

High-Performance Reduction Circuits Using Deeply Pipelined Operators on FPGAs

Ling Zhuo, Gerald R. Morris, and Viktor K. Prasanna

Department of Electrical Engineering

University of Southern California

Los Angeles, USA

{lzhuo, grm, prasanna}@usc.edu

Abstract

Field programmable gate arrays (FPGAs) have become an attractive option for accelerating scientific applications. Many scientific operations such as matrix-vector multiplication and dot product involve the reduction of a sequentially produced stream of values. Unfortunately, because of the pipelining in FPGA-based floating-point units, data hazards may occur during these sequential reduction operations. Improperly designed reduction circuits can adversely impact the performance, impose unrealistic buffer requirements, and consume a significant portion of the FPGA. In this paper, we identify two basic methods for designing serial reduction circuits, the *tree-traversal method* and the *striding method*. Using accumulation as an example, we analyze the design tradeoffs between the number of adders, buffer size and latency, and propose high-performance and area-efficient designs using each method. The proposed designs reduce multiple sets of sequentially delivered floating-point values without stalling the pipeline or imposing unrealistic buffer requirements. Using a Xilinx Virtex-II Pro FPGA as the target device, we implemented our designs and present performance and area results.

G.1.0.g Parallel algorithms C.3.e Reconfigurable hardware

I. INTRODUCTION

Field programmable gate arrays (FPGAs) are semiconductor devices that are configured by end users to implement complex digital logic circuits. The increased gate count and other features of modern FPGAs allow them to be used as reconfigurable computational kernels. In particular, FPGAs have become an attractive option for speeding up applications in scientific computing [1]–[4], cryptography [5], [6] and network-based system [7]. The dramatic increase in the computing power of FPGAs also has prompted several supercomputer vendors to develop high performance reconfigurable computers that combine general-purpose processors (GPPs) with FPGAs [8], [9].

However, the MHz-scale clock rate of FPGAs is relatively low compared with the GHz-scale clock rate of GPPs. If high-performance is the motivation for using FPGAs, then the design must exploit both spatial parallelism as well as temporal parallelism. That is, the design needs to employ multiple

deeply pipelined functional units which operate concurrently. In reconfigurable computers where FPGAs co-exist with GPPs, additional levels of parallelism exist. Therefore, research on FPGA-based systems is a new and important topic within parallel and distributed computing area.

Unfortunately, using pipelined floating-point units for data reduction on FPGA may cause data hazards. Here “reduction” refers to reducing a series of sequentially delivered values to one value using one type of binary operator. One example of reduction is the accumulation of a stream of floating-point values using one or more adders. Throughout the paper, we assume that the operators used are commutative and associative. Note that our work does not consider numerical accuracy or stability.

The reduction problem arises in FPGA-based designs for many scientific applications. For example, in both dot product and matrix-vector multiplication, since the source matrices and vectors are usually too large to be stored on the FPGA, they need to be read in sequentially from the external memory. Thus, a series of floating-point values need to be accumulated. Another example is the Lennard-Jones force calculation in molecular dynamics [10]. At the end of the calculation, values that are generated serially need to be accumulated.

There has been work on reducing expression trees using parallel processors [11], [12]. However, most of it assumes that the operators are single-cycled and uses multiple types of operators. Reduction circuits for reducing vectors using pipelined operators also have been proposed, but they cannot be applied to our work either. Some of them use $\lg(n)$ operators and are impractical for FPGA-based implementation when n is large [13]. Others do not work efficiently with multiple input sets [14] and require large buffer sizes that are also impractical for FPGA-based implementation.

In this paper, we propose high-performance and area-efficient FPGA-based reduction circuits. Using accumulation as a prototypical example, we first analyze the design tradeoffs among the number of floating-point adders, the buffer size and the latency of a design. We then study two basic methods for implementing reductions using deeply pipelined adders. In the *tree-traversal method*, the inputs are considered as leaves of a binary tree and the reduction of the inputs requires a traversal of the tree. In the *striding method*, the inputs are first reduced to α values, where α is the number of pipeline stages in the adder. The α values are then further reduced to the single output value. As far as we know, this work is the first one that has fully studied the design tradeoffs and fundamental ideas for reduction circuits.

Based on the tradeoff analysis, three different reduction circuit designs are discussed. All of the

proposed designs are able to reduce *multiple input sets of arbitrary sizes* without stalling the pipeline. In these designs, the numbers of adders are fixed and the buffer sizes are independent of the number of input sets. The first design uses the tree-traversal method, and employs two adders and buffers of total size $\Theta(\lg(n))$. The other two designs are both based on the striding method. The second design uses two adders and a buffer of size $\Theta(\alpha \lg(\alpha))$. The third design only uses one adder and needs two buffers of size $\Theta(\alpha^2)$. The latencies of the designs depend on both n and α . When n is much larger than α , the latencies are $\Theta(n)$ clock cycles.

We implemented our designs on a Xilinx Virtex-II Pro FPGA [15] using our own double-precision floating-point adders. Note that the adders in the circuits can also be replaced by other associative and commutative binary operators for other types of reduction, e.g., $\min(x_0, x_1, \dots, x_{n-1})$. We study the area and speed performance of our designs. Experimental results show that our designs are area-efficient and can achieve high throughput.

The rest of the paper is organized as follows. Section II presents some background information on FPGAs and reconfigurable computers. Section III introduces the reduction problem and presents a formal definition. This section also discusses prior work related to reductions and the tradeoffs for designing reduction circuits. Section IV and Section V propose designs using the tree traversal method and the striding method, respectively. The correctness of the designs is proved and their operations are illustrated using examples. Section VI presents the performance of our reduction circuits. Section VII concludes the paper.

II. BACKGROUND

FPGAs are a form of reconfigurable hardware. They offer flexibility in design like software, but with time performance that can be closer to Application Specific Integrated Circuits (ASICs). FPGAs contain reconfigurable slices, hardware primitives such as block random access memory (BRAM), integer multipliers and programmable input/output (I/O) blocks, embedded in a programmable interconnection fabric. Our focus is static random access memory (SRAM)-based FPGAs, which can be *reprogrammed* by a configuration bitstream. In theory, we can design any digital logic circuit and place it on an FPGA. In practice, the primary constraints are area, clock speed, and I/O.

Earlier FPGAs were used for integer and fixed-point kernels that were not computationally demanding. However, contemporary FPGAs are now used to accelerate floating-point scientific applications

and have achieved performance that is competitive with GPPs [2], [10]. Underwood predicts that by 2009, FPGAs will have an order of magnitude peak floating-point performance over GPPs [16].

The dramatic increase in the computing power of FPGAs has aroused strong interest in the supercomputing industry. Several vendors have developed high performance reconfigurable computer (RC) architectures that combine GPPs and FPGAs. Cray has the XD1, which currently has six user-programmable FPGAs per chassis [9]; SRC Computers offers the MAP processor, which includes two user-programmable FPGAs, in single-MAP workstations or multiple-MAP clusters [8]; SGI has offerings such as the RC100 Blade, which has two user-programmable FPGAs [17]. The processing elements (PEs) in a traditional parallel compute node consist of GPPs and the associated memory hierarchy. As shown in Figure 1, RC compute nodes consist of fixed PEs plus variable structure PEs that contain FPGA(s) and local memory. High-speed, high-bandwidth interconnects allows the GPP-based PEs and FPGA-based PEs to communicate with one another. In the RC systems, the FPGAs serve as hardware application accelerators.

RCs are an important development in the parallel and distributed computing world because they provide additional levels of parallelism. We have the macroscopic parallelism provided by multiple PEs and the instruction level parallelism (ILP) of GPPs. Moreover, FPGAs provide two additional dimensions of parallelism. The first dimension is the spatial parallelism, where multiple functional units are configured on one FPGA and perform computations in parallel. The second dimension

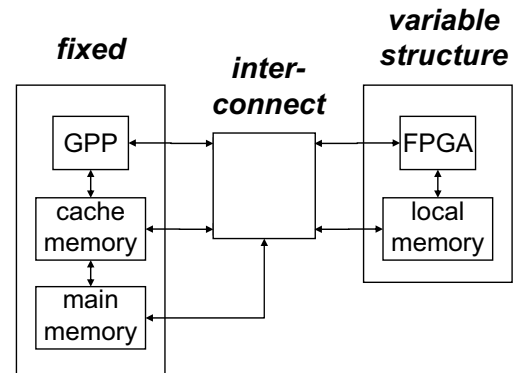


Fig. 1. Idealized RC compute node architecture

of parallelism comes from the pipelining in functional units, which also enables the FPGA-based designs to achieve high clock speed. However, the pipelining in the operators makes reduction of a series of data problematic when the data are delivered sequentially. This is explained in detail in the following section. On the other hand, sending a stream of floating-point values back to the GPPs for reduction consumes memory bandwidth and may even negate the performance gains derived from hardware acceleration; this is a *key* issue.

