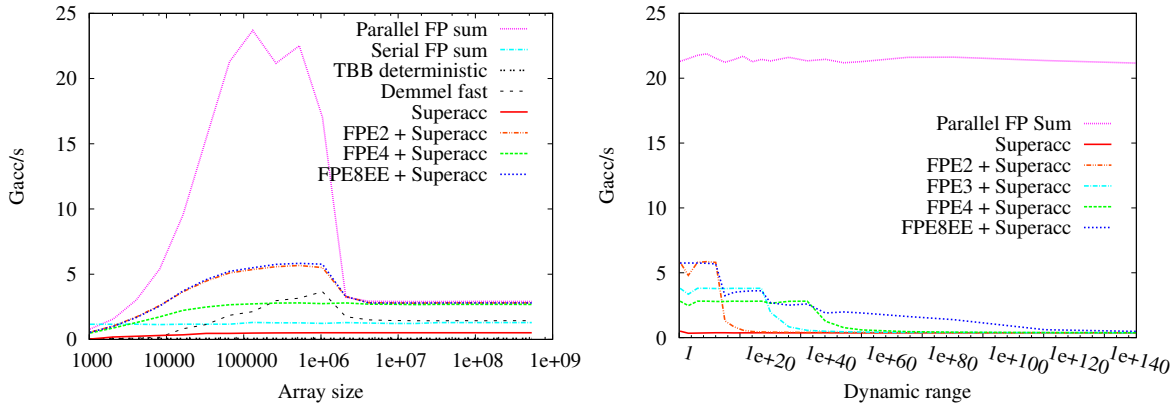
The Fast Deterministic Parallel Sum algorithm requires two passes on the data. In memory-bound scenarios, its execution takes twice the time of the unordered reduction and other single-pass algorithms. In addition, the overhead of the second kernel call impacts the performance of the GPU implementation.

A global MPI reduction is the next step within the fourth level. We have implemented the parallel reduction among compute nodes using `MPI_Reduce()` on the local sums represented as either `binary64` or superaccumulators. Fig. 7a shows the results of performance scaling of our and the other algorithms on the Mesu cluster[6] by varying the number of processors for the summation of $16e06$ double-precision floating-point numbers per each MPI process; such a large dataset size is chosen to ensure the out-of-cache case. For each processor count ranging from 1 to 64 [7], we measure the computation and reduction time of the parallel non-reproducible summation, the TBB deterministic summation, and our algorithm. These algorithms compute the local sums on each MPI process. Then `MPI_Reduce()` is applied to calculate the final sum. We attach two MPI processes to each node, that is composed of two Intel Sandy-Bridge processors. On the figure, the local summation time is colored red and the MPI reduction time is colored green. Since the MPI reduction time is 2-3 order of magnitude smaller than the local summation time even in case of superaccumulators, it is hardly visible on the figure. The total runtime of each algorithm is normalized by the total execution time of the parallel floating-point summation. The TBB deterministic sum is roughly 68 times slower than the parallel floating-point summation on one node. Hence, we cut this timing in order to provided better picture of the rest of the results that do not exceed the slowdown of 9 times.

For the whole range of processors, the execution time of each algorithm is dominated by the local summation time because of the dataset size (out-of-cache case). In addition, due to the equal distribution of computations among MPI processes (each MPI process sums up $16e06$ `binary64` numbers), the computation time is roughly equivalent on the whole range of MPI processes, while the reduction time changes depending on the number of MPI processes involved. The Fast Deterministic Parallel Sum algorithm is 4.72 times slower than the conventional parallel summation with 64 MPI processes (corresponds to 64 CPUs). In contrast, two of our implementations – floating-point expansions of size 2 and size 8 with the early-exit technique – deliver accurate as well as bit-wise reproducible results. The overhead of these two implementations compared to the conventional algorithm on 64 CPUs is 10.4 % and 10.2 %, accordingly.

