Modular Design of Fully Pipelined Reduction Circuits on FPGAs

Miaoqing Huang, Member, IEEE, and David Andrews, Senior Member, IEEE

Abstract—Fast and efficient reduction circuits are critical for a broad range of scientific and embedded system applications. High throughput reduction circuits are typically hand designed for specific vector lengths. These circuits need to be modified when the set lengths are changed. In this paper, we present a new design approach that can handle any set length or combination of different consecutive set lengths without stalling and generates in-order results. The flexibility of the design allows it to be used for any reduction operations, such as floating-point addition and multiplication. By providing a simple and efficient interface to the user and a modular architecture for the designer, the proposed technique has a broad impact across a wide range of custom hardware designs.

Index Terms—Fully pipelined reduction circuits, accumulator, field-programmable gate arrays, modular design, digital_circuits

1 INTRODUCTION

Reduction circuits, for example, accumulators, are used in many applications [1], [2] to reduce a vector of items to a single value. A reduction circuit in hardware is typically constructed based on the primitive operator, which performs the same operation on scalar inputs. When building the reduction circuit around the primitive operator, the design is trivial if the latency of the primitive operator is one clock cycle. However, the latency for complicated operations, such as double-precision floating-point addition and multiplication, is generally greater than one clock cycle if high operating frequency is desired. Therefore, many approaches had been proposed to design high-performance reduction circuits [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17]. However, as indicated in Section 2 most designs impose constraints, such as out-of-order output or pipeline stall, which make it inconvenient to adopt them in application design. In this paper, we propose an efficient modular architecture that can be used to construct fully pipelined reduction circuit when the primitive operator itself has a latency greater than one clock cycle. This architecture can handle any set length or combination of different consecutive set lengths without stalling and always generates the results in order.

The proposed architecture is a generic framework within which any reduction operation can be applied. Floating-point operations (e.g., accumulation) are used as examples in our designs due to their great importance within a broad range of scientific applications, such as sparse matrix-vector multiplication (SpMxV) [2]. SpMxV typically involves the multiplication of all the nonzero elements in a matrix by a vector prior to the application of the reduction addition. As the number of nonzero elements cannot be known in advance, the size of the accumulation can vary between rows. It is also important to ensure that data arrive in order for iterative methods. In this paper, we show two designs within our reduction architecture to demonstrate how designers can easily create fully pipelined floating-point accumulators using fully pipelined adders and FIFOs. Each design provides high throughput with the advantage of a standard interface. Both designs allow variable length vectors of operands to be input one item every clock cycle without stalling between the input vectors. Our architecture was not designed to compete directly with designs such as [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17] that seek minimal gate counts. Instead, our designs focus on bringing high performance with new levels of reuse and portability for logic designers. Nevertheless, evaluation results show that our designs occupy less resource than most previous work. These two designs were modeled in Verilog-HDL and verified using a real-life application, i.e., Hessenberg reduction running on an SGI RC100 reconfigurable computer using a Xilinx Virtex-4LX200 FPGA. We resynthesized our designs on other Xilinx and Altera devices targeted by other proposed designs to provide a fair comparative analysis. The results show that our architecture outperforms all previous work by a large margin in terms of both clock frequencies and overall latencies while maintaining portability and reuse.

This work generalizes our previously proposed accumulation circuit in [18] to cover any type of reduction circuit. In addition, a new resource-efficient design is proposed in this work. The remainder of this paper is organized as follows: Related work is briefly discussed in Section 2. The use case scenarios and the interface of our reduction circuit designs are discussed in Section 3. Section 4 discusses the hardware architecture of two fully pipelined accumulators in detail as case studies. The implementation results of a modular multiplication-based reduction circuit are discussed in Section 5. Finally, Section 6 concludes this work.
2 RELATED WORK

