Full-Speed Deterministic Bit-Accurate Parallel Floating-Point Summation on Multi- and Many-Core Architectures

Caroline Collange, David Defour, Stef Graillat, Roman Iakymchuk

Full-Speed Deterministic Bit-Accurate Parallel Floating-Point Summation 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 thus non-reproducible mainly due to non-associativity of floating-point operations. We introduce a solution to compute deterministic sums of floating-point numbers efficiently and with the best possible accuracy. Our multi-level algorithm consists of two main stages: a filtering stage that uses fast vectorized floating-point expansions; an accumulation stage based on superaccumulators in a high-radix carry-save representation. We present implementations on recent Intel desktop and server processors, on Intel Xeon Phi accelerator, and on AMD and NVIDIA GPUs. We show that the 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, long accumulator, multi-precision, multi- and many-core architectures.

1 Introduction

The increasing power of current computers enables one to solve more and more complex problems. That, consequently, leads to a higher number of floating-point operations to be performed, each of them causing a round-off error. Because of 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 [1]

- Quantum field theory;
- Supernova simulation;
- Semiconductor physics;
- Planetary orbit calculations.

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 will be a challenge.

Floating-point addition is non-associative and parallel reduction involving this operation is a serious issue as noted in the DARPA Exascale Report [2]. Such large summations typically appear within fundamental numerical blocks such as dot products or numerical integrations. Hence, the result may vary from one parallel machine to another or even from one run to another. These discrepancies worsen on heterogeneous architectures – such as clusters with GPUs or accelerators like Xeon Phi – which combine programming environments that may obey various floating-point models and offer different intermediate precision or different operators [3,4]. For instance, Intel acknowledges 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™ coprocessors, even for fixed numbers of threads or for serial code"* [5]. 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 issues of accuracy and reproducibility in the context of summation. We advocate to compute the correctly-rounded result of the exact sum. Besides offering strict reproducibility through an unambiguous definition of the expected result, our approach guarantees that the result has the best possible accuracy. However, the general use of correctly rounded sums has been considered impracticable since computing the exact sum 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*; 2. The overhead can be made negligible on large sums with low dynamic range using vectorized floating-point expansions.

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

2 Background

Floating-point (FP) arithmetic consists in approximating real numbers with a significand, an exponent, and a sign. The IEEE-754 standard, which was revised in 2008 [8], specifies floating-point formats and operations. In this paper, we consider the `binary64` or double precision format, although our strategy applies to the other formats. Floating-point allows to cover a wide *dynamic range* with nearly-constant precision. 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, as `binary64` can represent positive numbers from 4.9×10^{-324} to 1.8×10^{308}, it covers a dynamic range of 3.7×10^{631}.

Non-associativity of floating-point addition, which occurs due to the rounding error while performing the addition of numbers with different exponents, leads to the elimination of the lower bits of the sum. In contrast, the subtraction between numbers of the same sign and the same 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 information can be found in the main references for floating-point arithmetic [9,10].

Two approaches exist to perform floating-point addition without incurring round-off errors. The first solution aims at computing the error which occurred during rounding using an error-free transformation (see Section 2.1). The second solution exploits the finite range of representable floating-point numbers by storing every bit in a very long vectors of bits (see Section 2.2).

2.1 Floating-Point Expansion

In order to perform summations exactly, one has to recover errors which occur while rounding and keep track of them. FP *expansions* represent the result as an unevaluated sum of a fixed number of FP numbers, whose components are ordered in magnitude with minimal overlap to cover a wide range of exponents. Floating-point expansions of size 2 and 4 are described in [11] and [12], accordingly. They are based on an error-free transformation (EFT). The traditional EFT for the addition is 2Sum [13], depicted on Alg. 1. The 2Sum algorithm relies only on FP additions and subtractions. Adding one FP number to an expansion is an iterative operation. The FP number is first added to the head of the expansion and the rounding error is recovered as a floating-point number using an EFT such as 2Sum. The error is then recursively accumulated to the remainder of the expansion.

Algorithm 1 2Sum

$r \leftarrow a + b$
$z \leftarrow r - a$
$s \leftarrow (a - (r - z)) + (b - z)$

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 accuracy as long as every intermediate result can be represented

exactly as a sum of n FP numbers. This situation happen when the dynamic range of the sum is lower than $2^{52 \cdot n}$ (for binary64).

