Computer Science
Faculty of EEMCS

Streaming Reduction Circuit for Sparse Matrix Vector Multiplication in FPGAs

Master thesis
August 15, 2008

Supervisor:
dr.ir. A.B.J. Kokkeler

Committee:
dr.ir. A.B.J. Kokkeler
dr.ir. J. Kuper
ir. E. Molenkamp

Marco Gerards
0124699
gerardsmet@student.utwente.nl

Abstract

Floating point sparse matrix vector multiplications (SM×V) are kernel operations for many scientific algorithms. In these algorithms, the SM×V is often responsible for the biggest part of the processing time. It is thus important to speed-up the processing of the SM×V. To use an FPGA to do this is a logical choice since FPGAs are inherently parallel.

The core operation of the SM×V is to reduce arbitrarily many rows of values of arbitrary length to a single value for each row by summing all values within a row. This operation is called a *reduction operation*, the operator that implements this is called a *reduction circuit*. Reduction operations can use any binary operator that is commutative and associative. In the case of a SM×V this is a floating point adder. Because of pipelining of the floating point adder, extra complexity is introduced for reductions. Values need to be buffered and additional control logic is required. Furthermore, a proof is required to show that a certain buffer size is sufficient for every possible input. Important aspects of reduction circuits are thus buffer size, number of operators, latency, in-order output, area and clock speed.

In literature, many reduction circuit algorithms are proposed. However, none of these algorithms have met the design criteria I use in this thesis. Most algorithms either require multiple operators or have buffer sizes that depend on the input. The algorithms that do not have these restrictions have large buffers and deliver output out-of-order.

In this thesis an algorithm is introduced that uses 5 simple rules to check in which order values have to be reduced using a single associative and commutative binary operator. The latency of the reduction circuit is fixed and equals $2\alpha + \alpha\lceil\log_2 \alpha\rceil + 1$ clock cycles, the buffer size is $2\alpha + \alpha\lceil\log_2 \alpha\rceil + 1$ for the output buffer and $\alpha + 1$ for the input buffer. This is an improvement compared to designs described in literature. The buffer sizes and latency decrease if the minimal length of the input rows increases.

The actual implementation is implemented on a Xilinx Virtex-4 4VLX160FF1513-10 FPGA (see appendix A). The total design runs at 200 MHz and consists of 3556 slices, 9 BlockRAMs and 3 DSP48 slices.

Using this reduction circuit, the SM×V implementation is straightforward and requires a multiplier and a reduction circuit. Many of these combinations of a multiplier with a reduction circuit can be implemented in parallel. This results in a lot of processing power with the result that I/O will become the bottleneck.

Contents

Chapter 1

Introduction

The Finite Element Method (FEM) is a frequently used method to approximate the solution of partial differential equations. Because partial differential equations have an infinite dimensional state space, it is hard or impossible to solve these equations analytically.

Using the FEM, only a finite set of elements of the physical problem is considered. For example, when calculating the stress on a building, only certain elements of the building are taken into account. Therefor the problem becomes finitely dimensional. The FEM uses a matrix to describe the elements, this matrix is called the system matrix. This results in a numerically stable method to approximate the solution of the partial differential equation. However, for complex problems, the system matrix will be very big and computationally expensive to solve.

Some examples of problems that can be analyzed using FEM are calculating stresses on constructions (eg. buildings, bridges, etc), car crash simulation and Diffuse Optical Tomography (DOT). This thesis is based on earlier work on the FEM for DOT [21].

Diffuse Optical Tomography is used to reconstruct tissue characteristics. This technique is used in, but not limited to, breast cancer research. Near-infrared light is used to measure optical properties of tissue [7]. Using DOT, all kinds of properties of the tissue can be reconstructed. By using this information, tissue problems can be located and thus diseases can be found.

In the case of DOT, the system matrix is quite big (138,000x138,000) and requires a lot of multiplications. One of the characteristics of that matrix is that is is sparse. A sparse matrix is a matrix which contains more zeros than non-zeros. A key operation of the DOT process is to take the inverse of that matrix. The kernel operation of the matrix inverse is iteratively multiplying

a sparse matrix (filled with double precision floating point numbers) with a vector. The overall DOT process takes about 15 hours on the Graphics Processing Units (GPUs) used in [21]. In that research, one SM×V calculation takes about 4.4 ms. The goal of our research is to bring back this processing time to about 15 minutes, which is an acceptable time for a diagnostic.

To reach this goal, algorithms will be implemented on a Field Programmable Gate Array (FPGA). FPGAs are inherently parallel and offer good performance. Previous work has been done in this area in the form of a master thesis[19]. One of the conclusions was that a partial result adder will be needed for good performance of a sparse matrix vector multiplication (SM×V). A partial result adder can sum series of (floating point) numbers. These rows of floating point values do not need to have the same length, which increases the complexity of the problem (see [10]). The partial result adder is known as a reduction circuit in literature[23]. My goal is to make an efficient implementation of the sparse matrix vector multiplication. One of the key issues is to design and implement an efficient reduction circuit, which is the main subject of this thesis.

In chapter 2, the SM×V and reduction circuits are introduced. It is shown that reduction circuits are important and related work is studied in this chapter. The streaming reduction circuit design and implementation are studied in chapter 3. The results are evaluated in chapter 4. In chapter 5, conclusions are drawn. This thesis is concluded with chapter 6, in which opportunities for future research are discussed.

Chapter 2

Problem Analysis

2.1 Sparse Matrix Vector Multiplication

The implementation of an efficient sparse matrix vector multiplication (SM×V) is the main motivation of this thesis. The SM×V can be implemented in the same way as any other matrix multiplication. However, when doing this, the characteristic properties of the sparse matrix are ignored. A sparse matrix has more zeros than non-zeros, thus most processing time is wasted by multiplying by zero. The matrix in figure 2.1 is an example of a sparse matrix.

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 3 & 6 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 6 & 7 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 7 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 4 & 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 4 & 7 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

Figure 2.1: Sparse Matrix

This particular sparse matrix is dense around the diagonal. Outside a certain distance from the diagonal, all values are 0. For DOT, a sparse matrix with values close to the diagonal is used, so this example is representative. The SM×V multiplies a matrix with a vector. In this simple example, the operations shown in figure 2.2 effectively take place.

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 2 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 3 & 6 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 6 & 7 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 7 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 4 & 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 4 & 7 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} 2 \\ 3 \\ 6 \\ 1 \\ 9 \\ 5 \\ 3 \\ 6 \\ 3 \\ 2 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \\ y_9 \\ y_{10} \end{pmatrix} = \begin{pmatrix} 1 \times 2 \\ 1 \times 2 + 2 \times 3 + 3 \times 6 \\ 5 \times 6 \\ 1 \times 3 + 3 \times 6 + 6 \times 1 \\ 5 \times 6 \\ 6 \times 1 + 7 \times 9 \\ 7 \times 9 + 2 \times 5 \\ 4 \times 5 + 5 \times 3 \\ 4 \times 3 + 7 \times 6 \\ 1 \times 3 + 1 \times 2 \end{pmatrix}$$

Figure 2.2: Sparse Matrix

This illustrates why a lot can be gained by taking special care when executing a SM×V by only considering the non-zero elements. For the non-zeros, multiplications and additions do not have to be calculated.

Often, matrices are stored in a two dimensional array. However, with that representation it is not easy to discriminate between zeros and non-zeros. Most importantly, a zero has to be fetched from memory, before it is possible to determine that this is actually a zero. The memory bandwidth is the most important bottleneck in SM×V implementations [19], thus it should be avoided to read non-zero values.

Because of this, the Compressed Row Storage (CRS) Format is often used. It only stores sequences of non-zero elements where every sequence corresponds to the non-zero elements of a row in the matrix. This sequence of non-zero values that origins from a single matrix row, will be called a row of values in the remainder of this thesis. The matrix is stored in three vectors. The first vector is *val*, which stores the actual floating point values inside the matrix. The second is *col*, this vector stores the corresponding column the value is in. This means that the *val* and *col* vectors form a value-column pair. The elements in the vector *row* denote the positions in *val* and *col* where a new row in the matrix starts. For the previous matrix, the *val*, *row* and *col* vectors are:

val = (1 1 2 3 5 1 3 6 5 6 7 7 2 4 5 4 7 1 1)
col = (1 1 2 3 3 2 3 4 3 4 5 5 6 6 7 7 8 9 10)
row = (1 2 5 6 9 10 12 14 16 18)

When this is compared with the matrix, the *val* vector stores the non-zero values of rows in the matrix from the top row to the bottom row. Every index

in *col* can be matched with a value in *val*. For example, the 9 in *col* means that the corresponding value 1 in *val* is stored in the ninth column. The vector *row* holds the number of preceding values, before the corresponding row of values begins. For example, the value 16 in *row* means that row 8 (16 is the eighth number in *row*) starts with index 16. Thus row 8 starts with the value 4 at column 7, as it can be seen after looking up the value in *val* and column in *col* of the sixteenth value.

In the example, it was shown that the SM×V can be implemented as a series of multiplications. After these multiplications, all values that originate from one matrix row have to be summed. Thus a row of many (n) input values is summed, or *reduced*, to a single value. At the end, these reductions will result in one single value. From now on this step will be called *reduction*. If a row of n values has to be reduced using binary operations, at least $n-1$ operations take place. The reduction of n values can be visualized as a binary tree with n leaves and $n-1$ inner nodes. It is assumed in this thesis that every inner node has exactly two children. Please note that this tree does not have to be balanced, *any* binary tree with n leaves has exactly $n-1$ inner nodes.

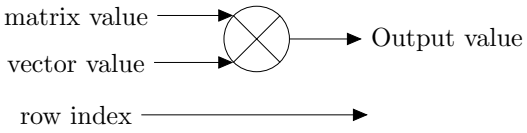


Figure 2.3: Multiplier

In hardware, floating point multiplication can be implemented using a double precision floating point multiplier. The result produced by the multiplier is a double precision floating point value. The delay caused by the pipeline of the multiplier is not important, as this just adds a constant delay to the system, this is illustrated in figure 2.3. To keep track of the rows, a row index is used to uniquely identify each row. The row index corresponds with a row inside the matrix and also indexes into the result vector in which the end result of reduction is placed. Thus the value and the row index form a pair as they traverse the SM×V implementation in the FPGA. In this thesis the value-index pair will also be called a value. It will be apparent from the context if value means a double precision floating point value or the pair that was just described. In figure 2.3, the row index was added explicitly. However, some implementations in literature add the row index implicitly by counting the number of values instead of passing the row index through the system. For example, if it is known at beforehand that a row contains 1000 values, the 5500th value belongs to the fifth row and thus has row index 5.

The multiplications result in a row of values which have to be reduced. This can be illustrated using the multiplication results of the example shown in figure 2.2. The multiplications will produce the following values (floating point value, row index):

(2,1), (2,2), (6,2), (18,2), (30,3), (3,4), (18,4), (6,4), (30,5), (6,6), (63,6), (63,7), (10,7), (20,8), (15,8), (12,9), (42,9), (3,10), (2,10)

Thus the row of values (2,2), (6,2) and (18,2) means that the results of the multiplications for row 2 in figure 2.2 are respectively 2, 6 and 18.

For reduction of these values, additional hardware is required. This hardware is called a *reduction circuit* in literature.

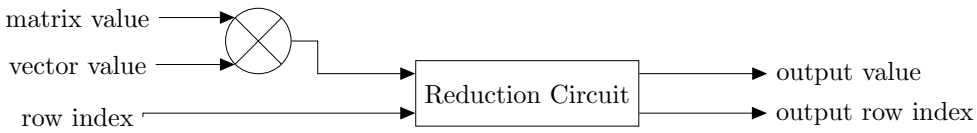


Figure 2.4: Streaming Multiply Accumulate

A streaming reduction circuit can stream in rows of floating point values where the reduction results will appear at the output. The reduction circuit will use a double precision floating point adder. In the remainder of this thesis, a double precision floating point adder is indicated where "adder" is written and the combination of a floating point multiplier with a streaming reduction circuit will be called a Streaming Multiply Accumulate (SMAC), see figure 2.4. The matrix is streamed into the SMAC, one value per clock cycle and after a certain latency the result appears at the output. The SMAC can be used to implement the SM×V efficiently. There are various ways to design a SM×V implementation using floating point adders, floating point multipliers, reduction circuits and SMACs. The area and speed of the reduction circuit should be taken into consideration in the design of the SM×V implementation. If area and speed are known, it is known how many reduction circuits will fit in the FPGA which determines the available parallelism. In section 2.2 the problems that occur when implementing a reduction circuit are discussed.

2.2 Reduction circuits

In section 2.1, the core of the SM×V calculation was brought back to the reduction of rows of values. It was mentioned that a reduction circuit is required for this reduction. However, it was not made clear why a reduction circuit is hard to implement for floating point values (for integer values this reduction is quite trivial as it will be shown below).

First a notation of the row of input values of the reduction circuit will be introduced. Instead of using pairs for the values, a more abstract notation can be used as the floating point value itself is not important for the order of reduction, only the row index influences this order. The row index is added to a value using a subscript. The superscript identifies the position of a value within a row. The partial result of the reduction of the n values $y^1, y^2, \ldots, y^n$ will be written as $y^{1,2,\ldots,n}$.

An example of such row of values is: $y_3^3 y_3^2 y_3^1 y_2^2 y_2^1 y_1^5 y_1^4 y_1^3 y_1^2 y_1^1$ (read from right to left: the right most value, is the first to enter the reduction circuit).

