Accurate Parallel Floating-Point Accumulation

Edin Kadric, *Student Member, IEEE*, Paul Gurniak, and André DeHon, *Member, IEEE*

Abstract—Using parallel associative reduction, iterative refinement, and conservative early termination detection, we show how to use tree-reduce parallelism to compute correctly rounded floating-point sums in $O(\log N)$ depth. Our parallel solution shows how we can continue to exploit the scaling in transistor count to accelerate floating-point performance even when clock rates remain flat. Empirical evidence suggests our iterative algorithm only requires two tree-reduce passes to converge to the accurate sum in virtually all cases. Furthermore, we develop the hardware implementation of two residue-preserving IEEE-754 double-precision floating-point adders on a Virtex 6 FPGA that run at the same 250 MHz pipeline speed as a standard adder. One adder creates the residue by truncation, requires only 22 percent more area than the standard adder, and allows us to support directed-rounding modes and to lower the cost of round-to-nearest modes. The second adder creates the residue while directly producing a round-to-nearest sum at 48 percent more area than a standard adder.

Index Terms—Floating-point arithmetic, IEEE-754, parallel, accumulation, accuracy, correct rounding, FPGA

1 Introduction

Microprocessor clock speeds grew exponentially during the VLSI era until the mid 2000's. Limits to pipelining [1] and power density ended processor clock scaling [2], [3]. Nonetheless, Moore's Law feature size scaling continues to deliver an exponential growth in transistors per integrated chip. To increase performance, we now also have to increase parallelism instead of simply increasing clock speed.

The end of frequency scaling presents a challenge for floating-point (FP) arithmetic. In this paper, we focus on the problem of summing N floating-point values. With integer arithmetic, we can exploit the associativity of addition to reorder the sum into a tree of depth $O(\log N)$ that admits significant parallelism. However, since floating-point addition is not associative, performing a similar reordering for the floating-point sum would produce a *different* result.

In consequence, floating-point addition must normally be performed sequentially as originally specified in the source code. This is a choice to sacrifice parallelism in order to maintain the determinism of the answer, an approach that not only fails to address the current need for increased parallelism, but also fails to provide guarantees on the accuracy of the answer. Although the sequential sum operation always gives the same result, it may still be inaccurate.

We extend our preliminary work [4] where we developed an algorithm for parallelizing floating-point addition while still guaranteeing a correctly rounded result. That is, we simultaneously address parallelism, accuracy, and determinism, by formulating the basic summation as one that computes a correctly rounded result. This directly addresses the IEEE-1788-2015 requirement for correctly rounded sum and dot-product reductions. We use an iterative, convergent tree-reduce summation that distills the sum (Section 3). *Distillation* reduces the original sequence to a lossless representation of the sum using a constant number of non-zero floating-point values (Section 4). Using a novel bound on residues, we can almost always determine the correctly rounded result long before the distillation completes (Section 5)—completing in only two iterations in virtually all cases (Section 8.1). We prove that the algorithm always terminates (Section 6.2). Our algorithm has $O(\log N)$ depth, providing a significant speed improvement over the commonly used sequential summation technique, with only twice the delay of the non-deterministic parallel one. Our contributions include:

(1) efficient early termination detection for accurate parallel reduction for all IEEE-754 rounding modes;
(2) residue feedback structure for the parallel reduction that guarantees eventual distillation;
(3) residue-preserving floating-point adder using only 22 percent more area than a standard floating-point adder;
(4) generalization of approach to arbitrary floating-point precision and exponent range;
(5) proper handling to avoid spurious overflows;
(6) characterization of the iterations required for convergence based on the condition number of the sum.

2 Background

2.1 Non-Associativity of Floating-Point

Floating-point addition is non-associative since the information lost after each operation depends on the order of the operations. Hence, compilers, high-level synthesis tools and modern implementations are often forbidden from transforming floating-point operations to avoid producing a *different* result from the source program. Unfortunately, the original versions were often written as sequential summations because it was most straightforward to capture the summation as a loop, and when hardware was more expensive, there was little concern for parallelism.

• *The authors are with the Department of Electrical and Systems Engineering, University of Pennsylvania, 200 S. 33rd St., Philadelphia, PA. E-mail: {ekadric, pgurniak}@seas.upenn.edu, andre@acm.org.*

Manuscript received 31 July 2015; revised 15 Jan. 2016; accepted 11 Feb. 2016. Date of publication 22 Feb. 2016; date of current version 14 Oct. 2016. Recommended for acceptance by A. Nannarelli.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2016.2532874

2.2 Correct Rounding

Given N floating-point numbers x_i, our goal is to compute the correct rounding, S_c, of the exact sum $S_r = \sum_{i=1}^{N} x_i$. Correct rounding means that if S_r is a floating-point number, then S_c will be its exact value, and if it is not, S_c will be one of the immediate floating-point neighbors of S_r, chosen according to the rounding mode used. That is, S_c is the value we would get if we computed S_r with infinite precision, then rounded it to the appropriate floating-point number. Note that correct rounding defines correctness of the result not in terms of the order of operations but in terms of the error introduced in the final result. This gives us freedom to transform the computation for additional parallelism as long as we can guarantee to achieve the correctly rounded result. Faithful rounding is similar, except that when S_r falls between two neighboring floating-point numbers, either can be chosen. The literature shows sequential software algorithms that compute both faithfully and correctly rounded sums using standard FPUs [5], [6], [7].

2.3 Residue-Preserving Addition

Most of the accurate accumulation software algorithms such as [5], [6], [7] rely on the same basic building block that was studied in detail by Kornerup et al. [8], a floating-point adder with residue (FPAR), which computes:

$$s = \mathrm{IEEE754RoundToNearest}(a + b), \tag{1}$$

$$r = (a + b) - s. \tag{2}$$

This returns the sum of two floating-point numbers, s—the same sum one would get from a normal floating-point addition—as well as the error (or residue), r, resulting from the operation, which is known to also be representable as a floating-point value when rounding to nearest [9]. With the residue preserved, the following invariant holds:

$$s + r = a + b. \tag{3}$$

To also support directed rounding, or simply to reduce implementation costs when rounding to nearest, we consider an FPAR with Truncation (FPART) and suggest performing the addition operations with truncation until the end of the summation algorithm. When a and b overlap, the FPART performs addition with truncation, which means that the pair (s, r) is such that s carries the MSBs of $a + b$, and r carries the remaining bits (s and r have the same sign when non-zero), and no effort is made to round s in a standard way based on the value of r. When a and b do not overlap, s gets the input with larger magnitude and r the smaller. In either case, the invariant in Eq. (3) still holds.

The FPAR building block can be implemented on a standard IEEE-754 FPU, for instance with Møller-Knuth's `TwoSum` algorithm [10]:

$(s, r) = TwoSum(a, b)$: //implements FPAR
$s = a + b$
$b' = s - a$
$a' = s - b'$; $\delta_b = b - b'$
$\delta_a = a - a'$
$r = \delta_a + \delta_b$.

Kornerup et al. [8] show that `TwoSum` is minimal, both in terms of number of operations (six) and depth of the dependency graph (five). They argue that algorithms with fewer floating-point operations that also require branching (e.g., [9]) are inferior, due to possible drastic performance losses after a mispredicted branch causing the instruction pipeline to drain. Previous work suggested that a hardware implementation of the FPAR could be faster with only a minor area overhead [11], [12], as it avoids the overhead of `TwoSum` due to FPUs, where residual bits computed early are discarded and then recomputed during later stages. Manoukian and Constantinides [13] showed an FPGA implementation of the FPAR for single-precision arithmetic. In Section 7, we provide a fast, native hardware implementation of the FPAR and FPART modules for double-precision.

2.4 Exploiting Hardware

There have been previous attempts at accelerating floating-point accumulation using specialized hardware. For example, Luo and Martonosi implement delayed addition techniques [14]. They reorder the addition operations but do not address the fact that the results would be different from the original summation (as well as inaccurate). Kapre and DeHon demonstrate that an iterative approach can be used to exploit parallelism and still generate the same result as a sequential sum [15]. This introduces determinism in the parallel approach, but it spends more time to reach a solution that is still not correct. Demmel and Nguyen's schemes [16], [17] also guarantee determinism, but not correctness. They trade off computation efficiency and accuracy. Leuprecht and Oberaigner [18] use a parallel reduce tree and recycle residues to compute an accurate sum, employing two full floating-point operations per sum per iteration to compute upper and lower bounds on the sum to detect convergence.

Similar to [18], our work addresses the accuracy and parallelism issues at the same time: We speed up computations by exploiting parallelism in what is commonly thought to be a non-parallel operation while guaranteeing that the computed result is correctly accumulated and rounded. We use a tree-like structure in the spirit of Leuprecht and Oberaigner [18] but with a more efficient convergence test that can be adapted for all IEEE-754 rounding modes.

3 Parallel Accumulation Strategy

Fig. 1 is a high-level illustration of our strategy for accurate parallel summation; it works as follows:

1) Use the truncation-based, floating-point adder (FPART) (Section 7.3) to perform additions preserving residues; this produces both a sum output s and a residue output r.
2) Perform a parallel tree-reduce sum on the sum output of the FPART with $\lceil \log N \rceil$ stages and $N - 1$ nodes; keep the residue output of each FPART for potential refinement and error estimation.
3) Perform the same parallel tree reduce on the FPART residues and compute an early termination condition—a conservative upper bound on the magnitude of the sum of residues, and, hence, potential error in our calculation (Section 5).
4) Iterate reducing the residues and refining the most significant output until our early termination condition indicates that the residues can be ignored.

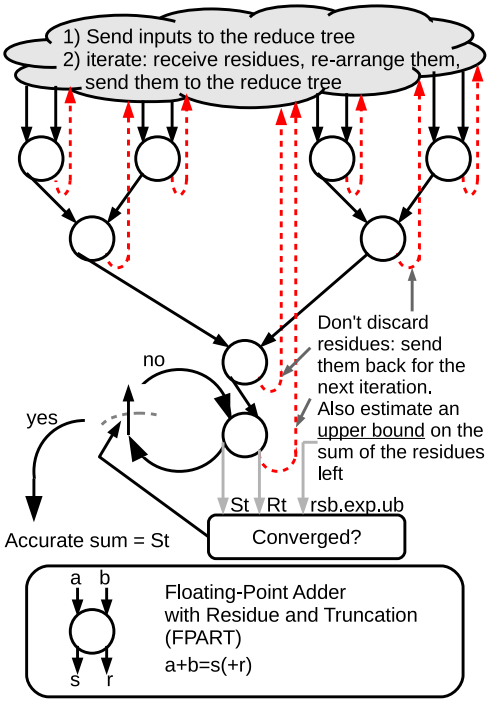

Fig. 1. Parallel accumulation strategy: accumulate using a reduce tree and feed back residues (errors): no information is discarded until the end.

