Sparse Matrix Vector Multiplication on a Field Programmable Gate Array

September 2007

Marcel van der Veen

University of Twente
Faculty of Electrical Engineering, Mathematics and Computer Science
Computer Architecture for Embedded Systems (CAES) group

Abstract

Sparse Matrix Vector Multiplication (SMVM) has been a subject of research in the computer science field quite some time. The SMVM is an expensive operation used in the Conjugate Gradient algorithm. The Conjugate Gradient is an iterative algorithm used to solve linear equations.
The Finite Element Method is used in a lot of engineering areas like structural analysis, fluid dynamics, heat transport and electromagnetism. Within the Finite Element Method, the Conjugate Gradient algorithm is used to solve large sets of linear equations.
Within the research field of SMVM there are two directions, one direction tries to get the most out of general purpose processors while the other direction tries to make a fast implementation on a Field Programmable Array. This project introduced a new method for the SMVM on a Field Programmable Gate Array called Small Bandwidth Coverage (SBC).
The main advantage of SBC is that the variation in performance caused by differences in the system matrices is only a factor two. Similar solutions have a variation in performance of a factor ten.

Preface

This project was carried out within the Computer Architecture for Embedded Systems (CAES) group. I would like to thank everybody of the CAES group for a great ambiance. Everyone was equal, professors, teachers, PhD-students and master students. I always suggest master students looking for a master thesis to take look at the CAES group because of the atmosphere.

I would like to mention a few people in particular.
Gerard Smit is professor and head of the CAES group. Despite his busy schedule he is always sincerely interested in the work of all the people working within the CEAS group. He is one of the main factors for providing a great atmosphere.
Pascal Wolkotte and Philip Hölzenspies are two PhD-students who are always interested in the work of others. Always asking good critical question and making suggestions to improve results.
I also want to thank my supervisors Bert Molenkamp and Andre Kokkeler for reviewing my results and reports. They gave me enough room to work out my ideas while keeping the project within bounds.

Contents

1. Introduction

The Finite Element Method (FEM) is a method often used for structural analysis to compute stress, deformations and internal forces on materials. It can also be used for fluid dynamics, heat transport, electromagnetism and other engineering areas. Recently this method is also used in a new research area: Volume Reconstruction in Diffuse Optical Tomography [1].
Car manufactures use the Finite Element Method for crash analysis. These analyses require a lot of computations. In [2] the execution time of different crash models for super computers is maintained. The crash analysis set is primarily used to compare the speed of different super computers with a real problem. One of the models is a crash with two cars (car2car model). The fastest time is set by the CRAY XT4 super computer with 512 AMD Dual Core Opteron processors running at 2.8 GHZ. Although this is a very fast system it still needs 6274 seconds (104 minutes) to complete [2]. A system with one Intel Dual Core Xeon processors running at 3.0 GHz needs 565261 seconds (157 hours) [2].
The speedup of the supercomputer scaled to 3.0 GHz is (565261/6274)*(3.0/2.8) = 96.5. Normally you would expect a speedup of 512.

The idea of Diffuse Optical Tomography is to reconstruct the 3D volume of tissue. This technique will be used at first instance for the diagnosis of breast cancer. The Volume Reconstruction algorithm in Diffuse Optical Tomography requires a lot of computations but needs to be completed in several minutes. At the moment, the target time is about fifteen minutes [1]. Besides the time to complete, other factors play a role such as costs and the size of the system. Because the amount of computations is substantial but not as much as the car2car model, a super computer would have enough processing power to fulfill the time requirement. The disadvantage is that every system would need a super computer, which is expensive and uses a lot of energy.

In [1] the largest computational part of the Volume Reconstruction algorithm is optimized for an Intel dual-core Xeon 5140 Woodcrest running at 2.3 GHz and a Nvidia GeForce 8800 GTX. The results showed a small advantage for the Intel processor. To get near the target time of fifteen minutes, 88 processors would be needed. The Intel processor achieved about 2% of its peak performance. The Nvidia GPU uses less than 1% of its peak performance. A streaming based FPGA implementation might achieve better results.

2. Problem analysis

2.1. Finite Element Method

2.1.1. Introduction

As explained in the previous chapter the Finite Element Method (FEM) can be used to analyze all kinds of physical processes. The physical processes are represented with models. One of the examples often used to explain the Finite Element Method is the determination of stress and the bending of a truss bridge.

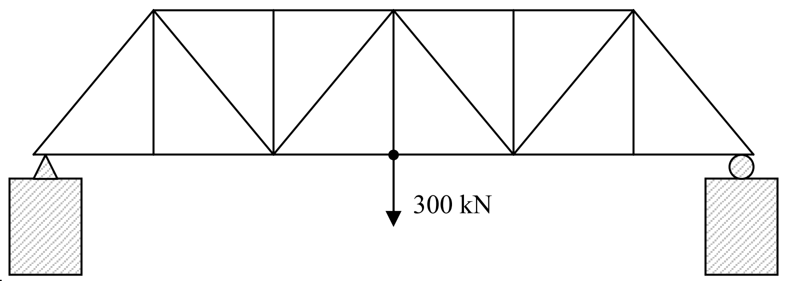


Figure 1: Truss bridge

The idea of FEM is to divide the mathematical model into a finite number of elements to construct a discrete model. For the discrete model of the truss bridge only one simple element has to be used. This simple element is the 2-node truss element (also known as bar). Figure 2 plots the discrete model of the truss bridge.

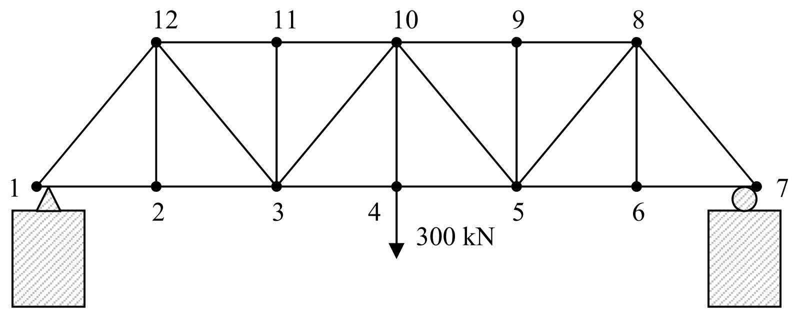


Figure 2: Discrete model of truss bridge

Each intersection of lines has become a node and every line is an element. The discrete model has 12 nodes (numbered from 1 to 12) and 21 elements. With some formulas it is easy to determine the individual reaction of the simple elements. By combining the individual reactions, the reaction of the complete truss bridge can be determined. The combined individual reactions result in a matrix, referred as system or stiffness matrix K. The size of the system matrix K depends on the number of nodes and the dimension used. The truss bridge is modeled in two dimensions with 12 nodes. This means that the size of the system matrix becomes 24 by 24, for each node there is an x and y component.

Within FEM the system that has to be solved is K **u** = **f**. The meaning of vector **u** and vector **f** depends on the problem modeled. For mechanics the vector **u** represents the nodal displacement and vector **f** represents the nodal forces. Without going into detail the vector **f** is known but the vector **u** is unknown.

2.1.2. System Matrix

Consider the following one-dimensional problem.

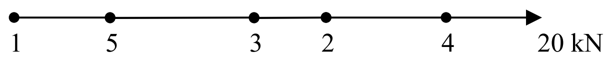


Figure 3: one dimensional structure

The system to solve is K **u** = **f**.

$$\begin{bmatrix} K_{x1x1} & K_{x1x2} & K_{x1x3} & K_{x1x4} & K_{x1x5} \\ K_{x2x1} & K_{x2x2} & K_{x2x3} & K_{x2x4} & K_{x2x5} \\ K_{x3x1} & K_{x3x2} & K_{x3x3} & K_{x3x4} & K_{x3x5} \\ K_{x4x1} & K_{x4x2} & K_{x4x3} & K_{x4x4} & K_{x4x5} \\ K_{x5x1} & K_{x5x2} & K_{x5x3} & K_{x5x4} & K_{x5x5} \end{bmatrix} \begin{bmatrix} u_{x1} \\ u_{x2} \\ u_{x3} \\ u_{x4} \\ u_{x5} \end{bmatrix} = \begin{bmatrix} f_{x1} \\ f_{x2} \\ f_{x3} \\ f_{x4} \\ f_{x5} \end{bmatrix}$$

Figure 4: System to solve of the structure defined in figure 3.

The force on a node only depends on the displacement of de node itself and the displacement of its neighbors. Thus the force on node one depends on the displacements of the node itself and the displacement of node five. The force on node two depends on the displacement of the node itself, node three and node four. Figure 3 represents the system matrix of the structure defined in figure 3. X indicates a non-zero value.

$$K = \begin{bmatrix} \mathrm{X} & & & & \mathrm{X} \\ & \mathrm{X} & \mathrm{X} & \mathrm{X} & \\ & \mathrm{X} & \mathrm{X} & & \mathrm{X} \\ & \mathrm{X} & & \mathrm{X} & \\ \mathrm{X} & & \mathrm{X} & & \mathrm{X} \end{bmatrix}$$

Figure 5: System matrix K of structure defined in figure 3.

The system matrix K in general has some important properties.

- The system matrix is sparse.
  As seen in the example the force on a node only depends on its neighbors. The discrete models of real problems have usually more than 1,000 nodes while the number of neighbors lies in the order of ten. Meaning on average only 1% of the values are non-zero values. This results in a sparse matrix.
- The system matrix is symmetric.
  Consider figure 3. The distance and the material of the element between node one and node five determine the value of K_{x1x5}. This value is the relation between the displacement of node five and the force on node one. Within the system matrix there is another value that represents the same element: K_{x5x1}. Thus for each element there are two values in the system matrix, one above and below the main diagonal.
- The system matrix can be reordered such that all non-zero values lie in a relatively small band.
  Reordering is covered in chapter 2.1.3.

2.1.3. Reordering

The numbering of the nodes is not fixed and can be chosen freely. The numbering of the nodes influences the position of the non-zero values in the system matrix K. For very large matrices a band matrix can have a great advantage. Common reordering methods to reduce the bandwidth are Cuthill McKee (CM) and reversed Cuthill McKee (RCM). RCM renumbers the nodes of the example problem in the following way.

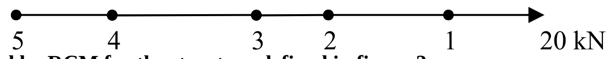


Figure 6: Nodes renumbered by RCM for the structure defined in figure 3.

The renumbered nodes result in the system matrix of figure 7.

$$K = \begin{bmatrix} X & X & & & \\ X & X & X & & \\ & X & X & X & \\ & & X & X & X \\ & & & X & X \end{bmatrix}$$

Figure 7: System matrix of structure defined in figure 6.

Figure 5 and figure 7 represent both the same problem while their shape is completely different. For large system matrices a relative small band can have a large advantage.

2.1.4. Solvers

To solve the system K **u** = **f**, two types of solvers can be used, direct and iterative. A direct solver will factorize the system matrix. Factorizing a large sparse matrix often yields in a dense matrix and is therefore impractical. Iterative solvers do not factorize the system matrix and are therefore preferred for large models. The Conjugate Gradient (CG) method is an iterative solver often used by FEM. The FEM used in the Volume Reconstruction algorithm also uses the CG algorithm.

2.2. Conjugate Gradient method

2.2.1. Introduction

The Conjugate Gradient method is an iterative linear equation solver. The major advantage of an iterative solver is that the matrix does not have to be factorized. Factorizing very large sparse matrices often yields in dense matrices. Handling very large dense matrices is impractical because of the storage requirements and the computational complexity.
The conjugate gradient method is an iterative method to calculate the vector **x** in the system A***x** = **b** where matrix A and vector **b** are given. Matrix A must be square, symmetric and positive-definite.

Although the intuition behind CG is discussed extensively in [3], it will be discussed briefly here. Because the idea behind Steepest Descent and Conjugate Gradient is discussed in a very intuitive way in [3] this explanation has the same outline. Also the same examples and figures as in [3] will be used. Another description of CG can be read in [4].

Another algorithm that solves the same system as CG is Steepest Descent. Both algorithms look very similar, the main difference is that CG converges faster. CG can be explained more easily from the explanation of Steepest Descent.

Steepest Descent and the Conjugate Gradient try to minimize the quadratic form of A***x**=**b**.
The quadratic form is:

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T A\mathbf{x} - \mathbf{x}^T\mathbf{b}$$

It can be proven that if A is positive definite and symmetric, minimizing f is the same as solving A***x**=**b**.

In [3] the following sample problem is used to explain the ideas of SD and CG.

$$A = \begin{pmatrix} 3 & 2 \\ 2 & 6 \end{pmatrix}, \mathbf{b} = \begin{pmatrix} 2 \\ -8 \end{pmatrix}$$

The quadratic form f(**x**) of the example is plotted in the following figure.

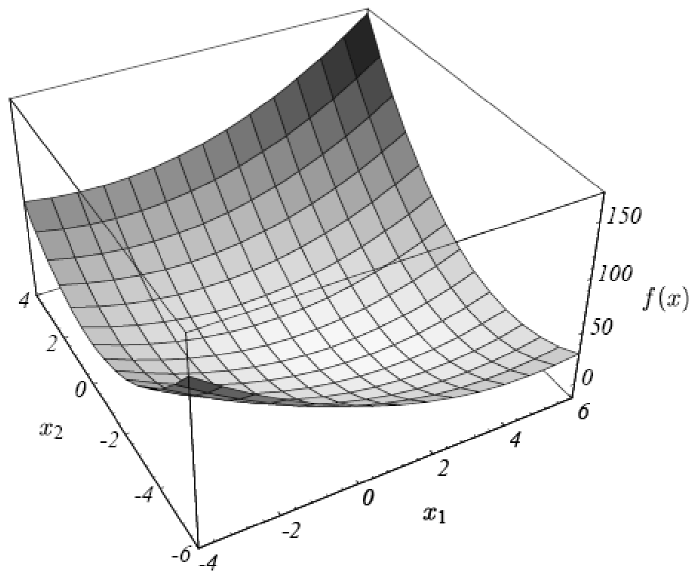


Figure 8: Quadratic form f(x)

The lowest point in figure 8 is the solution of the system A***x** = **b**. In this sample problem that is **x** = [2, -2]. This can be seen more easily in the contour plot (figure 9).

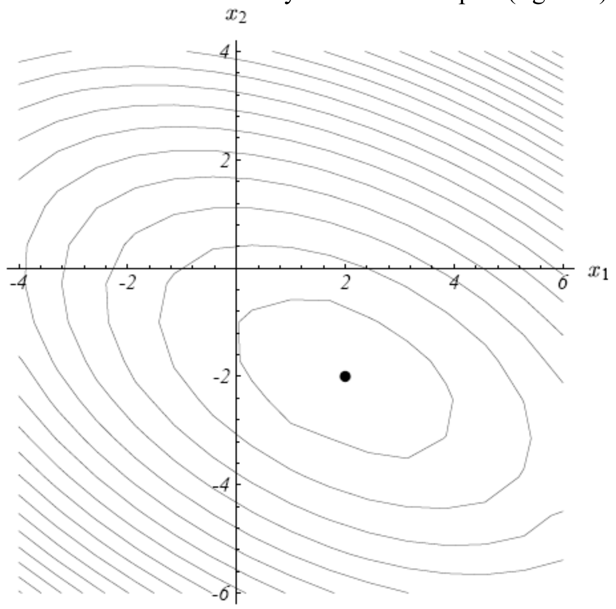


Figure 9: Contour plot of f(x)

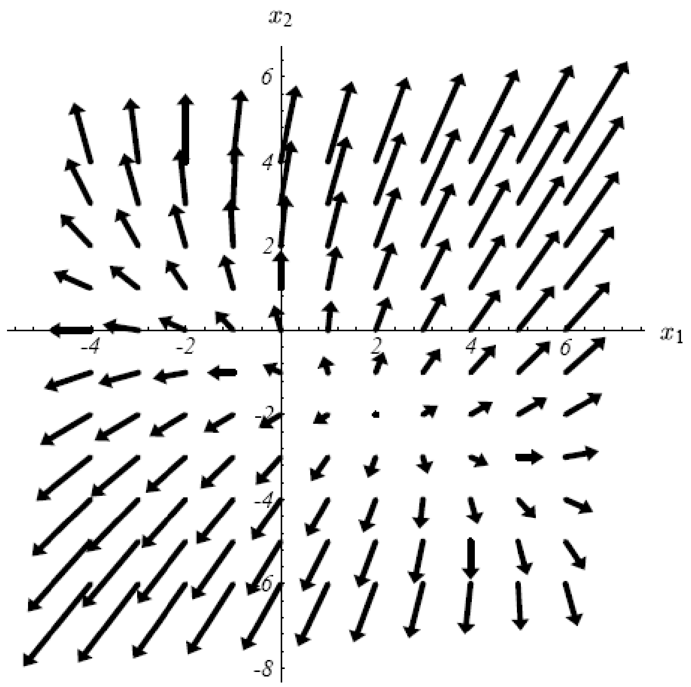


Figure 10: Gradient of f'(x)

De derivative of f(**x**) is defined as f'(**x**) and equals $f'(\mathbf{x}) = A\mathbf{x} - \mathbf{b}$
The gradient points in the direction of the steepest increase of f(**x**). f(**x**) can be minimized by setting f'(**x**) to zero.

2.2.2. Steepest Descent

The Steepest Descent method starts at an arbitrary point $\mathbf{x}_{(0)}$ and every iteration a step is taken to get closer to the solution of A***x**=**b**. Every iteration will result in a better approximation. The number of iterations will depend on the maximum error term specified and the speed of converge.