Here, the first row, row 1 has 5 values that have to be added. Row 2 has 2 values and row 3 has 3 values. Rows of arbitrary length should be supported by the reduction circuit. For this example, the reduction circuit will produce 3 results: $y_1^{1,2,3}$, $y_2^{1,2}$ and $y_3^{1,2,3}$.

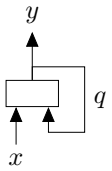


Figure 2.5: Reduction circuit ($\alpha = 1$)

In most integer accumulator designs, an adder without a pipeline is used. The adder consist of combinatorial hardware only, where the result will be available in the same clock cycle as the calculation begun. For reductions, the partial result should be used during the next clock cycle and thus the partial result has to be stored. One register, called an accumulator, is therefor added. This way a small pipeline is created. In this thesis, the depth of the pipeline equals the number of registers and will be designated as α. Thus the integer accumulator design has $\alpha = 1$. In the context of this thesis, the main components of a pipeline are the registers and only these registers will be shown in the figures in this thesis. The reason of not including the logic

Cycle start	*Cycle ready*	*Addition*	
1	2	y_1^1+0	
2	3	$y_1^2+y_1^1$	
3	4	$y_1^3+y_1^{1,2}$	
4	5	$y_1^4+y_1^{1,2,3}$	
5	6	$y_1^5+y_1^{1,2,3,4}$	
6	7	y_2^1+0	The result y_1 is available
7	8	$y_2^2+y_2^1$	
8	9	y_3^1+0	The result y_2 is available
10	11	$y_3^2+y_3^1$	
11	12	$y_3^3+y_3^{1,2}$	

Table 2.1: Possible schedule for an adder with pipeline depth of one ($\alpha = 1$) for input $y_3^3 y_3^2 y_3^1 y_2^2 y_2^1 y_1^5 y_1^4 y_1^3 y_1^2 y_1^1$

between registers is that the delay is analyzed, not the logic of the adder (or any other operator) itself. In figure 2.5 the accumulator design is shown, with its single pipeline register shown as a box.

In figure 2.5, a value x enters the reduction circuit and y leaves the reduction circuit. The example row of values that was introduced at the beginning of this section enters this reduction circuit in sequence, one value every clock cycle. This results in the schedule shown in table 2.1.

The algorithm to schedule a one stage pipeline as shown in figure 2.5 is: if the values q and x have the same row index, these two values have to be reduced. If these two values do not match, y will be the output of the reduction circuit, q will be disconnected during this clock cycle and x will be stored in the accumulator register. This can also be written as an addition of x and 0, like it was done in table 2.1. In pseudo code (executed every clock cycle):

```
if x.index == q.index then
  add x, q
  output nothing
else
  add x, 0
  output y
end if
```

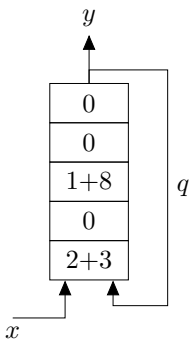


Figure 2.6: Reduction circuit ($\alpha = 5$)

A pipeline depth of one is not realistic when dealing with floating point values. Floating point adders are quite complex compared to integer adders. The floating point adder has to take care of aligning the decimal point of the input values and normalizing the results, among other things. Every subtask of the floating point adder requires one or multiple pipeline stages. When optimizing for speed, pipelining can not be avoided. When dealing with deep pipelines ($\alpha > 1$), the adder schedule is not as trivial as it was in the $\alpha = 1$ case. This results in scheduling complexity and additional buffers or logic. In figure 2.6, a reduction circuit with a pipeline depth of 5 is shown. Assume the values $1, 8, 2, 3$ enter this simplified reduction circuit. As an example, 1 and 8 enter the pipeline during the second clock cycle. The next clock cycle, no pair of values is available and nothing will enter the pipeline. The fourth clock cycle, 2 and 3 enter the pipeline, resulting in what is shown in figure 2.6. Values that enter the pipeline at time t shift through all registers within the pipeline and will eventually leave the pipeline at time $t+5$, or generally $t+\alpha$.

Cycle start	*Cycle ready*	*Addition*	
2	2+5	$y_1^1+y_1^2$	Wait for first two values
4	4+5	$y_1^3+y_1^4$	Next two values available
7	7+5	$y_2^1+y_2^2$	Start with row 2
8	8+5	$y_1^{1,2}+y_1^5$	Add partial results of row 1
9	9+5	$y_3^1+y_3^2$	Start with row 3
13	13+5	$y_1^{1,2,5}+y_1^{3,4}$	Add partial results of row 1
14	14+5	$y_3^{1,2}+y_3^3$	Add partial result of row 3

Table 2.2: Possible schedule for an adder with pipeline depth of 5 (for input $y_3^3y_3^2y_3^1y_2^2y_2^1y_1^5y_1^4y_1^3y_1^2y_1^1$)

Two things should be noticed here. First, in case $\alpha = 5$, there is more freedom compared to the case where $\alpha = 1$. Instead of waiting for two clock cycles, values that appear at x can be added to zero and placed into the pipeline directly. This would result in a design that approximates the accumulator design since only the input is reduced together with values at the output of the reduction circuit. The order and priority of reductions is called the *reduction schedule*, which is one of the main subjects of the next chapter. Because additions are commutative and associative, the order of reduction can be chosen freely. Second, the reader should notice that gaps are formed between values inside the pipeline. The values inside the pipeline have to be further reduced, but partial results do not leave the pipeline every clock cycle. Depending on the reduction schedule, instead of just gaps, values from many rows might appear simultaneously, possibly interleaved, inside the pipeline.

Table 2.2, shows an example of how to reduce the values $y_3^3y_3^2y_3^1y_2^2y_2^1y_1^5y_1^4y_1^3y_1^2y_1^1$ using a pipeline of $\alpha = 5$. At clock cycle 13 $(8 + 5)$, row 1 is still being processed, while the last value of this row entered the reduction circuit at clock cycle 4. Row 2 even finished before row 1 at clock cycle 12 $(7 + 5)$.

At clock cycle 8, value y_1^5 is used. But it was already available at clock cycle 5, so it had to be buffered. The same applied for the output, the result of the addition at clock cycle 4 is available at clock cycle 9. Because it is used at clock cycle 13, it has to be buffered.

In the previous example two design choices were implicitly made. The first and most important choice is that multiple rows can coexist in the adder. Although after reading this example, such an approach might seem logical. In

literature many examples can be found where this is avoided, at all cost. Some of these approaches are discussed in chapter 2.4. The second choice is that for each value, the row index is known. This was introduced in chapter 2.1. However, not all solutions in literature assume that this is the case (see section 2.4). Keeping track of the row index is often required when multiple rows can coexist in the adder.

When a single floating point pipelined adder is used, there will be partial results in the adder pipeline that have to be reduced further after the last value of a row of values enters the reduction circuit. Meanwhile a following row can enter the reduction circuit. This means that either the values that leave the adder have to be buffered for further reduction, and/or the incoming values have to be buffered until they can be processed. The key design issues are:

1. scheduling the adder efficiently
2. buffers should have a finite size

Buffers have to be added to the system. Because partial results are further reduced, the input can temporarily not be processed. It has to be shown that the buffer size for a chosen scheduling algorithm is sufficient for every input sequence (that arrive consecutive and in sequence), especially when the rows do not have a fixed (predetermined) length. Scheduling might become complex, which can have serious impact on the speed of the hardware, the number of buffers required and on the amount of logic required for the design.

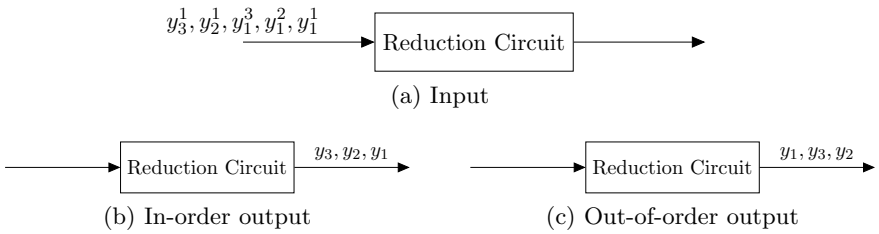


Figure 2.7: Examples of in-order and out-of-order output

Apart from the buffers that are always required, the reduction circuit will have other characteristics as well. The major characteristics are the maximum clock frequency and the area required. Besides that, out-of-order output is produced by some schedules. An out-of-order output means that the reduction result of a row can precede the reduction result of previous row. The schedule in table 2.2 produces out-of-order output which is illustrated in figure 2.7c. Some reduction circuits produce the results in-order (figure 2.7b). Another characteristic of reduction circuits is the delay before the result of a

row is available (counted from the moment on which last value of that row has entered the reduction circuit). This delay can be a fixed value or it might depend on the input. One other characteristic is the number of adders used in a design. These characteristics will be used to compare several existing reduction circuits.

For this project I state the following design goals:

- A reduction circuit clock frequency relatively close to the clock frequency of the adder
- Use a single adder
- The reduction circuit should not be significantly bigger than the adder
- In-order output
- Low delay, independant of the input

Thus reduction circuits can have many characteristics and thus also many shortcomings. For a further (alternative) introduction into reduction circuits, see [23]. The PhD thesis of Gerald R. Morris gives an overview on both SM×V and reduction circuits [10].

2.3 Implementations of SM×V

In literature there are several different approaches for implementing a SM×V on an FPGA. The most important approaches will be summarized here. The two criteria for the choice which algorithms I describe are: (1) can it be used to implement a SM×V for the DOT matrix on a Virtex 4 FPGA and (2) its efficiency, some algorithms do not effectively use all processing power.

2.3.1 Striping

Striping [6] is a method that avoids the reduction problem. A stripe is a sequence of values from the matrix, chosen such that the next value in the stripe is below or to the right and below the current value. Stripes with these characteristics are called Strictly Row Increasing Order (SRIO) stripes. In figure 2.8 two stripes are shown, one in light gray, the other in dark grey. Values from the same row cannot occur in one stripe. The entire Sparse Matrix is divided into such stripes. The SM×V is calculated using several processing elements (PEs). Each PE calculates a stripe, thus the PEs do not have to accumulate values from the same row. Instead of that, a systolic array is used (for a description of systolic arrays, see [6]).

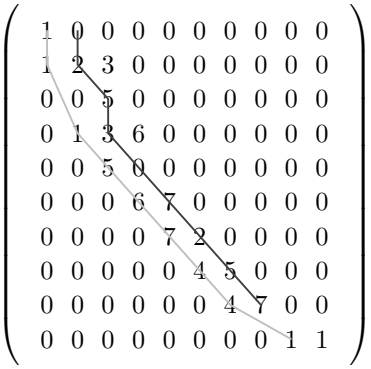


Figure 2.8: Striping

Although the reduction problem does not occur, the memory is read non-linearly, which will result in a degradation of performance for some types of memory. Utilization of PEs can be very low when striping is used depending on the input data according to [19], the utilization is between 3% and 80%.

The main reason of using SRIO stripes is to avoid the reduction problem. In this thesis I would like to show that this problem can be solved efficiently. Later it will be shown that with a straightforward design that uses a reduction circuit, its utilization can be very close to 100% while retaining linear memory access.

2.3.2 Plans

In [19] an alternative scheme is proposed. When striping is used, it is not guaranteed that the PEs have a high efficiency. To increase the efficiency, plans are used to schedule the PEs. A plan is a table filled with multiplications that have to take place in a certain order. The plans can contain an optimal schedule. However, plans require a lot of memory and memory bandwidth and will degrade the performance that way. The memory access is non-linear, which further degrades the performance. Furthermore, determining an efficient schedule can be computationally intensive.

2.3.3 Straightforward approach for implementing the SM×V

Another alternative method mentioned in [19] to simply calculate the SM×V is to use multiple SMACs in parallel. The number of SMACs that can be used is limited by IO and area only. Further studies regarding multiplication order are however still useful to reduce the required bandwidth. The discussion about implementing the SM×V will be deferred to section 4.2.

The advantages of such straightforward implementation are:

- Linear memory access
- Simple or no control logic for the SM×V itself
- The speed is easy to determine
- The utilization is almost 100%
- No pre-processing is required
- Easy to understand and design
- No overhead
- Works for all matrices
- Small buffers ($< \alpha \times \alpha$)
- Works for any row length and any number of input rows
- Scalable: the area increases linearly and the clock frequency decreases linearly as the pipeline depth increases

In [19] a reduction circuit that required eight adders was proposed, which was one of the weak points of this design. However, when a reduction circuit as described in section 2.2 is used, this drawback disappears. Because the weak points disappear when using a reduction circuit, the straightforward approach will be considered again in section 4.2.

2.4 Related work

2.4.1 Floating point adders

Various floating point adders that have been implemented in FPGAs are described in literature. They are hard to compare because the area estimates are given in different units, for information about FPGAs and how to compare logic, see Appendix A. It is also possible to generate floating point adders using Xilinx CoreGen. A list of adders can be found in table 2.3.

The adders generated by Coregen are the fastest of the compared adders. They are not much bigger than the other adders. The fourth adder is smaller, but has a significantly deeper pipeline. The logic to schedule this adder will require more buffering, which might remove this advantage.

With the Xilinx CoreGen software it is very easy to generate a floating point core. Because it is fast, not too big and licenses are already available, this

Source	*FPGA*	*Pipeline stages*	*Speed (MHz)*	*Size*
Paper [4]	Virtex-II 8000-4	9	135.5	1292 LUTs
Paper [9]	XC4036xlahq208-9	5	80	940 CLBs
Paper [9]	XCV100epq240-2	5	150	1059 CLBs
Paper [24]	Virtex-II Pro	21	220	910 LUTs
Paper [22]	Virtex-II Pro	18	200	1140 LUTs
CoreGen 3.0(3x DSP48)	Virtex-4 LX160	12	324	1220 LUTs
CoreGen 3.0(Logic)	Virtex-4 LX160	14	284	1274 LUTs
CoreGen 2.0(Logic)	Virtex-4 LX160	12	271	692 Slices
Xilinx DFPADD	Virtex-4 LX160	6	166	512 Slices

Table 2.3: Floating Point Adders

adder will be used for the design of an SMAC.