III. THE REDUCTION PROBLEM AND RELATED WORK

A. The Reduction Problem: An Intuition

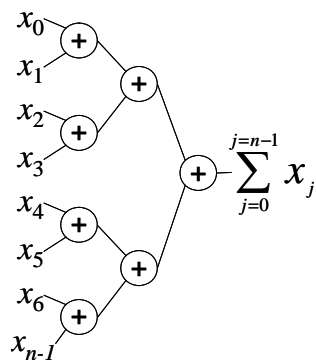


Fig. 2. Binary tree accumulator

Reductions are operations such as $\sum x_j$ that input one or more input sets and reduce them to a single value. Reductions can be implemented using a binary tree, e.g., to accumulate $n = 8$ numbers, we could use a full binary tree as shown in Figure 2. This binary tree of pipelined floating-point cores is a high-performance reduction architecture, which accepts an n -vector every clock cycle, and after the pipeline latency, emits one result every clock cycle. However, resource constraints preclude implementing a full binary tree even for modest values of n . Therefore, we must translate large reductions into smaller reductions and reduce the sequential stream

of values that are subsequently produced. Consider the dot product unit illustrated in Figure 3. It consists of the largest k -width dot product core that will fit on the FPGA followed by a looped adder to accumulate the stream of partial dot products. The operation appears to be straightforward. The vectors, \mathbf{x} and \mathbf{y} , are partitioned into k -vectors, \mathbf{u} and \mathbf{v} . At each clock edge one pair of k -vectors enter the pipelined k -width dot product core. There are $\lg(k)$ non-leaf levels in a full binary tree having k leaf nodes. If α_m and α_a are the latencies of the pipelined multiplier and adder floating-point cores, then the latency of the dot product core is $\alpha_d = \alpha_m + \alpha_a \lg k$ clock cycles. Therefore, after α_d cycles, the d_j partial dot products stream out, one value per clock cycle, and are accumulated

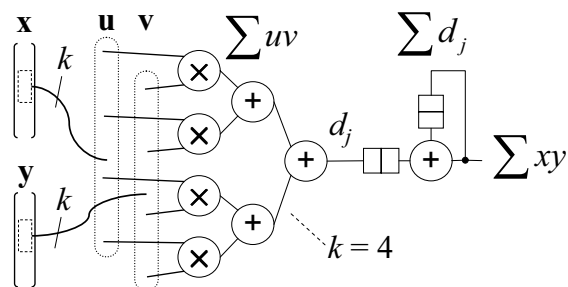


Fig. 3. Dot product unit (*broken*)

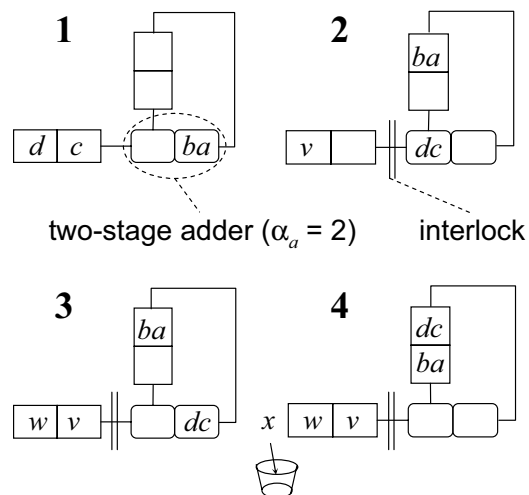


Fig. 4. Buffer overrun

by the looped adder to produce Σxy . Unfortunately, the naive adder loop fails because the adder is pipelined. Figure 4 shows a simple example of this failure for a pipelined adder having a latency of $\alpha_a = 2$. Suppose one dot product corresponds to the sequence (a, b, c, d) , and the next dot product corresponds to the sequence (v, w, x, y, z) , i.e., the sum $a + b + c + d$ is the value of the first dot product and $v + w + x + y + z$ is the value of the second dot product. In step 1, $a + b$ is in the last adder stage, and c and d are in the input buffers. In step 2, $c + d$, which are the final two input values in the current calculation, enter the adder pipeline. Value v from the next dot product calculation, is buffered. In step 3, w is also buffered. In step 4, we are still waiting to finish $a + b + c + d$. The interlock circuit prevents new values from entering the adder until the existing calculation is finished, so x gets dropped if we do not throttle the input stream (stall the pipeline) or use a bigger input buffer. Pipeline stalls significantly reduce performance. In addition, a buffer whose size increases linearly with n is unacceptable because n could be very large.

Thus, the reduction problem boils down to reducing multiple variable-length sets of sequentially delivered floating-point values without stalling the pipeline or imposing unreasonable buffer requirements. The reduction problem arises in FPGA-based designs for many scientific applications, including sparse matrix-vector multiplication [18], conjugate gradient [19], and Lennard-Jones force calculation in molecular dynamics [10]. Note that the clock speed of an FPGA-based design depends on the latency of the critical path. Thus, even with larger and faster FPGAs, to achieve high clock speed, the floating-point units still need to be pipelined and our reduction circuits are still useful.

B. The Reduction Problem: Formal Definition

We can define the reduction problem as follows: We are given p sets of inputs, with set i containing n_i floating-point values ($0 \leq i \leq p - 1$). The values in the sets are delivered sequentially as $[(s_{0,0}, \dots, s_{0,n_1-1}), \dots, (s_{p-1,0}, \dots, s_{p-1,n_p-1})]$. The problem is to reduce n_i values in set i into a single value such that $r_i = \sum_{j=0}^{n_i-1} s_{i,j}$, $i = 0, 1, \dots, p - 1$. The adders used are pipelined with α pipeline stages, and $\alpha > 1$.

Our goal is to design area-efficient and high-performance reduction circuits that satisfy the following requirements:

- 1) Reduce p sets of inputs correctly without causing any data hazard.
- 2) Accept one input during each clock cycle without stalling. When reducing multiple sets, out-

of-order outputs are allowed.

- 3) The number of adders employed is fixed, independent of p and n_i ($0 \leq i \leq p - 1$).
- 4) Low requirement on buffer size: buffer size of $\Theta(n_i)$ ($0 \leq i \leq p - 1$) or $\Theta(p)$ is unacceptable.

C. Previous Work

Research on reduction circuits started a couple of decades ago, when pipelined computers and vector computers first became available. Kogge proposed a method which uses $\lg(n)$ adders to reduce multiple input sets, where n is the maximum size of the input sets [13]. In this method, the reduction is regarded as an adder tree, and each adder is in charge of computations on one level of the tree. This method may be suitable for relatively small fixed-point FPGA-based implementations. However, due to the large size and design complexity of floating-point adders, floating-point FPGA-based implementations are infeasible for large values of n .

In [14], the authors proposed two reduction methods. Each method uses one adder and a buffer of fixed size. In these methods, n inputs are first reduced to α values which are then reduced to a single value, when $n \geq \alpha$. These methods are well suited for reducing one input set. However, for multiple input sets, the methods assume all the sets are available in the external memory before the computation starts. If the input sets are generated sequentially, a buffer is needed whose size increases with the number of sets. This is because the adder cannot accept new inputs when it is reducing the α values. Incoming inputs during this period of time have to be stored temporarily in a buffer. The large buffer requirement makes FPGA-based implementations of this method infeasible for large values of n .

Some researchers implement reduction using special-designed operators. In [20], the authors use variable precision floating-point adders and delayed normalization in order to reduce the number of pipeline stages. Our designs work with generic pipelined operators and do not require special operators. Moreover, the architectures and algorithms of our designs are independent of the implementations of the operators.

Several reduction circuits have been proposed that are suitable for implementations on FPGAs. When the size of each input set is a power of 2, the design in [21] performs reduction using one floating-point adder. This design uses $\lg(n)$ buffers of fixed size, where n is the upper bound on the size of an input set. However, this design does not work with arbitrary number of inputs. In [19], a partial summation unit using a single adder is used. However, this design, which was tailored for

high-performance in a high-level language based development environment, requires a fixed-sized output binary tree accumulator and $\Theta(n)$ buffer space. A similar design was employed in [22]. This design reduces the buffer requirement to $\Theta(\alpha)$, but still requires the large output accumulator. Essentially, we were able to trade space for performance in these particular cases. In the general case, however, designs tend to be area-constrained and we cannot afford the luxury of an output binary tree accumulator and $\Theta(n)$ buffer space. Obviously, none of the work described above meets the requirements listed in Section III-B.

In this paper, we propose three designs which are able to reduce multiple input sets of arbitrary sizes. These designs are based on two different methods. The numbers of adders in the designs are fixed and the required buffer sizes are moderate. Therefore, the designs have small areas and can be run at high clock speed. We also provide a thorough discussion on the design tradeoffs and the categorization of the intuitive ideas for reduction circuits. The designs are presented and analyzed based on the discussion.

D. Reduction Circuit Design Analysis

The general architecture of reduction circuits is shown in Figure 5. The datapath consists of the adders and the buffers. The control logic schedules the additions among the adders and controls the reads and writes of the buffers. The input values enter the architecture sequentially, either from an external memory or from another FPGA-based design. The output of the circuit is the final result of the reduction.

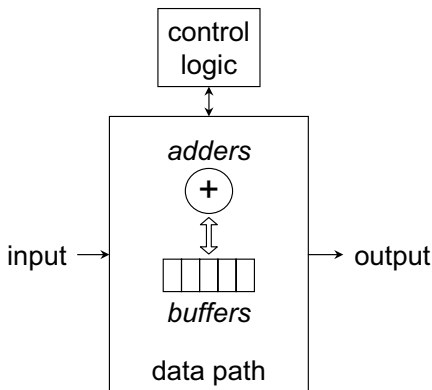


Fig. 5. General architecture of reduction circuits

We now discuss the design tradeoffs among the number of adders, the buffer size and the latency of the design. When a reduction circuit design contains multiple floating-point adders, it is possible that these adders perform computations concurrently. Thus, the latency of the design can be greatly reduced. However, the area of the design also increases with the number of adders.

Another tradeoff exists between the number of adders and the buffer size of the design. When the design contains enough adders, each output of an adder can be taken immediately by another adder if the output is not the final result of the reduction.

In this case, the buffer size required by the design is minimized, but the size of the design can be large. In other words, the pipeline stages in the adders serve as the storage for the intermediate results. On the other hand, if the design contains a large buffer to store the intermediate results, the number of the adders, and hence the area of the design, can be reduced.

There is also a tradeoff between the area of the design and the complexity of the control logic. When there are more adders in the design, each adder has a reduced workload, which simplifies the scheduling of the adders. On the other hand, when the design employs only a few adders, each adder has a higher workload and the intermediate results are stored in the buffers. In this case, the scheduling algorithm is more complex because it needs to avoid data hazards as well as buffer access conflicts.

