Streaming Reduction Circuit

Marco Gerards, Jan Kuper, André Kokkeler, Bert Molenkamp
University of Twente, Department of EEMCS
P.O. Box 217, 7500 AE Enschede, The Netherlands
m.e.t.gerards@ewi.utwente.nl

***Abstract*—Reduction circuits are used to reduce rows of floating point values to single values. Binary floating point operators often have deep pipelines, which may cause hazards when many consecutive rows have to be reduced.**

We present an algorithm by which any number of consecutive rows of arbitrary lengths can be reduced by a pipelined commutative and associative binary operator in an efficient manner. The algorithm is simple to implement, has a low latency, produces results in-order, and requires only small buffers. Besides, it uses only a single pipeline for the involved operation.

The complexity of the algorithm depends on the depth of the pipeline, not on the length of the input rows.

In this paper we discuss an implementation of this algorithm and we prove its correctness.

I. Introduction

The task of reducing a row of numbers to a single result occurs in many applications. For example, in the dot product of two vectors the results of pairwise multiplications have to be added to get the final value. In multiplications of (sparse) matrices several dot products have to be computed, where the vectors may contain different numbers of non-zero values. In applications such as (medical) image processing such matrices can be very large. Hence, multiple sequences of dot products will have to be calculated on a single processor, and thus sequences of rows of numbers have to be reduced to single results.

In practice, one often has to deal with floating point numbers, where addition typically is implemented by a pipeline of intermediate steps. As observed by several authors (see e.g., [1], [2]), this pipeline opens the possibility to exploit task level parallelism. Clearly, the consequence of doing so is a more complicated scheduling of the additions, especially when several input rows of floating point values have to be reduced and numbers from different rows may be present in the pipeline at the same time. Besides, since not all rows will be equally long, the reduction results of the input rows may leave the system in an order which differs from the order in which the input rows enter the system.

In this paper we present an algorithm for the general problem to reduce consecutive rows of values by a pipelined binary operation which is commutative and associative. Every clock cycle one value will enter the system and the binary reduction operation is performed by a single pipelined operator. All values are labeled with an index which indicates to which row a value belongs. After a row is fully reduced and its result has left the system, its label may be reused for a next row. The number of labels needed only depends on the depth of the pipeline, i.e., the number of labels needed is known in advance. Besides, it is relatively small, so the overhead caused by the necessary comparisons of these labels is small.

Intermediate results are stored in an additional buffer. Here too, the size of this buffer depends on the depth of the pipeline, not on the length or number of the input rows. Choosing a buffer of the same size as the pipeline is sufficient to prevent hazards, but may produce results in an order that differs from the order in which rows enter the system. A relatively small enlargement of this buffer is sufficient to guarantee in-order output.

Finally, we remark that our algorithm works for any depth of the operator pipeline.

We implemented the algorithm on an FPGA.

The remaining part of this paper is organized as follows. In section II we discuss related work; in section III we describe the algorithm that realizes our solution to the reduction problem and in section IV we prove that a limited buffer size is sufficient to avoid hazards and buffer overflow. In section V we discuss the labeling and its effects on latency and buffer sizes, in section VI several implementation issues on an FPGA are discussed and results are presented. Finally, in section VII we give some conclusions.

II. Related work

In [3] the method of striping is introduced to avoid the reduction problem for sparse matrix vector multiplications. A drawback of striping is that determining a schedule for multiplication is time consuming. Besides, for stripes that are smaller than the pipeline depth, available processing time is not used and thus wasted.

In [4] the main focus is on sparse matrix vector multiplications on reconfigurable computers. A buffer of α cells is assigned to every row that is present in the pipeline. First, each input row is reduced to α (or less) values, followed by reducing these α values in an adder tree to one result. This design requires a buffer of size $\alpha \times \alpha$, and it needs an adder tree for further reduction, hence the area needed is rather large.

In [2], the authors introduce and compare four different reduction circuits. The Partial Compacted Binary Tree (PCBT) reduction circuit uses a tree of additions. Input values appear at the leaves of the tree and there are n leaves where n is a power of two. The length of an input row is maximally n and shorter rows are padded with zeros. For each level in this tree, one adder is available to add values.

