Condense and Distill: fast distillation of large floating-point sums via condensation

Stef Graillat, Théo Mary

CONDENSE AND DISTILL: FAST DISTILLATION OF LARGE FLOATING-POINT SUMS VIA CONDENSATION*

STEF GRAILLAT† AND THEO MARY†

Abstract. Floating-point summation is a fundamental task at the heart of many scientific computing applications. When the sum is very ill conditioned, computing it accurately can become challenging. One can employ distillation methods, which consist in transforming an ill-conditioned sum into an equivalent, but well-conditioned one. However, distillation is a very expensive process. In this article, we propose Condense & Distill, a new distillation method that relies on a preprocessing step that we call condensation, because it transforms the original sum into a far smaller sum, which can then be distilled inexpensively. This condensation step exploits a new, key observation that floating-point addition is exact when the addends have both the same exponent and the same least significant bit. Condense & Distill thus requires accessing the exponent field of the summands. Compared with state-of-the-art summation methods with the same requirement such as the Demmel–Hida method [*SIAM J. Sci. Comput., 25 (2003), pp. 1214–1248*], Condense & Distill presents the significant benefit of running entirely in the working precision, with no need for extra precision. At the same time, it preserves the main advantages of the Demmel–Hida method compared with other methods, in particular those based on error-free transformations such as AccSum [*SIAM J. Sci. Comput., 31 (2008), pp. 189–224*]: namely, its cost is independent of the conditioning, and it exhibits near perfect parallel scaling. We present numerical experiments that confirm that Condense & Distill can reliably and efficiently distill large ill-conditioned sums, and performs favorably compared with other state-of-the-art summation methods.

Key words. summation, floating-point arithmetic, rounding error analysis, distillation, ill-conditioning

AMS subject classifications. 65G50, 65Y04, 65Y05, 65Y20

1. Introduction.

The summation of n floating-point numbers,

$$\sum_{i=1}^{n} x_i,$$

is one of the most fundamental tasks of scientific computing. In several applications, computing the sum accurately is challenging because it is both large and ill-conditioned, that is, its conditon number

$$\kappa = \frac{\sum_{i=1}^{n} |x_i|}{|\sum_{i=1}^{n} x_i|} \tag{1.1}$$

is large. For only moderately large values of κ, a possible approach is to simply evaluate the sum in higher precision arithmetic, such as the IEEE quadruple precision (fp128) arithmetic. However, this approach is no longer viable for really large values of κ, since in this case even quadruple precision is not sufficient to obtain an accurate result.

Floating-point summation has been studied for a long time. In his seminal book [2], Higham devoted an entire chapter to summation algorithms. Here we cannot review all the papers that deal with floating-point summations. We will only present the two main families of algorithms and their principle. A recent paper with an overview of other summation algorithms can be found in [4]. Several methods have been proposed to handle extremely ill-conditioned sums, the most popular of which we can categorize in two broad classes.

*Version of March 16, 2024.

†Sorbonne Université, CNRS, LIP6, Paris, France (stef.graillat@lip6.fr, theo.mary@lip6.fr)

- The first class relies on the finite number of exponents in typical (IEEE) floating-point arithmetic and on the fact that numbers with exponents not too far apart can be added exactly using extended precision accumulators. This is for example the case of Kulisch's accumulator [3], which sums all numbers in a very long accumulator, or of the Demmel–Hida algorithm [1], which sums together numbers of comparable exponent using higher precision arithmetic, such as quadruple precision. This strategy is also used in HybridSum [11] and OnlineExactSum [12].
- The second class exploits the fact that the error incurred by floating-point addition is itself a floating-point number. This error can be computed exactly via error-free transformations such as Fast2Sum. This class includes in particular the AccSum method [8], which computes a faithful rounding of the sum irrespective of its conditioning, and the PrecSum method [9], which computes the sum as if using K-fold precision (for a given K). Those algorithms have been improved respectively as FastAccSum and FastPrecSum in [7].

The methods from the first class have a double weakness: they require access to the exponent of the floating-point numbers, which can be expensive, and they require the use of extended precision accumulators. In contrast, the methods from the second class only require standard arithmetic operations on the summands. However, they also require a much larger number of floating-point operations, and their cost strongly depends on the conditioning. Moreover, they are much less parallel than the methods from the first class. In any case, both classes of methods can be quite expensive.

Many of the summation methods able to handle ill-conditioned sums rely on the process of *distillation*. Distillation consists in iteratively transforming the original, ill-conditioned sum into an equivalent but well-conditioned sum that can then be evaluated accurately. This is especially the case of the second class of methods mentioned above, although the first class can also be used to design distillation methods, see for example [1, sec. 5] for the Demmel–Hida method.

In this article, we propose a method to transform the original sum into another, equivalent sum, which is not necessarily better conditioned, but which is far smaller. The smaller sum can then be distilled inexpensively by any of the distillation methods mentioned above. We call the first transformation of the sum into a smaller equivalent one the process of *condensation*. In the natural language, condensation carries indeed the idea of compacting or contracting a large, complex system into a smaller, simpler one. Moreover, and quite appropriately, in the original physical meaning of the words, condensation (the process of transforming gas to liquid) is a crucial component of the process of distillation (the process of separating substances from a liquid). The entire process (condensing the original sum into a smaller one and then distillating it) thus gives rise to a new method to handle large ill-conditioned sums, which we call the Condense & Distill method.