IV. TREE-TRAVERSAL METHOD

A. *Intuitive Idea*

In the tree-traversal methods, the accumulation of a series of input values is represented by a binary adder tree. In this case, accumulating the inputs is equivalent to performing a breadth traversal of the tree. A simple solution is to use $n - 1$ adders to form a $\lceil \lg(n) \rceil$ -stage binary adder tree. The latency of this solution is $n + \alpha \lceil \lg(n) \rceil$ cycles. However, the large area of floating-point adders makes such a solution undesirable or, for large n , infeasible. In addition, since the inputs arrive sequentially, such an architecture makes uneconomical use of the adders. For example, the first adder at level 0 of the tree operates on the first two inputs to arrive. Before it can operate on any more inputs, it must wait for the next $n - 2$ inputs to arrive and enter the other adders in level 0. Such inefficiency exists at all levels of the tree.

The partially compacted binary tree (PCBT) design in Figure 6 can reduce this inefficiency. Instead of multiple adders at a given tree level, there is only one adder at that level. There is also a buffer with two entries at each level. When there are two values in the buffer, they are read from the buffer and passed to the adder. Thus, the design has $\lceil \lg(n) \rceil$ adders, a buffer size of $2 \lceil \lg(n) \rceil$, and a latency of $n + \alpha \lceil \lg(n) \rceil$ cycles. However, the utilization of the adders in this design is still very low. We know that the adder at level 0 takes new inputs every other clock cycle. The adder at level 1 must wait for two outputs of the adder at level 0; so, this adder only reads new inputs once every four cycles. In particular, the adder at level i only reads new inputs once every 2^{i+1} cycles. Additionally,

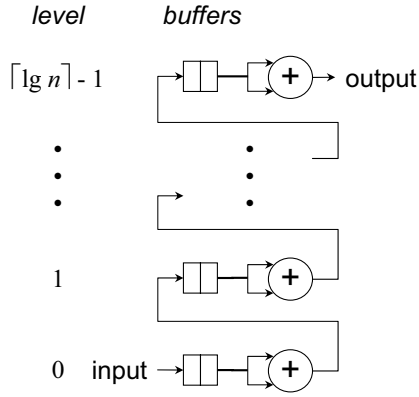


Fig. 6. Partially compacted binary tree architecture

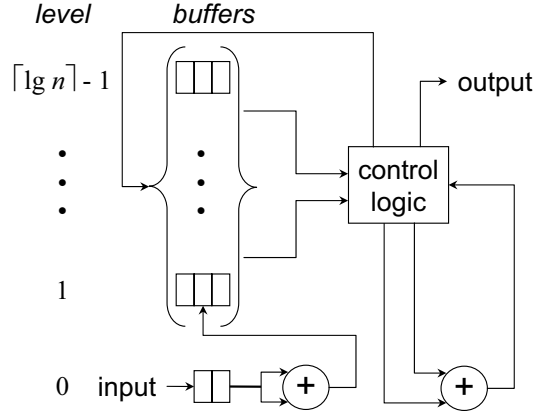


Fig. 7. Fully compacted binary tree architecture

this design requires $\lceil \lg(n) \rceil$ adders, which still may not be feasible for large values of n .

B. Proposed Design: Fully Compacted Binary Tree

1) *Architecture and Schedule*: We propose a fully compacted binary tree (FCBT) design that can reduce multiple input sets using only two floating-point adders. It works for arbitrary length sets, $n > 1$, up to a design-specific maximum ($n \leq n_{max}$). The architecture of the design is shown in Figure 7. The first adder performs reductions at level 0, and reads from buffer 0 during each clock cycle. The second adder is shared by levels 1, 2, \dots , $\lceil \lg(n) \rceil - 1$, and has to read from or write to different buffers in different clock cycles. When n is not a power of two, it is possible that some additions only have one input value and need to be padded with a zero value. We call such additions “singleton.” In our design, only the last addition at a given tree level can be singleton. For both adders in the design, there are two modes of addition: *normal* mode, and *singleton* mode. If buffer l contains 2 entries from the same set, the entries are read into an adder and a *normal* addition is performed. If buffer l only contains one entry from a set and $singleton_l = 1$, the entry is fed into an adder with a padded zero and a *singleton* addition is performed.

To set the correct value for $singleton_l$, we attach a label to each number that enters the buffers. The label shows how many of the inputs in the same set remain to be accumulated at that level. Figure 8 illustrates how the 5 inputs in a set are labeled at each level. The small boxes represent the inputs and their labels are shown below them. The inputs labeled as 0 do not really exist; they are shown in the figure to form the virtual binary tree on which the FCBT design is based. For each singleton add, an input is added with an input labeled as 0. We use t_u to denote the label for input

u . We know that at level l , $\lceil \frac{n}{2^{l+1}} \rceil$ additions are performed. Thus, if $t_u \geq 2^{l+1}$, u cannot be the last input of the set at level l . There must be another input, v , following u . Thus, $singleton_l = 0$, and u is then written into buffer l . When v arrives, u and v are then read as operands of the second adder. On the other hand, if $t_u < 2^{l+1}$, item u must be the last input of the set at level l . If no other input of the set is waiting, u requires a singleton add, and $singleton_l = 1$. However, if another input of the set is waiting for u , the addition of the two inputs yields the final result.

The scheduling algorithm for the FCBT design is shown in Figure 9. For convenience, we define the last j bits of expression, v , using the notation $v_{\langle j \rangle}$. We also use op_1 and op_2 to denote the outputs of the first adder and the second adder, respectively. We use $out_n = \lceil \lg(n) \rceil$ to represent the adder level at which the final result is found.

According to the algorithm, buffer 0 is read during every clock cycle. Buffer l ($l = 1, \dots, \lceil \lg(n) \rceil - 1$) is read when $C = 2^l - 1 + a2^l$ for some integer a , where C is the value of a clock cycle counter inside the controller. Counter C is $\lceil \lg(n) \rceil$ -bit wide. Buffer $l+1$ ($l = 0, \dots, \lceil \lg(n) \rceil - 2$) is written to when $C - \alpha = 2^l - 1 + b2^l$ for some integer b .

2) *Proof of Correctness:* We now prove the correctness of this design.

Theorem 1: The FCBT scheduling algorithm shown in Figure 9 guarantees a collision-free use of the adders.

Proof: The statement is obviously true for the first adder since it always adds two values as soon as they are available. The second adder can hold the additions from level $1, \dots, \lceil \lg(n) \rceil$. Since buffer l is read every 2^l clock cycles, the additions at level l at most occupy $\frac{1}{2^l}$ of the adder's utilization. Because $\lim_{n \rightarrow \infty} \sum_{l=1}^n \frac{1}{2^l} = 1$, the utilization of the second adder can never exceed 1. Thus, the usage of the second adder is also collision free. ■

According to the schedule in Figure 9, a buffer cannot grow without bound once we begin reading values from that buffer. Moreover, according to the schedule, buffer 0 at most contains 2 values before it is first read; buffer l ($1 \leq l \leq \lceil \lg(n) \rceil - 1$) at most contains 3 values when it is first read. Thus, there cannot be buffer overflow.

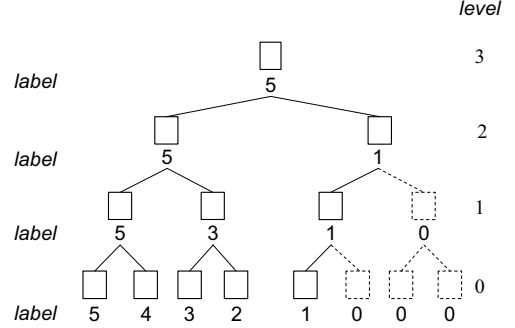


Fig. 8. Labels for FCBT inputs ($n = 5$)

```

{counter}
if the first adder is first used then
  C = 0
else
  C = C + 1
end if
{buffers}
write input port to buffer 0
if  $out_n \leq 1$  then
  write  $op_1$  to external memory
else
  write  $op_1$  to buffer 1
end if
for  $l = 1$  to  $\lceil \lg(n) \rceil - 2$  do
  if  $(C - \alpha)_{(l)} = 2^l - 1$  then
    if  $out_n \leq l + 1$  then
      write  $op_2$  to external memory
    else
      write  $op_2$  to buffer  $l + 1$ 
    end if
  end if
end for

{adders}
if  $singleton_0 = 0$  then
  read 2 values from buffer 0 into the first
  adder
else
  perform a singleton add in the first adder
end if
for  $l = 1$  to  $\lg(n) - 1$  do
  if  $C_{(l)} = 2^l - 1$  then
    if  $singleton_l = 0$  then
      read 2 values from buffer  $l$  into the
      second adder
    else
      perform a singleton add in the sec-
      ond adder
    end if
  end if
end for

```

Fig. 9. Scheduling algorithm for the FCBT design

We have proved the schedule of the FCBT design prevents collisions and that the buffers will not overflow. Since the binary operation, addition, is both associative and commutative, we do not have to worry about the ordering or grouping of the operations. According to the scheduling algorithm, each input and intermediate result are added once and only once. Thus, our FCBT design properly reduces an input set.

Theorem 2: The FCBT design reduces n inputs in less than $3n + (\alpha - 1)\lceil \lg(n) \rceil - 3$ cycles. When n is a power of 2, the FCBT reduces n inputs in $n + \alpha \lg(n)$ cycles.