Every iteration i a step is taken in the direction where f($\mathbf{x}_{(i)}$) decreases most quickly. This direction is the opposite of f'($\mathbf{x}_{(i)}$). This is equal to $-f'(\mathbf{x}_{(i)}) = \mathbf{b} - A\mathbf{x}_{(i)}$, which is defined as the residual. The residual can be described as the direction of the steepest descent.

For the example described above the starting point $\mathbf{x}_{(0)}$ = [-2, -2] is chosen. Every iteration a step is made along the direction of the steepest descent. This results in the following formula for the first iteration:
$\mathbf{x}_{(1)} = \mathbf{x}_{(0)} + \alpha\mathbf{r}_{(0)}$, where α determines the length of the step taken. In [3] α_i is defined as $\alpha_{(i)} = \dfrac{\mathbf{r}_{(i)}^T\mathbf{r}_{(i)}}{\mathbf{r}_{(i)}^T A\mathbf{r}_{(i)}}$.

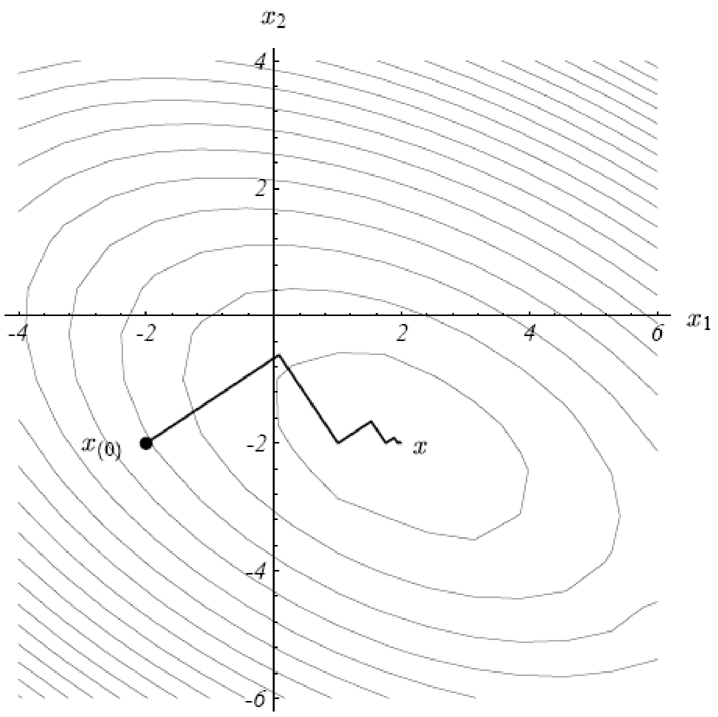


Figure 11: Convergence of Steepest Descent

In figure 11 the convergence of steepest descent for the example problem is shown. In [3] α_i is difined such that the solid lines (in the direction of steepest descent) are orthogonal to each other.

Summarizing the Steepest Descent method is described as:

$$\mathbf{r}_{(i)} = \mathbf{b} - A\mathbf{x}_{(i)}$$

$$\alpha_{(i)} = \frac{\mathbf{r}_{(i)}^T \mathbf{r}_{(i)}}{\mathbf{r}_{(i)}^T A \mathbf{r}_{(i)}}$$

$$\mathbf{x}_{(i+1)} = \mathbf{x}_{(i)} + \alpha \mathbf{r}_{(i)}$$

The method of Steepest Descent as described above requires two matrix-vector multiplications per iteration. By using another formula to compute the residual, only one matrix-vector multiplication is needed.

$$\mathbf{r}_{(0)} = \mathbf{b} - A\mathbf{x}_{(0)}$$

$$\mathbf{r}_{(i+1)} = \mathbf{r}_{(i)} - \alpha_{(i)} A \mathbf{r}_{(i)}$$

This way, the matrix-vector product $A\mathbf{r}_{(i)}$ is used for calculating both $r_{(i)}$ and $\alpha_{(i)}$.

2.2.3. Conjugate Gradient

The idea behind Conjugate Gradient is the same as behind Steepest Descent but instead of multiple steps in the same direction, steps in CG are never in the same direction. Steps in CG are not orthogonal to each other but conjugate.

Conjugate Gradient algorithm:

$$\vec{\mathbf{d}}_{(0)} = \vec{\mathbf{r}}_{(0)} = \vec{\mathbf{b}} - A\vec{\mathbf{x}}_{(0)}$$
$$\alpha_{(i)} = \frac{\vec{\mathbf{r}}_{(i)}^T \vec{\mathbf{r}}_{(i)}}{\vec{\mathbf{d}}_{(i)}^T A \vec{\mathbf{d}}_{(i)}}$$
$$\vec{\mathbf{x}}_{(i+1)} = \vec{\mathbf{x}}_{(i)} + \alpha_{(i)} \vec{\mathbf{d}}_{(i)}$$
$$\vec{\mathbf{r}}_{(i+1)} = \vec{\mathbf{r}}_{(i)} - \alpha_{(i)} A \vec{\mathbf{d}}_{(i)}$$
$$\beta_{(i+1)} = \frac{\vec{\mathbf{r}}_{(i+1)}^T \vec{\mathbf{r}}_{(i+1)}}{\vec{\mathbf{r}}_{(i)}^T \vec{\mathbf{r}}_{(i)}}$$
$$\vec{\mathbf{d}}_{(i+1)} = \vec{\mathbf{r}}_{(i+1)} + \beta_{(i+1)} \vec{\mathbf{d}}_{(i)}$$

In the context of this thesis it is to complex to describe the CG algorithm completely, see [3] for more details.

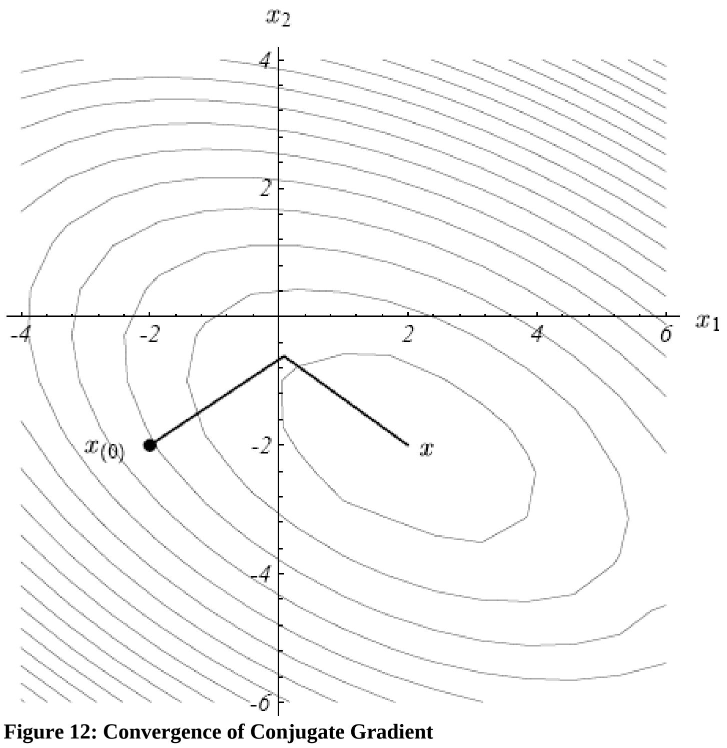


Figure 12: Convergence of Conjugate Gradient

In figure 12 the Convergence of Conjugate Gradient for the example problem is plotted. In general the number of iterations to converge is equal to the length of the vector x. In the sample problem the length of x is two and as can be seen in figure 12 the number of iterations is also two. This is regardless of the position of the starting point.

Unfortunately round off errors (because of floating point operations) result in accuracy loss. In chapter nine of [3] a convergence analysis of CG is done.

2.2.4. Complexity analysis

Variable α and β are scalars, **d**, **r** and **x** are vectors and A is a matrix. There is one initial step which requires a sparse matrix-vector multiplication unless the initial vector **x** only contains zeros.
The computation cost of the iterative part can be divided into different classes:

- One sparse matrix-vector multiplication
- Two inner products
- Three scalar vector multiplications
- Three vector additions/subtractions

The properties of the matrix have great impact on the computational complexity. The number of MAC operations for the SMVM is equal to the number of non-zeros. For dense matrix vector multiplication the number of multiplications is equal to the size of the matrix and does not depend on the number of non-zeros. The more non-zeros, the more dominant the SMVM operation is. In case of the Volume Reconstruction algorithm the SMVM takes 80% of the total required computational complexity of the CG algorithm.

2.3. *Sparse Matrix-vector multiplication*

In the conjugate gradient algorithm one of the operations is a sparse matrix-vector multiplication: A***x** = **y**, A is a sparse matrix, **x** is a dense vector and **y** is the dense result vector. This sparse matrix-vector multiplication has to be calculated every iteration of the CG algorithm. Matrix A does not change during the algorithm only vector **x** changes.

2.3.1. Compressed Row Storage Format

The storage of a dense matrix with n rows and m columns requires the storage of n * m elements. In case of sparse matrices such storage scheme is very inefficient because most of the elements are zero. There are several other schemes to store sparse matrices. These schemes have one thing in common; they only store the non-zero elements. To prevent that the structure gets lost also the indices of the non-zero elements must be stored. The aim of these schemes is that multiplications with zero are not executed. Only the non-zero elements are multiplied with the corresponding elements of the vector. The corresponding elements of the vector can be accessed directly because the indices of the non-zero elements are stored. The most used format to store sparse matrices is the Compressed Row Storage (CRS) format.

CRS uses three vectors to store a sparse matrix. Vector ‘val’ contains all the non-zero elements. The order in which they are stored is row-wise. The vector ‘col’ contains the column index of each element stored in vector ‘val’. The vector row indexes the start of a new ‘row’ within the ‘val’ and ‘col’ vectors. Notice that there is an additional index in the ‘row’ vector to indicate the end of the last row.

Example:

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

$$\begin{aligned} \text{val} &= \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \end{pmatrix} \\ \text{col} &= \begin{pmatrix} 1 & 4 & 3 & 2 & 3 & 4 \end{pmatrix} \\ \text{row} &= \begin{pmatrix} 1 & 3 & 4 & 6 & 7 \end{pmatrix} \end{aligned}$$

Figure 13: Storage of matrix in Compressed Row Storage format.

With the following pseudo code the SMVM of a sparse matrix in CSR format can be computed:

```
for (int i=1; i =< n; i++)
    for (int j = row(i); j < row(i+1); j++)
        y(i) = val(j)*x(col(j)) + y(i);
```

n is the number of rows of the matrix

2.4. FEM Matrices

In chapter 2.1.2 the general properties of the system matrix were given. These general properties are:

- The system matrix is sparse.
- The system matrix is symmetric.
- The system matrix can be reordered such that all non-zero values lie in a relative small band.

Although the general properties hold for all the system matrices there can be quite some differences between them. At [11] a collection of system matrices is maintained. Implementations for SMVM are often benchmarked with these matrices. Besides these matrices this project focuses on the system matrix of the Volume Reconstruction algorithm, described in [1].

There is a lot of variation in the performance of current implementations for SMVM. These variations in performance are a direct result of the variations of the system matrices. This chapter addresses the differences in system matrices.

To classify the system matrices the following properties will be used:
- Size of the matrix (n * n)
- Number of non-zeros
- Sparsity of the matrix
  This value indicates the number of non-zeros compared to the number of entries.
- Bandwidth
  Maximum difference in column index between the non-zero elements of one row.
- Relative bandwidth
  Ratio between n and the bandwidth.
- Sparsity of the band
  Indication on the number of non-zeros compared to the number of entries of the band.

2.4.1. Example of a regular system matrix

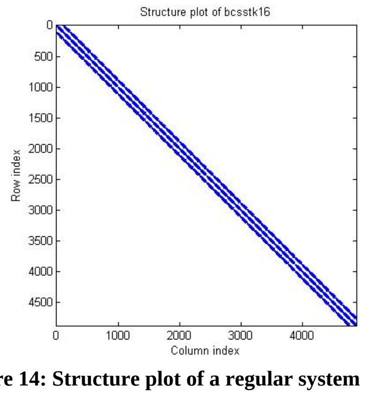


Figure 14: Structure plot of a regular system matrix

Figure 14 is a structure plot of matrix bcsstk16 which can be found at [11]. As can be seen in the figure, the matrix has a regular structure.

The regular matrix bcsstk16 has the following properties:

Size	4,884 * 4,884
Non-zeros	290,378
Sparsity	1.2 %
Bandwidth	277
Relative bandwidth	5.7 %
Sparsity of the band	21.5 %

Table 1: Properties of regular system matrix bcsstk16

2.4.2. Example of a irregular system matrix

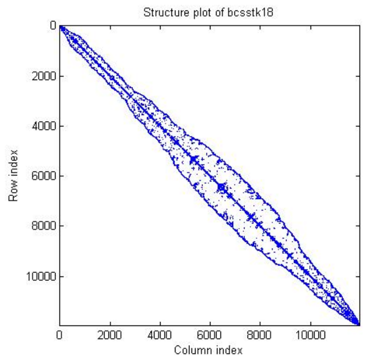


Figure 15: Structure plot of matrix bcsstk18

The matrix bcsstk18 (can be found at [11]) has the following properties:

Size	11,948 * 11,948
Non-zeros	149,090
Sparsity	0.1 %
Bandwidth	2,483
Relative bandwidth	20.8 %
Sparsity of the band	0.5 %

Table 2: Properties of system matrix bcsstk18

2.4.3. System matrix of the Volume Reconstruction algorithm

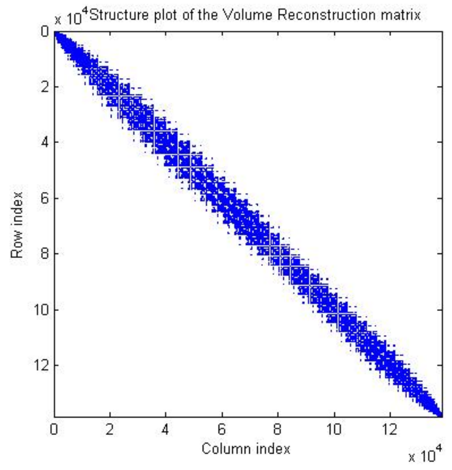


Figure 16: Structure plot of the Volume Reconstruction system matrix

Size	138,324 * 138,324
Non-zeros	2,460,562
Sparsity	0.013 %
Bandwidth	22,393
Relative bandwidth	16.2 %
Sparsity of the band	0.079 %

Table 3: Properties of Volume Reconstruction system matrix

Figure 16 might give the impression that the sparsity of the band is quite high (high percentage). Figure 17 is a zoomed structure plot. From this figure it is clear that the sparsity of the band is low.

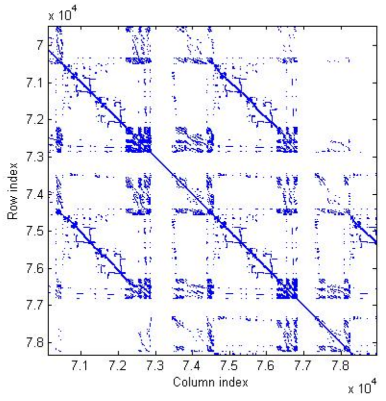


Figure 17: Zoomed structure plot of the Volume Reconstruction system matrix

The most important difference between the matrices described above, is the sparsity of the band. The most implementations of the SMVM achieve a reasonable performance for the regular system matrices as given in chapter 2.4.1. For the irregular system matrices the performance drops on average a factor 5 see [10]. This project focuses on the system matrix of the Volume Reconstruction algorithm. A good implementation for the SMVM will handle irregular system matrices almost as well as the regular system matrices.

2.5. Field Programmable Gate Array

There are several hardware architectures to perform complex tasks. One of these hardware architectures is a Field Programmable Gate Array (FPGA). FPGAs are devices that have a high performance potential while maintaining high flexibility.
An FPGA is a device which has a lot of components. Often there are a lot of simple components such as 4-input LUTs (Look Up Tables) to implement for example AND, XOR, NOR or user defined functions, which are often combined with a flipflop. More complex components are for example the dedicated 18*18 bit multipliers and the dedicated memory blocks.
The components are connected through a programmable interconnect. Complex functions can be implemented by combining the standard components which is done by configuring the FPGA. An FPGA design is thus represented by a configuration file. The same FPGA can be used for different algorithms just by loading another configuration file. Further, the time to market of an FPGA design is short compared to an ASIC design.
Often it is not so hard to implement an algorithm onto an FPGA, developing an implementation that uses the full potential of the FPGA is however difficult.

2.5.1. MAC unit

For digital signal processing algorithms there are two common number representations, fixed point and floating point. Fixed point has a low hardware cost but a low precision while floating point is more complex but has a high precision. For most digital signal processing algorithms, fixed point numbers have a sufficient precision. Because the hardware for fixed point operations is much simpler compared with floating point most of the digital signal processing algorithms are executed on fixed point hardware. Almost all the DSPs use fixed point numbers.

The Volume Reconstruction algorithm requires a high accuracy with a large range, fixed point hardware is thus not an appropriate choice. Within floating point, 32 and 64 bit are common widths. The Conjugate Gradient (CG) algorithm converges faster when higher accuracy is used [1]. The difference between the number of iterations CG needs to converge when using 32 or 64 bit floating point numbers is quite significant. Simulations have shown that the difference is roughly a factor two. The difference in the converge rate is caused by rounding off results. Rounding off numbers to 32 bit floating point numbers results in precision loss compared to 64 bit floating point numbers. The speedup in the number of iterations might justify the extra hardware cost of 64 bit floating point numbers compared to 32 bit floating point numbers see [1].