Most reduction circuit designs in the literature focus on floating-point accumulators. In [3], Luo and Martonosi proposed a self-alignment technique to improve the performance of floating-point accumulation. However, their design is not fully pipelined, i.e., the accumulator may need to stall internally to deal with overflow. The stall-related logic is very complicated and makes the overall approach difficult to scale. Further, as challenged by He et al. [4], its correctness and accuracy may be questionable. In [5], the self-alignment technique was adapted to implement a single-precision floating-point multiply-accumulator. The authors used a 4:2 compressor to handle the carry-save incoming mantissa fed by a Wallace tree. Paidimarri et al. further extended the work in [5] to a single-cycle single-precision accumulator [6]. He et al. proposed a group-alignment algorithm to design an accurate floating-point accumulator [4]. A pipeline stall is required between the processing of two consecutive data sets, which reduces the throughput of the accumulator and may increase the difficulties for using it in applications. In [7], an application-dependent accumulation circuit was proposed, in which the maximal buffer size is related to the target application. The results may become out-of-order when the circuit is dealing with multiple data sets. Furthermore, the number of inputs in a data set has to be the power of 2. A similar approach to design floating-point accumulator was presented in [8], which also generates out-of-order results. Three architectures, fully compacted binary tree (FCBT), dual strided adder (DSA), and single strided adder (SSA), consisting of adders, buffers, and complex control logics are proposed in [9]. FCBT requires the knowledge of the maximum number of items in a set in advance, which negates the design’s ability to operate in general scenarios. Both DSA and SSA produce out-of-order results when dealing with data sets of variable sizes. The out-of-order output makes it difficult to use them in hardware design. Bodnar et al. [10] demonstrated a variant design of a floating-point accumulator based on work reported in [9]. The design in [10] produces out-of-order results as well. An application-specific and FPGA-specific design of floating-point accumulating circuit is proposed in [11]. As claimed in their work, the parameters of the design need to be tuned for the target application, therefore limiting its broad usability. Sun and Zambreno proposed an architecture [12] where positive and negative operands in a set are summed separately into two intermediate results and then added together. As mentioned by the authors, the accuracy of their approach is a problem; and a sequence that would not overflow if summed sequentially may overflow in their design. Nagar and Bakos [13] attempted to reduce the complexity of control logic circuitry by integrating a coalescing reduction circuit within the low-level design of a base-converting floating-point adder. Unfortunately, the solution is currently incomplete. First, the use of a three-stage reduction circuit is based on the synthesis result across two Xilinx FPGA devices, which negates its ability to be used on other platforms or technologies. Second, a minimum set size is required due to the multiple-stage reduction circuit, making the application range of their solution very limited. In [14], Bachir and David tested various implementation options for the floating-point accumulator architectures proposed in [3], [4], and [11] on a Xilinx Virtex-5 device. A dual-stage accumulator was proposed on BEE3 platform [19] in [15] and [16]. As many previous work, the dual-stage accumulator architecture requires stalls in operation. The built-in floating-point accumulator was used to implement bio-molecular simulations on SRC-7 reconfigurable computer in [20]. However, the exact number of items to be accumulated has to be given while using this accumulator [21]. A floating-point accumulator on Convey HC-1 platform was proposed in [17], which consists of adders, buffers and control logic based on a couple of assumptions such as one input item per clock cycle and out-of-order results.

3 MODULAR REDUCTION CIRCUITS

In this section, we first describe the interface and the application scenario of the modular architecture. Then we present the core idea of the modular approach for designing fully pipelined reduction circuits.

3.1 Interface

In the most general scenario, a reduction circuit reduces arbitrary numbers of items in numerous data sets into single values. Our generic scenario also allows items from the input data sets to be input into the reduction circuit continuously or sporadically. After one data set is finished, the next data set should be allowed to be input into the circuit immediately or after some indefinite time, such as the example shown in Fig. 1a. In any case, the reduction circuit should accept the data as they are presented and produce correct results in the same order.

To meet the above requirements, we design the interface of the reduction circuit as a primitive operator shown in Fig. 1b. The input and output interface of the reduction circuit consist of the following signals:

- **reset**: reset the status of internal control logic and internal registers;
- **operand**: the input data item;
- **op_rdy**: indicate the validity of an input operand;
- **op_last**: indicate the last item in a data set; it should be asserted with the op_rdy signal of the last item;
- **result**: the reduced result of a data set; and
- **result_rdy**: indicate the validity of the result signal.

![Fig. 1. Operating diagram of the proposed fully pipelined reduction circuits.](image-url)
An example diagram demonstrating the sequence among these control signals is given in Fig. 1a. In this simple example, multiple data sets of different sizes are reduced and output by the reduction circuit in order. The result is marked as “ready” on the output after the $\text{op}_\text{last}$ signal is asserted. The latency between the last input item in a data set and the result is not fixed. In other words, the user needs to check the $\text{result}_\text{rdy}$ signal and reads the result when this signal is asserted.

### 3.2 Core Idea

Our proposed reduction circuit is implemented with fully pipelined operators to increase throughput and to achieve high operating frequency. If the latency of the primitive operator is 1 clock cycle, the operator itself is a reduction circuit. However, most advanced operations require a multiple-clock-cycle latency. For example, the latency of the double-precision floating-point adder generated by Xilinx Core Generator 10.1 on Virtex-5 device is 14. Although it is possible to use a single operator to perform reduction, the user would have to wait for $L$ clock cycles before pushing the next item into the operator, where $L$ is the latency of the operator.

