Designing Scalable FPGA-Based Reduction Circuits Using Pipelined Floating-Point Cores

Ling Zhuo, Gerald R. Morris, and Viktor K. Prasanna
Department of Electrical Engineering
University of Southern California
Los Angeles, CA 90089-2562
{lzhuo,grm,prasanna}@usc.edu

Abstract

The use of pipelined floating-point arithmetic cores to create high-performance FPGA-based computational kernels has introduced a new class of problems that do not exist *when using single-cycle arithmetic cores. In particular, the data hazards associated with pipelined floating-point reduction circuits can limit the scalability or severely reduce the performance of an otherwise high-performance computational kernel. The inability to efficiently execute the reduction in hardware coupled with memory bandwidth issues may even negate the performance gains derived from hardware acceleration of the kernel. In this paper we introduce a method for developing scalable floating-point reduction circuits that run in optimal time while requiring only* $\Theta(\lg(n))$ *space and a single pipelined floating-point unit. Using a Xilinx Virtex-II Pro as the target device, we implement reference instances of our reduction method and present the FPGA design statistics supporting our scalability claims.*

1. Introduction

Research on reconfigurable architectures covers a broad spectrum of topics such as System-on-Chip [2], run-time reconfiguration techniques [9], interconnection networks [8], and even fundamental floating-point operations like division and square root [5, 6]. The increased gate count, arithmetic capability, and other features of modern FPGAs [1, 13] now allow researchers to investigate general-purpose floating-point computational kernels such as linear algebra routines [12] or iterative solvers [4], and computational kernels from specific problem domains such as molecular dynamics [10].

Clearly, FPGAs are no longer restricted to integer applications and as a substitute for ASICs. Vendors such as SRC [11], and Cray [3] already offer high-end computing clusters having FPGA-based hardware acceleration capability. The use of FPGAs for accelerating floating-point computational kernels is a reality.

To get high-performance out of such FPGA-based kernels, deeply-pipelined floating-point arithmetic cores are essential. Unfortunately, the data hazards associated with pipelined floating-point functional units introduce a new class of problems that *do not exist* for single-cycle units. In particular, data reduction such as accumulation of sequentially delivered floating-point values, is problematic. A single-stage floating-point accumulator is out of the question because its operating frequency is too low. A multi-stage accumulator severely reduces the performance of an otherwise high-performance computational kernel by introducing a multi-cycle stage into the pipeline. Buffering inputs may not work since it can lead to either buffer overflow or pipeline stalls. Specialized techniques such as delayed addition [7] may also lead to pipeline stalls.

Sending a stream of floating-point values back to the host processor for reduction consumes precious memory bandwidth [1] and may even negate the performance gains derived from hardware acceleration of the kernel; this is a *key* issue. Take matrix-vector multiply, $\mathbf{y} = A\mathbf{x}$, as an example. The input data size is clearly $\Theta(n^2)$, since A is an $n \times n$ matrix. The hardware must calculate $y_i = \sum_{j=0}^{n-1} a_{ij}x_j$ ($i = 0, \ldots, n-1$) for every element of vector $\mathbf{y}$. If the FPGA-based unit cannot efficiently reduce the $a_{ij}x_j$ ($i, j = 0, \ldots, n-1$) values to produce each y_i value, then the output data size is $\Theta(n^2)$ rather than $\Theta(n)$.

In this paper we introduce a method for designing scalable, stall-free floating-point reduction circuits that run in optimal $\Theta(n)$ time while requiring only $\Theta(\lg(n))$ space and a single pipelined floating-point unit.

1 "free FLOPS, expensive bytes"

1.1. Background

The classic reduction example is adding n numbers using a binary tree having $n-1$ adders and $\lg(n)$ tree levels. Reductions crop up in a number of situations such as dot product, vector norm, and matrix multiply. However, due to the resource utilization and routing complexity of floating-point cores on FPGAs, it is prohibitive to implement $n-1$ functional units on a single chip. Designers must translate large parallel cases into a sequence of smaller cases and reduce the stream of values that are subsequently produced. Furthermore, in some kernels, such as the Lennard-Jones force calculation, the values are calculated sequentially. Either way, we need techniques for reducing sequentially delivered floating-point values.

1.2. Example Design Scenario

In this section we use a design scenario to show the reduction problem is not trivial when dealing with pipelined functional units. A simplified matrix-vector multiply (MxV) is illustrated in Figure 1. We partition the matrix and vector into subrows of size k. At each clock edge the next k pairs enter the leaf-node multipliers. A few clock cycles later, the corresponding single subrow value, $r_{i,j}$, comes out of the root node, on the next cycle, $r_{i,j+k}$ comes out, and so forth. The box labeled "reduce" adds all the $r_{i,j}$ values and outputs y_i. We symbolize this reduction using an adder, an input buffer (ibuf), and a loop back through an output buffer (obuf).

