Parallel Computing 49 (2015) 83–97

Contents lists available at ScienceDirect

Parallel Computing

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

Numerical reproducibility for the parallel reduction on multi- and many-core architectures

Sylvain Collange [a], David Defour [b], Stef Graillat [c,d], Roman Iakymchuk [c,d,e,*]

[a] INRIA – Centre de recherche Rennes – Bretagne Atlantique, Campus de Beaulieu, Rennes F-35042, Cedex France
[b] DALI–LIRMM, Université de Perpignan, 52 avenue Paul Alduy, Perpignan F-66860, France
[c] Sorbonne Universités, UPMC Univ Paris 06, UMR 7606, LIP6, 4 place Jussieu, Paris F-75005, France
[d] CNRS, UMR 7606, LIP6, F-75005 Paris, France
[e] Sorbonne Universités, UPMC Univ Paris 06, ICS, Paris F-75005, France

ARTICLE INFO

Article history:
Received 16 February 2015
Revised 29 August 2015
Accepted 6 September 2015
Available online 14 September 2015

Keywords:
Parallel floating-point summation
Reproducibility
Accuracy
Long accumulator
Error-free transformations
Multi- and many-core architectures

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: first, a filtering stage that relies on fast vectorized floating-point expansion; second, 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 co-processors, 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.

1. Introduction

The increasing computational 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 dot product or numerical integration. As finite-precision 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

* Corresponding author at: Sorbonne Universités, UPMC Univ Paris 06, UMR 7606, LIP6, 4 place Jussieu, F-75005 Paris, France. Tel.: +33 1 44 27 88 76.
E-mail addresses: sylvain.collange@inria.fr (S. Collange), david.defour@univ-perp.fr (D. Defour), stef.graillat@lip6.fr (S. Graillat), roman.iakymchuk@lip6.fr (R. Iakymchuk).

http://dx.doi.org/10.1016/j.parco.2015.09.001
0167-8191/

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*[1]; 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:

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

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

The IEEE-754 standard [8], which was revised in 2008, specifies floating-point formats, see Table 1, 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 allows numbers 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}.

Table 1
Main floating-point formats in the IEEE-754 standard.

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	$u = 2^{1-53} \approx 2,22 \times 10^{-16}$	$\approx 10^{\pm 308}$

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 cancellation phenomenon which consist in the elimination of the lowest-order bits of the sum. For example, denoting $\oplus$ the addition in `binary64`floating-point arithmetic, $(-1 \oplus 1) \oplus 2^{-53} \neq -1 \oplus (1 \oplus 2^{-53})$ since $(-1 \oplus 1) \oplus 2^{-53} = 2^{-53}$ and $-1 \oplus (1 \oplus 2^{-53}) = 0$. Thus, the accuracy of a floating-point summation depends on the order of the evaluation. More detailed explanation can be found in the main references on floating-point arithmetic [9,10].

Two approaches enable the addition of floating-point numbers 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 and uses it to correct the answer and is described in 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 and is described in Section 2.2.

[1] We use names long accumulator and superaccumulator interchangeably.

2.1. Floating-point expansions

Floating-point numbers can be summed exactly using floating-point arithmetic only, by recovering and tracking rounding errors. If a and b are two floating-point numbers, $r := a \oplus b$ is a floating-point number, but $a + b$ is not 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 [10]. Such an algorithm is called error-free transformation. The traditional error-free transformation for addition is `TwoSum` [11], depicted in Algorithm 1. The `TwoSum` algorithm relies only on floating-point additions and subtractions.

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

```
1 Function [r, s] = TwoSum(a, b)
      Data: a and b are two floating-point numbers.
      Result: r and s hold the result and the error of the sum, accordingly.
2     r ← a + b
3     z ← r − a
4     s ← (a − (r − z)) + (b − z)
```

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. Accumulating one floating-point number to an expansion is an iterative operation: the floating-point number is first added to the head of the expansion. 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.

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

The main advantage of this solution is that expansions only involve basic floating-point operations and no memory accesses, which allows very efficient implementations on modern architectures thanks to pipelining and SIMD units. In this case, the accuracy of small-size expansions is insufficient for the summation of numerous floating-point numbers or sums with a large dynamic range. In addition, 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 stores 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 away 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 a 4288-bit accumulator to compute exact dot products of `binary64` vectors. The summation is performed without loss of information by accumulating every floating-point input number in the long accumulator as illustrated in Fig. 1. The superaccumulator is suitable to compute the exact result of a 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.

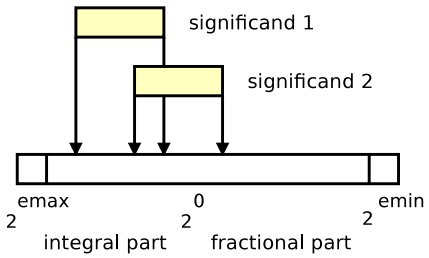


Fig. 1. Long accumulator.

3. Hierarchical superaccumulation scheme

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

We follow a multi-level approach presented in Algorithm 2 and shown in Fig. 2. The accumulation of floating-point numbers involves five levels. This decomposition suits well the memory hierarchy as well as the nested parallelism on modern architectures. In addition, such approach makes full use of SIMD, multi-thread, and multi-core parallelism.

Algorithm 2: Multi-level summation.

```
1 Function ExSUM(arr)
    Data: arr is an input array of floating-point numbers.
    Result: Reproducible and correctly rounded result of summation.
2   for i = 0 → n − 1 do
3     | ExpansionAccumulate(arr[i])
4   end
5   Local reduction of superaccumulators        ▷ This reduction is implemented in the Histogram style
6   MPI reduction of superaccumulators
7   Rounding to the target format
```

The first level consists of floating-point expansions in order to perform fast accumulation of 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 lightweight thread maintains its own expansion of size p. In both cases, floating-point expansions remain in the processor registers. The first level only involve floating point computations and no memory accesses beside reading the input data.

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 fast local memories like cache (on CPUs and Xeon Phi) or local memory (on 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 local superaccumulator. In the fourth level, all local 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 2–4 of the proposed algorithm have several concepts in common with the histogram problem [15]. Histogram is a commonly used analysis tool in image processing and data mining applications. It computes the frequency of occurrence of each data element. Parallel implementations of histograms first divide the input array among threads or groups of threads. Second, it processes each sub-array and stores the result in local sub-histograms, which are long fixed-point accumulators such as superaccumulators, but without the carry or borrow propagation. Third, those sub-histograms are merges into a single resulting histogram.

