Accurate and Efficiently Vectorized Sums and Dot Products in Julia

Chris Elrod, François Févotte

Accurate and Efficiently Vectorized Sums and Dot Products in Julia

Chris Elrod
Baylor University
Waco, TX, USA
Email: elrodc@gmail.com

François Févotte
TriScale innov
Palaiseau, France
Email: ff@triscale-innov.com

***Abstract*—This paper presents an efficient, vectorized implementation of various summation and dot product algorithms in the Julia programming language. These implementations are available under an open source license in the `AccurateArithmetic.jl` Julia package.**

Besides naive algorithms, compensated algorithms are implemented: the Kahan-Babuška-Neumaier summation algorithm, and the Ogita-Rump-Oishi simply compensated summation and dot product algorithms. These algorithms effectively double the working precision, producing much more accurate results while incurring little to no overhead, especially for large input vectors.

This paper also tries and builds upon this example to make a case for a more widespread use of Julia in the HPC community. Although the vectorization of compensated algorithms is no particularly simple task, Julia makes it relatively easy and straightforward. It also allows structuring the code in small, composable building blocks, closely matching textbook algorithms yet efficiently compiled.

***Index Terms*—Vectorization, Compensated Algorithms, Julia programming language**

I. Introduction

Computing the dot product of two vectors and, perhaps to a lesser extent, summing the elements of a vector, are two very common basic building blocks for more complex linear algebra algorithms. As such, any change in their performance is likely to affect the overall performance of scientific computing codes; any change in their accuracy is likely to induce a loss of reproducibility in overall computed results.

Such a lack of reproducibility is becoming more and more common as the performance of supercomputers increase, due to a combination of multiple factors. First, an increase in computational power allows for ever more expensive models (such as, for example, finer grids for the discretization of partial differential equations), which in turn generates larger vectors, whose summation is often more ill-conditioned. Second, the order in which operations are performed in modern systems is far from being deterministic and reproducible: vectorization, parallelism, and compiler optimizations are only a few examples of possible causes for unexpected changes in the execution order of a given program. Since floating-point (FP) arithmetic is not associative, such non-determinism in execution order entails non-reproducibilities in computed results. For these reasons, accurate summation and dot product algorithms are sometimes useful.

However, increasing the algorithms accuracy at the price of an increase in their computing cost is not acceptable in all cases. For programs that already use double precision (64-bit FP numbers), increasing the precision is not an option, since quadruple precision (128-bit FP numbers) remains too expensive for most purposes[1]. On the other hand, hitting the memory wall means that many algorithms (and in particular BLAS1-like algorithms such as summation and dot product) are limited by the memory bandwidth in a wide range of vector lengths on modern systems. In turn, this means that CPUs are idling for a fraction of the computing time, which could instead be used to perform additional operations for free, as long as these operations do not involve additional memory accesses. Much work has therefore been devoted to computing sums and dot products in a way that is both fast and accurate, leading to efficient algorithms implemented in a variety of programming languages.

The work presented here comes from the observation[2] that, as of November 2018, no such algorithm was currently implemented in an efficient way in the Julia programming language. This paper describes an attempt made to implement accurate summation and dot product algorithms in Julia, in a way that is as efficient as possible, and in particular makes use of SIMD vector units. All the code described here is available under an open source license in the `AccurateArithmetic.jl` package. This paper will also try to make the argument that writing such efficient implementations of compensated algorithms is a relatively easy and straightforward task thanks to the nice properties of the Julia language.

Part II of this paper takes the viewpoint of computational sciences and discusses some aspects related to the use of vector instructions in fast implementations of summation algorithms. Part III then takes the point of view of floating-point arithmetic, and reviews a subclass of accurate summation and dot product algorithms: compensated algorithms. Nothing particularly new is presented in these parts, except maybe some specific details about interactions between vectorization and compensation (paragraph III-C), that have to be dealt with in order to gain the full benefit of both worlds: speed and

[1] at least on x86-64 architectures, where quadruple precision is implemented in software

[2] https://discourse.julialang.org/t/floating-point-summation/17785

Algorithm 1 Scalar, naive summation

1: {Initialization:}
2: $a \leftarrow 0$

3: {Loop on vector elements:}
4: **for** $i \in 1 : N$ **do**
5: $\quad a \leftarrow a \oplus x_i$
6: **end for**

7: **return** a

Algorithm 2 Vectorized, naive summation

1: {Initialization:}
2: $\mathbf{a} \leftarrow 0$

3: {Loop on full packs:}
4: **for** $j \in 1 : \left\lfloor \frac{N}{W} \right\rfloor$ **do**
5: $\quad i \leftarrow W(j-1)+1$
6: $\quad \mathbf{p} \leftarrow (x_i, x_{i+1}, x_{i+2}, x_{i+3})$
7: $\quad \mathbf{a} \leftarrow \mathbf{a} \oplus \mathbf{p}$
8: **end for**

9: {Reduction of SIMD accumulator:}
10: $a \leftarrow \mathbf{vsum}(\mathbf{a})$

11: {Loop on remaining elements:}
12: **for** $j \in W\left\lfloor \frac{N}{W} \right\rfloor + 1 : N$ **do**
13: $\quad a \leftarrow a \oplus x_i$
14: **end for**

15: **return** a

accuracy. Part IV gives some insight about the Julia implementation of such compensated algorithms, and discusses the performance obtained on some test systems. These results are summarized in part V, along with some closing notes and outlooks.