The DSP48 enhanced adder will be used since it uses less logic and it is the fastest Coregen adder. For every DSP48 enhanced adder, 3 DSP48 slices are used. Because there are 96 DSP48 Slices available, this means that 32 adders can be added if only DSP48 slices are considered.

2.4.2 Fully Compacted Binary Tree

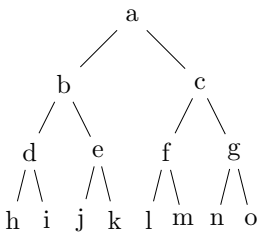


Figure 2.9: FCBT ($n = 8$)

The Fully Compacted Binary Tree algorithm (FCBT) algorithm[23] uses two adders for the reduction of rows of values. As the name of the algorithm suggests, the algorithm works using a binary tree. The binary tree is a complete binary tree of floating point additions. If n values have to be reduced, $n - 1$ additions have to take place. An example of a binary tree of additions is shown in figure 2.9. The values enter the reduction circuit at the leaf level, level 3 in this example. The values at the other levels are partial (intermediate) results that should be further reduced.

Level	*Operations*	*Pace* *(execution every ... clock cycles)*	*Reductions per* 16 clock cycles
3	8	2	8
2	4	4	4
1	2	8	2
0	1	16	1

Table 2.4: FCBT

Since one value enters the reduction circuit at every clock cycle, at most one addition has to take place at every clock cycle. So, in the case of a tree of adders, only one adder is active on *average*, the other adders are not used. The designers of the FCBT algorithm show that every clock cycle, at most two additions take place in this adder tree. This is the reason why they use two adders for this design.

Their algorithm maps the complete tree on two adders. They call this a virtual adder tree. The lowest level (the leaf nodes, in the example this is level 3) is handled by the first adder. Thus every two clock cycles it adds two values producing one result. Thus at the input, only one value is buffered.

The other adder takes care of all other levels in the tree. For each level, a small buffer is reserved. Results from level l are placed into the buffer of level $l - 1$. A counter is used to cycle over all levels in the tree. The level determines how many clock cycles the physical adder is used to reduce values on this level. In the example of figure 2.9, the result of the reduction of values h and i on level 3 are placed in the buffer at level 2.

In table 2.4, the pace of these reductions are shown for the example from figure 2.9. For example, at level 0 only one reduction has to take place every 16 clock cycles. The number of reductions for all levels that are reduced by the second adder is $4 + 2 + 1 = 7$ reductions every 16 clock cycles. The second adder executed the additions for all non-leaf nodes, so one addition is scheduled for each node in the tree.

The algorithm requires a minimal row length of one and the maximum row length has to be known at design time. This maximum length determines the depth of the adder tree. Since the algorithm requires buffers at each level of the tree, the buffers grow as the maximum length of the input rows increases.

Since two adders are used and the sizes of the buffers scale with the length of the input buffers, this design does not meet the design criteria.

2.4.3 Dual Strided Adder

The Dual Strided Adder algorithm (DSA)[23] also uses two adders for reduction. Unlike the FCBT algorithm, the DSA algorithm is independant of the number of input rows and the length of the input rows.

The algorithm uses three buffers. At the input, there is a buffer that stores input values that have to wait because the adders are currently reducing the partial results. There are two buffers for partial results. When a new row of values arrives at the input, one adder starts to reduce it. The other adder is reserved to reduce previous rows that are not fully reduced yet.

Since this design uses two adders, this does not meet the criteria.

2.4.4 Single Strided Adder

The Single Strided Adder (SSA)[23] algorithm is quite similar to the DSA algorithm. The SSA algorithm uses one adder at the cost of increased required buffer size. The algorithm used to schedule the adder is quite complex and I would like to refer to [23] for a detailed description of the algorithm. For this design, the buffers grow quadratically as the depth of the pipeline increases, while the output of the reduction circuit is out-of-order.

2.4.5 Tracking Reduction Circuit

In Bodnar et al. [4], a reduction circuit is described that tracks all rows which are being reduced by the reduction circuit. The number of rows that can maximally coexist in the system is determined by simulation. For each row, buffers are reserved. How the algorithm exactly works is not entirely clear from the paper. It is not even clear whether the algorithm actually works correctly, as the authors do not (formally) proof the correctness of the algorithm and limited simulation results are given.

2.4.6 Adder tree with FIFO

In Morris et al. [11] another approach is discussed for computing sparse matrix vector multiplications, although this research focuses on reconfigurable computers. The reduction circuit uses an $\alpha \times \alpha$ buffer of floating point values. A row in this buffer represents partially reduced values from a single matrix row. Only α buffer rows are required to reduce an arbitrary number of rows of arbitrary size.

When a value enters the system, this value is reduced together with a value from this buffer row. The values in this buffer are initially set to zero. After α clock cycles, the result will be written to this same buffer row. The position

within the row circulates. At the end, the complete row is reduced to maximally α values. To reduce these values, the entire row is fed into an adder tree. The disadvantage of this approach is obvious. In total α adders are required together with a buffer size of $\alpha \times \alpha$.

2.4.7 Group alignment

In He et al. [8], an alternative to reduction circuits is introduced. The main focus of this article is that when floating point arithmetic is used, numerical errors occur. Because of this, precision is lost when it is assumed that floating point additions are commutative and associative. Instead of scheduling operations, the floating point adder is changed such that it accepts a value every clock cycle. Internally fixed point arithmetic is used, thus the full range of floating point values can not be reached. Thus floating point is only at the interface such that it can be a drop in replacement for reduction circuits, the adder is in reality just a fixed point adder. Numerical precision is not the focus of this thesis. For SM×V the full range of double precision floating point values is required.

2.4.8 SIMD MDMX Instruction Set Architecture

In Corbal et al. [5], reductions using SIMD multimedia instructions are discussed. SIMD instruction sets like MMX [15] on the Pentium Processor and MIPS Digital Media eXtension (MDMX) for the MIPS are very popular for multimedia applications. Many multimedia algorithms require reductions. For example, motion estimation requires many accumulations and a minimum operator. For another algorithm, IDCT, many additions are required. An overview of algorithms and the number of reduction operations required can be found in [5]. It is clear that reductions do not only cause design difficulties when using FPGAs, but also when designing GPPs they can be problematic.

MMX instructions store their result in a special SIMD register. The MMX registers do not have many bits, so in case of many multiplications or additions, the register size is not sufficient. In such cases MMX applications use promotion, to move partial results to bigger registers.

However, In MDMX, packed accumulators store their results in an accumulator register which can be used for further reductions. The packed accumulator reduces the accumulator register together with the input. This is mainly done is to avoid data promotion, so more registers are available for parallel processing. The accumulator register has more bits than the other registers. Only partial results at the output of an operator are accumulated. Because of this, the latency is significant.

Algorithm	*Buffer sizes*	*#adders*	*Latency*	*In-order output*
FCBT	$3\lceil log_2 n \rceil$	2	$2n + (\alpha - 1)\lceil log_2 n \rceil$	Yes
DSA	$\alpha\lceil log_n \alpha + 1 \rceil$	2	$\alpha\lceil log_2 \alpha + 1 \rceil$	No
SSA	$2\alpha^2$	1	$2\alpha^2$	No
Tracking	Unknown	1	Unknown	Unknown

Table 2.5: Reduction circuits

2.5 Conclusion

The designs described in section 2.4 either require multiple adders or place a limit of the length on the input rows. The summary of previous work is not complete, a lot of research has been committed in this area, many requiring multiple adders [18, 12, 23]. Other designs have been introduced which have buffer sizes depending on the input [13, 14]. Some solutions can only reduce a single row of values [17]. The design in [4] does not have these limits, but it is not clear how it was implemented. Besides that, this last design doesn't meet the requirements placed on performance. The SSA design is the only design that approximates the design requirements, but still does not meet the demands because the buffer size is $\alpha \times \alpha$ while the output is out-of-order. A summary of the algorithms discussed in this chapter is shown in table 2.5.

In the next chapter an alternative design that does not have the restrictions (low clock frequency, multiple adders, large area, out of order output or a high delay), will be proposed.

Chapter 3

Reduction Circuit

In the previous section, several designs of reduction circuits were discussed. These designs did not meet the design criteria. In this section an alternative design for a reduction circuit is introduced. First the algorithm is stated and its correctness is proven. At the end of this chapter, an implementation is discussed.

3.1 Algorithm

Figure 3.1 shows a reduction circuit with a operator pipeline ($\mathcal{P}$), a buffer ($\mathcal{I}$) at the input and a buffer ($\mathcal{O}$) at the output of the pipeline. Values enter the reduction circuit from the left and are placed in the input buffer. As mentioned in section 2.2, I assume that every clock cycle a value enters the system. The values are grouped in rows, where each row has to be reduced by a given commutative and associative binary operator. Immediately after one row ends, the next row starts. Values are marked by a row discriminator such that it is clear which value belongs to which row. A row discriminator differs from the row index, the row discriminator identifies the row within the reduction circuit, while the row index identifies rows globally in the system. Keeping track of these row discriminators distinguishes this reduction circuit from the reduction circuits that were discussed in section 2.4. Our algorithm is such that only a limited number of row discriminators is sufficient, after a certain number of rows the same row discriminators can be re-used, this is discussed in detail in section 3.3. The size of the row discriminator depends on the depth of the operator pipeline, for a deeper pipeline there are naturally more values that will come out of the pipeline. When these values come out of the pipeline and do not appear at the input anymore, these values have to be reduced while the input is stalled. Because of this, the number of values in the input buffer will increase. This will be further discussed and proven in section 3.2.

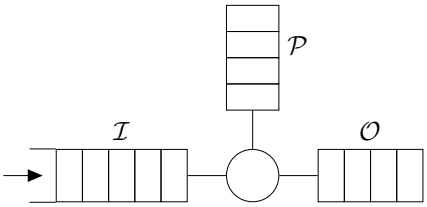


Figure 3.1: Reduction circuit ($\alpha = 4$)

Apart from the operator pipeline (denoted as $\mathcal{P}$) there are two buffers (see Figure 3.1): one for buffering the input (denoted as $\mathcal{I}$) and one for storing the output of the pipeline (denoted as $\mathcal{O}$). The input buffer $\mathcal{I}$ is a FIFO, the output buffer $\mathcal{O}$ is normal RAM memory. In fact, the input buffer is a modified FIFO which can make two values available instead of one. For the output buffer RAM memory is used, since that makes it possible to access values directly.

Apart from these two buffers and the pipeline there also is a controller. However, in this section and the next I abstract away from the controller.

Let us assume that the depth of the pipeline $\mathcal{P}$ is α. In section 3.2 it will be proven that, to prevent hazards and buffer overflow, it is sufficient to a choose size α+1 for the input buffer $\mathcal{I}$, and a size α for the output buffer $\mathcal{O}$.

Clearly, the operator pipeline has to be fed with two values at a time, so at the start of reducing consecutive rows, every two clock cycles the first two values from the input buffer (say x, y) can be entered into the pipeline, if they have the same row discriminator. Suppose the depth of the pipeline is α, then α clock cycles after x and y enter the pipeline, the (partial) result z of x and y leaves the pipeline, where z carries the same row discriminator as x and y. If the value u at the input at the moment z becomes available is marked with the same discriminator, then z and u will be entered into the pipeline, such that only one instead of two values will be taken out of the input buffer.

A second possibility is that the next value u from the input belongs to another row than the output z from the pipeline. In that case the output of the pipeline z is stored in the output buffer, or, in case a value v with the same row discriminator as z is already present in the output buffer, z and v will be entered into the pipeline. Hence, in such situations no value from the input buffer $\mathcal{I}$ will enter the pipeline.

Typically, while a row of floating point values with row discriminator k is still in the process of entering the system, outputs of the pipeline may have the same row discriminator k or not. In case a pipeline output has the same row discriminator, it will be combined with the first value of the input buffer and together they will enter the pipeline. This may occur repeatedly, and values in the input buffer may be "picked up" by the output of the pipeline to enter the pipeline at the beginning.

Likewise, a value v in the output buffer $\mathcal{O}$ will be "picked up" by the output z of $\mathcal{P}$ in case the row discriminators of v and z are the same.

When a row with a given row discriminator has been reduced to a single value with that row discriminator, and no other value with the same row discriminator is present in the system anymore, this value might be released to the outer world as soon as it leaves the pipeline and its row discriminator might be made available for reuse by a next row of input values. However, to simplify the correctness proof somewhat (see section 3.2), we postpone this moment of releasing a final result to the outer world for a while, and store such a value coming from $\mathcal{P}$ into the output buffer $\mathcal{O}$ anyway. From there it will be released when the *last* value of the row being processed (denoted k) leaves the input buffer and enters the pipeline. At that moment the corresponding cell in $\mathcal{O}$ will be "claimed" for values of row k. Since the output buffer may contain the final results of more than one input row, the choice for which final result is released has to be taken with care, to avoid that some value will have to wait indefinitely.

Note that when no value from a given row is present in the system anymore, values for this row will not reappear in the input buffer either. Note further, that values carrying different row discriminators may be present in the pipeline at the same time. Finally, note that there may be clock cycles at which no value is ready to leave the pipeline.