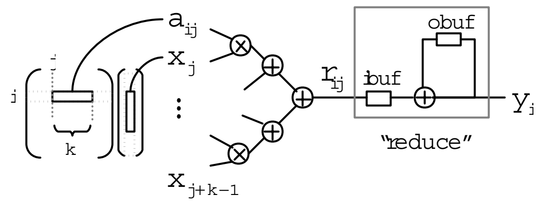


Figure 1. Matrix-Vector Multiply

1.2.1. Reduction Method A single-cycle floating-point adder runs at such a low frequency that it would severely reduce the performance of the pipeline. Using a pipelined adder and waiting for each pair to traverse the pipeline introduces a multi-cycle stage. Specialized techniques such as delayed addition, aside from being specific to accumulation reductions, may lead to pipeline stalls and are also dismissed. For high performance we use an α-stage pipelined floating-point adder running at the same speed as our input tree. When the first two subrow values, $r_{1,1}$ and $r_{1,2}$, are available, we enter them into the adder pipeline. The pipeline delay introduces a data hazard that prevents us from immediately adding the $r_{1,1}+r_{1,2}$ partial sum to $r_{1,3}$. Therefore, every other clock cycle we enter the next pair of $r_{1,j}$ into the pipeline, leaving *bubbles*[2] every other pipeline slot. After α cycles, partial sums appear at obuf. At this point we inject a value from obuf and the newly arriving $r_{1,j}$ into the adder pipeline (or a pair of the $r_{1,j}$ if there is no adder output). Within a few passes, all the bubbles bleed out of the pipeline. In this steady-state condition we have a full pipeline, an empty input buffer, and are inserting the obuf value and the newly arriving $r_{1,j}$ into the pipeline.

1.2.2. Buffer Growth Subrow $r_{2,1}$ arrives creating a potential pipeline schedule conflict. Since we don't know about the method being presented in this paper, we design an interlock circuit to prevent the adder from taking values out of ibuf. Recall, we have α row 1 partial sums in the pipeline. It takes α cycles to "empty" the pipeline, but these α partial sums are reentered pairwise into the pipeline so we now have $\alpha/2$ partial sums spaced every 2 slots, then $\alpha/4$ partial sums spaced every 4 slots, and so forth. Therefore, it takes $\alpha\lg(\alpha)$ cycles to drain row 1 from the pipeline. In the meantime ibuf has stored $\alpha\lg(\alpha)$ subrow values from row 2. The interlock circuit now permits injection of $r_{2,j}$ pairs from ibuf. After α cycles, the pipeline starts emitting row 2 partial sums into obuf. Therefore, we enter one of the ibuf values and the obuf value into the pipeline. The steady state for row 2 requires $\Theta(\alpha\lg(\alpha))$ storage. Since ibuf grows by $\Theta(\alpha\lg(\alpha))$ on every row, we need $\Theta(n\alpha\lg(\alpha))$ storage, *which is not realizable for large n.*

1.2.3. Options We have two options: i) stall the pipeline (presumably unacceptable), or ii) come up with a different design. We could develop a solution that exploits unique features of the given design scenario. However, a customized design solution for this scenario may not work in another. In this paper we present a reduction method that does not rely upon the characteristics of a particular problem.

1.3. Problem Statement

We have a design-specific maximum size, n, and p sets of floating-point numbers,

$$\left[\left(r_{0,0},\ldots,r_{0,n_1-1}\right),\ldots,\left(r_{p-1,0},\ldots,r_{p-1,n_p-1}\right)\right]$$

where $n_k \leq n$ for all k ($k=0,\ldots,p-1$). Throughout this paper, we assume n_k is a power of two. Suppose these sets arrive sequentially for reduction via a commutative, associative binary operation. The problem is to design a hazard-free reduction circuit from deeply-pipelined floating-point cores without requiring $\Theta(n)$ storage, without introducing

2 Pipeline stages that are not doing any useful work, i.e., empty slots.

a multi-cycle stage, and without introducing stalls into an otherwise high-performance computational pipeline.