In [7] a 64 bit floating point MAC unit for an FPGA is presented. This MAC unit exploits the dedicated 18x18 bit multipliers that are present in the order of hundreds on current FPGAs. The 64-bit MAC unit presented in [7] uses nine 18x18 multipliers and has twelve pipeline stages to achieve high performances. The largest FPGA of the Virtex-II pro family from Xilinx, the XC2VP100 (in this project the target FPGA), can hold 31 of these 64-bit floating point MAC units running at 170 MHz (speed grade -6).

2.5.2. Processing Element

As the name already does suggests, a Processing Element (PE) has to process something. It has to perform an operation on data. In case of SMVM the only operations are multiplications and additions. From this property it follows that a PE at least has to be able to compute a multiplication or an addition, but much more complex designs are possible. A PE might for example also perform a combined multiply-accumulate instruction. Other variations are the number of inputs. There are at least two operands required for both the multiply and accumulate instructions, but this might be extended to for example eight operands. It depends completely on the algorithm what kind of combination of PEs gives the best performance. The choice is to have either a lot of simple PEs, a few complex PEs or a combination of simple and complex PEs.

Usually a design can be split into a number of PEs and a common part. The common parts are for example memories shared by PEs, busses shared by PEs, communication between PEs, etcetera. Some parts are only used by one PE. These parts are thus related to a specific PE. Often it is beneficial to have a memory block that is only used by one PE. The same holds for communication busses between memory and the MAC unit.

At this moment the PEs are specified to have at least a MAC unit as presented in [7]. This MAC unit has three operands, is fully pipelined (four multiplication and eight adder pipeline stages) and every clock cycle a MAC result can be computed. These assumptions result in a relatively simple PE which might result in a relatively high number of PEs per FPGA. The size of the memories of a PE depends on the algorithm and will be specified in the following chapters

3. Design requirements

In all known SMVM implementations the limiting factor is the available memory bandwidth. The complete matrix cannot be stored on the FPGA itself. The matrix has to be stored in external memory.
Thus to compute the SMVM, the matrix has to be transferred from external memory to the FPGA at least once. The main target of the design is to use the available memory bandwidth as efficiently as possible.

The implementation of the SMVM thus has the following requirements:

- The design has to keep up with the memory interface; it should not become the bottleneck.
- The overhead that might be needed to schedule the SMVM must be kept to a minimum.
  Most of the memory bandwidth must be used to transfer the matrix and not to transfer overhead.
- The design has to be scalable in the available memory bandwidth.
  More bandwidth should mean a faster transfer and thus a faster computation. This implies that the utilization of the Processing Elements has to be reasonably high.
- The design has to compute the SMVM of regular and irregular system matrices evenly well.

4. Previous work

There are many papers written on the implementation of Sparse Matrix-Vector Multiplication. Within the research field of SMVM there are two main directions. One direction tries to make a fast implementation on processors (GPP, Cell processor, VLIW processors); the other direction tries to make a fast implementation on FPGAs. This chapter addresses three FPGA implementations with their strong and weak points.

4.1. Striping based implementation

A recent paper about a SMVM implementation is [10]. Their implementation uses a stripe method which was introduced by R. Melhem in [9]. The stripe method is discussed in chapter 4.4.

Strong points:
- Streaming based implementation

Weak points:
- Utilization of implementation differs a lot because of the differences of the FEM matrices.
- They never address the problems their method has and how they solved it.
- The performance of their implementation is not linear in the available memory bandwidth as can be seen in [12].
- They used benchmark matrices from [11], but forgot to index them.

4.2. Preprocessor implementation

Another recent paper is [13], this implementation uses a preprocessing stage to compute the SMVM with a high utilization of the PEs. A major part of their implementation is discussed in chapter 5.

Strong points:
- High utilization of the PEs

Weak points:
- Complete matrix stored on the FPGA. Matrices that do not fit on one FPGA need multiple FPGAs.
- Complete result vector stored on FPGA.
- Implementation computes $\mathbf{y} = A^p * \mathbf{x}$, $_{p\geq 1}$.

4.3. Straight forward implementation

A straight forward implementation is introduced in [14]. This method computes the SMVM directly with the standard CSR format.

Strong points:
- No preprocessing required
- Simple partial results adder implementation
- No overhead
- Works evenly well for regular and irregular system matrices

Weak points:

- High utilization difference between multiplier and partial result adder (eight adders and only one multiplier)
- The performance of their implementation is not linear in the available memory bandwidth (because of the use of only one multiplier).

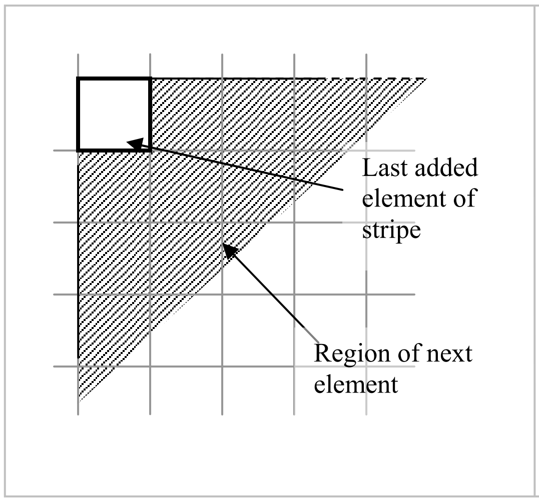
4.4. Stripe method

The use of a systolic array to compute the matrix vector multiplication where the matrix has a dense band has proved its use in [8]. In [9] a method is described to compute the matrix vector multiplication where the matrix has a sparse band. The method described in [9] achieves higher efficiencies for sparse band matrices compared to [8].

Consider a sparse matrix vector multiplication A***x** = **y**. A is a sparse matrix with bandwidth b, **x** is a dense vector and **y** is the dense result vector.
In [8] to compute the matrix vector multiplication, b processing elements (PEs) are required. Each PE multiplies a straight-diagonal of A with the vector **x**. The efficiency of this approach is determined by the sparsity of the band b. A dense band means a high efficiency and a sparse band will result in a low efficiency. As explained in chapter 2.4 the band of sparse matrices is often sparse.
R. Melhem explained in [9] a method to improve the low efficiency for Sparse Matrix-Vector Multiplication (SMVM) if the band is sparse by lowering the number of PEs. This is done by covering the non-zero elements of the matrix by stripes. The collection of all the stripes required to cover all the non-zero elements is called a stripe cover. The number of PEs is equal to the number of stripes needed to cover all the non-zero elements.

4.4.1. Stripes

There are various ways to construct a stripe through a matrix for the coverage of the non-zero elements. In [10] a classification is given, increasing order (IO), strictly increasing order (SIO), strict-column increasing order (SCIO) and strict-row increasing order (SRIO). These classifications can be explained with regions. Each stripe is constructed in an iterative way, every iteration an element is added to the stripe.



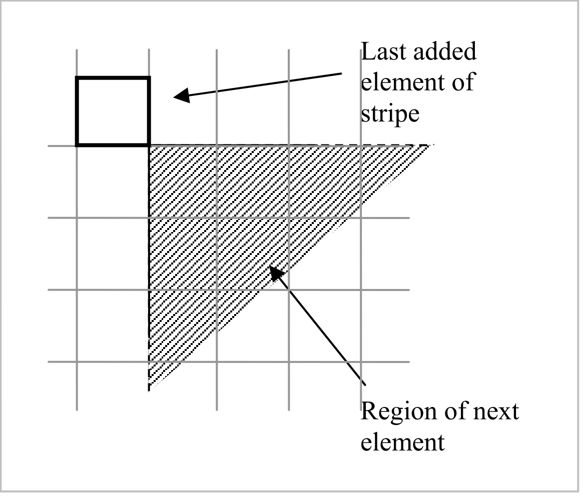
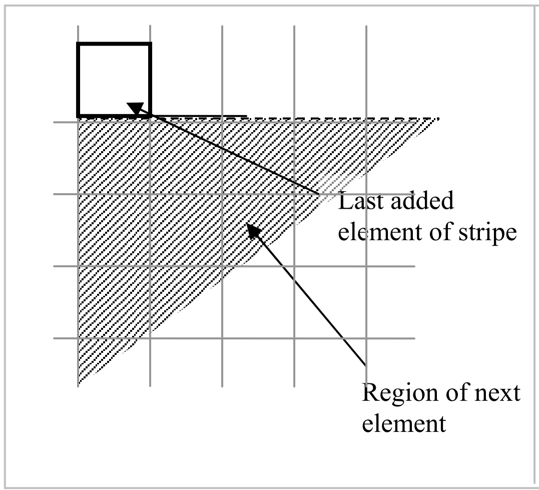


Figure 18: Region of IO Stripes

Figure 19: Region of SIO stripes



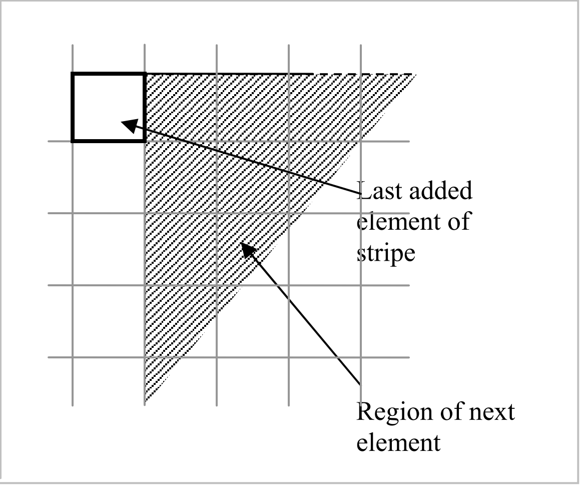


Figure 20: Region of SRIO stripes

Figure 21: Region of SCIO stripes

R. Melhem uses SIO stripes in [9], SRIO stripes are used in [10] and IO stripes are used in [6]. SCIO stripes are not used because they do not have advantages over the other striping schemes.

The stripes in [9] are SIO and have thus the following properties:

- A stripe contains at most one element of every row.
- A stripe contains at most one element of every column.
- From the first property it follows that the longest stripe covers at most n elements.
- Because of property one the elements on the same row are covered by different stripes.

The last property gives a lower bound on the number of stripes. The lower bound of the number of stripes is equal to the maximum number of non-zero elements on a row of the matrix.

In [10] a small modification on the region to construct stripes is proposed. The region is extended to be able to cover non-zero elements in the same column. This leads to SRIO stripes, which have almost the same properties as SIO stripes. The only difference is that a SRIO stripe can contain multiple elements of the same column.

Examples:

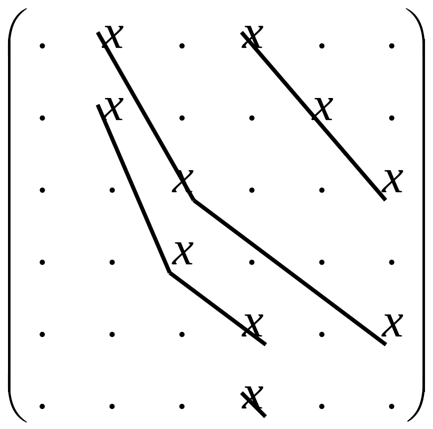


Figure 22: Four SIO Stripes

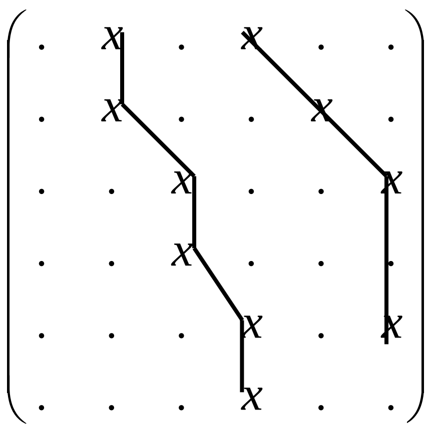


Figure 23: Two SRIO Stripes

The number of SRIO stripes is less or equal to the number of SIO stripes to cover all the non-zero elements [10] although they have the same lower bound on the number of stripes. Only SRIO stripes will be covered in the next chapters because they can give better results, see [10].

4.4.2. Construction of SRIO stripes

There are several ways to construct SRIO stripes. In [10] two types are described, top-down striping (TDS) and bottom-up striping (BUS). Both methods return the same number of stripes thus it is sufficient to only explain TDS. On each row a number of non-zero elements need to be covered by a stripe. TDS starts at the last non-zero element of the first row. The last non-zero element of a row is the non-zero element with the highest column index of that row. The non-zero element of the starting point is assigned to the first stripe. The second last element of the first row is assigned to the second stripe. These steps are repeated until the first element of the first row is assigned to a stripe.
TDS continues at the last element of the second row. If the column index of the element is larger or equal to the column index of the first stripe, it is assigned to the first stripe else it tries to assign it to a stripe with a column index smaller or equal to the column index of the element. The same is done with the second last element of the second row, etcetera.
TDS assigns thus the non-zero elements to stripes from top to bottom and from right to left. Right to left means from the highest column index to the lowest. The stripes are thus constructed like a cheese-slicer would slice an apple.

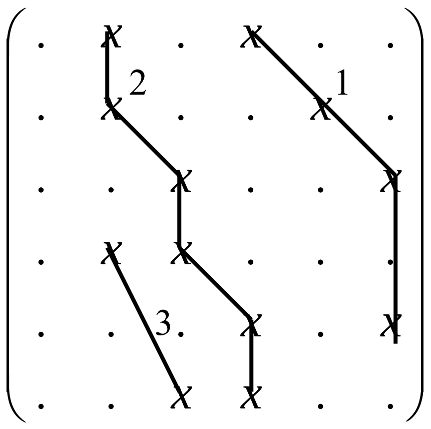


Figure 24: Stripes constructed with TDS method

4.4.3. Systolic array

The method described in [10] uses a number of processing elements (PEs) together forming a systolic array. Each PE is able to compute a multiply accumulate.

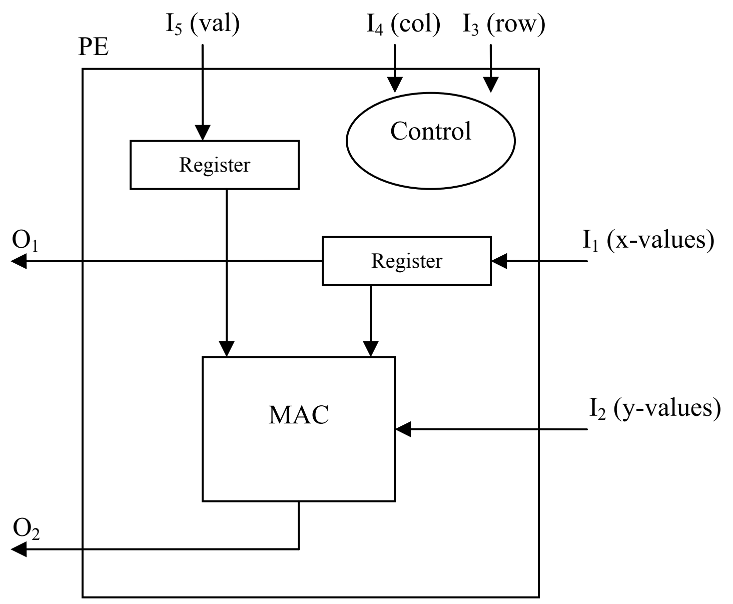


Figure 25: Processing element of the systolic array

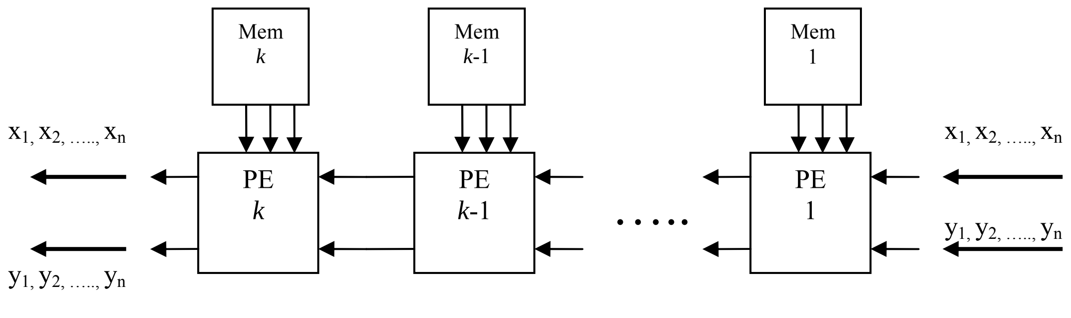


Figure 26: Systolic array for SMVM

Each processing element has five inputs and two outputs.
The input I_1 is for the x-values, I_2 for the y-values and the group I_3, I_4, I_5 for the stripe values (row index, column index and value). Each PE has a local memory to hold the values of the stripe it processes. Vector **x** and **y** stream through the system from right to left. In the initial case all the y-values are zero. Between PEs the y-values are partial results and after the last PE the actual values are available. Communication between PEs is done through FIFO queues.

Each processing element processes a stripe. The first stripe is processed by the first PE, the second stripe by the second PE, etcetera. Figure 24 indicates the numbering order for the stripes, figure 26 indicates the numbering order for the PEs. The number of PEs is thus equal to the number of stripes to cover all the non-zero elements. The complete result vector **y** streams through the system once. A PE can either compute a partial result and add it to an element of **y** or pass it to the FIFO of the next PE, this depends on the stripe the PE is processing. If for example a stripe covers an element on the first row, the PE that processes the stripe will compute a partial result and add it to the first element of the result vector **y** and then passing it on. If the stripe does not cover an element of the first row, the first element of the vector **y** will be passed on without computing and adding a partial result. To compute a partial result a multiplication of two values is required, adding the partial result to an element of **y** takes an addition of two values. Each PE thus requires a MAC unit to be able to perform these operations.