The condensation step is based on the key observation that floating-point numbers with exponents not too far apart can be added exactly, even *in the working precision*, as long as some congruence condition of their least significant digits is satisfied. In particular, base-two numbers with the same exponent and the same least significant bit can be added exactly in the working precision. Condense & Distill thus belongs to the first class of summation methods mentioned above, since it requires access to the exponent of the summands. It is most similar to the Demmel–Hida method: it also adds together numbers of the same exponent, but does not require any extra precision. The entire condensation process can be performed in the working precision; only the final condensed sum needs to be distilled using extended precision (or, for that matter, any other distillation method). This is achieved at the cost of accessing the least significant bit of the summands, which is a negligible overhead compared with the cost of accessing their exponent.

Therefore, Condense & Distill allows for significant improvements, not only in terms of performance (because operations in the working precision are faster), but also in terms of robustness/portability. For example, our algorithm can easily accomodate quadruple precision as the working precision. Compared with the second class of summation methods (AccSum, etc.), Condense & Distill shares the main strengths of the first class: its performance can be made completely independent of the conditioning of the sum, and it exhibits nearly perfect parallel scaling.

The rest of this article is organized as follows. In section 2, we carry out an analysis to determine conditions for the floating-point addition $x+y$ to be exact. We leverage this analysis in section 3 to develop the Condense & Distill method. We experimentally showcase the use of Condense & Distill against traditional distillation algorithms in section 4, where we also analyze the parallel scaling of our algorithm. Finally we provide our concluding remarks in section 5.

2. When is $x+y$ exact?. Consider a floating-point number system $\mathbb{F}$ with base β, exponent range $(e_{\min}, e_{\max})$, and significand of length $t \geq 2$. To denote $x \in \mathbb{F}$ we will use the notation

$$x = \pm(\beta^{e_x} + k_x \varepsilon_{e_x}), \quad \varepsilon_{e_x} = \beta^{e_x+1-t}, \quad k_x \in \mathbb{N},$$

where e_x is the unbiased exponent of x and ε_{e_x} is the space between adjacent floating-point numbers in the interval $[\beta^{e_x}, \beta^{e_x+1}]$. Note that $k_x < (\beta-1)\beta^{t-1}$ since $x < \beta^{e_x+1}$.

Given $x, y \in \mathbb{F}$ of the same sign, conditions for the subtraction $x-y$ to be exact are known by Sterbenz lemma (see [10] or [6]). In this section, we determine conditions for the addition $x+y$ to be exact. We begin with the very general result below.

THEOREM 2.1. *Let $x, y \in \mathbb{F}$ of the same sign $\sigma = \pm 1$ such that*

$$x = \sigma(\beta^{e_x} + k_x \varepsilon_{e_x}),$$
$$y = \sigma(\beta^{e_y} + k_y \varepsilon_{e_y}).$$

Assuming (without loss of generality) that $|x| \leq |y|$, then $x+y \in \mathbb{F}$, and thus the addition is exact, iff one of the following conditions is met:

(i) $x = 0$;
(ii) $|x+y| < \beta^{e_y+1}$, $e_y - e_x \leq t-1$, *and* $k_x \equiv 0 \bmod \beta^{e_y-e_x}$;
(iii) $|x+y| = \beta^{e_y+1}$, $e_y + 1 \leq e_{\max}$, $e_y - e_x \leq t-1$, *and* $k_x \equiv 0 \bmod \beta^{e_y-e_x}$;
(iv) $|x+y| > \beta^{e_y+1}$, $e_y + 1 \leq e_{\max}$, $e_y - e_x \leq t-2$, *and* $k_x + k_y\beta^{e_y-e_x} \equiv 0 \bmod \beta^{e_y-e_x+1}$.

Proof. Case (i) is trivial. For the remaining cases, we consider positive x and y, the negative case being analagous. We first note that $x+y \in [\beta^{e_y}, \beta^{e_y+2}]$, so that if $x+y$ is to be exact its exponent can only be e_y or e_y+1.

- Let us first consider the case (ii), where $x+y < \beta^{e_y+1}$. Then $x+y \in \mathbb{F}$ iff ε_{e_y} divides $x+y-\beta^{e_y} = k_y\varepsilon_{e_y} + x$, that is, iff ε_{e_y} divides $x = \beta^{e_x} + k_x\varepsilon_{e_x}$. We first prove that it is necessary for $\varepsilon_{e_y} = \beta^{e_y+1-t}$ to divide β^{e_x}, which is only possible if $e_y - e_x \leq t-1$. If this condition is not met, then $e_x \leq e_y - t$ and so $\beta^{e_x} \leq \varepsilon_{e_y}/\beta$; moreover, $k_x\varepsilon_{e_x} < (\beta-1)\beta^{e_x} \leq (\beta-1)\beta^{e_y-t} = (\beta-1)\varepsilon_{e_y}/\beta$ and therefore $\varepsilon_{e_y} > x$ cannot divide x. We conclude that $e_y - e_x \leq t-1$ is a necessary condition for $x+y \in \mathbb{F}$ (when $x \neq 0$). The condition is not sufficient, since ε_{e_y} must also divide $k_x\varepsilon_{e_x}$, which happens iff k_x is congruent to $0 \bmod \beta^{e_y-e_x}$. This concludes case (ii).
- Case (iii) is identical to case (ii), except we must guarantee that β^{e_y+1} exists by barring overflow with the condition $e_y + 1 \leq e_{\max}$.