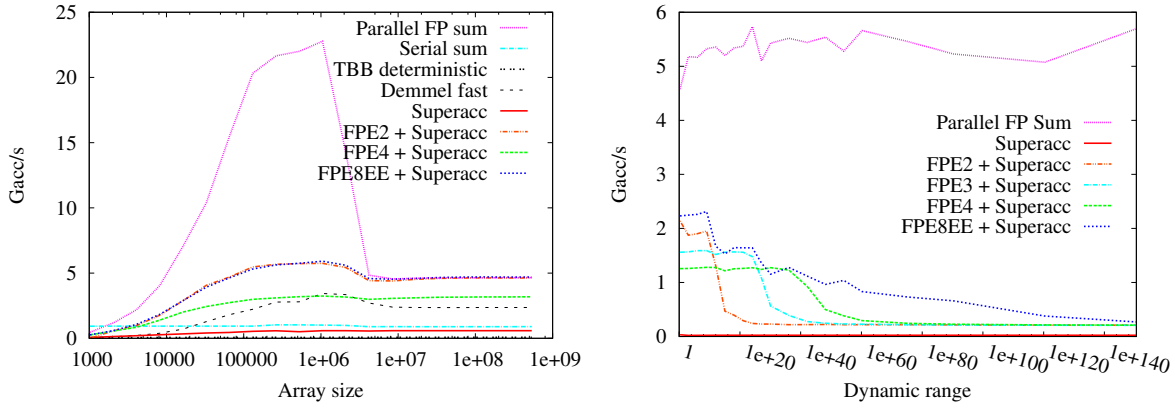
[6] `http://mesu-smn.dsi.upmc.fr/mediawiki/index.php5/Main_Page#Mesu`

[7] Due to the extensive workload on the Mesu cluster and the policies there, we can run at most 64 MPI processes, meaning utilize only a half of the whole cluster.



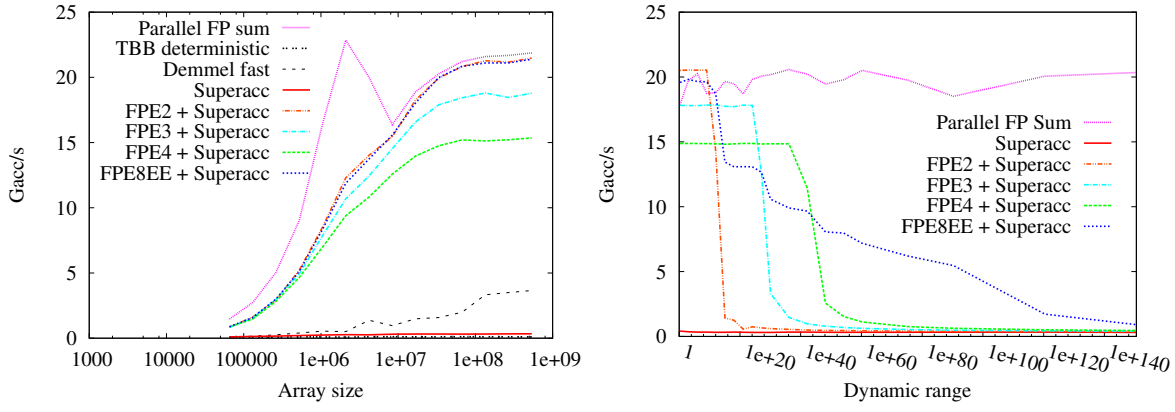
(a) Platform A.

(b) Platform A: $N = 2e06$.



(c) Platform B.

(d) Platform B: $N = 2e06$.

