Appearing in *IEEE Symposium on Computer Arithmetic* (ARITH18, 2007), June 25–27, 2007

# Optimistic Parallelization of Floating-Point Accumulation

Nachiket Kapre
Computer Science
California Institute of Technology
Pasadena, CA 91125
nachiket@caltech.edu

André DeHon
Electrical and Systems Engineering
University of Pennsylvania
Philadelphia, PA 19104
andre@acm.org

***Abstract*— Floating-point arithmetic is notoriously non-associative due to the limited precision representation which demands intermediate values be rounded to fit in the available precision. The resulting cyclic dependency in floating-point accumulation inhibits parallelization of the computation, including efficient use of pipelining. In practice, however, we observe that floating-point operations are "mostly" associative. This observation can be exploited to parallelize floating-point accumulation using a form of optimistic concurrency. In this scheme, we first compute an optimistic associative approximation to the sum and then relax the computation by iteratively propagating errors until the correct sum is obtained. We map this computation to a network of 16 statically-scheduled, pipelined, double-precision floating-point adders on the Virtex-4 LX160 (-12) device where each floating-point adder runs at 296 MHz and has a pipeline depth of 10. On this 16 PE design, we demonstrate an average speedup of 6× with randomly generated data and 3–7× with summations extracted from Conjugate Gradient benchmarks.**

## I. Introduction

Scientific computing applications rely upon floating-point arithmetic for numerical calculations. For portability, almost all applications use the industry-standard floating-point representation IEEE-754 [1] which provides uniform semantics for operations across a wide range of machine implementations. Each IEEE floating-point number has a finite-precision mantissa and a finite-range exponent specified by the standard, and the standard defines correct behavior for all operations and necessary rounding. A fully-compliant hardware implementation requires hundreds of logic levels and hence these units are heavily pipelined (*e.g.* floating-point units in *Intel Itanium 2* and *Intel Pentium 4* have a latency of 4 clock cycles [2], FPGA floating-point cores from Sandia [3] have a latency of 10–14 cycles).

The finite precision and range of floating-point numbers in the IEEE format requires rounding in the intermediate stages of long arithmetic calculations. This limited precision representation makes floating-point arithmetic non-associative. While there are well-known techniques in the literature for maintaining unbounded precision in the middle of long sequences of floating-point calculations [4], [5], this comes at the cost of performing many more primitive floating-point operations. As a result, the limited-precision and non-associativity of IEEE floating point is generally accepted as a reasonable tradeoff to permit fast hardware implementation of floating-point arithmetic. Consequently, portable floating-point computations must always be performed strictly in the order specified by the sequential evaluation semantics of the programming language. This makes it impossible to parallelize most floating-point operations without violating the standard IEEE floating-point semantics. This restriction is particularly troublesome when we notice that the pipeline depths of high-performance floating-point arithmetic units is tens of cycles, meaning common operations, such as floating-point accumulation, cannot take advantage of the pipelining, but end up being limited by the latency of the floating-point pipeline rather than its throughput.

In this paper, we show how to parallelize floating-point accumulation while obtaining the **same** answer as the simple, sequential specification. In Section II, we review how simple associativity fails in floating-point summations, highlight where floating-point summations occur in practice, and introduce the technique of parallel-prefix summation. Our algorithm, described in Section III, implements a form of optimistic concurrency that utilizes parallel-prefix sums to speedup the summation. We show theoretical performance benefits and cost analysis for the algorithm in Sections III-C and III-D. We evaluate the algorithm using sample datasets in Section IV, and provide a concrete mapping onto a specific hardware implementation in Section V.

## II. Background

### A. Non-Associativity of Floating-Point Accumulation

The limited precision and range of the IEEE floating-point representation cause floating-point accumulation to be non-associative. As an illustrative example, Figure 1 shows a case where associativity does not hold. If associativity held, we could perform the calculation either sequentially (shown on left) or using a balanced reduce tree (right) and obtain the same result. However, as the example shows, the two different associations yield different results. For portability and proper adherence to the IEEE floating-point standard, if the program specifies the sequential order, the highly parallel, balanced reduce tree implementation would be non-compliant; it would produce incorrect results for some sets of floating-point values.

Previous attempts to pipeline floating-point accumulation, such as [6], do so only at the expense of assuming associativity

© 2007 IEEE

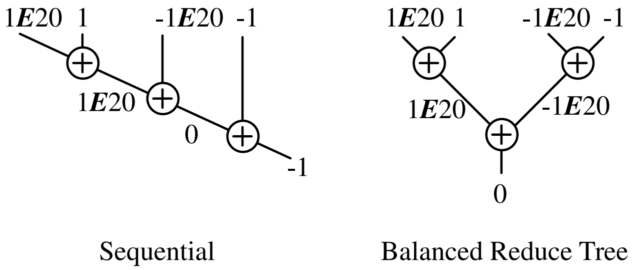


Fig. 1. Floating-Point Addition

and thereby producing non-compliant results. In contrast, the techniques introduced here show how to exploit associativity while obtaining results identical to the sequential ordering specified in the program.

### *B. Example: Conjugate Gradient Sums*

Conjugate Gradient (CG) is a scientific computing application whose parallelism can be severely limited by sequential accumulation. CG is a popular iterative numerical technique for solving a sparse, linear system of equations represented by $A \times x = b$, where $A$ is a square $n \times n$ matrix and $x$ and $b$ are vectors of length $n$. Sparse Matrix-Vector Multiply (SMVM) is the dominant computation kernel in CG. In SMVM, we compute dot products between the rows of $A$ and the vector $x$ which effectively require us to sum the products of the non-zero matrix values with their corresponding vector entries in $x$. We use an implementation of SMVM due to deLorimier *et al.* [7] which performs the dot-product summations in parallel on distinct processing elements as a reference for this study. For sparse matrices, the number of non-zero entries per row can be unbalanced, with average rows requiring sums of only 50–100 products, and exceptional rows requiring much larger sums. If each dot product sum must be sequentialized, the size of the largest row can severely limit the parallelism in the algorithm and prevent good load balancing of the dot products. In addition to these dot-product sums, a typical CG iteration requires a few global summations with length equal to the size of the vectors, $n$. For large numerical calculations, $n$ can easily be $10^4$, $10^5$ or larger; if these summations must be serialized, they can become a major performance bottleneck in the task, limiting the benefits of parallelism.