We present below a more detailed level-by-level explanation of the proposed approach, illustrated in Algorithm 2 and Fig. 2.

3.1. Level 1: filtering

We propose to extend the classical approach described in Section 2.1. We maintain a vector of p floating-point numbers corresponding to floating-point expansions of size p. We update this expansion using 1,1,0 error-free transformations, Algorithm 3. This new algorithm is amenable to parallelization, vectorization, and pipelining by maintaining and updating multiple expansions in parallel.

Algorithm 3 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 pth 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 p. In this case, the non-zero residue x is forwarded to the private superaccumulator (described in Section 3.2). We also propose a version of Algorithm 3 that consists in stopping the accumulation loop as soon as the residue is zero ($x \equiv 0$). We call this technique an *early-exit* optimization. Its performance depends on two factors: the distribution of input numbers and the ability of the architecture to handle irregular branches.

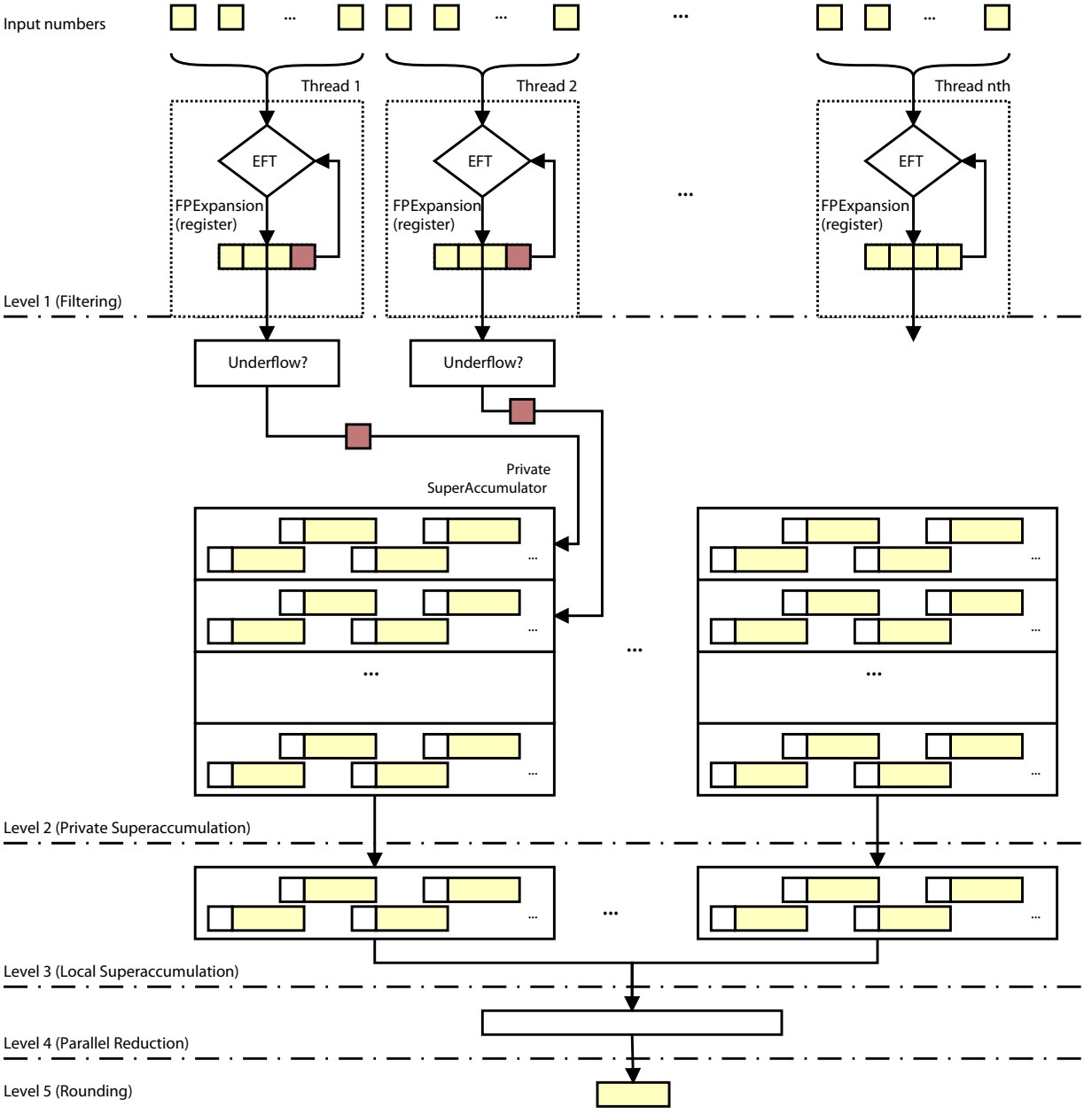


Fig. 2. Multi-level summation approach.

Algorithm 3: Floating-point expansion a of size p with error-free transformations.

```
1 Function ExpansionAccumulate(x)
    Data: x is a floating-point number.
2   for i = 0 → p − 1 do
3     (a[i], x) ← TwoSum(a[i], x)
4   end
5   if x ≠ 0 then
6     Superaccumulate(x)
7   end
```

3.2. Levels 2 and 3: private and local 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 us to accumulate floating-point numbers in amortized $O(1)$ time regardless of the length of the long accumulator. The underlying algorithm 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 called *digits* 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. 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 $acc[i]$ has a weight of 2^{wi}. The exact sum s is expressed as

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

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 Algorithm 4 and demonstrated in Fig. 3. 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, three steps are taken. First, the carry bit resulting from the overflow is recovered. Second, the M upper carry-save bits of the digit are cleared and combined with the carry bit. Third, these $M+1$ carry bits are added to the next digit. If the latter addition triggers an integer overflow, the process is repeated until the carry propagation chain ends. 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. We refer to the propagation of carry-save bits from one digit to the next as a partial normalization. A full normalization propagates all carry-save bits along the accumulator.