Proof: It takes n cycles for all the inputs to arrive. The last value enters buffer 0 at cycle n , and then traverses the first floating-point adder. Then it enters buffer 1, the second floating-point adder, buffer 2, the second floating-point adder, \dots , buffer $\lceil \lg(n) \rceil - 1$, and the second floating-point adder. Clearly it goes through the floating-point adder $\lceil \lg(n) \rceil$ times. At buffer l ($l = 1, \dots, \lceil \lg(n) \rceil - 1$), it waits for at most $2^l - 1$ cycles before being read by the floating-point adder. Therefore, the latency of the algorithm is at most $n + \alpha \lceil \lg(n) \rceil + \sum_{l=1}^{\lceil \lg(n) \rceil - 1} (2^l - 1) \leq n + \alpha \lceil \lg(n) \rceil + 2n - 2 - \lceil \lg(n) \rceil - 1 = 3n + (\alpha - 1) \lceil \lg(n) \rceil - 3$ cycles. When n is a power of 2, the waiting time at all buffers equals zero. Thus, the latency of the algorithm is $n + \alpha \lg(n)$ cycles. ■

clock cycle	counter value	buffer contents ($n = 7$)				adder pipelines ($\alpha = 4$)				
1	000	s_2	s_1		buffer 0				adder 1	
2	001			s_3	buffer 0	s_1+s_2			adder 1	
3	010	s_4		s_3	buffer 0		s_1+s_2		adder 1	
4	011		s_5		buffer 0	s_3+s_4		s_1+s_2	adder 1	
5	100		s_5	s_6	buffer 0		s_3+s_4	s_1+s_2	adder 1	
6	101	s_7			buffer 0	s_5+s_6		s_3+s_4	adder 1	
		s_1+s_2			buffer 1					
7	110				buffer 0	s_7	s_5+s_6		s_3+s_4	adder 1
		s_1+s_2			buffer 1					
8	111	s_1+s_2	s_3+s_4		buffer 1		s_7	s_5+s_6	adder 1	
9	000	s_1+s_2	s_3+s_4		buffer 1		s_7	s_5+s_6	adder 1	
10	001			s_5+s_6	buffer 1			s_7	adder 1	
						$s_1+\dots+s_4$			adder 2	
11	010	s_7		s_5+s_6	buffer 1				adder 1	
							$s_1+\dots+s_4$		adder 2	
12	011				buffer 2	$s_5+s_6+s_7$		$s_1+\dots+s_4$	adder 2	
13	100				buffer 2		$s_5+s_6+s_7$	$s_1+\dots+s_4$	adder 2	
14	101	$s_1+\dots+s_4$			buffer 2			$s_5+s_6+s_7$	adder 2	
15	110	$s_1+\dots+s_4$			buffer 2			$s_5+s_6+s_7$	adder 2	
16	111	$s_1+\dots+s_4$	$s_5+s_6+s_7$		buffer 2				adder 2	
17	000	$s_1+\dots+s_4$	$s_5+s_6+s_7$		buffer 2				adder 2	
18	001	$s_1+\dots+s_4$	$s_5+s_6+s_7$		buffer 2				adder 2	
19	010				buffer 2	$s_1+\dots+s_7$			adder 2	

Fig. 10. Snapshot of the FCBT design

The FCBT design also works for multiple input sets. If we can prove that neither the inputs nor the intermediate results from two input sets can ever be added, we will have shown that the FCBT design properly reduces multiple sets. We use the terminology *item* to refer to either a raw input value or a partial reduction of several input values. We use the term *mingled* to describe an item having mixed composition, i.e., an item having values from more than one input set.

Theorem 3: In the FCBT design, r_i only contains items of set i .

Proof: We use a proof by contradiction. Suppose buffer $l + 1$ accepts the first mingled item, i.e., buffer $l + 1$ contains an item, $V = v_{i1} + v_{i2}$, where item v_{i1} has only items from set $i1$, and v_{i2} has only items from set $i2$. Without loss of generality, we assume $i1 < i2$. Clearly items v_{i1} and v_{i2} were both in buffer l at some earlier clock cycle. There could not have been another item from set $i1$ in buffer l at the same time as v_{i1} . Otherwise, that item, rather than v_{i2} , would have been entered into the adder pipeline with v_{i1} . Since the multiple sets enter buffers sequentially, item v_{i1} entered buffer l ahead of item v_{i2} . Furthermore, since no other item from set u was in buffer l , v_{i1} must

either be a singleton at level l or be the final sum of set $i1$. However, if v_{i1} were a singleton item, it would have been added with a 0; if v_{i1} were the final sum of set u , it would have been written to the external memory and not to buffer l . Thus, we have our contradiction and proof that the algorithm only produces reductions containing items from a single input set. ■

3) *Snapshot*: In this section, we give a snapshot for small input sizes and a small α value to show how the FCBT design works. In Figure 10, we accumulate seven inputs ($n = 7$), and both adders have four pipeline stages ($\alpha = 4$). We use a 3-bit counter and three 3-slot buffers. If a buffer or pipeline is empty and will not be used again, we do not show it in the snapshot. Several points about this example are worth noticing. In clock cycle seven, a singleton add is performed, that is, the first pipeline adds s_7 with a padded zero. In clock cycle eight, although buffer 1 contains two items, it is not read, because the last bit of the counter is not 0. Instead, buffer 1 is read in clock cycle 9, when the last bit of the counter becomes 0. Similarly, buffer 2 is not read in clock cycle 16 because the last two bits of the counter are not 01. Instead, buffer 2 is read in clock cycle 18, when the last two bits of the counter are 01.

V. STRIDING METHOD

A. Intuitive Idea

The striding method for designing reduction circuits is based on a different idea: when $n \geq \alpha$, an α -stage pipelined floating-point adder can reduce n input values into α segments without any data hazard. Figure 11 illustrates the summation of n inputs into α intermediate results when $\alpha = 4$.

clock cycle	adder pipeline ($\alpha = 4$)			
1	s_1			
2	s_2	s_1		
3	s_3	s_2	s_1	
4	s_4	s_3	s_2	s_1
...				
8	s_4+s_8	s_3+s_7	s_2+s_6	s_1+s_5
...				
12	$s_4+s_8+s_{12}$	$s_3+s_7+s_{11}$	$s_2+s_6+s_{10}$	$s_1+s_5+s_9$
...				
n	$s_4+s_8+s_{12}+\dots+s_n$	$s_3+s_7+s_{11}+\dots+s_{n-1}$	$s_2+s_6+s_{10}+\dots+s_{n-2}$	$s_1+s_5+s_9+\dots+s_{n-3}$

Fig. 11. Reducing n inputs into α segments ($\alpha = 4$).

We use the term *coalesce* to refer to the final stage of reduction when all the partial reductions currently in the adder are coalesced into a single value. We can use the same adder to perform the

coalescence. However, during the coalescing stage, the adder cannot accept new inputs and they must be stored in a buffer. Since coalescing happens to every input set, the buffer size is now dependent on the number of input sets, p . When p is large, which is usually the case in matrix-vector multiplication, such a design is infeasible.

B. Proposed Design: Dual Strided Adder

1) *Architecture and Schedule*: We propose a dual strided adder (DSA) design whose buffer size is independent of the number of input sets as well as the sizes of the input sets. It contains two adders, one input buffer, and two output buffers with one entry each. After the last input in a set

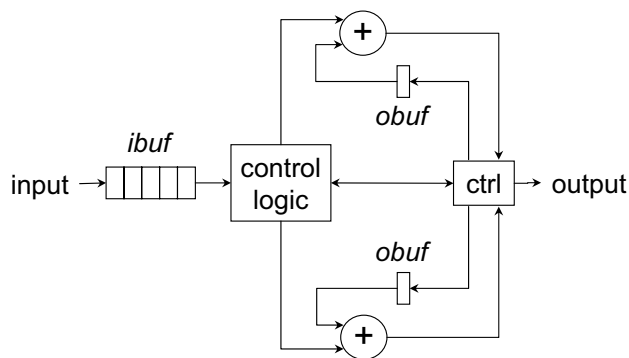


Fig. 12. Dual strided adder architecture

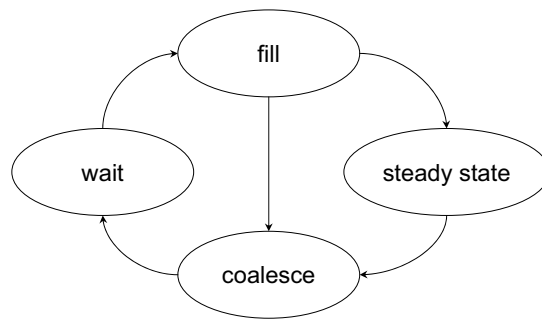


Fig. 13. DSA modes of operation

arrives, there are multiple partial reductions still in the adder, so another adder begins reducing the new set while the first adder finishes reducing the previous set. If both adders are busy, new inputs are buffered until an adder becomes available. The architecture of the DSA design is shown in Figure 12. The incoming input values are first stored in *ibuf*, which is a FIFO buffer. The FIFO, which also functions as a delay queue, ensures that an uninterrupted stream of values are delivered. Thus, the adders never have to stall once they begin reading data values for a given set. For each input set, *ibuf* delivers α pairs, one pair per clock cycle, and then delivers a single value, one per clock cycle until the end of the set. The adders in the DSA design operate in one of four modes as shown in Figure 13.

Wait Mode: The adder is waiting for a new set of values to arrive. When the first two values arrive, the circuit transitions into the *fill* mode.

Fill Mode: In this mode, the adder pipeline is filling up but has not yet produced any outputs. Refer to clock cycles 1 – 4 of Figure 14 for an example. After α cycles, partial sums appear at the adder output. At this point, the circuit transitions into the *steady state* mode. If the last value in the

set arrives while in the *fill* mode, the circuit transitions into the *coalesce* mode.

clock cycle	ibuf contents		adder pipeline ($\alpha = 4$)			
1	s_1	s_2	s_1+s_2			
2	s_3	s_4	s_3+s_4	s_1+s_2		
3	s_5	s_6	s_5+s_6	s_3+s_4	s_1+s_2	
4	s_7	s_8	s_7+s_8	s_5+s_6	s_3+s_4	s_1+s_2
5	s_9		$s_1+s_2+s_9$	s_7+s_8	s_5+s_6	s_3+s_4
6	s_{10}		$s_3+s_4+s_{10}$	$s_1+s_2+s_9$	s_7+s_8	s_5+s_6

Fig. 14. Initial DSA modes

Steady State Mode: In the steady-state mode the pipeline is full and a single value is in the FIFO. The adder pipeline output value and the single FIFO value are inserted into the adder. As each FIFO value is read, the newly arriving input is written into the FIFO. The idea is illustrated in cycles 5 – 6 of Figure 14. At any point in time, the pipeline is filled with partially reduced values. After the last value in the set is read from the FIFO, the adder stops reading the FIFO and the circuit transitions into the *coalesce* mode.