### *C. Parallel Prefix*

A common technique for parallelizing associative operations is parallel-prefix reduction. It allows us to compute the result of an associative function over $N$ inputs in $O(\log(N))$ time. The computation also generates $N-1$ intermediate values as part of the process. For example, a parallel-prefix *accumulate* on an input sequence $[x_1,x_2,x_3,x_4]$ generates an output sequence $[x_1,(x_1+x_2),(x_1+x_2+x_3),(x_1+x_2+x_3+x_4)]$ which consists of 3 intermediate sums of the input as well as the final sum of all 4 inputs. Several styles of parallel-prefix are found in practice (*e.g.* Brent-Kung [8], Sklansky [9], Kogge-Stone [10], Han-Carlson [11]). The exact number of calculations required depends on the style of prefix used, but is usually more than the simple, serial sum. Parallel-prefix allows us to tradeoff extra computation for lower latency generation of results. A key requirement, however, is that the reducing function be associative. Consequently, this technique does not directly apply to floating-point accumulation.

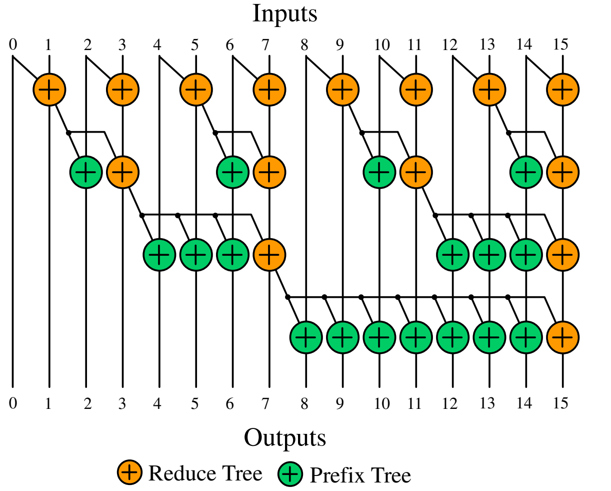


Fig. 2. Sklansky Parallel-Prefix Tree

As an illustration, we show the working of Brent-Kung and Sklansky parallel-prefix adders. For the Sklansky adder shown in Figure 2, we can see that the final result is computed by recursively adding pairs of operands in $\log_2(N)$ steps. This is the “reduce” tree. We compute the remaining $N-1$ values in the idle slots, at each stage, of this “reduce” tree. This we call the “prefix” tree. The Sklansky-style parallel-prefix operation requires $N/2$ additions at each stage of the tree. Since all additions at a given stage in the tree are completely independent, they can be run in parallel. This is what makes this technique attractive for parallelizing associative functions. The Brent-Kung adder shown in Figure 3 has the same “forward reduce” tree as the Sklansky adder. The final $N-1$ values are computed differently. For that, we use a “reverse prefix” tree which requires less than $N$ operations but takes an additional $\log_2(N)-1$ steps. Thus, the total number of operations in Brent-Kung adders is just under $2N$, meaning it requires only twice as many additions as the simple, sequential sum. The Sklansky adder has the advantage of computing the required results with low latency, while the Brent-Kung adder does the same with fewer operations and a deeper tree. Our hybrid scheduling algorithm, described in Section V-B, borrows the best features from these two schemes. A tutorial discussion of parallel-prefix computations can be found in [12], [13].

## III. The Algorithm

### *A. Idea*

As the example in Figure 1 shows, trivial associativity could fail to yield the correct answer for floating-point accumulation. This is unfortunate, since associativity could allow us to exploit parallelism to speedup the summation. However, many additions will not suffer from this lack of associativity, and most will suffer only slightly. Thus, most of the time, our

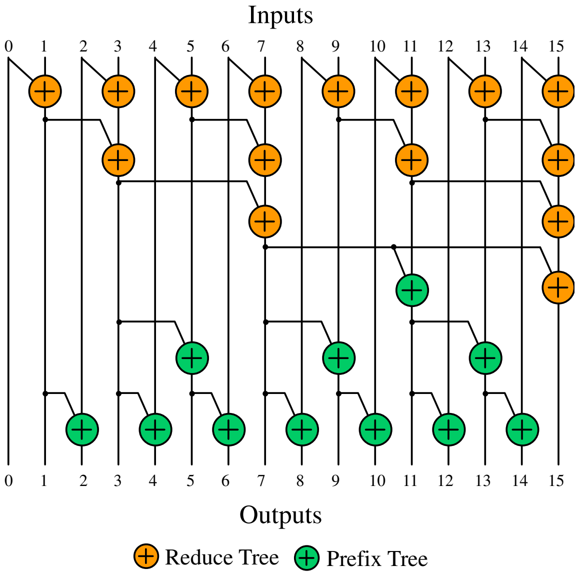


Fig. 3. Brent-Kung Parallel-Prefix Tree

associative answer will be a close approximation to the correct answer. However, if we do not provide IEEE floating-point semantics, the results will differ from machine-to-machine creating additional complexity for the developer and user. We can retain correct semantics and still exploit associativity using the following idea: we use an associative calculation to *approximate* the result and recompute only the portion of the accumulation that was *detected* to be erroneous. Thus, we need to sequentialize only the associativity exceptions in our result rather than the entire accumulation.