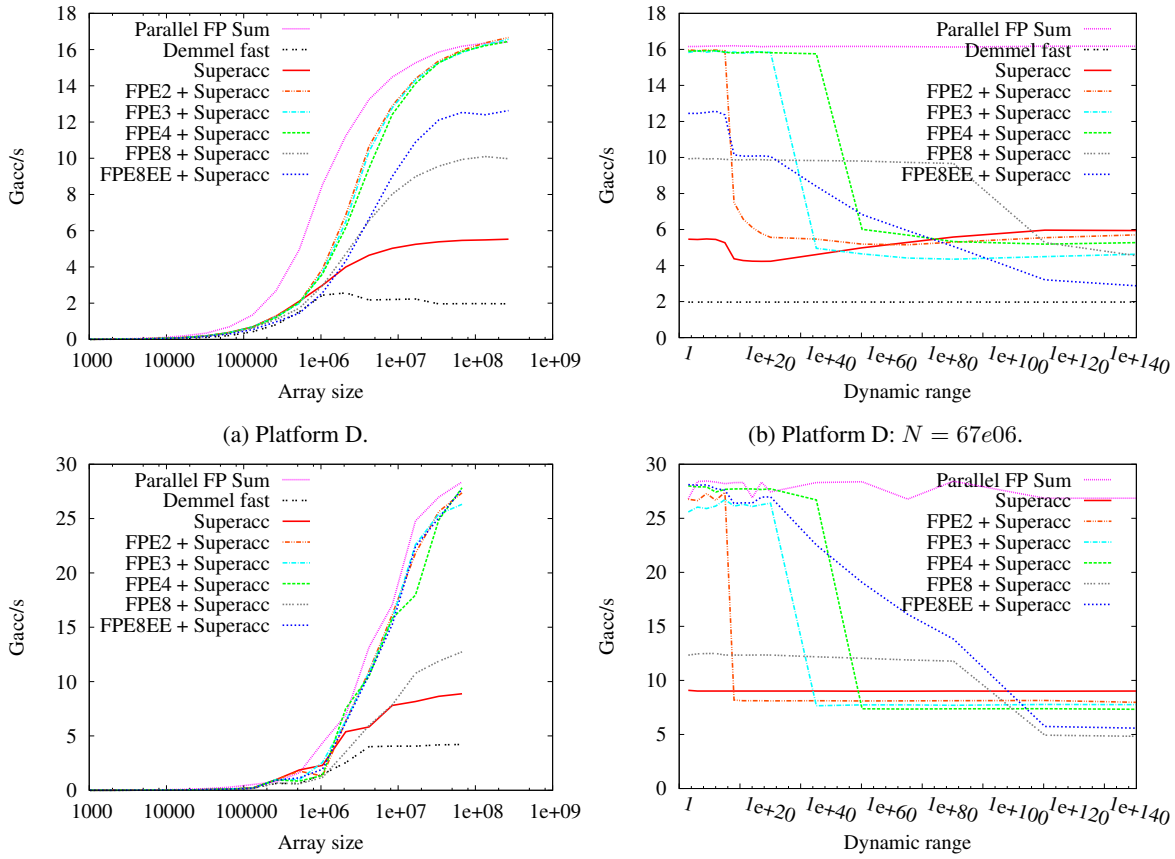


(e) Platform C.

(f) Platform C: $N = 67e06$.

Fig. 5: The summation performance results on modern desktop and server Intel CPUs as well as the Intel Xeon Phi co-processor, see Tab. 1; Gacc/s stands for billions of accumulations per second.

We extend our study to analyze the impact of communication (parallel reduction in our case) on the performance of the above-mentioned algorithms by fixing the number of MPI processes and varying the problem size per each process, see Fig. 7b. The results show that for the in-cache cases the total computation time is dominated by the parallel reduction. This is clearly seen for our algorithm since the parallel reduction on superaccumulators takes much more time than on single `binary64`. When our implementations are already out-of-cache, but the parallel summation still benefits from the in-cache scenario, the performance gap between our implementations and the conventional summation grows and reaches its maximum at $n = 1e06$. That also corresponds to the results on single node, see Fig. 5c. For the out-of-cache cases, these performance gap decreases significantly – so that for $n = 16e06$



(a) Platform D.