The main advantage of this solution is that the expansion can stay in registers during the computations. However, the accuracy is insufficient for the summation of numerous FP numbers or sums with a huge dynamic range. Moreover, their complexity grows linearly with the size of the expansion.

2.2 Superaccumulator

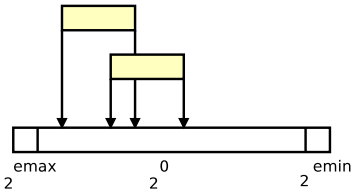


Fig. 1: Kulisch long accumulator.

An alternative to expansions is to use a very long fixed-point accumulator. The length of the accumulator is chosen such that every bit of information of the input format can be represented (`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 the accumulation of products of 64-bit IEEE floating-point values. The addition is performed without loss of information by accumulating every floating-point input number in the long accumulator, see Fig. 1. The Kulisch accumulator is a 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 make vectorization challenging.

3 Hierarchical Superaccumulation Scheme

Most prior work on exact accumulation have been targeting ill-conditioned problems that could not be satisfied using the accuracy provided by the conventional floating-point accumulation. Our objective is different – we are concerned about computing a deterministic and accuracy-guaranteed result for sums that are typically evaluated using floating-point arithmetic today. We expect such sums to be well-conditioned and cover a moderate dynamic range.

We follow a multi-level approach shown in Fig. 2. The accumulation of FP numbers is split into five levels. This decomposition is suitable for the nested parallelism of modern architectures. It makes full use of SIMD, multi-thread, and multi-core parallelism.

The first level consists of FP expansions to filter large numbers, which are expected to be the major part of all inputs. Each thread maintains its own set of expansions. As tens to thousands of threads running in parallel use their private registers to store FP expansions, this solution will speed-up computations as well as avoiding memory accesses.

The second level is only invoked whenever the accuracy provided by expansions is not enough. Threads will send the 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, to private superaccumulators. Private superaccumulators are stored in a fast local memory like cache or shared memory depending on the architecture. They will contain exact partial sums.

During the third step, inside each processor, private superaccumulators are merged into a single scalar superaccumulator. In the fourth level, all scalar superaccumulators are combined together into one single superaccumulator accessible by all threads. This step is performed by using a standard parallel reduction. 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 the proposed algorithm at the levels 2-4 has several concepts in common with the histogram problem [15].