The same paper introduces the Fully Compacted Binary Tree (FCBT) algorithm, which uses two adders. The first adder reduces values at the leaves of the tree, the other adder reduces values at internal nodes. PCBT requires $\lceil \log_2 n \rceil$ adders, while FCBT requires 2 adders but it requires more buffers. This algorithm too places a restriction on the length of input rows.

Again in the same paper two reduction circuits are introduced that do not put restrictions on n. The first, Dual Strided Adder (DSA), uses two adders, where each adder reduces a row of values. The second, Single Strided Adder (SSA), uses just one adder but needs $2\alpha^2$ buffer space and outputs results out-of-order.

In [5] an alternative approach to reduction circuits is described. Instead of scheduling an adder, the adder itself is changed. This approach focuses on an improvement of the numerical accuracy of the result and is not generic in the sense that it is easily applicable to other operators than addition.

On general purpose processors extended with an SIMD instruction set, reduction problems occur according to Corbal et al. [6]. The authors give examples of applications for which reduction problems occur. One of the conclusions is that latency, for SIMD instruction sets, has a significant influence on the performance.

Our approach works for any number of rows of any length, and is applicable to any binary operation that is commutative and associative. It is able to produce results in-order. Our approach is characterized by its use of just one pipelined operator, and the overhead and additional buffer space depends on the depth of the pipeline only and is smaller than in the methods mentioned above. Also the latency of our approach is lower or comparable to the methods above.

III. Algorithm

As mentioned in the introduction, we assume that every clock cycle a value enters the system. These values are grouped in rows, where each row has to be reduced by a given commutative and associative binary operator. For example, all values in a row are numbers which have to be added. Immediately after one row is finished, the next row starts.

Each incoming value is marked by an index that indicates the row to which the value belongs, i.e., values have the same row index if they belong to the same row. When a row is completely reduced and its final result left the system, its row index can be re-used. Hence, the size of row indexes is limited. This size depends on the depth of the operator pipeline, not on the number of input rows.

The system consists of the operator pipeline (denoted as $\mathcal{P}$) and two buffers (see Figure 1): one for buffering the input (denoted as I) and one (denoted as $\mathcal{R}$) for storing the (partial) results of the pipeline. The input buffer I is a modified FIFO which can make two values available instead of one as with a standard FIFO, whereas the result buffer $\mathcal{R}$ is RAM memory. Apart from these buffers and the pipeline there is also a controller in the architecture, described in section VI on implementation.

Now suppose that consecutive rows of values enter the input buffer from the left, one value per clock cycle. With two at a time these values have to be fed into the pipeline from below.

For the result coming out of the pipeline there are the following possibilities: it either is stored in $\mathcal{R}$, or it is combined with a value from I or $\mathcal{R}$ having the same row index, and fed into the pipeline again immediately. Note that there may be clock cycles at which the pipeline will not deliver a result.

We formulate five rules which cover all possible situations to combine values and which determine the next input for the operator pipeline. In these rules we use the following notation: let the depth of the pipeline be α, then $\mathcal{P}_1$ and $\mathcal{P}_\alpha$ denote the entry and exit element of the pipeline, respectively. I_1 is the rightmost element of the input buffer, I_2 is second from the right.

The five rules, in order of priority, are:

1) If there is a value available in $\mathcal{R}$ with the same row index as $\mathcal{P}_\alpha$, then this value from $\mathcal{R}$ will enter the pipeline together with $\mathcal{P}_\alpha$.
2) If I_1 has the same index as $\mathcal{P}_\alpha$, then I_1 and $\mathcal{P}_\alpha$ will enter the pipeline.
3) If there are at least two elements in I, and I_1 and I_2 have the same index, then they will enter the pipeline.
4) If there are at least two elements in I, but I_1 and I_2 have different indexes, then I_1 will enter the pipeline together with the unit element of the operation dealt with by the pipeline (thus for example, 0 in case of addition, 1 in case of multiplication).
5) In case there are less than two elements available in I, no elements will be entered into the pipeline.