4.4.4. Utilization

To estimate the best case utilization of the MAC units the assumption is made that passing on an element to the next PE with or without computing and adding a partial result takes one clock cycle. This assumption implies that the MAC unit is fully pipelined. The result of the assumption is that the time to stream the result vector **y** through the system in the best case is equal to the length of the result vector **y**. In case of a matrix A with size n x n, n clock cycles are needed to stream the result vector **y** through the system regardless of the amount of partial results.
Suppose k stripes are required for the coverage of all the non-zero elements of a matrix. The systolic array would contain k PEs. In the ideal case n*k partial products could be computed and added to the result vector **y**. The number of partial products is equal to the number of non-zero elements. The utilization of the best case scenario of the system can be computed with the formula: nnz/(n*k).
nnz is the number of non-zero elements.
n is the size of the vector **y**.
k is the number of stripes and equal to the number of processing elements.

The best case utilization of the MAC units for the SMVM with a stripe cover varies greatly as can be seen in [10]. The overall utilization can be as high as 80% but also as low as 3%. This difference is caused by the sparsity and the irregularity of the band of the matrix. To support this, three examples are given. The first two examples can be downloaded from the Matrix Market website [11].

Example one:

Matrix name:	s3rmt3m
Size:	5,357 x 5,357
Non-zero elements:	207,123 0.72%
Stripes:	72

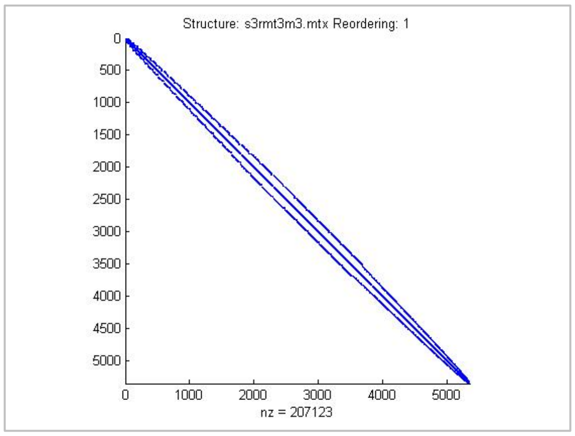


Figure 27: Structure plot of matrix s3rmt3m

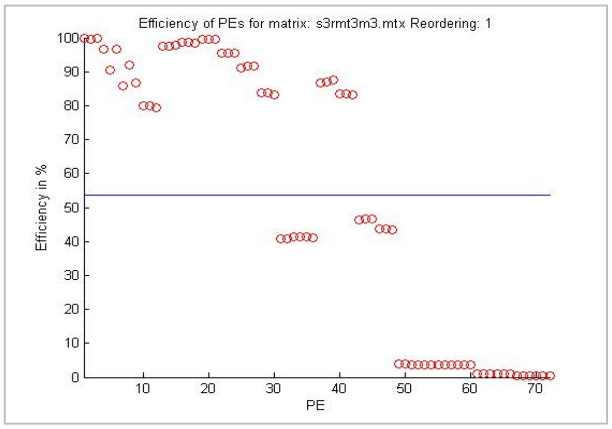
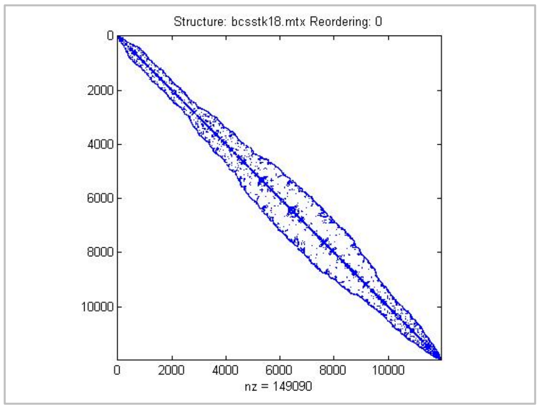


Figure 28: Utilization of PEs for matrix s3rmt3m

The solid line drawn in figure 28 represents the efficiency of the total design. In this case the overall efficiency is thus 54%.

Example two:

Matrix name:	bcsstk18.mtx
Size:	11,948 x 11,948
Non-zero elements:	149,090 0.1%
Stripes:	216



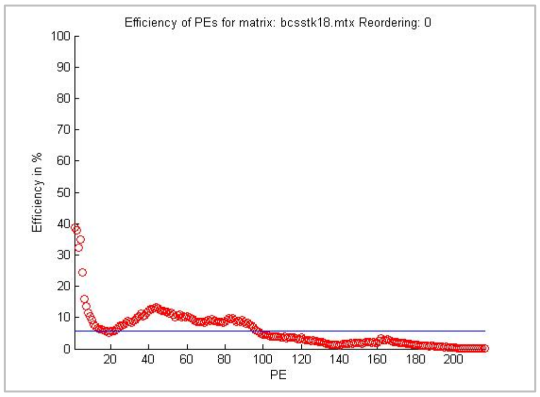


Figure 29: Structure plot of matrix bcsstk18

Figure 30: Utilization of PEs for matrix bcsstk18

Because of the irregularity and sparsity within the band of matrix bcsstk18, the overall efficiency is only 5%.

Example three:

Matrix name:	Volume Reconstruction matrix
Size:	138,324 x 138,324
Non-zero elements:	2,460,562 0.013%
Stripes:	555

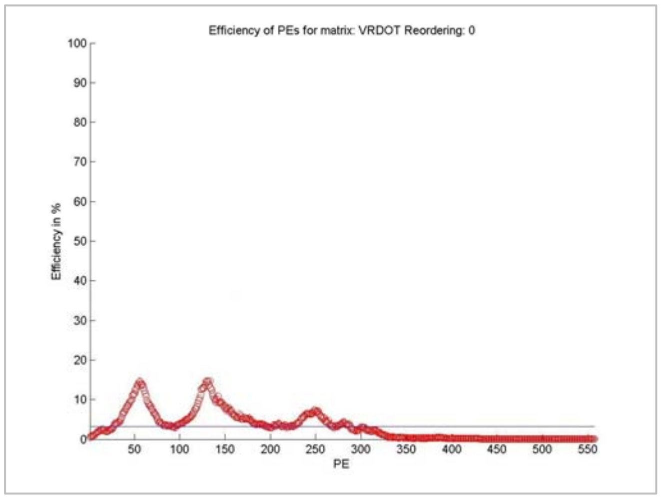


Figure 31: Utilization of PEs for Volume reconstruction matrix

The overall utilization for the Volume Reconstruction matrix is about 4%.

4.4.5. Cause of low utilization

The number of non-zeros a stripe covers directly influences the utilization of the PEs. To understand why the stripe method results in a low utilization for certain matrices, the construction process of a stripe has to be reviewed.

The following situation occurs frequently in the construction of stripes.

$$\begin{pmatrix} .. & .. & & & e_1 & \\ .. & & e_2 & & & \\ .. & e_3 & & & & \\ .. & & & e_4 & & \\ .. & & & & & e_5 \end{pmatrix}$$

Figure 32: Part of a matrix

Figure 32 represents a part of a matrix, e_1 till e_5 represent non-zero elements. At a certain point stripe s_1 is constructed. The construction of a stripe is iterative; every iteration a non-zero element is added. Suppose the last added non-zero element added to s_1 is e_1. Because of the “construction rules” (region of next element) e_2 till e_4 cannot be covered anymore by s_1. After covering e_1 the only element that could be covered by the same stripe is e_5. This yields in a low utilization of the PE that processes stripe s_1, utilization of $s_1 = 2/5 = 0.4$.
The situation explained above occurs often with matrices of real problems. In the example above there are only three rows between e_1 and e_5. With matrices of real problems the distance between two elements covered by a stripe are a lot larger (between 100 and 1000 rows). The utilization of these stripes is often less then 1%.

4.4.6. Conclusion

The problem with a SMVM is to exploit the sparsity of the matrix. In this research field a number of methods are proposed to accomplish this. The use of a systolic array combined with a stripe cover is one of these methods. In [10] and in this report, analysis on the utilization of a stripe cover is done for several matrices. For sparse band matrices with a regular structure the method can achieve a high utilization but for irregular sparse band matrices the utilization is significantly lower. As seen in chapter 2.4.3, the band of the Volume Reconstruction matrix is also sparse and irregular. Analysis showed a best case utilization of 4% using SRIO stripes.

4.5. Parallel Matrix Communication Network

In [6] a modification on the ideas in [9] is proposed. This modification leads to Parallel Matrix Computation Network (PMCN). This method also uses a systolic array to compute the SMVM. In chapter 4.4.1 of this report a classification for stripes is given.
In the original idea of R. Melhem in [9], SIO stripes are used. In chapter 4.4 the modification of using SRIO stripes proposed in [10] is discussed. This modification leads to better results.
PCMN is also a modification of the stripe scheme. Instead of using SIO stripes, IO stripes are used. These IO stripes are also known as staircases. Instead of a stripe cover a staircase cover is used.

Examples:

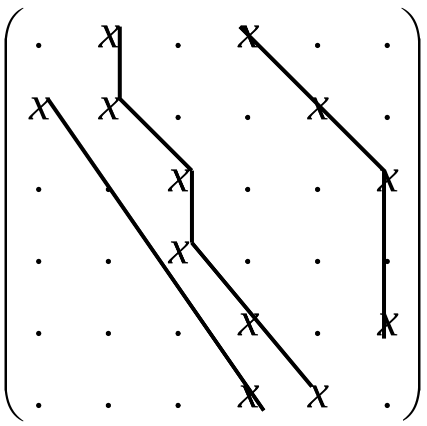


Figure 33: Three SRIO Stripes

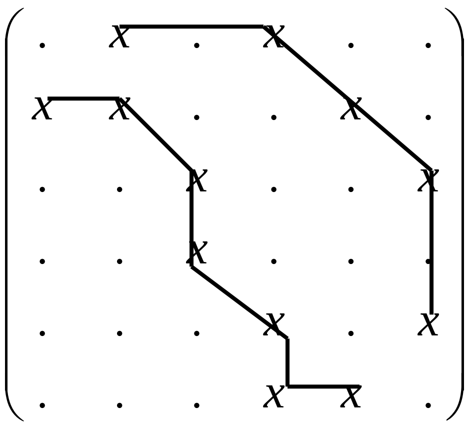


Figure 34: Two IO Stripes / Staircases

In PCMN, the number of PEs is equal to the number of staircases. In general the number of staircases is lower than the number of SRIO stripes. The maximum length of a SRIO stripe is n (maximum number of non-zero it can cover), the maximum length of a staircase is 2*n-1.

In [6] an analysis is done to compare the original idea of R. Melhem to PMCN. In their analysis they defined a “global cycle” to measure the number of “time steps”. With these definitions they prove that PCMN is “superior to the algorithm of Melhem in terms of hardware requirements, while using exactly the same number of time steps”. The quote is taken from their abstract and brings every reader in excitement. Unfortunately the definitions “global cycle” and “time steps” are very vague and are not related to any possible hardware implementation. One of the questions that immediately arise is why the increased maximum length of a staircase (twice the maximum length of a SRIO stripe) does not have any impact on the time needed to compute the SMVM. The efficiency of a PE, processing a stripe with maximum length n, is 100%. This implies that processing a staircase with maximum length of 2*n-1 cannot be processed in the same time as processing a SIO stripe of maximum length n. The dependencies that arise when a staircase covers more than one non-zero element on a row introduces extra cycles. This effect is explained with an example in chapter 4.5.1.

4.5.1. Parallel Matrix Computation Network - Example

Consider the SMVM of matrix A with vector **x**. Vector **x** contains the values A, B and C.
The computation is represented in the following picture.

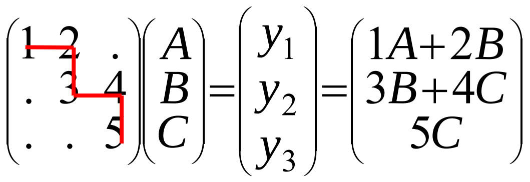

$\begin{pmatrix}1 & 2 & .\\ . & 3 & 4\\ . & . & 5\end{pmatrix}\begin{pmatrix}A\\B\\C\end{pmatrix}=\begin{pmatrix}y_1\\y_2\\y_3\end{pmatrix}=\begin{pmatrix}1A+2B\\3B+4C\\5C\end{pmatrix}$

Figure 35: Example of SMVM with staircases

The application considered in this report requires floating point representation. To achieve high performance, floating point MAC units are usually pipelined. In [7] a possible design of a 12 stage pipelined MAC unit is presented. The use of 12 pipeline stages results in a delay of 12 cycles. Without dependencies the delay does not play a role in the performance of the system.
To compute the SMVM of figure 35, one staircase is needed. In the first stage the PE starts the computation of a partial result of y_1.

Cycle 1: $y_1' = 1A + 0$

In the second stage it computes the second partial product of y_1. The partial product has to be added to the value of y_1. Thus the second stage can only start when the result of the first stage is available. Because the MAC unit has 12 pipeline stages, the second stage can not start before cycle 13.

Cycle 13: $y_1 = 2B + y_1'$

The same analysis can be done for then rest of the computations.
Stage three can immediately start because there are no dependencies.

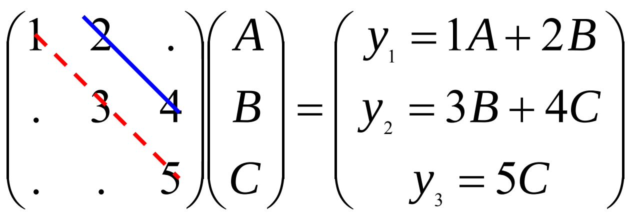
Cycle 14: $y_2' = 3B + 0$

Cycle 26: $y_2 = 4C + y_2'$

Cycle 27: $y_3 = 5C + 0$

In cycle 1 the first multiply accumulate is started and in cycle 27 the last one. In the same time the MAC unit could have started 27 multiply accumulates. For this example the efficiency of the MAC unit is only 18.5%.

4.5.2. SIO Example



$$\begin{pmatrix} 1 & 2 & . \\ . & 3 & 4 \\ . & . & 5 \end{pmatrix}\begin{pmatrix} A \\ B \\ C \end{pmatrix} = \begin{pmatrix} y_1 = 1A + 2B \\ y_2 = 3B + 4C \\ y_3 = 5C \end{pmatrix}$$

Figure 36: Example of SMVM with SIO stripes

Stripe one is defined as the solid line, stripe two is defined as the dashed line. Processing element one processes stripe one and PE two processes stripe two. A detailed description of the systolic array can be found in chapter 4.4.3.

Timing analysis of the example for PE one:
In the first clock cycle, PE one passes value A to PE two.
Cycle 1: Pass value A to next PE.
In the second clock cycle it starts the computation of a partial result of y_1.
Cycle 2: $y_1' = 2B + 0$
In the third clock cycle it starts the computation of a partial result of y_2.
Cycle 3: $y_2' = 4C + 0$
In the fourth clock cycle it passes on value y_3.
Cycle 4: Pass value y_3 to next PE.

Timing analysis of the example for PE two:
In the first stage of PE two it has to compute a partial result of y_1. PE two can only start this computation after it has received the partial result y_1' from PE one. Thus PE two cannot start before cycle 14 (2+12).
Cycle 14: $y_1 = 1A + y_1'$
The second stage a partial result of y_2 is computed. Because of the dependencies, this computation cannot start before cycle 15 (3+12).
Cycle 15: $y_2 = 3B + y_2'$
The last stage starts at cycle 16.
Cycle 16: $y_3 = 5C + 0$

To obtain the highest possible efficiency, the system continuously has to calculate SMVMs. This can be done by multiplying the matrix with multiple vectors. This optimization can be used in the overall algorithm because multiple columns have to be computed. Thus at cycle 5 PE one can start a new round to compute the SMVM with a new vector. This means that every four clock cycles a result vector is available. Within four clock cycles eight multiply accumulates could be computed by means of two PEs. For this example only five are computed. The efficiency for this example is thus 62.5%, which is a lot better than the staircase solution.

4.5.3. Conclusion

PMCN is introduced by the inventors as a better method to compute SMVM than the method of Melhem. The analysis in [6] to support their claim is very abstract and does not relate to a possible hardware implementation. PMCN and the method of Melhem are both projected onto an example in this report. PMCN needed less hardware but more clock cycles to compute the SMVM. The efficiency of PMCN was only 18.5% for this example while the solution of Melhem achieved an efficiency of 62.5%.
From the example it is illustrated that PMCN can have a lower efficiency then the method of Melhem if floating point hardware is used because of the pipelined MAC unit. In case of fixed point hardware without a pipelined MAC unit, PMCN might achieve higher efficiencies compared to the method of Melhem.

5. Alternative 1: Plans

5.1. Computing the SMVM on one PE

In chapter 2.5.1 it is already indicated that the number of MAC units that the target FPGA can hold is 31. This chapter explains the implementation of a PE and possible optimizations to give a base for a multiple PE design.