II. The computational science point of view

For the sake of simplicity, let us only discuss here the case of the summation algorithm:

$$\mathbf{sum}(x) = \sum_{i=1}^{N} x_i,$$

where $x \in \mathbb{R}^N$. Since a dot product can also be expressed as a summation, no loss of generality is entailed by this choice:

$$\mathbf{dot}(x, y) = \sum_{i=1}^{N} x_i \, y_i,$$

where $x \in \mathbb{R}^N$ and $y \in \mathbb{R}^N$.

Algorithm 1 presents the most naive possible algorithm for a summation, in its scalar version. An accumulator a stores partial sums. During each loop iteration, a new vector element x_i is added to this accumulator. Note that this accumulation is performed using the computer arithmetic addition operation $\oplus$, which operates on FP numbers, rounds its result, and is therefore only an approximation of the mathematical $+$ operator which operates on real numbers. In the remainder of this paper, similar notations will be used for other operators, such as $\ominus$ denoting the FP subtraction.

In Algorithm 2, this algorithm is refined to make use of vector instructions. In the following, the system will be assumed to efficiently implement SIMD instructions on vectors (hereafter referred to as "packs" so that the "vector" term can unambiguously refer to mathematical vectors such as the operand x) of width W. For `x86_64` architectures (which are the main target of the algorithms described here), the AVX extension for example supports efficiently operating on 256-bit registers holding $W = 4$ double-precision FP numbers. All variables denoted in bold font represent such packs, and are meant to be stored and manipulated as SIMD registers. The main loop in Alg. 2 (lines 4–8) now operates on a SIMD accumulator, in which W vector elements are added at once at each iteration. The number of iterations is therefore divided by W. In case the vector length is not a multiple of W, a remainder has to be accounted for. There are multiple ways of doing this; in this example, the remaining elements are added in a scalar loop. Before doing this, the SIMD accumulator has to be reduced to a scalar: this is the purpose of the **vsum** function, which sums all W components of a SIMD register:

$$\mathbf{vsum}(\mathbf{a}) = \mathbf{a}[1] \oplus \mathbf{a}[2] \oplus \ldots \oplus \mathbf{a}[W],$$

where $\mathbf{a}[i]$ represents the i-th element in a SIMD pack $\mathbf{a}$.

Although such an implementation is already much more complex than its scalar version, it is not enough to obtain optimal performance on modern hardware. Because of the loop-carried dependency of the accumulator and instruction latency in superscalar hardware architectures, full performance can only be obtained if multiple instructions can be running at the same time. This means breaking that sequential dependency between one iteration and the next; a traditional way to do that consists in partially unrolling the main loop. An illustration of this technique is given in Algorithm 3, where the loop is unrolled twice ($U = 2$). A consequence of this technique is that there are now as many SIMD accumulators as the level of partial unrolling U (2 in this case). These accumulators need to be summed together and reduced before proceeding with the accumulation of remaining scalar elements.

If the number of packs $\left\lfloor \frac{N}{W} \right\rfloor$ is not a multiple of U, an additional type of remainder needs to be handled: the full packs that were not handled in the partially unrolled loop. In the algorithm presented here, SIMD accumulator $\mathbf{a}_1$ is used to accumulate these.

Figure 1 illustrates the gains to be expected from such a technique: for various vector sizes, the total summation time (normalized by the vector size) is plotted against the unrolling level. The parameter given in the x-axis is the base-2 logarithm of the unrolling level U. The figure shows that a value $\log_2(U) = 3$ seems to be an optimal trade-off (on

Algorithm 3 Vectorized, naive summation with partial unrolling ($U = 2$ in this example)

{Initialization:}
$\mathbf{a}_1 \leftarrow 0$
$\mathbf{a}_2 \leftarrow 0$

{Loop on full packs, unrolled twice:}
for $j \in 1 : \left\lfloor \frac{N}{2W} \right\rfloor$ **do**
  $i_1 \leftarrow 2\,W(j-1)+1$
  $\mathbf{p}_1 \leftarrow (x_{i_1}, x_{i_1+1}, x_{i_1+2}, x_{i_1+3})$
  $\mathbf{a}_1 \leftarrow \mathbf{a}_1 \oplus \mathbf{p}_1$
  $i_2 \leftarrow 2\,W(j-1)+W+1$
  $\mathbf{p}_2 \leftarrow (x_{i_2}, x_{i_2+1}, x_{i_2+2}, x_{i_2+3})$
  $\mathbf{a}_2 \leftarrow \mathbf{a}_2 \oplus \mathbf{p}_2$
end for

{Loop on remaining full packs:}
for $j \in 2\left\lfloor \frac{N}{2W} \right\rfloor + 1 : \left\lfloor \frac{N}{W} \right\rfloor$ **do**
  $i \leftarrow W(j-1)+1$
  $\mathbf{p} \leftarrow (x_i, x_{i+1}, x_{i+2}, x_{i+3})$
  $\mathbf{a}_1 \leftarrow \mathbf{a}_1 \oplus \mathbf{p}$
end for

{Reduction of SIMD accumulators:}
$\mathbf{a_1} \leftarrow \mathbf{a}_1 \oplus \mathbf{a}_2$
$a \leftarrow \mathbf{vsum}(\mathbf{a}_1)$

{Loop on remaining elements:}
for $j \in W\left\lfloor \frac{N}{W} \right\rfloor + 1 : N$ **do**
  $a \leftarrow a \oplus x_i$
end for

return a

this particular system): this corresponds to unrolling the loop $U = 2^3 = 8$ times. Depending on the vector size, the gains achieved can be as large as a $7\times$ speed-up w.r.t. the version without any unrolling ($\log_2(U) = 0$).