A simple solution to build a fully pipelined reduction circuit is to configure multiple operators into a chain as shown in Fig. 2. The chain then feeds the last operator, which reduces the partial results into a single value. By adopting a technique similar to log-sum [22], an $N$-operator chain reduces a block of $2^N$ items into a partial result and lowers the data rate at the same time. If we assume that the original data rate is one item per clock cycle, the data rate drops to one item per $2^N$ clock cycles after the $N$-operator chain. By concatenating $N = \lfloor \log_2 L \rfloor$ operators into a chain and putting one additional operator at the end, we build a reduction circuit that is fully pipelined and is capable of handling an arbitrary length data set. Unfortunately, this simple design is not able to deal with the general case shown in Fig. 1a. Two more capable designs are discussed in the following text, in which we use the accumulation as a case study.

## 4 Fully Pipelined Accumulators

### 4.1 Modular Fully Pipelined Architecture (MFPA)

The architecture of our first fully pipelined accumulator is shown in Fig. 3. It consists of $\lfloor \log_2 L \rfloor + 1$ fully pipelined adders, one FIFO and the associated control logic. The overall architecture is divided into two logical parts: partial sum reduction and accumulation. The first part consists of $\lfloor \log_2 L \rfloor$ adders that reduce the original items in a data set into partial sums. The second part accumulates these partial sums. The constituent adder has an interface similar to Fig. 1b, i.e., two control signals, $\text{op}_\text{rdy}$ and $\text{result}_\text{rdy}$, besides operand and result.

The first $N = \lfloor \log_2 L \rfloor$ adders form a chain that reduces the frequency of the inputs to $A_{\text{acc}}$, the adder after the FIFO. Each adder in the adder-chain takes two inputs and produces one output. For each pair of items, the first item is saved in a register $\text{rin}$ before the arrival of the second item. The status of register $\text{rin}$ is indicated by register $\text{sin}$, whose state transition diagram is illustrated in Fig. 4d. In the normal case, the adder at each level will perform the addition of two items once both become available at the input ports, which is indicated by the $\text{op}_\text{rdy}$ signal. The last item in a set will be added to either the previous item or zero depending on if the last item is even-numbered or odd-numbered. This selection is realized using a 2-to-1 multiplexer with an $\text{op}_\text{select}$ signal. As mentioned before, the $\text{op}_\text{last}$ signal indicates the arrival of the last item in a data set. Internally, this signal will travel down through the shift registers to indicate the last partial sum at each level.
Since $2^N \geq L$, it is guaranteed that the data arrival interval to $A_{acc}$ is greater than or equal to its latency for most of the time. If the number of items in the original data set is $p$, then the adder-chain will reduce the number of items (to be accumulated) to $P = \lceil \frac{p}{2} \rceil$. In other words, the items $\{x_1, x_2, \ldots, x_p\}$ in the original data set are reduced to $\{X_1, X_2, \ldots, X_P\}$ by the adder-chain.

Given a sequence of $P$ items, $\{X_1, X_2, \ldots, X_P\}$, the interval between two items in the first $P - 1$ items is guaranteed to be $2^N$ clock cycles since

$$X_j = \sum_{i=(j-1)2^{N+1}}^{j2^{N+1}} x_i, \quad j = 1, 2, \ldots, P - 1.$$  

However, the last partial sum $X_P$ in the same set may arrive any time between 1 clock cycle and $2^N$ clock cycles after $X_{P-1}$ arrives because the size of the original set is arbitrary and the architecture is not supposed to stall. Assuming the data sets following the current set (being accumulated) are all very short and their sizes are less than $2^N$, then there is only one $X_i$ (i.e., $X_1$) after the adder-chain for these short data sets. If $X_i^{(m)}$ denotes $X_i$ in set $m$, $X_P^{(m)} = X_1^{(m+1)} + X_2^{(m+2)} + \ldots$ may arrive when $A_{acc}$ is performing $X_{P-1}^{(m)} + \sum_{i=1}^{P-2} X_i^{(m)}$. To solve this data hazard and keep the results in order, these data items are saved in the FIFO first. Since the latency of $A_{acc}$ is $L$ clock cycles, at most $L$ storage cells are required to save these data temporarily.

The output of shift register $N$ is concatenated with the result from $Adder N$ and fed into the FIFO. In other words, the data width of the FIFO is $w + 1$-bit, where $w$ is the operand precision of the accumulator. The extra bit is used to identify if the associated partial sum is the last item in a data set. The result_rdy signal of $Adder N$ serves as the $f\_wr\_en$ signal to the FIFO. The FIFO applies a first-word fall-through policy, i.e., the first word is always put at the output port and its validity is indicated by signal $f\_valid$, which is equivalent to empty signal of a traditional FIFO with opposite meaning. The full signal of the FIFO is not used in our architecture because it is guaranteed that the FIFO will never become full in its design.