Algorithm 4: Private accumulation.

```
1  Function = Superaccumulate(x)
       Data: x is a floating-point number.
2      e ← exponent(x)
3      e_w ← ⌊e / w⌋
4      k ← e_w + ⌈(1023+52) / w⌉
5      x_s ← x × 2^(−w×e_w)                          ▷ Scale x
6      while x_s ≠ 0 do
7          x_r ← round(x_s)                          ▷ Get high part as FP
8          x_i ← llrint(x_s)                         ▷ High part as integer
9          AccumulateWord(x_i, k)                    ▷ Adds x_i to acc[k] and checks carries
10         x_s ← x_s − x_r                           ▷ Low part as FP
11         x_s ← x_s × 2^w                           ▷ Scale x_s
12         k ← k − 1
13     end
```

Once all input numbers are accumulated into private superaccumulators, these superaccumulators are merged together within a group of threads into a single local superaccumulator. We use the classic reduction algorithm, albeit with a larger payload than in the customary floating-point case. Since a reduction is a commonly-provided parallel primitive, we rely on the highly optimized versions for each architecture. 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 is lost 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 per local superaccumulator.

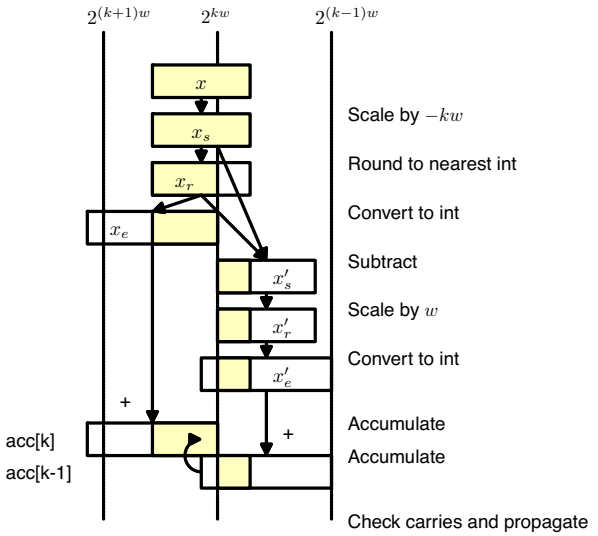


Fig. 3. Private accumulation scheme.

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 local 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. For that, we use two implementations: a reduction of each element of global superaccumulators and a reduction of the entire superaccumulators.

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 at indexes from $k-2$ to 0, and add it to the word at the position $k-2$. 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 at indexes k, $k-1$, and $k-2$ are added together by following the `Add3` algorithm [18], which relies on the round-to-odd rounding.

3.4. Performance model for Algorithm 2

The challenge of optimizing the first step is to select a suitable value of the floating-point expansion size p. A large value of p will lead to large expansions that eventually reduce the number of calls to the second level (superaccumulators). However, this will increase register usage. A small value of p 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.

This trade-off can be captured by a theoretical performance model. For an input vector of size n with dynamic range d, assuming the early-exit optimization technique, the cost of accumulation can be formulated as:

$$n \times \left\lceil \frac{d}{52} \right\rceil \times \frac{C_{fpe}}{VL} + n_t \times p \times VL \times C_{sa}, \quad \text{when } d < 52p,$$

$$n \times \left(p \times \frac{C_{fpe}}{VL} + C_{sa} \right) + n_t \times p \times VL \times C_{sa}, \quad \text{when } d \geq 52p.$$

where n_t is the thread count, $C_{fpe} = 6$ flops, see Algorithm 1, is the cost of the expansion update, VL is the architecture-dependent vector length on SIMD architectures (4 with AVX and 1 on GPUs), and $C_{sa} = 16$ flops $+2$ indirect memory accesses is the cost of the superaccumulator update. The right-hand side term is the cost of flushing expansions to superaccumulators at the end of the summation and gets negligible as n increases. This performance model captures the "staircase" effect observed in the dynamic range plots, see Fig. 4 (b, d, and f) and Fig. 5 (b and d).

4. Implementation 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 Table 2: desktop and server Intel CPUs, the Intel Xeon Phi many-core accelerator, and

Table 2
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 × 4-way Multi-Threading	1.053 GHz
D	NVIDIA Tesla K20c	13 SMs × 192 CUDA cores	0.705 GHz
E	AMD Radeon HD 7970	32 CUs × 64 units	0.925 GHz

both NVIDIA and AMD GPUs. We verified the accuracy of our implementations by comparing the computed results with the ones produced by the multiple precision library MPFR [19,20], which is not multi-threaded and runs only on CPUs. 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.

4.1. Implementation

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 us 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 among cluster nodes 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 the memory throughput. For the performance experiments, we use 236 threads with the compact affinity.

We develop hand-tuned OpenCL implementations 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. Baseline algorithms and experimental setup

As baselines, we consider several algorithms:

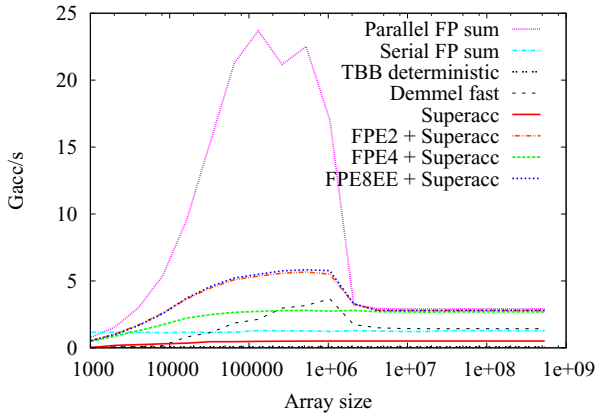
1. A vectorized and parallelized non-deterministic reduction (indicated as “Parallel FP sum” in all figures) that is computed using vector intrinsics and OpenMP reduction on CPUs and Xeon Phi, and using the standard parallel reduction algorithm like the one introduced in [15] for histograms on GPUs;
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 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; min/max operations were not widely supported at the time the implementation was written. 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.
4. The one reduction reproducible summation, *Alg. 6 Sequential Reproducible Summation* [22], (referred as ReproBLAS) from the ReproBLAS library [2];
5. The single-sweep reduction [23] with two and three levels (cited as bitrep2 and bitrep3, accordingly) from the bitrep library [3].