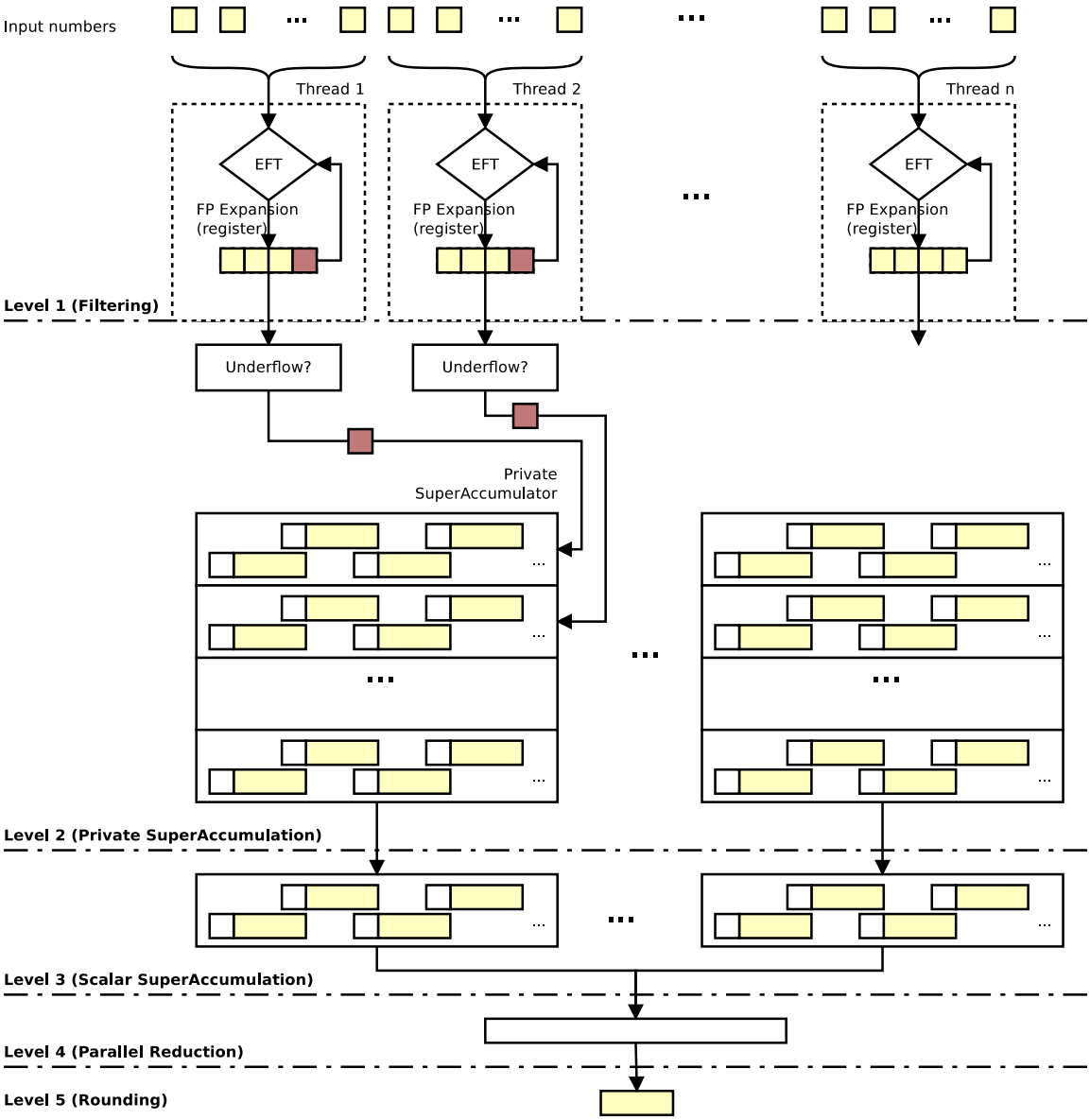


Fig. 2: Five-level summation.

3.1 Level 1: Filtering

We propose to extend the classical approach described in Section 2.1 in order to maintain expansions of n floating-point numbers, see Alg. 2. This algorithm is suitable for vectorization and pipelining by maintaining multiple expansions in parallel.

Underflow occurs when the result of the accumulation cannot be represented as a floating-point expansion of n numbers. It can be detected by checking if the last rounding error x is non-zero; each non-zero residue is covered by the scalar superaccumulator, see Section 3.2. We also propose a version of this algorithm that consists in stopping the accumulation loop as soon as the residue is zero ($x = 0$). This technique is named *early-exit*. Its performance depends on two factors: the distribution of input numbers and the ability of the architecture to handle irregular branches.

The design challenge of the first step is to select a suitable value of n. A large value of n will lead to large expansions that reduce transfers to the superaccumulator, but increase register usage. A small value of n will impose a limited representation range for the expansion, driving more transfers towards the superaccumulator but

Algorithm 2 ExpansionAccumulate(x)

for $i = 0$ **to** $n - 1$ **do**
  $(a_i, x) \leftarrow 2\mathrm{Sum}(a_i, x)$
end for
if $x \neq 0$ **then**
  Superaccumulate(x)
end if

reducing register pressure. In either case, all components of the floating-point expansion are accumulated into the superaccumulator at the end of the summation.

3.2 Level 2 and 3: Scalar Superaccumulator

We follow the general strategy of the Complete Register implementation proposed by Bohlender and Kulisch [16] and adapt it for a software use. This strategy allows to accumulate FP numbers in amortized $O(1)$ time regardless of the length of the long accumulator; it 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 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, delaying the carry propagation for 2^M accumulation steps. Unlike Bohlender and Kulisch's strategy, 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. 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 lowest and the largest binary floating-point exponents, respectively. In our implementation, we choose $w = 56$ and $M = 8$.