The $A_{acc}$ performs the computation of $\sum_{i=1}^{P} X_i$. The addition of $X_j + \sum_{i=1}^{j-1} X_i$ ($j = 1, 2, \ldots, P$) is triggered by signal $acc\_op\_rdy$, which serves as the $f\_rd\_en$ signal as well. When there is a valid item at the output port of the FIFO, it has to check whether this item and the item that is previously pushed into the $A_{acc}$ belong to the same data set. This decision is made by checking the flag signal $final\_round$, which indicates the last addition in the accumulation of a data set, i.e., $X_P^{(m)} + \sum_{i=1}^{P-1} X_i^{(m)}$. As mentioned before, the last item in a data set is indicated by an extra bit, $last\_item$, along the data item. The last_item signal will trigger the status change of the $final\_round$ signal, as shown in Fig. 4e. If the previous addition in the pipeline of $A_{acc}$ is not the final addition in a data set, the addition of the current output of the FIFO with $int\_result$ has to wait until $int\_result$ becomes valid, which is indicated by $have\_int\_result$. Otherwise, the current output of the FIFO can be pushed into the $A_{acc}$ right away by adding zero onto it. The status change of $have\_int\_result$ and the value assignment to $int\_result$ are illustrated in Fig. 5. Once the final addition in a data set is pushed into the $A_{acc}$, a $final\_add$ signal is generated at the same moment and pushed into a shift register whose output indicates the availability of the summation of a data set.

## 4.2 Area-Efficient Modular Fully Pipelined Architecture (AeMFPA)

A careful evaluation of the prior MFPA design shows that the $N = \lceil \log_2 L \rceil$ adders are not fully utilized. The utilization pattern of the adders is shown in Fig. 2. The overall utilization of the $N$ adders is given in (1), which is less than 1

$$\sum_{i=1}^{N} \left(\frac{1}{2}\right)^i = 1 - \left(\frac{1}{2}\right)^N < 1. \quad (1)$$

Therefore, it is possible to combine the $N$ adders into a single adder to decrease the transistor count and still provide the same throughput. The work in [9] demonstrates the use of one single adder for accumulation, but it requires complex scheduling control logic and out-of-order output. In our second design, we keep the same simple interface as before, but make the accumulator more area efficient. We call the second design AeMFPA, as shown in Fig. 6.

The second part of AeMFPA is identical to the MFPA design. The difference comes from the first part of the...
AeMFPA design. We use only one single fully pipelined adder to perform the reduction operation. The challenge of this reduction circuit is to guarantee that the overall design remains robust in its ability to deal with a random data input pattern. Particularly, two techniques are applied to realize this objective:

- Put the partial sums of different levels separately in different FIFOs;
- Add by-pass shift registers to solve resource contention regarding the first adder.

As shown in Fig. 2, data reduction requires \( N \) levels, each generating a partial sum. With the exception of the partial sums generated by the last level \( N \), the partial sums at level \( i \) are the inputs of level \( i + 1 \). In MFPA, there is one adder at each level. Therefore, there is no resource contention and the output of an adder can be directly connected to the inputs of the downstream adder. In the AeMFPA design, all levels share the same adder and compete for this sole resource. To offset this contention, data storage is provided to save partial sums before they are fed back into the adder. To keep final results in order and simplify the control logic, the partial sums as well as the original items are put into different pairs of FIFOs corresponding to various levels, as shown in Fig. 6. A pair of FIFOs is used at each level so that two operands can be fetched simultaneously. By using FIFOs, it is sure that the items of different data sets will not mix together and the accumulation circuit always generates in-order results.

Since \( N \) pairs of FIFOs or registers (\( N - 1 \) pairs of FIFOs from level 1 to level \( N - 1 \) and a pair to registers at level \( N \)) are connected to the same Adder, a policy is required to determine which pair of operands should be selected. In our design, we assign increasing priorities from level 1 to level \( N \), with level \( N \) having the highest priority. When multiple pairs of FIFOs have items to be added, the two items in the pair of FIFOs of the highest priority are selected for processing. In this configuration, the pair of FIFOs at level \( N \) only needs to have a depth of 1 since the pair of operands at level \( N \) will be fetched right away once they become available. Therefore, the FIFOs at level \( N \) are replaced by a pair of registers, as shown in Fig. 6. This selection process is implemented by Control C in AeMFPA. All valid signals of the FIFOs and the status of the register pairs at level \( N \) are checked to select the pair of operands for the Adder. Besides the two operands, Control C needs
to generate the following two extra attributes with the outgoing partial sum:

- The level of the outgoing partial sum, i.e., the following level. For example, if the selected pair of operands is from level \(i\), the sum of these two operands needs to be saved in level \(i + 1\). This attribute is pushed into a shift register that is parallel to the Adder. Because there are \(N\) levels, the width of the shift register is \(\lceil\log_2 N\rceil\) bits.
- If two selected operands are the last two items in the current level, the outgoing partial sum will be the last item in the following level. This attribute is pushed into another 1-bit-wide shift register parallel to the Adder. The output of this shift register and the outgoing partial sum will be saved into FIFOs, which are \(w + 1\)-bit wide.