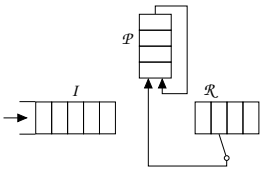
According to these rules, there are situations in which no value from I will enter $\mathcal{P}$. Thus elements will accumulate in the input buffer. In section IV we will prove that size $\alpha+1$, α are sufficient for I, $\mathcal{R}$, respectively.

We remark that the above algorithm does not guarantee that the results will come out of the system in the same order as the rows came in. If the size of $\mathcal{R}$ is enlarged to $2\alpha+\alpha\lceil \log_2 \alpha \rceil + 1$, the output of the system can be produced in-order as will be shown in section V. Note that all buffer

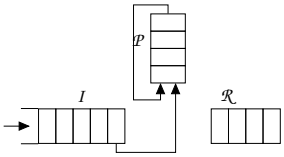
sizes depend on the (fixed) depth of the pipeline, not on the (variable) length of the input rows.

In the beginning of a reduction process, rule 2 and 5 will be used most, whereas later on in the process each rule may be chosen. Clearly, values carrying different row indexes may be present in the pipeline at the same time.

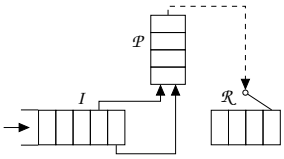
A row is reduced completely when its last value has arrived in $\mathcal{R}$. From there it will be output from the system when its cell in $\mathcal{R}$ is reserved for a next row k. This is done at the moment that the *last* element of row k leaves the input buffer and enters the pipeline. Since $\mathcal{R}$ may contain the final results of more than one row, the choice for which result is output has to be taken with care, to avoid that some value will have to wait indefinitely.



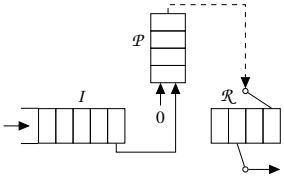
(a) Rule 1



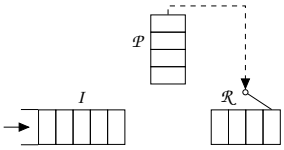
(b) Rule 2



(c) Rule 3



(d) Rule 4



(e) Rule 5

Figure 1. Rules.

In case of rules 3–5 the pipeline result $\mathcal{P}_\alpha$ may or may not exist. If $\mathcal{P}_\alpha$ exists, it will be stored into $\mathcal{R}$ in the cell reserved for the row with the same index as $\mathcal{P}_\alpha$.

In case of rule 4 element I_1 is the last element of a row. Alternatively, rule 4 might also put that element directly into the result buffer such that the rather useless combination with the unit element of the operation involved is not performed. However, the gain of doing this is small, while additional hardware is required and $\mathcal{R}$ then requires an additional write port.

IV. Proof

In this section we prove that it will be sufficient to choose size α for $\mathcal{R}$, and size $\alpha+1$ for I.

At every clock cycle the following events will occur: depending on the situation in the buffers one of the rules is applied, and a new value is stored into the input buffer. As initial state we take the situation after the first value entered the input buffer, but no rule is applied yet. Hence, in all states there will be at least one number in the input buffer I such there will always be at least one rule that can be applied.

A cell in the result buffer $\mathcal{R}$ is called *full* if it contains a value, otherwise it is called *empty*. During the process, cells in the result buffer $\mathcal{R}$ will be assigned to row indexes. The cell in $\mathcal{R}$ that is assigned to k will be denoted as $\mathcal{R}_k$, and it will be used to store values with index k. If a cell in $\mathcal{R}$ is not assigned to a row number, it is called *free*. In the initial situation all cells in $\mathcal{R}$ will be free.

We assume that there will be dummy elements in all cells of $\mathcal{R}$ before the reduction starts, so in the intial state all cells in $\mathcal{R}$ will be full.