5.1.1. Plans

As explained in chapter 2.5.2 a PE is able to perform MAC instructions. This means that the complete SMVM can be executed on one PE. The following pseudo code represents a normal (not sparse) matrix vector multiplication of matrix A times vector **x** and the result is vector **y**. Matrix A has n * n elements.

```
for (int i = 1; i < n; i++) {
    y(i) = 0;
    for (int j = 1; j < n; j++) {
      y(i) = A(i,j)*x(j) + y(i);
    }
}
```

The outer for loop loops over the rows and the inner for loop loops over the columns. In case of a sparse matrix the inner loop is modified such that the loop is only over the non-zero elements.
The problem with this schedule is that the pipeline stages of the MAC unit will result in a delay because of the dependencies of value y(i). For the following example the number of pipeline stages of the MAC unit is assumed to be one.

$$\begin{pmatrix} 1 & 2 & . \\ . & 3 & 4 \\ . & . & 5 \end{pmatrix}\begin{pmatrix} A \\ B \\ C \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 1A+2B \\ 3B+4C \\ 5C \end{pmatrix}$$

Figure 37: Example SMVM

In cycle one the PE can start the computation of y_1'=1A+0. The problem is that the PE cannot start the computation of y_1=2B+ y_1' at cycle two because y_1' is not yet available because of the pipeline latency. The PE can only start at cycle three. Thus for the first row, one MAC slot is not used. In [13] a method is described to use the MAC unit more efficiently. This is done by computing multiple results at the same time. For this particular example at cycle one the PE can start the computation of y_1'=1A+0, at cycle two y_2'=3B+y_2, at cycle three y_1=2B+ y_1', at cycle four y_2=4C+ y_2' and at cycle five y_3=5C+y_3. With this schedule the utilization of the MAC unit is 100%. This optimization is also known as loop unrolling.
The following pseudo code represents this scheme for dense matrices:

```
for (int i = 1; i < n; i=i+2) {
    y(i) = 0; y(i+1) = 0;
    for (int j = 1; j < n; j++) {
      y(i) = A(i,j)*x(j) + y(i);           //Cycle one of MAC unit
      y(i+1) = A(i+1,j)*x(j) + y(i+1);     //Cycle two of MAC unit
    }
}
```

Incase of sparse matrices not every row has the same number of non-zeros. Computing consecutive rows in parallel will not result in an efficient schedule. A near optimal schedule can be achieved by computing the longest rows (most non-zero elements) first. In case of the MAC unit proposed in [7] the adder pipeline has eight stages. This means that eight rows will be processed in parallel.

As explained before a near optimal schedule can be achieved by processing the longest rows (with the most non-zero elements) first. Further each non-zero element has a column index which must be used to index the vector and an additional boolean to indicate the completion of the computation of a row. All these variables can be seen as a plan to compute the SMVM in a near optimal way. In the FPGA this “plan” is stored in a memory. Further a PE also needs a memory with the vector **x** and a memory to store the result vector **y**.

Consider the following more complex example with an adder pipeline of four stages.

$$\begin{pmatrix} 1 & 2 & . & . & . & . \\ . & 3 & 4 & . & . & . \\ . & . & 5 & . & . & . \\ . & 6 & 7 & 8 & 9 & . \\ . & . & 10 & 11 & . & 12 \\ . & . & . & 13 & . & . \end{pmatrix} \begin{pmatrix} A \\ B \\ C \\ D \\ E \\ F \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{pmatrix} = \begin{pmatrix} 1A+2B \\ 3B+4C \\ 5C \\ 6B+7C+8D+9E \\ 10C+11D+12F \\ 13D \end{pmatrix}$$

Figure 38: Example SMVM

From the matrix and the number of pipeline stages the following plan can be constructed.

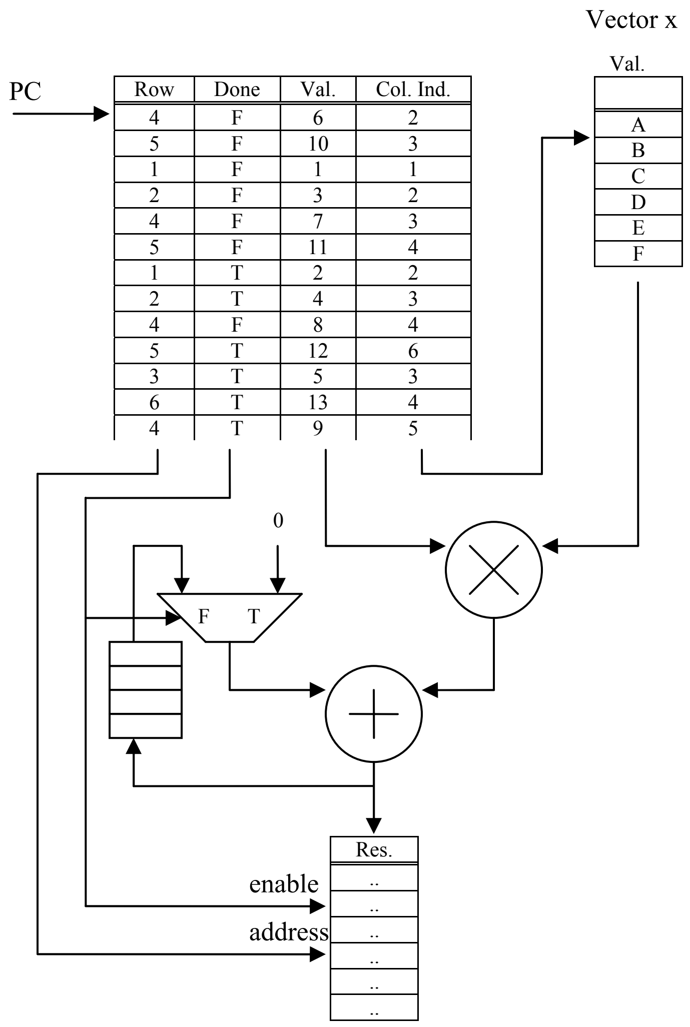
Row	Done	Val.	Col. Ind.	
4	F	6	2	← PC
5	F	10	3	
1	F	1	1	
2	F	3	2	
4	F	7	3	
5	F	11	4	
1	T	2	2	
2	T	4	3	
4	F	8	4	
5	T	12	6	
3	T	5	3	
6	T	13	4	
4	T	9	5	

Figure 39: "Plan" for the computation of SMVM of the example

For simplicity the column row is added to the plan. A near optimal schedule can be constructed by starting with the longest row. In this example row number four is the longest (most non-zero elements) so it is scheduled first. It does not matter which non-zero element is processed first. It makes sense to just start with the left most non-zero element; in this case it has the value six. The column index of the non-zero element is two. Because this is not the last non-zero element of the row, the value of done is false. Summarized, the first row of the plan has the values 4, false, 6 and 2. Because of the adder pipeline the second non-zero element of row four is processed at row five of the plan. The second row of the plan will be occupied by the first non-zero element of the second largest row, etcetera. Incase of equally long rows (same number of non-zero elements) it does not matter which one is scheduled first.
Thus before the actual SMVM is executed a preprocessing phase is needed to make “plans”.

5.1.2. PE Design

Figure 40 represents the design of the PE for execution of plans as suggested in [13].



Row	Done	Val.	Col. Ind.
4	F	6	2
5	F	10	3
1	F	1	1
2	F	3	2
4	F	7	3
5	F	11	4
1	T	2	2
2	T	4	3
4	F	8	4
5	T	12	6
3	T	5	3
6	T	13	4
4	T	9	5

Figure 40: PE design

For simplicity the four pipeline stages are placed outside the adder and implemented as four registers. The column Col. Ind. is used to index the vector **x**. The column Row is used to index the result memory.

5.1.3. Mapping of the system matrix

In the previous examples the number of adder pipeline stages was small compared to the number of adder pipeline stages of the MAC unit proposed in [7] which is eight. Simulations have shown that this did not have a great impact for the Volume Reconstruction matrix. The utilization of the MAC unit was better than 99%.

The following enumeration recapitulates the properties of the application matrix. The numbers are approximations.

- Non-zero elements: 2,5M
- Size: 138k x 138k
- Bandwidth: 22k
- Average number of non-zero elements per row: 20

The application requires 64-bit values and the number of non-zeros of the matrix is about 2,5M, this means that the storage of one plan for the complete SMVM requires at least 2,5M * 64 = 160 Mbit. The largest Virtex-II pro FPGA, the XC2VP100, has only 7,992 Kbit. Thus to compute the sparse matrix vector multiplication, the original plan has to be split into multiple smaller plans. On average there are twenty non-zero elements on each row. If the complete memory of the FPGA would be used to store only the values of a plan than approximately (7,992*1024)/(64*20) ≈ 6400 rows could be covered. The matrix has approximately 138k rows. To cover all the 138 rows of the matrix, at least ⌈(138k/6400) = 22 plans are required and hence 22 FPGAs. Besides the storage of the values of the matrix, the values of the vector must be stored, the result values and indices to index the memories. The value of 22 is a lower bound on the total number of plans.

The complete storage of the vector **x** would require 138k*64 ≈ 8,700 Kbit, which is more than the largest Virtex-II pro has available as memory. A plan that covers the complete matrix requires the complete storage of the vector **x**. The conclusion from the previous paragraph was that the coverage with one plan is not possible because of the limited amount of memory of current FPGAs. Several smaller plans are required to cover the matrix. A plan can be defined as a strategy to compute a part of the SMVM by one PE such that the storage does not exceed the local memory of the PE.

Construction of a plan can be done is several ways. If the rows a plan covers can be any row of the matrix, every plan requires storage of the complete vector **x**, which is not possible. A better idea is that a plan only covers consecutive rows. The advantage is that only a part of the vector **x** has to be stored.

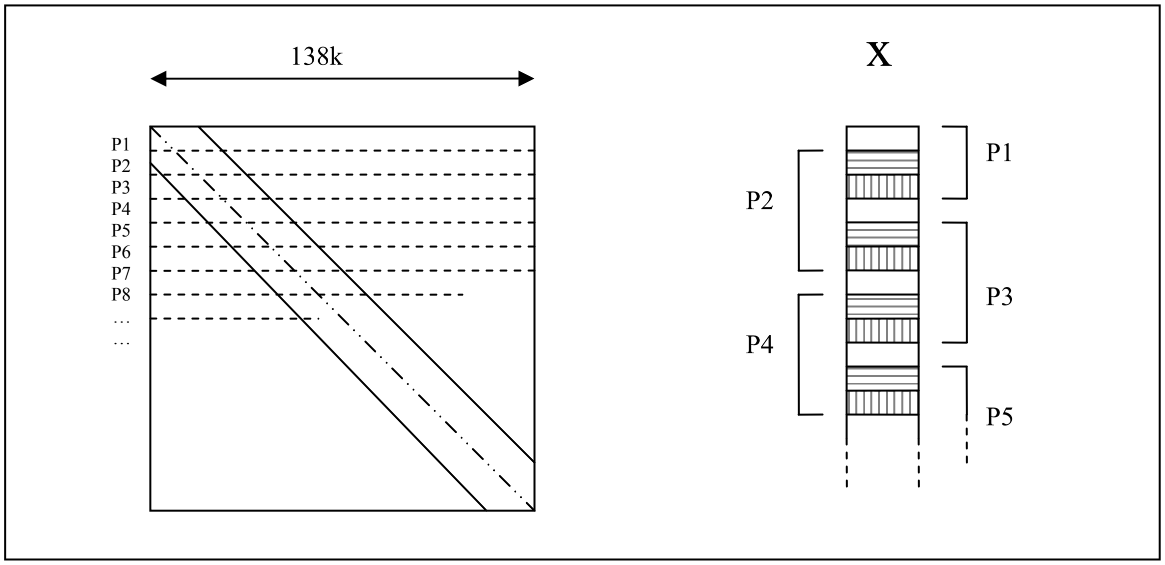


Figure 41: Dividing the matrix into "plans"

The bandwidth of the matrix is approximately 22k elements. Each plan can index the number of rows it covers plus the size of the bandwidth. The upper bound on the number of rows a plan covers is 6400. This would require (6400+22k)*64 ≈ 1818 Kbit of storage for the vector **x** for each plan.
Until this point all the computations where performed with an upper bound on the number of rows a plan could cover on average. The computations where only based on the storage of the values in the plan. To index the complete vector **x**, 18 bits are required. But a plan only has to index a part of the vector **x**. The upper bound on the number of elements to index is 6400+22k ≈ 28.4K, which can be indexed with 15 bits.

For the boolean value Done, one bit is required. In a real implementation the column row is not present. Instead an additional memory is used to index the results, which is discussed later. For each non-zero element 64+15+1 = 80 bits are required. The number of results is equal to the number of rows a plan covers. The upper bound is 6400*13 ≈ 82 Kbits.
The memory to index the results will be kept constant; the other parts will be variable with the number of rows. Solving the following formula gives a more realistic idea on the average number of rows a plan covers.
On average each row has twenty non-zero elements, for each non-zero element 80 bits are needed. The size of a plan is thus the number of rows times the average number of non-zeros times 80 bits.

$$S_{plan} = N_{rows} \times \mu_{elements} \times r_{bits}$$
$$S_{plan} = N_{rows} \times 20 \times 80$$

The number of elements for the storage of a part of vector x is the number of rows a plan covers plus the bandwidth.

$$S_{vec\_x} = (N_{rows} + 22k) \times 64$$
$$S_{result} \approx 82 \times 1024$$
$$S_{FPGA} = 7{,}992 \times 1024$$
$$S_{FPGA} \geq S_{plan} + S_{vec\_x} + S_{result}$$
$$S_{FPGA} - S_{result} \geq 64N_{rows} + 1600N_{rows} + 22k \times 64$$
$$N_{rows} \approx 4000$$
$$N_{plans} = \lceil 138k / N_{rows} \rceil = 35$$

To compute the complete SMVM on one PE multiple plans are needed. The memories in FPGAs are often implemented as true dual port. This means that there can be two processes reading or writing at the same time if not on the same address. The PE executes a plan in order, thus after it has processed a non-zero element it can be overwritten by a non-zero element of the next plan. Suppose a PE processes a non-zero element every clock cycle and every clock cycle a new non-zero element can be loaded into memory. If this is the case than loading a plan takes the same amount of time as executing a plan. Switching from one plan to another does not require extra clock cycles.
A plan requires a certain part of the vector **x** available in memory. Suppose a PE is executing plan p_d. To execute this plan part vec_d of the vector **x** has to be in memory. After the PE has executed the last non-zero element of plan p_d it switches to plan p_{d+1}. Plan p_{d+1} requires part vec_{d+1} of the vector **x** to be in memory. To guarantee that switching to another plan does not take any additional clock cycles the parts vec_d and vec_{d+1} must be in memory at the same time. The size of vec_d is about 26k elements. The total length of the vector **x** is 138k elements.
The size vec_d is thus 10% of the size of the vector **x**. The total number of parts of the vector **x** is equal to the number of plans which is 35. The conclusion is that the part vec_d has overlap with the previous part (vec_{d-1}) and the next part (vec_{d+1}).
For this example the number of rows a plan covers is 4k with on average 20 non-zero elements on each row. Further there are three external memories, one holding the plans, another holding the vector **x** and one to store the result vector. There are three memory interfaces to the three external memories. One memory interface is used to load plans into the internal memory of the FPGA by overwriting the already loaded plan. Another memory interface will be used to store the results into external memory and the last memory interface is used to load parts of the vector x.

Suppose the first plan p_1 and vec_1 are loaded into memory. The PE starts executing the first plan. While the PE is processing plan p_1 plan p_2 will be loaded by overwriting the non-zero elements that already have been processed. At the same time vec_2 has to be loaded into memory. Because of the overlap with vec_1 it is sufficient to only load the part that isn't already in memory.

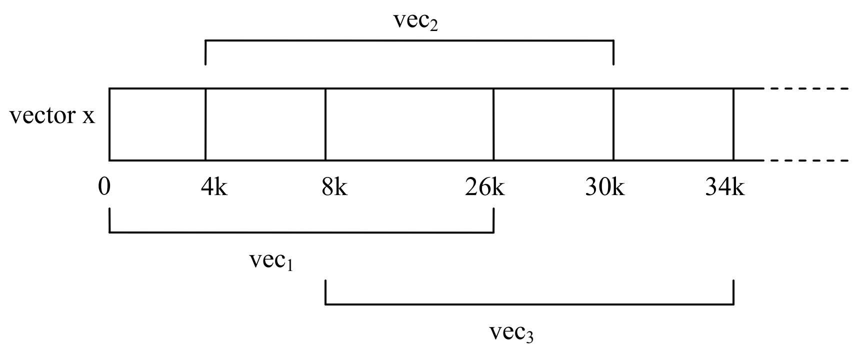


Figure 42: Dividing vector x into parts

In the time the PE is executing plan p_1, 4k elements of vec_2 have to be loaded into memory. The number of non-zeros of a plan is equal to 20*4k = 80k. Thus the memory interface for loading new plans must have a higher bandwidth than the memory interface used for loading parts of vector **x**. The difference is a factor twenty.