(b) Platform D: $N = 67e06$.

(c) Platform E.

(d) Platform E; $N = 67e06$.

Fig. 6: The summation performance results on NVIDIA and AMD GPUs, see Tab. 1.

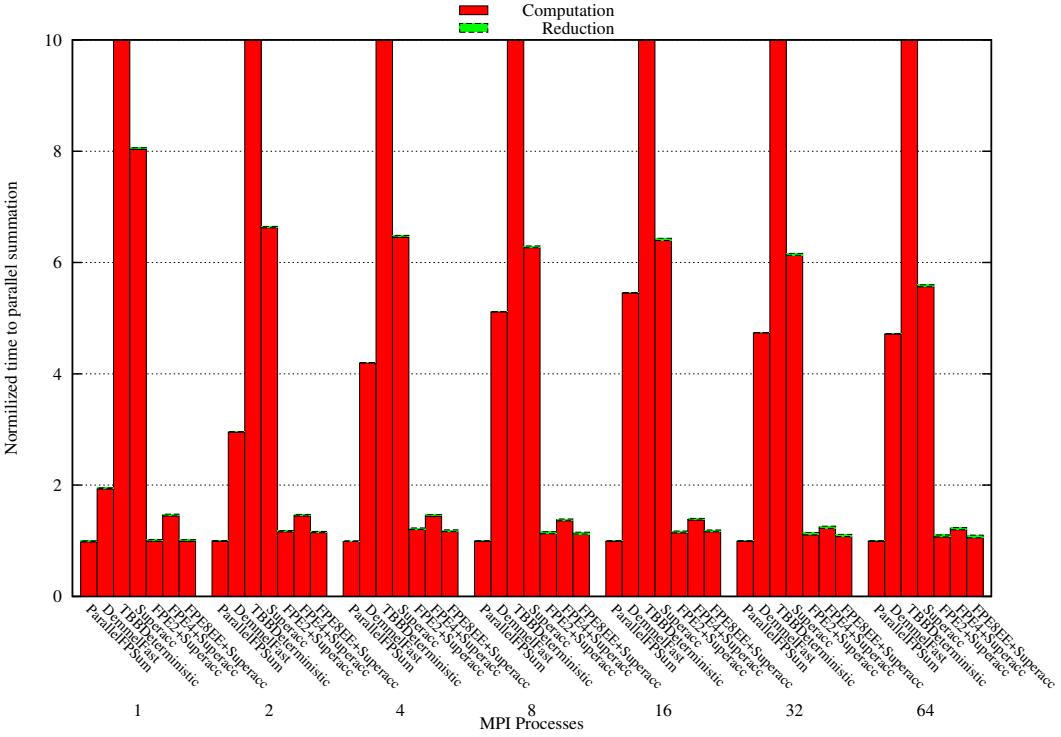
floating-point expansions of size 2 and size 8 with the early-exit technique (both use superaccumulators when needed) are only $8.7\,\%$ and $9.1\,\%$ off the parallel summation performance, respectively.