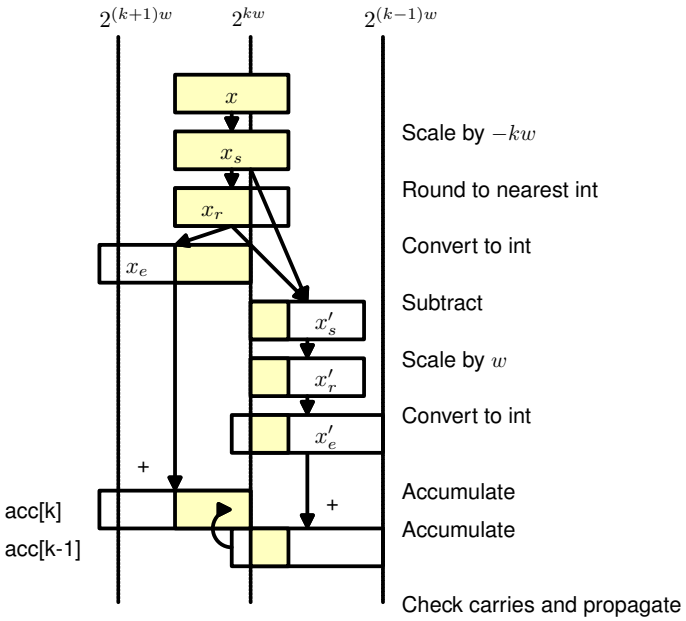


Fig. 3: Scalar accumulation scheme.

The use of 64-bit integer arithmetic allows the digit size w to be greater or equal than the floating-point significand size (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. 3. This algorithm is based on rounding scaled floating-point numbers. First, the exponent of the input number x is used to compute the indexes k and $k-1$ of the two digits of 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 superaccumualtor. When the result of the accumulation exceeds the range of the accumulator word, a partial normalization is performed along the propagation of the carry or borrow. In this case, the upper M bits of each digit are transferred to the next digit until the carry propagation chain stops. The carry or borrow propagation is only needed at most every 2^{M-1} accumulation steps and typically requires only a single step. At the end of the summation, a complete normalization pass propagates all carries.

3.3 Level 4 and 5: Reduction and Rounding

Once all input numbers are accumulated into local superaccumulators, those superaccumulator are combined together. We use the classic reduction algorithm, albeit with a larger payload than customary. Reduction is a common parallel primitive available in highly optimized version on various system 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 bits of carry enable a normalization-free reduction across up to 256 threads.

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 word $k-2$. The sticky bit is used to discriminate between a number that falls exactly between two FP 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, which is based on rounding to odd [18].

4 Implementations and Results

This section details our implementations of the multi-level scheme and evaluate them on the whole range of parallel platforms 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 one produced by the multiple precision library MPFR [19].

Table 1: Hardware platforms employed in the experimental evaluation.

A	Intel Core i7-4770 (Haswell)	4 cores with Hyper-Threading	3.400 GHz
B	Intel Xeon E5-2450 (Sandy Bridge-EN)	2 × 8 cores	2.890 GHz
C	Intel Xeon Phi 3110P	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

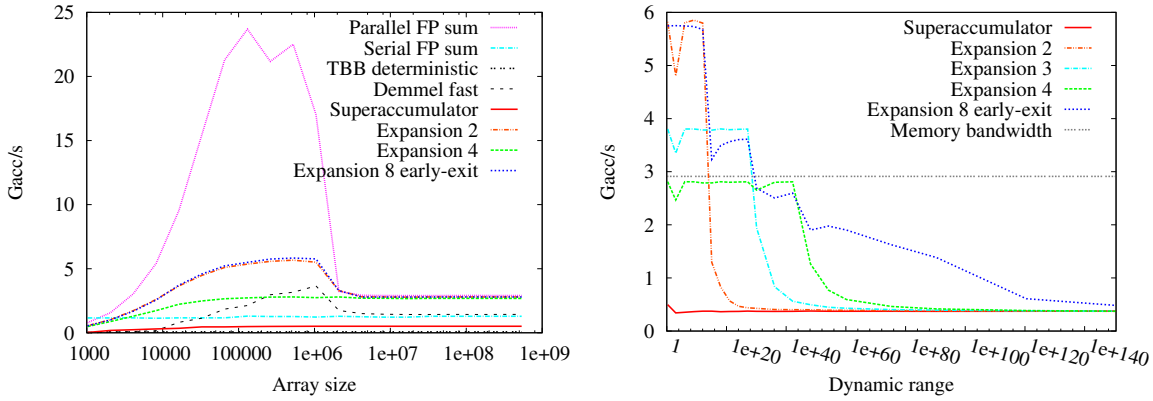
As baselines, we consider:

1. A vectorized and parallelized non-deterministic reduction;
2. The deterministic FP reduction provided by the Intel TBB library [20];
3. The Fast Deterministic Parallel Sum algorithm proposed by Demmel and Nguyen [7].

4.1 Implementation

Our implementations strive to use all resources of modern processors: SIMD instructions, fused multiply-add, and multi-threading on multi-core CPUs and Xeon Phi; local memory and atomic instructions on GPUs.