Fig. 2 shows our summation strategy using an $N = 8$ input example with mantissa of size $p = 3$. In this example, a sequential software sum yields the inaccurate result of 1.00e4. A simple parallel reduce tree yields 1.00e5, a different, also inaccurate, result. Our approach yields the accurate result of 1.10e5. Our early termination detection allows us to determine that the non-zero residues left in the tree ($1.00e0$ and $-1.00e0$) are too small to affect the sum of the main outputs of the two tree-reduce steps (1.00e5+1.00e4=1.10e5). We thus converge after two iterations.

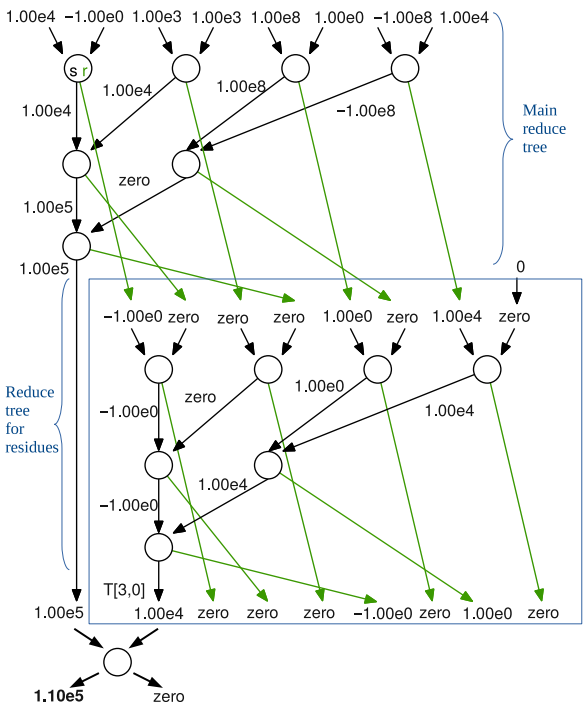

Fig. 2. Example summation through our reduce tree ($p = 3$).

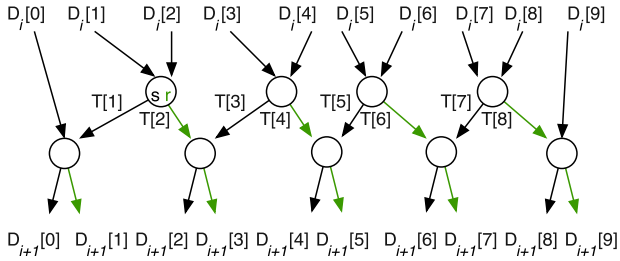

Fig. 3. Distill chain.

We arrange the feedback of residues in the tree to guarantee that they are eventually *distilled* (Section 4.1) to a fixed-length sequence of floating-point values that will perfectly represent the sum to provide more easily provable guarantees on convergence (Section 4.2). We also identify difficult, but typically uncommon, cases that are not resolved with our conservative upper bound, and we organize the sum to guarantee their eventual resolution (Section 5.2).

4 Distillation and Reduction

In this section, we describe the structure of the parallel reduce tree that also performs distillation. We first introduce a distill chain that accumulates the exact sum by representing the complete sum as a collection of floating-point values, the distillation [19], [20]. We argue that the distill chain will always converge, and we identify loose bounds on the cycles required for convergence (Section 4.1). We then show that we can integrate distillation into a reduce tree (Section 4.2).

To make our results general, we use p to define the number of mantissa bits (53 for IEEE double-precision floating-point, 24 for single-precision, including the implicit 1) and e to define the number of exponent bits (11 for double-precision and 8 for single-precision).

4.1 Distill Chain

We can completely represent any sum of floating-point values, within the limits of the floating-point precision range, using a set of non-overlapping floating-point values:

$$Sum = \sum_{0 \le i < L_d} D[i]. \tag{4}$$

In the worst case, we need enough mantissa bits to cover the entire floating-point range. The range is from the p bits at the minimum exponent, $emin$, of $-2^{e-1} + 2$ to the maximum exponent, $emax$, of $2^{e-1} - 1$, for a total of $2^e + p - 3$ mantissa bits. Each floating-point value provides p bits, so we can cover the entire range with L_d floating-point numbers:

$$L_d = \left\lceil \frac{2^e + p - 3}{p} \right\rceil = \left\lceil \frac{2^e - 3}{p} \right\rceil + 1. \tag{5}$$

For IEEE double-precision floating-point, $L_d = 40$.

Claim. We can distill any sequence of N floating-point numbers into L_d floating-point numbers using the distill chain (Figs. 3 and 4).

The distill chain is arranged as an odd-even transposition sort [21] using FPART units to sort by exponent instead of sort units. When the mantissa ranges of the inputs to the FPART

```
DISTILLCHAIN(Input):
for i = 0 to N-1 do
  D[i] ← Input[i]
repeat
  NoChange ←true
  for i = 0 to ⌊(N − 1) /2⌋ do
    T[2i+1],T[2(i+1)]←FPART(D[2i+1],D[2(i+1)])
    NoChange←NoChange∧(T[2i+1]=D[2i+1])
    NoChange←NoChange∧(T[2(i+1)]=D[2(i+1)])
  if even(N) then
    T[N-1]=D[N-1]
  T[0]=D[0]
  for i = 0 to ⌊N/2⌋ do
    D[2i],D[2i+1]←FPART(T[2i],T[2i+1])
    NoChange←NoChange∧(D[2i]=T[2i])
    NoChange←NoChange∧(D[2i+1]=T[2i+1])
  if odd(N) then
    D[N-1]=T[N-1]
until NoChange
```

Fig. 4. Distill chain.

do not overlap, $(\max(A.exp, B.exp))-(\min(A.exp, B.exp)) > p-1$, the FPART will simply sort the results by the magnitude of the exponent. The sort is arranged to eventually order the components from largest exponent ($D[0]$) to smallest ($D[N-1]$). It uses N FPARTs to perform distillation with $\lfloor \frac{N-1}{2} \rfloor$ in a first stage and $\lfloor \frac{N}{2} \rfloor$ in a second stage as shown in Fig. 3. The first stage pairs odd elements, $2i+1$, with their larger even neighbor at $2(i+1)$, while the second stage pairs odd elements, $2i+1$, with their smaller even neighbor at $2i$. Fig. 4 defines the operation of the distill chain.

To understand how the chain works, we can first think of it as an exponent sorter. On each cycle, two neighbors are compared. If they are in the appropriate order, the exponent of the value in the smaller position is larger than the exponent of the value in the larger position: they remain in the same position. If they are out of order, the positions are swapped to place them in order. The two stages guarantee that each cycle through the distill chain will compare each value with its left and right neighbors, allowing it to move one position to the left or right in each cycle. If the elements are properly ordered, then no values change positions in a cycle. In the worst case, a value moves from one end of the chain to the other, a total of N positions.

Now consider the FPART units instead of sort units. On each cycle, we add two floating-point values. If the values do not overlap, the FPART simply sorts the values by the exponent. When the values overlap, the FPART adds the values and the sum output, s, has a larger exponent than the residue output, r, so the outputs are also in sorted order. However, unlike a sorter, the values that come out can be very different from the ones that went in. Due to cancellation, they could both have smaller exponents. When the signs of the values are the same, we can end up with a larger exponent for the s output and a smaller one for r. Either case may leave the results unsorted within the distill chain. However, if we continue to cycle the chain, the values will be moved to their proper position where they will potentially interact to produce a new sum and residue.

To get a weak bound on the number of cycles until the distill chain converges, we can reason about the sum of the exponents in the distill chain and the maximum number of cycles to sort the values into order or into a position that allows overlap and hence intersection. We establish convergence by arguing that one of two things occurs on every cycle: either there is an overlap that results in a reduction in the total weighted exponent sum, or there is no overlap and the distill chain makes progress on sorting the values. If the distill chain manages to only sort the values for N cycles with no overlaps, the distill chain has converged.

We define the weighted exponent sum, $ESUM$, as:

$$ESUM = \sum_{0 \le i < N} (1 - 2^{emin - D[i].exp}). \tag{6}$$

This has the property that the cost of an exponent grows with its magnitude (elements with exponent=$emin$ have value 0, while elements with exponent=$emax$ have value almost 1), so the sum reduces as magnitudes reduce. It also has the property that smaller exponents, when reduced by one, provide a larger reduction than increasing a large exponent by one. This means that, if we transform from $A+B$ to $s+r$ with $A.exp > B.exp$, such that $s.exp = A.exp + 1$ and $r = B.exp - 1$, there is a net reduction in $ESUM$

$$\Delta ESUM = ESUM(s+r) - ESUM(A+B) \tag{7}$$

$$\begin{aligned} &= \Big((1 - 2^{emin-(A.exp+1)}) + (1 - 2^{emin-(B.exp-1)}) \Big) \\ &\quad - \big((1 - 2^{emin-A.exp}) + (1 - 2^{emin-B.exp}) \big) \\ &= 2^{emin-A.exp-1} - 2^{emin-B.exp}. \end{aligned} \tag{8}$$

This is negative since $A.exp > B.exp$.

Each FPART will do one of three things:

1) $(\max(A.exp, B.exp) - \min(A.exp, B.exp) < p-1)$: s will have an exponent at most $\max(A.exp, B.exp)+1$ and r will have an exponent less than or equal to $\min(A.exp, B.exp)-1$, resulting in a reduction in $ESUM$.

2) $(\max(A.exp, B.exp) - \min(A.exp, B.exp) = p-1)$: the r exponent reduces by at least one compared to $\min(A.exp, B.exp)$. The s result may be equal to $\max(A.exp, B.exp)$, be reduced, or be one larger. Even in the case when $r.exp$ is only one smaller than the minimum exponent and $s.exp$ increases the maximum exponent by one, $ESUM$ is reduced as shown above (Eq. (8)).

3) $(\max(A.exp, B.exp) - \min(A.exp, B.exp) \ge p)$: the values will be sorted by the exponents, and $ESUM$ will remain unchanged.