4.3 Data-dependent performance

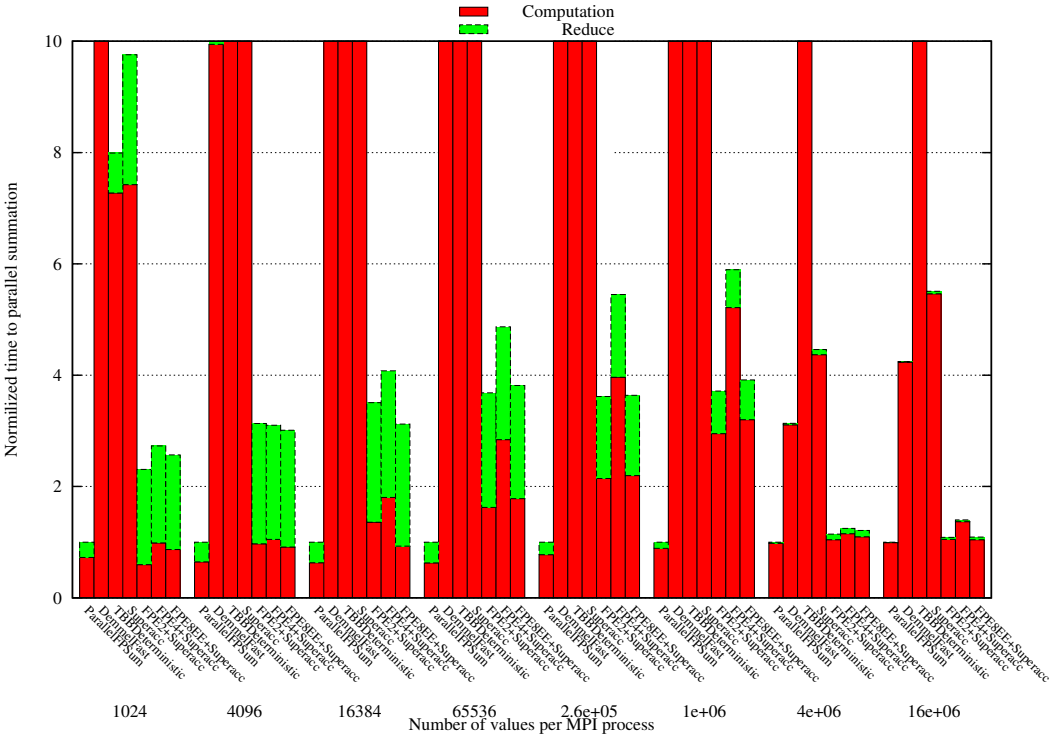
We validate the discussed algorithms using various dynamic ranges for the input arrays by generating pseudo-random numbers following a log-uniform number distribution. This distribution represents the worst case for our algorithm as the exponents of floating-point numbers follow a uniform distribution. Since running experiments on large datasets clamps all curves below the memory bandwidth limit on CPUs, we present results on datasets with 2 millions (Platforms A-C) and 67 millions (Platforms D and E) of elements in order to highlight the difference between implementations.

Figs. 5b, 5d, 5f, 6b and 6d represent the scenario when the array size is fixed, but the range of its elements varies. Floating-point expansions adapt dynamically to the range of input numbers. The expansion cache with 8 words and the early-exit technique saturates the memory bandwidth for dynamic ranges up to 10^{30}, 10^{90}, 10^{15}, 10^{50}, and 10^{50} on platforms A, B, C, D, and E, accordingly. One can observe that in the worst case, our method is between 3 times (on Platform E) and 44 times (on Platform C) slower than the non-deterministic sum. However, it still reaches 64% of the performance of the Demmel's and Nguyen's Fast Deterministic Parallel Sum and exceeds the performance of the serial sum.

The results of the Fast Deterministic Parallel Sum and the TBB deterministic sum algorithms on Platforms A-C are not represented on the dynamic range figures as their performance does not depend on the value of the inputs. The runtime of these methods is constant for any dynamic ranges, for example, see Fig. 6b.



(a) Varying the number of MPI process to sum $n = 16e06$ elements per each.



(b) Fixed the number of MPI processes to 64 and vary the array size.

Fig. 7: Performance scaling on the Mesu cluster.

Superaccumulator performance is also close to constant, because of its accumulation time that is $O(1)$ for any input value. Minor variations occur due to load-store forwarding effects on CPUs and atomic conflicts on NVIDIA GPUs.

These experiments show that our multi-level algorithm is faster than existing alternatives and it is competitive with the standard parallel reduction for input vectors with moderate dynamic range. This confirms our assumptions and objectives stated earlier in Section 3. Moreover, our multi-level algorithm guarantees accurate, deterministic, and reproducible results for numbers of any dynamic range.