- In case (iv), $\beta^{e_y+1} < x + y < \beta^{e_y+2}$, we also need $e_y + 1 \leq e_{\max}$ to bar overflow. Then, $x + y \in \mathbb{F}$ iff ε_{e_y+1} divides $x + y - \beta^{e_y+1} = \beta^{e_y}(1 - \beta) + k_y\varepsilon_{e_y} + x$. First, we note that $\varepsilon_{e_y+1} = \beta^{e_y+2-t}$ divides β^{e_y} for $t \geq 2$, so $x + y \in \mathbb{F}$ iff ε_{e_y+1} divides $k_y\varepsilon_{e_y} + \beta^{e_x} + k_x\varepsilon_{e_x}$. Second, we prove that it is necessary for ε_{e_y+1} to divide β^{e_x}, which requires $e_y - e_x \leq t - 2$. Indeed, if this condition is not met, $e_x \leq e_y + 1 - t$, and so $x < \beta^{e_x+1} \leq \beta^{e_y+2-t} = \varepsilon_{e_y+1}$. Thus $x + y < y + \varepsilon_{e_y+1} < \beta^{e_y+1} + \varepsilon_{e_y+1}$, but we are in the case $x + y > \beta^{e_y+1}$, so $x + y \notin \mathbb{F}$. Therefore $e_y - e_x \leq t - 2$ is necessary and ε_{e_y+1} divides β^{e_x}. Moreover, for the condition to be sufficient, we also need ε_{e_y+1} to divide $k_y\varepsilon_{e_y} + k_x\varepsilon_{e_x}$, which happens when $k_x + k_y\beta^{e_y-e_x}$ is congruent to 0 mod $\beta^{e_y-e_x+1}$. ∎

Theorem 2.1 fully characterizes the conditions for the addition of two floating-point numbers $x + y$ to be exact. The conditions essentially boil down to two components: the exponents of x and y must not be too far apart, and their mantissas must satisfy some congruence condition. Making use of this characterization in practice could however be complex. Interestingly, the conditions become much simpler if we specialize them to numbers sharing the same exponent ($e_x = e_y$).

COROLLARY 2.2. *Let $x, y \in \mathbb{F}$ of same sign $\sigma = \pm 1$ and same exponent e, such that*

$$x = \sigma(\beta^e + k_x\varepsilon_e),$$
$$y = \sigma(\beta^e + k_y\varepsilon_e).$$

Then $x + y \in \mathbb{F}$, and thus the addition is exact, iff either
(i) $|x + y| < \beta^{e+1}$ or
(ii) $e + 1 \leq e_{\max}$ and $k_x + k_y \equiv 0 \bmod \beta$.

Case (i) of Corollary 2.2 corresponds to cases (i) and (ii) of Theorem 2.1, while case (ii) of the corollary corresponds to cases (iii) and (iv) of the theorem. Note that case (i) of the corollary is only possible for $\beta > 2$, since in binary floating-point arithmetic, the sum of two numbers in the range $(2^e, 2^{e+1}]$ necessarily yields a number in the range $(2^{e+1}, 2^{e+2}]$. Moreover, recall that floating-point numbers can be expressed as

$$x = \beta^{e-t} \sum_{i=1}^{t} d_i \beta^{t-i}, \tag{2.1}$$

where the digits d_i satisfy $0 \leq d_i \leq \beta - 1$ with $d_1 \neq 0$ for normalized numbers. Since β divides β^{t-i} for $i < t$, the condition $k_x + k_y \equiv 0 \bmod \beta$ further simplifies to a condition on the least significant digits d_t^x and d_t^y of x and y:

$$d_t^x + d_t^y \equiv 0 \bmod \beta, \tag{2.2}$$

that is, $d_t^x + d_t^y$ must be either 0 or β. For $\beta = 2$, this further simplifies to $d_t^x = d_t^y$, yielding the following result.

COROLLARY 2.3. *If $x, y \in \mathbb{F}$ with $\beta = 2$ have the same sign, exponent, and least significant bit, then barring overflow their addition is exact.*

Corollary 2.3 provides a necessary condition that is much simpler to check, since it only requires to access the sign, exponent, and least significant bit of x and y. In the next section we propose an algorithm that exploits this observation to condense a large sum into an equivalent one with far fewer summands.