III. The arithmetic point of view

A. Accuracy and condition number

The naive summation algorithm presented above produces results of relatively poor quality when used on ill-conditioned vectors. Throughout this paper, the computer arithmetic will be assumed to follow the IEEE-754 standard [6]. Under this hypothesis, the accuracy of the naive summation algorithm is given by

$$\left| \tilde{s}_{\text{naive}}(x) - \sum_{i=1}^{N} x_i \right| = K_N \, \epsilon \sum_{i=1}^{N} |x_i|,$$

where $\tilde{s}(x)$ denotes the result of naive summation on vector $x \in \mathbb{R}^N$, K_N is a constant depending only on the vector size N, and ϵ is the machine epsilon for the FP format used in the computation. All examples in this paper use double precision (known as `binary64` in IEEE-754), for which $\epsilon = 2^{-53} \simeq 1.11 \times 10^{-16}$. In other words, the relative error on the naive summation results verifies

$$\varepsilon_{\text{naive}}(x) = \left| \frac{\tilde{s}_{\text{naive}}(x) - \sum_{i=1}^{N} x_i}{\sum_{i=1}^{N} x_i} \right| = K_N \, C(x) \, \epsilon, \qquad (1)$$

where the condition number of the summation was introduced:

$$C(x) = \frac{\sum_{i=1}^{N} |x_i|}{\sum_{i=1}^{N} x_i}.$$

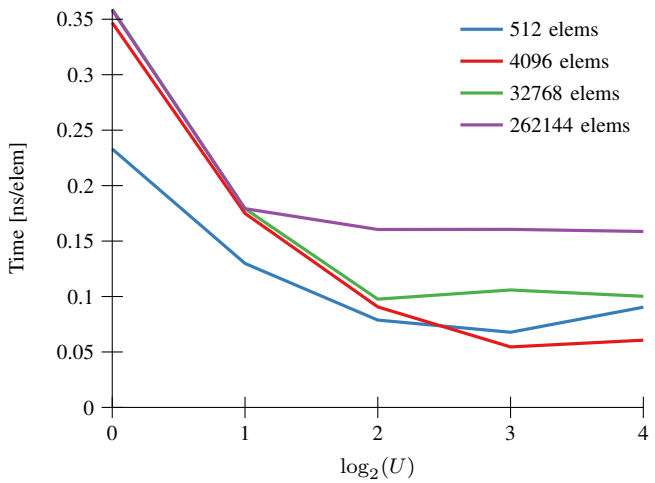
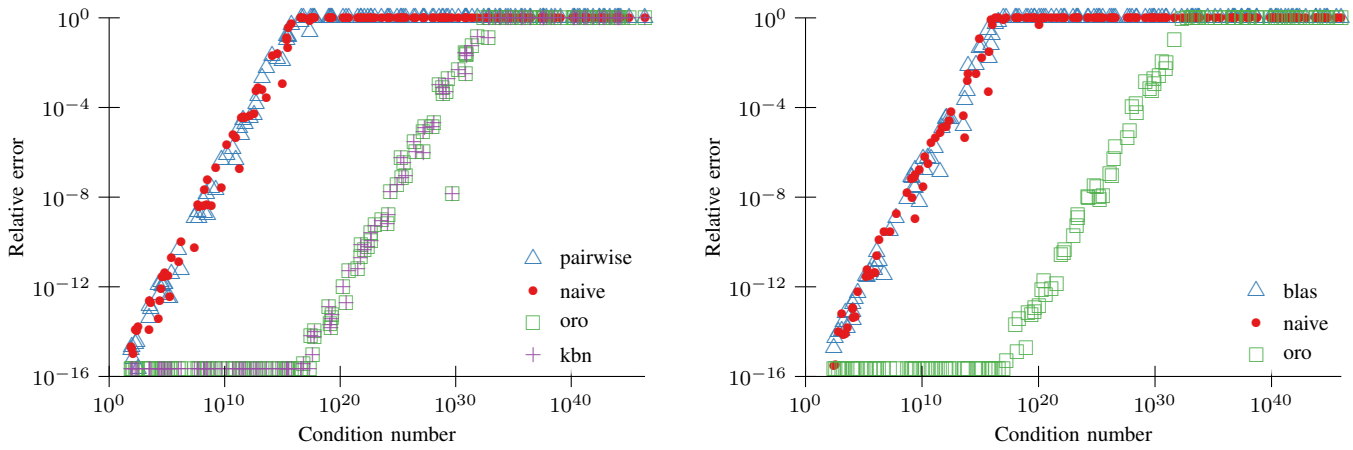


Fig. 1: Performance of a vectorized, naive summation as a function of the unrolling level. Computing times are normalized by the vector size.