1.4. Organization of this Paper

Section 2 describes our method for developing a compact, highly-scalable design to reduce p sets of floating-point values. The design only requires a single α-stage functional unit, $\Theta(\lg(n))$ storage, and, without stalling the pipeline, performs each reduction in the optimal $\Theta(n)$ time. In Section 3, we implement reference design instances based on our method and present statistics showing how the designs scale across a broad range of input sizes. We used a Xilinx Virtex-II Pro as the target device. Section 4 draws the conclusions.

2. Reduction Method

In this section we describe our method. We first introduce the intuition behind our method. We then present a single-set version of the algorithm and prove its correctness. We extend the algorithm to deal with multiple sets, and close with proofs of our performance and scalability claims.

2.1. Intuitive Idea

Before presenting our technique, we explain the intuition behind it. We use an adder in this section, but realize the technique works for any associative, commutative binary reduction operator. Start by visualizing a full binary reduction tree: n inputs , $n-1$ adders, and $\lg(n)$ tree levels $(0 \ldots \lg(n)-1)$. Since our problem statement requires sequential arrival of the inputs, we observe that adders at the same tree level begin execution at different times. Replace the level 0 (leaf) adders with a buffer feeding a single adder, replace the level 1 adders with another buffer feeding another adder, and so forth. Rather than having $n-1$ floating-point adders, we end up with $\lg(n)$ buffers and $\lg(n)$ floating-point adders as suggested in Figure 2.

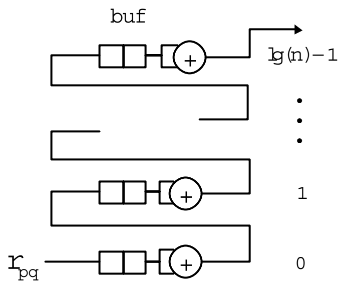


Figure 2. Simplified Binary Tree

Since the n elements are coming in sequentially, we notice none of the $\lg(n)$ adders in the simplified binary tree are fully utilized. The adder at level 0 can only perform an operation every 2 clock cycles, the adder at level 1 can only perform an operation every 4 clock cycles, and so forth. With careful scheduling, we can interleave all these additions in a single adder. Instead of the $\lg(n)$ buffers and $\lg(n)$ floating-point adders shown in Figure 2, we only need $\lg(n)$ buffers and a single α-stage floating-point adder as illustrated in Figure 3.

2.2. Description of the Architecture

We have some control circuitry, an α-stage pipelined adder, a $\lg(n)$-bit wide counter (C), and $\lg(n)$ buffer levels $(0 \ldots \lg(n)-1)$. Each level has a buffer of three 64-

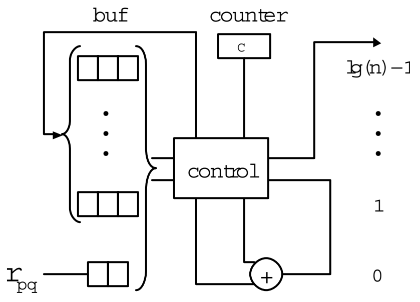


Figure 3. Reduction Architecture

bit words (except level 0, which only needs 2). Notionally, the r_{pq} inputs are clocked into buffer 0 every cycle. When we have two values, we destructively read buffer 0 and enter the pair into the pipeline. Two cycles later we enter the next pair, etc. A key observation is that we are only using the pipeline half the time. *This is the crux of our design technique*, we interleave partially reduced values in the open pipeline slots and buffer the intermediate results. This raises three questions, which we answer in the sections that follow:

- Are there enough pipeline slots?
- Can we schedule the slots without collision?
- How many buffers are needed at each level?

2.3. Available Slots

The idea of interleaving partial sums in available pipeline slots will only work if there are enough slots to handle arbitrarily large values of n. This section provides the necessary proof.

Theorem 1 *A single pipeline holds sufficient slots to simultaneously accommodate the partial reductions for any n.*

Proof. Since the pipeline always ingests two values, buffer 0, in Figure 4, can never cause more than $1/2$ of the pipeline slots to be used no matter how many numbers arrive. The partially reduced values produced by the pipeline are placed in buffer 1. Again, since they are taken by pairs these values can never cause more than $1/4$ of the slots to be used. This geometric series continues, we see that buffer i can never cause more than $1/2^{i+1}$ of the slots to be used. We want our technique to work no matter how many values arrive so we sum all these contributions

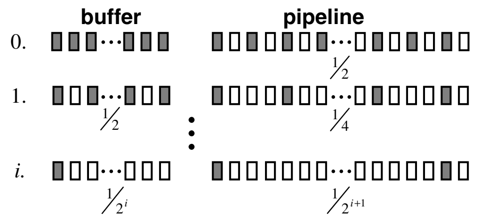


Figure 4. Contributions to the Fullness of the Pipeline

$$\lim_{n\to\infty}\sum_{i=0}^{n} 1/2^{i+1} = 1$$

Thus, we can never completely fill the pipeline by interleaving higher level partial reductions for any value of n. ■

2.4. Pipeline Schedule

The pipeline schedule is dictated by the buffer read and write schedule. To avoid any confusion, we note that "reading a buffer" is equivalent to entering a pair of values into the pipeline, and "writing a buffer" is equivalent to removing a single partial sum from the end of the pipeline. In this section we describe our scheduling approach and prove it guarantees a collision-free use of the pipeline.