### B. Description

A block diagram and accompanying pseudocode for our optimistic-associativity algorithm for floating-point accumulation is shown in Figure 4.

1) We first compute a parallel-prefix sum `Prefix[]` as an approximation to the required sum from the set of inputs `Inputs[]`.
2) To detect completion, we follow this with an induction step, where we check the validity of the sum at each position `Prefix[i]` in the associative sum vector `Prefix[]` using the associative sum at the previous position `Prefix[i-1]` and `Input[i]`.

$$\texttt{Error[i]} = \texttt{Prefix[i]} - (\texttt{Prefix[i-1]} + \texttt{Input[i]})$$

   This is similar to the *Induction* step of mathematical proofs using induction.
3) If every inductive step is error free (`Error[i]` is 0 for all `i`), then we know the final sum must be correct.
4) If there are any errors in the inductive step ($\texttt{Error[i]} \neq 0$), we need to propagate these errors back into the associative sum vector `Prefix[]` to correct the result. We do this by first performing a parallel prefix on the error vector `Error[]` to generate `ErrPrefix[]`.
5) We then add the resulting error correction vector `ErrPrefix[]` into `Prefix[]` to improve our approximation of the sequential sum.
6) Since error propagation is also a floating-point operation and therefore non-associative, the result of this update might still not be correct. Hence, we repeat the algorithm iteratively from Step 2 until we detect convergence; that is, the error vector is composed entirely of 0 entries.

This solution is "optimistic" in that we optimistically assume that the operation can be performed associatively. However, we check the validity of that assumption and recalculate when the assumption is wrong. The solution is "relaxing" in that we are continually updating our approximation with better information until it converges.

This is an example of the "Common Case" principle in system engineering. The common case is that the operations are associative to the required precision. Rather than penalizing all the cases, we handle the associative case efficiently. When it fails, we detect the failure and iterate until we converge.

In the worst case, this algorithm reduces to full sequentialization. That is, there is an error at the position following the one that was previously corrected. Thus, we are perfectly resolving one sequential addition step in each iteration. From this simple observation, we can conclude that convergence is always guaranteed. If it were to only resolve one position on each iteration, the algorithm would be much slower and less efficient than a conventional, sequential addition. However, as we will see, this worst case never happens in practice.

### C. Analysis

In this section, we summarize the computational requirement and latency of our algorithm assuming an unbounded amount of hardware. Each "iteration" here involves:

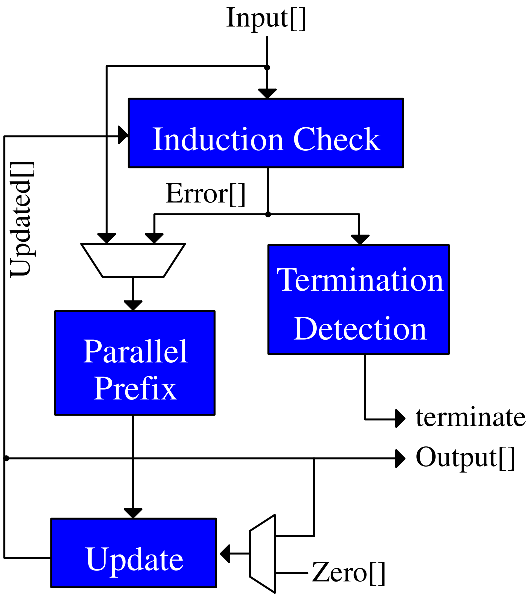
- Assuming a Sklansky-style parallel-prefix computation, the parallel-prefix sum (Step 1 and 4 in Figure 4) requires $\frac{N}{2} \times \log_2(N)$ floating-point additions with a critical path of $\log_2(N)$ sequential floating-point additions. Operations at a given level in the prefix tree (See Figure 2) can be executed in parallel.

$$\begin{aligned} N_{prefix} &= \frac{N}{2} \times \log_2(N) && (1) \\ T_{prefix} &= \log_2(N) \times T_{fpadd} && (2) \end{aligned}$$

- The Induction computation (Step 2 in Figure 4) requires $N$ additions and $N$ subtractions. The latency of this step is only two floating-point operations (one add followed by one subtract) since all operations can performed in parallel.

$$\begin{aligned} N_{induction} &= 2 \times N && (3) \\ T_{induction} &= 2 \times T_{fpadd} && (4) \end{aligned}$$

- Termination Detection (Step 3 in Figure 4) involves a simple comparison to the constant zero on each entry in the error vector followed by a trivial AND reduce. These can also be performed in parallel and, as we will see



```
// Step 1
Prefix[] = ParallelPrefix(Input[]);
Output[] = coreLoop(Input[], Prefix[]);

coreLoop(Input[], Prefix[]) {
terminate = true;
// Step 2
parallel for (int i=0; i<N; i++)
    specSum = Input[i] + Prefix[i-1];
    Error[i] = Prefix[i] - specSum;
    if (Error[i]!=0)
        terminate = false;
    end if
end parallel for
// Step 3
if (!terminate)
    // Step 4
    ErrPrefix[] = ParallelPrefix(Error[]);
    // Step 5
    parallel for (int i=0; i<N; i++)
          Updated[i] = ErrPrefix[i]
                      + Prefix[i];
    end parallel for
    // Step 6
    return coreLoop(Input[], Updated[]);
else
    // finish Step 3
    return Prefix[];
end if
}
```

Fig. 4. The Optimistic Associative Floating-Point Accumulation Algorithm

in Section V-B, they can be overlapped with the next iteration so that they do not contribute to the total latency of each "iteration".