3. Fast distillation via condensation. We now describe the Condense & Distill algorithm, which exploits Corollary 2.3 to compute rapidly and exactly

$$s = \sum_{i=1}^{n} x_i, \quad x_i \in \mathbb{F}. \tag{3.1}$$

Condense & Distill consists of two steps. The first step is to condense the sum by adding pairs of summands sharing the same exponent, sign, and least significant bit (hereinafter abbreviated as LSB), until no such pairs remain. As we will prove below, the number of remaining summands is then bounded by a small value. This first condensation step therefore transforms the original sum into another sum with a much smaller number of summands. The second step is to then distill this much smaller sum via any traditional distillation method. The condensation step thus serves as a preprocessing to accelerate the distillation step.

To prove the algorithm's exactness and cost, we conceptually describe it as building a forest (a disjoint set of trees). We first place the summands x_i as leaf nodes on a level determined by their exponent. Then we repeatedly sum pairs of siblings with identical sign and LSB and place the (exact) result on the level above, until all pairs of siblings have either a different sign or a different LSB. At this point there thus remains at most 4 nodes per level, and the number of non-empty levels is itself bounded by $L = \lceil \log_2 n \rceil + d$, where d is a constant independent of n that equals the number of different exponents among the summands x_i. L is certainly bounded by the total number of possible exponent values of the floating-point system (e.g., 2047 with IEEE binary64), and can be much smaller for typical datasets which do not cover the entire exponent range. The final result is therefore given as the exact unevaluated sum of at most $4L$ floating-point numbers.

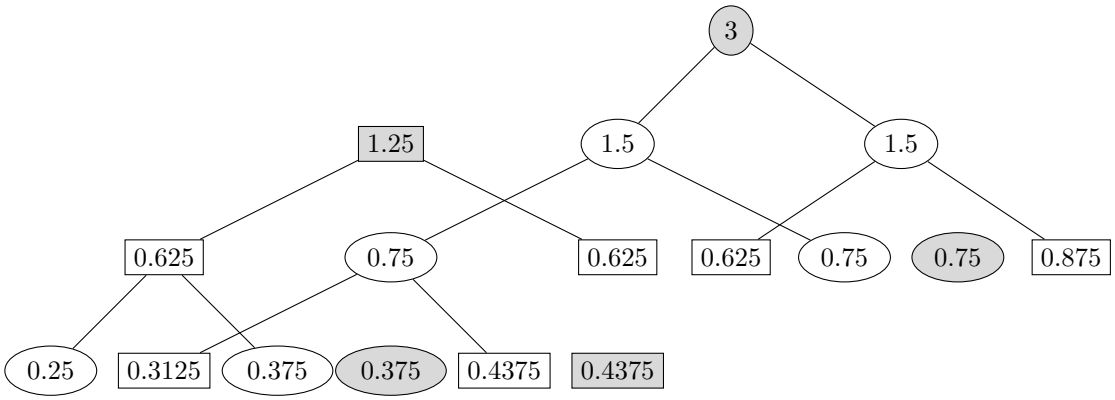

FIG. 3.1. *Illustration of the proposed summation algorithm for a simple floating-point system with $t = 3$. The shaded nodes are the remaining values whose unevaluated sum is equal to the exact result. Ellipse and rectangle nodes correspond to numbers with an LSB of 0 and 1, respectively.*

We illustrate this algorithm in Figure 3.1, using a simple floating-point system $\mathbb{F}$ with $t = 3$. The eleven leaf nodes correspond to the input summands x_i, whose exact sum is $s = 5.8125$ (we only consider positive summands here for simplicity, negative numbers would be treated separately and similarly). All non-leaf nodes correspond to partial results obtained during the computation and, to easily check that they are indeed floating-point numbers, we provide the list of elements of

$\mathbb{F}$ in the interval $(0.25, 3)$:

$$0.25, 0.3125, 0.375, 0.4375, 0.5, 0.625, 0.75, 0.875, 1, 1.25, 1.5, 1.75, 2, 2.5, 3.$$

Ellipse nodes correspond to numbers with an LSB equal to 0, and can thus be summed exactly with other ellipse nodes on the same level. Rectangle nodes correspond to numbers with an LSB equal to 1, and can similarly be summed exactly with other rectangle nodes on the same level. The five shaded nodes are the root nodes and correspond to the remaining numbers that cannot be summed exactly with another node on the same level. They form an unevaluated sum whose result is equal to the exact sum:

$$s = 0.375 + 0.4375 + 0.75 + 1.25 + 3.$$

It is interesting to remark that the addition of some of these numbers can be represented exactly, namely $0.75 + 1.25 = 2 \in \mathbb{F}$. This is indeed consistent with Theorem 2.1: defining $x = 0.75$ and $y = 1.25$, we have $e_x = -1$, $e_y = 0$, $k_x = 2$, $k_y = 1$, and the condition (iv) of the theorem holds: $k_x + k_y \beta^{e_y - e_x} = 4 \equiv 0 \bmod 2^2$. However, our algorithm will not exploit this because it only tries to sum numbers of identical exponent, for which we only need to check the LSB.

It is important to note that the forest structure of the algorithm is purely conceptual and does not actually need to be built. We also do not need access to all summands previous to the beginning of the computation. Algorithm 3.1 describes an online implementation (which adds summands as they become available, and in any order) that requires at most $4L$ accumulators.