Consequently, on every cycle one of two things happens. Either $ESUM$ is reduced, or the distill chain performs a cycle of sorting the values. To get a weak upper bound on the number of times a reduction can occur, we note that all reductions are larger than $2^{emin-emax}$. If we take $\Delta ESUM_{min} = 2^{emin-emax} = 2^{3-2^e}$ and note that the worst case is reducing $ESUM$ from all values at $emax$ ($ESUM$ slightly below N) to all values at $emin$ ($ESUM = 0$), the bound on the number of reduce operations is:

$$T_{reduce} < \frac{N}{2^{-2^e}} = 2^{2^e} N. \tag{9}$$

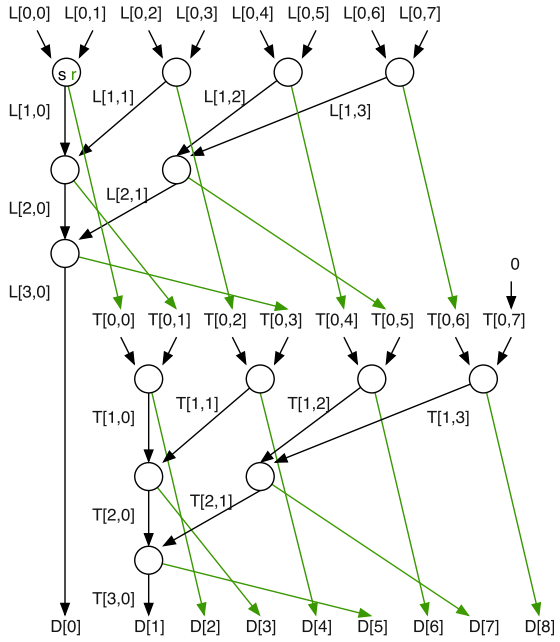

Fig. 5. Distilling tree.

This is the maximum number of times our N sort steps can be interrupted, giving us:

$$T_{distill\_converge} \leq N \times T_{reduce} \leq 2^{2^e} N^2. \tag{10}$$

This weak bound can likely be tightened considerably by exploiting the fact that the minimum reduction is actually much larger than assumed above. Since we avoid running the distill chain to convergence in the results that follow, thanks to early termination detection (Section 5), we leave tightening this result as an open question for future work.

4.2 Distillation Tree

Ignoring residues, a parallel reduce tree computes the sum in $\log(N)$ depth. We can feed the residues back into the tree to sum the residues. What is less obvious is (a) where to route the residues in the tree, and (b) how to guarantee that the residues in the tree will reduce. This is important to our early termination detection (Section 5). The distill chain provides guidance for arranging the residue feedbacks in the tree and guarantees of convergence. The idea is to perform the tree reduce on the s output of the FPART units and arrange the residue connections to guarantee that each pair of passes through the tree performs an odd-even sort step similar to the distill chain. As a result, the tree acts like a distill chain, eventually sorting all the non-zero residues to the low position so that they interact and are sorted into decreasing exponent order. The reduce tree portion accelerates the production of the most significant floating-point residue as with any tree reduce. To achieve the sorting, the basic step is two reduce trees offset by one so that one interacts residues with their nominally larger position neighbor and one with the nominally smaller similar to the odd-even transposition sort in the distill chain (Figs. 5 and 6). Worst case, this will converge like the distillation chain. When there are no changes in values between the input and output of the FPARTs, the distillation tree has converged.

TREEDISTILL($Input$):
for i=0 **to** N-1 **do**
  $L[0,i] \leftarrow Input[i]$; $T[0,i] \leftarrow 0$
// balance out to power of 2 to keep simple
for i=N **to** $2^{\lceil \log_2(N) \rceil}$-1 **do**
  $L[0,i] \leftarrow 0$; $T[0,i] \leftarrow 0$
repeat
  $NoChange \leftarrow$true
  // reduce largest + swap left
  for $l = 1$ **to** $\lceil \log_2(N) \rceil$ **do**
    for $i = 0$ **to** $\lceil \frac{N}{2^l} \rceil - 1$ **do**
      $ti \leftarrow (i \times 2^l + 2^{(l-1)} - 1)$
      $L[l,i], T[0,ti] \leftarrow$ FPART($L[l\text{-}1,2i]$,$L[l\text{-}1,2i+1]$)
      $NoChange \leftarrow NoChange \wedge (L[l,i]{=}L[l\text{-}1,2i])$
      $NoChange \leftarrow NoChange \wedge (T[0,ti]{=}L[l\text{-}1,2i+1])$
  $D[0] \leftarrow L[\lceil \log_2(N) \rceil, 0]$
  // reduce 2nd largest + swap right
  for $l = 1$ **to** $\lceil \log_2(N-1) \rceil$ **do**
    for $i = 0$ **to** $\lceil \frac{N-1}{2^l} \rceil - 1$ **do**
      $ti \leftarrow (i \times 2^l + 2^{(l-1)} + 1)$
      $T[l,i], D[ti] \leftarrow$ FPART($T[l\text{-}1,2i]$,$T[l\text{-}1,2i+1]$)
      $NoChange \leftarrow NoChange \wedge (T[l,i]{=}T[l\text{-}1,2i])$
      $NoChange \leftarrow NoChange \wedge (D[ti]{=}T[l\text{-}1,2i+1])$
  // feedback results
  $D[1] \leftarrow T[\lceil \log_2(N-1) \rceil, 0]$
  for i=0 **to** N-1 **do**
    $L[0,i] \leftarrow D[i]$
until $NoChange$

Fig. 6. Iterative distillation tree.

5 Early Termination Detection

While the details above show that the distill chain and tree will converge in a bounded amount of time, it could take a long time to completely converge. However, we can almost always determine the converged value for the accurate sum long before the distill tree has converged. Section 8 shows cases where early termination detection reduces the number of iterations from 73 down to 2 (Table 1a, $\delta = 1{,}500$, Data1).

5.1 Termination Condition

To determine when we have enough information to return the correctly rounded result, we look at S_t and R_t, the largest two values in the distillation ($D[0]$, $D[1]$ when converged) and gross summary information on the remaining residues in the distill tree to calculate a conservative upper bound to the magnitude of the sum of the remaining residues, rsb, the Residue Sum Bound. We can then check for convergence by testing the condition $\text{CONV}(S_t, R_t, rsb)$ (Fig. 8). If the condition succeeds, we conclude that even an upper bound on what is left in the tree is not sufficient to influence the most-significant floating-point values, S_t, R_t, and they can be rounded to determine the accurate sum. CONV also identifies a third case, which we call *undet*, where it will not be possible to determine convergence by only examining S_t, R_t, and rsb, and a fourth case where we must modify S_t and R_t to drive convergence. We show how to handle these cases in Sections 5.1.4 and 5.2.

TABLE 1
Experimental Results

a) Exponential Distribution of Data (ED=Early Termination Detection; values shown as {mean, standard deviation} pairs)

	Data1 $\kappa = 1$			Data2 $\kappa = 4E1$			Data3 $\kappa = 7E16$			Data4 $\kappa = \infty$		
δ	# its [no ED]	% non-0	seq % err	# its [no ED]	% non-0	seq % err	# its [no ED]	% non-0	seq % err	# its [no ED]	% non-0	seq % err
100	2,0 [25,0.3]	58, 0.9	2E-13, 2E-17	2,0 [23,0.3]	56, 0.9	4E-13, 4E-16	2,0 [2,0]	0, 0	1E2, 4E-2	3,0 [3,0]	0, 0	∞
500	2,0 [31,0.9]	91, 0.5	5E-14, 1E-17	2,0 [31,1.0]	91, 0.5	3E-13, 5E-16	2,0 [2,0]	0, 0	1E2, 2E-2	11,0 [11,0]	0, 0	∞
1000	2,0 [55,2.0]	95, 0.3	4E-14, 7E-18	2,0 [55,2.3]	95, 0.3	1E-13, 2E-16	2,0 [2,0]	0, 0	1E2, 9E-3	21,0 [21,0]	0, 0	∞
1500	2,0 [73,3.1]	97, 0.3	3E-14, 6E-18	2,0 [70,3.4]	97, 0.3	9E-14, 8E-17	2,0 [2,0]	0, 0	1E2, 7E-3	31,0 [31,0]	0, 0	∞

b) Uniform Distribution of Data (values shown as {mean, standard deviation} pairs)

	Data1 $\kappa = 1$			Data2 $\kappa = 3E2$			Data3 $\kappa = 3E17$			Data4 $\kappa = \infty$		
δ	# its [no ED]	% non-0	seq % err	# its [no ED]	% non-0	seq % err	# its [no ED]	% non-0	seq % err	# its [no ED]	% non-0	seq % err
100	2,0 [2,0]	0, 0	1E-13, 2E-17	2,0 [2,0]	0, 0	6E-13, 1E-17	2,0 [2,0]	0, 0	2E2, 3E-1	2,0 [2,0]	0, 0	∞
500	2,0 [2,0]	0, 0	1E-13, 2E-17	2,0 [2,0]	0, 0	5E-13, 5E-16	2,0 [2,0]	0, 0	2E2, 3E-1	2,0 [2,0]	0, 0	∞
1000	2,0 [2,0]	0, 0	1E-13, 2E-17	2,0 [2,0]	0, 0	6E-13, 7E-16	2,0 [2,0]	0, 0	2E2, 3E-1	2,0 [2,0]	0, 0	∞
1500	2,0 [2,0]	0, 0	1E-13, 2E-17	2,0 [2,0]	0, 0	8E-13, 1E-15	2,0 [2,0]	0, 0	2E2, 3E-1	2,0 [2,0]	0, 0	∞

To derive an expression for rsb and CONV, we first define $nzcnt$ as the number of non-zero residues left in the tree and $maxexp$ as the maximum exponent of these residues, both of which can be trivially computed during the summation tree reduction (see end of Section 7.1). We then write an upper bound on the sum of the residues:

$$|error| \leq nzcnt \times 2^{maxexp+1}. \tag{11}$$

By definition, we want rsb to be an upper bound on $|error|$:

$$rsb = nzcnt \times 2^{maxexp+1} \geq |error|. \tag{12}$$

We can now approximate rsb as an exponent only:

$$rsb.exp.ub = \lceil \log_2(nzcnt) \rceil + maxexp + 1. \tag{13}$$

$rsb.exp.ub$ is an upper bound on the exponent of rsb: the smallest integer k such that $2^k \geq rsb$.