$$N_{termination} = N \quad (5)$$

$$T_{termination} = (T_{fpadd} + \log_2(N) \times T_{AND}) \quad (6)$$

- Finally, the Update operations (Step 5 in Figure 4) requires $N$ additions to add the error vector into the prefix sum. These additions can also be parallelized.

$$N_{update} = N \quad (7)$$

$$T_{update} = T_{fpadd} \quad (8)$$

If we perform $k$ iterations, then the total number of floating-point operations is:

$$\begin{aligned} N_{total} &= k \times (N_{prefix} + N_{induction} + N_{update} \quad (9) \\ &\qquad + N_{termination}) \\ &\approx N \times \left(\frac{1}{2}\log_2(N)k + 4k\right) \quad (10) \end{aligned}$$

Thus, we perform $\left(\frac{1}{2}\log_2(N)k + 4k\right)$ times as much work as the simple, sequential summation. However, these operations can be parallelized and, if scheduled properly, will outperform the sequential summation. The termination detection step can be overlapped with the next iteration and hence is counted only once. Assuming complete parallelization of other steps, the total number of cycles required will be:

$$\begin{aligned} T_{total} &= k \times (T_{prefix} + T_{induction} + T_{update}) \quad (11) \\ &\quad + T_{termination} \\ &\approx k \times (\log_2(N) + 3) \times T_{fpadd} \quad (12) \end{aligned}$$

For reference, the sequential sum will take:

$$T_{seqsum} = N \times T_{fpadd} \quad (13)$$

As the size of the sum ($N$) gets larger, the forced sequentialization in the prefix sum (the $\log$ term in Equations 1 and 12) becomes less significant and greater speedups ($T_{seqsum}/T_{total}$) are possible. We can represent this ideal estimate of speedup as:

$$\begin{aligned} Speedup &= \frac{T_{seqsum}}{T_{total}} \\ &= \frac{N \times T_{fpadd}}{k \times (\log_2(N) + 3) \times T_{fpadd}} \\ &\approx \frac{N}{k \times \log_2(N)} \quad (14) \end{aligned}$$

### *D. Finite Hardware Analysis*

For interesting sizes of $N$, it would typically be impractical to provide hardware for complete parallelization of the algorithm. Hence, we consider cases with limited amounts of hardware (*i.e.* a fixed number of Floating-Point Processing Elements, $PE$). The exact number of cycles required for the mapping to finite hardware will depend on the efficiency of the scheduling algorithm used and the inherent communication latencies, $T_{comm}$, between the hardware blocks. Nonetheless,

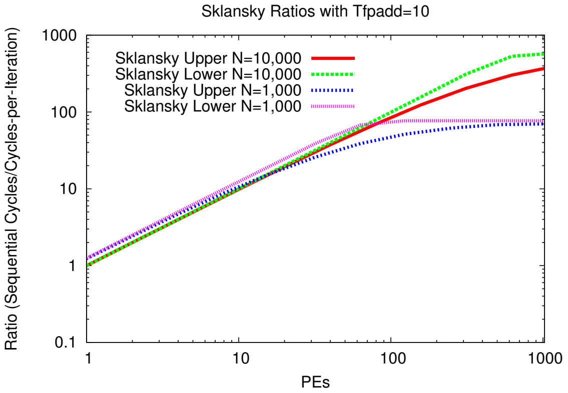


Fig. 5. Ratio of sequential cycles ($T_{seqsum}$) to a Single Iteration of the Optimistic Algorithm

we can write simple analytical equations for the upper and lower bounds for cycles required assuming perfect scheduling, a particular parallel-prefix strategy, and uniform communication latency. For the bounds here we assume a Sklansky-style parallel-prefix summation and schedule a single iteration of the relaxation algorithm.

For the upper-bound on cycles, we assume that the floating-point adder latency and the communication latency between adders cannot be hidden.

$$T_{upper}\left(N, PE\right) = \log_2(N) \times \left(\frac{N/2}{PE} + T_{fpadd} + T_{comm}\right) + 3\left(\frac{N}{PE} + T_{fpadd}\right) \qquad (15)$$

For the lower-bound on cycles, we assume they can be overlapped and charge only the maximum of the latency and throughput bound at each stage.

$$T_{lower}\left(N, PE\right) = \log_2(N) \times \left(\max\left(\frac{N/2}{PE}, T_{fpadd} + T_{comm}\right)\right) + 3\left(\max\left(\frac{N}{PE}, T_{fpadd}\right)\right) \qquad (16)$$

Assuming $T_{fpadd} = 10$, Figure 5 shows ratios between the sequential case ($T_{seqsum}$) and the upper and lower bounds for various $N$ as a function of the number of processing elements, $PE$. This ratio can also be interpreted as the breakeven number of iterations $k$. If $k$ is less than this ratio, the parallel case on the associated number of PEs is faster, otherwise the parallel case is slower than the sequential case.

Note that, these bounds apply only for the pure-Sklansky scheme, and, as we will see later in Section V-B, we can reduce $T$ by decomposing the computation. The tighter schedule enables greater speedups and increases the breakeven $k$.

## IV. Iterations

As Figure 5 suggests, a key question to address is: How many iterations does it typically take for the algorithm to converge? In this section, we empirically characterize the iterations required using both randomly generated data and data from a Conjugate Gradient benchmark.

### A. Generating Random Test Data

We used random datasets for initial analysis of the benefits of the proposed algorithm. We generated floating-point numbers by randomly picking the sign, mantissa, and exponents. We varied the exponent within different ranges to understand the impact of the range on the number of iterations required by the algorithm.