We note that the assumption in Corollary 2.3 that x and y have the same sign is not required: if their signs are different, the subtraction $x + y$ is exact by Sterbenz's lemma, since numbers with the same exponent certainly satisfy $x/2 \le y \le 2x$, and this holds regardless of the LSB of x and y. Therefore, the algorithm could also add pairs of summands with the same exponent and different signs, reducing the maximum number of accumulators (and terms in the unevaluated result) from $4L$ to $2L$. However, the drawback is that we would need to recompute the exponent of the result of each addition, since the exponent of a subtraction of two numbers with the same exponent can have an arbitrarily small exponent depending on how close the two numbers are. In contrast, by restricting the pairs to have the same sign, we know that the exponent of $x + y$ is exactly one more than that of x and y.

4. Numerical experiments. We present a set of numerical experiments to assess the performance of the Condense & Distill method and its behavior with respect to various parameters such as the dimension n and the condition number κ. We also present a parallel implementation of the method and study its scalability.

4.1. Experimental protocol. All the experiments were performed on one node of the Olympe supercomputer, equipped with two 18-core Intel Skylake processors. All code was compiled with gfortran version 9.3.0 and with the -O3 optimization flag.

We test the methods on ill-conditioned sums randomly generated as follows. Assuming $n = 2k+1$ is odd (if n is even we simply add one extra zero summand), we generate k random summands x_i in the range $[10^{-e}, 10^{e}]$, where e is a fixed parameter that determines the width of the dynamic range of the x_i values; we have used $e = 32$ throughout all experiments. We set another k summands to $-x_i$ and set the last summand to $10^e/\kappa$. Finally we randomly shuffle all summands. The exact sum is equal to $10^e/\kappa$, and its conditioning is of the order of κ.

4.2. Comparison with Demmel–Hida and AccSum. We begin by comparing the performance of our new Condense & Distill method with that of the Demmel–Hida and AccSum methods.

Algorithm 3.1 Condense & Distill method.

Input: n summands x_i and a distillation method `distill`
Output: $s = \sum_{i=1}^{n} x_i$

Initialize $\mathrm{Acc}(e, s, b)$ to 0 for $e = e_{\min} : e_{\max}$, $s \in \{-1, 1\}$, $b \in \{0, 1\}$.
for all x_i in any order **do**
  $e = \texttt{exponent}(x_i)$
  $s = \texttt{sign}(x_i)$
  $b = \texttt{LSB}(x_i)$
  $\texttt{insert}(\mathrm{Acc}, x_i, e, s, b)$
end for
$x_{\text{condensed}} = \texttt{gather}(\mathrm{Acc})$
$s = \texttt{distill}(x_{\text{condensed}})$

function $\texttt{insert}(\mathrm{Acc}, x, e, s, b)$
  if $\mathrm{Acc}(e, s, b) = 0$ **then**
    $\mathrm{Acc}(e, s, b) = x$
  else
    $x' = \mathrm{Acc}(e, s, b) + x$
    $\mathrm{Acc}(e, s, b) = 0$
    $b' = \texttt{LSB}(x')$
    $\texttt{insert}(\mathrm{Acc}, x', e + 1, s, b')$
  end if
end function

function $x_{\text{condensed}} = \texttt{gather}(\mathrm{Acc})$
  $i = 0$
  for all nonzero $\mathrm{Acc}(e, s, b)$ **do**
    $i = i + 1$
    $x_{\text{condensed}}(i) = \mathrm{Acc}(e, s, b)$
  end for
end function

Figure 4.1 plots the time cost of each method for varying κ and for two fixed values of n, $n = 10^7$ (left) or $n = 10^8$ (right). The figure shows that, as expected, the cost of the Condense & Distill and Demmel–Hida methods is independent of κ, with Condense & Distill being roughly 35% faster because it avoids using quadruple precision. In contrast, the cost of AccSum strongly depends on κ, growing at a rate of roughly $\log \kappa$. As a result, the time comparison between AccSum and Condense & Distill depends on κ: for only moderately ill-conditioned sums, AccSum is faster, but as κ increases Condense & Distill (and even Demmel–Hida) eventually outperforms AccSum, potentially by very large factors if the sum is extremely ill conditioned. The cutoff value of κ for which Condense & Distill outperforms AccSum also seems to decrease as n increases: it is equal to $\kappa \simeq 10^{35}$ for $n = 10^7$ and $\kappa \simeq 10^{20}$ for $n = 10^8$.