2.4.1. Description of Schedule We initialize the $\lg(n)$-bit counter, C, shown in Figure 3 to 0 when buffer 0 is first read. The counter is incremented every clock cycle, and values arrive every clock cycle, so the next time we read buffer 0 the low order bit of C will be 0. We could express this as $C = *0$, where $*$ is a wildcard. After α cycles, the output of the pipeline is written into buffer 1, therefore the write schedule for buffer 1 is just $C-\alpha = *0$. We have used all the $*0$ slots, so all we have left are the $*1$ slots. We read buffer 1 when $C = *01$, and write buffer 2 when $C - \alpha = *01$. This leaves the $*11$ slots. We read buffer 2 when $C = *011$, and write buffer 3 when $C - \alpha = *011$, and so forth. This relationship between the buffer read and write schedule, pipeline depth, and counter C, is formalized as

$$C = 2^i - 1 + a2^{i+1} \quad (1)$$
$$C - \alpha = 2^i - 1 + a2^{i+1} \quad (2)$$

for some integer, a, where Equation 1 is the read schedule for buffer i, and Equation 2 is the write schedule for buffer $i + 1$.

2.4.2. Proof the Schedule is Collision-Free We have shown there are enough slots in the pipeline to simultaneously accommodate all the partial reductions and proposed a schedule for pipeline use. We now prove the proposed schedule allows collision-free use of the pipeline.

Theorem 2 *The schedule shown in Equation 1 and Equation 2 guarantees a collision-free use of the pipeline.*

Proof. We use a proof by contradiction. Assume buffer i and buffer j, where $j > i$, are scheduled for a read at the same time. From Equation 1 we have

$$2^i - 1 + a2^{i+1} = 2^j - 1 + b2^{j+1}$$

for some integers a and b, so

$$\begin{aligned} 2^i(1+2a) &= 2^j(1+2b) \Rightarrow \\ (1+2a) &= 2^{j-i}(1+2b) \end{aligned}$$

Since $j > i$, $(1 + 2a)$ must be even. But a is an integer, so $(1 + 2a)$ can never be even; we have our contradiction and proof that the read schedule is collision free. Values cannot change positions inside the pipeline so the write schedule is also collision free. Thus, the theorem is proved. ■

2.5. Buffer Growth and Maximum Size

Our reduction method only works if the buffer at each level: i) does not grow without bound, and ii) is limited to some constant size. This section presents proofs of both.

Theorem 3 *A buffer cannot grow without bound once we begin reading values from that buffer.*

Proof. For buffer i, *one* value is written (produced) every 2^i cycles, and *two* values are read (consumed) every 2^{i+1} cycles. Thus,

$$rate_{produced} = rate_{consumed}$$

and we cannot have unbounded growth once we start reading from the buffer. ■

Theorem 4 *The given schedule and the buffer size, $s = 3$, guarantee there can be no buffer overflow.*

Proof. We again use a proof by contradiction. Let the number of values in buffer i be denoted as b_i, and assume the first overflow, $b_i = 4$, occurs at time, t_{of}. Let the next read after the overflow occur at time, $t_{nr} = t_{of} + a$. Since $a = 0$

would make t_{of} a read cycle (no overflow), and reads are scheduled every 2^{i+1} cycles, we require $0 < a < 2^{i+1}$. Our schedule puts the previous read 2^{i+1} cycles earlier, that is

$$t_{pr} = t_{nr} - 2^{i+1} = (t_{of} + a) - 2^{i+1} \qquad (3)$$

In the worst scenario, buffer i has not overflowed and $b_i = 3$ just before time t_{pr}. At time t_{pr}, there are two cases: (1) we consume 2 of the values and leave $b_i = 1$, and (2) a value arrives precisely at time t_{pr} so we consume 2 of the values and leave $b_i = 2$. To overflow buffer i in case (1), we have to produce 3 more values by time t_{nr}. This takes *at least* $2 \times 2^i + 1 = 2^{i+1} + 1$ cycles. For case (2), we have to produce 2 more values by time t_{nr}. Since a value just arrived, it takes *exactly* $2 \times 2^i = 2^{i+1}$ cycles. Thus, if we define $c \in \{0, 1\}$, we can express our production time for both cases as, $t_{prod} = 2^{i+1} + c$. If there is an overflow, the following formula must be true:

$$t_{pr} < t_{of} - t_{prod} = t_{of} - \left(2^{i+1} + c\right)$$

Combining this with Equation 3, we get

$$(t_{of} + a) - 2^{i+1} < t_{of} - \left(2^{i+1} + c\right)$$

so $a < -c$. This result violates our condition $0 < a < 2^{i+1}$. We get our contradiction, and the maximum buffer size, $s = 3$, holds. ■

2.6. Single-Set Reduction Algorithm

Our algorithm for the single-set case is shown in Figure 5. For convenience, we define the last j bits of expression v, using the notation $v_{\langle j \rangle}$.

Theorem 5 *The reduction algorithm shown in Figure 5 properly reduces a single set of input values.*

Proof. We use a proof by construction. We have already proved the schedule prevents collisions, and the buffers will not overflow, so these do not need to be considered. Since the binary operation, addition, is both associative and commutative, we don't have to worry about the ordering or grouping of the operations. Therefore, our algorithm is correct if each adder input is operated on exactly once at each tree level, i.e., we don't add something more than once, or skip something. In the algorithm, each element is first stored in buffer 0, and then sent to the pipeline as an operand. Therefore, each element is operated on *at least* once. Moreover, when the element is read by the pipeline, buffer 0 no longer keeps a copy of it. Thus none of the subsequent operations will use the element as an operand, i.e., it is operated on *only* once. The schedule guarantees the partially reduced value coming out of the pipeline goes to the buffer at the next level, thus no operator touches it before it goes into buffer 1. Similar reasoning shows that each partially reduced value is operated on *exactly* once at every level. Therefore our algorithm is correct. ■

```
1: {counter}
2: if pipeline is first used then
3:   $C = 0$
4: else
5:   $C = C + 1$
6: end if
7: {buffers}
8: write input port to buffer 0
9: for $i = 0$ to $\lg(n) - 2$ do
10:   if $(C - \alpha)_{\langle i+1 \rangle} = 2^i - 1$ then
11:     write pipeline to buffer $i + 1$
12:   end if
13: end for
14: {pipeline}
15: for $i = 0$ to $\lg(n) - 1$ do
16:   if $(C_{\langle i+1 \rangle} = 2^i - 1)$ and
      (buffer $i$ has more than 1 values) then
17:     read 2 values from buffer $i$ into pipeline
18:   end if
19: end for
20: if the output of the pipeline is valid then
21:   if $C - \alpha = n/2 - 1$ then
22:     write pipeline to external memory
23:   end if
24: end if
```

Figure 5. Single-Set Reduction Algorithm

2.7. Multiple-Set Reduction Algorithm

The reduction algorithm can be extended to handle multiple sets of inputs with minimal modification. Suppose, we have p sets of floating-point numbers,

$$\left[(r_{0,0}, \ldots, r_{0,n_1 - 1}), \ldots, (r_{p-1,1}, \ldots, r_{p-1,n_p - 1})\right]$$

where $n_k \leq n$ for all $k = 0, \ldots, p - 1$. Recall, n, is some design-specific maximum size. Our approach requires each element to carry the set number, k, and the set size, n_k, with it as it travels through the pipeline. We simply build an α-stage companion pipeline to carry this meta-data. Before being written to the buffer, each output of the pipeline is tested to see if it is the final result of a set. If it is the final result, it is written to the external memory rather than the buffer. The exit test is, $i = \lg(n_k)$, where i is the potential destination buffer number. For notational convenience, we call a successful test *done* in our algorithm. Additionally, to keep the notation consistent, we define the output at the highest level as *buffer* $\lg(n)$ even though there is no actual buffer at this level. The modifications to the algorithm are minimal, so we only show the changed lines.

In the following proofs, we use the terminology *element* to refer to either a raw input value or a partial reduction of several input values. We also use the term *mingled* to de-

```
    ⋮
10: else if $((C-\alpha)_{\langle i+1\rangle} = 2^i - 1)$ and (not done) then
    ⋮
21: if (done) then
    ⋮
```

Figure 6. Reduction Algorithm

scribe an element having mixed composition, i.e., having values from more than one input set.