### B. Conjugate Gradient Data

The main limitation of random data sets is their inability to correctly model the natural correlation of floating-point numbers in real applications. After encouraging initial results using random data, we generated real data from a CG implementation and applied those for our experiments. We use matrices from the Matrix Market [14] benchmark for these experiments; each matrix ran for 1000 iterations. From the simulation results, we selected only those floating-point sums of the length under test, $N$, or greater; for homogeneous presentation and analysis of results, we only summed the first $N$ numbers in cases where the actual sums were longer than $N$.

### C. Evaluating the Algorithm

To evaluate convergence, we wrote Scheme [15] for the parallel-prefix algorithm. Scheme supports a faithful implementation of the IEEE-754 standard. We implemented Sklansky-style parallel-prefix addition for summing $N$ numbers ($64 \leq N \leq 1024$), and our hybrid prefix style (Section V-B) for the 1024 input accumulation. We used datasets described in Sections IV-A and IV-B to compute the number of iterations required for termination of our optimistic algorithm.

### D. Iteration Distribution

In Figure 6, we plot the distribution of iterations required for a set of 1000 randomly-generated floating-point accumulations of length 1024. We observe that, in most cases, the iterations required are indeed small; that is, roughly three iterations are required for the average case, while 90% of all cases require <5 iterations. With an average of three iterations, Figure 5 suggests the parallel scheme is faster when we have $PE > 4$.

To illustrate why this works, consider the sum:

$$1.23E1 + 8.43E5 + 3.76E4 + 6.5E0 + 2.43E30 + 7.8E25$$

Summing the first 4 numbers, we get: 8.806188E5. Adding those to 2.43E30, we get 2.43E30 because of the limited 53-bit mantissa precision for double-precision floating point. The sum of the first 4 numbers is actually irrelevant. Consequently, we can safely add $2.43E30 + 7.8E25$ to provide the result. More importantly, non-associativity errors *before* the $2.43E30$ are actually irrelevant to the final result and can be ignored.

The above example is an extreme case where *all* the precision of a particular prefix does not contribute to the final result. In general, we may need only part of the precision of a

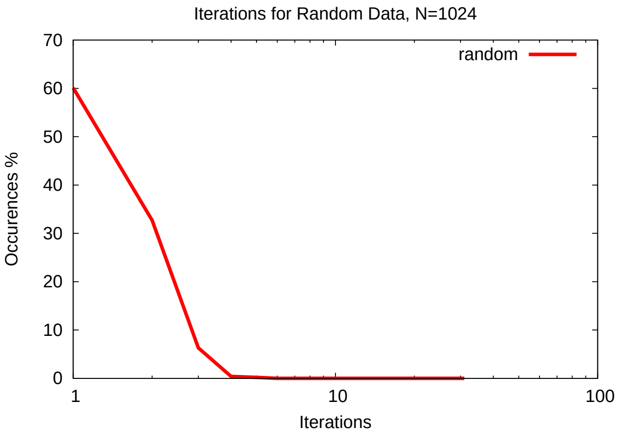


Fig. 6. Number of Iterations for Full-Range Random Dataset

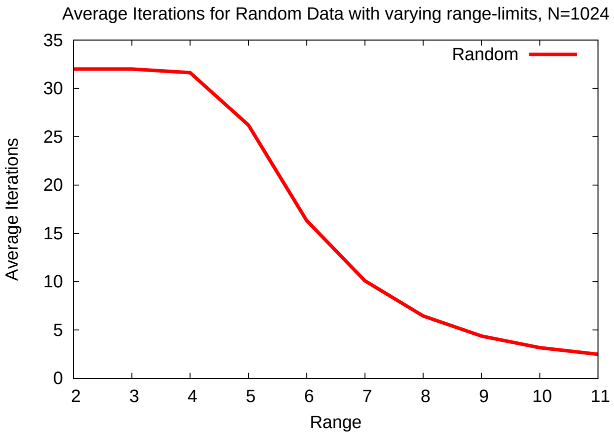


Fig. 7. Number of Iterations for Dataset with different range limits

prefix. As long as the errors in the prefix due to associativity are below the precision required form the prefix, there is no need to compute the result perfectly. A similar observation can be made about the suffix when its magnitude is small compared to some prefix.

Consequently, the algorithm tends to iterate only to the extent that non-associative precision errors actually matter to the final result. The length of the dependency chains to resolve these non-associative errors is typically short compared to the sequential length of the original summation.

### *E. Effect of Limiting Range*

It is reasonable to expect that there may be a relation between number of iterations required for the algorithm to converge and the range of the exponents in the summation sequence. Here, the range of exponents is defined as:

$$range = \log_2\left(\max_i(|x_i|)\right) - \log_2\left(\min_i(|x_i|)\right) \tag{17}$$

In Figure 7, we see that the number of iterations required increases as we limit the range of random numbers appearing in each summation. With limited range floating-point numbers, there are fewer chances for a prefix or suffix to dominate the sum; that is, with all numbers close in range, much of the precision of each floating-point addition in the accumulation has an impact on the final sum. As a result, the impact of the non-associative floating-point additions increases, resulting in an increase in number of iterations in our algorithm. In the worst case, for small ranges, we see the average number of iterations is 32 when summing 1024 numbers.

### *F. Impact of CG Dataset*

Fortunately, in practice, we do not need this worst-case of 32 iterations. In Figure 8 we plot a distribution of iterations required for different CG graphs for variety. We observe that the average number of iterations required in most cases is between 3 and 8. The iterations required are similar to random data with a limited range of 7–9.