We confirm this last trend in Figure 4.2, where we compare the performance of the methods for an increasing n and a fixed value of κ. The figure shows that Condense & Distill becomes more and

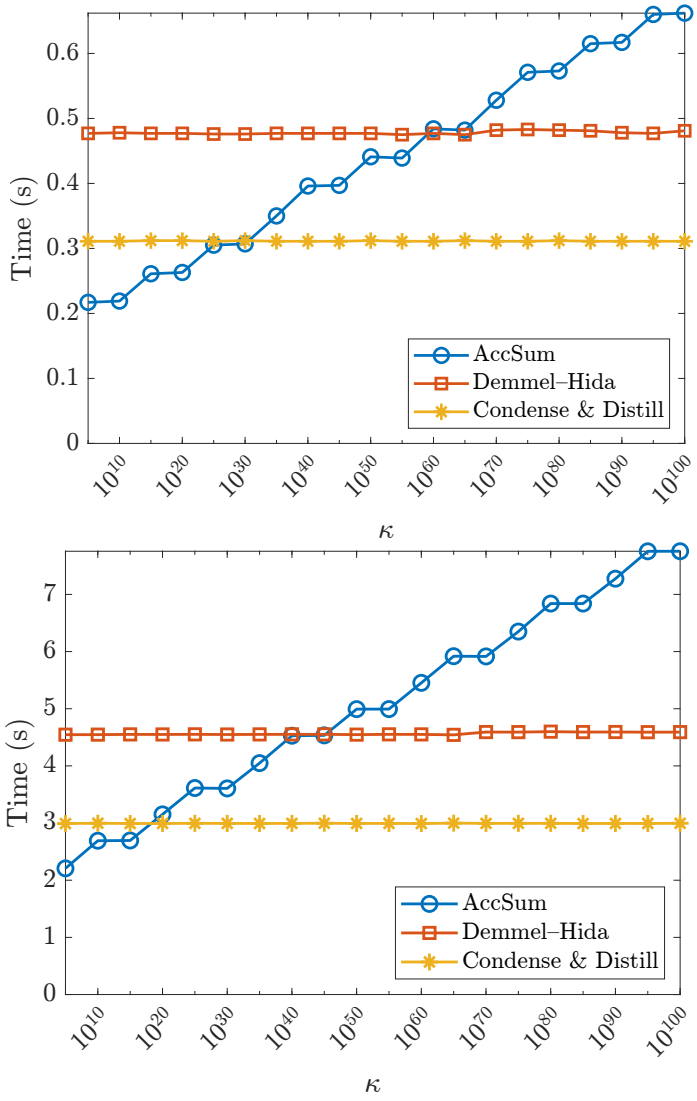

FIG. 4.1. *Comparison between the Demmel–Hida, AccSum, and Condense & Distill algorithms, as a function of the condition number κ and for two dimensions $n = 10^7$ (top) and $n = 10^8$ (bottom). All algorithms are run sequentially (1 thread).*

more competitive with respect to AccSum as n increases. While AccSum is faster for small sums, it is eventually outperformed by Condense & Distill and even by Demmel–Hida, for sufficiently large sums. The cutoff value of n for which Condense & Distill outperforms AccSum similarly decreases as κ increases: for example, it is equal to $n \simeq 10^7$ for $\kappa = 10^{30}$ and $n \simeq 10^6$ for $\kappa = 10^{60}$.

4.3. Parallel scaling. In the previous comparison, the methods are executed sequentially (using only 1 thread) but, as mentioned, Condense & Distill, like Demmel–Hida and similar meth-

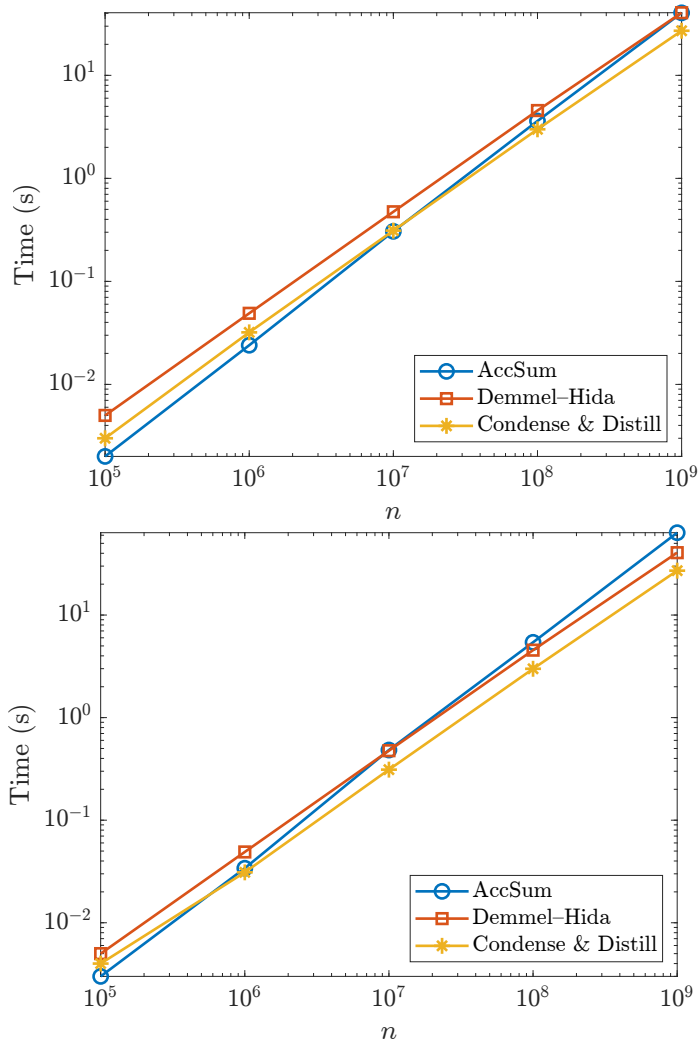

FIG. 4.2. *Comparison between the Demmel–Hida, AccSum, and Condense & Distill algorithms, as a function of the number of summands n and for two condition numbers $\kappa = 10^{30}$ (top) and $\kappa = 10^{60}$ (bottom). All algorithms are run sequentially (1 thread).*