Similar results can be obtained for the dot product. Following the line of reasoning introduced in [12], Figure 2 plots the relative error as a function of the condition number, in a log-log scale. Randomly generated input vectors of sizes ranging from 100 to 110 are produced by the algorithm described in [12] to feature condition numbers ranging from 1 to 10^{45}. In the figure, errors lower than ϵ are arbitrarily set to ϵ; conversely, when the relative error is more than 100% (i.e no digit is correctly computed anymore), the error is capped there in order to avoid affecting the scale of the graph too much. In Figure 2a, the "pairwise" line describes the accuracy of Julia's standard summation function (`Base.sum`), which uses a pairwise summation algorithm; the "naive" line corresponds to a naive summation implementation similar to the one described in Alg. 3 and available as `AccurateArithmetic.sum_naive`. Both algorithms behave as described by Equation (1): they start losing accuracy as soon as the condition number increases, computing only noise when the condition number exceeds $1/\epsilon \simeq 10^{16}$. A very similar behavior is observed in Figure 2b for the two naive dot product implementations: "blas" denotes Julia's default `LinearAlgebra.dot` function, which internally uses OpenBLAS's `ddot` implementation [13]; "naive" denotes a naive dot product implementation



(a) Summation (b) Dot product

Fig. 2: Comparison of the accuracy of various summation and dot product algorithms, as a function of the condition number

Algorithm 4 `fast_two_sum` error-free transform

Require: $(a, b) \in \mathbb{F}^2$, such that $|a| \geq |b|$
Ensure: $x = a \oplus b$ and $x + e = a + b$
  $x \leftarrow a \oplus b$
  $e \leftarrow (a \ominus x) \oplus b$

developed along the lines of Algorithm 3 and available as `AccurateArithmetic.dot_naive`.

B. Accurate sum and dot product

As said above, much work has been devoted to developing more accurate and/or reproducible summation and dot product algorithms, of which an extensive survey would not fit the format of this document. For the sake of simplicity, let us consider here that a large fraction of accurate summation algorithms leverage a common and very simple idea: use a larger accumulator to store partial sums. In its crudest variant, this might consist in using mixed precision, where a 64-bit FP number accumulates partial results in the summation of 32-bit floats. More elaborate algorithms consider longer fixed-point accumulators (superaccumulators), capable of representing the result of every possible sum of FP numbers of a given format; this is for example the case in ExBLAS [5]. The xsum algorithm [10] is a variation on this concept. Algorithms such as those of ReproBLAS [4] use smaller accumulators, which focus on making the result of the summation independent of the order of the summands, but also feature increased accuracy w.r.t. naive summation.

In this work, we focus on a family of so-called "compensated algorithms", where the error introduced by each accumulation is computed using "error-free transforms" (EFTs), and re-introduced later in the computation. As an example, Alg. 4 describes the `fast_two_sum` algorithm [3], which takes two FP numbers (a, b) as inputs and returns two FP numbers (x, e) such that

$$\begin{cases} x = a \oplus b, \\ x + e = a + b. \end{cases}$$

As such, e can be considered as the round-off error introduced in the FP addition $a + b$. Note that in the particular case of `fast_two_sum`, operands a and b have to be sorted by decreasing magnitude in order for the EFT to be correct.

The compensated summation algorithm due to Kahan, Babuška and Neumaier (KBN) [1], [7], [11] is probably the most famous in this category. A scalar implementation of the KBN algorithm in Julia is available in the `KahanSummation.jl` package. Ogita, Rump and Oishi showed in [12] that the KBN algorithm could be re-interpreted as a "cascaded summation" (Alg. 6) based on the `fast_two_sum` EFT. Since `fast_two_sum` inputs need to be sorted by magnitude in order for the transform to be error-free, Ogita, Rump and Oishi propose to build a similar cascaded summation based on the `two_sum` EFT [8] (Alg. 5), whose practical implementation does not require any test or branch to sort the inputs and should therefore be more suitable for efficient vectorization. This `two_sum`-based cascaded summation will be referred to as the ORO algorithm in the following. Compensated summation algorithms built with this technique (as well as compensated dot product algorithms built in a similar way) can be shown to produce results that are as accurate as if they had been computed using a naive algorithm at twice the working precision, rounding the final result to the nearest working precision value:

$$\varepsilon_{\text{comp}}(x) = K_N \; C(x) \; \epsilon^2. \tag{2}$$

For the sake of completeness, let us mention that EFTs can also be used to build Floating-Point Expansions (FPEs), in which a higher-precision number is represented as the (unevaluated) sum of multiple FP numbers. FPEs of size 2, also known as double-double [2], are at the core of the implementation of the XBLAS library [9] whose algorithms

Algorithm 5 `two_sum` error-free transform

Require: $(a, b) \in \mathbb{F}^2$
Ensure: $x = a \oplus b$ and $x + e = a + b$
$x \leftarrow a \oplus b$
$y \leftarrow x \ominus a$
$e \leftarrow (a \ominus (x \ominus y)) \oplus (b \ominus y)$

Algorithm 6 Cascaded summation.
`EFT_sum` can be either `fast_two_sum` (which gives the KBN algorithm) or `two_sum` (which gives the ORO summation algorithm)

1: {Initialization:}
2: $(\sigma, \tau) \leftarrow (0, 0)$
3: {Loop on vector elements:}
4: **for** $i \in 1 : N$ **do**
5: $(\sigma, e) \leftarrow$ **`EFT_sum`**(σ, x_i)
6: $\tau \leftarrow \tau \oplus e$
7: **end for**
8: **return** $\sigma \oplus \tau$

are very similar in spirit to Ogita, Rump and Oishi's cascaded summation, except in the way lower-order terms are handled.