To deal with the possible situations in a precise way, five rules are formulated. In the formulation of these rules, the values of the input buffer whose turn it is to be entered into the pipeline will be denote by $\mathcal{I}_1$, and possibly $\mathcal{I}_2$. The value that leaves the pipeline will be the last one in the pipeline, thus it will be denoted by $\mathcal{P}_\alpha$. As mentioned already, there need not exist a value $\mathcal{P}_\alpha$ at every clock cycle, i.e., cell $\mathcal{P}_\alpha$ may be empty. If $\mathcal{P}_\alpha$ is mentioned in the rules below, it is assumed that it exists.

The rules are given in order of priority, i.e., starting from rule 1 the first rule that is applicable has to be chosen. For a better understanding of the rules we refer to figure 3.2. In all five parts of this figure, three buffers are depicted. The left buffer is the input buffer $\mathcal{I}$, the middle buffer is

the pipeline $\mathcal{P}$, and the right buffer is the output buffer $\mathcal{O}$. The arrows represent the step that the corresponding rule formulates. At each step, the values within the pipeline will move one cell forward, i.e. upward in the picture.

The five rules, in order of priority, are:

1. If there is a value available in $\mathcal{O}$ with the same row discriminator as $\mathcal{P}_\alpha$, then these two values will enter the pipeline.

2. If $\mathcal{I}_1$ has the same row discriminator as $\mathcal{P}_\alpha$, then $\mathcal{I}_1$ and $\mathcal{P}_\alpha$ will enter the pipeline.

3. If there are at least two values in $\mathcal{I}$, and $\mathcal{I}_1$ and $\mathcal{I}_2$ have the same row discriminator, then they will enter the pipeline.

4. If there are at least two values in $\mathcal{I}$, but $\mathcal{I}_1$ and $\mathcal{I}_2$ have different row discriminators, then $\mathcal{I}_1$ 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).

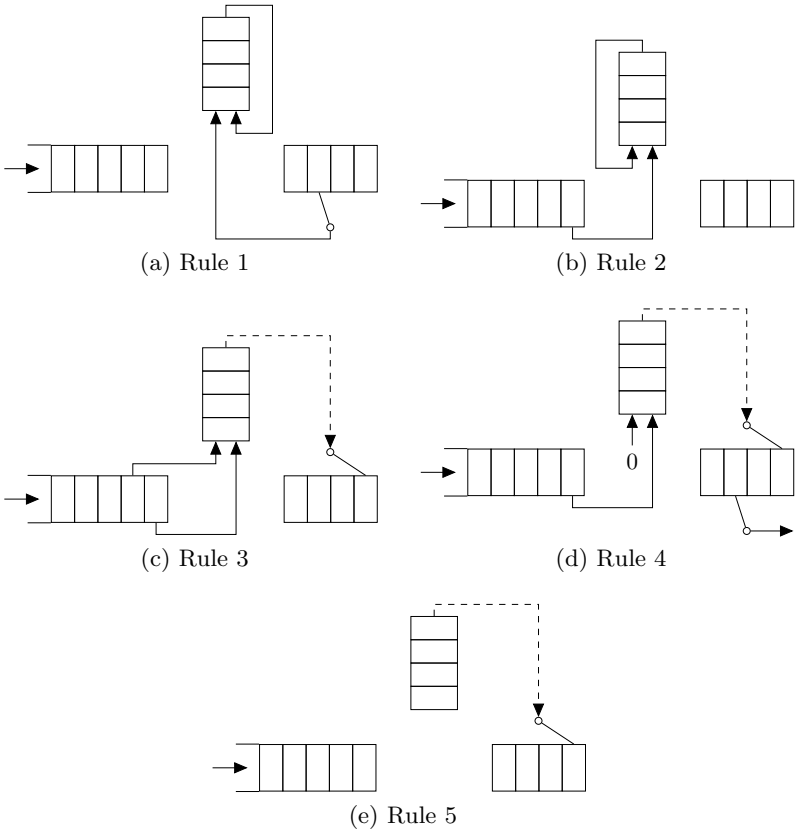
5. In case there are less than two values available in $\mathcal{I}$, no values will be entered into the pipeline.

Note that in case of rules 3–5 it may well be the case that the output $\mathcal{P}_\alpha$ exists. However, in those cases neither rule 1 nor rule 2 is applicable. That is to say, if one of the rules 3–5 is applicable, it still is possible that $\mathcal{P}_\alpha$ will be stored into $\mathcal{O}$ waiting to be picked up, or waiting to be released to the outer world in case it is the final result of its row.

As can be seen in the rules above, there are situations in which no value from $\mathcal{I}$ will enter $\mathcal{P}$. Thus values will accumulate in the input buffer. In the next section it will be shown that sizes α+1 and α are sufficient for $\mathcal{I}$ and $\mathcal{O}$, respectively.

At this point we remark that some optimizations of the above algorithm are possible. First of all, in case of rule 4, value $\mathcal{I}_1$ is a single last value of a row. That value may be put directly into the output buffer such that the rather useless combination with the unit element of the operation involved is not performed.

Secondly, 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. With a limited enlargement of the output buffer $\mathcal{O}$, the results can be released in-order, as will be shown in section 3.3.



(a) Rule 1 (b) Rule 2

(c) Rule 3 (d) Rule 4

(e) Rule 5

Figure 3.2: Rules

3.2 Proof

3.2.1 Definitions

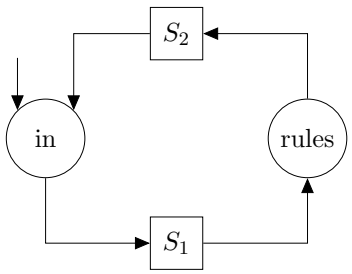


Figure 3.3: States of the reduction circuit used in the proof

When proving the correctness of the reduction circuit design, it is easier to split the design into two states. Figure 3.3 shows these two states as S_1

and S_2. The actions *in* and *rules* occur between the states in the figure. The system begins with the action *in*, which places an input value into $\mathcal{I}$. The number of values contained in $\mathcal{I}$ is notated as $N_{\mathcal{I}}$. Similar notations are used for $\mathcal{P}$ and $\mathcal{O}$ where the number of values they contain is notated as $N_{\mathcal{P}}$ and $N_{\mathcal{O}}$ respectively. Thus, during the *input* step, a value is placed into $\mathcal{I}$ and as a result $N_{\mathcal{I}}$ is increased by one and the system will reach state S_1.

During the transition from S_1 to S_2 the actual rules, as discussed in section 3.1, are executed inside *rules*. After this, the action in *in* take place and the process repeats itself. In an implementation, both actions might be executed simultaneously.

The output buffer has a capacity of α cells. Every cell corresponds to a row discriminator used for a row of values inside $\mathcal{P}$. Rows that occur in $\mathcal{O}$ can not occur in $\mathcal{I}$ anymore. Since $\mathcal{P}$ has α cells, α cells is sufficient for $\mathcal{O}$.

When the last value of a row enters the pipeline (in the context of the proof that follows), the value will receive a row discriminator. Since there are as many cells in $\mathcal{O}$ as there are in $\mathcal{P}$, there is at least one cell with a reduction result, call this cell i. The value just placed in P_1 will receive the row discriminator i (see section 3.1). The following theorem will be used later in this chapter to proof that $\mathcal{O}$ is bounded and $N_{\mathcal{I}} \leq \alpha + 1$.

Theorem 1 *During state S_1, the following statements are always true:*

1. $N_{\mathcal{I}} \geq 1$

2. $N_{\mathcal{P}} + N_{\mathcal{O}} \geq \alpha$

3. $N_{\mathcal{I}} + N_{\mathcal{P}} + N_{\mathcal{O}} \leq 2\alpha + 1$

The induction proof immediately follows in two steps: the initial state and the induction step:

3.2.2 Initial state

To make the proof easier to follow, it will be assumed that $\mathcal{O}$ contains α dummy values. Since these values cannot be used by any rule, they will not influence the algorithm. This means that the proof still holds if the dummies would not be used in practice.

The first statement is true at all times since the input was just placed in $\mathcal{I}$ before these statements are checked. The second statement holds since $N_{\mathcal{P}}$ is

zero and $N_{\mathcal{O}}$ is α. It is easy to see that the third statement also holds. To complete the proof of this theorem, the induction step will be checked next.

3.2.3 Induction step

The induction step shows that given that the statements are true during the state S_1, they also hold the next time this state is reached (S_1'). The number of values $N_{\mathcal{I}}$, $N_{\mathcal{P}}$ and $N_{\mathcal{O}}$ will change during the actions *rules* and *in* (see figure 3.3) before the state S_1' is reached. The number of values in $\mathcal{I}$, $\mathcal{P}$ and $\mathcal{O}$ during the state S_1' will be written as $N_{\mathcal{I}}'$, $N_{\mathcal{P}}'$ and $N_{\mathcal{O}}'$ respectively.

As already mentioned for the initial state, the first statement is always true.

Since $\mathcal{O}$ starts with α dummies ($N_{\mathcal{O}} = \alpha$) of which one is released ($N_{\mathcal{O}}' = N_{\mathcal{O}} - 1$) when a new row enters the pipeline ($N_{\mathcal{P}}' = N_{\mathcal{P}} + 1$), this means the total count ($N_{\mathcal{O}}' + N_{\mathcal{P}}'$) can never get below α, thus the second statement is always true. Or in other words, $N_{\mathcal{P}}$ increases when $N_{\mathcal{O}}$ decreases and $N_{\mathcal{O}}$ will increase again when the result of this row is placed into $\mathcal{O}$.

To show that the induction step holds for the third invariant statement, the state has to be checked for all five rules:

Rule 1
If there is a value available in $\mathcal{O}$ with the same row discriminator as $\mathcal{P}_\alpha$, then these two values will enter the pipeline

This rule uses $\mathcal{P}_\alpha$ and one value from $\mathcal{O}$, thus $N_{\mathcal{P}}$ and $N_{\mathcal{O}}$ both decrease by one. The rule will place one value in P_1, thus $\mathcal{P}$ will increase by one. One value will be placed in $\mathcal{I}$. Thus during S_1', the number of values inside the buffers are:

$N_{\mathcal{I}}' = N_{\mathcal{I}} + 1 = N_{\mathcal{I}} + 1$
$N_{\mathcal{P}}' = N_{\mathcal{P}} + 1 - 1 = N_{\mathcal{P}}$
$N_{\mathcal{O}}' = N_{\mathcal{O}} - 1$

And thus: $N_{\mathcal{I}}' + N_{\mathcal{P}}' + N_{\mathcal{O}}' = N_{\mathcal{I}} + N_{\mathcal{P}} + N_{\mathcal{O}}$

Thus statement three hold for this rule.

Rule 2
If $\mathcal{I}_1$ has the same row discriminator as $\mathcal{P}_\alpha$, then $\mathcal{I}_1$ and $\mathcal{P}_\alpha$ will enter the pipeline

During this rule, one value from $\mathcal{I}$ and $\mathcal{P}_\alpha$ are used, decreasing both $N_\mathcal{I}$ and $N_\mathcal{P}$ by one. During this rule, $\mathcal{O}$ will not change. The number of values in $\mathcal{P}$ will increase by one because of the value placed in $\mathcal{P}_1$. After *rules* took place, *in* will put a value in $\mathcal{I}$ before S_1' is reached. During S_1', the number of values inside the buffers are:

$N_\mathcal{I}' = N_\mathcal{I} + 1 - 1 = N_\mathcal{I}$
$N_\mathcal{P}' = N_\mathcal{P} + 1 - 1 = N_\mathcal{P}$
$N_\mathcal{O}' = N_\mathcal{O}$

And thus: $N_\mathcal{I}' + N_\mathcal{P}' + N_\mathcal{O}' = N_\mathcal{I} + N_\mathcal{P} + N_\mathcal{O}$

If the value from $\mathcal{I}$ was the last of its row, a result from $\mathcal{O}$ is released. In this case, the number of values inside the buffers are:

$N_\mathcal{I}' = N_\mathcal{I} + 1 - 1 = N_\mathcal{I}$
$N_\mathcal{P}' = N_\mathcal{P} + 1 - 1 = N_\mathcal{P}$
$N_\mathcal{O}' = N_\mathcal{O} - 1$

And thus: $N_\mathcal{I}' + N_\mathcal{P}' + N_\mathcal{O}' = N_\mathcal{I} + N_\mathcal{P} + N_\mathcal{O} - 1$

In both cases it is easy to check that statement three still holds.

Rule 3
If there are at least two values in $\mathcal{I}$, and $\mathcal{I}_1$ and $\mathcal{I}_2$ have the same row discriminator, then they will enter the pipeline

Two values from $\mathcal{I}$ are used, thus $N_\mathcal{I}$ decreases by two. After this, the input is placed in $\mathcal{I}$ and $N_\mathcal{I}$ increases by one. Because of the application of this rule $N_\mathcal{P}$ increases by one and $\mathcal{O}$ remains unchanged or decreases by one if the value is released. During state S_1', the number of values inside the buffers are, if no value from $\mathcal{O}$ is released:

$N_\mathcal{I}' = N_\mathcal{I} - 2 + 1 = N_\mathcal{I} - 1$
$N_\mathcal{P}' = N_\mathcal{P} + 1$
$N_\mathcal{O}' = N_\mathcal{O}$ or $N_\mathcal{O}' = N_\mathcal{O} - 1$

And thus: $N_\mathcal{I}' + N_\mathcal{P}' + N_\mathcal{O}' = N_\mathcal{I} + N_\mathcal{P} + N_\mathcal{O}$

If the value from $\mathcal{I}$ was the last of its row, a result from $\mathcal{O}$ is released. In this case, the number of values inside the buffers are:

$N'_{\mathcal{I}} = N_{\mathcal{I}} - 2 + 1 = N_{\mathcal{I}} - 1$
$N'_{\mathcal{P}} = N_{\mathcal{P}} + 1$
$N'_{\mathcal{O}} = N_{\mathcal{O}} - 1$

And thus: $N'_{\mathcal{I}} + N'_{\mathcal{P}} + N'_{\mathcal{O}} = N_{\mathcal{I}} + N_{\mathcal{P}} + N_{\mathcal{O}} - 1$

It is easy to check that statement three holds.