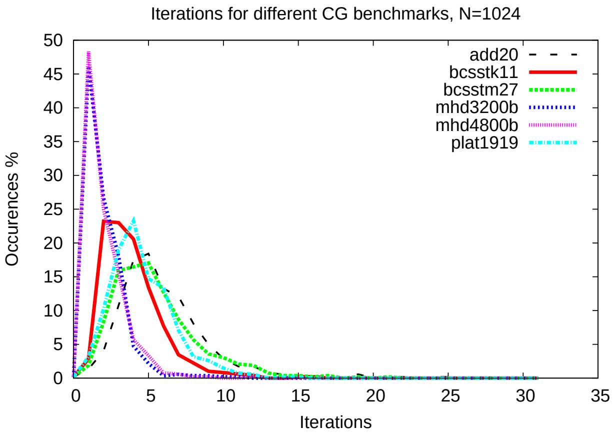


Fig. 8. Number of Iterations for CG Dataset

### *G. Effect of Sequence Length*

As shown in Figure 9, the average number of iterations, $k$ grows slowly with the length of the sum, $N$. The slow growth suggests a greater opportunity for speedups for longer sums.

### *H. Speedup Potential*

Figure 9 suggest an average $k < 7$ for sequences of length 1000. Adding these to upper and lower bound estimates (Equations 15 and 16), we can estimate speedup as a function of PEs as shown in Figure 10.

## V. Implementation

### *A. Hardware Performance Model*

To demonstrate concretely the benefits of our algorithm, we map a length 1024, double-precision floating-point addition to an FPGA implementation with 16 floating-point processing-elements (PEs) interconnected via a statically-routed butterfly fat-tree network (Figure 11) [16]. The raw floating-point performance of FPGAs is beginning to rival or surpass general-purpose processors with hardware floating-point units [17]. However, parallelism and deep pipelining are necessary to fully exploit the floating-point capacity of these FPGAs (*e.g.*

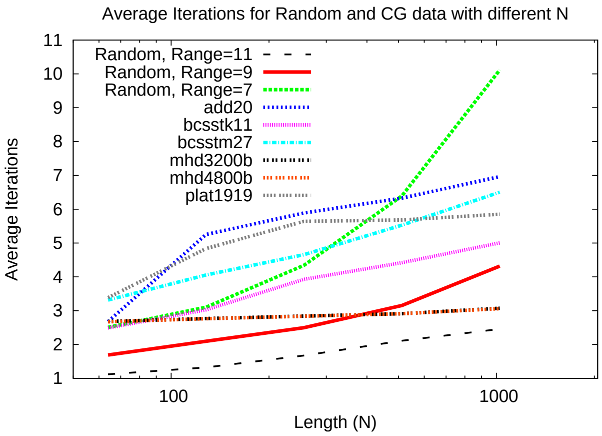


Fig. 9. Iterations vs. Length($N$) for all Data Sets

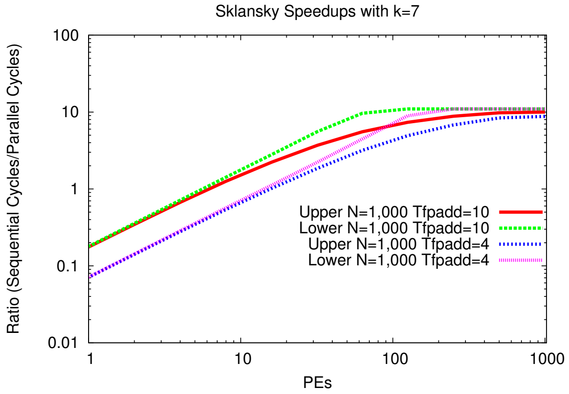


Fig. 10. Speedup versus PEs

$T_{fpadd}$=10–14 in [3]). Consequently, techniques such as ours provide a path to exploit this potential performance.

Each PE in our design has a statically-scheduled, double-precision floating-point adder and multiplier from the Sandia library [3] and an integer adder. The floating-point adder take 571 Virtex-4 slices while a floating-point multiplier takes 801 slices and runs at 296 MHz on a Virtex4-LX160 (speed-grade -12) device. Multipliers and integer adders are not needed for the simple accumulation, but are included because they would typically be required by algorithms which precede the summation. For example, in a dot product, the multipliers would compute the products, then feed the products into the parallel summation. The integer adders also find use for simple comparisons. Each PE also has tightly coupled local memory stores built out of Virtex Block RAMs that hold a portion of the vector to be accumulated, intermediate sums, and the instructions to implement the prefix schedule [7]. We use a 64-bit wide, statically routed communication network allowing each of the 16 PEs to produce or consume a double-precision number on each cycle. The key timing parameters for this implementation are summarized in Table I. Based on these latency figures, we compute achievable speedups.

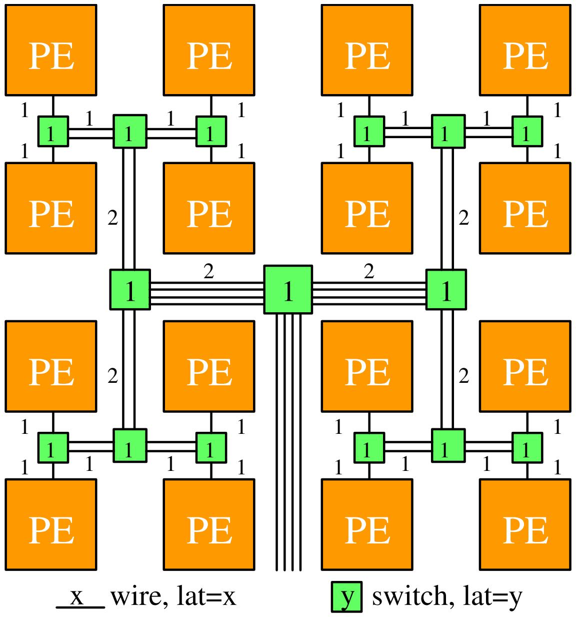


Fig. 11. Graph Machine with 16 PEs

TABLE I
IMPLEMENTATION MODEL PARAMETERS

| Variable | Value | Description |
|---|---|---|
| $T_{fpadd}$ | 10 | Pipeline latency of a Floating-Point addition |
| $T_{comm}$ | 3 | Communication latency in same 2-tree |
| | 7 | Communication latency in same 4-tree |
| | 13 | Communication latency in same 8-tree |
| | 19 | Communication latency in same 16-tree |

## *B. Scheduling*

In Section III-D we showed analysis for the time complexity of the algorithm when mapped to a finite number of floating adders (PEs). We computed theoretical lower and upper bounds for the total cycles required as shown in Equations 15 and 16. In this section, we outline a hybrid prefix strategy and schedule that improves upon even the simple Sklanskly lower bound in Equation 16 when scheduling floating-point sums of length 1024 onto 16 PEs. We design a careful schedule in which we overlap computation with communication wherever possible.