If the size of all data sets is a multiple of \(2^N\), the Adder can never become a structural hazard due to (1). Conversely in the extreme case where a multiple-item set is followed by many single-item sets, these single items need to be pushed down without delay. In the MFPA design, the continuity of single-item sets will cause the utilization of all adders in the adder-chain to be 1. Under this extreme case, the utilization sum of all adders will be \(N\), causing the single Adder to form a structural hazard in the AeMFPA implementation. To solve this resource contention issue, \(N\) layers of shift registers are added within the AeMFPA architecture. At each level, if the last partial sum in a data set is odd-numbered, it is pushed into the corresponding shift register. For example, assuming there are 11 partial sums at level 2, the first 10 partial sums will be added pair by pair through the Adder. The last partial sum will be pushed into the shift register 2 instead.

The valid output of shift register \(i\) (indicated by its most significant bit) and a valid output (indicated by \(\text{add}_\text{rdy}\)) of the Adder for level \(i\) (indicated by \(\text{level}_\text{sel}\)) are put into the pair of FIFOs in a ping-pong fashion. In other words, the partial sums in a data set are put into a FIFO-o and FIFO-e one after the other indicated by Register \(oe\), as shown in Fig. 4c. The very first partial sum in a data set is always put into FIFO-o. Control \(D_1\) is simpler because it only deals with the original items in a data set. Because we use FIFOs to keep all intermediate results in order, the partial sums belonging to the same level always arrive in a sequence. In Fig. 4c, either \(\text{sr}_\text{out}_i\) (the output of shift register \(i\)) or \(\text{add}_\text{out}\) (the output of the Adder) can be valid for level \(i\) at one moment, not both.

Control \(E\) is a simplified version of Control \(D\). Whenever there is either a valid output from shift register \(N\) or a valid output from the Adder for level \(N\), the output is pushed into the FIFO.

### 4.3 Alternative Design of AeMFPA (A^2eMFPA)

In the description of the AeMFPA design, the depth of the \(N - 1\) pairs of FIFOs in the partial sum reduction has not been specified. As the analysis shows in this section, the depth of these FIFOs can be very shallow, which provides various options for implementing these FIFOs. The FIFOs at level \(N\) can be replaced by a pair of registers as this level has the highest priority. For the depth of FIFOs at other levels, it can be analyzed under the worst case scenario. In an extreme case, when there is a pair of items in level \(i\) and all the levels with higher priorities have pairs of items that are ready to be added, the pair of items in level \(i\) needs to wait for \(N - i\) clock cycles before being fetched into the Adder. In this time interval, the highest possible data arrival rate is 1 item each clock cycle when these coming items belong to different single-item sets. For this worst case \(N - i\) cells are required within the FIFO. Besides the additional pair of storage cells, the depth of FIFOs at level \(i\) is

\[
\text{Depth}(i) = N - i + 1, i = 1 \ldots N.
\]

As the depth of the FIFOs in the partial sum reduction is very shallow, these FIFOs can be replaced with registers. This replacement requires modification to the corresponding control logic if the FIFO interface is discarded. Alternatively the FIFO interface can be kept as is to keep the control logic intact when registers are used in the implementation of FIFOs.

#### 4.4 Data Accuracy

Floating-point operation is not associative [23]. The accumulation performed by the proposed architectures is in a order different from the sequential accumulation. Therefore, the result may not be compliant with the result of an accumulator carrying out a sequential addition. In floating point addition, adding a large number with a tiny number will lose precision because the tiny number will be shifted out of the range. For instance, the addition of 1E50 and \(-1\) is still 1E50. For a sequence of input (1E50, \(-E50, 2\), \(-1\)), both the sequential accumulator and the proposed architecture will return the exact result 1. For a different sequence of input (1E50, \(-1\), \(-E50\), 2), the sequential accumulator would return 2, and the proposed architecture would return 0. However, it is almost unavoidable to break the addition sequence in the design of fully pipelined accumulation circuit. In the proposed architecture, the partial sum reduction part reduces every \(2^N\) items in a data set into a partial sum. These partial sums are then added together sequentially.