5 Related Works

Many techniques have been proposed to increase the accuracy of floating-point summation. One is based on a long accumulator as described in Section 2.2. This solution is suitable for hardware implementations due to the constant accumulation time [16]. A recent analysis of the software cost of long accumulators [22] advocates their usage on new architectures. However it does not incorporate implementations nor numerical validations.

Another approach to exact summation consists in pure floating-point algorithms. The idea is to store the exact sum as an expansion or an unevaluated sum of floating-point numbers [23]. However, as we mentioned already, this solution solely increases the accuracy of basic operations, but it neither reaches bit-accurate results nor is efficient for large precisions. Therefore, some recent works are focusing on hybrid solutions that store the sum as floating-point numbers of fixed exponent [24,25] without completely avoiding the previous drawbacks. These algorithms are mainly sequential and are not suitable for parallel implementations. This is mainly due to the need to scan all the numbers before starting the computation of the sum. Arbitrary precision libraries – like MPFR [19] – are able to provide correct rounding. However, they are not designed to achieve acceptable performance for reproducible results. In case of the MPFR library, it is also not multi-threaded. For instance, our solution is three orders of magnitude faster than MPFR, see Section 4.

To enhance reproducibility – defined as getting a bitwise identical floating-point result from multiple runs of the same code – Intel proposed a "Conditional Numerical Reproducibility" (CNR) in its Math Kernel Library (MKL). However, CNR is slow and does not give any guarantee on the accuracy of the result. Demmel and Nguyen recently introduced a family of algorithms for reproducible summation in floating-point arithmetic [7]. These algorithms always returns the same answer. They first compute an absolute bound of the sum and then round all numbers to a fraction of this bound. So, the addition of the rounded quantities is exact. Since the computed sum may be less accurate than the non-deterministic one, this solution offers no guarantees on the accuracy. It also induces a twofold slowdown as data transfers and reductions need to be performed twice: for computing the bound and the sum. As Section 4 shows, our algorithm is faster in the bandwidth-constrained scenarios with moderate dynamic ranges. Demmel and Nguyen have also improved the previous results [26] by using one single reduction step among nodes. Such an improvement yielded roughly $15\,\%$ overhead on 2048 processors compared to the Intel MKL `dasum()`, but it shows roughly 4.5 times slowdown on 32 processors. Demmel and Nguyen have extended their concept to reproducible BLAS routines, distributed in their ReproBLAS library[8]. Arteada et al. [27] used Demmel and Nguyen's algorithms with improved communication between nodes based on both one- and two-reductions. They were able to reach the same accuracy with slightly better performance results (the overhead is within $10\,\%$) compared to the conventional summation in case of the single reduction algorithm.

6 Conclusions and Future Work

We presented a solution to achieve correct rounding for the floating-point summation problem, along with implementations on multi- and many-core architectures. This yielded results that are both reproducible and accurate to the last bit. For dataset larger than 10^7 elements with dynamic ranges up to 10^{15}, the proposed approach solves the summation problem with similar performance as the classic highly optimized parallel floating-point summations; however, the parallel floating-point summation provides neither correct rounding nor reproducibility. These competitive results were achieved using the expansion of size 4 on both multi-core platforms and GPUs, and the expansion of size 8 with the early-exit optimization on Xeon Phi for low to medium dynamic ranges. Such dynamic ranges correspond to the most common cases that occur in practice. Thanks to the multi-level strategy, we were able to deliver bit-accurate results for large dynamic ranges at 25% of the parallel floating-point summation performance on multi-core and GPU platforms. Furthermore, we demonstrated that the multi-level correctly-rounded summation outperforms the

[8] `http://bebop.cs.berkeley.edu/reproblas/`

other recently proposed reproducible summation algorithms – like Demmel and Nguyen's algorithm – while offering correct rounding.