Rule 4
If there are at least two values in $\mathcal{I}$, but $\mathcal{I}_1$ and $\mathcal{I}_2$ have different row discriminators, then $\mathcal{I}_1$ will enter the pipeline together with the unit element of the operation dealt with by the pipeline

One value from $\mathcal{I}$ is used, thus $N_{\mathcal{I}}$ decreases by one. After this, the input is placed in $\mathcal{I}$ and $N_{\mathcal{I}}$ increases by one. If this rule is applicable the value from $\mathcal{I}$ is always the last value of a row. Thus $N_{\mathcal{O}}$ decreases by one since a value is released and one cell becomes free. During the state S'_1, the number of values inside the buffers are:

$N'_{\mathcal{I}} = N_{\mathcal{I}} - 1 + 1 = N_{\mathcal{I}}$
$N'_{\mathcal{P}} = N_{\mathcal{P}} + 1$
$N'_{\mathcal{O}} = N_{\mathcal{O}} - 1$

And thus: $N'_{\mathcal{I}} + N'_{\mathcal{P}} + N'_{\mathcal{O}} = N_{\mathcal{I}} + N_{\mathcal{P}} + N_{\mathcal{O}}$

It is immediately clear invariant statement 3 still holds.

Rule 5
In case there are less than two values available in $\mathcal{I}$, no values will be entered into the pipeline

When this rule is used, $N_{\mathcal{I}} = 1$. If $N_{\mathcal{I}} > 1$, either rule 3 or rule 4 would have been used.

After the application of this rule and placing the input into $\mathcal{I}$, $N'_{\mathcal{I}}$ is 2. Since $N_{\mathcal{P}} < \alpha$ since no value was placed in $\mathcal{P}_1$ during the application of this rule and $N_{\mathcal{O}} \leq \alpha$ it follows that $N_{\mathcal{P}} + N_{\mathcal{O}}$ is maximally $2\alpha - 1$. Thus $N'_{\mathcal{I}} + N'_{\mathcal{P}} + N'_{\mathcal{O}} \leq$

$2\alpha + 1$.
□

3.2.4 Conclusion

Since $N_{\mathcal{I}} + N_{\mathcal{P}} + N_{\mathcal{O}} \leq 2\alpha + 1$ and $N_{\mathcal{P}} + N_{\mathcal{O}} \geq \alpha$, it follows immediatly that $N_{\mathcal{I}} \leq \alpha + 1$. This means that if the input buffer has $\alpha + 1$ cells, the input buffer will never overflow.

It should be noticed that certain assumptions were made about the output buffer. An assumption is made on when values are released and dummies were placed in the output buffer. As noticed before, the dummies do not affect the algorithm, thus they can be left out in the implementation. Furthermore, the output buffer does not affect the algorithm as long as it does not release values too soon. So even for another output buffer implementation than the one assumed in this section, the proof still holds.

3.3 Discriminators

In section 3.1 it was mentioned that every row of values has a row discriminator unique within the reduction circuit. In the application of a matrix vector multiplication, every row of values can get the row index as the row discriminator. For an actual implementation this is not desirable. One disadvantage of using the row index as row discriminator 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. Any fixed choice of number of bits will result in a restriction of the number of rows that can be reduced by the reduction circuit if every row index would be uniquely identified.

The row discriminator is used to uniquely identify a row within the reduction circuit. This means that the row discriminator can actually be reused after the reduction result is released from $\mathcal{O}$. The reduction circuit assigns a row discriminator to every new row that enters the system. Thus values that enter the system have to be marked in such 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 discriminator can not be reused.

The maximum number of rows in the system determines the size of the output 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 3.2 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 α+1 clock cycles. Trivially,

that also holds for the last value of a row. Now assume that the row was long enough to completely fill the operator pipeline. The pipeline and the output buffer together can maximally contain α values of a single row. In α clock cycles, the number of values of this row is halved. To fully reduce α values of the same row in the pipeline, $\lceil \log_2\ \alpha \rceil$ times α clock cycles have to take place. After the last reduction begins, it takes another α clock cycles before the final result leaves the pipeline.

This means that after the last value of a row enters the reduction circuit, at maximum $2\alpha + \alpha\lceil\ log_2\ \alpha\rceil + 1$ clock cycles have to pass before this row discriminator can be reused. After that, the final result is available in the output buffer. This value will be sent to the output and its row discriminator can be reused. It is possible to cycle over all possible row discriminators. When doing this, it is guaranteed that the output of the reduction circuit is in-order and the delay is $2\alpha + \alpha\lceil\log_2\ \alpha\rceil + 1$ counted from the last value of a row that entered the pipeline. The output is in-order since the reduction circuit cycles over all row discriminators and releases results in the same order as they entered the reduction circuit.

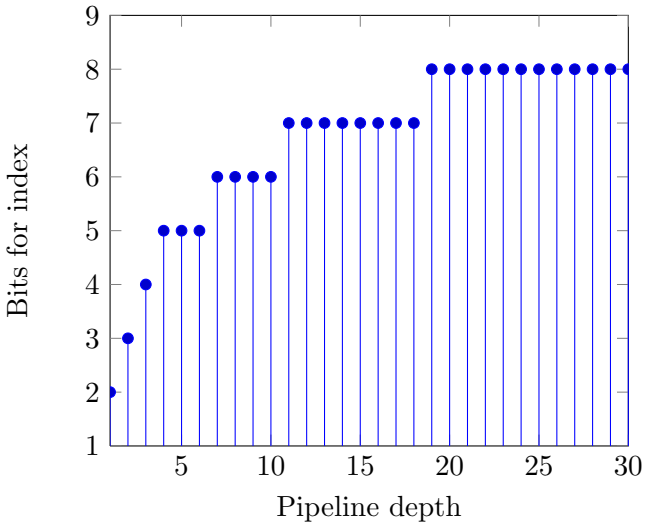


Figure 3.4: Number of bits required for unique identification of rows

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 discriminators are required by the system, which will also be the size of the output buffer. The number of bits required to identify a row directly follows from the number of row discriminators and is shown in figure 3.4. If

value	y_3^1	y_2^1	y_2^2	y_1^4	y_1^3	y_1^2	y_1^1
marker bit	1	0	0	1	1	1	1

Table 3.1: Rows marked using a marker bit

the minimal length of a row is greater than one, the number of rows that are in the reduction circuit at the same time decreases and thus the number of required row discriminators decreases. For large sparse matrices, the minimal row length is often longer than $2\alpha + \alpha\lceil \log_2 \alpha \rceil + 1$ for practical values of α. For example, for $\alpha = 10$ a row will remain maximally 60 clock cycles in the system after the last value of the row entered the reduction circuit. If the minimum length of a row is 60, two row discriminators are sufficient to identify all rows in the system.

3.4 Implementation

The Xilinx Virtex-4 LX Development Kit from AvNET was used for the implementation of the reduction circuit. This development board contains a Xilinx Virtex-4 4VLX160FF1513-10 FPGA. In section 2.4, the choice was made to use a floating point adder generated using the Xilinx CoreGen tool. After the assignment of discriminators is discussed, I will introduce two implementations of the reduction circuit. The first implementation had some drawbacks which I will discuss. The second implementation is the implementation that will be used in the remainder of this thesis.

3.4.1 Assigning discriminators

Floating point values and row discriminators enter the reduction circuit in pairs. First the reduction circuit assigns a row discriminator to each row. This row discriminator requires less bits than the row index that was used for the matrix, which was shown in section 3.3.

An alternative to using a row index to indicate the beginning of a new row, is to add a single bit to each value that enters the system which is flipped whenever a new is started. This is illustrated in table 3.1. In that case the reduction circuit will be able to handle rows of *any* length. Because for the SM×V application preserving row indexes is useful, row indexes are used instead of a marker. Although this alternative is not very useful for the matrix application, however it can be important for other applications that do not want to restrict the number of input rows and thus this method is more generic.

The assignment of discriminators to rows of values can be implemented efficiently. The current implementation detects new rows by comparing the

row index of the current value with the row index of the previous value. In case they do not match, the next row discriminator is determined. In section 3.3 it was mentioned that a fixed number of these row discriminators are required for a given α. If the number of row discriminators is a power of two, operations on row discriminators can be implemented efficiently. Thus the number of row discriminators is rounded upwards to a power of 2.

If b bits are used for row discriminators, a b bits counter can be used to calculate new row discriminators. When a new row is detected, the counter is increased by one. Because the number of row discriminators is rounded upwards to a power of two, a simple counter is sufficient since the counter has to calculate modulo the the number of row discriminators. The new value of the counter is used as the row discriminator for the new row. The row index is stored in a BlockRAM (see A.2), using the row discriminator as an index into $\mathcal{O}$.

For releasing values from $\mathcal{O}$ there are two alternative methods. The first method uses new rows at the input to push out the results of previous rows. If the row is fully reduced, the final result is held in the output buffer, waiting for a signal that it can be released. The moment a row discriminator is assigned, the row to which this row discriminator previously belonged is ready and can be released. Although this might seem awkward at first, this is true by definition, the number of row discriminators that are used were carefully chosen such that this holds. The row index of this value is read from the BlockRAM. The value is released from the output buffer and will appear at the output of the reduction circuit together with its row index.

When a row is fully reduced, the row will not be released before another row gets this same row discriminator. However, it might take a lot of time before this happens. The disadvantage of this approach is that the latency of the system is dependent on future rows. For example, if there are 128 row discriminators, a result will not leave the reduction circuit before 128 rows have entered the reduction circuit. The second method to release values solves this disadvantage at the cost of more area. A value is released $2\alpha+\alpha\lceil\log_2\alpha\rceil+1$ clock cycles after the last value of the row entered the reduction circuit. Thus this signal should be delayed for $2\alpha + \alpha\lceil\log_2\alpha\rceil + 1$ clock cycles. This can be efficiently implemented using a shift register.

3.4.2 First implementation

Two implementations of the reduction circuit were tested. The first implementation did not separate the floating point values from the row discriminators. The input FIFO was made using BlockRAM. Both the data for control (row discriminators) and the data for the datapath (double precision floating point

values) are stored in this FIFO. This approach is logical from the perspective of the data. Data and row discriminators that belong together are stored and processed together.

The longest path determines the maximum clock frequency. In this implementation, the longest path can be found in the controller. This longest path starts at the input FIFO BlockRAMs, which have a long delay. This results in a decreased performance. Furthermore, since the values for the datapath and the row indexes for control are not separated, it might become harder for the synthesis and Place and Route (PAR) tools to make optimizations.

3.4.3 Second implementation

Instead of considering a value as a single entity, it is also possible to look at values as two separate streams of data. The row discriminators go into the controller and the floating point values enter the datapath. This means that there are two FIFOs, one for row discriminators and one for values.

In this implementation the row discriminators are stored in logic instead of BlockRAMs. The controller is a separate entity that has a FIFO for row discriminators which are used by the algorithm and sets the control signals of the datapath. This has several advantages. The longest path will no longer begin with a BlockRAM read, which reduces the initial delay. Furthermore, this approach makes it easier for the synthesis and PAR tools to optimize the design because the row discriminators and values are routed and processed separately.
This design is shown in figure 3.5. The datapath consists of two BlockRAMs (since 3 ports are required) that form the input FIFO, a double precision floating point pipelined adder and two BlockRAMs for the output buffer (two ports are required). 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.

3.4.4 Controller triplication

To optimize the implementation for speed, the longest path has to be analyzed. To get a good design, paths have to be balanced. This means that other paths have a delay similar to the longest path. Thus if the current longest path is removed from the implementation, the new longest path will have a comparable delay and thus it will be hard to optimize this system by analyzing a single longest path.

For the first implementation, the longest path was in the controller. One method to decrease the delay of the path is by inserting a register in the

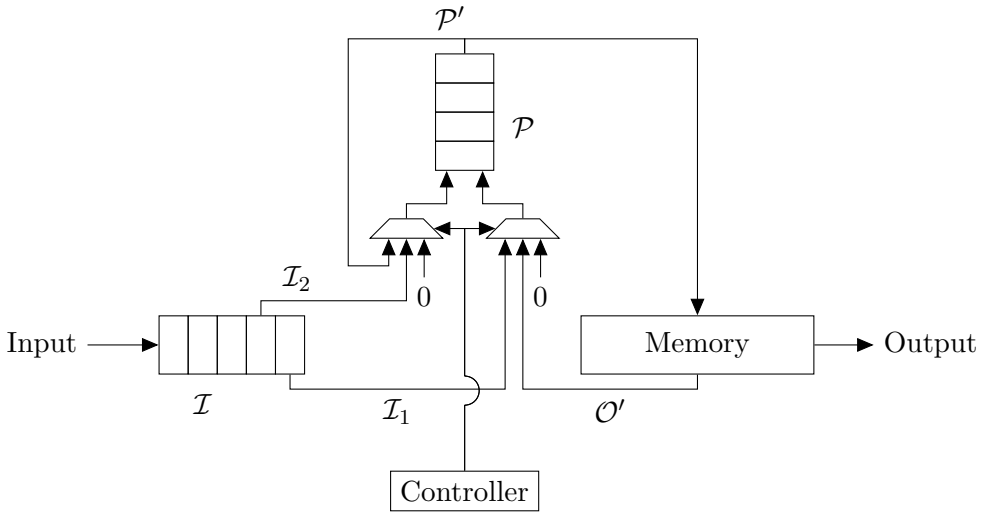


Figure 3.5: Reduction circuit datapath ($\alpha = 4$)

middle of this longest path. An easy way to do this is simply placing a register somewhere in the path, the synthesis tool can use register retiming to balance the path. However, adding a register will result in an undesirable delay since the control signal is available one clock cycle late.