Theorem 6 *A reduced output from the reduction algorithm shown in Figure 6 only contains elements from a single input set.*

Proof. We use a proof by contradiction. Suppose buffer $i+1$ accepts the first mingled element, i.e., buffer $i+1$ contains an element, $R = r_u + r_v$, where element r_u has only elements from set u, and r_v has only elements from set v. Without loss of generality, we assume $u < v$. Clearly elements r_u and r_v were both in buffer i at some earlier clock cycle. There could not have been another element from set u in buffer i at the same time as r_u. Otherwise, it, rather than r_v, would have been entered into the pipeline with r_u. Since the multiple sets enter buffers sequentially, element r_u entered buffer i ahead of element r_v. Furthermore, since no other element from set u was in buffer i, r_u must be the final sum of set u. However, if r_u were the final sum of set u, our exit test would have caused r_u to be written to the external memory and not to buffer i, so r_u cannot be the final sum of set u. We have our contradiction and proof that the algorithm only produces reductions containing elements from a single input set. ■

Theorem 7 *The reduction algorithm shown in Figure 6 properly reduces each set of input values.*

Proof. We use a constructive proof. When there are multiple sets of inputs, the pipeline still reads/writes at most one buffer in each clock cycle using the same schedule as before. The incoming rate for a buffer may be reduced when a final result is written to the external memory. However, the outgoing rate of each buffer remains unchanged. Therefore, the algorithm remains free of data collision and buffer overflow. The algorithm has not introduced any new operations, so we know it still reduces all the input values, i.e., does not add values more than once, or skip values. A reduced output from the algorithm only contains elements from a single input set. Therefore, the algorithm properly reduces each set of input values. ■

2.8. Performance and Scalability

In this section we prove our method produces designs that have optimal time performance and are highly scalable.

2.8.1. Time Performance As noted in Section 1.1, designers must translate large parallel cases into a sequence of smaller cases and reduce the stream of values that are subsequently produced. Alternatively, in some kernels, values are calculated sequentially. Either way, we reduce sequentially delivered floating-point values. Given n such serially delivered values, the optimal time performance is clearly $\Theta(n)$.

Theorem 8 *Our reduction runs in optimal $\Theta(n)$ time.*

Proof. We use a constructive proof. We compare the latency of our algorithm with the theoretical minimum latency needed to reduce n serial inputs, α_{min}. The best we could ever achieve even with an unlimited number of buffers and α-stage adders is

$$\alpha_{min} = n + \alpha \lg(n) = \Theta(n)$$

since there are n serial inputs and $\lg(n)$ tree levels. We compute the latency of our reduction algorithm by examining the last input value as it goes through the reduction circuit. The last value enters buffer 0 at cycle n, and then traverses the pipeline, buffer 1, the pipeline ... buffer $\lg(n)-1$, the pipeline. Clearly it goes through the pipeline $\lg(n)$ times, and at buffer i, it waits at most $2^{i+1}-1$ cycles before being read by the pipeline. Therefore, the latency of our algorithm, α_{rm}, is

$$\begin{aligned}\alpha_{rm} &\leq n + \alpha \lg(n) + \textstyle\sum_{i=0}^{\lg(n)-1}\left(2^{i+1}-1\right) \Rightarrow \\ &\leq n + \alpha\lg(n) + 2n - 1 - \lg(n) \Rightarrow \\ &\leq 3n + (\alpha-1)\lg(n) - 1 \Rightarrow \\ &\leq 3\left(n + \alpha\lg(n)\right) \Rightarrow \\ &\leq 3\alpha_{min}\end{aligned}$$

Thus, $\alpha_{rm} = \Theta(n)$, which is the optimal time performance. ■

2.8.2. Scalability For a floating-point FPGA-based kernel, there are two aspects to scalability. The first is the number of floating-point units; since our method only requires a single floating-point unit no further discussion is needed. The second aspect is the amount of storage needed. Since there are $\lg(n)$ constant-size buffers, our reduction method uses $\Theta(\lg(n))$ space.

3. Experimental Results

Using the Xilinx ISE 6.2i and Mentor Graphics ModelSim 5.7 development tools with a Xilinx Virtex-II Pro XC2VP7 [13] as target device, we implement several designs based on our method. We present some design statistics, and show how the designs scale across a broad range of input sizes.