During the reduction process, cell $\mathcal{R}_k$ will be set free, if $\mathcal{P}_\alpha$ is the *final* value with index k that is present in the system. Still, $\mathcal{P}_\alpha$ will be stored in $\mathcal{R}_k$ such that free cells too will be full. At the moment that the *last* value of some row k' leaves input buffer I and enters pipeline $\mathcal{P}$, a free cell in $\mathcal{R}$ will be chosen as $\mathcal{R}_{k'}$. At that moment the value present in this cell in $\mathcal{R}$ will be output from the system as the final result of row k.

Since there are α cells in the pipeline $\mathcal{P}$, maximally α cells in the result buffer $\mathcal{R}$ will be assigned to row indexes. At the moment that a cell has to be assigned to a new row index, i.e., at the moment the the last value of that row leaves I and enters $\mathcal{P}$, the first cell of $\mathcal{P}$ will be empty such that at most $\alpha-1$ cells may contain values. Hence, at most $\alpha-1$ cells in $\mathcal{R}$ will be assigned to row indexes. Therefore it is sufficient that $\mathcal{R}$ contains α cells such that at least one cell in $\mathcal{R}$ will be free at each moment that a free cell is needed.

We will denote the number of *values* in $I, \mathcal{P}, \mathcal{R}$ at a certain moment by $N_I, N_\mathcal{P}, N_\mathcal{R}$, respectively.

We need the following lemmas, expressing *invariants* of the process.

Lemma 1.: $N_\mathcal{P} + N_\mathcal{R} \geq \alpha$.

Proof.: If a cell in $\mathcal{R}$ is *not* assigned to an index, it contains a dummy element. If a cell in $\mathcal{R}$ *is* assigned to

some index k and it is empty, then there will be at least one value with index k in $\mathcal{P}$. Hence, $N_{\mathcal{P}} + N_{\mathcal{R}} \geq \alpha$.

Lemma 2.: $N_I + N_{\mathcal{P}} + N_{\mathcal{R}} \leq 2\alpha + 1$.

Proof.: We proceed by induction on the number of states that the process goes through.

Initial state.: In the initial state $\mathcal{P}$ is empty and $\mathcal{R}$ is filled with dummies, thus $N_{\mathcal{P}} = 0$ and $N_{\mathcal{R}} = \alpha$. The result follows immediately.

Induction step.: For the induction step we assume that *before* the application of a rule the lemma holds for N_I, $N_{\mathcal{P}}$, $N_{\mathcal{R}}$. For each rule we show that *after* the application of that rule the lemma holds for the changed numbers of elements N'_I, $N'_{\mathcal{P}}$, $N'_{\mathcal{R}}$ as well. We repeat that the rules are ordered according to their priority.

Rule 1.: Let k be the index of $\mathcal{P}_\alpha$. If Rule 1 is applicable then cell $\mathcal{R}_k$ is full, and the values from $\mathcal{P}_\alpha$ and from $\mathcal{R}_k$ will be inserted into $\mathcal{P}_1$. After the application of this rule, the numbers of elements in the various buffers are as follows:

$$\begin{aligned} N'_I &= N_I + 1, \\ N'_{\mathcal{P}} &= N_{\mathcal{P}} - 1 + 1, \\ N'_{\mathcal{R}} &= N_{\mathcal{R}} - 1. \end{aligned}$$

It is easy to check that the lemma holds for N'_I, $N'_{\mathcal{P}}$, $N'_{\mathcal{R}}$, given that it holds for N_I, $N_{\mathcal{P}}$, $N_{\mathcal{R}}$.

Rule 2.: If Rule 2 is applicable, then the values in $\mathcal{P}_\alpha$ and in I_1 have the same row index k, and these two values will enter $\mathcal{P}_1$. The effect on the numbers of elements in I and $\mathcal{P}$ is as follows:

$$\begin{aligned} N'_I &= N_I - 1 + 1, \\ N'_{\mathcal{P}} &= N_{\mathcal{P}} - 1 + 1. \end{aligned}$$

In case I_1 is the last element of row k, it follows that $N'_{\mathcal{R}} = N_{\mathcal{R}} - 1$, otherwise $N'_{\mathcal{R}} = N_{\mathcal{R}}$. In both cases it is easy to check that the lemma holds for N'_I, $N'_{\mathcal{P}}$, $N'_{\mathcal{R}}$.