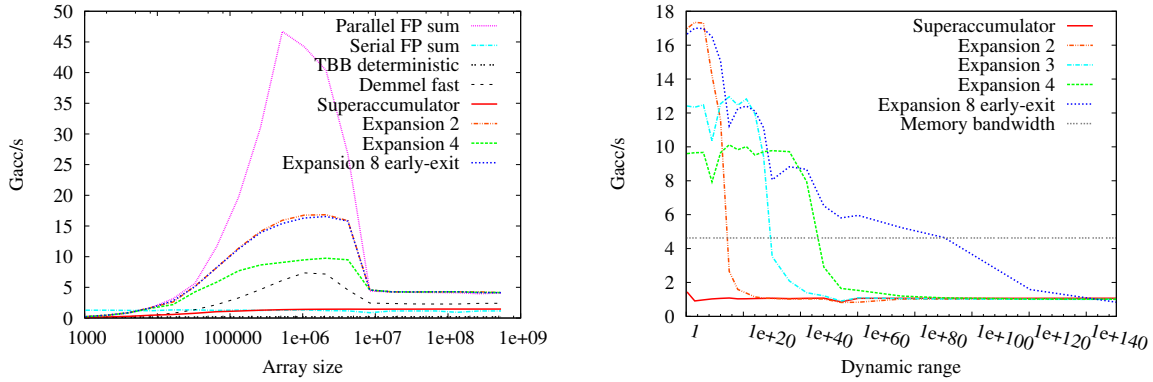
The Haswell micro-architecture offers a FMA unit that can run in parallel with the FP adder. Hence, we use AVX intrinsics to benefit from the 4-way SIMD and implement 2Sum by replacing half of the 6 additions/subtractions



(a) A function of dataset size.

(b) A function of input range; $N = 2e06$.

Fig. 4: The summation performance on the platform A (Intel Core i7-4770).



(a) A function of dataset size.

(b) A function of input range; $N = 2e06$.

Fig. 5: The summation performance on the platform B (Dual Intel Xeon E5-2450)

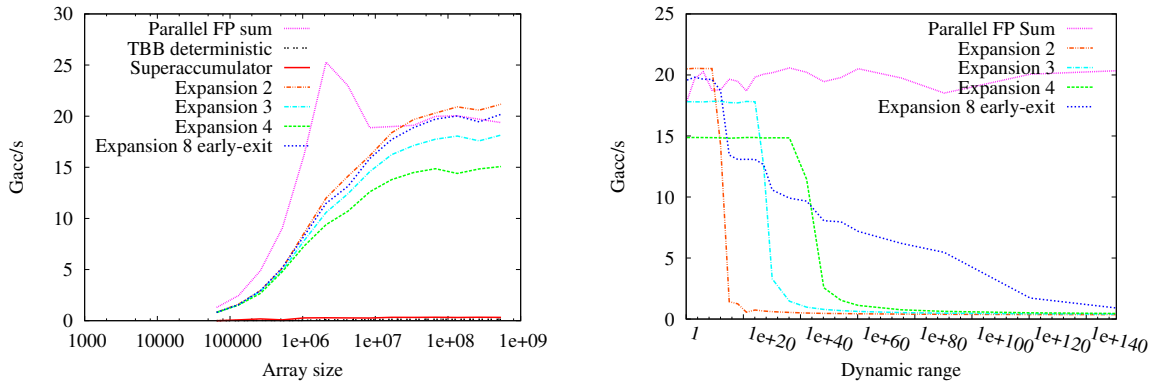
by FMAs (multiplying by 1). The latter optimization allows to use both the FP adder and FMA units which can run in parallel on the Haswell micro-architecture; therefore, each core can execute two SIMD additions per cycle. Thread-level parallelism is exposed using OpenMP in order to take advantage of multi-core and hardware multi-threading.

On the Intel Xeon Phi 3110P 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 a single OpenCL implementation for both NVIDIA and AMD GPUs. Our implementation benefits from the classic histogram algorithm – use 16 superaccumulators per work group that are located in local memory – and the reduction algorithm for merging private and scalar superaccumualtors. We apply atomic operations to resolve the concurrency between threads, which share a single superaccumualtor, while scattering the input data into the proper digits within a superaccumualtor. In order to avoid bank conflicts, superaccumulators are interleaved together to spread their digits among different memory banks. From the conducted experiments, we notice that the configuration of 2048 working groups with 256 threads delivers the best performance.