3.1. FPGA Implementation

To avoid large multiplexers at the input and output ports of the adder, we combine the $\lg(n)$ small buffers into one big storage. Instead of connecting to $\lg(n)$ data buses, each port of the adder only connects to one data bus. The buffers are realized using Block RAMs (BRAMs). We did not choose Distributed RAMs (DRAMs) because they use more slices and may make the routing complexity prohibitive. Address logic controls the memory location that is read or written by the adder. To simplify the address logic, we use a 4-word rather than a 3-word buffer. Thus the memory address contains $\lg(\textit{number of buffers}) + 2$ bits, that is, $\lg(\lg n) + 2$ bits. The first $\lg(\lg n)$ bits refer to the buffer being read or written, and the last 2 bits refer to the location inside the buffer. For example, suppose there are 3 buffers. To read the second entry of buffer 1, the read address will be 0101.

In our method, the adder needs to read two elements from a buffer in one clock cycle. Moreover, because buffer 0 is written every cycle, we also need to write to two different buffers in one clock cycle. Thus, the storage that implements the buffers needs 4 ports for read and write. Since the Block RAMs only provide two read/write ports, our solution is to clock the memory twice as fast as the other components of the design. Using the Xilinx Clocking Wizard [13] we obtain a doubled clock frequency from one of the DCM modules in the FPGA fabric.

In our experiments, we use the IEEE-format floating-point adders described in [5] for both the single precision (32-bit) and double precision (64-bit) implementations. Both of the adders are pipelined and achieve high clock speed. Their characteristics are shown in Table 1.

	32-bit	64-bit
Pipeline Delay	18	18
Area(Slices)	439	1140
Clock Speed(MHz)	230	200

Table 1. Floating-Point Adders

Table 2 shows the characteristics for 32-bit and 64-bit implementations of our design. In these implementations, n is fixed at 2^4. Due to the read delay of the Block RAM, the pipeline delays of the adders are increased by 2 when integrated into our reduction circuit. As seen in the table, our design uses additional slices for the address logic and the α-stage meta-data pipeline.

3.2. Design Scalability

In this section we show how the area and the speed of the reduction circuit vary with n. As n increases, the memory address logic becomes more complex. Thus the design occupies more slices and the achievable clock speed decreases. *However, as shown in Figure 7a, as n increases from 2^4 to 2^{24}, the area of a 32-bit design increases by about 40%, and the degradation in speed is slightly over 10%. From Figure 7b we see that the area of a 64-bit design increases by less than 33%, and the degradation in the speed is no more than 15%.* The small decrease in area for the 64-bit circuit at large values of n is due to variations in the place and route algorithm. Clearly our method is highly scalable across a broad range of input sizes. Furthermore the actual number of slices used, even for the largest design gives our method an exceptionally small footprint, which should easily fit into most design situations.

Adder	32-bit	64-bit
Adder Delay	20	20
Area(Slices)	633	1323
Clock Speed(MHz)	180	175

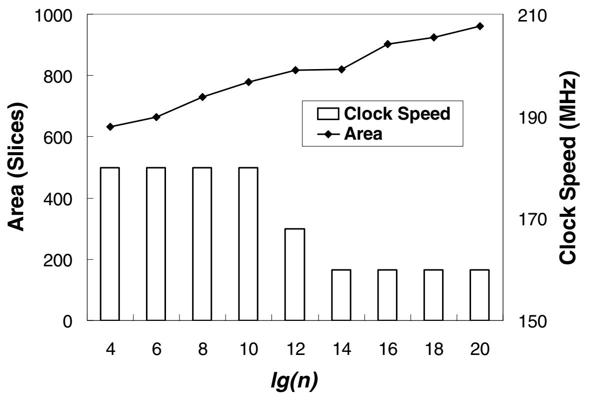
Table 2. Reduction Circuit ($n = 16$)

3.3. Applications

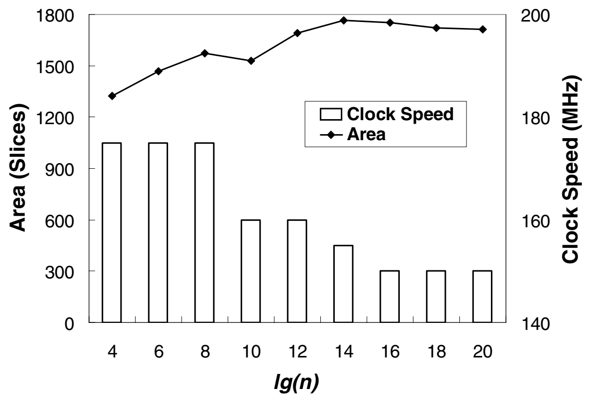
Certainly multiple instances of our circuit can be used to implement long summations of floating-point numbers in parallel. However, our reduction circuit is generally used as part of some other kernel. For example, dot product, used widely to determine whether vectors are orthogonal, requires accumulation of floating-point numbers and can use our design. In [15], we proposed an FPGA-based architecture for sparse matrix-vector multiply. In the architecture, a reduction circuit is used to accumulate the intermediate results. In [14], our design is used in FPGA-based designs for BLAS operations. Besides the linear algebra operations, our design can also be used in more complex applications. For example, in [10], our design can be used to perform the accumulation of all three force components on the FPGA. As a final example, our design can be used to implement the accumulation associated with the transcendental floating-point functions, e.g., cos, sin.