C. Vectorized implementation of compensated algorithms

The remainder of this paper concentrates on the efficient, vectorized implementation of compensated algorithms such as those introduced by Ogita, Rump and Oishi. Looking at the cascaded summation algorithm 6, one can observe the similarity to a naive summation (Alg. 1):

- the single accumulator a (Alg. 1, line 2) is replaced by an EFT-based accumulator (σ, τ) (Alg. 6, line 2)
- the accumulation operation itself (Alg. 1, line 5) is replaced by the use of an EFT to accumulate high-order terms in σ and compute an error (Alg. 6, line 5) followed by a simple accumulation of low-order terms in τ (line 6).

Nevertheless, a few questions arise when trying to vectorize such an algorithm. Looking at the efficiently vectorized version of the naive summation (Alg. 3), it appears that there are quite a few additional places where intermediate results are summed:

(**acc+pack**) lines 8, 11 and 17: addition of a pack of vector elements into a SIMD accumulator;
(**acc+acc**) line 20: addition of a SIMD accumulators into another SIMD accumulator;
(**`vsum`**(**acc**)) line 21: sum of all elements of a SIMD accumulator, yielding the initial value of a scalar accumulator;
(acc+elem) line 24: addition of a (scalar) vector element to a scalar accumulator.

While summing an ill-conditioned vector, a catastrophic cancellation might occur in any of these sums, so that special care must be taken to compensate for errors at all these locations so that the overall accuracy of the compensated algorithm is not lost. The (acc+elem) accumulation type is exactly the same as what appears in a standard (scalar) cascaded summation, and should therefore be handled in the way described by Alg. 6. The (**acc+pack**) accumulation is the straightforward SIMD generalization of this operation, and can be handled in the same way, provided that vectorized EFT implementations are available.

Algorithm 7 Addition of two EFT-based SIMD accumulators

In: two EFT-based SIMD accumulators
$\mathbf{a}_1 = (\boldsymbol{\sigma}_1, \boldsymbol{\tau}_1)$ and $\mathbf{a}_2 = (\boldsymbol{\sigma}_2, \boldsymbol{\tau}_2)$
Out: EFT-based SIMD accumulator representing $\mathbf{a}_1 \oplus \mathbf{a}_2$

$(\boldsymbol{\sigma}, \mathbf{e}) \leftarrow$ **`EFT_sum`**$(\boldsymbol{\sigma}_1, \boldsymbol{\sigma}_2)$
$\tau \leftarrow (\boldsymbol{\tau}_1 \oplus \boldsymbol{\tau}_2) \oplus e$
return $(\boldsymbol{\sigma}, \boldsymbol{\tau})$

Algorithm 8 Summation of an EFT-based accumulator

In: EFT-based SIMD accumulator $\mathbf{a} = (\boldsymbol{\sigma}, \boldsymbol{\tau})$ of width W
Out: EFT-based scalar accumulator $a =$ **`vsum`**$(\mathbf{a})$

$(\sigma, \tau) \leftarrow (0, 0)$
for $i \in 1 : W$ **do**
$(\sigma, e) \leftarrow$ **`EFT_sum`**$(\sigma, \boldsymbol{\sigma}[i])$
$\tau \leftarrow (\tau \oplus \boldsymbol{\tau}[i]) \oplus e$
end for
return (σ, τ)

In order to keep the "spirit" of the cascaded summation algorithm, we propose to handle the two remaining accumulation types (**acc+acc**) and (**`vsum`**(**acc**)) using algorithms 7 and 8 respectively, in such a way that high-order terms are always summed using an EFT, whose error term is simply accumulated with the low-order terms.

The soundness of this approach can be assessed by looking at the relative error of the results produced by vectorized implementations of the compensated algorithms. Looking at Figure 2a, it is clear that both compensated algorithms behave as predicted, with relative errors following (2): KBN and ORO give the exact same results, and only start losing accuracy for condition numbers in the order of $1/\epsilon \simeq 10^{16}$, computing meaningless results when the condition number reaches $1/\epsilon^2 \simeq 10^{32}$. The compensated dot product algorithm labeled "dot_oro" on figure 2b re-uses the same vectorized cascaded summation algorithm, using an additional `two_prod` EFT to compute error terms for the elementwise product of the two vectors, as proposed in [12]. Again, this compensated dot product implementation behaves accordingly to the theory, with relative errors following (2).

IV. Julia implementation & benchmarking

A. Implementation in Julia

The full Julia implementation of all algorithms described here (naive and compensated versions of the summation and dot product) is available under an open source license in the

Listing 1 `two_sum` implementation in Julia

```
function two_sum(a::T, b::T) where {T}
    SIMDops.@explicit

    x = a + b
    y = x - a
    e = (a - (x - y)) + (b - y)
    return x, e
end
```

`AccurateArithmetic.jl` package[3]. Any reader interesting in the details can therefore look at the code, and only some interesting features will be highlighted here. While doing so, we will also try and make the case that the Julia programming language is well suited to easily develop efficient implementation of such algorithms. In particular, it should be noted that the actual implementation closely follows the algorithmic description made here, and that all individual algorithms can be implemented in their own separate functions, that the Julia compiler will then assemble in an efficient way. This makes it easy to maximize code re-use, as well as to structure the code as building blocks, which will later allow for an easy exploration of new or unusual combinations of elementary components.