After inserting this register, the control signals are available one clock cycle too late. Thus the controller should determine the control signals of the *next* clock cycle to compensate for this delay. However, not all information for the controller is available during the previous clock cycle. The controller has to know which values from $\mathcal{I}$ are used. Thus it should know which values will be present in $\mathcal{I}$ during the next clock cycle. The solution is to require that always four values are present in $\mathcal{I}$. At clock cycle t, the controller determines the control signals for clock cycle $t + 1$. Obviously, the control signals for t were calculated on clock cycle $t - 1$. Most notably is the control signal *icnt*, the number of values removed from $\mathcal{I}$ during the current clock cycle. This is also shown in figure 3.6.

An example of a possible scenario is that at time t, rule 3 (I_1 and I_2 are used) is used and thus two values from $\mathcal{I}$ will be consumed (and thus $icnt = 2$). These values will no longer be contained in $\mathcal{I}$ at clock cycle $t + 1$. So if a prediction at time t was made that rule 3 will have to be used at $t + 1$ by looking at $\mathcal{I}$, the wrong rule might be used. Perhaps, at time $t + 1$, the values $\mathcal{I}_1$ and $\mathcal{I}_2$ might not match. Generally, three situations can be identified, depending on *icnt*:

icnt = 0 The controller can precisely determine the control signals for the next clock cycle. The input buffer will not change, so the current state will be the same at the next clock cycle.

icnt = 1 The controller will not give a good result if $\mathcal{I}_1$ and $\mathcal{I}_2$ will be used, since one value ($\mathcal{I}_1$) will be consumed during the current clock cycle. At the next clock cycle, the values in $\mathcal{I}$ will be shifted by one. Thus $\mathcal{I}_2$ and $\mathcal{I}_3$ should be used as input for the controller instead of $\mathcal{I}_1$ and $\mathcal{I}_2$. In this case, the result will be correct.

icnt = 2 Just like for $icnt = 1$, other inputs have to be used, since at the current clock cycle two values will be consumed ($\mathcal{I}_1$ and $\mathcal{I}_2$). Now $\mathcal{I}_3$ and $\mathcal{I}_4$ have to be used as input for the controller, since these correspond to $\mathcal{I}_1$ and $\mathcal{I}_2$ during the next clock cycle.

If *icnt* would be used to select the input of the controller, the longest path will be extended further. This is not desirable. Instead, the controller can be triplicated, such that at clock cycle t, three possible sets of control signals will be available. At clock cycle $t + 1$, the value of *icnt* is used as a multiplexer input to select the correct control signals. This multiplexer is located outside the controller separated by a register and does not add to the length of the longest path in the controller. This design is shown in figure 3.6.

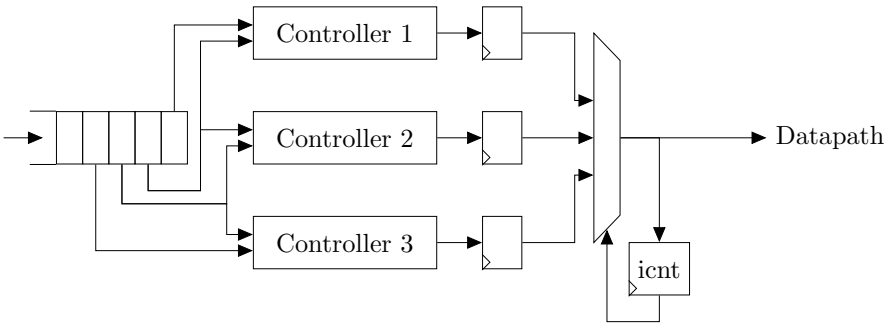


Figure 3.6: Reduction circuit controller (triplicated) ($\alpha = 4$)

This approach was only used in the first implementation. In the second implementation another optimization was used that will be described next. However, controller triplication might be usable in the second implementation to reduce the length of the longest path. However, due to time constraints this option was not explored.

3.4.5 Fixed priority arbiter

In the second implementation, a fixed priority arbiter [20] was used as controller. The fixed priority arbiter used in the reduction circuit has five request signals as input, one for every rule defined in the algorithm. The requests correspond with the conditions of the rules of the reduction algorithm. For example, the conditions for the fourth rule are: $\mathcal{I}_1$ and $\mathcal{I}_2$ both exist, but come from different rows. If the conditions for a request hold, the request signal will be '1'. Otherwise it will be '0'.

The fixed priority arbiter has a single output, the grant signal. The grant signal is essentially the rule that has to be executed. Thus the fixed priority arbiter picks the request that has the highest priority and sets the grant signal accordingly. The VHDL code to implement the fixed priority arbiter is simply:

```
grant <= ADD_OUT when rule_valid(0 downto 0) = "1" else
         ADD_IN  when rule_valid(1 downto 0) = "10" else
         IN_IN   when rule_valid(2 downto 0) = "100" else
         IN_ZERO when rule_valid(3 downto 0) = "1000" else
         IDLE;
```

The request signals are created using the following VHDL code:

```
rule_valid(0) <= '1' when adder_fifo(0).valid  = '1'
                    and adder_fifo(0).discr = nxt_output.discr
                    and nxt_output.valid = '1' else '0';
rule_valid(1) <= '1' when adder_fifo(0).valid  = '1'
                    and adder_fifo(0).discr = input_queue(0).discr
                    and input_queue(0).valid = '1' else '0';
rule_valid(2) <= '1' when input_queue(0).valid = '1'
                    and input_queue(1).valid = '1
                    and input_queue(1).eq2prev = '1' else '0';
rule_valid(3) <= '1' when input_queue(1).valid = '1' else '0';
rule_valid(4) <= '1';
```

The request signals are generated in parallel. In the first design, a big if-else statement was used that calculated the grant signal directly. Although this is also parallel, the fixed priority arbiter is a generic method to write this down with the advantage that synthesis tools can easily optimize this.

3.4.6 Output buffer contents

When determining the request signals it is important to know that a value is in the output buffer, to determine if the first rule has to be executed. Depending on the design, it can be required that this information is available

Row discriminator	1	2	2	3	3	3	4	4	4	4
Value (first approach)	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
Value (second approach)	1.0	2.0	2.0	3.0	3.0	3.0	4.0	4.0	4.0	4.0

Table 3.2: Example input rows

one clock cycle earlier.

The output buffer is simply one or more BlockRAMs. To determine if a certain buffer position contains a value, it is not desirable to use BlockRAM as it introduces an additional delay while BlockRAM has a limited number of read/write ports. A std_logic_vector is used that contains a bit for each entry in the output buffer to determine if it is valid. However, reading one bit from a big std_logic_vector appeared to be very slow as a large demuxer is used to do this. Currently this is the one of the longest paths in the system.

3.4.7 Testing

I used ModelSim [3] to simulate the reduction circuit. The first approach to test the reduction circuit was to make the length of a row equal to its row index, see table 3.2. Every floating point value is set to 1.0. For example, row 10 of length 10 will yield the result 10.0. The drawback of this approach is that, if there is a delay at the input of the datapath, it will go unnoticed. Any shift in time of the floating point value will not influence the outcome of this test.

The second approach is more robust in this regard. The row index itself will be used as a floating point value, instead of using 1.0 as the floating point value, see table 3.2. The reduction result of any row will be the square of the row index. Any delay in the datapath will now be noticed. Besides that this method of testing is a useful testing tool during simulation, test values can be easily generated inside the FPGA and checked inside the FPGA as well. In my case, I used LEDs to show the status of the test. This test is plausible to work if it shows success for the input row as just described. If this input row is slightly altered, the test should fail. The reduction circuit was successfully tested on the Xilinx Virtex-4 LX160 Development Kit by AVNet using the techniques described here.

3.4.8 Results

The implementation of the reduction circuit was tested on the Xilinx Virtex-4 4VLX160FF1513-10 FPGA. The pipelined binary operator used is a floating point adder ($\alpha = 12$) which was generated using Xilinx CoreGen. The available resources on the FPGA and the resources and characteristics of the

	Available on FPGA	*FP Adder*
Slices	67584	n.a.
Clock Freq. (MHz)	n.a.	253
BlockRAMs	288	0
DSP48	96	3
LUTs	135168	1220
FFs	135168	1139

Table 3.3: Characteristics adder and available resources ($\alpha = 12$)

	Reduction circuits (number of bits)						
	1	2	3	4	5	6	7
Slices	2587(3.8%)	2680(4.0%)	2752(4.1%)	2844(4.2%)	3043(4.5%)	3178(4.7%)	3556(5.3%)
MHz	253	253	253	251	220	210	200
BRAMs	9(3.1%)	9(3.1%)	9(3.1%)	9(3.1%)	9(3.1%)	9(3.1%)	9 (3.1%)
DSP48	3(3.1%)	3(3.1%)	3(3.1%)	3(3.1%)	3(3.1%)	3(3.1%)	3 (3.1%)
LUTs	1651	2962	3007	3060	2287	3237	2927
FFs	2936	1734	1840	1982	3257	2447	3437

Table 3.4: Reduction circuit ($\alpha = 12$)

floating point adder (in isolation) are shown in table 3.3. The resources used by the reduction circuit are shown in table 3.4. In this table, the pipeline depth is kept fixed, while the number of bits for the row discriminator is varied. For slices, BlockRAMs and DSP48 slices, the percentage of the total number of the available resources is shown (see Appendix A for information about FPGAs).

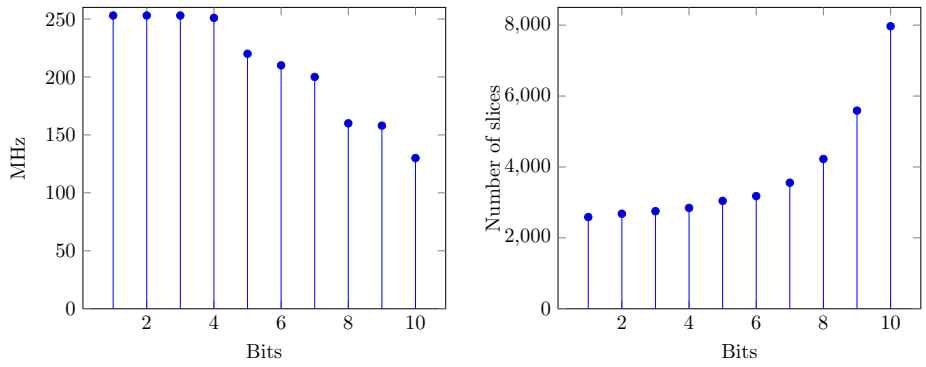
The clock frequency for 1, 2 and 3 bits is 253 MHz. In these cases the clock frequency is limited by the clock frequency of the floating point adder. The clock frequency can be increased by selecting a faster adder, however, this faster adder will have a bigger delay. If 7 bits are used, all possible row lengths can be reduced using this reduction circuit. Theoretically about 19 reduction circuits fit in the FPGA independant of the number of bits. In practise FPGAs are filled for about 50% such that the PAR tooling has enough freedom to place and route. Thus, about 10 reduction circuits is a more practical estimation.

Chapter 4

Evaluation of Results

4.1 Reduction circuit evaluation

In section 3.4.8, characteristics like area and clock frequency were discussed. In this section these results are further analyzed with an emphasis on scalability.



(a) MHz against number of bits used for the row discriminator

(b) Slices against number of bits used for the row discriminator

Figure 4.1: Reduction circuit characteristics plotted against number of bits ($\alpha = 12$)

In figure 4.1a, the clock frequency is plotted against the number of bits used for the row discriminator. This plot shows how the speed changes as the number of bits used for the row discriminator increases. The number of bits depends on both α and on the minimal row length, see section 3.3. This makes the number of bits used for the row discriminator an important variable for analysis.

In figure 4.1b, the number of slices is plotted against the number of bits. For $\alpha = 12$, 7 bits are sufficient to handle all row lengths and thus more bits

have no practical application. Extending the row discriminator beyond 7 bits can be used to study the scalability of the design, even for deeper pipelines that require more than 7 bits. For this reason, the number of bits is extended further than 7 bits for both plots.

The previous analysis shows how the slices and clock frequency changes when the number of bits used for the row discriminator change. When using the reduction circuit in SM×V, the effect of the pipeline depth on the area, clock frequency, buffer size and latency are equally important. The effect of different pipeline depths is plotted in figure 4.2. Instead of using a pipelined adder, I replaced the pipelined adder by a binary *and*, such that every part of the reduction circuit will be used and not optimized away, followed by a shift register of α registers deep. This way, only the reduction circuit overhead is measured, not that of the adder.