All the libraries and routines were compiled with the same compiler under the same compiler options. For instance, on CPUs we used the version 4.9.0 and `gcc` or `g++` compilers from GNU Compiler Collection with the `-march=core-avx-i -fabi-version=0 -O3` compiler options.

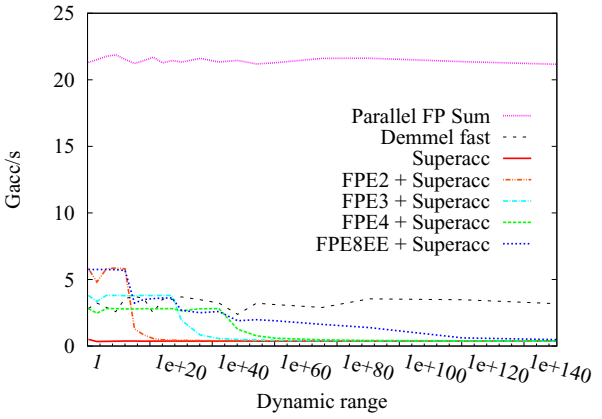
4.3. Single node evaluation

Fig. 4 (a, c, and e) and Fig. 5 (a and c) 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

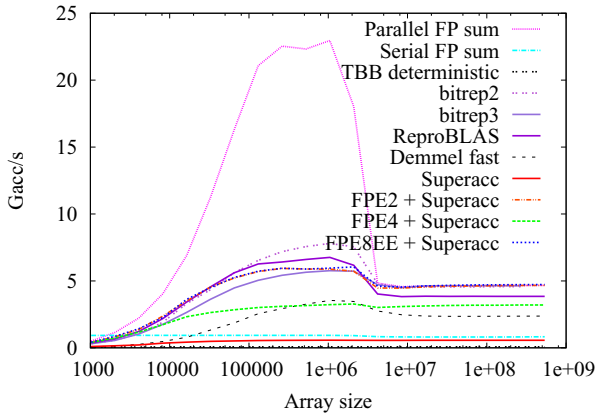
[2] http://bebop.cs.berkeley.edu/reproblas/
[3] https://github.com/andyspiros/bitrep



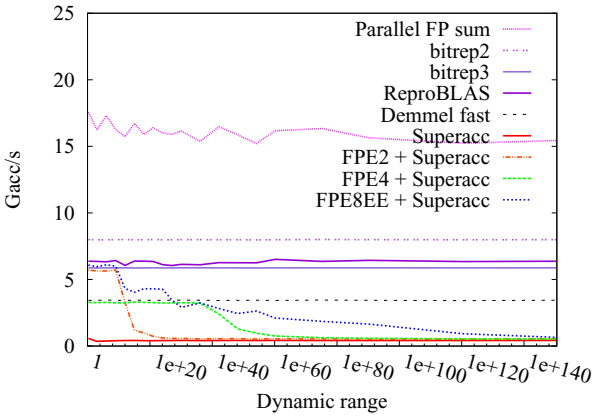
(a) Platform A: Haswell.



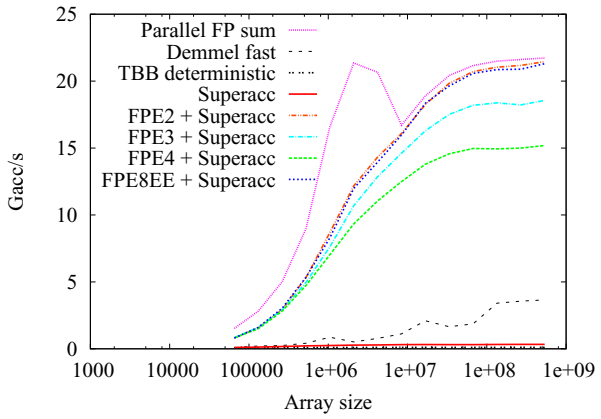
(b) Platform A: $n = 2e06$.



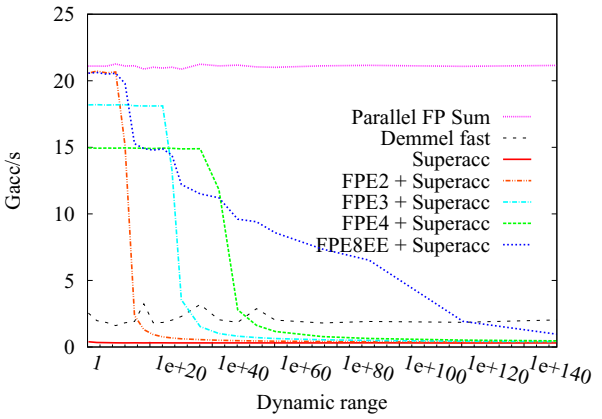
(c) Platform B: Sandy Bridge.