ods, is very amenable to parallelism. We have implemented a parallel version of Condense & Distill, which can exploit p threads by splitting the summands into p blocks and condensing each block in parallel. This yields a condensed sum with at most $4Lp$ summands, which can be sequentially condensed into an even smaller sum with at most $4L$ summands, before being sequentially distilled. Figure 4.3 analyzes the parallel scaling of this method with a varying number of threads from 1 to 36. We consider both strong scaling (right plot, with fixed $n = 10^8$) and weak scaling (left plot, with $n = 3 \times 10^6$ per thread). The method exhibits near perfect scaling, as expected.

As mentioned, similar scaling can also be expected from the Demmel–Hida method. In contrast,

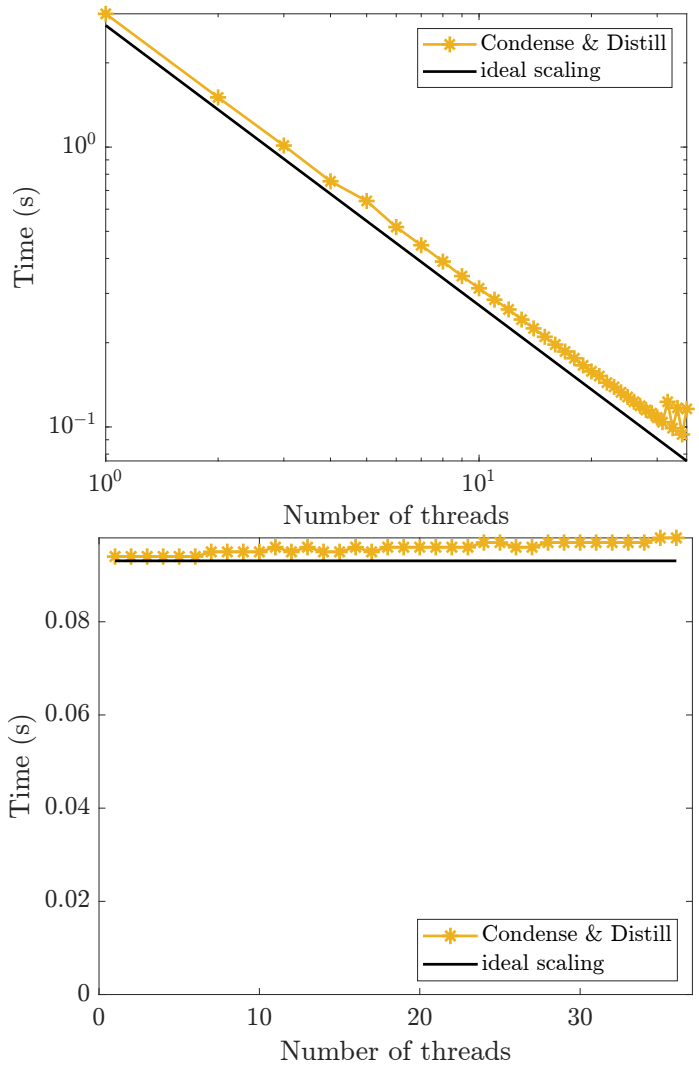

FIG. 4.3. *Parallel scaling of Condense & Distill, using from 1 to 36 threads. Top: strong scaling ($n = 10^8$ is fixed). Bottom: weak scaling ($n = 3 \times 10^6$ per thread).*

AccSum and similar methods offer much less parallelism. We do not study the parallel scaling of AccSum here, but refer to [5], which shows that AccSumK can achieve at best a parallel efficiency of only 50%. Therefore, in a parallel setting, we can expect the performance comparison between Condense & Distill and AccSum to be even more in favor of the former, even for small values of κ.

4.4. Quadruple precision as the working precision. We finally illustrate how condensation can be even more beneficial in the case where the working precision is quadruple precision, which may for example be necessary for applications requiring a high level of accuracy. In this situation Condense & Distill is clearly at an advantage compared with both Demmel–Hida and