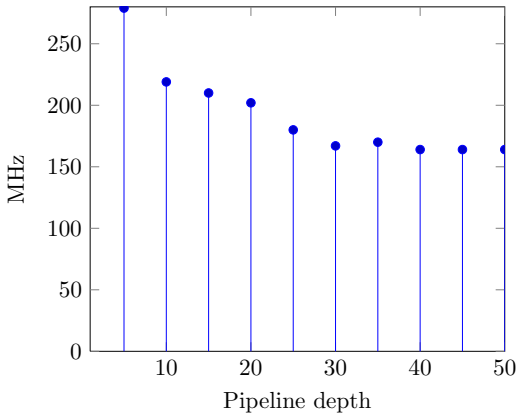
The clock frequency decreases slowly as the depth of the pipeline increases. Even for very deep pipelines, the characteristics are very good compared to related work. For example in [23], a recent study, clock speeds of 165 MHz are shown for a faster FPGA. Pipeline depths that can be found in current floating point adders on FPGAs are in the range of 10 to 20. Deeper pipelines can be expected for future adder designs [5]. The number of slices increases linearly. The number of buffers is simply calculated using $2\alpha + \alpha\lceil\log_2 \alpha\rceil + 1$ for $\mathcal{O}$ plus $\alpha + 1$ for $\mathcal{I}$. The latency is exactly $2\alpha + \alpha\lceil\log_2 \alpha\rceil + 1$. Both the buffer size and latency were discussed in section 3.3.

4.2 Sparse Matrix Vector Multiplication

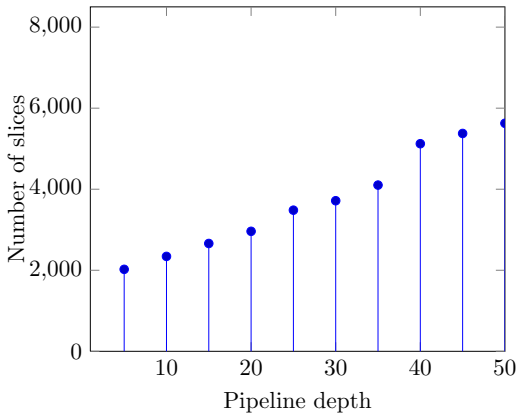
In section 2.2, it was shown that reduction circuits can be used to calculate the SM×V. When using an SMAC (figure 4.3) as introduced in section 2.1, the SM×V can be calculated by streaming in the matrix and vector, the output will be the result vector.

In figure 4.3, it is shown that the vector and matrix are streamed into the SMAC. The multiplication of the floating point values take place in the floating point multiplier, which is pipelined. Meanwhile, the row index is delayed (this is not explicitly shown in this figure). The reduction circuit reduces the input rows. The output of the reduction circuit is the result of the sparse matrix vector multiplication.

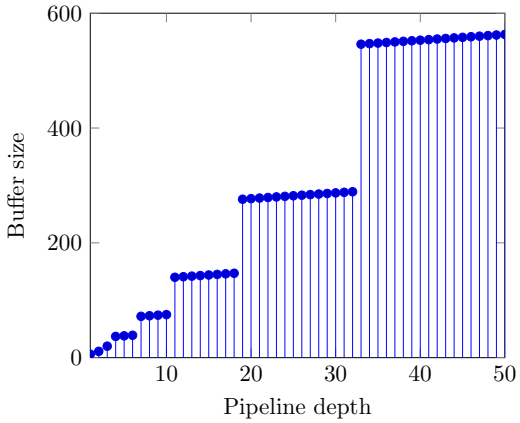
The strength of this solution is that no additional control logic is required. The matrix and the vector are streamed into the FPGA. Furthermore, multiple SMACs units can be placed into the FPGA, yielding an increased performance.



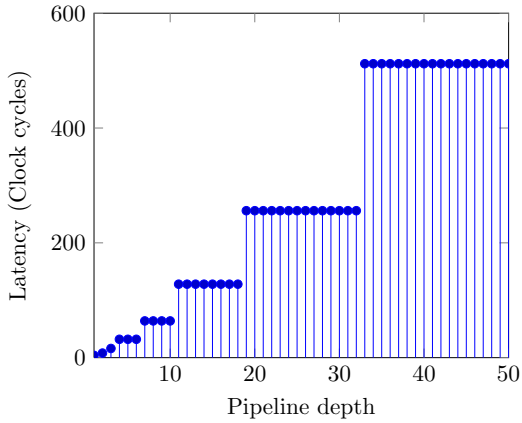
(a) MHz against pipeline depth



(b) Number of slices against number pipeline depth



(c) Buffers against number of bits for the row discriminator



(d) Latency (in clock cycles) against number of bits for the row discriminator

Figure 4.2: Reduction circuit characteristics plotted against pipeline depth (no floating point adder)

4.3 Speculation on performance

In section 3.4.8, it was shown that, in theory 19, reduction circuits fit in the FPGA that is used. However, a multiplier has to be added to the streaming reduction circuit to form an SMAC. This was not tested in practise, but an estimate of 8 SMACs already seems to be pessimistic, if multipliers that use 9 DSP48 slices are used. This means that 8 SMACs can be used to process 8 values at 200 MHz. Thus 1.6 billion values can be processed in a second, if the available bandwidth is ignored. When expressing this in GFLOPS (Giga Floating Point Operations Per Second), it should be taken into account that a SMAC is 2 FLOPS. Thus the design of 8 SMACS has a guaranteed

processing power of 3.2 GFLOPS. Although this is not a lot compared to general purpose processors, there is no overhead, there are no pipeline flushes or whatsoever.

The Diffuse Optical Tomography matrix discussed in [21] has 2,460,562 non-zero values. Since 8 SMACs can process 1600M values per second, the calculation takes 1.37ms per SM×V. In [21] calculating a SM×V on a GPU takes 3.4 ms. Thus using an FPGA gives an improvement of a factor 2.5 when the bandwidth is ignored, thus the processing time of the DOT algorithm is lowered to about 6 hours. As mentioned before, this is a pessimistic estimation of the available processing power.

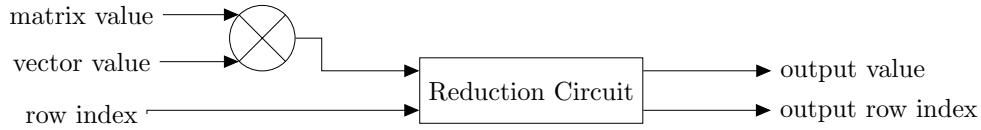


Figure 4.3: Streaming Multiply Accumulate

The memory bandwidth on the FPGA development board that was used is limited. Therefor the actual problem that remains is getting the data to be processed from memory into the FGPA (2 values per SMAC per clock cycle) and writing results back to memory. Whenever a result is produced, it has to be written to the result vector. Thus essentially three memories per Streaming MAC are sufficient.

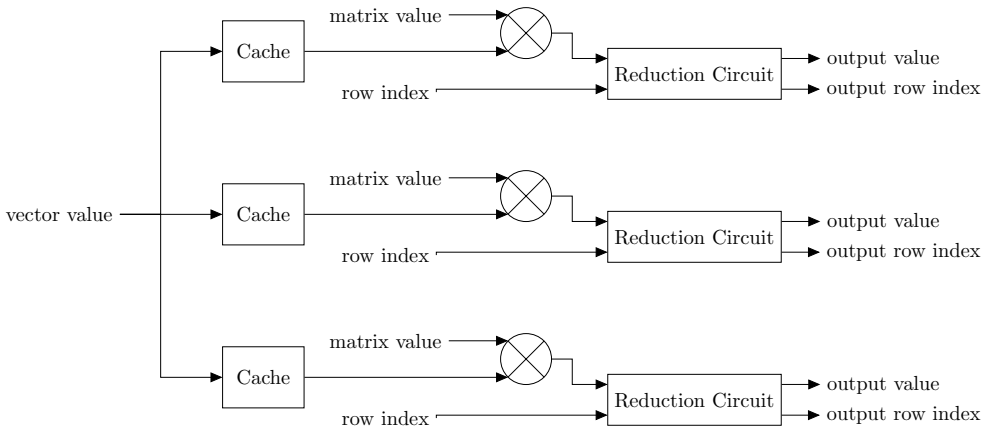


Figure 4.4: SM×V with multiple SMACs and caching

Further optimizations are possible to reduce the required IO bandwidth. Since for each Streaming MAC, a value from the vector is read, bandwidth might be wasted. Caching can be used such that the vector does not have to be re-read for each matrix row, see figure 4.4. The maximum distance between the first non-zero value on a row and the last is 22,393. This is called the bandwidth of the matrix and is illustrated in figure 4.5. If 22,393 values are cached, only one value has to be read from the input vector per clock cycle, assuming maximally one row is processed per clock cycle. The row in the matrix that is processed next is shown dashed in figure 4.5. The cached vector is shown in figure 4.6 where the area of in the vector matched the rows marked in figure 4.5. The area in marked figure 4.6 is the next state of the cache, which matched the area marked in figure 4.5.

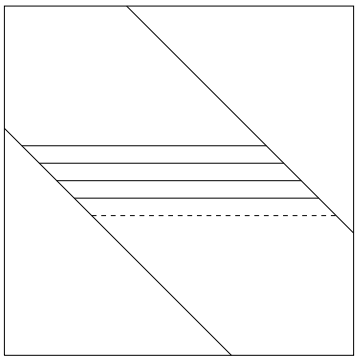


Figure 4.5: Sparse Matrix bandwidth

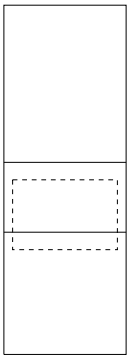


Figure 4.6: Cached vector

For s SMACs, s caches of each $22,393$ floating point values are used. Thus $22,393 \times 64 \times s$ bits of memory are required. When using 18kbit BlockRAMS,

this means $78 \times s$ BlockRAMS are required. Every SMAC requires 9 BlockRAMs as buffers, see table 3.4. Thus for each SMAC, 87 BlockRAMs are used when caching is used. Since the Xilinx Virtex-4 XC4VLX160 has 288 BlockRAMs, 3 SMACs fit into the FPGA if caching is used, this is shown in figure 4.4. Since BlockRAMs can run at 500 MHz while the SMAC runs at 200MHz, the memory can be accessed two times every clock cycle. Because of this two times the number of SMACs fit into the FPGA, thus 6 SMACs fit in the FPGA. One external memory is used for the input vector, 6 for the matrix. Additional memory has to be used for the *row* vector and the results. I did not determine the number of accesses required for *row* and for the results, although the memory bandwidth is very low and possible one additional memory module is sufficient.

The Diffuse Optical Tomography matrix has 2,460,562 non-zero values. Since 6 SMACs can process 1200M values per second, the calculation takes 2.05ms per SM×V. This is still a better performance than the performance mentioned in [21].

The design easily scales to other FPGAs. On the Xilinx Virtex-5 the SMACs can easily run at 300 MHz in the general case. The Xilinx Virtex-5 XC5VF200T has 912 BlockRAMs, which means means 10 SMACs will fit in this FPGA when caching is used. Since these run at 300MHz, the SM×V calculation will take 0.82ms, thus more than four times faster than the GPU implementation. Likewise, the memory bandwidth can be increased by using faster memory. The 200MHz DDR memory assumed is perhaps too pessimistic.

Further studies regarding memory usage are still useful to reduce the required bandwidth. I will come back to this in section 6.1.

Chapter 5

Conclusions

It was shown in literature that reduction circuits are important for the implementation of many scientific and multimedia algorithms on FPGAs [23]. It is possible to avoid using reduction circuits, but in my opinion this will result in a design that is hard to understand and hard to reason about. Since values have to be reordered when reduction circuits are avoided, the complexity increases and the efficiency decreases. With an efficient reduction circuit, it will no longer be required to avoid reduction circuits.

The performance of reduction circuits, measured in latency, speed, area, buffers and the output order, will improve significantly if multiple rows of values are reduced simultaneously by one reduction circuit. The most important method to accomplish this performance gain is keeping track of row discriminators. Most solutions found in literature count operations which makes the algorithm dependant on the length of input rows (see section 2.4). The number of row discriminators and thus the number of bits required for comparators can be reduced by identifying how many rows can coexist in the reduction circuit at the same time. The row indexes as they appear at the input can be re-mapped to small subset of row discriminators.

To make sure the delay is kept to a minimum, rows that have been longer in the system should get a priority in reduction. Because of this design choice, buffers used at the output of the pipelined operator can be kept small. If these buffers are slightly enlarged, results can be produced in-order, while the buffers remain smaller than those of the SSA algorithm [23] which produces results out-of-order.

The proposed algorithm is compared with work from literature in table 5.1. The FCBT algorithm requires bigger buffers if the length of the input rows (n) increases and thus it is not scalable. The DSA algorithm uses two adders, while one of the design requirements was using a single adder. The SSA

Algorithm	Scalable	*Buffer sizes*	*#adders*	*Latency*	*In-order output*
FCBT	No	$3\lceil log_2 n \rceil$	2	$2n + (\alpha - 1)\lceil log_2 n \rceil$	Yes
DSA	Yes	$\alpha\lceil log_n \alpha + 1 \rceil$	2	$\alpha\lceil log_2 \alpha + 1 \rceil$	No
SSA	Yes	$2\alpha^2$	1	$2\alpha^2$	No
Tracking	Yes	Unknown	1	Unknown	Unknown
Proposed algorithm	Yes	$3\alpha + \alpha\lceil log_2 \alpha \rceil + 2$	1	$2\alpha + \alpha\lceil \log_2 \alpha \rceil + 1$	Yes

Table 5.1: Reduction circuits

algorithm has the most favorable characteristics of all work described in literature. However, the buffers grow quadratically while the pipeline depth increases and has out-of-order output, thus it does not meet the design criteria. The tracking algorithm from [4] is not described in detail and it was not proven that it works correctly. The streaming reduction circuit described in this thesis has a low delay, small number of buffers and is fast when compared to reduction circuits described in literature. The proposed algorithm does meet the design criteria. As it was mentioned in chapter 6, there is even room for improvement of both the clock frequency and the area usage.