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



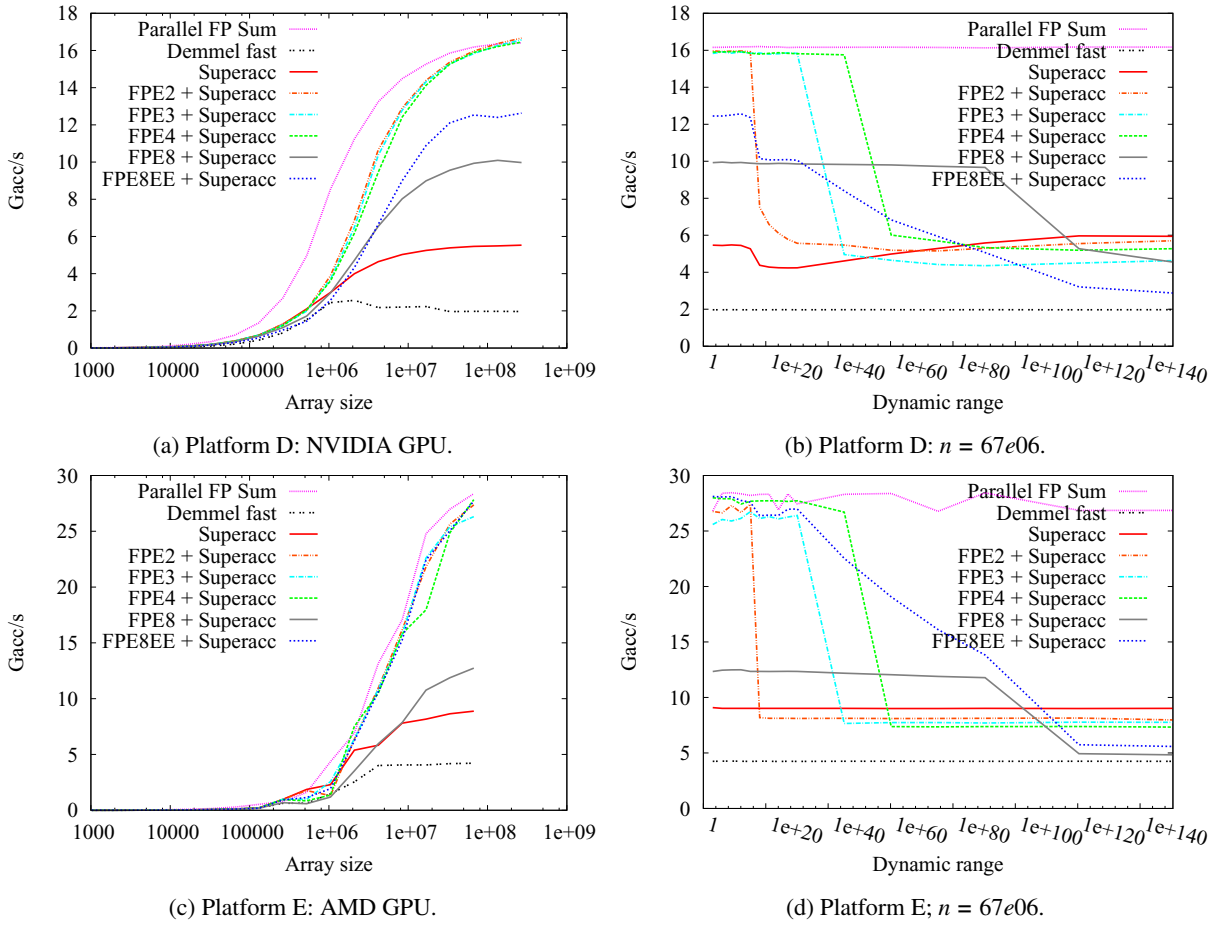
(e) Platform C: Xeon Phi.



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

Fig. 4. The summation performance results on modern desktop and server Intel CPUs as well as the Intel Xeon Phi co-processor, see Table 2; Gacc/s stands for billions of accumulations per second.

numbers. For these experiments, we use numbers with a dynamic range of 1 (numbers with identical exponents), which corresponds to the best case for our method. In the legends of Figs. 4–6, "Superacc" corresponds to our algorithm for computing deterministic sums that is solely based on superaccumulators, see Section 3.2; "FPEp + Superacc" stands for our algorithm with floating-point expansions of size p ($p = 2 : 8$) 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.



(a) Platform D: NVIDIA GPU.

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

(c) Platform E: AMD GPU.

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

Fig. 5. The summation performance results on NVIDIA and AMD GPUs, see Table 2.

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 on 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.

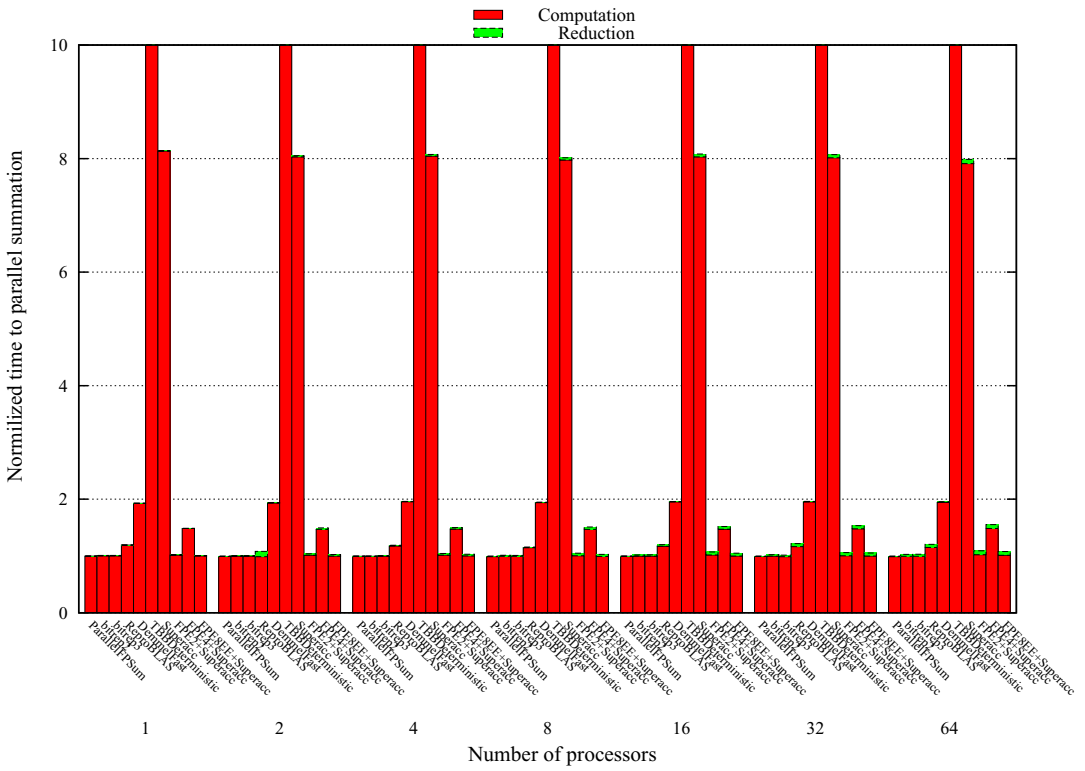
When the input dataset fits within caches, the non-reproducible parallel sum outperforms our expansion cache implementations by a factor of 2 on NVIDIA GPU to 5 on Intel Haswell. However, the deterministic reduction of TBB runs two orders of magnitude slower (75 Macc/s on Platform A and 136 Macc/s on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