Coalesce Mode: The *coalesce* mode begins whenever the last value in a set is read from the FIFO. After coalescing the pipeline contents down to a single value, the adder goes back to the *wait* mode.

2) Proof of Correctness:

Theorem 4: It takes no more than $\alpha \lceil \lg(\alpha) + 1 \rceil$ cycles to coalesce an α -stage adder.

Proof: We proceed from the case which has the most unfinished work. This is when the α -stage adder pipeline has a value in every position, and the output buffer, *obuf*, is empty. This is depicted for $\alpha = 5$ in clock cycle t of Figure 15. After α cycles, all the partial reductions have moved through the pipeline. Every time there is one value in *obuf*, and one value at the pipeline output they are inserted back into the pipeline for further reduction. This means there are now $\alpha/2$ partial reductions in the pipeline, spaced every 2 positions, as depicted at clock cycle $t + 5$ in Figure 15.

After every α -cycle pass, the pipeline has $1/2$ the number of partial sums. After multiple passes through the pipeline there is a single value as depicted in cycle $t + 12$. It then takes an additional α cycles for this final reduced value to appear at the output. This repeated division 2 corresponds to $\lceil \lg(\alpha) \rceil$. Since it takes α cycles for each pass through the pipeline, and it takes $\lceil \lg(\alpha) + 1 \rceil$ passes to coalesce the pipeline, it takes at most, $\alpha \lceil \lg(\alpha) + 1 \rceil$ cycles to coalesce the pipeline. ■

clock cycle	adder pipeline ($\alpha = 5$)					obuf
t	p_1	p_2	p_3	p_4	p_5	
$t+1$		p_1	p_2	p_3	p_4	p_5
$t+2$	p_4+p_5		p_1	p_2	p_3	
$t+3$		p_4+p_5		p_1	p_2	p_3
$t+4$	p_2+p_3		p_4+p_5		p_1	
$t+5$		p_2+p_3		p_4+p_5		p_1
$t+6$			p_2+p_3		p_4+p_5	p_1
$t+7$	$p_4+p_5+p_1$			p_2+p_3		
$t+8$		$p_4+p_5+p_1$			p_2+p_3	
$t+9$			$p_4+p_5+p_1$			p_2+p_3
$t+10$				$p_4+p_5+p_1$		p_2+p_3
$t+11$					$p_4+p_5+p_1$	p_2+p_3
$t+12$	$p_1+\dots+p_5$					

Fig. 15. Coalescing a 5-stage adder

Theorem 5: Given two α -stage adders, an input buffer size of $\alpha \lceil \lg(\alpha) + 1 \rceil$ will not overflow.

Proof: We use a proof by contradiction. Assume the $\alpha \lceil \lg(\alpha) + 1 \rceil$ buffer has just overflowed for the first time. Let the next set to be processed be set j . This means we have $\alpha \lceil \lg(\alpha) + 1 \rceil$ values from set j (and perhaps later sets) in the buffer, and one more value has just arrived causing the overflow. If either adder were in *wait*, *fill*, or *steady state* mode then buffer values would be consumed on each cycle and an overflow could not occur. Thus both pipelines must be in the *coalesce* mode. The longest it can take to coalesce a set is $\alpha \lceil \lg(\alpha) + 1 \rceil$ cycles. But it took at least $\alpha \lceil \lg(\alpha) + 1 \rceil + 1$ cycles to cause an overflow. We have our contradiction and proof. ■

Theorem 6: The DSA design reduces n inputs in $n + \alpha \lceil \lg(\alpha) + 1 \rceil$ clock cycles.

Proof: As the FIFO never stalls, values are delivered every clock cycle. Clearly it takes n cycles for the last value to be written into the FIFO. We compute the latency of the DSA design by examining the last input value as it goes through the architecture. The last value enters the FIFO at cycle n , and then the pipeline. It takes no more than $\alpha \lceil \lg(\alpha) + 1 \rceil$ cycles to coalesce the pipeline. Thus, the latency of the design is $n + \alpha \lceil \lg(\alpha) + 1 \rceil$ cycles. ■

The total buffer size needed by the DSA design is $\alpha \lg(\alpha) + 3$. As all the values of each input set only go through one adder, input values from two different sets cannot be mingled. Thus, the DSA design also reduces multiple sets properly.

C. Proposed Design: Single Strided Adder

1) *Architecture and Schedule:* We next propose the single strided adder (SSA) design, which only uses one adder but requires larger buffer size. In other words, we trade area for buffer space. The intuitive idea behind the SSA design is this: Suppose α distinct sets are stored in a buffer and each set has α items. Then we can interleave the additions from the α sets so that the adder is fully utilized and no data hazard occurs. In this way, reducing α^2 items from α distinct sets at most takes α^2 clock cycles. At the same time, another buffer of size α^2 is needed to store the new inputs. Thus, we can reduce multiple inputs sets with one adder and two buffers of size α^2 .

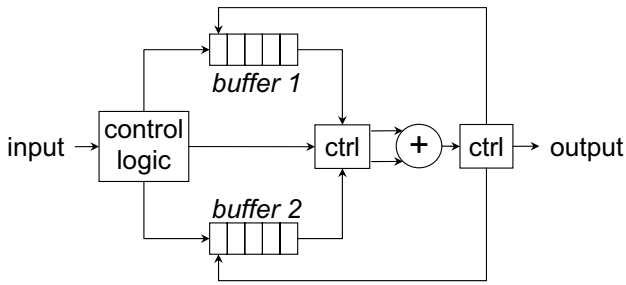


Fig. 16. Single strided adder architecture

The architecture is shown in Figure 16. It contains one floating-point adder and two buffers of size α^2 . The input can either enter one of the buffers, or enter the adder directly. The adder selects its operands from the input and the buffers. If the output of the adder is the final result of a set, it is written to the external memory; otherwise, it is written back to one of the buffers. The buffer which accepts new inputs is denoted as Buf_{in}^f . For set i , if $n_i \leq \alpha$, n_i inputs are written into Buf_{in}^f directly; otherwise, α inputs are written into Buf_{in}^f and are then added with the remaining new inputs of the set. In the n_i (if $n_i > \alpha$) or α (if $n_i \leq \alpha$) cycles in which Buf_{in}^f accepts new inputs, the adder is used by another buffer. This buffer is called Buf_{red}^f , which stores no more than α items for each set. Reading from Buf_{red}^f column by column, the adder interleaves α additions from α distinct sets. The results of these additions are written back to Buf_{red}^f . Buffer 1 and Buffer 2 function as Buf_{in}^f and Buf_{red}^f alternately. When Buf_{in}^f is full, the two buffers are swapped. That is, Buf_{in}^f becomes Buf_{red}^f and Buf_{red}^f becomes Buf_{in}^f .

Since n_i can be of any value larger than 1, when Buf_{in}^f is full, it may contain more than α distinct sets. The problem now is how to interleave the additions in these sets so that data hazards do not occur. We try to fit all the sets into α rows. This fitting problem can be reduced to the *subset-sum problem* which is NP-complete [23], if no input set is allowed to be partitioned among two rows. Therefore, we allow an input set to be split among two rows. Suppose the n_i ($n_i < \alpha$) or α ($n_i \geq \alpha$) items of set i are written into Buf_{in}^f in row-major order. We know that the items in a set can only exist in two consecutive rows. Otherwise, a set will have more than α items in Buf_{in}^f , which is not allowed by the algorithm.

Figure 17 describes the scheduling algorithm of the SSA design. It is assumed that the buffers allow read-after-write access. That is, if an entry is read and written in the same clock cycle, the output of the buffer is the value which is written. Two points need to be noted in the algorithm:

- 1) The first column of Buf_{red} is read twice. During the second read operation, $Buf_{red}[j\alpha]$ contains the intermediate result of the set which is split between row $j - 1$ and row j ($j = 1, \dots, \alpha - 1$).
- 2) It is possible that an input set is split between two buffers. This case can be seen as the set is split among row 0 and row $0 - 1$. A special register is used to store the intermediate result from row $0 - 1$. It is then added to $Buf_{red}[0]$.

if the input is among the first α inputs of a set
then

write the input into Buf_{in}

if it is the first input of the set, increment i

if Buf_{red} is not empty **then**

if $l < \alpha$ and ($Buf_{red}[j\alpha + l]$ and $Buf_{red}[j\alpha + l - 1]$ are in the same set) **then**

add $Buf_{red}[j\alpha + l]$ with $Buf_{red}[j\alpha + l - 1]$

else if $l = \alpha$ **then**

if a set is split between row $j - 1$ and row j **then**

add $Buf_{red}[(j - 1)\alpha + l - 1]$ with $Buf_{red}[j\alpha]$

else if a set is split between two buffers **then**

add $Buf_{red}[0]$ with $temp$

end if

end if

increment j

if $j + 1 = \alpha$, then $j = 0$, increment l

end if

else

adds the input with the k th item of set i in Buf_{in} , increment k

end if

{buffers}

if Buf_{in} is full or (no new input and Buf_{red} is empty) **then**

swap Buf_{in} and Buf_{red}

$j = 0$, $l = 1$, $k = 0$, $i = -1$

end if

{output of the adder}

if one operand is from $Buf_{in}[x]$ **then**

if it is not from the last set in Buf_{in} **then**
write the adder output to $Buf_{in}[x]$

else

write the adder output to $Buf_{red}[x]$

end if

else if one operand is the last item of a set in Buf_{red} **then**

write the output to the external memory

else if operands are from $Buf_{red}[y]$ and $Buf_{red}[y - 1]$ **then**

write the output to $Buf_{red}[y]$

else if one operand is from row j and is the last item of a set split between row $j - 1$ and row j **then**

write the output to $Buf_{red}[j\alpha]$

else if one operand is from row 0 and is the last item of a set split between two buffers **then**

write the output to $Buf_{red}[0]$

else if one operand is from row $\alpha - 1$ and is the last item of a set split between two buffers **then**

write the output to $temp$

end if

Fig. 17. Scheduling algorithm for the SSA design

2) *Proof of Correctness*: We now prove the correctness of the SSA design.