The function CONV(S_t, R_t, rsb) is shown in Fig. 8. It takes two non-overlapping inputs $S_t, R_t \leftarrow$FPART($D[0]$,$D[1]$) with $|S_t| \geq |R_t|$ (since S_t and R_t are outputs from an FPART following each distill tree step (see Fig. 14), this relation always holds), as well as rsb. Based on the values of S_t, R_t, and rsb, we are able to decide whether we can round S_t and make it the final converged sum, or whether other iterations of the algorithm are required. Convergence is achieved if and only if the following condition is satisfied:

$$\text{round}(S_t + R_t + rsb) = \text{round}(S_t + R_t - rsb). \tag{14}$$

Furthermore, when convergence fails, CONV distinguishes cases that may converge with additional iterations from "undetermined" (undet) cases. An undet case means that rsb is low enough that S_t would not change by more than 1 ULP (Units in the Last Place), and that it is so low that it may not be able to interact with R_t. Undet cases may not resolve simply by continuing to iterate residue reduction on the distill tree. When an undet case is reached, we modify the values in the summation and change our convergence condition (Section 5.2). We show example strings that cause undet cases in Fig. 9 (with $p = 13$).

To better describe the conditions in Figs. 8 and 9, we introduce a few more definitions. In Fig. 8, $|R_t|$ patterns, patt ($|R_t|$), are shown assuming their MSB starts right after the LSB of S_t (that is, they start at 1/2 ULP); we conceptually pad $|R_t|$ so that it immediately follows S_t. We also define a function lco (Lowest in Chain of Ones), which gives the index of the least significant 1 in the first chain of ones of a string—the most significant of these chains when there are more than one (see Fig. 9). Note that the indices are assumed to increase towards the right, with the first bit not included in S_t being at index 0, the second at index 1, and so on.

$rsb.i$ (rsb index) compares how $rsb.exp.ub$ aligns with $R_t.exp$:

$$rsb.i = S_t.exp - p - rsb.exp.ub, \tag{15}$$

$rsb.i$ marks the dividing line between sum bits that have been resolved and bits that have not. The bits more significant than $rsb.i$ are *resolved* in that they will at most change due to a carry from the resolution of bits at or below $rsb.i$.

5.1.1 CONV Algorithm

Certainly, when rsb is zero, there is no residue to impact the result, so we converge. Otherwise, as shown in Fig. 8, if $rsb.exp.ub \geq R_t.exp$, a different result may be obtained if rsb is added or subtracted, so convergence fails *(i)*. When $rsb.exp.ub < R_t.exp$, we know that we have resolved everything up to the most significant bit of R_t, except for a potential carry. We further require $rsb.i \geq 3$, so at least the first three bits past S_t are resolved (1/2, 1/4 and 1/8 ULP), except for a potential carry *(ii)*; we could relax this requirement but it would complicate the case analysis that follows. With $rsb.i \geq 3$, we can reason about $|R_t|$ patterns that include the top three bits of R_t since we know those bits are resolved and can only be impacted by a carry into the low bit. We distinguish between two rounding mode families: round-to-nearest (Section 5.1.2), and directed rounding (Section 5.1.3). We must also resolve the special case where rounding can reduce the S_t exponent (Section 5.1.4).

5.1.2 Round to Nearest Modes

Assuming we do not need to address a reduction in the S_t exponent, we identify four cases from the $|R_t|$ pattern.

patt($|R_t|$)=00xxx. There are two zeros after the LSB of S_t and before any $|R_t|$ bit is set (at indices 0 and 1). The zero at index 1 is a squash bit that stops carries from propagating into index 0. Hence, no carry can propagate into S_t, and the round bit cannot be set: we can round to nearest *(iii)*.

patt($|R_t|$)=11xxx. If R_t is positive, $R_t - rsb$ is greater than 1/2 ULP so we round up. $R_t + rsb$ is also greater than 1/2 ULP and may carry into S_t but then would leave the remainder below 1/2 ULP, so we round up in this case as well: we converge *(iv)*. A similar reasoning applies if R_t is negative, but we round down.

patt($|R_t|$)=01xxx. Here we have two cases. If $rsb.i > \text{lco}(|R_t|)$, then neither $R_t + rsb$ nor $R_t - rsb$ can cause a carry

```
RESOLVEDROPEXP(S_t,R_t):
Z ← 2^(S_t.exp−p+1)
Z.sign=S_t.sign
S_t,s_0←FPART(S_t, −Z)
R_t,r_0←FPART(R_t, Z)
if R_t.exp = S_t.exp − p + 1 then
  Z_0 ← 2^(S_t.exp−p+1)
  Z_0.sign=S_t.sign
  S_t ← S_t + Z_0
  R_t ← R_t − Z_0
  R_t,r_0←FPART(R_t, r_0)
return S_t,R_t,r_0
```

Fig. 7. Function to resolve special case when rounding.

that will set the bit to the left of the chain of ones: we converge *(v-a)*. Otherwise, if $rsb.i \leq \text{lco}(|R_t|)$, then $R_t + rsb$ and $R_t - rsb$ may or may not cause a carry that changes index 0 in a different way: convergence fails *(v-b)*; this leads to an undet case.

patt($|R_t|$)=10xxx. Again we have two cases: either $rsb.i$ is greater than the index of the first 1 of $|R_t|$ after the 1 in the most significant bit, in which case neither $R_t + rsb$ nor $R_t - rsb$ would have an effect on that 1, and the tie is resolved in the same way: we converge *(vi-a)*. Otherwise, $R_t + rsb$ would make the tie deviate in a different way than $R_t - rsb$, so convergence fails *(vi-b)*, leading to an undet case.

5.1.3 Directed Rounding Modes

Directed rounding modes round in a specific direction (toward 0, toward $+\infty$, toward $-\infty$). In directed rounding, we first check if $rsb = 0$. If it is, we know how to round based on R_t, and we converge *(x)*. Otherwise ($rsb \neq 0$) we check the $|R_t|$ pattern:

patt($|R_t|$)=0xxx. If there is a zero at index 0, then convergence is achieved *(xi)*, since the zero acts as a squash bit, preventing the sum of residues to propagate into S_t. Furthermore, because $rsb.exp.ub < R_t.exp$, we know that the sum of residues, including R_t, will not be zero.

patt($|R_t|$)=1xxx. If there is a one at index 0, we do the same check as above for round-to-nearest: Either $rsb.i > \text{lco}(|R_t|)$, in which case there will be no overflow into S_t, and we know how to round *(xii-a)*, or $rsb.i \leq \text{lco}(|R_t|)$, in which case there could be an overflow, so convergence fails *(xii-b)*. This leads to an undet case.

5.1.4 dropExp

As described above, round-to-nearest modes look at the bit immediately below S_t (index 0, 1/2 ULP), the *round bit*, to determine whether to round up or round down, except in the exceptional case when S_t is a power of two and has a different sign from R_t. In this case, the S_t exponent may drop by 1 due to cancellations in the residues, so we call it dropExp (Fig. 7). A similar dropExp issue can arise for directed rounding modes. We thus first check if we have a dropExp case and transform the dropExp case into a non-dropExp case so it can be resolved with the same steps as the general cases above.

The problem here is the different signs, so our strategy is to resolve S_t and R_t to the same sign, which we can do by borrowing a low bit from S_t when R_t is sufficiently large. If R_t is small (patt($|R_t|$)=000xxx and $rsb.i \geq 3$), both $S_t + R_t + rsb$ and $S_t + R_t - rsb$ round the same way due to the squash bit at index 2 (1/8 ULP), so we converge *(vii)*. Otherwise, we subtract $Z = 2^{S_t.exp-p+1}$ from S_t if S_t is positive, $Z = -2^{S_t.exp-p+1}$ if S_t is negative, i.e., we subtract the least significant bit of S_t, such that it is not a power of two any more—the new S_t has an exponent that is lower by 1, and all 1's in the mantissa, except the least significant bit, to which we assign the value Y, defined below. We also update (R_t,r_0)←FPART(Z, R_t), which flips the sign of R_t, so S_t and R_t now have the same sign. If the new R_t has exponent $S_t.exp - p + 1$ (if the new S_t and R_t overlap), we transfer the most significant bit of R_t to the least significant bit of S_t: we set $Y = 1$, and we update R_t accordingly. Otherwise, if the new R_t has exponent less than $S_t.exp - p + 1$, we set $Y = 0$, and we do not update R_t. Fig. 9 shows an example of this, where *(viii-a)* gets transformed into *(viii-b)*. The 1 bit shown in red is r_0, an extra residue that we may have displaced from R_t by this addition. We add r_0 to the current set of residues. At this point we do not have a dropExp case any more. Note that r_0 will be at most one bit, which can happen when patt($|R_t|$)=00xxx. For it to be 2-bit wide, it would need to come out of the patt($|R_t|$)=000xxx case, but that is a convergence case.

```
CONV(S_t, R_t, rsb):
rsb.i ← S_t.exp-p-rsb.exp.ub
if rsb.exp.ub ≥R_t.exp AND rsb ≠0 then
  return FAIL (i)
else if rsb.i < 3 then
  return FAIL (ii)
else if ((S_t.sign≠R_t.sign) and (|S_t|=2^(S_t.exp)) then
  if patt(|R_t|)=000xxx then
    return CONV (vii)
  else
    return DROPEXP (viii-a, ix-a)
else if RoundMode=ToNearest then
  return NEAREST(S_t, R_t, rsb, rsb.i)
else if RoundMode=Directed then
  return DIRECT(S_t, R_t, rsb, rsb.i)
```

```
NEAREST(S_t, R_t,rsb,rsb.i):
if patt(|R_t|)=00xxx then
  return CONV (iii)
else if patt(|R_t|)=11xxx then
  return CONV (iv)
else if patt(|R_t|)=01xxx then
  if rsb.i>lco(|R_t|) then
    return CONV (v-a)
  else
    return UNDET (v-b)
else if patt(|R_t|)=10xxx then
  tmp ← |R_t| − 2^(R_t.exp)
  if rsb.exp.ub<tmp.exp then
    return CONV (vi-a)
  else
    return UNDET (vi-b)
```

```
DIRECT(S_t, R_t, rsb, rsb.i):
if rsb=0 then
  return CONV (x)
else if patt(|R_t|)=0xxx then
  return CONV (xi)
else
  if rsb.i>lco(|R_t|) then
    return CONV (xii-a)
  else
    return UNDET (xii-b)
```

Fig. 8. Early convergence detection function: CONV(S_t, R_t, rsb).