The key idea is to reduce the total number of floating-point operations by decomposing the computation into a series of hierarchical Sklanksy-prefix steps. This avoids computing the full $\frac{N}{2}$ operations at each step in the prefix tree (Figure 2) but produces only a partial set of results. We preserve full prefix semantics by computing the rest of the intermediate prefix sums from the partial results produced by the hierarchical decomposition.

We distribute the set of 1024 floating-point numbers such that each of the 16 PEs gets a sequence of 64 numbers. We then schedule the PEs and the network based on the computational and communication requirements of the different steps of our algorithm. Floating-point operations in the Update and Induction steps (Figure 4) are completely data-parallel and need only nearest-neighbor communication. These are

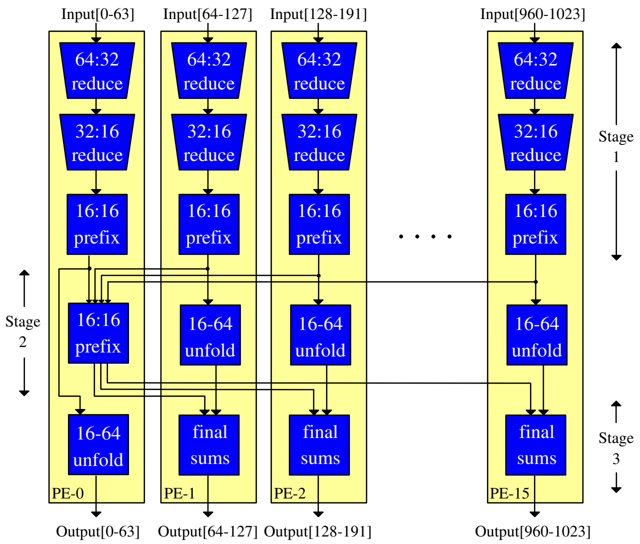


Fig. 12. Sketch of the Hybrid-Schedule

trivially scheduled. The parallel-prefix step, however, has a $\log_2(N) \times T_{fpadd}$ sequential critical path as well as more communication requirements. To schedule this step, we use a hybrid scheme where we first hierarchically decompose the prefix into three stages as shown in Figure 12. We can see in Table II that this reduces the number of floating-point additions from the full Sklansky-style computation by 45%. The first stage is run in parallel on all the PEs and involves computing an independent prefix on the local set of numbers. The second stage is mapped to a single PE (PE-0 in Figure 12) and computes a prefix on the results from the first stage. Concurrently, with this stage, the rest of the PEs are unfolding the intermediate prefix sums for the 64 local inputs. At the end of these two stages, we have a partial global prefix of the inputs. In the third stage, the partial global prefix is then distributed to corresponding PEs to independently generate the remaining prefix sums. During stage 2, since PE-0 was busy computing the partial global prefix, it was unable to compute the unfolding. Moreover, unlike other PEs, PE-0 is not required to calculate the intermediate sums from the global prefix. Hence, in stage 3, we can conveniently schedule the unfolding of the intermediate prefix sums on PE-0. The termination detection step is scheduled to run concurrently on the integer adder since it involves a trivial zero comparison. We can use the idle floating-point adder to speculatively start the execution of the next iteration. When termination is detected, the speculative iteration is flushed from the execution pipeline. The number of cycles required to schedule all these steps is shown in Equations 18.

TABLE II
FLOATING-POINT OPERATIONS PER ITERATION

| Prefix-Scheme | Floating-Point Operations |
|---|---|
| Brent-Kung | 2036 |
| Sklansky | 5120 |
| Hybrid | 2847 |

TABLE III
AVERAGE SPEEDUPS FOR DIFFERENT DATASETS

| Dataset | Average Speedup |
|---|---|
| add20 | 3.3 |
| bcsstk11 | 4.6 |
| bcsstm27 | 3.5 |
| mhd3200b | 7.3 |
| mhd4800b | 7.4 |
| plat1919 | 4.0 |
| random, range=11 | 6.4 |

$$\begin{aligned}
T_{update} &= 64 \\
T_{induction} &= 128 \\
T_{prefix} &= 248 \\
T_{iteration} &= T_{update} + T_{induction} + T_{prefix} \\
&= 248 + 64 + 128 = 440 \\
T_{termination} &= 84 \\
T_{total} &= k \times (T_{iteration}) + T_{termination} \\
&= k \times 440 + 84 \qquad (18)
\end{aligned}$$

### C. Speedups for 16 PE Implementation

On the 16-PE FPGA-based design point, scheduled using the hybrid scheme, this translates into speedups in the range of 2–11 over the non-associative, sequential implementation for the different Conjugate Gradient benchmarks and random datasets as shown in Figure 13. We tabulate the average speedups achieved in Table III. We achieve average speedups of 3–7 at the expense of using 16 times the hardware of the sequential case.