#### 4.5 Evaluation and Results

##### 4.5.1 Run Time Verification

We first verified the correctness and accuracy of our reduction circuit designs by running a QR eigenvalue algorithm [24], [25] in which the double-precision floating-point accumulator is used in the Hessenberg reduction step. The Hessenberg reduction involves the matrix/vector and vector/vector multiplications of various lengths in arbitrary orders. We implemented and ran Hessenberg reduction on an SGI RC100 reconfigurable computer with our designs implemented as co-processors on a Xilinx Virtex-4LX200 FPGA device. The results of the FPGA accelerated version matched the software implementation on an Itanium 2 1.66-GHz CPU while achieving a \(20\times\) speedup for a \(480 \times 480\) matrix.

##### 4.5.2 Comparative Analysis

A summary comparison of the two proposed modular designs with other previous architectures is provided in Table 1. The AeMFPA design consists of two implementations. The first is a direct implementation of Fig. 6,
Another half day due to the design of 
since it involves internal data distribution and 
The implementation of AeMFPA and 
and the concept behind the logic is very straightforward. 
all the details have been given in this work 
implement an accumulator following the MFPA approach 
reduced design time. It is 
incorporated into systems that rely on in-order results. The 
modules on Altera FPGA device are also reported 
for implementing these FIFOs, we did not intend to 

demonstrate the portability of the proposed architecture. 
the FIFOs were generated using Xilinx Core Generator 10.1 and Altera Megacore 10.0 on 
two platforms respectively. A single adder consists of 
in pipeline stages. The FIFOs were synthesized within 
platform. The data width of a FIFO is 
bits, i.e., 64-bit double-precision floating-point variable 
variable plus 1-bit attribute. The depth of the FIFO within the 
accumulation is 16. In the AeMFPA architecture, six more 
FIFOs are needed in the partial sum reduction part. As 
mentioned in Section 4.3, the required depths of these three 
pairs of FIFOs are 2, 3, and 4, respectively. However, each 
FIFO still needs two banks of BRAM to implement on Xilinx 
FPGAs. On Virtex-II and Virtex-5 devices, the maximum 
data width of a single bank of BRAM is 36 bits. Therefore 
two banks of BRAM are needed to implement the 65-bit-
wide FIFO. Furthermore, the minimum depth of a FIFO is 16 
when Core Generator is used to generate the 65-bit-wide FIFO. 
whereas we ended up using the same \(65 \times 16\) FIFO 
module to implement the seven FIFOs in the AeMFPA 
architecture. In the implementation of \(A^2\)MFPA, we used 
registers to implement the FIFOs in the partial sum reduction 
to allow the other part of the AeMFPA design to be used 
without modification. The resource requirements of the 
primitive adder and three implementations on three FPGA 
devices are listed in Table 2.

Table 3 lists the implementation details of a double-
precision floating-point accumulator using eight different 
designs on Xilinx XC2VP30. The results for PCBT, FCBT, DSA, SSA, and FAAC are referenced from [9] and [12], 
respectively. Our architectures are synthesized by Xilinx 
XST and place-and-routed by Xilinx ISE 10.1 with default 
setting. Apparently, our architecture enjoys a higher 
frequency and a lower latency. Both MPFA and AeMFPA 
operate at frequencies that are 20-46 percent faster than other 
architectures, respectively. The comparatively low 
frequency of our \(A^2\)MPFA implementation on Xilinx 
FPGAs is due to the integration of register-based FIFO into 
the whole design. The critical path of the AeMFPA design is 
within Control C. With the addition of logic-based FIFOs in 
the \(A^2\)MPFA design, the critical path become longer, which 
reduces the frequency of the whole circuit. Since the 
purpose of the \(A^2\)MPFA implementation is to demonstrate 
the tradeoff between the memory usage and logic require-
ment for implementing these FIFOs, we did not intend to

![Table 1](image)