Theorem 7: The SSA scheduling algorithm shown in Figure 17 guarantees a collision-free and

hazard-free use of the adder and does not cause buffer overflow.

Proof: The use of the adder is collision-free because the adder only reads from Buf_{red} when the Buf_{in} is accepting new inputs.

We next prove data hazards do not occur in the algorithm. When Buf_{in} is full, each column of it contains items from α distinct sets. Otherwise, there must be a set that contains more than α items in Buf_{in} , which is not allowed by the algorithm. Therefore, through interleaving additions in α distinct rows, we interleave additions in α distinct sets. Thus the use of the adder is hazard-free.

We next prove the buffers do not overflow. Initially, both buffers are empty. Without loss of generality, suppose Buffer 1 is used as Buf_{in} , and Buffer 2 is used as Buf_{red} . After the inputs of α sets enter the architecture, Buffer 1 is full and is designated as Buf_{red} . The algorithm then takes α^2 clock cycles to read from Buffer 1. During these clock cycles, the first α inputs of α consecutive sets are written into Buffer 2. When the remaining inputs of the sets enter the architecture, the adder accepts operands from Buffer 2, and not Buffer 1. After α sets enter the architecture, Buffer 1 is empty and Buffer 2 is full; then Buffer 1 is designated as Buf_{in} , and Buffer 2 is designated as Buf_{red} . The algorithm continues in this way until the last input enters the architecture. Thus, the buffers never overflow. ■

Theorem 8: The circuit correctly reduces multiple input sets, i.e., $r_i = \sum_{j=0}^{n_i-1} s_{i,j}$, $i = 0, \dots, p-1$.

Proof: We first prove that each set in Buf_{red} is reduced correctly. Clearly, this is true if set i is not split between two rows. Suppose set i is split between row $j-1$ and row j , $j = 1, \dots, \alpha-1$. The items in row $j-1$ are summed up and the intermediate result is written to $Buf_{red}[(j-1)\alpha + \alpha - 1]$. If only one item of the set is stored in row j , it must be $Buf_{red}[j\alpha]$ because the items of a set are stored consecutively in row-major order. As $Buf_{red}[(j-1)\alpha + \alpha - 1]$ is added with $Buf_{red}[j\alpha]$, the set is reduced correctly.

If two or more items of set i are stored in row j , they are summed up and the intermediate result is written to $Buf_{red}[j\alpha]$ before $l = \alpha$ (l is used in Figure 17). Otherwise, the set contains more than $\alpha-1$ items in row j , and thus contains more than α items in Buf_{in} . Since $Buf_{red}[(j-1)\alpha + \alpha - 1]$ is added with $Buf_{red}[j\alpha]$ when $l = \alpha$, the set is reduced correctly. The case in which set i is split between two buffers is similar to the case where the set is split between row 0 and row 0-1. Thus, we can prove the set is reduced correctly using a similar analysis.

We now prove that r_i only contains items from a single set i . Suppose during some clock cycle, the adder accepts two operands from different sets. This cannot happen when $l < \alpha$ because it is not allowed by the algorithm. Suppose when $l = \alpha$, the items in $Buf_{red}[(j-1)\alpha + l - 1]$ and $Buf_{red}[j\alpha]$ are from different sets. In this case, no set is split between row $j-1$ and row j ; hence these items will not be read by the adder. Similarly, when $l = \alpha$, if the items in $Buf_{red}[0]$ and in $temp$ are from different sets, they will not be read by the adder. We thus reach a contradiction. ■

Theorem 9: The circuit reduces p sets in at most $(\sum_{i=0}^{p-1} n_i + 2\alpha^2)$ cycles.

Proof: Since the reduction circuit never stalls reading the inputs, the last element of the last set enters the circuit in clock cycle $\sum_{i=0}^{p-1} n_i$. At this time, if Buf_{in} is full and Buf_{red} is empty, they are swapped. Hence the adder reads from Buf_{red} , which is full. This takes α^2 clock cycles.

If only q ($q < \alpha^2$) entries of Buf_{in} is written as the last input enters, Buf_{red} is not empty. Thus, the adder first reads from Buf_{red} before the two buffers are swapped. This takes $\alpha^2 - q$ clock cycles. Then Buf_{in} becomes Buf_{red} and is read by the adder. The operands of the last addition enter the adder in clock cycle $\sum_{i=0}^{p-1} n_i + (\alpha^2 - q) + \alpha(\alpha - 1) + \lceil \frac{q}{\alpha} \rceil$. As $q > \lceil \frac{q}{\alpha} \rceil$, the final result of the p th set is available in clock cycle $\sum_{i=0}^{p-1} n_i + (\alpha^2 - q) + \alpha(\alpha - 1) + q + \alpha = \sum_{i=0}^{p-1} n_i + 2\alpha^2$. ■

D. Snapshot

We illustrate the operations of an $\alpha = 4$ SSA design using a snapshot. During clock cycle 0, $Buf_{in} = \text{Buffer 1}$ and $Buf_{red} = \text{Buffer 2}$; both buffers are empty. The inputs are stored in the buffers in row-major order. In particular, the memory addresses increase from left to right, and from top to bottom. The data in the adder travels from the top to the bottom.

There are 10 input sets of size 3. We show the data in the buffers and the adders in the selected clock cycles in Figure 18.

VI. PERFORMANCE ANALYSIS

A. Summary of Proposed Designs

Table I summarizes the characteristics of the proposed designs. The PCBT design, which is included in the tables, is unrealistic for most FPGA-based designs because of the large number of floating-point adders. However, it is very simple to implement and is one of the earliest reduction circuits. By including the PCBT design, our analysis of the tradeoffs among the number of adders, the latency, and the buffer size can be more complete.

clock cycle	buffer 1 contents	adder ($\alpha = 4$)	buffer 2 contents	notes
16	s_{00} s_{01} s_{02} s_{10}			At clock cycle 16, buffer 1 is full, and we swap Buf_{in} and Buf_{red} .
	s_{11} s_{12} s_{20} s_{21}			
	s_{22} s_{30} s_{31} s_{32}			
	s_{40} s_{41} s_{42} s_{50}			
20		$s_{40} + s_{41}$	s_{51} s_{52} s_{60} s_{61}	In clock cycles 17-20, the adder reads operands from 4 rows of buffer 1. Since s_{22} is not in the same set as s_{30} , they are not added.
		$s_{11} + s_{12}$		
	s_{22} s_{30} s_{31} s_{32}	$s_{00} + s_{01}$		
		$s_{40} + s_{41} + s_{42}$		
24		$s_{30} + s_{31}$	s_{51} s_{52} s_{60} s_{61}	In clock cycles 21-24, the adder continues reading from buffer 1. $s_{11} + s_{12}$ is written to the first location in row 1 of buffer 1.
	$s_{11} + s_{12}$ s_{20} s_{21}	$s_{00} + s_{01} + s_{02}$	s_{62} s_{70} s_{71} s_{72}	
	s_{22} s_{50}			
			s_{51} s_{52} s_{60} s_{61}	
28		$s_{30} + s_{31} + s_{32}$	s_{62} s_{70} s_{71} s_{72}	Starting from clock cycle 29, the intermediate results of split rows are summed up.
	$s_{11} + s_{12}$	$s_{20} + s_{21}$	s_{80} s_{81} s_{82} s_{90}	
	s_{22}			
		s_{50}		
32			s_{51} s_{52} s_{60} s_{61}	At clock cycle 32, buffer 1 is empty and the adder begins to reduce sets in buffer 2. Note that $Temp = s_{50}$.
		$s_{20} + s_{21} + s_{22}$	s_{62} s_{70} s_{71} s_{72}	
		$s_{11} + s_{12} + s_{10}$	s_{80} s_{81} s_{82} s_{90}	
			s_{91} s_{92}	
48		$s_{91} + s_{92} + s_{90}$		At clock cycle 48 the operands of the last addition enter the adder
		$s_{63} + s_{60} + s_{61}$		
		$s_{51} + s_{52} + s_{50}$		

Fig. 18. Snapshot of the SSA design

The FCBT design is relatively easy to implement. It requires two adders, has the smallest buffer size, but has one of the longest theoretical latencies for even modest values of n . The DSA design has a relatively modest buffer size that is independent of n , uses two adders, and has the shortest theoretical latencies. Unfortunately, the DSA design also has the most complex implementation, which will become clear later. Finally, the SSA design only employs one adder, has a reasonable theoretical latency for larger values of n , but has a relatively complex implementation and the largest buffer size.

The PCBT design and the FCBT design need to know the maximum number of inputs before the computation starts, while both the DSA design and the SSA design have no upper bound on input size. On the other hand, when there are multiple input sets, the throughput for individual sets using the PCBT design and the FCBT design is easy to determine. Except the first set, the throughput for each set equals its number of inputs. For the DSA design and the SSA design, however, the throughput depends not only on the number of inputs in the current set, but also on the sizes of

the previous and the subsequent sets. The DSA design can produce out-of-order outputs whenever the sizes of the inputs sets vary. A long (earlier) row might still be coalescing while a short (later) row finishes. A classic example of when this might occur is if the DSA design is used as part of a sparse matrix-vector multiplication. In our DSA design, we include a meta data pipeline to carry row identification information along with the data. When the input sizes are less than α , the outputs of the SSA design may also be out of order. In this case, we can first write the outputs into an output buffer, and reads them out in order. As each input set at least contains 2 elements, the maximum size of the output buffer is $\frac{\alpha^2}{2}$.