As a first example, Listing 1 gives the Julia implementation of the `two_sum` EFT (Alg. 5). This implementation, which is a nearly direct translation from the textbook algorithm, works for any type of inputs, including SIMD packs. The `SIMDops.@explicit` macro at the beginning arranges for elementary operators such as + or − to be vectorized when they operate on SIMD packs, using the `SIMDPirates.jl` package[4]. Special care is taken to ensure that no further optimization is performed (such as fusing multiplications and additions) which would potentially break the EFT: this is the meaning of `@explicit`.

As a second, more complicated example, Listing 2 shows the partially unrolled loop corresponding to lines 4–12 in Alg. 3. Again, although the problem at hand is much more complicated, the Julia implementation closely follows the algorithmic description. The addition of a SIMD pack to an accumulator has been abstracted away in an `add!` function, which can have specific implementations (methods, in the Julia terminology) for each accumulator type. This way, the same function implementing Alg. 3 can be used as a building block for all kinds of summation algorithms (naive or compensated). Following the same kind of logic, the mechanism loading SIMD packs is designed in such a way that it can handle a 1-tuple containing the input vector in the case of the summation, but also load packs from two input vectors contained in a 2-tuple in the case of the dot product.

[3]https://github.com/JuliaMath/AccurateArithmetic.jl
[4]https://github.com/chriselrod/SIMDPirates.jl

Listing 2 Partial unrolling of the vectorized loop

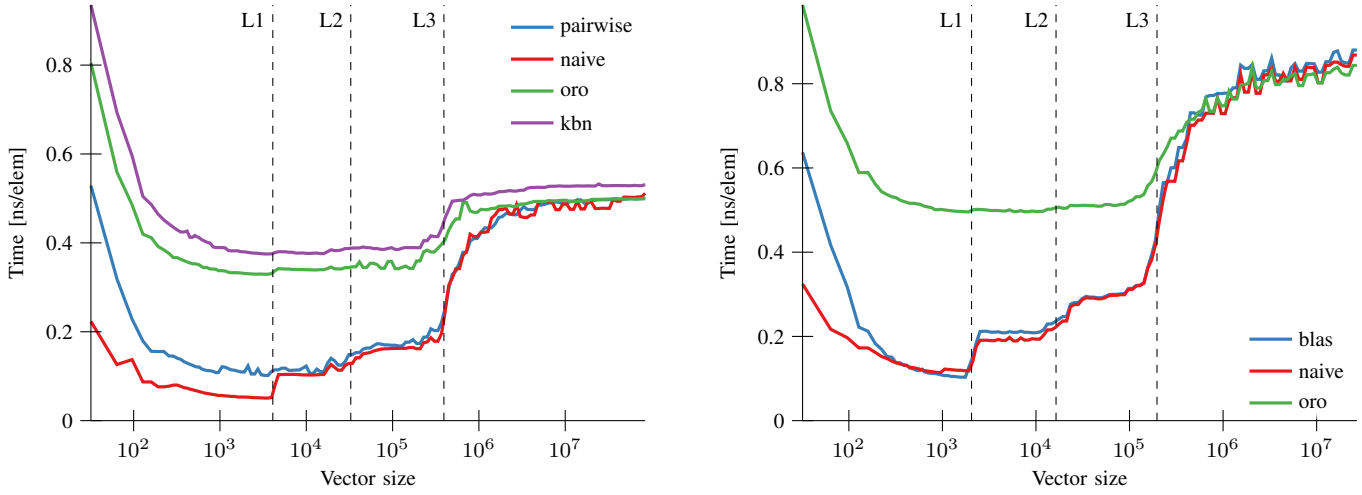
```
# ...
px = pointer.(x)
offset = 0

# Main loop
for j in 1 : N÷($W*$U)

  # Unroll U times
  Base.Cartesian.@nexprs $U i -> begin

    # Load a pack of width W and type T
    # starting at position (px+offset)
    p_i = vload.(Vec{$W,$T}, px.+offset)

    # Add pack p_i into accumulator acc_i
    add!(acc_i, p_i...)

    # Update offset for the next pack
    offset += $W * $sizeT
  end
end

# ...
```

B. *Performance evaluation*

Although the implementation makes use of small, composable functional units, the Julia compiler specializes and optimizes the program for the specific combination of building blocks being used, so as to emit very efficient code.

Figure 3 plots, for various implementations of the sum and dot product, the run time (renormalized per element) against the vector size. These results have been obtained on a system featuring an Intel Core i5 6200U, which is a typical mobile AVX2-enabled Skylake-generation CPU. It can be observed that our naive implementation performs a bit better than Julia's standard pairwise summation for small vector sizes, but almost identically afterwards. Once vectors start having significant sizes (more than a few hundreds of elements), both implementations are memory bound, as expected of a typical BLAS1 operation. This explains why significant decreases in the performance can be observed when the vector can't fit into the L2 cache (around 30k elements, or 256kB on the test system) or the L3 cache (around 400k elements, or 3MB on the test system). Our explicitly vectorized naive summation remains memory bound in the L1 cache, where it gains a speed-up of about $2\times$ with respect to Julia's automatically vectorized pairwise summation.