4.2 Performance scaling

Figs. 4a, 5a, 6a, 7a and 8a 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 on each platform. On large datasets, the performance of the baseline unordered sum as well as the expansion cache is constrained by memory bandwidth on all platforms. On the platform A , both algorithms achieve 23.3 GB/s over the theoretical peak bandwidth of 25.6 GB/s. Indeed, the peak double-precision performance of this CPU is 218 GFlops. This



(a) A function of dataset size.

(b) A function of input range; $N = 67e06$.

Fig. 6: The summation performance on the platform C (Intel Xeon Phi)

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 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 the platform A and 136 Macc/s on the 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 the platform A. The results on the platform B show a similar trend as on the platform A, albeit with the higher ratio between the computations and the memory transfers.

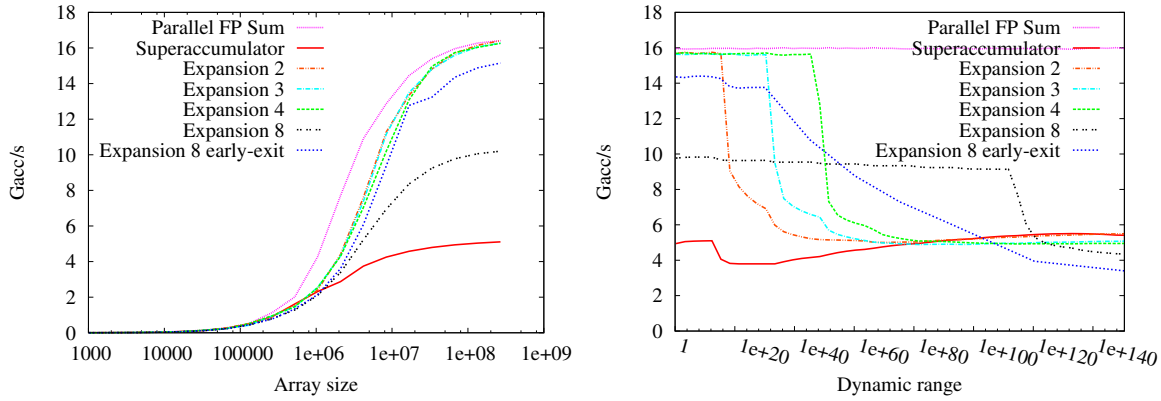
Due to the memory allocation limit on the platforms D and E, the maximum array size used is $268e06$ and $67e06$, respectively. On both platforms, the peak performance is reached for problems of size $67e06$ and larger, where it says as a plateau. The best achieved efficiency is roughly 63 % and 73 % on NVIDIA and AMD GPUs, accordingly.

As Figs. 4 to 8 show, the implementation that is solely based on the superaccumualtor delivers the worst performance. This is because superaccumualtors are placed in the local memory that increases the access time significantly, especially during the second phase, see Fig. 2. In contrast, the combination of superaccumualtor and various expansions, also with the early-exit optimization, performs much better since their expansions are stored in private memory, which requires only few cycles to access and add elements. Furthermore, the implementations with expansions do not use atomic operations that, therefore, improves the performance. To sum up, for large arrays our approach with either 2-, 3-, 4-, or 8-word expansions and the early-exit optimization reaches the same performance as the parallel unordered FP summation on all five platforms.

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 since the exponents of FP numbers follow the uniform distribution. As running experiments on large datasets clamps all curves below the memory bandwidth limit on CPUs, we present results on the small datasets with 2 millions of elements in order to highlight the difference between implementations.