5.2 Handling Undetermined Cases (Undet)

As Fig. 8 notes, there are a few cases that cannot be directly resolved by examining only S_t, R_t, and rsb. These are cases where we could perform *faithful* rounding based on the information in S_t, R_t, and rsb, but *correct* rounding demands

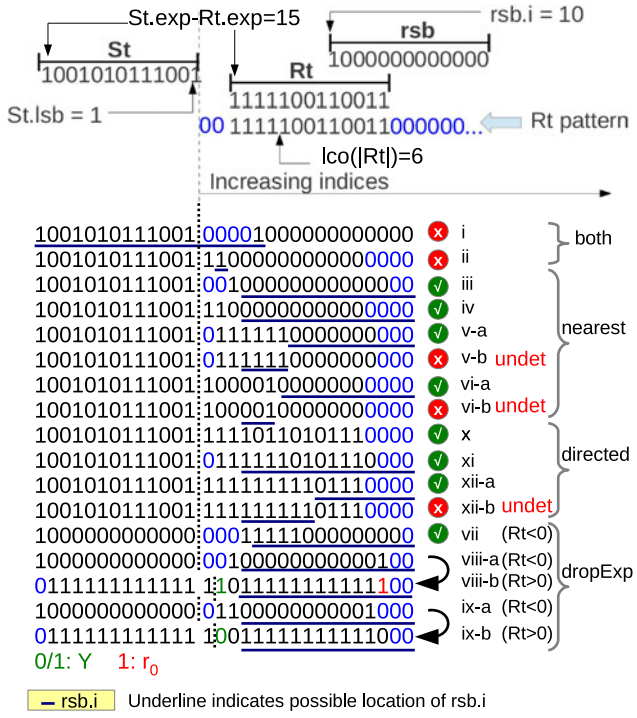

Fig. 9. Example input configurations in the CONV function.

that we know the resolution of bits that are below the ones we have resolved in S_t and R_t. We call these "undet" since we cannot break the tie between rounding up or down based on the information available. If the undet cases are not treated, the early convergence check may always fail once we reach the undet condition, forcing us to (a) run until the distill tree converges and (b) look at all the distilled values to determine the final rounded result.

Consider the following example corresponding to failure *(vii-b)* in Fig. 8, with $p = 5$ and assuming rsb is set by the only residue remaining in the tree:

$$S_t = 1.1010e27, R_t = 1.0000e22, rsb = 1.0000e10$$
$$\text{round}(S_t + R_t + rsb) = 1.1011e27$$
$$\text{round}(S_t + R_t - rsb) = 1.1010e27$$

While convergence is not achieved, R_t is non-overlapping with the remaining residue and no reduction can be performed. Indeed, $1.0000e10$ and $1.0000e22$ could be the only non-zero values in the tree other than S_t and they cannot combine. $1.0000e22$ is ultimately summed with the previous total $1.1010e27$, and again, we end up with:

$$S_t = 1.1010e27, R_t = 1.0000e22, rsb = 1.0000e10$$

That is, we really need to know if the residue sum will be negative, leading to a round down, or positive, leading to a round up. Knowing that would demand that we resolve bits or distilled values lower than R_t, but our simple definition for rsb does not capture that. We could define a more complicated termination condition that kept track of convergence of other elements of the distillation (D[2], D[3],...) and move the approximation in the residue sum to the suffix. However, these cases can be handled more compactly by modifying the sum and keeping a few additional bits of state. This allows us to keep the convergence check simple, looking at only a small, constant number of bits.

```
RESOLVENEARESTUNDET(D, S_t, R_t)
Init: off←1;
if (positive(R_t.sign)) then
    R_ts.positive ←true
    R_t ← R_t − 2^(S_t.exp−p)
else
    R_ts.positive ←false
    R_t ← R_t + 2^(S_t.exp−p)
D[0] ←S_t; D[1] ←R_t
while True do
    D←DISTILLTREESTEP(off,D)
    off←1-off //Toggle 0, 1
    S_t, R_t←FPART(D[0],D[1])
    rsb.exp.ub, nzcnt←RSBEXP(2,D)
    if (R_t.exp > rsb.exp.ub OR nzcnt=0) then
        // sign(R_t + rsb) = sign(R_t − rsb)
        if R_t > 0 then
            return R_ts.positive?rnd_up(S_t) : S_t
        else if R_t < 0 then
            return R_ts.positive?S_t : rnd_down(S_t)
        else if R_t = 0 then
            if R_ts.positive then
                return round(S_t + 2^(S_t.exp−p))
            else
                return round(S_t − 2^(S_t.exp−p))
    else
        Continue; // Eventually true by Th. 1
```

Fig. 10. Nearest rounding undet resolution.

Undetermined cases are resolved differently depending on the rounding mode chosen.

5.2.1 Round-to-Nearest

In round-to-nearest, we start by subtracting a 1 right after the LSB of S_t (index 0 in Fig. 9). That is, we add $-2^{S_t.exp-p}$ to the set of distillation residues if R_t is positive, $2^{S_t.exp-p}$ if R_t is negative. This way, we remove the round bit, and we then only check for the following condition:

$$\text{sign}(R_t + rsb) = \text{sign}(R_t - rsb).$$

Indeed, we now only need to know whether what is left in the tree is positive or negative to determine whether it will shift the tie up or down, and hence whether it will affect the LSB of S_t or not. This prevents the distillation from being clogged, allowing smaller values to interact and resolve within R_t. Fig. 10 shows the complete algorithm for resolving the undet case in round-to-nearest mode.

In fact, this handling of undet conditions deals with the case that Kornerup et al. use to prove the impossibility of correctly rounding the sum of three or more floating-point numbers using a round-to-nearest, ties-to-even approach with depth less than 1,939 double-precision IEEE-754 operations [8]. We are able to cover such a depth with limited resources because repeated iterations allow us to dynamically increase the DAG depth until termination, exceeding 1,939 if needed.

```
RESOLVEDIRECTEDUNDET(D)
Init: off←1
// Only called if R_t.exp=S_t.exp-p and rsb.exp.ub<R_t.exp
and R_t.sign=S_t.sign
if (positive(R_t.sign)) then
  potential_carry ← −2^(S_t.exp−p+1)
else
  potential_carry ← 2^(S_t.exp−p+1)
while (D[N-1] ≠ 0) do
  // eventually true; see comment in CORRECTROUND
  D←DISTILLTREESTEP(off,D)
  off←1-off //Toggle 0, 1
D[N-1] = potential_carry
S_t, R_t←FPART(D[0],D[1])
while True do
  rsb.exp.ub, nzcnt←RSBEXP(2,D)
  // always wait for resolution of top bit of R_t
  if (R_t.exp > rsb.exp.ub OR nzcnt=0) then
    // sign(R_t + rsb) = sign(R_t − rsb)
    if sign(R_t ± rsb) > 0 then
      return round for value above S_t
    else if sign(R_t ± rsb) < 0 then
      return round for value below S_t
    else
      return S_t // R_t = rsb = 0
  else
    Continue; // Eventually true by Th. 1
  D←DISTILLTREESTEP(off,D)
  off←1-off //Toggle 0, 1
  S_t, R_t←FPART(D[0],D[1])
```

Fig. 11. Directed rounding undet resolution.

5.2.2 Directed Rounding

The problem in the undet directed rounding case *(xii-b)* is that we may have a chain of ones following S_t and into R_t at least up to the position that could be affected by the rsb. So, if S_t and R_t are positive and rsb is the maximum possible positive value, it is possible there will be a carry into S_t, but if rsb is the maximum possible negative value, there will not be. As a result, we cannot determine the rounding. Even if we continue distillation, the chain of 1's could extend through many distilled terms.

The question here is whether or not R_t and the remaining residues carry a 1 into S_t. We can use a similar trick to the nearest rounding case by assuming the carry and adding a 1 to the LSB of S_t. To determine if this was correct, we compensate by adding a -1 LSB to the residue. If the residue sign is positive, it turns out we were correct to treat the carry, and keep it. However, if the residue sign comes out negative, we know that R_t and the residue would not have generated the carry, so we remove it. Thus, this case, too, can be reduced to determining the sign of the residue.

6 Detailed Algorithm and Convergence

The last three sections have described all the components of our accurate summation. This section addresses the technicalities of how it all works, showing how the components come together (Section 6.1), proving that this full formulation will converge (Section 6.2), and dealing with the case of potential overflow (Section 6.3).

```
DISTILLTREESTEP(first,Input):
for i=0 to first-1 do
  R[i]←Input[i]
for i=0 to N-1-first do
  L[0, i] ← Input[i + first]
// balance out to power of 2 to keep simple
for i=(N-first) to 2^⌈log2(N)⌉-1 do
  L[0, i] ← 0
for l = 1 to ⌈log2(N)⌉ do
  for i ← 0 to ⌈N/2^l⌉ − 1 do
    ti ← i × 2^l + 2^(l−1)+first
    L[l, i], R[ti] ← FPART(L[l-1,2i],L[l-1,2i+1])
R[first]←L[⌈log2(N)⌉,0]
return R
```

Fig. 12. Distill tree step.

6.1 Algorithm

The complete, correctly-rounded parallel accumulation algorithm is shown in Figs. 7 and 8, and Figs. 10 through 14. The main loop in Fig. 14 calls a tree reduction (Fig. 12) and residue computation (Fig. 13) on every iteration, before checking for convergence (Fig. 8) and returning the result. If convergence fails, we proceed with another iteration until it is successful. It also identifies the undet conditions and handles their resolution separately (Figs. 10 and 11).

6.2 Termination Condition Convergence

In this section we prove the theorem that establishes that the early termination detection conditions defined in Section 5.1 and used in the correctly rounded sum algorithm (Fig. 14) and the undet resolution algorithms (Figs. 10 and 11) will always be met, eventually. The bound on the convergence of the distillation tree (Section 4.2) provides a bound on the number of iterations it requires to reach these conditions.

Since the tree distills its inputs, we know that the largest exponent, $maxexp$, outside of the most significant two values, S_t and R_t, will eventually not overlap with R_t. So, we know the following will eventually hold:

$$maxexp \leq R_t.exp - p. \tag{16}$$

Here, if R_t is a subnormal, we take $R_t.exp$ as the exponent of the most significant 1.