4. Conclusion

We have demonstrated a method for designing a reduction circuit from deeply-pipelined floating-point cores. Our data show that reduction circuits based on our method scale extremely well. *As n increases from 2^4 to 2^{24}, the area of a 32-bit design increases by about 40%, and the degradation in speed is slightly over 10%. The area of a 64-bit design increases by less than 33%, and the degradation in the speed is no more than 15%.* Furthermore the actual number of slices used, even for the largest de-



(a) 32-bit



(b) 64-bit

Figure 7. Characteristics of the Reduction Circuit for Various n

sign, corresponds to an exceptionally small footprint, which should easily fit into most design situations. The maximum clock frequency, even for the largest design was 150MHz, which is certainly acceptable. Designs targeted for the SRC MAPstation, for example, only require a 100MHz frequency [11]. The $\Theta(n)$ optimal time performance, $\Theta(\lg(n))$ space, coupled with a diminutive footprint and excellent scalability make this an ideal candidate for a number of applications where multiple sets of serially delivered floating-point numbers must be reduced.

Acknowledgement

Supported by the United States National Science Foundation under award No. CCR-0311823 and in part by award No. ACI-0305763.

References

[1] Altera Corporation. `http://www.altera.com/`.

[2] N. W. Bergmann and J. A. Williams. The Egret platform for reconfigurable System-on-Chip. In *Proceedings of the IEEE International Conference on Field-Programmable Technology*, pages 340–343, Tokyo, December 2003.

[3] Cray Inc. Cray XD1$^{\text{TM}}$. `http://www.cray.com/products/xd1/`.

[4] W. Fithian, S. Brown, R. Singleterry, and O. Storaasli. Iterative matrix equation solver for a reconfigurable FPGA-based hypercomputer. `http://www.starbridgesystems.com/resources/publications`, September 2003.

[5] G. Govindu, L. Zhuo, S. Choi, and V. K. Prasanna. Analysis of high-performance floating-point arithmetic on FPGAs. In *Proceedings of the 11th Reconfigurable Architectures Workshop*, Santa Fe, NM, April 2004.

[6] M. Leeser and X. Wang. Variable precision floating-point division and square root. In *Proceedings of the 8th Annual High Performance Embedded Computing Workshop, HPEC 2004*, pages 47–48, Lexington, MA, September 2004.

[7] Z. Luo and M. Martonosi. Accelerating pipelined integer and floating-point accumulations in configurable hardware with delayed addition techniques. *IEEE Transactions of Computers*, 49(3):208–218, March 2000.

[8] P. Lysaght and D. Levi. Of gates and wires. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, page 132, Santa Fe, NM, April 2004.

[9] U. Malik, K. So, and O. Diessel. Resource-aware run-time elaboration of behavioural FPGA specifications. In *Proceedings of the IEEE International Conference on Field-Programmable Technology*, pages 68–75, Hong Kong, December 2002.

[10] R. Scrofano and V. K. Prasanna. Computing Lennard-Jones potentials and forces with reconfigurable hardware. In *Proceedings of the Interational Conference on Engineering Reconfigurable Systems and Algorithms*, pages 284–290, Las Vegas, NV, June 2004.

[11] SRC Computers. MAPstation$^{\text{TM}}$. `http://www.srccomp.com/MAPstations.htm`.

[12] K. D. Underwood and K. S. Hemmert. Closing the gap: CPU and FPGA trends in sustainable floating-point BLAS performance. In *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2004.

[13] Xilinx Inc. `http://www.xilinx.com`.

[14] L. Zhuo and V. K. Prasanna. Design Tradeoffs for BLAS Operations on Reconfigurable Hardware. submitted to the 34th International Conference on Parallel Processing, 2005.

[15] L. Zhuo and V. K. Prasanna. Sparse Matrix-Vector Multiplication on FPGAs. In *Proceedings of the 13th ACM International Symposium on Field-Programmable Gate Arrays*, Montery, California, February 2005.