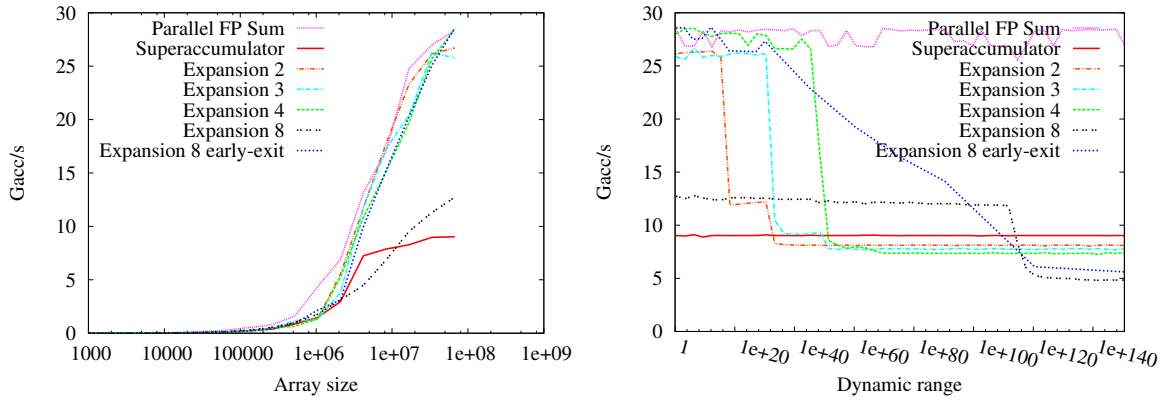
Figs. 4b, 5b, 6b, 7b and 8b represent the scenario when the array size is fixed, but the range of its elements varies. Floating-point expansions adapt dynamically to the range of input numbers and maintain a minimal overlap between words. On platforms A and B, the expansion cache with 8 words and the early-exit optimization saturates the memory bandwidth for dynamic ranges up to 10^{30} and 10^{90}, accordingly. On the platform C, the 2-word expansion delivers the sustainable performance for dynamic ranges up to 10^{15}. On both platforms D and E, the 4-word expansion resists the memory bandwidth for dynamic ranges up to 10^{50}. As a reference for the comparison, the estimated diameter of the Universe over Planck's length is about 1.4×10^{62} [21]. In the worst case of sums with tremendous dynamic range, our method can be from 3 (on the platform E) to 44 times (on the platform C) slower than the non-deterministic sum. However, it is still capable to reach 64% of the performance of the Demmel's fast reproducible sum and to exceed the performance of the serial sum.



(a) A function of dataset size.

(b) A function of input range; $N = 67e06$.

Fig. 7: The summation performance on the platform D (NVIDIA Tesla K20c)



(a) A function of dataset size.

(b) A function of input range; $N = 67e06$.

Fig. 8: The summation performance on the platform E (AMD Radeon HD 7900)

5 Related Works

Many techniques have been proposed to increase the accuracy of FP summation. One of them is based on the usage of a long accumulator, see Section 2.2. This solution is suitable for hardware implementations owing to constant-time accumulation [16]. The analysis of its cost, which has first appeared in the article [22] by John D. McCalpin, advocates its usage on new architectures, but without tests and implementations.

Another approach to exact summation consists in pure FP algorithms. The idea behind is to store the exact sum as an expansion or an unevaluated sum of FP numbers [23,24]. However, as we mentioned early, this solution is design solely to increase the accuracy of basic operations, but neither to reach bit-accurate results nor to be efficient for large precision. Therefore, some recent works are focusing on hybrid solutions that store the sum as FP numbers of fixed exponent [25,26] without completely avoiding previous drawbacks. Those algorithms are mainly sequential and not suitable for parallel implementation. This is mainly due to the need to scan all the numbers before beginning the computation of the sum. Arbitrary precision libraries – like MPFR [19] – are able to provide correct rounding. But, they are not design to achieve acceptable performance for reproducible results – 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 MKL (Math Kernel Library). However, CNR is slow and does not give any guarantee on the accuracy of the result. Demmel and Nguyen recently introduced an algorithm for reproducible summation in floating-point arithmetic [7]. This algorithm always returns the same answer. It first computes an absolute bound of the sum and then rounds 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 [27]. As Section 4 shows, our algorithm is faster in the bandwidth-constrained scenarios with moderate dynamic ranges. The authors also provide some reproducible BLAS routines in their ReproBLAS library[5].

6 Conclusions

We presented a solution to achieve correct rounding for the floating-point summation problem, along with implementations on multi- and many-core architectures. It yields results that are both reproducible and as accurate as possible. For dataset larger than 10^7 elements with dynamic ranges up to 10^{15}, the proposed approach solves the summation problem with the performance comparable to the classic highly optimized parallel FP summations; however, the parallel FP summation provides neither correct rounding nor reproducibility. These competitive results were achieved using the expansion 4 on both multi-core platforms as well as GPUs and the expansion 8 with the early-exit optimization on Xeon Phi for low to medium dynamic ranges; such dynamic ranges correspond to the most common cases occurred in practice. Thanks to the multi-level strategy, we were able to deliver the bit-accurate results for large dynamic range at 25% of parallel FP summations performance on multi-core and GPUs platforms. Furthermore, we demonstrated that the multi-level correct summation outperforms the other recently proposed reproducible summation algorithms – like the Demmel's and Nguyen's algorithm – and, moreover, it offers 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 that are more expensive but 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 the 64-bit arithmetic and conversions in SIMD instruction sets, the fast floating-point exponent extraction, and the fast memory gather and scatter would further improve the performance of the superaccumulation.