Also due to distillation, the total number of non-zeros will be at most L_d. Excluding S_t and R_t, this leaves:

$$nzcnt \leq L_d - 2. \tag{17}$$

Theorem 1. *If $p > e + 3$, eventually, $rsb.exp.ub < R_t.exp - 2$.*

```
RSBEXP(first,Input):
maxexp ← −∞; nzcnt ← 0
for i ← first to N-1 do
  if Input[i] ≠ 0 then
    maxexp ← max(maxexp, (Input[i].exp))
    nzcnt++
rsb←nzcnt × 2^(maxexp+1)
rsb.exp.ub←⌈log2(nzcnt)⌉+maxexp+1
return rsb.exp.ub, nzcnt
```

Fig. 13. Residue summary.

```
CorrectRound(Input)
Init: off←0
while True do
  D←DistillTreeStep(off,D)
  off←1-off //Toggle 0, 1
  S_t, R_t←FPART(D[0],D[1])
  rsb.exp.ub←RSBExp(2,D)
  if CONV(S_t, R_t, rsb)=CONV then
    return round(S_t, R_t)
  else if CONV(S_t, R_t, rsb)=DROPEXP then
    if (D[N-1] ≠ 0) then
      Continue; // when N > L_d, distillation eventually
      guarantees this becomes true; when N ≤ L_d, use
      an (N+1)-tree.
    else
      D[0],D[1],D[N-1]←ResolveDropExp(S_t,R_t)
      Continue; //We know rsb.i ≥ 3 and patt(|R_t|) was
      not 000xxx, so once we transform R_t to have the
      same sign as S_t, no resolution of the residue will
      flip it back. This case only happens once.
  else if CONV(S_t, R_t, rsb)=UNDET then
    //residues can only affect the LSB of S_t
    if DirectedRounding then
      return ResolveDirectedUndet(D, S_t, R_t)
    else
      return ResolveNearestUndet(D, S_t, R_t)
  else if CONV(S_t, R_t, rsb)=FAIL then
    Continue; //Reduces to UNDET or CONV by Th. 1
```

Fig. 14. Correctly rounded sum algorithm.

Proof. By definition (Eq. (13)):

$$rsb.exp.ub = \lceil \log_2(nzcnt) \rceil + maxexp + 1.$$

Combining the bounds on $maxexp$ and $nzcnt$:

$$rsb.exp.ub \leq \lceil \log_2(L_d) \rceil + R_t.exp - p + 1.$$

$rsb.exp.ub < R_t.exp - 2$ will hold as long as:

$$\lceil \log_2(L_d) \rceil - p + 1 < -2, \tag{18}$$

$$\left\lceil \log_2\left(\left\lceil \frac{2^e - 3}{p} \right\rceil + 1 \right) \right\rceil - p + 3 < 0.$$

Since $p \geq 1$, the first term can be bounded as:

$$\left\lceil \log_2\left(\left\lceil \frac{2^e - 3}{p} \right\rceil + 1 \right) \right\rceil \leq \lceil \log_2(2^e - 3 + 1) \rceil \leq e. \tag{19}$$

Substituting Eq. (19) into Eq. (18), this gives:

$$e - p + 3 < 0.$$

This final equation reduces to $p > e + 3$. □

Together with distillation convergence and the treatment for undet cases, this guarantees that we will always be able to detect convergence by looking at S_t, R_t, $nzcnt$ and $maxexp$. $rsb.exp.ub < R_t.exp - 2$ guarantees that the $rsb.i \geq 3$ condition eventually holds, so we can safely wait on it to become true before making an undet decision.

Note that both IEEE single- and double-precision meet the $p > e + 3$ requirement. IEEE half-precision (binary16, $e = 5$ and $p = 11$) and quadruple-precision (binary128, $e = 15$ and $p = 113$) also meet the $p > e + 3$ requirement. In general, the other binary interchange formats also meet the $p > e + 3$ requirement since IEEE 754-2008 requires $e = round(4\log_2(k)) - 13$ where $k = p + e$ ($k \geq 128$). For floating-point number systems where $p \leq e + 3$, this just means that we must examine more residues as part of the largest residue, R_t. The generalization is to separate out k values, $R_{t0}, R_{t1}, \ldots, R_{t(k-1)}$. $maxexp$ and $nzcnt$ are computed on the residues smaller than $R_{t(k-1)}$, and the revised CONV must look at all k of the R_{ti} values. k is set such that $k = \left\lceil \frac{e+3}{p} \right\rceil$.

6.3 Special Case: Overflow

Although the FPA, FPAR and FPART modules we describe in Section 7 support infinity, the algorithm we have detailed so far does not handle *intermediate* overflows. In particular, an unfortunate ordering of the inputs could cause an intermediate sum to overflow to infinity, driving the final sum to infinity in cases where the accurate sum is finite and can be represented in the given floating-point system.

Using the distill tree, we can handle this case as follows: When the addition of two numbers would overflow, instead of producing $\pm\infty$, we make the FPART return the outputs in sorted order (larger input as s, smaller input as r). This way, no information is lost. The distillation and sorting continue. If the distillation eventually produces values that would cancel with these values, the sorting would bring them together, cancellation would occur, and the distillation can converge normally. Our early convergence detection scheme will work once proper cancellation of these large values is allowed to occur. However, if we reach a steady state where all numbers are in stable, sorted order and there are still adjacent entries that would overflow, we know that no cancellation will occur—every pair of adjacent numbers has gone through an FPART, meaning there must be no adjacent, overlapping numbers with opposite signs. In this steady state, we can correctly determine that the result is $\pm\infty$. We can detect this by having each FPART provide an output signal to indicate when it has performed a sort to avoid overflow. If we reach the distillation tree termination condition and any of the FPARTs has asserted this signal, we can conclude that a real overflow has occurred. These intermediate overflows cause more iterations on average. This technique allows us to provide correct operation in all cases, without complicating the implementation. In typical usage, we expect this to be an uncommon case.

7 Add with Residue Unit

The key operation in our tree reduce is the FPART unit. For completeness, this section describes the design of the FPART unit and quantifies its resources. We start by describing the base FPA design and its extension to an FPAR that supports "round-to-nearest, ties-away-from-0", the most complex of the IEEE-754 rounding modes. Independent of our development of the FPAR [4], other groups have developed FPAR units with similar resource requirements [13], [22].

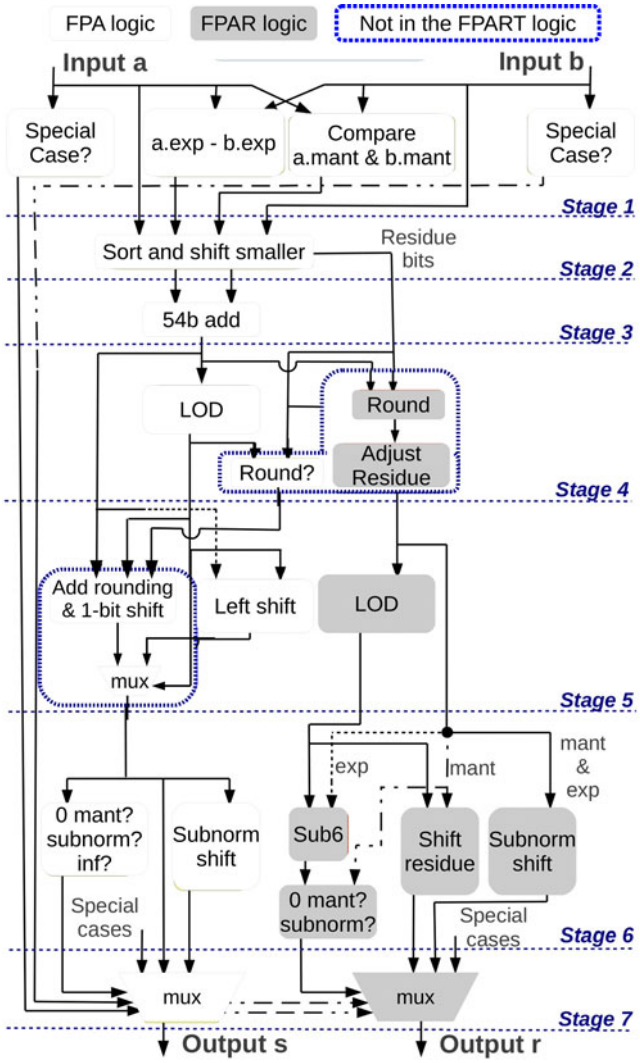

Fig. 15. Structure of the FPAR unit.

7.1 Floating-Point Adder

A 64-bit IEEE-754 double-precision number contains 1 sign bit, 11 exponent bits ($e = 11$), and 52 mantissa bits plus an implicit leading 1 ($p = 53$), and handles subnormal numbers, zero, infinity and Not-A-Number (NaN). The adder computes a rounded IEEE-754 sum, s, according to Eq. (1).

Our FPA is divided into seven pipeline stages. The white part of Fig. 15 shows the general organization, which is similar to the improved single-path floating-point adder in [23] (Fig. 8.8, p. 428), except that we use a Leading One Detector (LOD) instead of a Leading One Predictor (LOP) in order to save area. The datapath operates as follows: First, the mantissas are compared and the exponents subtracted (stage 1), before being used to shift right the smaller one's mantissa (stage 2). During the right shift, the bits that would have normally been discarded are saved and used to compute rounding requirements in stage 4. Before that, the mantissas are added together (stage 3) and the LOD determines the resulting change in exponent (stage 4), which is then used to determine rounding requirements (also stage 4), as well as in stage 5 to normalize the mantissa. Stage 5 rounds and shifts the number, which can be parallelized to improve delay since the costly dynamic left shift is only needed when a and b are completely overlapping, and thus no rounding is needed. Otherwise, the position of the mantissa's MSB can only change by at most one in either direction, so that in the case where rounding is added, normalization is performed with a much cheaper 1-bit right or left shift as suggested in [23]. A multiplexer is then used to select the proper case. Stage 6 checks if the result is zero (zero mantissa), subnormal (negative exponent), and if it has overflowed to infinity (maximum exponent), at the same time as computing a subnormal version assuming the exponent is negative. Finally, stage 7 selects among the rounded normalized number, its subnormal version, a 0 output, and a special case number determined in stage 1.

We described this hardware design in Bluespec SystemVerilog [24] and implemented it on a Virtex 6 FPGA (xc6vlx240t, speed grade of -1). It occupies an area of 1517 Lookup Tables (LUTs) and runs at 250 MHz (the critical path delay is 3.996 ns after place and route). The slowest and most area consuming operations are the dynamic shifts, the additions, and the 54-bit comparison in stage 1. Computing an update to $nzcnt$ and $maxexp$ from a single residue (Section 5.1) has a critical path of 2.378 ns and only costs 53 LUTs, making the rsb calculation small compared to the floating-point additions.