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

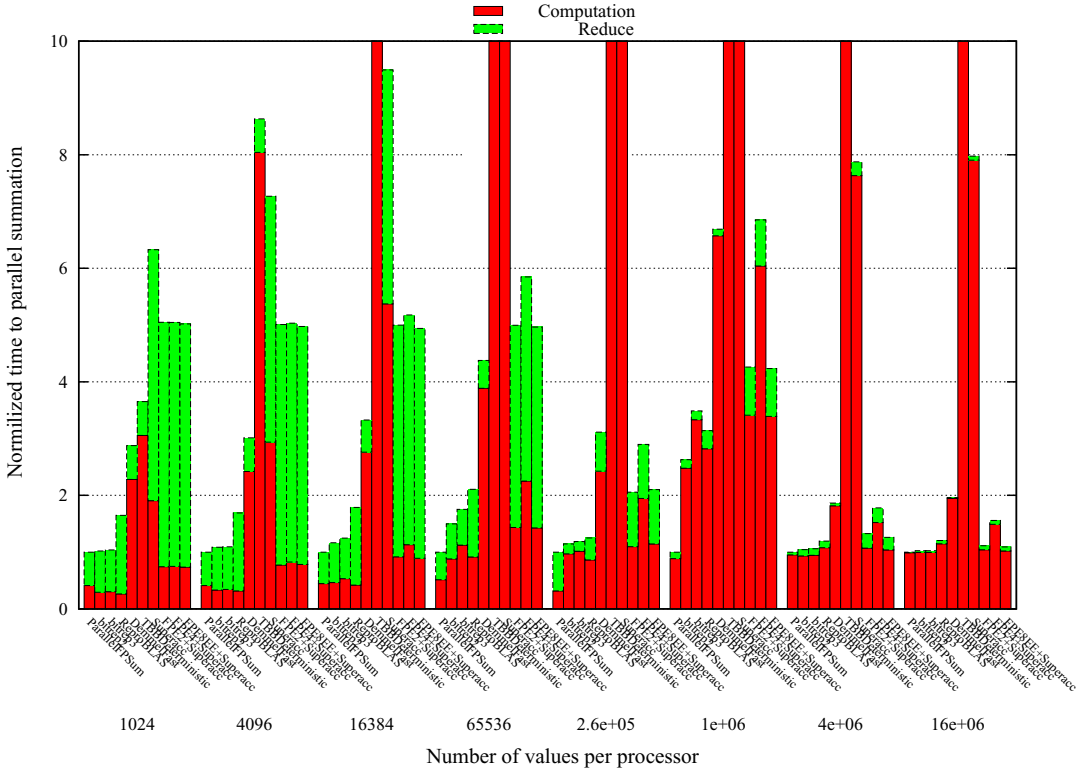
The single-pass reductions from the ReproBLAS and bitrep libraries outperform our implementations for middle range problems. However, for larger problems, the one-reduction summation from ReproBLAS runs 12 % slower than our implementations with floating-point expansions of size 2 and 8 with the early-exit technique in conjunction with superaccumulators. Our implementation exhibits commensurate performance with the single-sweep reductions from bitrep, but with correct rounding. The single-pass reductions do not guarantee correctly rounded results: for instance, the single-sweep reduction with two levels (bitrep2) often loses some bits of accuracy even for datasets with the dynamic range 1.

4.3.1. Impact of the dynamic range

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. To highlight the effect of varying dynamic ranges, we choose a



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



(b) Varying the array size for the fixed number of processors ($np = 64$).

Fig. 6. Performance scaling on the Mesu cluster.

single configuration, where memory performance is not a bottleneck, with a large-enough dataset that fits in the last-level cache; we select such datasets in order to reduce the initialization overhead and the measurement noise, and also because running experiments on large datasets clamps all curves below the memory bandwidth limit on CPUs. This corresponds to the peak performance in most cases. The only exception is "Parallel FP sum" on Platform B, which is limited by the L3 cache bandwidth. We present results on datasets with 2 millions of elements on Platforms A and B and 67 millions on Platforms C–E in order to show the difference between implementations.

Figs. 4(b, d, and f) and 5(b and d) 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 efficiency of various implementations with floating-point expansions depends mainly on the size p of expansions and the dynamic range d of input numbers. Regarding the latter, for small dynamic ranges the small size of expansions works well since the error, even if it is zero, is propagated only few times and there are only several switches to superaccumulators. However, for larger dynamic ranges one would prefer using expansions of size 6 or 8, because of the costly switches to superaccumulators in the case of small expansions. Expansions with the early-exit technique combine benefits from small and larger expansions as they can cover small and large dynamic ranges, delivering good performance thanks to the early-exit technique. However, their performance strongly depends on the ability of the architecture to handle irregular branches and the size of the register space.

The performance of the Fast Deterministic Parallel Sum algorithm, the TBB deterministic sum algorithm, the single-sweep reduction from bitrep, and the one-reduction summation from ReproBLAS does not depend on the value of the inputs and, therefore, the runtime of these methods is constant for any dynamic ranges; for example, see Fig. 4 (b, d, and f) and Fig. 5 (b and d). Due to that, only some of these methods are shown as references on the dynamic range figures.

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

To sum up, these experiments show that our multi-level algorithm is either faster than or deliver equivalent performance to the existing alternatives and it is comparable with the standard parallel reduction for input vectors with moderate dynamic ranges. This confirms our assumptions and objectives stated earlier in Section 3. Moreover, our multi-level algorithm guarantees accurate, deterministic, and reproducible results for input data of any dynamic ranges. Furthermore, we observe that the dynamic range has no measurable effect on the cache-friendliness of algorithms.

4.3.2. Memory footprint

The memory footprint of the multi-level summation algorithm (Algorithm 2) can be characterized by the following formula:

$$\left(p \times VL + \frac{d+1}{w}\right) \times 8 \times n_t,$$

where p is the size of the floating-point expansion, d is the dynamic range, w is defined in Section 3.2, 8 is the size in bytes of `binary64`, VL is the vector length, and n_t is the number of threads involved. On CPU platforms, the total footprint of the superaccumulator and the floating-point expansion is always less than 1 KB. Thus, these structures always fit in L1 cache and their size is negligible compared to the input data size. Moreover, floating-point expansions can be kept in registers and the superaccumulator in L1 cache on CPU or local memory on GPU.