TABLE I
SUMMARY OF THE PROPOSED DESIGNS

Design	No. of Adders	Buffer Size	Latency for n inputs	Out-of-order Output
PCBT	$\lceil \lg(n) \rceil$	$2 \lceil \lg(n) \rceil$	$n + \alpha \lceil \lg(n) \rceil$	No
FCBT	2	$3 \lceil \lg(n) \rceil$	$\leq 3n + (\alpha - 1) \lceil \lg(n) \rceil$	No
DSA	2	$\alpha \lceil \lg(\alpha) + 1 \rceil$	$n + \alpha \lceil \lg(\alpha) + 1 \rceil$	Yes
SSA	1	$2\alpha^2$	$\leq n + 2\alpha^2$	Yes

B. Performance Metrics

We use the following metrics to evaluate our designs:

- **Area:** The number of configurable slices used by the design.
- **Clock Speed:** The maximum achievable speed of the design. As one floating-point value enters the design during each clock cycle, the design is able to handle an incoming data rate of $64 \times \text{clock speed}$ bits/s.
- **Total Time:** The period of time between when the first input arrives and when the last result is output.
- **Throughput:** It is computed as $(\text{total number of outputs})/(\text{output time})$. The output time is the period between when the first result is generated and when the last result is output. For applications which sequentially generate input sets for reduction, high throughput is more desirable than shorter latency. Thus, when multiple sets are reduced, throughput is a better measurement of performance than total time.

To evaluate the area-efficiency of our designs, we also use the following metrics:

- **Area-Time Product:** It is calculated as $Area \times Total\ Time$, in thousand slices $\times \mu s$. A lower value indicates better performance.
- **Throughput-To-Area Ratio:** It is calculated as $Throughput/Area$, in thousand results per slice per second. A higher value indicates better performance.

C. Design Implementations

We implemented our proposed designs on a Xilinx Virtex-II Pro XC2VP7 FPGA, which is a small device. One XC2VP7 contains only 4928 slices and about 100 KB of on-chip block RAM (BRAM) memory. We used Xilinx ISE 7.1i [15] and Mentor Graphics Modelsim 6.0a [24] for development.

The characteristics of our deeply pipelined IEEE-format double-precision floating-point adder, which is described in [25], are shown in Table II. Note that the proposed designs are mostly independent of the implementations of the adder, except that the buffer size of the DSA and SSA vary with α . Thus, any adder, actually any associative and commutative binary operator, can be plugged into our designs with few modifications to the architectures and the algorithms.

TABLE II
CHARACTERISTICS OF 64-BIT FLOATING-POINT ADDER

No. of Pipeline Stages (α)	Area (Slices)	Clock Speed (MHz)
14	892	170

The characteristics of the implemented circuits are shown in Table III, when $n = 128$. In this case, the PCBT design needs 7 adders, and had to be implemented on a larger device, XC2VP30, which contains 14,003 slices.

Except for the PCBT design, the buffers in the designs are implemented using the BRAMs on the FPGA. BRAMs are hardware primitives embedded in the FPGA fabrics. Each Virtex-II Pro BRAM is an 18 Kb true dual-port RAM with two independently controlled synchronous ports that access a common storage area. Since BRAMs do not occupy configurable slices, the number of BRAMS employed does not directly impact the programmable logic slice area of the designs.

As shown in Table III, the PCBT design needs the most adders and uses slice logic for buffers, therefore its area is much larger than the other designs. For the FCBT design, about 40% of the area is used for memory access logic and control logic. Its clock speed is constrained by the clock rate of the adder. The DSA design requires two adders and has an area that is less than the SSA

TABLE III
CHARACTERISTICS OF REDUCTION CIRCUITS ($n = 128$)

Design	Area (Slices)	Clock Speed (MHz)	No. of BRAMs
PCBT	6,808	165	-
FCBT	2,859	170	10
DSA	2,215	142	3
SSA	1,804	165	6

but more than the FCBT. It also only requires three BRAMS. Unfortunately, the combination of the multiple large multiplexers, and complex asynchronous FIFO control logic significantly limited the clock speed of this design.

The SSA design only uses one adder and takes up the smallest area. However, because the algorithm is complex, its control logic occupies more than 50% of its area. Moreover, its clock speed is slightly lower than the FCBT design but significantly higher than the DSA design. On the other hand, since the FCBT design needs to carry the labels and set indices along with the inputs, it needs more BRAMs than the SSA design.

As n increases, the number of adders in the PCBT design increases and the routing complexity increases. Thus, the area and the clock speed of the design vary with n . Although the number of adders is fixed in the FCBT design, the memory address logic becomes more complex for large n . Therefore, the design occupies more slices and the achievable clock speed decreases as n increases.

As shown in Figure 19, as n increases from 2^4 to 2^{20} , the area of the PCBT design increases by 433%, while the area of the FCBT design only increases by about 20%. However, as shown in Figure 20, the degradation in the clock speed of both designs is almost the same. This is because the control logic of the FCBT design is more complex and its complexity increases with n . Nonetheless, the clock speed degradation of the FCBT design is less than 25% as n increases from 2^4 to 2^{20} .

D. Performance Analysis

We now analyze the performance of our proposed designs. Again, the performance of PCBT design is included. Figure 21 shows the total time and the area-time product of our proposed designs for reducing a single set of n inputs when $n = 128$. The total time is measured in μs . We see that although the FCBT design takes longer time to complete the reduction than the PCBT design, it achieves a smaller area-time product because its area is much smaller. Even though the theoretical

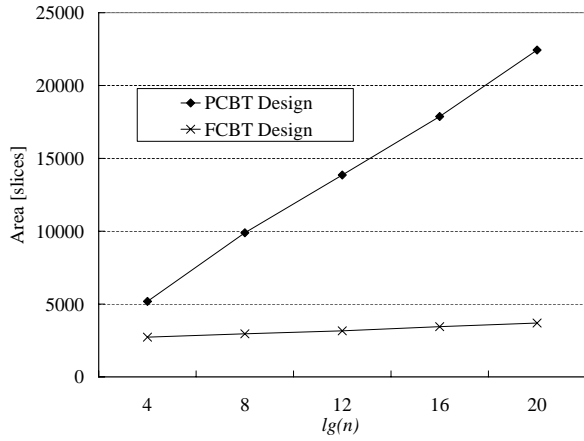


Fig. 19. Area vs. lg(n)

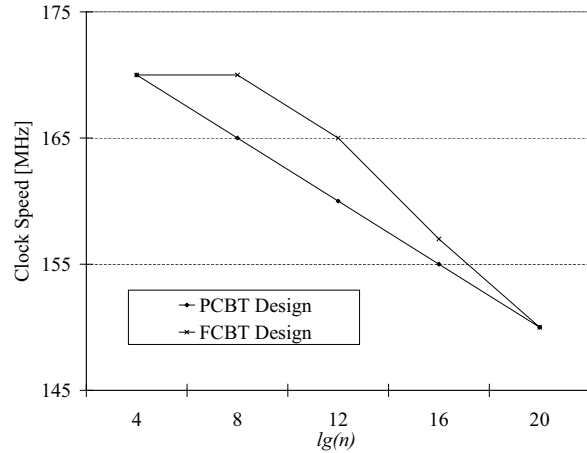


Fig. 20. Clock Speed vs. lg(n)

latency (in cycles) of the DSA design is smaller than the latency of the FCBT, the low clock speed causes the total time to be larger than that of the FCBT. The smaller area of the DSA design causes its area-time product to be a little smaller than the FCBT design. The total time of the SSA design is the longest because it has to reduce the entire Buf_{red} even if the Buf_{red} only contains α items. However, as its area is the smallest, its area-time product is just a little larger than that of the FCBT design.

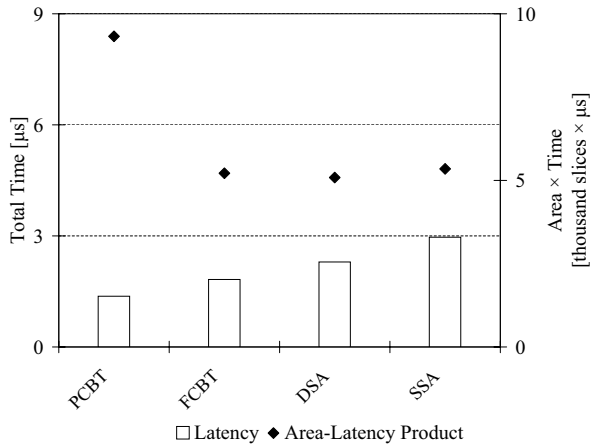


Fig. 21. Latency and area-latency product for reducing a single set of 128 inputs

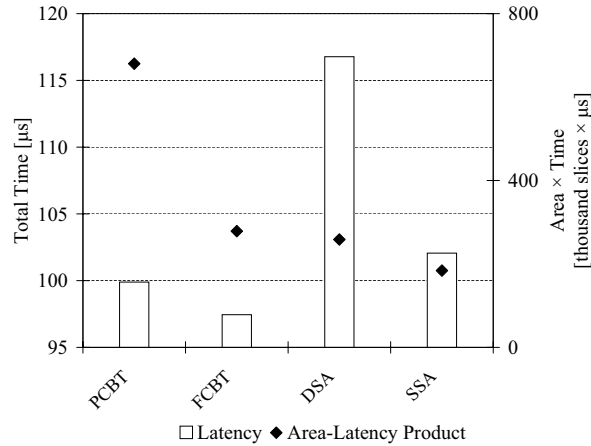


Fig. 22. Latency and area-latency product for reducing 128 sets of 128 inputs

Figure 22 shows the total time and the area-time product of our proposed designs for reducing n sequential sets of n inputs, when $n = 128$. The FCBT design uses less time than the PCBT design and has a smaller area-time product. The much lower clock speed of the DSA design causes the total time of this design to be quite a bit larger than the other designs. The latency of the SSA design is

still the longest, but the SSA design achieves the smallest area-time product thanks to its small area.