7.2 Floating-Point Adder with Residue

The FPAR can be built by extending the FPA. Fig. 15 shows its organization, with the additional hardware shown in gray. Stages 1, 2 and 3 are mostly the same, except for the additional registers to communicate the residue bits to the subsequent stages instead of discarding them. Unfortunately, we cannot perform a leading-1 search on r at the same time as s (stage 4) since we first need to know where the s mantissa will end in order to know where to start the search for r. Therefore, this step is moved to stage 5, and is performed after "residue adjustment", which is moved to stage 4. Residue adjustment is the process of subtracting from the residue the number that is added to the sum due to rounding; it is controlled by a rounding unit similar to the one for the sum: adding 1 to the LSB of s is coupled with subtracting 1 one position above the MSB of r and vice versa. However, we only need to know whether the MSB of the sum has moved by one bit at most in order to determine the position at which the residue should be adjusted, since moving by more than one position would mean that a and b are canceling each other and that the residue is zero. This three-case check can be performed at low cost during stage 4 even without knowing the sum's leading-1 information and is followed by an adder to compute the rounded residue. Stage 5 then performs a leading-1 search on the residue. Stage 6 normalizes and computes a subnormal version, both of which require dynamic shifts, together with checking whether the residue is 0 (0 mantissa) or subnormal (this time checking for a negative normalized exponent $exp_{norm} = exp - LO_{idx}$, where LO_{idx} is the index of the leading 1). Finally, in stage 7, a second multiplexer chooses among the different possible residue outputs.

We are thus able to exploit parallelism to produce the residue without affecting the clock frequency and number of pipeline stages. In particular, since we are able to

determine the index at which the residue must be adjusted one cycle earlier than for the sum, we can change the order of operations while maintaining a similar delay: first round, then search for the leading-1. During stages 6 and 7, the final checks can also be performed in parallel, and because the FPAR already performs costly dynamic shifts for the subnormal cases, performing another dynamic shift for the common residue case in parallel does not impact the delay (except for extra routing). This parallelization also works because even though exp_{norm} must be computed, that result is not required by the dynamic shifters in stage 6, but only by the comparator of stage 7 to check whether the result is subnormal. Indeed, if the number is subnormal then only the exponent information is needed to shift the mantissa properly (not the leading-1's position). When it is not subnormal then only the leading-1 information is needed to shift the mantissa (not the exponent).

The FPAR requires 2252 LUTs, only 48 percent more than the FPA. It also uses seven pipeline stages running at 250 MHz. The critical path delay is 3.999 ns. Manoukian and Constantinides implemented a similar single-precision FPAR on a Virtex 6, also with no delay overhead, and 47 percent area overhead [13]. Nathan et al. implemented a custom chip for a similar unit with 54 percent area overhead [22]. These comparable overheads from independent implementations confirm that they are a good estimate of the required resources.

7.3 FPART

We observe that the FPAR unit can be simplified if it only truncates the sum. We thus introduce the FPART (FPAR with Truncation), which "PARTitions" the exact sum into two parts: the sum and the residue. This does not give an IEEE-754 compliant sum for every operation but still yields an information-preserving sum and residue (Eq. (3)) that is adequate for the algorithm described in the previous sections, where we only round once at the end.

The FPART is similar to the FPAR, except that it does not contain any of the rounding modules, shown in Fig. 15 with a dotted contour. Therefore, the sum portion of the computation can determine all of its bits simply by summing the aligned a and b mantissas and detecting the leading 1; there is no need to know the shape of the discarded bits. However, the residue portion of the computation still needs the LOD information from the sum portion before it can perform an LOD on its own mantissa. The LOD information is needed because a shift in the sum's exponent will change the index of the first bit considered part of the residue, although it will not change the shape of the residue bits; that is, no bit is added or subtracted.

The FPART occupies an area of 1,851 LUTs, only 22 percent more than the FPA. It is also comprised of seven pipeline stages running at 250 MHz: The critical path delay is 3.994 ns.

While we quote specific results from our FPGA implementation, we expect a custom implementation would achieve similar results—The FPAR should achieve the same latency as the base FPA and require only fractionally more area, either an extra half if it rounds at every step or only an extra quarter otherwise. The custom implementation of FPAR in [22] supports this expectation.

8 Experimental Results

8.1 Experimental Setup and Results

In order to evaluate the expected number of iterations of our *CorrectRound* algorithm (Fig. 14), we implemented and simulated it. We use different datasets, which can be more or less suited to the algorithm depending on the amount of cancellation that occurs. For example, if all the numbers are small except for two large opposite numbers whose sum is zero, this will force an additional iteration over the case where those numbers are not there. To measure how much the sum can change with respect to a small change in the summands, we define the condition number of the N x_i datapoints similarly to [25], [26]:

$$\kappa = \frac{\sum_{i=1}^{N} |x_i|}{|\sum_{i=1}^{N} x_i|}.$$

For low κ, the data is said to be well-conditioned, and the algorithm should easily converge, whereas for high κ, it is said to be ill-conditioned and is expected to make convergence more difficult.

We generate datasets similar to those used by Zhu and Hayes [26]. Data #1 consists of random positive numbers, so that $\kappa = 1$. Data #2 is similar except that it contains both positive and negative numbers, resulting in a low κ. Data #3 is similar to Anderson's ill-conditioned data [5]: we first generate Data #2, then compute the mean using standard floating-point arithmetic (thus introducing some error), before subtracting it from each data point. This results in a higher κ. Half of Data #4 is randomly generated positive numbers, and the other half is their exact opposites, such that the exact sum is 0, and $\kappa = \infty$.

In addition to the uniformly distributed random data used in previous work (e.g., [5], [26]), we also use an exponential one. We define an exponential distribution as one where the individual bits of the IEEE-754 representation are randomly picked from a uniform distribution, thus tending to produce numbers with all allowed exponent values instead of concentrating them on the highest positive and negative exponents. This prevents the numbers from being too close together and reduces mantissa overlap, resulting in data that does not reduce as efficiently as a uniform distribution.

We define the parameter δ as the maximum possible difference in exponent ranges in the original summands, which is taken into account when generating the random data. We use $N = 2^{12}$ as the number of summands in the dataset. The results are similar for other values of N, except for very low ones such as $N = 4$, where convergence sometimes happens after only 1 iteration, or after 3 iterations due to an undet case. Tables 1a and 1b show results after repeating each experiment 1,000 times, for the round-to-nearest-tie-to-even mode and the four basic datasets. The other four rounding modes have similar results and they are not shown here. We report both average and standard deviation for the different metrics: the number of iterations the algorithm takes before converging (with and without Early Detection (ED) of termination), the percentage of non-zero values remaining in the tree when convergence is achieved, and the percentage error when computing the sum using a simple sequential software, non-exact summation.

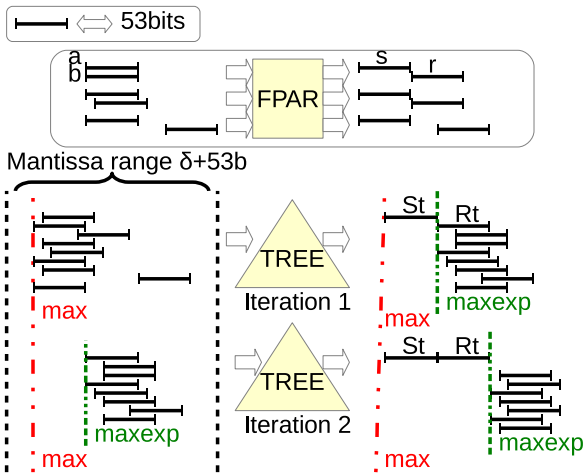

Fig. 16. Effects of the algorithm on the mantissa ranges.

8.2 Discussion of the Results

Table 1a shows that even when there are many non-zeros left in the tree (Data #1 and #2), our algorithm is able to determine that they will not affect the final sum, so that it can discard them and converge faster, significantly reducing the number of iterations before convergence. Table 1 also reminds us that the non-accurate summation error can become intolerably large (Data #3 and #4).

We observe that in virtually all cases, the algorithm takes exactly two iterations to converge (with a 0 standard deviation in number of iterations), no matter how ill-conditioned the data is, except for the extreme case of $\kappa = \infty$ and exponentially distributed data. This empirically validates our inexpensive convergence test since we get the same number of iterations suggested in [18], whose termination detection was much more expensive (two full floating-point additions per tree node). We also note that the highly unlikely undet case was never encountered in these experiments.

Fig. 16 illustrates why exactly two iterations are needed most of the time. After two numbers a and b are processed by an FPART unit, they come out as two new numbers with non-overlapping mantissas, where the MSB has typically only shifted by a few bits at most: a major shift of m positions is extremely unlikely for randomly chosen bits, in the order of 2^{-m}. Therefore, with extremely high probability, the final sum S_t at the end of the tree will occupy the 53 bits to the immediate left of the final residue R_t, whereas R_t will occupy the same bits as several other residues remaining in the tree, including the largest one that determines $maxexp$. Since $rsb.exp.ub = \lceil \log_2(nzcnt) \rceil + maxexp + 1$ (Eq. (13)), rsb easily overlaps with S_t and the convergence test fails. However, after a second pass in the tree, the new R_t still occupies the bits to the immediate right of S_t, but this time the remaining residues and $maxexp$ occupy the bits to the immediate right of R_t, making it extremely unlikely that $\log_2(nzcnt)$ would be large enough to cause an overlap with S_t. A dataset needs to be carefully designed to get more than two iterations, as we did with the exponentially distributed Data #4. In that case, major cancellations throughout the tree translate into a $maxexp$ that is often larger than S_t, thus failing the convergence test. In fact, we observe empirically that in this case, the number of iterations is

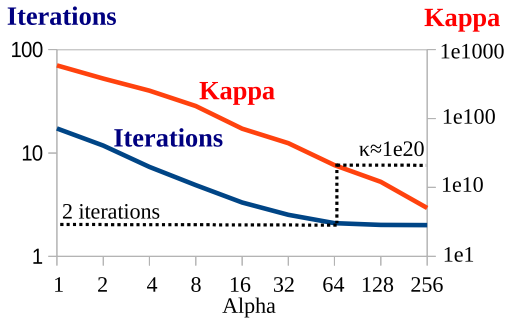

Fig. 17. Average number of iterations versus κ.

about $i \approx \frac{\delta}{53} + 1$, 53 being the mantissa range of an IEEE-754 number. The exponent range δ translates into $\delta + 53$ bits that can be covered by the dataset. We can then divide this range into i equal parts of 53 bits each, where each part is being resolved as zero during one iteration, thus taking i iterations before convergence overall.