Rule 3.: If there are at least two values in I, and the values in I_1 and in I_2 have the same row index, then these two values are removed from I and inserted into $\mathcal{P}_0$.

There may or may not be a value $\mathcal{P}_\alpha$ in the exit cell of $\mathcal{P}$. Rule 1 is not applicable, so if $\mathcal{P}_\alpha$ (with index k) exists, then cell $\mathcal{R}_k$ has to be empty. In that case this causes an increase of $N_{\mathcal{P}}$ with one, whereas $N_{\mathcal{R}}$ will decrease with one.

Besides, value I_2 may or may not be the last element of a row. If it is not the last element, it will not influence the number of elements in $\mathcal{R}$, otherwise this number will be decreased by one.

Again, the truth of the lemma for the changed numbers of elements easily follows.

Rule 4.: If there are at least two values in I, and the row indexes of I_1 and I_2 are different, then I_1 enters the pipeline together with the identity element of the binary operator. Together with the newly entering value into the input, the effect of this is that N_I remains unchanged, whereas $N_{\mathcal{P}}$ increases by one.

In this situation the element in cell I_1 is the last element of its row, so $N_{\mathcal{R}}$ decreases by one.

Finally, as with rule 3 there may or may not be an element in $\mathcal{P}_\alpha$, but as before, for both cases the total effect of this is zero. Hence, for this rule too, the lemma is true.

Rule 5.: Finally, if there are less than two elements in I, it follows that there is exactly one element in I (see above). None of the previous rules is applicable, and no elements will enter the pipeline. As before, there may or may not exist a value $\mathcal{P}_\alpha$.

Sofar, the effect on the total of $N_{\mathcal{P}}$ and $N_{\mathcal{R}}$ is zero. However, N_I increases by one because of the new element that enters the input. Since $N_I = 1$, it follows that $N'_I = 2$.

Since no elements enter the pipeline, cell $\mathcal{P}_1$ will be empty after the application of this rule. Hence $N_{\mathcal{P}} + N_{\mathcal{R}} \leq 2\alpha - 1$. Thus

$$N'_I + N'_{\mathcal{P}} + N'_{\mathcal{R}} \leq 2\alpha + 1.$$

This completes the proof of lemma 2.

An immediate consequence of these two lemma's is the following:

Theorem.: In every state it is the case that $N_I \leq \alpha + 1$.

Hence, size $\alpha + 1$ for the input buffer will be sufficient to avoid buffer overflow.

V. Indexes

In section III it was mentioned that every row of values has a unique row index in the reduction circuit, however no details were given about this. In the application of a matrix vector multiplication, every row of values can get the row number of the matrix row which equals the index in the result vector. For an actual implementation this is not desirable. One disadvantage of using the row number is that it may require many bits, depending on the matrix size. More bits will result in more hardware and might result in a lower clock frequency due to the size of the comparators which have to be used in the reduction circuit. In addition, any fixed choice of bits will result in a restriction of the number of rows that can be reduced by the reduction circuit.

The row index is used to uniquely identify a row in the reduction circuit. This means that the row index can actually be reused after a row is fully reduced. The reduction circuit assigns a row index to every new row that enters the system. Thus values that enter the system have to be marked in such a way that the reduction circuit can determine where a new row starts. As long as this row is being processed by the reduction circuit, this row index can not be reused.

The maximum number of rows in the system determines the size of the result buffer. When the last value of a row has entered the reduction circuit, the clock cycles have to be counted to determine when the row is fully reduced. It was proven in section IV that an input buffer size of α+1 is

sufficient. This implies that, when a value enters the input buffer every clock cycle, each value can remain in the input buffer maximally $\alpha+1$ clock cycles. Trivially, that also holds for the last element of a row. Now assume that the row was long enough to completely fill the operator pipeline. The pipeline and the result buffer together can maximally contain α values of a single row. In α clock cycles, the number of values of this row is halved. In order to fully reduce α values this way, $\lceil log_2 \alpha \rceil$ times α clock cycles have to take place. After this, it takes another α clock cycles before the final result is available.