5.1.4. FPGA implementation

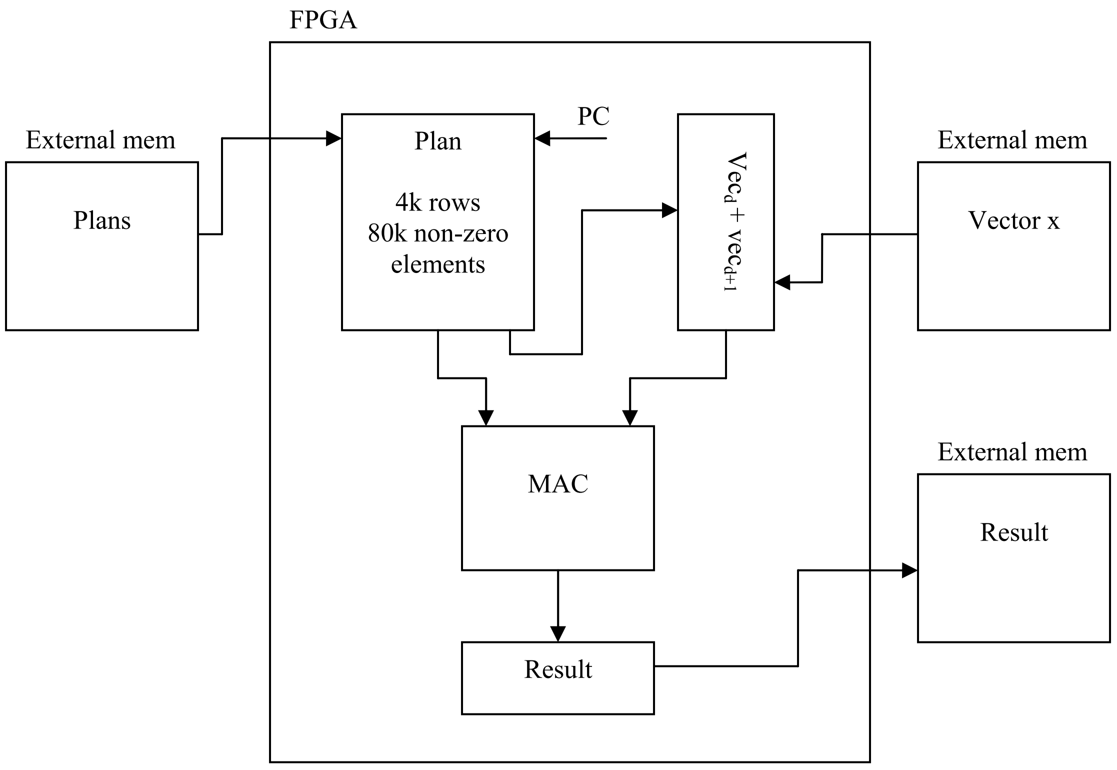


Figure 43: FPGA implementation

The memory bandwidth requirements for this system are very high. The question is if this system could be optimized such that the utilization of the PE stays at 100% while using less memory bandwidth.

5.1.5. Possible optimization

In the above examples the SMVM used only one vector. Each non-zero element is processed only once. A plan consisted of 80k non-zero elements that could index 26k elements of the vector **x**. This implies that elements of the vector **x** are indexed more than once.
Loading data from external memory is very costly. The idea is thus to use the data that is already on chip as much as possible. The size of a plan is very large compared to the size of the vector. Executing a plan multiple times would decrease memory bandwidth requirements. This can be done if there are multiple vectors that must be multiplied with the same matrix.
Suppose there are five vectors that must be multiplied with the same matrix. In the first run the PE executes plan p_1 with vector one, in the second run it executes p_1 with vector two, in the third run it executes p_1 with vector three, etcetera. Every non-zero element is now used five times. The memory bandwidth requirement for loading a new plan is now an fifth of the original memory bandwidth requirement, while the other memory bandwidth requirements remain the same. Figure 44 gives a schematic view of such system.

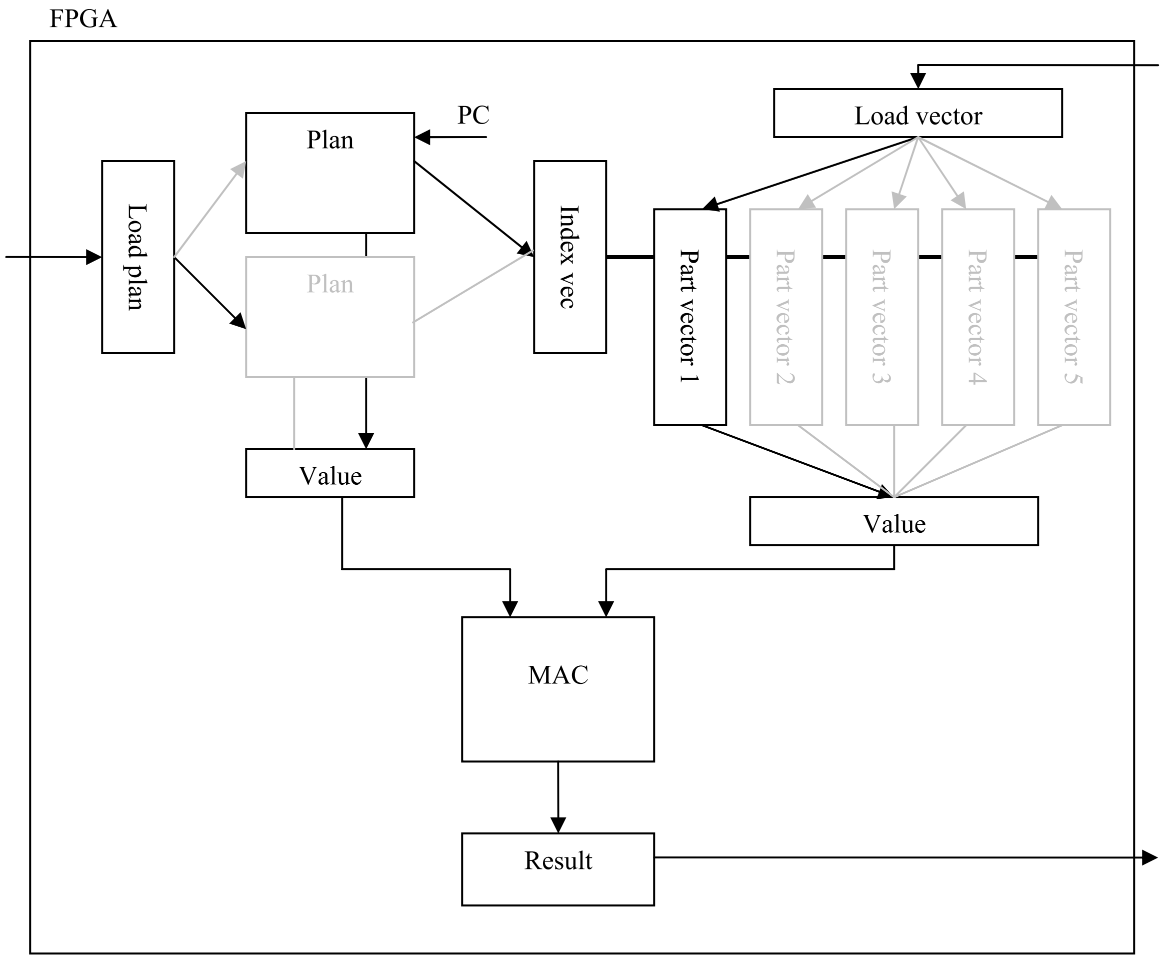


Figure 44: FPGA implementation with multiple vectors

In the original scheme where the SMVM was only executed with only one vector, there was only one memory to hold the plan. This was possible because the new plan could directly overwrite the old plan with the same speed the PE executed. This requires a high memory bandwidth. The advantage of multiplying with multiple vectors in parallel is that the bandwidth requirements are lower. For this particular example the time to load a new plan may be five times longer than the time the PE would execute the plan. However a new plan cannot overwrite the plan the PE is executing. A solution is to use two memories of the same size that alternate their function. The memories can either be used by the PE to execute the plan or it can be used to store the new plan from external memory.
The gray boxes in figure 44 are not active for the execution of the plan.

The consequence of the additional memories is that there is less memory available for a plan. The number of plans will increase and the number of rows a plan covers will decrease.

$$S_{2\times plan} = 2 \times N_{rows} \times 20 \times 80$$

$$S_{vec\_x} = (2 \times N_{rows} + 22k) \times 64$$

$$S_{5\times vec\_x} = 5 \times S_{vec\_x}$$

$$S_{result} \approx 82 \times 1024$$

$$S_{FPGA} = 7{,}992 \times 1024$$

$$S_{FPGA} \geq S_{2\times plan} + S_{5\times vec\_x} + S_{result}$$

$$S_{FPGA} - S_{result} \geq 640 N_{rows} + 3200 N_{rows} + 110k \times 64$$
$$N_{rows} \approx 276$$
$$N_{plans} = \lceil 138k / N_{rows} \rceil = 500$$

This chapter explained an algorithm to compute the SMVM on one PE. The PE was defined as a unit which had only one MAC unit to perform computations. From this definition an algorithm was proposed which achieved a very high utilization of the MAC unit with a limited requirement on the memory bandwidth. The utilization of the MAC unit was not addressed formally because simulations have shown that with a relative high number of rows (in the order of 1000) the lower bound of the utilization was 99%.
This chapter explains how a PE computes parts of the SMVM, the memory bandwidth interfaces required and an optimization for the memory bandwidth. This is done to give a base for a design with multiple PEs.

5.2. Computing the SMVM on Multiple PEs

The largest Virtex-II pro FPGA can hold 31 MAC units, which run at 170 MHz [7]. The peak performance of this FPGA with this type of MAC unit is 31*170M*2 = 10,5 MFLOPS (double precision). In the previous chapter only one MAC-unit was used which means that at most 1/31 of the peak performance of the FPGA was used. Using more MAC units might achieve better results. In this chapter as well as the previous, the PEs are specified to only have one MAC unit to perform computations.

5.2.1. Introduction

In the previous chapter the idea of plans was explained. A PE executes a plan on a vector or multiple vectors, loads new plans and loads elements of vectors. Figure 43 is taken as a base for a multiple PE design.
Assume a design with four PEs that all execute plans and there are 200 plans to compute the SMVM. The first fifty plans could be assigned to the first PE, plans 51 till 100 can be assigned to the second PE, etcetera.

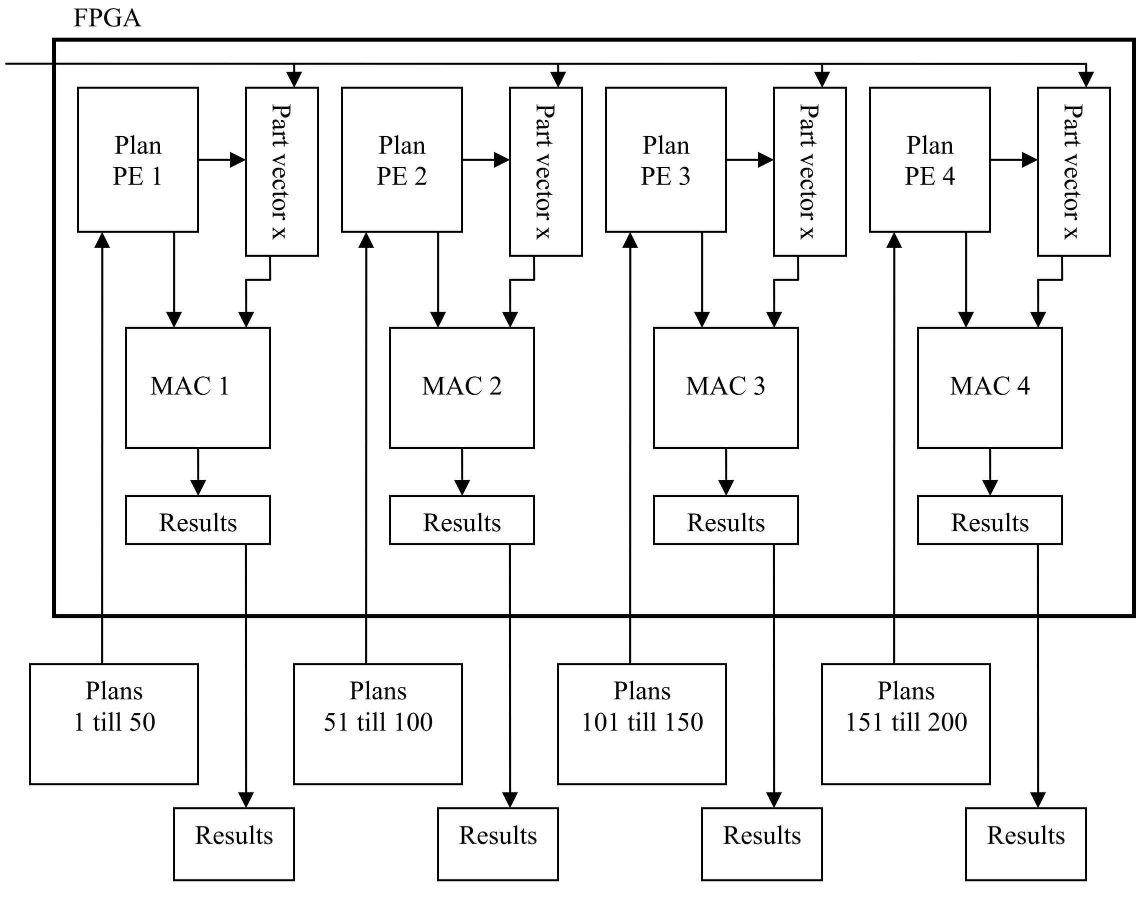


Figure 45: Possible four PE implementation

For every additional PE, two additional memory interfaces are required. But this isn't the largest disadvantage. To store a part of the vector x, at least 22k elements must be stored because of the bandwidth of the matrix. These elements are represented with 64 bits, 22k*64 ≈ 1400 Kbits. This is already 20% of the available memory of the FPGA. Thus at most five PEs could be used with this scheme. An optimization would be to only store the elements of the vector **x** that are really going to be used by a plan. Table 4 shows that not all the elements of a part of the vector are really used.

Number of rows a plan covers	Number of elements of vector **x** for a plan	Avg. Number of elements of vector **x** really used	Percentage	Max number of elements really used by a plan
32	32+22k	232	1.1%	1080
64	64+22k	402	1.8%	1698
128	128+22k	707	3.2%	2468
256	256+22k	1222	5.5%	3235
512	512+22k	2088	9.0%	3799

Table 4: Element usage of the Volume Reconstruction matrix

In the preprocessing phase where the plans are made, also the indexes of the elements of the vector **x** that are needed by a plan could be saved. Thus if a plan covers for example 128 rows, it covers on average 128*20 = 2560 non-zero elements and needs on average 707 elements of the vector **x**. For each plan the indexes of the 707 elements can be stored. Loading a new plan means now loading a plan and loading the values of the vector **x** needed by the plan.
Switching from one plan to another should not cost any additional cycles. Thus two memories are needed for the storage of the vector **x**. Both memories alternate their function; a memory is either used to load new values or to provide the PE with the value indexed by the plan. In the ideal case executing a plan should take longer than loading new elements. For the example above where a plan covers 128 rows, executing the plan takes at least 2560 (128*20) clock cycles. A realistic assumption about the memory bandwidth is that every clock cycle an element of the vector could be loaded. With that assumption one memory interface has sufficient bandwidth to support three PEs. The aim is to use as much MAC units as possible; the maximum number of MAC units that could be implemented is 31 [7]. This would require $\lceil(31/3) = 11$ memory interfaces for loading elements of the vector **x**.
An element of the vector **x** may be used in several plans. Plans that use the same element of the vector lie often next to each other. If the PEs share a bus that streams the elements of the vector **x,** each PE can copy an element into its own local memory when it needs that element. Suppose there is a system with four PEs, taking advantage of the shared bus means that they have to execute plans close together. For example PE 1 will execute plan 1, PE 2 will execute plan 2, etcetera. Thus in the first run the first four plans are mapped onto the four PEs. In the second run plan 5 till 8 will be mapped, etcetera. Suppose each PE executes plans each covering 128 rows. If the PEs do not share a bus, on average 4*707 = 2828 elements of the vector x must be loaded for each run. In case of the shared bus it is on average 2088 elements (4*128 = 512), an improvement of 26%. Eight PEs executing plans that cover 32 rows, results in loading 8*232 = 1856 elements without a shared bus or loading 1222 elements with a shared bus. This is an improvement of 34%. This effect is illustrated in Table 5.

PEs	Rows per Plan	Without shared bus	With shared bus	Improvement
4	128	2828	2088	26%
8	32	1856	1222	34%
8	64	3216	2088	35%
16	32	3712	2088	44%

Table 5: Improvement shared bus

Summarized, each PE has two small local memories for storage of elements of the vector **x**, the memories alternate their function, a memory is either used by the PE to execute the plan or to store new elements of the vector **x**. Further the PEs share a bus for loading the elements.