4.4. Multiple node evaluation

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. 6a shows the results of performance scaling of our and other reduction algorithms on the Mesu cluster[4] by varying the number of processors for the summation of 16 millions double-precision floating-point numbers per each processor; such a large dataset size is chosen to ensure the out-of-cache scenario. For each processor count ranging from 1 to 64, we measure the computation and reduction time of the parallel non-reproducible summation, the TBB deterministic summation, single-sweep reductions from bitrep, one-reduction summation from ReproBLAS, and our algorithm. These algorithms compute the local sums on each MPI process. Then, `MPI_Reduce()` or `MPI_Allreduce()` are applied to calculate the final sum. We use two MPI processes per node that is composed of two Intel Sandy-Bridge processors. In case of ReproBLAS and bitrep, we use one MPI process per core, so that we have 512 MPI processes in total instead of 64 MPI processes and 8 OpenMP threads per each process in our implementations. 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 the 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.

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

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 a 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 (the out-of-cache scenario). In addition, due to the equal distribution of computations among MPI processes (each MPI process sums up 16 or 2 millions of `binary64` numbers in our or both ReproBLAS and bitrep cases, accordingly), 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 single-sweep reduction with two and three levels from bitrep are 2.9% and 3.1% slower than the conventional parallel non-reproducible summation on 64 CPUs. The Fast Deterministic Parallel Sum ("Demmel fast") algorithm is roughly two times (95.5%) slower than the conventional summation, while the one-reduction parallel summation from ReproBLAS is only 20.6% off the conventional summation. In contrast, two of our implementations – floating-point expansions of size 2 and size 8 with the early-exit technique – deliver both correctly rounded and reproducible results for a limited overhead of 9.2% and 7.8% compared to the conventional algorithm.

4.4.1. Impact of MPI communication

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 processors/MPI processes and varying the problem size per each processor, see Fig. 6b. 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 `binary64`s. 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 a single node, see Fig. 4c. For larger input date sizes n, this performance gap decreases significantly – so that for $n = 16e06$ floating-point expansions of sizes 2 and 8 with the early-exit technique (both use superaccumulators when needed) remain our best implementations with the identical performance results to Fig. 6a.

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 [24] 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 [25]. 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 focus on hybrid solutions that store the sum as floating-point numbers of fixed exponent [26,27] without completely avoiding the previous drawbacks. These algorithms are mainly sequential and are not suitable for parallelization. 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. Moreover, the MPFR library 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 bit-wise identical floating-point result from multiple runs of the same code – Intel proposed a "Conditional Numerical Reproducibility" (CNR) in its Math Kernel Library (MKL). Although, CNR guarantees reproducibility, it does not ensure correct rounding, meaning the accuracy is arguing. Additionally, the cost of obtaining reproducible results with CNR is high. For instance, for large arrays the MKL's summation with CNR is 85 − 93% slower than both the regular MKL's and our reproducible summation; the later two deliver comparable performance.

Demmel and Nguyen [7] recently introduced a family of algorithms for reproducible summation in floating-point arithmetic . These algorithms always return 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 [22,28] by using one single reduction step among nodes. Such an improvement yielded roughly 20% overhead on 1024 processors compared to the Intel MKL `dasum()`, but it shows roughly 3.4 times slowdown on 32 processors. Demmel and Nguyen have extended their concept to reproducible BLAS routines, distributed in their ReproBLAS library. Arteaga et al. [23] used Demmel's 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 better performance results (the overhead is within 10%) compared to the conventional summation in case of the single reduction algorithm.

Neal [29] proposes a multistage accumulator in order to improve the accuracy of the sample mean computation in R. It uses "small" and "large" superaccumulators. The small superaccumulators are made of 6764−bit integer digits. The large superaccumulators consist of one 64−bit digit per possible combination of sign and exponent bit, which corresponds to 4096 64−bit plus an additional 4016 16−bit numbers to keep track of the amount of added values per digit. Each digit uses a carry-save representation similarly to what has been proposed in this article. This solution works well in its serial version, but no parallel

implementation has been designed yet. Perhaps, this is due to the large memory requirement of the large superaccumulator (more than 41KB).

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 datasets larger than 10^7 elements with dynamic ranges up to 10^{15}, the proposed approach solves the summation problem with similar performance to 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 sizes 2 and 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 is either comparable with or outperforms the other recently proposed reproducible summation algorithms – such as the Demmel's and Nguyen's algorithms implemented in the ReproBLAS and bitrep libraries – while offering correct rounding.

We followed a multi-level approach for correct rounding that is similar to the one used in elementary functions [30]. 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 ultimate goal is to design algorithms and provide corresponding implementations to ensure both accurate and reproducible results of fundamental linear algebra operations – like those included in the BLAS library – on a set of modern parallel architectures from desktop and server processors to Intel Xeon Phi co-processors and GPU accelerators. Moreover, we plan to conduct a priori error analysis of the derived ExBLAS (Exact BLAS) routines.

We have started the ExBLAS project by addressing the common operation among the BLAS routines – (parallel) summation. In order to construct exact (meaning accurate, reproducible, and fast) routines that involve multiplication, e.g. dot product, we would need to integrate exact multiplication into the current exact summation approach. Such multiplication could require only 2 flops if the FMA are supported, which is the case on modern architectures. Matrix-vector and matrix-matrix products can be virtually seen as a chain of dot products and, of course, blocking. Triangular solvers can also be interpreted in the similar manner except from the division by diagonal elements, which is really an issue. We have followed this approach and obtained preliminary results for triangular solver [31] as well as matrix-matrix multiplication [32]. Although the performance of ExTRSV and ExGEMM lag behind their non-reproducible counterparts, their outputs are consistently reproducible and accurate, in terms of rounding-to-nearest in case of the later, independently of threads scheduling and data partitioning. To sum up, the parallel exact summation is the crucial and common building block of the ExBLAS routines. More information on the ExBLAS project as well as the sources can be found in [33].

Acknowledgment

This work undertaken 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 ANR as part of the "Investissements d' Avenir" program.