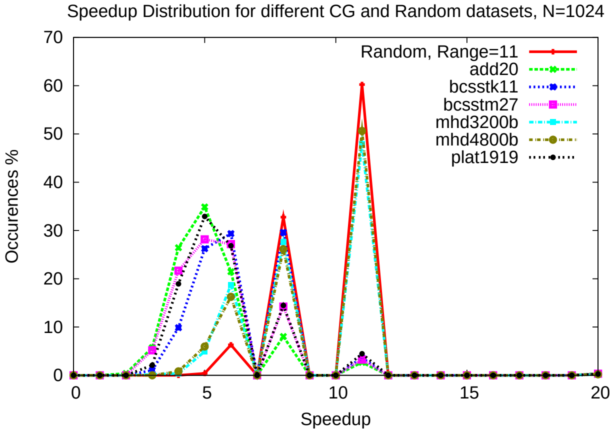


Fig. 13. Speedup for Random and CG Datasets

## VI. Discussion

### A. Generality

The optimistic associative sum technique used here is applicable to any floating-point addition, not just IEEE. What the technique actually provides is a transformation to perform a parallel, associative summation on a number of identical copies of a base floating-point adder core while achieving the **same** semantics as if the sum were performed sequentially on a single floating-point adder core. The rate of convergence may differ for floating-point cores with different behavior and semantics than the IEEE double-precision core assumed here.

### B. Design Exploration

As Section II-C suggests and Section V-B reinforces there are many different formulations for parallel prefix. These tradeoff latency and throughput. The different prefix shapes may also have an impact on the rate of convergence and hence the number of iterations, $k$, required. We have not performed a systematic study of the scheduling options. Rather, the results presented here are an indication that there are schedules which make this technique interesting and useful. We expect there is room to improve upon these results by more carefully exploring the parallel-prefix design space.

## VII. Conclusions

Floating-point accumulation has been widely assumed to be non-parallelizable due to non-associativity of IEEE floating-point arithmetic. In this paper, we demonstrate how to parallelize floating-point accumulation while retaining sequential IEEE accumulation semantics. For this purpose, we devised a ‘relaxation’-algorithm based on optimistic concurrency that iteratively converges to the final result. We observe that the algorithm requires very few iterations in practice for both randomly generated datasets as well as those extracted from a real Conjugate Gradient benchmark. We mapped our algorithm to a 16-PE implementation and optimized a schedule for summing $1024$ numbers to demonstrate speedups. We show that, on average, the $1024$ input accumulation takes 3–7 iterations and manages to achieve an overall speedup of 3–7× compared to the sequential sum.

## Acknowledgments

This work was supported in part by DARPA under grant FA8750-05-C-0011 and the NSF CAREER program under grant CCR-0133102.

Authors benefited from discussions with Jon Ramirez; these discussions and his implementation helped identify many important issues which the work here addresses.

## References

[1] I. S. Committee, *IEEE Standard for Binary Floating-Point Arithmetic*, IEEE, 345 East 47th Street, New York, NY 10017, July 1985, ANSI/IEEE Std 754-1985.

[2] C. McNairy and D. Soltis, “Itanium 2 processor microarchitecture,” *IEEE Micro*, vol. 23, no. 2, pp. 44–55, March/April 2003.

[3] K. S. Hemmert and K. D. Underwood, “Poster: Open source high performance floating-point modules,” in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2006.

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

[5] S. M. Rump, T. Ogita, and S. Oishi, “Accurate floating-point summation,” Faculty of Information and Communication Sciences, Hamburg University of Technology, Institute for Reliable Computing, Hamburg University of Technology, Schwarzenbergstrasse 95, Hamburg 21071, Germany, Tech. Rep. 05.12, November 2005.

[6] Z. Luo and M. Martonosi, “Accelerating pipelined integer and floating-point accumulations in configurable hardware with delayed addition techniques,” *IEEE Transactions on Computers*, vol. 49, no. 3, pp. 208–218, March 2000.

[7] M. deLorimier and A. DeHon, “Floating-Point Sparse Matrix-Vector Multiply for FPGAs,” in *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, February 2005, pp. 75–85.

[8] R. P. Brent and H. T. Kung, “A regular layout for parallel adders,” *IEEE Transactions on Computers*, vol. 31, no. 3, pp. 260–264, March 1982.

[9] J. Sklansky, “Conditional-sum addition logic,” *IRE Transactions of Electronic Computers*, vol. EC-9, pp. 226–231, June 1960.

[10] P. Kogge and H. Stone, “A parallel algorithm for the efficient solution of a general class of recurrence relations,” *IEEE Transactions on Computers*, vol. C-22, no. 8, pp. 786–793, August 1973.

[11] T. Han and D. Carlson, “Fast area-efficient VLSI adders,” in *8th Symposium of Computer Arithmetic*, September 1987, pp. 49–56.

[12] W. D. Hillis and G. L. Steele, “Data parallel algorithms,” *Communications of the ACM*, vol. 29, no. 12, pp. 1170–1183, December 1986.

[13] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann Publishers, Inc., 1992.

[14] NIST, “Matrix market,” <http://math.nist.gov/MatrixMarket/> , June 2004, maintained by: National Institute of Standards and Technology (NIST).

[15] R. Kelsey, W. Clinger, and J. Rees, “Revised$^5$ report on the algorithmic language Scheme,” *Higher-Order and Symbolic Computation*, vol. 11, no. 1, August 1998.

[16] N. Kapre, N. Mehta, M. deLorimier, R. Rubin, H. Barnor, M. J. Wilson, M. Wrighton, and A. DeHon, “Packet-Switched vs. Time-Multiplexed FPGA Overlay Networks,” in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*. IEEE, 2006, pp. 205–213.

[17] K. Underwood, “FPGAs vs. CPUs: Trends in Peak Floating-Point Performance,” in *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, February 2004, pp. 171–180.