The ORO algorithm, when implemented with a suitable unrolling level ($2^2 = 4$ unrolled iterations in this case), is CPU-bound when vectors fit inside the cache. However, when vectors are too large to fit into the L3 cache, the implementation becomes memory-bound again, which means that it achieves the same performance as the standard summation. In other words, the improved accuracy is free for sufficiently



(a) Summation (b) Dot product

Fig. 3: Comparison of the performance of various summation and dot product implementations (Intel Core i5 6200U)

large vectors. For smaller vectors, the accuracy comes with a slow-down no larger than $3\times$ for vectors which fit in the L2 cache or $2.5\times$ in the L3 cache.

The KBN algorithm performs less well on this system: although the `fast_two_sum` EFT uses fewer operations than `two_sum`, the need for a test sorting the arguments by magnitude makes it less suitable for a vectorized implementation on this system.

The same kind of conclusions can be reached for the dot product. In order to make an apples-to-apples comparison, the use of threads has been disabled in OpenBLAS. Naive implementations are always memory bound (the L1 cache boundary can be seen clearly in this case). The ORO compensated implementation incurs a slow-down no larger than $5\times$ in the L1 cache, which decreases to around $2.5\times$ in the L2 cache, $1.7\times$ in the L3 cache, and no overhead outside the cache.

The same tests can be performed on other type of architectures, and lead to different conclusions. For example, Figure 4 presents the results obtained on a system powered by an Intel Xeon Silver 4110 CPU, of the same Skylake generation, but supporting the AVX512 extension. On this system, it is interesting to note that the KBN algorithm performs better than ORO. This can probably be attributed to this generation of CPUs better handling of the branches in `fast_two_sum`, thanks to the operands masking capabilities introduced in AVX512. Compensated summation algorithms remain memory bound much longer, with KBN achieving the same performance as naive algorithms even in the L3 cache. With respect to Julia's pairwise summation, a $4\times$ slow-down is observed in the L1 cache, which decreases to $2.5\times$ in the L2 cache. A $5-10\%$ overhead is observed outside the cache.

As for the dot product, the OpenBLAS implementation seems to be a bit more optimized than our naive implementation, with a $10-20\%$ difference in speed overall. The compensated dot product implementation achieves the same performance as the naive one in the L3 cache, while incurring a $3.5\times$ slow-down in the L2 cache and a 7% overhead outside the cache.

V. Closing notes

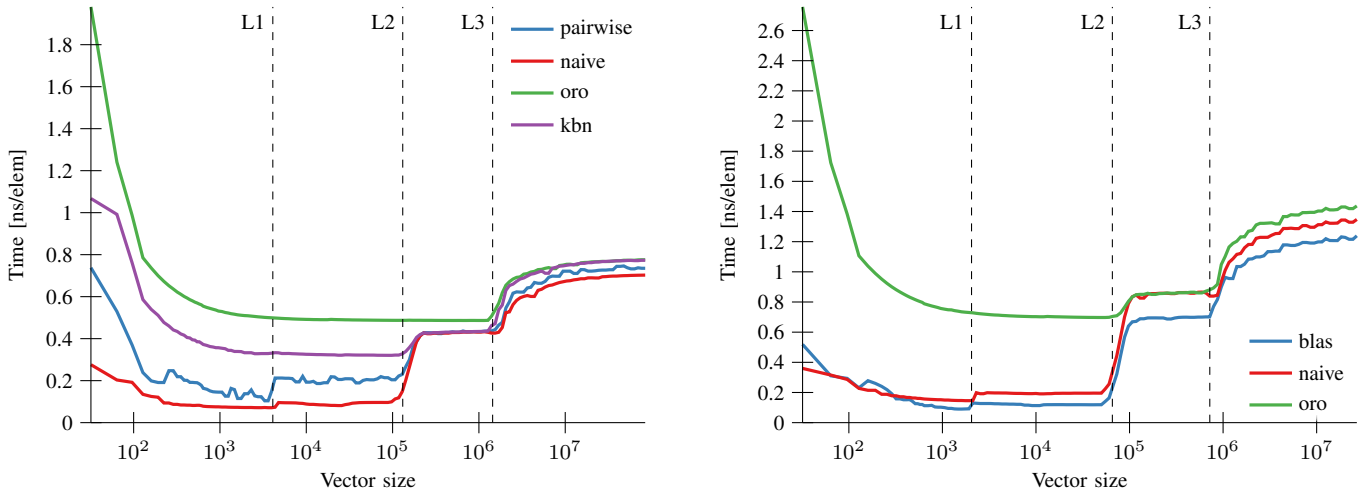
In summary, the work presented here tries to bring into the Julia ecosystem efficient, vectorized implementations of state-of-the-art compensated summation algorithms. All the code described here is available as an open source Julia package on github:

https://github.com/JuliaMath/AccurateArithmetic.jl

The family of algorithms implemented here is described in [12]: the Kahan-Babuška-Neumaier summation algorithm, and the simply compensated Ogita-Rump-Oishi summation and dot product algorithms. In accordance to the theory, these algorithms are observed to effectively double the floating-point precision (yielding, for double-precision inputs, a result as accurate as if it had been naively computed using FP numbers with a 106-bit mantissa).

From a performance viewpoint, these implementations are shown to incur no significant penalty with respect to naive algorithms, as soon as input vectors are larger than the L3 cache. For smaller vectors fitting in the L2 cache, the slow-down is no larger than approximately $3\times$, the precise value depending on the architecture. As observed by Ogita, Rump and Oishi, the branches of the KBN algorithm make it a bit less efficient than the ORO algorithm on AVX2 architectures. However, benchmarks on AVX512 systems suggest that this might not always be the case, so that compensated algorithms based on the `fast_two_sum` EFT might have to be reconsidered.