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).

References

1. Bailey, D.H., Barrio, R., Borwein, J.M.: High-precision computation: Mathematical physics and dynamics. Applied Mathematics and Computation **218**(20) (2012) 10106–10121
2. Bergman, K., al.: Exascale computing study: Technology challenges in achieving exascale systems. DARPA Report (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 (2013)
5. Katranov, A.: Deterministic reduction: a new community preview feature in Intel® Threading Building Blocks. Technical report, Intel (2012)
6. Doertel, K.: Best known method: Avoid heterogeneous precision in control flow calculations. Technical report, Intel (2013)
7. Demmel, J., Nguyen, H.D.: Fast reproducible floating-point summation. In: 21st IEEE Symposium on Computer Arithmetic, Austin, Texas, USA. (2013) 163–172

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

8. IEEE: IEEE standard for floating-point arithmetic. IEEE Std 754-2008 (2008)
9. Higham, N.J.: Accuracy and stability of numerical algorithms. Second edn. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA (2002)
10. Muller, J.M., al.: Handbook of floating-point arithmetic. Birkhäuser (2010)
11. Li, X.S., al.: Design, implementation and testing of extended and mixed precision BLAS. ACM Transactions on Mathematical Software **28**(2) (2002) 152–205
12. Hida, Y., Li, X.S., Bailey, D.H.: Algorithms for quad-double precision floating point arithmetic. In: Proc. 15th IEEE Symposium on Computer Arithmetic, IEEE Computer Society Press, Los Alamitos, CA, USA (2001) 155–162
13. Knuth, D.E.: The Art of Computer Programming: Seminumerical Algorithms. Addison-Wesley (1997) Third edition.
14. Kulisch, U., Snyder, V.: The exact dot product as basic tool for long interval arithmetic. Computing **91**(3) (2011) 307–313
15. Shams, R., Kennedy, R.: Efficient histogram algorithms for nvidia cuda compatible devices. Proc. Int. Conf. on Signal Processing and Communications Systems (ICSPCS) (2007) 418–422
16. Bohlender, G., Kulisch, U.: Comments on fast and exact accumulation of products. In: Applied Parallel and Scientific Computing. Springer (2012) 148–156
17. Defour, D., de Dinechin, F.: Software carry-save for fast multiple-precision algorithms. In: 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 `http://www.mpfr.org`.
20. Reinders, J.: Intel Threading Building Blocks. First edn. O'Reilly & Associates, Inc., Sebastopol, CA, USA (2007)
21. Defour, D., de Dinechin, F., Muller, J.M.: Correctly rounded exponential function in double precision arithmetic. Technical Report RR2001-26, LIP, École Normale Supérieure de Lyon (July 2001) Available at ftp://ftp.ens-lyon.fr/pub/LIP/Rapports/RR/RR2001/RR2001-26.ps.Z.
22. McCalpin, J.D.: Is "ordered summation" a hard problem to speed up? `http://blogs.utexas.edu/jdm4372/2012/02/15/`.
23. Priest, D.M.: Algorithms for arbitrary precision floating point arithmetic. In: 10th IEEE Symposium on Computer Arithmetic, IEEE (1991) 132–143
24. Shewchuk, J.R.: Robust adaptive floating-point geometric predicates. In: Proceedings of the twelfth annual symposium on Computational geometry, ACM (1996) 141–150
25. Rump, S.M.: Ultimately fast accurate summation. SIAM Journal on Scientific Computing **31**(5) (2009) 3466–3502
26. Zhu, Y.K., Hayes, W.B.: Algorithm 908: Online exact summation of floating-point streams. ACM Transactions on Mathematical Software (TOMS) **37**(3) (2010) 37
27. Demmel, J., Nguyen, H.D.: Numerical reproducibility and accuracy at ExaScale (invited talk). In: 21st IEEE Symposium on Computer Arithmetic, Austin, Texas, USA. (2013)
28. CR-Libm: CR-Libm – a library of correctly rounded elementary functions in double-precision.