The implementation of the reduction circuit runs at 200 MHz on a Xilinx Virtex-4 LX160-10 FPGA in the generic case (see section 3.4). If assumptions are made regarding the minimal length of rows of input values, less rows can coexist in the reduction circuit and the number of bits to identify rows can be brought back. As a result, the clock frequency increases while the area of the reduction circuit decreases. When the pipeline depth increases, the performance of the system (measured in clock frequency, area, number of buffers and latency) decreases linearly by approximation.

Multiple SMAC units can be supplied with data in parallel, increasing performance. It is estimated that eight SMAC units easily fit on the specific FPGA (Xilinx Virtex-4 LX160-10), used in this project. Theoretically, a performance improvement of a factor 2.5 can be achieved, if memory bandwidth limitations are ignored. The effects of limited memory bandwidth has to be investigated in further detail.

Chapter 6

Future work

6.1 Matrix implementation

In section 4.2 it was mentioned that more research effort is required to come up with a better SM×V design. The speed of the SM×V is influenced by memory bandwidth, the number of SMACs and caching. Furthermore, the SMAC and the SM×V were not implemented yet.

6.2 Reduction circuit

As described in section 3.4.4, controller triplication can be used to improve the speed of the controller. The current longest path is related to the output buffer as was explained in section 3.4.6. If both issues can be improved, this might result in an increase of the clock frequency. A lot of registers were inserted at several places to help the synthesis and PAR tools to optimize the design (for example, when register retiming is used). However, this increases the area of the reduction circuit. It should be carefully investigated where and how many registers have to be added. Registers that are not required should be removed to bring back the area of the reduction circuit.

6.3 SIMD Adoption

In section 2.4, SIMD instructions were discussed in general and the MDMX instruction set in particular. The disadvantage of this instruction set is the high latency. Since the latency is α clock cycles per value, the total latency is αn clock cycles for a row of length n. The reduction circuit introduced in this thesis has a low and fixed latency independant of n (significantly lower than αn, for $n > \alpha$). Furthermore, it is generic and works for any row length, where the length of rows may vary within a stream of rows.

It might be worthwhile to investigate whether SIMD instructions can be improved using reduction circuits. Special instructions exist that can specify that an input value has to be added to the current contents of the packed accumulator, even several accumulators can be selected. Essentially, the accumulator can be related to the row discriminator of the reduction circuit. There is an instruction that can be used to move the accumulator result to a MDMX register after truncation, rounding and/or clipping. This instruction can still be used, although the execution of this instruction has to be postponed until the reduction circuit is ready and can output the value. Thus $2\alpha + \alpha\lceil \log \alpha \rceil + 1$ clock cycles after the last value of a row entered the system, the register with the result can be read. Since values are packed, the reduction circuit can control multiple parallel operators. Thus special instructions can be added to a SIMD processor to send a row of input values to a reduction circuit and read the reduction result.

It seems pretty straightforward to replace the current reduction accumulators in SIMD instruction sets by a reduction circuit as described in this thesis. However, a direct replacement is not possible since the values at the input of the reduction circuit should be in sequence (thus rows should not be interleaved). Furthermore, if at one clock cycle no value enters the reduction circuit, the end of a row is assumed. This last problem can be solved by inserting zeros as dummy values. However, the user should avoid interleaving rows. Since for most applications values that belong together are grouped, this does not have to be a problem. This requires more research, including benchmarking.

6.4 Expression Evaluation

Expression evaluation is not directly related to the main topic of this thesis. Expression evaluation, as descibed in [16], is a form of reduction where multiple kinds of operators are used. Expressions consist of binary operators (for example +, -, / and *) and parenthesis. In [16], the assumption was made that the expression tree is balanced. Expression evaluation is related to the reduction problem stated in section 2.2, although for expression evaluation the expressions are not always associative. Essentially, expressions should be evaluated in a certain order.

The algorithm proposed in [16] is based on the FCBT algorithm described in section 2.4, where the input is also processed in order, level by level. The reduction algorithm cannot be used directly; operation 1 assumes associativity. The buffer size of $log_2 n$ proposed in [16] that uses a complete binary tree is acceptable, without associativity it is hard and perhaps impossible to do better. In figure 6.1 a collection of operators is shown. The collection of operators together work just like a single operator in the reduction problem,

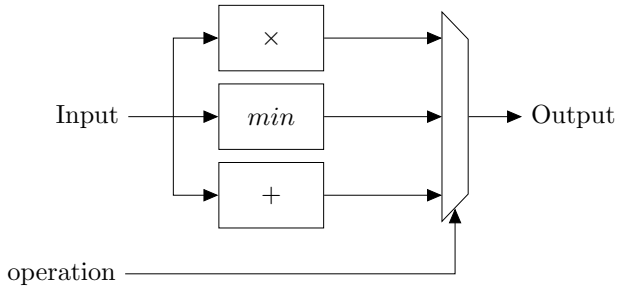


Figure 6.1: Collection of operators (replacement of a single operator)

with the difference that a control signal is required to notify the output multiplexer which operator produces the correct result. When the expression tree is not complete, they use an additional collection of operators.

Although the streaming reduction circuit algorithm cannot be used here, the idea of using row indexes instead of counters to determine the position in the expression tree can be used, this was described in section 2.4 for the FCBT algorithm. Besides a row index for identifying the row (or now expression) index, an identifier to indentify the position within the expression tree is introduced, the *node number*. Nodes are numbered starting with the root node, which has 1 as its row index. The left child of a node gets as node number $parent \times 2$, the right child gets $1 + parent \times 2$ as node number. Thus to determine the node number of the parent, simply divide by two. From the parent node, the operation is determined using a lookup table.

Now it does not matter if the expression tree is complete, or not, removing the need of a second collection of operators. If the tree is balanced, this approach will work. In [16] a balanced tree is assumed, there exist algorithms for balancing according to this paper thus they only studied balanced trees. The impact of balancing should be researched. Further work in this area can aim at reducing the number of buffers, using unbalanced trees, streaming the operator lookup table together with the tree and a proof of correctness.

To summarize, reduction circuits can not be used directly to implement expression evaluation. However, the knowledge about reduction circuits gives insight into expression evaluation, as it is a related problem. The main improvement that can be made is using identifiers to identify both the expression and the position within the expression. A result of using identifiers is that only one collection of operators is required. Since such collection of operators is very big (only a few fit in an FPGA), this is a significant improvement.

Appendix A

Field Programmable Gate Arrays

In this appendix, the terminology related to the Virtex-4 Field Programmable Gate Array (FPGA) is explained.

A.1 Logic

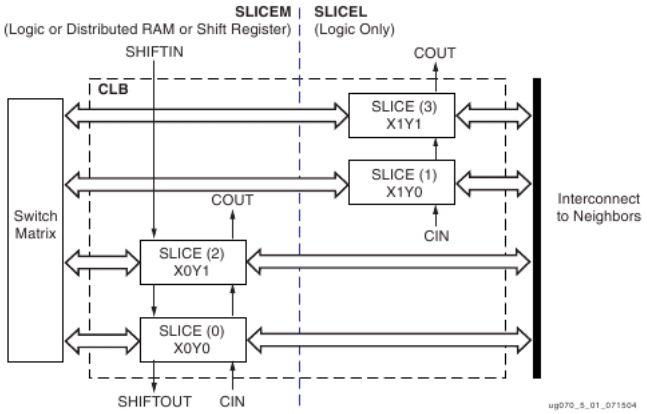


Figure A.1: Configurable Logic Block [1]

The logic is the most characteristic part of an FPGA. The logic consists of Configurable Logic Blocks (CLBs), see figure A.1. CLBs are connected through an interconnect network. Each CLB consists of four slices. Slices contain logic ceels like lookup tables (LUTs) and flip-flops. A lookup table is used to construct logic functions. One CLB consists of 4 slices, 8 LUTs, 8

flip-flops. A CLB can be configured as a 64 bits (distributed) RAM.

When mapping LUTs and flip-fops onto slices, it is not possible to determine how many slices are used. For example, for 16 flip-flops and 16 LUTs, it is possible to use one CLBs (4 slices). On the other hand, resources can not always be shared, for example because of routing or simply because the logic is not related. In that case, 2 CLBs (8 slices) are required instead of 4 slices, one CLB for the flip flops and one CLB for the 16 LUTs. This makes it hard, if not impossible, to compare area if they are not expressed using the same types of logic.

A.2 BlockRAMs

A BlockRAM is a RAM resource on the Virtex FPGAs (see figure A.2). Each BlockRAM has two ports. Two values can be read and two can be written every clock cycle using these two ports, but only two addresses can be used. This means one of the values that is read and one of the values that is written share the address. If multiple ports are required, multiple BlockRAMs are used. For example, to construct a memory with one write port and 3 read ports, 3 BlockRAMs are used. The data input and its address is connected to all BlockRAMs. Which each of the 3 BlockRAMs supplies one of the 3 read ports.

A.3 Logic

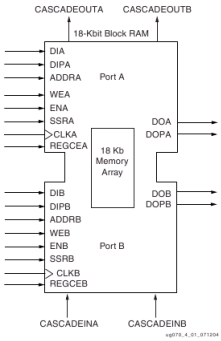


Figure A.2: BlockRAM [1]

Each BlockRAM contains 18kbit memory. BlockRAMs can be configured as RAM or as a FIFO. Pipelining is possible to increase the clock frequency, at the cost of a clock cycle delay. It is even possible to use two clock domains

for a BlockRAM, although I didn't use this feature. The FPGA used for this project contains 96 BlockRAMs.

A.4 DSP48 Slices

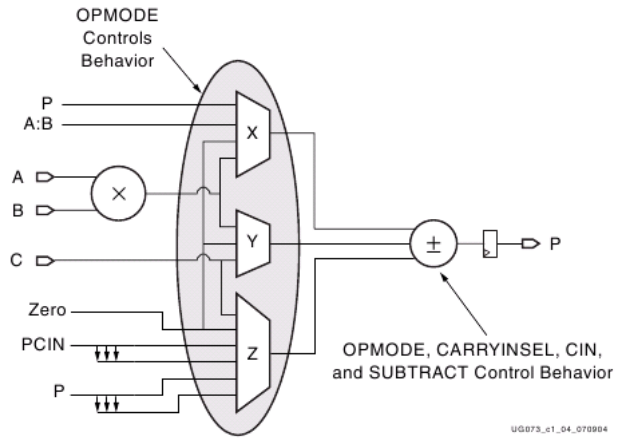


Figure A.3: DSP48 Slice (Simplified) [2]

DSP48 slices, or XtremeDSP slices, contain dedicated logic for arthmetic calculations (see figure A.3). The DSP48 slice consists of a 18x18 bits 2's complement multiplier and a 36 bits adder. The input, adder, multiplier and pipeline registers can be connected in several ways. The DSP48 can be reconfigured at every clock cycle.

DSP48 slices can be connected to eachother to create bigger operators, filters and floating point operators. For this project Coregen is used to create a floating point adder that consists of 3 DSP48s and CLBs.

Bibliography

[1] *Virtex-4 User Guide*, 2004.

[2] *XtremeDSP for Virtex-4 FGPAs*, 2004.

[3] ModelSim, 2007.

[4] 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*, pages 303–304, 2006.

[5] 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*, pages 83–94, 2001.

[6] Y. El-Kurdi, D. Giannacopoulos, and W. Gross. Hardware Acceleration for Finite-Element Electromagnetics: Efficient Sparse Matrix Floating-Point Computations With FPGAs. *Magnetics, IEEE Transactions on*, 43(4):1525–1528, 2007.

[7] A. Gibson, J. Hebden, and S. Arridge. Recent advances in diffuse optical imaging. *Phys. Med. Biol*, 50(4):1–43, 2005.

[8] C. He, G. Qin, M. Lu, and W. Zhao. Group-Alignment based Accurate Floating-Point Summation on FPGAs.

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

[10] G. Morris. *Mapping sparse matrix scientific applications onto fpga-augmented reconfigurable supercomputers.* PhD thesis, University of Southern California, 2006.

[11] 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*, pages 323–330, 2006.

[12] G. Morris and V. Prasanna. Sparse Matrix Computations on Reconfigurable Hardware. *Computer*, 40(3):58–64, 2007.

[13] L. Ni and K. Hwang. Vector reduction methods for arithmetic pipelines. *Sponsored by IEEE*, 20, 1983.

[14] L. Ni and K. Hwang. Vector-Reduction Techniques for Arithmetic Pipelines. *Transactions on Computers*, 100(34):404–411, 1985.

[15] A. Peleg, U. Weiser, I. Center, and I. Haifa. MMX technology extension to the Intel architecture. *Micro, IEEE*, 16(4):42–50, 1996.

[16] R. Scrofano, L. Zhuo, and V. Prasanna. Area-Efficient Arithmetic Expression Evaluation Using Deeply Pipelined Floating-Point Cores. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16(2):167–176, 2008.

[17] H. Sips and H. Lin. An improved vector-reduction method. *IEEE Transactions on Computers*, 40(2):214–217, 1991.

[18] J. Sun, G. Peterson, and O. Storaasli. Sparse Matrix-Vector Multiplication Design on FPGAs. *Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on*, pages 349–352, 2007.

[19] M. van der Veen. Sparse matrix vector multiplication on a field programmable gate array. Master's thesis, University of Twente, 2007.

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

[21] W. Wiggers. Architecture for volume reconstruction in diffuse optical tomography. Master's thesis, University of Twente, 2007.

[22] L. Zhuo, G. Morris, and V. Prasanna. Designing Scalable FPGA-Based Reduction Circuits Using Pipelined Floating-Point Cores. *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 147a–147a, 2005.

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

[24] L. Zhuo and V. Prasanna. Scalable and modular algorithms for floating-point matrix multiplication on FPGAs. *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, 2004.