The Julia implementation provided here is very compact and modular, which makes it a very suitable platform for further



(a) Summation (b) Dot product

Fig. 4: Comparison of the performance of various summation and dot product implementations (Intel Xeon Silver 4110)

experiments. Future work might include implementations of other types of accurate summation and dot product algorithms, such as those based on fixed-point-like accumulators and used in other software packages such as ExBLAS, XBLAS, ReproBLAS or xsum. Alternatively, the compensated summation and dot product algorithms implemented here could be used as a basis to implement accurate versions of other BLAS functions like matrix decompositions (QR, Cholesky) or solving triangular linear systems.

An other potential direction for future work would be to focus on improving the accuracy of single- or half-precision algorithms (for example *via* the use of mixed precision) or to develop similar implementations specifically targeting GPUs.

Acknowledgements

The authors feel indebted to the members of the Julia discourse community[5] and in particular Jeffrey Sarnoff and Laurent Plagne. This work would not have started without their encouragement. Their insightful comments and help greatly contributed to increase the quality of this paper.

References

[1] Ivo Babuška. Numerical stability in mathematical analysis. In *Proceedings of the IFIP Conference*, Amsterdam, 1968.

[2] David H. Bailey. A fortran 90-based multiprecision system. *ACM Trans. Math. Softw.*, 21(4):379–387, December 1995.

[3] T.J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18:224–242, 1971.

[4] J. Demmel and H. D. Nguyen. Parallel reproducible summation. *IEEE Transactions on Computers*, 64(7):2060–2070, July 2015.

[5] Roman Iakymchuk, Sylvain Collange, David Defour, and Stef Graillat. ExBLAS: Reproducible and accurate BLAS library. In *Proceedings of the Numerical Reproducibility at Exascale (NRE2015) workshop held as part of the Supercomputing Conference (SC15)*, Austin, TX, USA, November 2015.

[6] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1–70, 2008.

[7] William Kahan. Further remarks on reducing truncation errors. *Communications of the ACM*, 8(1), January 1965.

[8] Donald Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison-Wesley, 1969.

[9] Xiaoye S. Li, James W. Demmel, David H. Bailey, Greg Henry, Yozo Hida, Jimmy Iskandar, William Kahan, Suh Y. Kang, Anil Kapur, Michael C. Martin, Brandon J. Thompson, Teresa Tung, and Daniel J. Yoo. Design, implementation and testing of extended and mixed precision BLAS. *ACM Trans. Math. Softw.*, 28(2):152–205, June 2002.

[10] Radford M. Neal. Fast exact summation using small and large superaccumulators. Technical report, University of Toronto, May 2015.

[11] Arnold Neumaier. Rundungsfehleranalyse einiger Verfahren zur Summation endlicher Summen. *ZAMM (Zeitschrift für Angewandte Mathematik und Mechanik)*, 54:39–51, 1974.

[12] Takeshi Ogita, Siegfried M. Rump, and Shin'ichi Oishi. Accurate sum and dot product. *SIAM J. Sci. Comput.*, 26:1955–1988, 2005.

[13] Qian Wang, Xianyi Zhang, Yunquan Zhang, and Qing Yi. Augem: automatically generate high performance dense linear algebra kernels on x86 cpus. In *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2013.

[5] https://discourse.julialang.org/

APPENDIX
REPRODUCIBILITY RESOURCES

The `AccurateArithmetic.jl` Julia package implements all algorithms described in this paper. It also contains everything needed to reproduce the results shown in this paper. Below is a full list of the actions to be taken by anyone wanting to try and reproduce the results on their own Linux system. Commands prefixed with a "`sh>`" prompt are to be entered in a shell; commands prefixed with a "`julia>`" prompt are to be entered in a Julia interactive session. The Julia programming language should have been downloaded and installed beforehand from:

https://julialang.org/downloads/

Any version above v1.0.0 should work; the results presented in this paper have been obtained using Julia v1.1.1.

Be aware that step (4) in this procedure might take a few hours to complete. Afterwards, all measurements should be available as JSON files in the `AccurateArithmetic.jl/test` directory. After step (5), all figures showed in this paper should be available as PDF files in the same directory.

Should anyone actually run this procedure, the authors would really like to get a copy of their results, so as to gain more insight about the behavior of compensated algorithms on a variety of systems.

```
# (1) Get the repository
sh> cd /tmp
sh> git clone --branch paper-correctness-2019 \
  https://github.com/JuliaMath/AccurateArithmetic.jl.git
sh> cd AccurateArithmetic.jl/test

# (2) Install required dependencies
#     and run the test suite
sh> julia --project
julia> using Pkg
julia> pkg"instantiate"
julia> pkg"test"
julia> exit()

# (3) Additional dependencies for performance tests
sh> julia --project=
julia> using Pkg
julia> pkg"add BenchmarkTools Plots Printf Statistics Test"
julia> exit()

# (4) Run the performance tests
sh> julia --project -O3 -L perftests.jl -e 'run_tests()'

# (5) Plot the graphs
#     (this step can be run on another machine, provided
#     that all JSON files resulting from the previous
#     step are copied)
sh> julia --project -L perfplots.jl -e 'plot_results()'
```