This work was also supported by the FastRelax project through the ANR public grant (reference: ANR-14-CE25-0018-01).

References

[1] D.H. Bailey, R. Barrio, J.M. Borwein, High-precision computation: mathematical physics and dynamics, Appl. Math. Comput. 218 (20) (2012) 10106–10121, doi:10.1016/j.amc.2012.03.087.

[2] K. Bergman, al., Exascale computing study: Technology challenges in achieving exascale systems, 2008, (DARPA Report).
[3] N. Whitehead, A. Fit-Florea, Precision & performance: Floating point and IEEE 754 compliance for NVIDIA GPUs, Technical Report, NVIDIA, 2011.
[4] M. Corden, Differences in Floating-Point Arithmetic Between Intel® Xeon® Processors and the Intel® Xeon Phi™ Coprocessor, Technical Report, Intel, 2013.
[5] A. Katranov, Deterministic Reduction: a new Community Preview Feature in Intel® Threading Building Blocks, Technical Report, Intel, 2012.
[6] K. Doertel, Best Known Method: Avoid heterogeneous precision in control flow calculations, Technical Report, Intel, 2013.
[7] J. Demmel, H.D. Nguyen, Fast reproducible floating-point summation, in: Proceedings of the 21st IEEE Symposium on Computer Arithmetic, Austin, Texas, USA, 2013, pp. 163–172, doi:10.1109/ARITH.2013.9.
[8] IEEE Computer Society, IEEE standard for floating-point arithmetic, IEEE Standard 754-2008, 2008. Available at http://ieeexplore.ieee.org/servlet/opac?punumber=4610933.
[9] N.J. Higham, Accuracy and stability of numerical algorithms, second ed., Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2002, doi:10.1137/1.9780898718027.
[10] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, S. Torres, Handbook of Floating-Point Arithmetic, Birkhäuser Boston, 2010, doi:10.1007/978-0-8176-4705-6.
[11] D.E. Knuth, The Art of Computer Programming, Volume 2: Seminumerical Algorithms, third ed., Addison-Wesley, 1997.
[12] X.S. Li, J.W. Demmel, D.H. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, S.Y. Kang, A. Kapur, M.C. Martin, B.J. Thompson, T. Tung, D.J. Yoo, Design, implementation and testing of extended and mixed precision BLAS, ACM Trans. Math. Softw. 28 (2) (2002) 152–205, doi:10.1145/567806.567808.
[13] Y. Hida, X.S. Li, D.H. Bailey, 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, pp. 155–162, doi:10.1109/ARITH.2001.930115.
[14] U. Kulisch, V. Snyder, The exact dot product as basic tool for long interval arithmetic, Computing 91 (3) (2011) 307–313, doi:10.1007/s00607-010-0127-7.
[15] R. Shams, R. Kennedy, 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, pp. 418–422.
[16] G. Bohlender, U. Kulisch, Comments on fast and exact accumulation of products, in: K. Jónasson (Ed.), Applied Parallel and Scientific Computing, Lecture Notes in Computer Science, 7134, Springer, 2012, pp. 148–156, doi:10.1007/978-3-642-28145-7_15.
[17] D. Defour, F. De Dinechin, 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] S. Boldo, G. Melquiond, Emulation of a FMA and correctly rounded sums: proved algorithms using rounding to odd, IEEE Trans. Comput. 57 (4) (2008) 462–471, doi:10.1109/TC.2007.70819.
[19] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, P. Zimmermann, MPFR: a multiple-precision binary floating-point library with correct rounding, ACM Trans. Math. Softw. 33 (2) (2007) 13, doi:10.1145/1236463.1236468.
[20] MPFR Dev Team, The GNU MPFR Library, Available via the WWW. Cited 28 May 2014. http://www.mpfr.org.
[21] J. Reinders, Intel Threading Building Blocks, first ed., O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2007.
[22] J. Demmel, H.D. Nguyen, Parallel reproducible summation, IEEE Trans. Comput., 64 (7) (2015) 2060–2070.
[23] A. Arteaga, O. Fuhrer, T. Hoefler, Designing bit-reproducible portable high-performance applications, in: Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium, in: IPDPS '14, IEEE Computer Society, Washington, DC, USA, 2014, pp. 1235–1244, doi:10.1109/IPDPS.2014.127.
[24] J.D. McCalpin, Is "ordered summation" a hard problem to speed up?, 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/.
[25] J.R. Shewchuk, Robust adaptive floating-point geometric predicates, in: Proceedings of the Twelfth Annual Symposium on Computational Geometry, ACM, 1996, pp. 141–150, doi:10.1145/237218.237337.
[26] S.M. Rump, Ultimately fast accurate summation, SIAM J. Sci. Comput. 31 (5) (2009) 3466–3502, doi:10.1137/080738490.
[27] Y.-K. Zhu, W.B. Hayes, Algorithm 908: online exact summation of floating-point streams, ACM Trans. Math. Softw. 37 (3) (2010) 37:1–37:13, doi:10.1145/1824801.1824815.
[28] J. Demmel, H.D. Nguyen, Numerical reproducibility and accuracy at ExaScale (invited talk), in: Proceedings of the 21st IEEE Symposium on Computer Arithmetic, Austin, Texas, USA, 2013, pp. 235–237, doi:10.1109/ARITH.2013.43.
[29] R.M. Neal, Fast exact summation using small and large superaccumulators, Technical Report, University of Toronto, 2015.
[30] CR-Libm, CR-Libm – a library of correctly rounded elementary functions in double-precision., 2007, Available via the WWW. Cited 28 May 2014. http://lipforge.ens-lyon.fr/www/crlibm/.
[31] R. Iakymchuk, D. Defour, S. Collange, S. Graillat, Reproducible triangular solvers for high-performance computing, in: Proceedings of the 12th International Conference on Information Technology - New Generations (ITNG), 2015, pp. 353–358.
[32] R. Iakymchuk, D. Defour, S. Collange, S. Graillat, Reproducible and Accurate Matrix Multiplication for GPU Accelerators, Technical Report, hal-01102877, LIP6, ICS, DALI–LIRMM, INRIA, 2015.
[33] R. Iakymchuk, S. Collange, D. Defour, S. Graillat, ExBLAS – Exact BLAS, https://exblas.lip6.fr/.