<table>
<thead>
<tr>
<th>Design*</th>
<th>Adders</th>
<th>Buffer Size</th>
<th>Frequency</th>
<th>Total Latency per Set</th>
<th>Latency per Set</th>
<th>In-order</th>
<th>Fully Pipelined</th>
<th>Scalable</th>
<th>Control Logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCBT [9]</td>
<td>(\log_2 p)</td>
<td>(2[\log_2 p])</td>
<td>Decrease with (p)</td>
<td>(p + L[\log_2 p])</td>
<td>Predictable</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Simple</td>
</tr>
<tr>
<td>FCBT [9]</td>
<td>2 (3[\log_2 p])</td>
<td>Decrease with (p)</td>
<td>(&lt;3p + (L - 1)[\log_2 p])</td>
<td>Predictable</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Complex</td>
<td></td>
</tr>
<tr>
<td>DSA [9]</td>
<td>2 (L[\log_2 L + 1])</td>
<td>Stable</td>
<td>(p + L[\log_2 L + 1])</td>
<td>Not Predictable</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Complex</td>
<td></td>
</tr>
<tr>
<td>SSA [9]</td>
<td>1 (2L^2)</td>
<td>Stable</td>
<td>(&lt;p + 2L^2)</td>
<td>Not Predictable</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Complex</td>
<td></td>
</tr>
<tr>
<td>FAAC [12]</td>
<td>(&lt;\log_2 k)</td>
<td>(2k)</td>
<td>Stable</td>
<td>(p + k + L[\log_2 L + 1])</td>
<td>Predictable</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Medium</td>
</tr>
<tr>
<td>MPFA</td>
<td>(\log_2 L + 1)</td>
<td>(L)</td>
<td>Stable</td>
<td>(\leq p + \log_2 L + 2)</td>
<td>Predictable</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Simple</td>
</tr>
<tr>
<td>AeMFPA</td>
<td>(2)</td>
<td>(&lt;2L)</td>
<td>Stable</td>
<td>(\leq p + \log_2 L + 2)</td>
<td>Predictable</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Simple</td>
</tr>
<tr>
<td>(A^2)MFPA</td>
<td>(2)</td>
<td>(L)</td>
<td>Stable</td>
<td>(\leq p + \log_2 L + 2)</td>
<td>Predictable</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Simple</td>
</tr>
</tbody>
</table>

*p: number of items in a set; \(L\): latency of the constituent adder; \(k\): latency of the simplified adder in [12].
design high-speed FIFOs based on registers. Instead, these register-based FIFOs were coded in a very straightforward fashion. Table 4 lists the results of stand-alone implementations for the FIFOs of three different depths. It can be observed that the performance of the register-based FIFOs is highly technology-dependent.

The total latency of our architecture is greater than the one of FAAC in terms of clock cycles. However, both MFPA and AeMFPA outperform FAAC by 13 and 12 percent in terms of absolute latency in $\mu$s. The resource requirements of MFPA, AeMFPA, and $Ae^{2}MFPA$ is only 80, 50, and 60 percent of FAAC, respectively. $Ae^{2}MFPA$ is able to significantly reduce the use of BRAMs compared with AeMFPA at a cost of a slight increase in logic. Overall, our AeMFPA enjoys the smallest area $\times$ latency in all eight designs.

The implementation results on an Altera Stratix-II EP2S180 FPGA device demonstrate the similar pattern regarding the resource utilization. The MFPA occupies the most logic because it uses the most constituent adders. When the FIFOs in the AeMFPA architecture are implemented using registers, the use of registers increases by 26 percent in $Ae^{2}MFPA$ compared with AeMFPA. Different from the Xilinx platform, the register-based FIFOs do not contain the critical path in the $Ae^{2}MFPA$ implementation on Altera platform. Instead the removal of the BRAM from the design increases the overall frequency of the whole design by 18 percent compared with AeMFPA.

5 Reduction Circuit with Multiplication Operation

The two designs proposed in Figs. 3 and 6 can be applied to reduction circuits with other types of operations. In this section, we discuss the required modification and the implementation results of a reduction circuit with double-precision floating-point multiplication.

Two minor changes are required to implement a multiplication-based reduction circuit:

- The constituent operator needs to be replaced by a fully pipelined multiplier.
- Both the reset value of the int_result register and the constant input to the multiplexers need to be "1."

The other parts of the designs can be kept as is. The implementation results on both Xilinx and Altera platforms are shown in Table 5. It can be observed that the implementation results of multiplication-based reduction circuit show the similar pattern as the accumulators. MPFA takes more resource than AeMPFA; however, MPFA generally runs faster than the other one. This similar trend clearly demonstrates the excellent applicability of the proposed approach on different reduction operations across various platforms.

### Table 2

<table>
<thead>
<tr>
<th>Design</th>
<th>Xilinx XC2VP30</th>
<th>XCSVLX110T</th>
<th>Altera EP2S180</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slices</td>
<td>Freq. (MHz)</td>
<td>Slices</td>
<td>Freq. (MHz)</td>
</tr>
<tr>
<td>Adder</td>
<td>528</td>
<td>162</td>
<td>267</td>
</tr>
<tr>
<td>Subtract</td>
<td>1,367</td>
<td>187</td>
<td>-</td>
</tr>
<tr>
<td>Sadder</td>
<td>1,506</td>
<td>176</td>
<td>-</td>
</tr>
<tr>
<td>Sadder</td>
<td>1,798</td>
<td>191</td>
<td>-</td>
</tr>
<tr>
<td>FAAC [12]</td>
<td>6,252</td>
<td>162</td>
<td>2,269</td>
</tr>
</tbody>
</table>