We followed a multi-level approach for correct rounding that is similar to the one used in elementary functions [28]. This approach consists in optimizing common cases while keeping track of the error in order to route the few difficult cases to dedicated routines; these routines are more expensive, however they guarantee deterministic and bit-accurate results. As the performance of hardware architectures becomes increasingly limited by the so-called memory wall, we expect that the overhead of those dedicated routines will decrease.

We showed that an optimized library implementation of exact summation is viable on current architectures, even in the absence of dedicated hardware support for long accumulation. By guaranteeing a single well-specified result for sums, like the IEEE-754 standard does for basic operations, it offers determinism and numerical safety for a cost that is negligible in most cases. This supports the efforts to make complete arithmetic available as a basic data type in programming languages based on a standard [16].

Rather than requiring hard-wired superaccumulation units, which may delay the adoption of correctly rounded accumulation techniques, we advocate the introduction of more general primitives in instruction sets which may be beneficial to other numerical algorithms. For instance, the complete support for 64-bit arithmetic and conversions in SIMD instruction sets, fast floating-point exponent extraction, and fast memory gather and scatter would further improve the performance of the superaccumulation.

Our eventual goal is to apply the multi-level algorithm to derive reproducible, accurate, and fast library for fundamental linear algebra operations – like those included in the BLAS library – on a set of modern parallel architectures, including Intel Xeon Phi many-core co-processors and GPU accelerators. Moreover, we plan to conduct a priori error analysis of the derived ExBLAS (Exact BLAS) routines. More information on the ExBLAS project as well as the sources can be found in [29].

Acknowledgement

This work undertaken (partially) in the framework of CALSIMLAB is supported by the public grant ANR-11-LABX-0037-01 overseen by the French National Research Agency (ANR) as part of the "Investissements d'Avenir" program (reference: ANR-11-IDEX-0004-02).

This work was granted access to the HPC resources of The Institute for scientific Computing and Simulation financed by Region Île-de-France and the project Equip@Meso (reference ANR-10-EQPX-29-01) overseen by the French National Research Agency (ANR) as part of the "Investissements d'Avenir" program.

References

1. Bailey, D.H., Barrio, R., Borwein, J.M.: High-precision computation: Mathematical physics and dynamics. Appl. Math. and Comput. **218**(20) (2012) 10106–10121
2. Bergman, K., al.: Exascale computing study: Technology challenges in achieving exascale systems. DARPA Report (September 2008)
3. Whitehead, N., Fit-Florea, A.: Precision & performance: Floating point and IEEE 754 compliance for NVIDIA GPUs. Technical report, NVIDIA (2011)
4. Corden, M.: Differences in floating-point arithmetic between Intel® Xeon® processors and the Intel® Xeon Phi™ coprocessor. Technical report, Intel (March 2013)
5. Katranov, A.: Deterministic reduction: a new community preview feature in Intel® Threading Building Blocks. Technical report, Intel (May 2012)
6. Doertel, K.: Best known method: Avoid heterogeneous precision in control flow calculations. Technical report, Intel (August 2013)
7. Demmel, J., Nguyen, H.D.: Fast reproducible floating-point summation. In: Proceedings of the 21st IEEE Symposium on Computer Arithmetic, Austin, Texas, USA. (2013) 163–172
8. IEEE Computer Society: IEEE Standard for Floating-Point Arithmetic. IEEE Standard 754-2008 (August 2008) Available at `http://ieeexplore.ieee.org/servlet/opac?punumber=4610933`.
9. Higham, N.J.: Accuracy and stability of numerical algorithms, second ed. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA (2002)
10. 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)
11. Knuth, D.E.: The Art of Computer Programming, Volume 2: Seminumerical Algorithms, third ed. Addison-Wesley (1997)
12. Li, X.S., Demmel, J.W., Bailey, D.H., Henry, G., Hida, Y., Iskandar, J., Kahan, W., Kang, S.Y., Kapur, A., Martin, M.C., Thompson, B.J., Tung, T., Yoo, D.J.: Design, implementation and testing of extended and mixed precision BLAS. ACM Trans. Math. Softw. **28**(2) (2002) 152–205