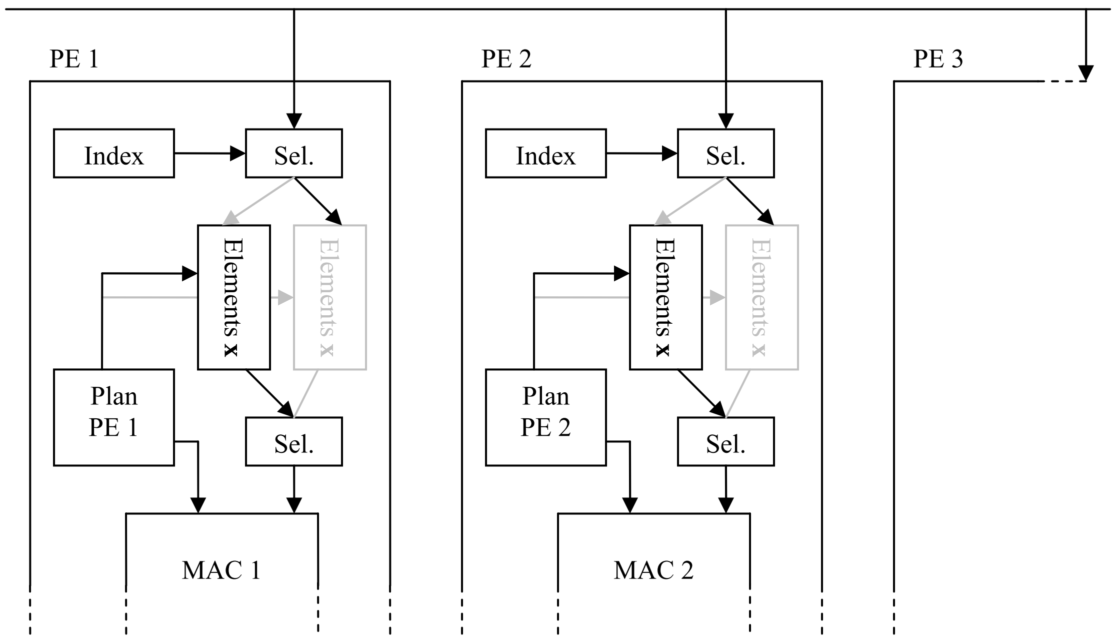


Figure 46: Vector optimization

The index block is a memory that is used to indicate whether an element on the shared bus must be stored or not, this depends on the plan. The values of the index memory are determined by the plan.

As already explained in chapter 5.1 where the SMVM is computed with only one PE, it is not efficient to execute a plan only once. It is better to execute a plan on multiple vectors. In chapter 5.1.5 a construction is proposed to load a plan with a lower speed than a PE would execute it. For multiple PEs this advantage can be used to load new plans for all the PEs with only one memory interface. This requires two memories for the storage of the plans plus two memories for the storage of Index values.

The results of the PEs are not stored in consecutive order. The order in which the results are available is completely random within the rows a plan covers. In the preprocessing phase the order is known. This information can be used by a PE to store its results in a memory. After the PE has computed all the results, the result memory can be read to get the results in order. Retrieving the results can be done while the PE is computing new results if there are two result memories. Storing the results from the local memories of the PEs to the external memory might be done with one memory interface. The resulting design in depicted in figure 48.

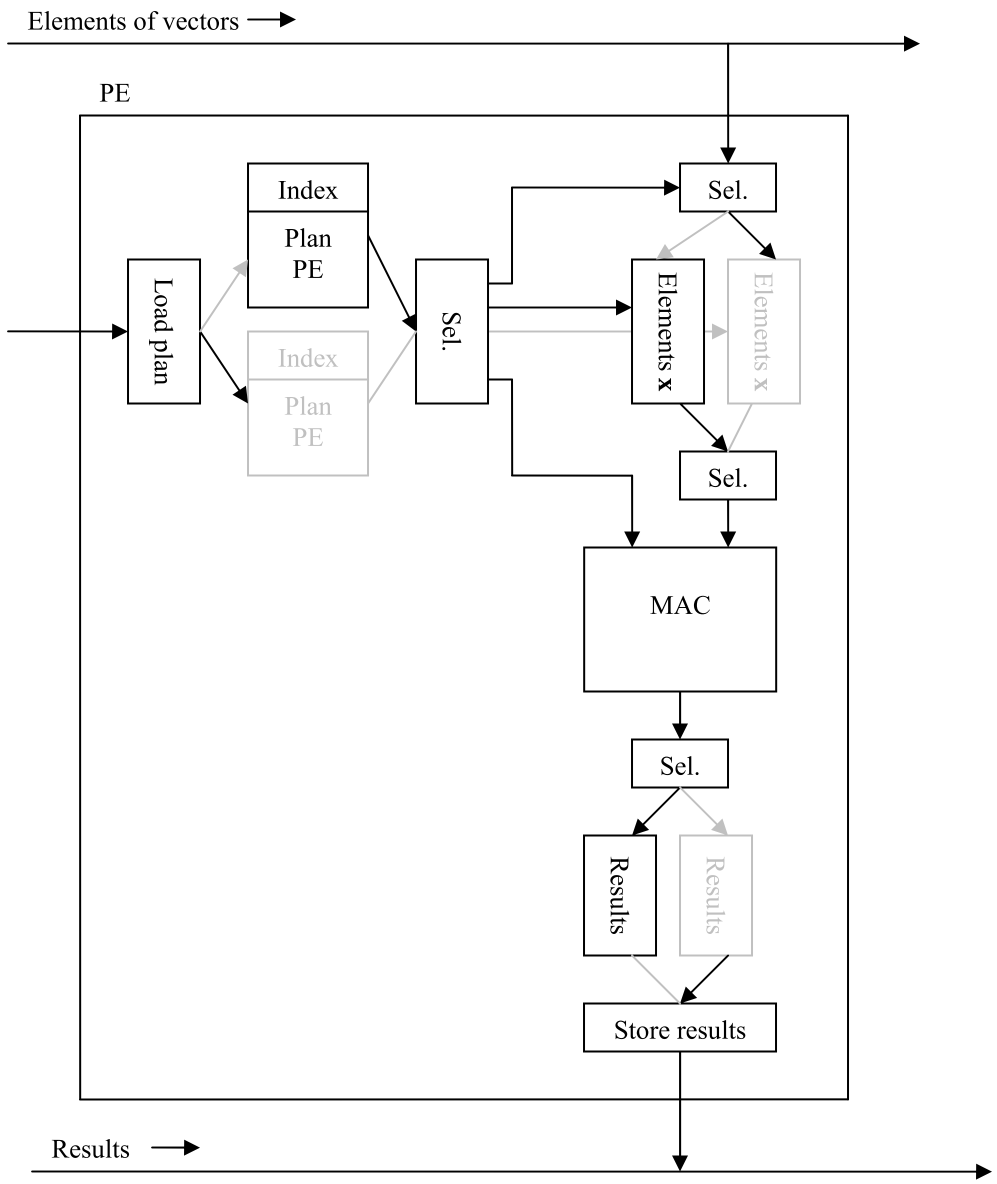


Figure 47: Optimized PE implementation

5.2.2. FPGA implementation

This section illustrates the architecture for a complete SMVM design with three PEs.

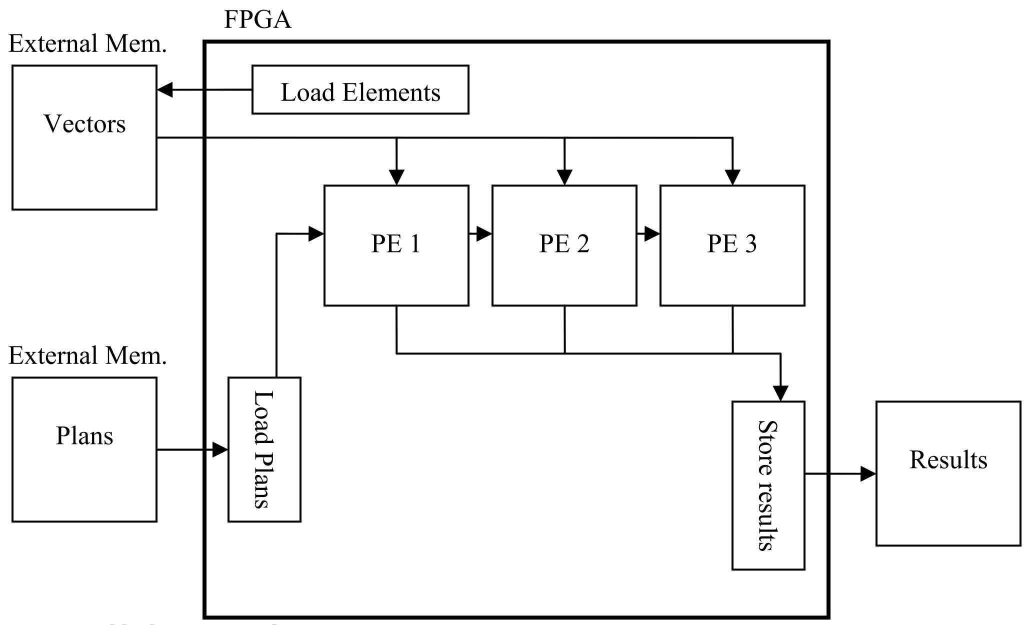


Figure 48: Possible three PEs implementation

This design uses three memory interfaces, one to load the plans, one to load elements of vectors and one to store the results. The number of PEs determines the bandwidth requirements of the memory interfaces. Having more PEs means a larger vector to be loaded and more results per clock cycle.
The plans are executed in parallel. For this example the plans are executed in blocks of three. For this block the definition super plan will be used. A super plan is defined as a collection of plans that are executed in parallel. For each super plan, elements of the vector must be loaded from external memory and the results are stored in external memory. Executing a super plan means that a number of plans are executed in parallel on multiple vectors. While a super plan is executed, a new super plan is loaded. Loading a super plan is loading the plans of the super plan into the PEs, loading new indexes and the rest of the information required for the execution of the new super plan.

The bottleneck of a SMVM is always the memory bandwidth. The proposed algorithm has some optimizations to use the available memory bandwidth more efficiently. It is also scalable in terms of more memory interfaces. The following scheme gives an idea how to use four memory interfaces.

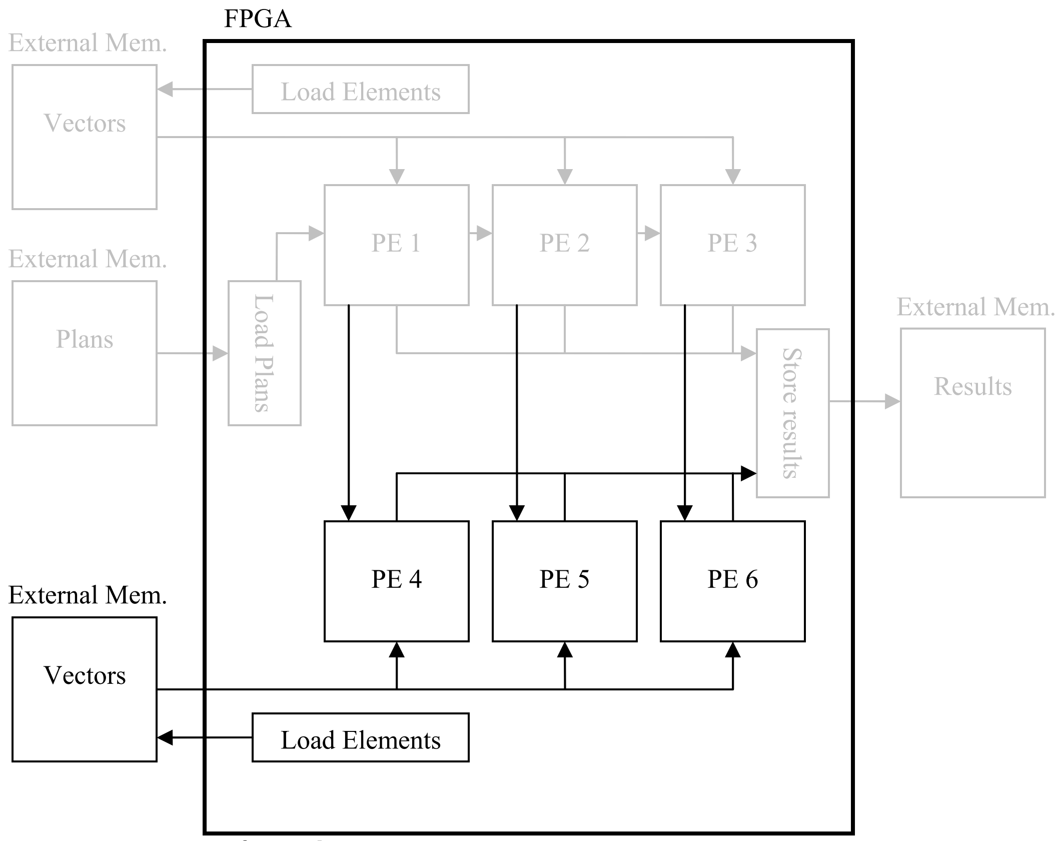


Figure 49: Four memory interface implementation

The gray colored parts are the parts that where also present in the design with three memory interfaces, the black colored parts are new. In this example PE 4 till PE 6 will execute the plans of PE 1 till PE 3 on multiple other vectors. By sharing plans between PEs the memories of the FPGA are used more efficiently.

5.2.3. Conclusion

To get the most out of the FPGA the following optimizations were proposed:

Optimization	Result
Only store elements of vector really needed	Efficient use of memory Needed to implement more than 5 PEs (see section 5.2.1)
Using a shared bus to load elements of vector	Lowers the requirements of the memory bandwidth
Multiplying with multiple vectors	Lowers the requirements of the memory bandwidth
More memory interfaces	Using memory more efficiently

Table 6: Proposed optimizations

Now the complete design is specified a few definitions are made to explain the preprocessing phase more easily:

- A **global vector** is the vector of the SMVM.
- A **local plan** is a schedule used to achieve high utilizations of a MAC unit. A local plan is executed by one PE. For each local plan there is a local vector.
- The **local vector** contains only the elements of the global vector required by a local plan.
- A **super plan** is a collection of local plans that are executed in parallel on multiple PEs. Each super plan has a super vector.
- A **super vector** contains all the elements of the collection of local vectors.
- A local vector is thus a subset of a super vector and a super vector is a subset of a global vector.

The proposed solutions require an additional preprocessing task. In the next chapter the preprocessing phase is discussed.

5.3. Preprocessing

In the preprocessing phase the plans, super plans, relative indices of elements, etcetera, have to be computed. These factors all depend on the design that is implemented. The preprocessor needs to know the number of PEs implemented, the size of the local memories, the number of memory interfaces and the memory bandwidths.
The number of memory interfaces and the size of the memories of the PEs determine the design. On beforehand no optimal size of the local memories can be determined. The optimal size highly depends on the matrix of the SMVM.
Only the non-zero elements of the matrix are stored. This is not only an advantage for the storage but also for the preprocessor. The preprocessor only has to loop over the non-zero elements. The number of non-zero elements of the matrix of this project is about 2.5M. A sparse matrix is usually stored in the CSR format but for the examples a simplified CSR like storage scheme will be used which will be referred as SR (Sparse Row) format. For each non-zero element the column and row index will be stored.

The goal of the preprocessor is to assigns as many rows of the matrix to a PE as possible. The number of PEs, the size of the memories of the PEs and the relative indexing makes this a challenge.

Each PE has effectively three memories (only three of the six are used for execution of a plan at a time, see figure 47), one memory to hold the plan, one to hold the local vector and one to store the results.
Every row should fit into these memories. Thus for every row three checks have to be done.

- If a row can be added to a PE it is scheduled in the plan of the PE and column indexes are converted to a relative index for the local vector.
- If a PE cannot process more rows, the preprocessor continues with the next PE.
- If the preprocessor has completed the plan of the last PE, the indexes of the super vector are known so the relative indexes for the local vectors can be computed.

5.3.1. Example

The following example shows the steps of the preprocessor for the SMVM of figure 50.

$$\begin{pmatrix} 1 & 2 & . & . & . & . \\ . & 3 & 4 & . & . & . \\ . & . & 5 & . & . & . \\ . & 6 & 7 & 8 & 9 & . \\ . & . & 10 & 11 & . & 12 \\ . & . & . & 13 & . & . \end{pmatrix} \begin{pmatrix} A \\ B \\ C \\ D \\ E \\ F \end{pmatrix} = \begin{pmatrix} y_1 = 1A + 2B \\ y_2 = 3B + 4C \\ y_3 = 5C \\ y_4 = 6B + 7C + 8D + 9E \\ y_5 = 10C + 11D + 12F \\ y_6 = 13D \end{pmatrix}$$

Figure 50: Example of a SMVM

Storage of the matrix in a simple form of the CSR format:

$$\begin{array}{r|ccccccccccccc} \text{Value:} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\ \text{Column:} & 1 & 2 & 2 & 3 & 3 & 2 & 3 & 4 & 5 & 3 & 4 & 6 & 4 \\ \text{Row:} & 1 & 1 & 2 & 2 & 3 & 4 & 4 & 4 & 4 & 5 & 5 & 5 & 6 \end{array}$$

Figure 51: Storage of the example in the SR format

Suppose the properties of the implemented system are as follows:
The system has three PEs. Each PE can hold four local vector values (four values of A till F), four non-zeros of matrix A and can store two results. There are no restrictions on the number of elements of the super vector.
For each step the preprocessor takes, the values of the memories are shown as well as the super vector and relative indexes of the local vector.

The preprocessor starts by assigning as many non-zero elements as possible to the first PE. It does this row by row. For this example row one has two non-zero elements and is assigned to PE 1.

PE 1:

Non-zero element:	Relative index local vector memory:
1	1
2	2

Index of element global vector
1 (Value A)
2 (Value B)

Relative index result memory
1

Super vector: Index elements global vector: 1, 2

The preprocessor tries to assign the second row to PE 1. PE 1 still has enough memory to store two non-zero elements and two vector elements thus row two can be assigned to PE 1. Notice that the assignment of row two to PE 1 results in storing only one additional element of the global vector.

PE 1:

Non-zero element:	Relative index local vector memory:
1	1
2	2
3	2
4	3

Index of element global vector
1 (Value A)
2 (Value B)
3 (Value C)

Relative index result memory
1
2

Super vector: Index elements global vector: 1, 2, 3

Because each PE can compute two rows at most, the preprocessor continues with PE 2. The third row is assigned to the second PE.

PE 2:

Non-zero element:	Relative index local vector memory:
5	1

Index of element global vector
3 (Value C)

Relative index result memory
1

Super vector: Index elements global vector: 1, 2, 3

Notice the size of the super vector does not increase because the third element was already needed by the first PE.