This means that after the last value of a row enters the reduction circuit, $2\alpha + \alpha\lceil log_2\ \alpha \rceil + 1$ clock cycles have to pass before this row index can be reused. After that, the final result is available in the result buffer. This value will be sent to the result and its row index can be reused. It's possible to cycle over all possible row indexes. When doing this, it is guaranteed that the result of the reduction circuit is in-order and the delay is $2\alpha + \alpha\lceil log_2\ \alpha \rceil + 1$.

In the case that every row has a length of one, which is the worst case, every clock cycle a new row can enter the system. In that case, $2\alpha + \alpha\lceil log_2\ \alpha \rceil + 1$ row indexes are required by the system, which will also be the size of the result buffer. If the minimal length of a row is greater than one, the number of rows simultaneously in the reduction circuit decreases and thus the number of required row indexes decreases. For a relatively low minimal row length, in comparison to the size of the matrix in many applications, this number quickly reduces to one. For example, for $\alpha = 10$ a row will remain in the system for maximally 60 clock cycles after the last value of the row entered the pipeline. Hence, if the minimum length of a row is 60, two row indexes are sufficient to identify all rows in the system.

VI. Implementation

A row index is assigned to new values when they enter the system as was explained in section V. The datapath consists of two BlockRAMs that form the input FIFO, a pipelined adder and two BlockRAMs for the result buffer (see Figure 2). The input of the pipelined adder is selected by two multiplexers which are controlled by the controller. The control lines from the controller to the buffers are not shown in this figure.

The checks for all five rules are processed in parallel inside the controller. If the checks for a rule is passed, a grant signal is generated for this rule. To determine which rule has to be used at the next clock cycle, we use a fixed priority arbiter [7]. The arbiter and the checks for all five rules do not require a lot of hardware. This means that the control for this algorithm is easy and efficient to implement. The buffers and the adder are thus the biggest parts of the reduction circuit. For efficiency reasons, we store the floating point values inside BlockRAMs, while we use logic for the indexes. This increases the speed of our design at the cost of additional logic.

	FP Adder (*in isolation*)	*Reduction Circuit* (*including the adder*)	
		Any n	$n \geq 128$
Slices	n.a.	3556	2587
Clock Freq. (MHz)	280	200	253
BlockRAMs	0	9	9
DSP48	3	3	3
LUTs	1220	2927	1651
FFs	1139	3437	2936

Table I
REDUCTION CIRCUIT ($\alpha = 12$)

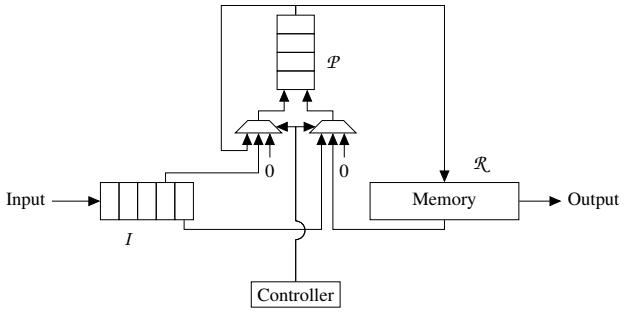


Figure 2. Reduction circuit datapath ($\alpha = 4$)

The implementation of the reduction circuit was tested on the Xilinx Virtex-4 4VLX160FF1513-10. The pipelined binary operator we used is a floating point adder ($\alpha = 12$) which was generated using Xilinx CoreGen. The area and the speed of the adder (in isolation) and of the reduction circuit (including the adder) are shown in table I. If an assumption is made about the minimal length of input rows, the number of bits to identify a row can be reduced. Although this results in a less generic system, the area is decreased and the clock frequency increases as is shown in table I for $n \geq 128$.

Table II shows the results of the implementations of various reduction circuits by Zhuo et al. [2]. These algorithms where implemented on a Xilinx Virtex-II Pro XC2VP7 FPGA. Although this is not the same FPGA as we have used, results are quite similar since most of the logic is used for buffers. These results show that for $n \geq 128$ the area sizes can be compared. They used an adder which is somewhat smaller and less deep that our adder, but also slower. Thus the results are given to give the reader an idea how our implementation compares to theirs, not to draw immediate conclusions from the area the algorithms use.