### Table 4

<table>
<thead>
<tr>
<th>F FIFO (w×d)</th>
<th>Registers</th>
<th>Slices</th>
<th>Freq. (MHz)</th>
<th>Registers</th>
<th>Slices</th>
<th>Freq. (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>65×2</td>
<td>132</td>
<td>79</td>
<td>73</td>
<td>304</td>
<td>132</td>
<td>73</td>
</tr>
<tr>
<td>65×3</td>
<td>205</td>
<td>340</td>
<td>211</td>
<td>253</td>
<td>197</td>
<td>206</td>
</tr>
<tr>
<td>65×4</td>
<td>263</td>
<td>350</td>
<td>209</td>
<td>244</td>
<td>263</td>
<td>213</td>
</tr>
</tbody>
</table>

*On Virtex-2, each slice includes two input LUTs and two registers. On Virtex-5, each slice includes four 6-input LUTs and four registers. On Altera Stratix II, an ALUT is a 4-input LUT.

### Table 5

<table>
<thead>
<tr>
<th>Design</th>
<th>Xilinx XC2VP30</th>
<th>XCSVLX110T</th>
<th>Altera EP2S180</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slices</td>
<td>Freq. (MHz)</td>
<td>Slices</td>
<td>Freq. (MHz)</td>
</tr>
<tr>
<td>PCBT [9]</td>
<td>6,808</td>
<td>-</td>
<td>165</td>
</tr>
<tr>
<td>PCBT [9]</td>
<td>2,859</td>
<td>10</td>
<td>170</td>
</tr>
<tr>
<td>DSA [9]</td>
<td>2,215</td>
<td>3</td>
<td>142</td>
</tr>
<tr>
<td>SSA [9]</td>
<td>1,804</td>
<td>6</td>
<td>165</td>
</tr>
<tr>
<td>FAAC [12]</td>
<td>6,522</td>
<td>0</td>
<td>162</td>
</tr>
<tr>
<td>MFPA</td>
<td>5,499 $^2$</td>
<td>207</td>
<td>198</td>
</tr>
<tr>
<td>AeMFPA</td>
<td>3,130 $^{14}$</td>
<td>204</td>
<td>198</td>
</tr>
<tr>
<td>Ae$^{2}$MFPA</td>
<td>3,737 $^{2}$</td>
<td>144</td>
<td>198</td>
</tr>
</tbody>
</table>

$^*p = 128$, $L = 14$, $k = 4$. DSA, SSA, FAAC, MFPA, AeMFPA, and $Ae^{2}$MFPA can deal with arbitrary size of data sets.

$^\dagger$Accumulate a data set of 128 items.

$^\ddagger$All Block RAMs that implement FIFOs are deeply underutilized.
6 Conclusion

In this paper, we propose a novel and modular architecture to construct fully pipelined reduction circuits comprising constituent fully pipelined operators, trivial storage requirement, and simple control logic. Two designs based on this modular architecture are presented. Both designs are capable of performing efficient reduction over data sets of arbitrary sizes without stalling, and generating results in order. Due to the modularity of the design, these two designs can be applied on any reduction operations, including floating-point addition and multiplication. Implementation results of fully pipelined floating-point accumulation circuits on both Xilinx and Altera devices show that both designs have higher frequency and lower latency than all previous proposed accumulators. The implementation results of a fully pipelined floating-point multiplication-based reduction circuit demonstrate the broad applicability of the proposed architecture.

Acknowledgments

The authors would like to thank Liang Men for implementing the fully pipelined reduction circuits of both addition and multiplication-based reduction circuit. The authors would like to thank Liang Men for implementing both designs are capable of performing efficient reduction over data sets of arbitrary sizes without stalling, and generating results in order. Due to the modularity of the design, these two designs can be applied on any reduction operations, including floating-point addition and multiplication. Implementation results of fully pipelined floating-point accumulation circuits on both Xilinx and Altera devices show that both designs have higher frequency and lower latency than all previous proposed accumulators. The implementation results of a fully pipelined floating-point multiplication-based reduction circuit demonstrate the broad applicability of the proposed architecture.

References


Miaoqing Huang received the BS degree in electronics and information systems from Fudan University, China, in 1998, and the PhD degree in computer engineering from The George Washington University, in 2009. He is an assistant professor in the Department of Computer Science and Computer Engineering at the University of Arkansas. His research interests include reconfigurable computing, high-performance computing architectures, cryptography, and cache design in solid-state drives. He is a member of the IEEE.

David Andrews received the PhD degree from Syracuse University, NY, in 1992, where he developed new approaches for performing application specific analysis of parallel computing systems. He is the Mullins Endowed chair of computer engineering at the University of Arkansas. His research interests include the general area of embedded systems architecture. He is a senior member of the IEEE.