13. Hida, Y., Li, X.S., Bailey, D.H.: Algorithms for quad-double precision floating point arithmetic. In: Proceedings of the 15th IEEE Symposium on Computer Arithmetic, IEEE Computer Society Press, Los Alamitos, CA, USA (2001) 155–162
14. Kulisch, U., Snyder, V.: The Exact Dot Product As Basic Tool for Long Interval Arithmetic. Computing **91**(3) (March 2011) 307–313
15. Shams, R., Kennedy, R.: Efficient histogram algorithms for NVIDIA CUDA compatible devices. In: Proceedings of the International Conference on Signal Processing and Communications Systems (ICSPCS), Gold Coast, Australia. (2007) 418–422
16. Bohlender, G., Kulisch, U.: Comments on fast and exact accumulation of products. In Jónasson, K., ed.: Applied Parallel and Scientific Computing. Volume 7134 of Lecture Notes in Computer Science. Springer (2012) 148–156
17. Defour, D., De Dinechin, F.: Software carry-save for fast multiple-precision algorithms. In: Proceedings of the 35th International Congress of Mathematical Software, Beijing, China, World Scientific (2002) 2002–08
18. Boldo, S., Melquiond, G.: Emulation of a FMA and Correctly Rounded Sums: Proved Algorithms Using Rounding to Odd. IEEE Transactions on Computers **57**(4) (2008) 462–471
19. Fousse, L., Hanrot, G., Lefèvre, V., Pélissier, P., Zimmermann, P.: MPFR: A Multiple-precision Binary Floating-point Library with Correct Rounding. ACM Trans. Math. Softw. **33**(2) (2007) 13
20. MPFR Dev Team: The GNU MPFR Library Available via the WWW. Cited 28 May 2014. `http://www.mpfr.org`.
21. Reinders, J.: Intel Threading Building Blocks, first ed. O'Reilly & Associates, Inc., Sebastopol, CA, USA (2007)
22. McCalpin, J.D.: Is "ordered summation" a hard problem to speed up? (February 2012) Available via the WWW. Cited 28 May 2014. `http://blogs.utexas.edu/jdm4372/2012/02/15/is-ordered-summation-a-hard-problem-to-speed-up/`.
23. Shewchuk, J.R.: Robust adaptive floating-point geometric predicates. In: Proceedings of the twelfth annual symposium on Computational geometry, ACM (1996) 141–150
24. Rump, S.M.: Ultimately fast accurate summation. SIAM J. Scientific Computing **31**(5) (2009) 3466–3502
25. Zhu, Y.K., Hayes, W.B.: Algorithm 908: Online Exact Summation of Floating-Point Streams. ACM Trans. Math. Softw. **37**(3) (2010) 37:1–37:13
26. Demmel, J., Nguyen, H.D.: Numerical Reproducibility and Accuracy at ExaScale (invited talk). In: Proceedings of the 21st IEEE Symposium on Computer Arithmetic, Austin, Texas, USA. (April 2013) 235–237
27. Arteaga, A., Fuhrer, O., Hoefler, T.: Designing bit-reproducible portable high-performance applications. In: Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium. IPDPS '14, Washington, DC, USA, IEEE Computer Society (2014) 1235–1244
28. CR-Libm: CR-Libm – a library of correctly rounded elementary functions in double-precision. (November 2007) Available via the WWW. Cited 28 May 2014. `http://lipforge.ens-lyon.fr/www/crlibm/`.
29. Iakymchuk, R., Collange, S., Defour, D., Graillat, S.: ExBLAS – Exact BLAS `https://exblas.lip6.fr/`.