We would like to emphasize that the buffer sizes required by the PCBT and FCBT algorithms depend on the input, and do not scale for long input rows. The amount of slices or

	PCBT	*FCBT*	*DSA*	*SSA*
Slices	6808	2859	2215	1804
Clock Freq. (MHz)	165	170	142	165
BlockRAMs	0	10	3	6
Adders	$\lceil \log_2 n \rceil$	2	2	1

Table II
REDUCTION CIRCUIT FROM [2] ($n = 128$, $\alpha = 14$)

BlockRAMs increases as the input grows, besides that this places a design time restriction on the input. Compared to our algorithm, the DSA algorithm uses more adders, while it does not produce in-order output. The SSA algorithm does not scale well for deep pipelines and also produces its output out-of-order. In order to compensate for this, the SSA algorithm has to be extended with a reorder buffer and will require more slices than shown in table II. According to Corbal et al. [6], it can be expected that operator pipelines will become deeper thus scalability is an important problem that should be considered when designing reduction circuits.

VII. CONCLUSION

For reduction using a pipelined floating point binary operations, an implementation using one operator is feasible. If an efficient scheduling algorithm is used, a small and fast implementation is possible. The reduction algorithm we proposed uses a single binary operator to reduce an arbitrary number of rows. The rows can be of variable length, while the reduction circuit produces in-order output. To the best of our knowledge, our algorithm is the first algorithm that is capable of doing this. Besides this, the algorithm is short, efficient and is intuitively clear.

An algorithm which is comparable to the proposed algorithm is the SSA algorithm [2]. However, that algorithm is more complex than the algorithm proposed in the present paper, since the required buffer space is an order of magnitude larger and the output is out-of-order. The other algorithms proposed in [2] require multiple operators. When using a reduction circuit, e.g. sparse matrix vector multiplications can be calculated without any complex schedule.

The number of row indexes required to uniquely identify each row is α, though for a fast implementation with in-order output $2\alpha + \alpha\lceil log_2\ \alpha \rceil + 1$ row indexes are required. This has direct influence on the required number of cells in the result buffer, which is $2\alpha + \alpha\lceil log_2\ \alpha \rceil + 1$. The latency (since the last input entered the reduction circuit) introduced by this design is $2\alpha + \alpha\lceil log_2\ \alpha \rceil + 1$ clock cycles.

REFERENCES

[1] M. Bodnar, J. Humphrey, P. Curt, J. Durbano, and D. Prather, "Floating-Point Accumulation Circuit for Matrix Applications," *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06)-Volume 00*, pp. 303–304, 2006.

[2] L. Zhuo, G. Morris, and V. Prasanna, "High-Performance Reduction Circuits Using Deeply Pipelined Operators on FPGAs," *IEEE Trans. Parallel Distrib. Syst.*, pp. 1377–1392, 2007.

[3] R. Melhem, "Parallel solution of linear systems with striped sparse matrices." *Parallel Computing*, vol. 6, no. 2, pp. 165–184, 1988.

[4] G. Morris, R. Anderson, and V. Prasanna, "An FPGA-Based Application-Specific Processor for Efficient Reduction of Multiple Variable-Length Floating-Point Data Sets," *Proceedings of the IEEE 17th International Conference on Application-specific Systems, Architectures and Processors (ASAP'06)-Volume 00*, pp. 323–330, 2006.

[5] C. He, G. Qin, M. Lu, and W. Zhao, "Group-alignment based accurate floating-point summation on fpgas," *ERSA*, pp. 136–142, 2006.

[6] J. Corbal, R. Espasa, and M. Valero, "On the efficiency of reductions in μ-SIMD media extensions," *Parallel Architectures and Compilation Techniques, 2001. Proceedings. 2001 International Conference on*, pp. 83–94, 2001.

[7] M. Weber, "Arbiters: Design ideas and coding styles," *SNUG, Boston*, 2001.