Therefore, in order to see more than two iterations, we had to define extremely ill-conditioned cases that are unlikely to occur in practice (condition number $\kappa \to \infty$). In order to determine how large κ must be to exceed two iterations, we introduce Data #5, which is generated in the same way as Data #4, except that we do not replicate α numbers ($\alpha \in [0, N/2]$). $\alpha = 0$ is equivalent to Data #4; $\alpha = N/2$ is equivalent to Data #2. The α parameter allows us to explore even larger κ values than Data #3, before reaching $\kappa = \infty$ in Data #4. For $\delta = 1{,}500$, we find that $\kappa > 10^{20}$ is required before we need more than two iterations (see, Fig. 17).

9 Comparisons

Assuming a constant number of iterations, our algorithm achieves the asymptotically optimal FLOP count $\Theta(N)$—the same as simple summations that ignore precision and other efficient correctly rounded approaches (e.g., [6], [20]). With sufficient hardware, our design can achieve $O(\log N)$ latency, superior to traditional correctly rounded summations including iFastSum [6] and Demmel and Hida's Radix sort [20], both of which are $\Theta(N)$. Only Leuprecht and Oberaigner's correct summation algorithm achieved $O(\log N)$ latency [18]. Our $O(\log N)$ latency is the same as a precision-ignoring sum, and Kapre and DeHon's sequential-semantics-preserving sum [15].

Table 2 compares the FLOP counts of the major, representative algorithms, including the revision of prior work

TABLE 2
FLOP Count Comparison

Accumulation Algorithm	FLOP /input /iter	Expected # of iter	Expected FLOP /input	Depth
iFastSum [6]	6	2	12	$O(N)$
with our FPAR	1.5	2	3	
Simple, inaccurate	1	1	1	$O(\log N)$
Optimistic Sequential [15]	5	8	40	$O(\log N)$
Leuprecht and Oberaigner [18]	7	2	14	$O(\log N)$
with our FPAR	2.5	2	5	
This Work (FPAR)	1.5	2	3	$O(\log N)$
FPART	1.25	2	2.5	

using our FPAR. We count 1.5 FLOP for our FPAR and 1.25 FLOP for our FPART due to their area (Section 7). All three correct rounding algorithms (iFastSum, Leuprecht and Oberaigner, and ours) take two iterations in almost all cases. We estimate Kapre and DeHon as eight iterations with 5 FLOP per iteration. Leuprecht and Oberaigner use two FLOP per tree node for convergence detection. We do not count our termination computation as a FLOP since it takes less than 4 percent of the area of the FPA (see end of Section 7.1). Our algorithm achieves a lower FLOP count than previous algorithms, tying only with iFastSum, which has depth $O(N)$ instead of our $O(\log N)$ depth.

To avoid $O(N)$ area on the reduce tree, the strategy can be adapted to support pipelined, streaming accumulations consuming any number of inputs per cycle [4].

10 Conclusion

Floating-point values can be summed in parallel to produce a correctly rounded result in $O(\log N)$ depth. We have introduced a lightweight test for early termination detection and provided evidence that our algorithm is fast and predictable—only requiring two iterations in virtually all cases—and a proof that it will terminate. We have shown that our algorithm can support all five IEEE-754 standard rounding modes using an FPART unit. We implemented the FPART as an extension of a standard FPA, performing all the additional computations in parallel with already necessary operations; as a result, the FPART runs as fast as the standard FPA, while requiring only 22 percent more area.

Acknowledgments

Support for Paul Gurniak from the Rachleff Scholar's Program was instrumental in initiating this work. This material is based in part upon work supported by the office of naval research (ONR) under Contract No. N000141010158. The views expressed are those of the authors and do not reflect the official policy or position of ONR or the U.S. Government. Valuable feedback from the anonymous reviewers on our preliminary work [4] and the journal submission improved the generality, clarity, precision, and presentation of the paper.

References

[1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger, "Clock rate versus IPC: The end of the road for conventional microarchitectures," in *Proc. 27th Annu. Int. Symp. Comput. Archit.*, 2000, pp. 248–259.

[2] M. Horowitz, E. Alon, D. Patil, S. Naffziger, R. Kumar, and K. Bernstein, "Scaling, power, and the future of CMOS," in *Proc. IEEE Int. Electron Devices Meet., Techn. Digest.*, Dec. 2005, pp. 7–15.

[3] S. H. Fuller and L. I. Millett, Eds. (2011). *The Future of Computing Performance: Game Over or Next Level?* Washington, DC, USA: National Academies Press, [Online]. Available: http://www.nap.edu/catalog.php?record_id=12980.

[4] E. Kadric, P. Gurniak, and A. DeHon, "Accurate parallel floating-point accumulation," in *Proc. IEEE Symp. Comput. Arithmetic*, Apr. 2013, pp. 153–162.

[5] I. J. Anderson, "A distillation algorithm for floating-point summation," *SIAM J. Sci. Comput*, vol. 20, pp. 1797–1806, 1999.

[6] Y.-K. Zhu and W. B. Hayes, "Correct rounding and a hybrid approach to exact floating-point summation," *SIAM J. Sci. Comput.*, vol. 31, no. 4, pp. 2981–3001, Jul. 2009.

[7] S. M. Rump, "Ultimately fast accurate summation," *SIAM J. Sci. Comput.*, vol. 31, no. 5, pp. 3466–3502, Sep. 2009.

[8] P. Kornerup, V. Lefèvre, N. Louvet, and J.-M. Muller, "On the computation of correctly rounded sums," *IEEE Trans. Comput.*, vol. 61, no. 3, pp. 289–298, Mar. 2012.

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

[10] D. E. Knuth, *The Art of Computer Programming: Seminumerical Algorithms*, 3rd ed. Boston, MA, USA: Addison-Wesley, 1997, vol. 2.

[11] F. de Dinechin, J. Detrey, O. Cret, and R. Tudoran, "When FPGAs are better at floating-point than microprocessors," ENS Lyon, Lyon, France, Tech. Rep. ensl-00174627, 2007. [Online]. Available: http://prunel.ccsd.cnrs.fr/ensl-00174627.

[12] W. Dieter, A. Kaveti, and H. Dietz, "Low-cost microarchitectural support for improved floating-point accuracy," *IEEE Comput. Archit. Lett.*, vol. 6, no. 1, pp. 13–16, Jan.–Jun. 2007.

[13] M. V. Manoukian and G. A. Constantinides, "Accurate floating point arithmetic through hardware error-free transformations," in *Proc. Int. Conf. Reconf. Comput.*, 2011, pp. 94–101.

[14] Z. Luo and M. Martonosi, "Accelerating pipelined integer and floating-point accumulations in configurable hardware with delayed addition techniques," *IEEE Trans. Comput.*, vol. 49, no. 3, pp. 208–218, Mar. 2000.

[15] N. Kapre and A. DeHon, "Optimistic parallelization of floating-point accumulation," in *Proc. IEEE Symp. Comput. Arithmetic*, Jun. 2007, pp. 205–213.

[16] J. Demmel and H. D. Nguyen, "Fast reproducible floating-point summation," in *Proc. IEEE Symp. Comput. Arithmetic*, Apr. 2013, pp. 163–172.

[17] J. Demmel and H. D. Nguyen, "Numerical reproducibility and accuracy at exascale," in *Proc. IEEE Symp. Comput. Arithmetic*, Apr. 2013, pp. 235–237.

[18] H. Leuprecht and W. Oberaigner, "Parallel algorithms for the rounding exact summation of floating point numbers," *Computing*, vol. 28, pp. 89–104, 1982.

[19] D. Priest, "Algorithms for arbitrary precision floating point arithmetic," in *Proc. IEEE Symp. Comp. Arithmetic*, Jun. 1991, pp. 132–143.

[20] J. Demmel and Y. Hida, "Accurate and efficient floating point summation," *SIAM J. Sci. Comput.*, vol. 25, no. 4, pp. 1214–1248, Apr. 2003.

[21] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Burlington, ON, Canada: Morgan Kaufmann, 1992.

[22] R. Nathan, B. Anthonio, S.-L. Lu, H. Naeimi, D. J. Sorin, and X. Sun, "Recycled error bits: Energy-efficient architectural support for floating point accuracy," in *Proc. Int. Conf. High Perform. Comput., Netw. Storage Anal.*, 2014, pp. 117–127. [Online]. Available: http://dx.doi.org/10.1109/SC.2014.15.

[23] M. Ercegovac and T. Lang, *Digital Arithmetic* (ser. The Morgan Kaufmann Series in Computer Architecture and Design). Burlington, ON, Canada: Morgan Kaufmann, 2003.

[24] (2012). Bluespec, Inc., "Bluespec SystemVerilog 2012.01.A." [Online]. Available: http://www.bluespec.com.

[25] N. J. Higham, "The accuracy of floating point summation," *SIAM J. Sci. Comput*, vol. 14, pp. 783–799, 1993.

[26] Y.-K. Zhu and W. B. Hayes, "Algorithm 908: Online exact summation of floating-point streams," *ACM Trans. Math. Softw.*, vol. 37, no. 3, pp. 37:1–37:13, Sep. 2010. [Online]. Available: http://doi.acm.org/10.1145/1824801.1824815.

Edin Kadric received the BEng and MEng degrees in electrical engineering from McGill University, Canada, in 2010 and 2011, respectively. Since 2012, he has been working towards the PhD degree at the University of Pennsylvania, Philadelphia, PA, USA. His research interests include high-performance, low-power, and high-reliability computer architectures and applications. He is a student member of IEEE.

Paul Gurniak received the BS degree in electrical engineering from the University of Pennsylvania, Philadelphia, PA, USA. He is a staff software engineer at MaxPoint Interactive. He was a member of the Rachleff Scholars Program and has received the Atwater Kent Prize in electrical engineering for his academic accomplishments. Contact him at pgurniak@seas.upenn.edu.

André DeHon received the SB, SM, and PhD degrees in electrical engineering and computer science from the Massachusetts Institute of Technology, Cambridge, MA, USA, in 1990, 1993, and 1996, respectively. From 1996 to 1999, he co-ran the BRASS group in the Computer Science Department, University of California at Berkeley. From 1999 to 2006, he was an assistant professor of computer science at the California Institute of Technology, Pasadena, CA, USA. Since 2006, he has been in the Electrical and Systems Engineering Department, University of Pennsylvania where he is currently a full professor. He is broadly interested in how we physically implement computations from substrates, including VLSI and molecular electronics, up through architecture, CAD, and programming models. He places special emphasis on spatial programmable architectures (e.g., FPGAs) and interconnect design and optimization. He is a member of IEEE.