The throughput and the throughput-to-area ratio of the proposed designs and the PCBT design are shown in Figure 23. We see that the throughput of the FCBT design is almost the same as that of the PCBT design. Actually, after the first result, both designs generate one result every 128 clock cycles. The difference in their throughputs is caused by their different clock speeds. As expected, the low clock speed of the DSA design seriously impacts its throughput. On the other hand, the SSA design achieves the highest throughput and throughput-to-area ratio. In this experiment, the SSA design generates the first α results after the 256th set are reduced to α values. After that, α results are generated every $128 \times \alpha$ clock cycles.

From the experimental results, we see that our proposed designs are area-efficient and can achieve high performance. In terms of total time, the FCBT design is suitable for reductions of both one set and multiple sets. The low clock speed of the DSA design and its higher area relative to the SSA design make the SSA design a more suitable candidate when multiple sets are reduced. As n increases, the incoming data rate that can be handled by the DSA design and the SSA design remains stable, while the data rate for the FCBT decreases with the clock speed.

E. Applications

In [18], we proposed an FPGA-based design for Sparse Matrix-Vector Multiplication (SpMXV), $y = Ax$. The nonzero elements in each row are partitioned into sub-rows of size k . During each clock cycle, k values in a sub-row enters k multipliers. The resulting k multiply results are reduced by an adder tree. A reduction circuit is needed in the architecture to accumulate the outputs of the adder tree which are generated sequentially.

In the experiments of [18], we used the FCBT design. In [18], the circuit contains 7 floating-point adders, and the total delay of reducing p input sets is $\sum_{i=0}^{p-1} n_i + 7\alpha$. The SpMXV design achieved

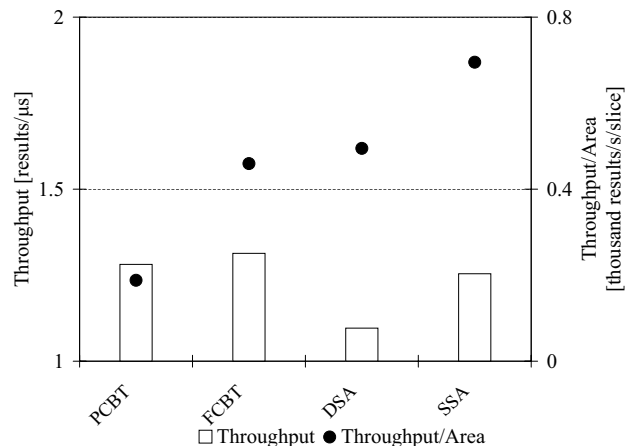


Fig. 23. Throughput and throughput-to-area ratio for reducing 128 sets of 128 inputs

more than 350 MFLOPS for every sparse matrix we tested [18].

With a smaller reduction circuit, the size of the design in [18] will be greatly reduced without any modification to any other part of the design. Suppose the SSA design is used, the total delay for reducing p input sets increases to $\sum_{i=0}^{p-1} n_i + 2\alpha^2$. In practice, α is under 20 and $\sum_{i=0}^{p-1} n_i$ is over tens of thousand. Thus, the SpMXV design achieves almost the same MFLOPS performance with a much smaller area.

VII. CONCLUSION

In this work, we investigated the design of FPGA-based reduction circuits from both the algorithmic and architectural level. Using accumulation reduction as an example, we analyzed the design tradeoffs between the number of floating-point operators, buffer size, and latency. We also looked at the two basic methods for implementing reductions using deeply pipelined operators. Our literature study suggests that this work is the first one that has fully studied the design tradeoffs and fundamental ideas for reduction circuits.

Based on the tradeoff analysis, the FCBT design, the DSA design and the SSA design are proposed. All of these designs are able to reduce multiple input sets of arbitrary sizes without stalling the pipeline. In these designs, the numbers of adders are fixed and the buffer sizes are independent of the number of input sets. We presented the area and clock speeds of the proposed designs and analyzed their performance. Among the designs, the FCBT design is relatively easy to implement. It uses two adders, requires a buffer of size $\Theta(\lg(n))$, and has the longest theoretical latency. The DSA design needs a buffer of size $\Theta(\alpha \lg(\alpha))$ and has the shortest theoretical latency. However, the complex implementation of this design significantly limits its clock speed. The SSA design only employs one adder, but requires the largest buffer size, $\Theta(\alpha^2)$. It has a reasonable theoretical latency for large values of n , and achieves the highest throughput in the experiments. Experimental results show that our designs are area-efficient and can achieve high performance. Sparse matrix-vector multiplication is used as an example to illustrate the application of our designs.

ACKNOWLEDGEMENT

This work was supported by the United States National Science Foundation under grant No. CCR-0311823 and in part by No. ACI-0305763. It was also supported in part by the Department of Defense High Performance Computing Modernization Program.

We would like to thank the anonymous reviewers for their helpful comments.

REFERENCES

- [1] C. Wolinski, F. Trouw, and M. Gokhale, "A Preliminary Study of Molecular Dynamics on Reconfigurable Computers," in *Proc. of The International Conference on Engineering Reconfigurable Systems and Algorithms (ERSA'03)*, Nevada, USA, June 2003.
- [2] K. D. Underwood and K. S. Hemmert, "Closing the Gap: CPU and FPGA Trends in Sustainable Floating-Point BLAS Performance," in *Proc. of 2004 IEEE Symposium on Field-Programmable Custom Computing Machines*, California, USA, April 2004.
- [3] M. Smith, J. Vetter, and S. Alam, "Scientific Computing Beyond CPUs: FPGA Implementations of Common Scientific Kernels," in *Proceedings of 2005 MAPLD International Conference*, Washington, D.C., USA, September 2005.
- [4] Y. Bi, G. Peterson, L. Warren, and R. Harrison, "Hardware acceleration of parallel lagged-fibonacci pseudo random number generation," in *Proc. of Proc. of Engineering of Reconfigurable Systems and Algorithms*, Nevada, USA, June 2006.
- [5] P. Kancharla and D. Buell, "The Advanced Encryption Standard on the HC-36m Reconfigurable Computer," in *Proc. of Military Applications of Programmable Logic Devices*, Washington D.C., USA, September 2003.
- [6] C. Shu, K. Gaj, and T. A. El-Ghazawi, "Low latency elliptic curve cryptography accelerators for nist curves over binary fields," in *Proc. of IEEE International Conference on Field-Programmable Technology*, Singapore, December 2005.
- [7] C. Conger, I. Troxel, D. Espinoza, V. Aggarwal, and A. George, "NARC: Network-Attached Reconfigurable Computing for High-performance, Network-based Applications," in *Proc. of 8th International Conference on Military and Aerospace Programmable Logic Devices*, Washington D.C., USA, September 2005.
- [8] SRC Computers Inc., "MAP[®] Processor," <http://www.srccomp.com/HardwareElements.htm>.
- [9] Cray Inc., "Cray XD1[™]," <http://www.cray.com/products/xd1>.
- [10] R. Scrofano and V. K. Prasanna, "Computing Lennard-Jones Potentials and Forces with Reconfigurable Hardware," in *Proc. of International Conference on Engineering of Reconfigurable Systems and Algorithms*, June 2004, pp. 284–290.
- [11] C.-C. Lam, D. Cociorva, G. Baumgartner, and P. Sadayappan, "Memory-Optimal Evaluation of Expression Trees Involving Large Objects," in *International Conference on High Performance Computing*, Calcutta, India, December 1999.
- [12] D. Bader, S. Sreshta, and N. Weisse-Bernstein, "Evaluating arithmetic expressions using tree contraction: A fast and scalable parallel implementation for symmetric multiprocessors (SMPs)," in *Proceedings of the 9th International Conference on High Performance Computing*, ser. Lecture Notes in Computer Science, vol. 2552, December 2002, pp. 63–75.
- [13] P. M. Kogge, *The Architecture of Pipelined Computers*. Hemisphere Pub. Corp., 1981.
- [14] L. M. Ni and K. Hwang, "Vector Reduction Methods for Arithmetic Pipelines," in *Proc. of the 6th International Symposium on Computer Arithmetic*, June 1983, pp. 144–150.

- [15] Xilinx Incorporated, "<http://www.xilinx.com>."
- [16] K. Underwood, "FPGAs vs. CPUs: Trends in peak floating-point performance," in *FPGA'04: Proceedings of the 2004 ACM/SIGDA Twelfth International Symposium on Field Programmable Gate Arrays*, Monterey, CA, February 2004.
- [17] SGI Inc., "SGI RASC™ Technology," <http://www.sgi.com/products/rasc>.
- [18] L. Zhuo and V. K. Prasanna, "Sparse Matrix-Vector Multiplication on FPGAs," in *Proc. of the 13th ACM International Symposium on Field-Programmable Gate Arrays*, California, USA, February 2005.
- [19] G. R. Morris, R. D. Anderson, and V. K. Prasanna, "A hybrid approach for mapping conjugate gradient onto an FPGA-augmented reconfigurable supercomputer," in *Proceedings of the 14th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06)*, Napa, CA, April 2006.
- [20] X. Wang, S. Braganza, and M. Leeser, "Advanced components in the variable precision floating-point library," in *Proceedings of the 14th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06)*, Napa, CA, April 2006.
- [21] L. Zhuo, G. R. Morris, and V. K. Prasanna, "Designing Scalable FPGA-Based Reduction Circuits Using Pipelined Floating-Point Cores," in *Proc. of the 12th Reconfigurable Architectures Workshop*, Colorado, USA, April 2005.
- [22] G. R. Morris, R. D. Anderson, and V. K. Prasanna, "An FPGA-based application-specific processor for efficient reduction of multiple variable-length floating-point data sets," in *Presented at IEEE 17th International Conference on Application-specific Systems, Architectures and Processors (ASAP'06)*, Steamboat Springs, CO, September 2006.
- [23] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. The MIT Press, 2001.
- [24] Mentor Graphics Corp., "<http://www.mentor.com/>."
- [25] G. Govindu, R. Scrofano, and V. K. Prasanna, "A Library of Parameterizable Floating-Point Cores for FPGAs and Their Application to Scientific Computing," in *Proc. of International Conference on Engineering Reconfigurable Systems and Algorithms*, June 2005.