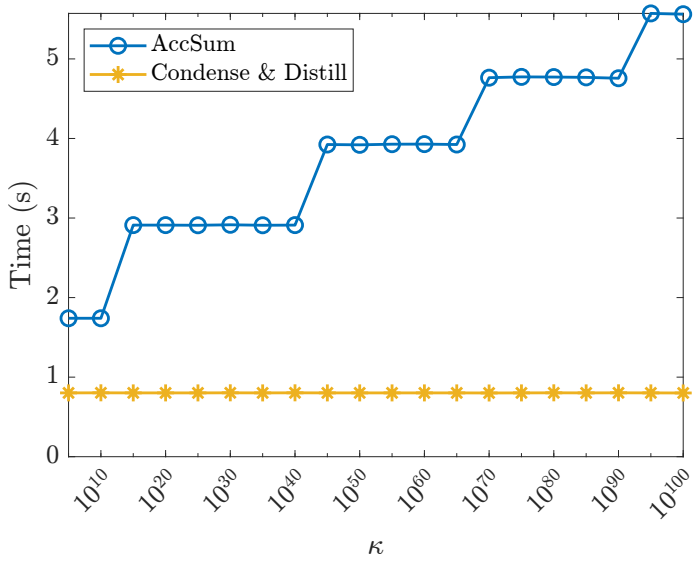

FIG. 4.4. *Comparison between AccSum and Condense & Distill using quadruple precision as the working precision ($n = 10^7$, both algorithms are executed using 1 thread).*

AccSum. Indeed, since Demmel–Hida requires extended precision, using quadruple precision as the working precision would require access to an even higher precision, which is unavaible on most architectures. The only solution would be to rely on an arbitrary precision library, but this would likely be very expensive and we do not explore this option further. As for AccSum, it can easily be executed in quadruple precision since it also runs entirely in the working precision. However, the cost comparison with Condense & Distill tips even more in favor of the latter, because the relative cost of accessing the summands exponent is smaller with respect to the cost of arithmetic operations (which are much more expensive in quadruple precision). This is illustrated in Figure 4.4, which shows that Condense & Distill achieves even larger speedups with respect to AccSum, and even for small condition numbers.

5. Conclusion. We have proposed a new distillation method, Condense & Distill (Algorithm 3.1), which employs a preprocessing condensation step to turn a large ill-conditioned sum into a still ill-conditioned, but far smaller sum, which is then distilled inexpensively via traditional distillation methods. The condensation step relies on Corollary 2.3, which proves that floating-point numbers with the same exponent and least significant bit can be added exactly. Compared with other summation methods that also require accessing the exponent field of the summands, such as the Demmel–Hida method [1], Condense & Distill can run entirely in the working precision. As a result, Condense & Distill is faster, and is also more portable since it does not require any extra precision to be available. Compared with distillation methods based on error-free transformations, such as AccSum [8], Condense & Distill's cost does not increase with the conditioning, and exhibits much better parallel scaling. Overall, we have thus shown Condense & Distill to be an efficient method to distill large ill-conditioned sums.

Acknowledgments. We thank Massimiliano Fasi and Mantas Mikaitis for a discussion at the 2022 Creativity workshop in Manchester, organized by Nick Higham and Dennis Sherwood, that led to the observation that numbers with the same exponent and least significant bit can be added exactly.

Funding. This work was partially supported by the InterFLOP (ANR-20-CE46-0009), NuSCAP (ANR-20-CE48-0014), and MixHPC (ANR-23-CE46-0005-01) projects of the French National Agency for Research (ANR).

REFERENCES

[1] J. Demmel and Y. Hida, *Accurate and efficient floating point summation*, SIAM Journal on Scientific Computing, 25 (2004), pp. 1214–1248.
[2] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second ed., 2002.
[3] U. W. Kulisch and W. L. Miranker, *The arithmetic of the digital computer: A new approach*, SIAM Review, 28 (1986), pp. 1–40.
[4] M. Lange, *Toward accurate and fast summation*, ACM Trans. Math. Softw., 48 (2022).
[5] X. Lei, T. Gu, S. Graillat, H. Jiang, and J. Qi, *A fast parallel high-precision summation algorithm based on accsumk*, Journal of Computational and Applied Mathematics, 406 (2022), p. 113827.
[6] J.-M. Muller, N. Brunie, F. de Dinechin, C.-P. Jeannerod, M. Joldes, V. Lefèvre, G. Melquiond, N. Revol, and S. Torres, *Handbook of floating-point arithmetic*, Birkhäuser/Springer, Cham, second ed., 2018.
[7] S. M. Rump, *Ultimately fast accurate summation*, SIAM Journal on Scientific Computing, 31 (2009), pp. 3466–3502.
[8] S. M. Rump, T. Ogita, and S. Oishi, *Accurate floating-point summation part i: Faithful rounding*, SIAM Journal on Scientific Computing, 31 (2008), pp. 189–224.
[9] S. M. Rump, T. Ogita, and S. Oishi, *Fast high precision summation*, Nonlinear Theory and Its Applications, IEICE, 1 (2010), pp. 2–24.
[10] P. H. Sterbenz, *Floating-point computation*, Prentice-Hall Inc., Englewood Cliffs, N.J., 1974. Prentice-Hall Series in Automatic Computation.
[11] Y.-K. Zhu and W. B. Hayes, *Correct rounding and hybrid approach to exact floating-point summation*, SIAM J. Sci. Comput., 31 (2009), pp. 2981–3001.
[12] Y.-K. Zhu and W. B. Hayes, *Algorithm 908: Online exact summation of floating-point streams*, ACM Transactions on Mathematical Software (TOMS), 37 (2010), pp. 1–13.