Row four has four non-zero elements, PE 2 only has room for three non-zero elements. Thus row four has to be assigned to PE 3.

PE 3:

Non-zero element:	Relative index local vector memory:
6	1
7	2
8	3
9	4

Index of element global vector
2 (Value B)
3 (Value C)
4 (Value D)
5 (Value E)

Relative index result memory
1

Super vector: Index elements global vector: 1, 2, 3, 4, 5

PE 3 cannot store more non-zero elements, thus it cannot process more rows. Each PE processes now a number of rows. The first round is now specified.
Because now the super vector is known, the index into the super vector can be computed.

PE 1:

Index of element global vector	Index of element super vector
1 (Value A)	1 (Value A)
2 (Value B)	2 (Value B)
3 (Value C)	3 (Value C)

PE 2:

Index of element global vector	Index of element super vector
3 (Value C)	3 (Value C)

PE 3:

Index of element global vector	Index of element super vector
2 (Value B)	2 (Value B)
3 (Value C)	3 (Value C)
4 (Value D)	4 (Value D)
5 (Value E)	5 (Value E)

The first super plan is now completed, the preprocessor continues with the second super plan.

The preprocessor assigns row five and six to PE 1.

PE 1:

Non-zero element:	Relative index local vector memory:
10	1
11	2
12	3
13	2

Index of element global vector
3 (Value C)
4 (Value D)
5 (Value E)

Relative index result memory
1
2

Super vector: Index elements global vector: 3, 4, 5

PE 1:

Index of element global vector	Index of element super vector
3 (Value C)	1 (Value C)
4 (Value D)	2 (Value D)
5 (Value E)	3 (Value E)

With the properties specified two rounds are required. In the first round rows 1 till 4 are processed and in the second round row 5 and 6.

5.4. Conclusion

One of the major disadvantages of this solution is the additional information required for the execution of the SMVM. This extra information are the indexes that indicate which elements to load for the super vector and the indexes to load the elements for the local vector. This extra information is stored in external memory; this means that memory bandwidth is used to transfer the additional information. Because the memory bandwidth is the bottleneck this is an undesired effect.
Another disadvantage is loading the elements of the super vector from external memory. This is done by loading only the required elements of the vector. The data in external memory is thus randomly accessed. The memory bandwidth of random access will be far less than the memory bandwidth of block transfers. Besides these two disadvantaged, the system is also very complex.

6. Alternative 2: Small Band Coverage

This chapter presents the design of the Small Band Coverage method. This system is a result of two major modifications of the stripe method described in chapter 4.4. The first two chapters are a short recapitulation of the stripe method.

6.1. Short Review of Stripe Method

The idea behind the stripe method is to compute the SMVM with a systolic array. This systolic array consists of multiple PEs. The complete description of the stripe method can be found in chapter 4.4.

All the non-zero elements of the matrix are covered with stripes. A PE processes one of these stripes. Figure 52 plots the region for the construction of a SIO stripe.

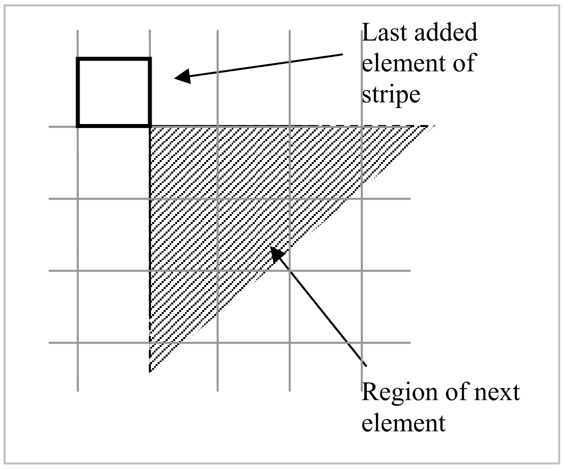


Figure 52: Region of SIO stripes

Because of the defined region, SIO stripes have the following properties:

- A stripe contains at most one element of every row.
- A stripe contains at most one element of every column.
- From the first property it follows that the longest stripe covers at most n elements.
- From the first property it follows that the elements on the same row are covered by different stripes.

The stripe method uses a number of processing elements (PEs) together forming a systolic array. Each PE is able to compute a multiply accumulate.

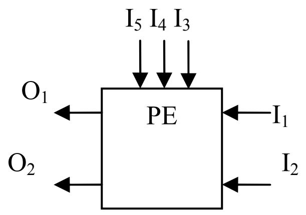


Figure 53: Processing element of the systolic array

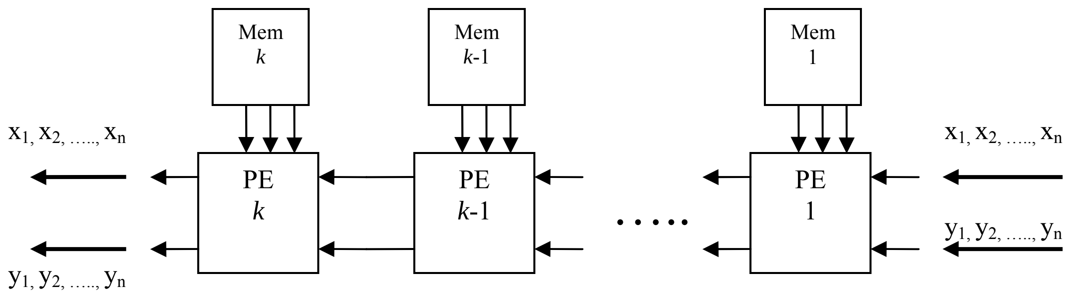


Figure 54: Systolic array for SMVM

6.1.1. Utilization of stripe method

As explained in chapter 4.4.4 the utilization of the stripe method can be very high for regular system matrices but for irregular system matrices the utilization is very low. The best case utilization varies between 1% and 80%. In chapter 4.4.5 the cause for the large differences in the utilization of the design is explained.

The “construction rules” (the region of the next element) of the stripes are the cause of the low utilization of the PEs. The rules forbid the covering of non-zero elements with a column index less or equal then the previous element covered. Without this rule the utilization could be a lot higher.

The restrictions on the construction of stripes are caused by the design of the systolic array. As can be seen in figure 54 vector **x** and result vector **y** stream through the system.
A non-zero element has a column index and row index. The column index is used to index the **x** vector and the row index is used to index the partial result vector **y**. A PE that processes a non-zero element has to receive the correct x- and partial y-value before it can start the computation. The following non-zero element processed by the PE has to have a larger column and row index than its predecessor because the PE will only receive successive elements of vector **x** and partial result vector **y**.

6.2. *Small Band Coverage*

The design of a PE could be changed such that it stores for example the last ten received elements of vector **x**. In that case a non-zero element will still have to have a row index larger than its predecessor but the column index will have to be larger than the column index of its predecessor minus ten. This modification leads to an additional region for the coverage of non-zero elements.

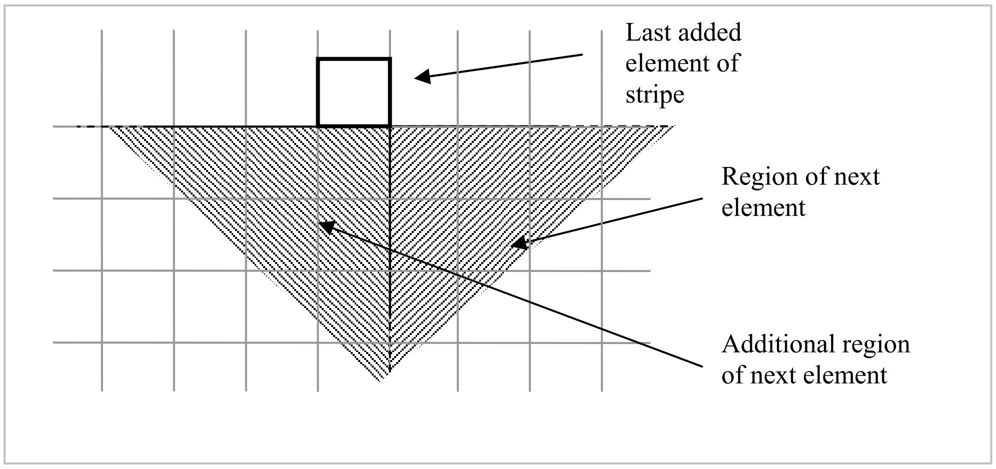

Last added element of stripe
Region of next element
Additional region of next element

Figure 55: Additional region for stripes

Between the PE there are FIFO buffers to stream vector **x** and the result vector **y**. The PEs are either a consumer or a producer for these buffers. Producing in this context can also be moving an element from the input to the output as happens with vector **x**.
In the original idea of stripes a PE may block one of the streams (result vector **y** or vector **x**) a number of clock cycles.

Suppose a PE processes a non-zero element with a column index of 60 and a row index of 32. The PE has to wait at least 60 cycles before it receives x_{60}. In the mean time the PE might block the y-stream if it already received y_{32}'. The result is that the following PE will not receive y_{32}' before cycle 61. In the best case the time to compute the SMVM would be the time to stream the result vector **y** without being blocked by a PE. In the worst case it takes twice this time because a PE can only block one of the streams at a time. The amount of blocking also determines the buffer size between the PEs.

An additional modification on the original stripe method would be that every PE has to consume and produce one element of vector **x** and one element of vector **y** every clock cycle. With this modification the computation of the SMVM takes the same amount of time it takes to stream the result vector **y**. Because of this modification a PE could only compute a straight diagonal of the matrix which would result in very low utilizations for irregular system matrices. Combining this modification with the proposed modification of the PE (saving the last received elements of vector **x**) stretches the straight diagonal from one element to a band of multiple elements wide while guaranteeing a fixes latency for the SMVM.

6.3. Design of PE

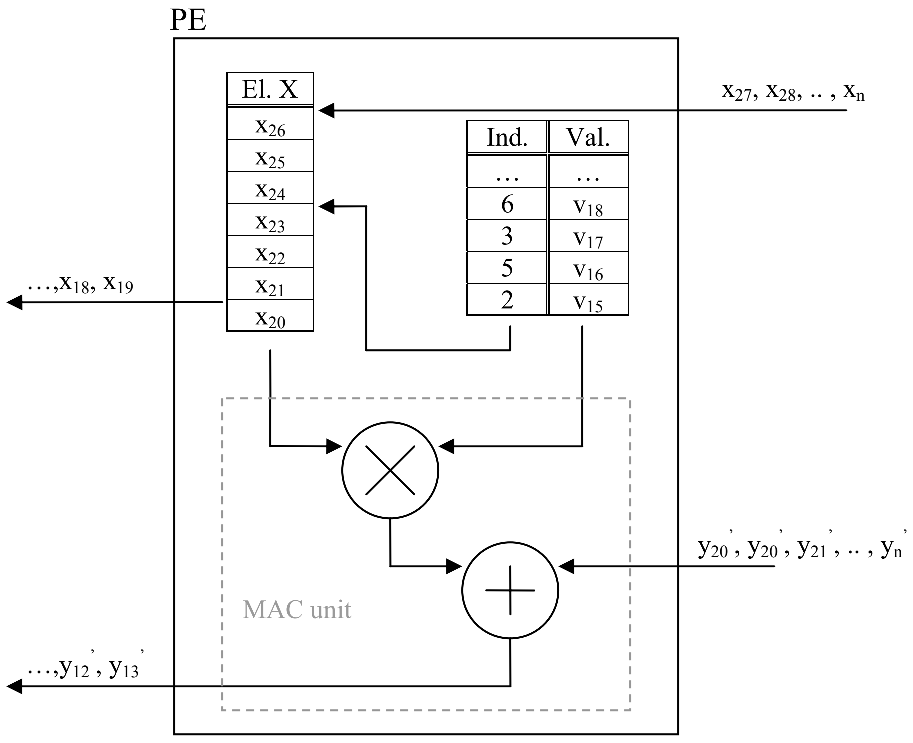


Figure 56: Design of PE

The main components of the processing element are the MAC unit, the FIFO buffers containing the values (Val.) and column indices (Ind.) of the non-zero elements and a memory that holds elements of the vector x (El. X).
The idea is that elements of the vector x stream trough the PEs. At the start of a new row (index of partial result is one higher) an element of the vector **x** is loaded into memory and an element is loaded from memory and passed to the next PE. With dual port memory this operation can be executed in one clock cycle.
The column indices are used to load x-values from memory to multiply these values with the values of the non-zero elements. If vector x is streamed, there would be three processes that access the same memory. A better solution would be to stream the vector x using two cycles, in the first cycle write a new value and in the second cycle read and pass a value to the next PE. During these two cycles the memory is still available for the process that schedules the multiplications of the non-zero elements.

6.4. PE band coverage

As mentioned, the modifications (every cycle consume and produce one element for both streams and save the last received elements of the vector **x)** results in limitations and opportunities for the coverage of the non-zero elements for a PE. The non-zero elements a PE can process lie in a band. A band has two properties, namely the width and the position. The width of the band is equal to the number of saved elements of vector **x**, while the position indicates the highest diagonal of the band. Incase of the situation of figure 57 the difference is zero with a bandwidth of six, which means all non-zero elements on the main diagonal plus the five diagonals below the main diagonal could be processed. This situation is presented in figure 58.

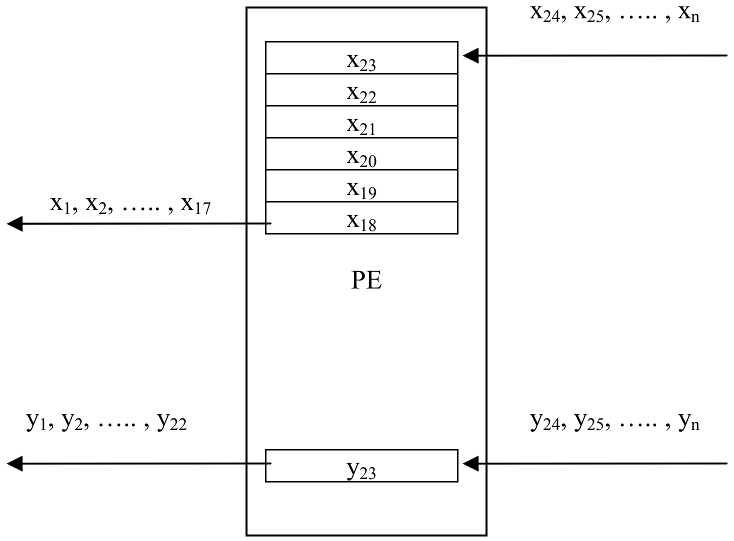

$x_{24}, x_{25}, \ldots, x_n$
x_{23}
x_{22}
x_{21}
x_{20}
x_{19}
x_{18}
$x_1, x_2, \ldots, x_{17}$
PE
$y_1, y_2, \ldots, y_{22}$
$y_{24}, y_{25}, \ldots, y_n$
y_{23}

Figure 57: Processing element saving last six elements of vector x

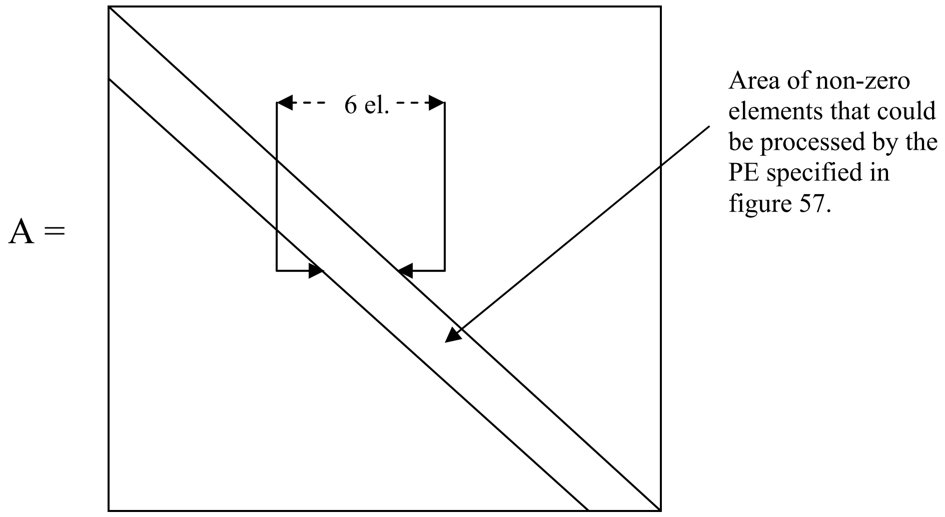

A =
6 el.
Area of non-zero elements that could be processed by the PE specified in figure 57.

Figure 58: Covered area of PE with position zero and bandwidth six

With the stripes there were restrictions for the coverage of the non-zero elements. At this point it may be clear that there are (still) two restrictions on the coverage of the non-zero elements for the new design.

- A PE may only process one non-zero element per row.
- The non-zero elements must lie in the covered area (band) of the PE.

6.5. SMVM with Small Band Coverage

The systolic array composed with the new PEs should cover the band of the matrix completely to compute the SMVM. A straightforward but resource (in terms of memory usage) inefficient solution would be that every PE covers the complete band of the matrix. A more resource efficient solution divides the large band into smaller bands. Each small band than has to be covered by at least one PE. The size and positions of these small bands depend on the matrix of the SMVM.

The bands of the PEs do not have to lie exactly next to each other. They may overlap; it could be such that the first five PEs cover the same band or that the band of PE 6 has some overlap with the band of PE 5.

6.5.1. Example

As an example, suppose the SMVM of a matrix A has to be computed with seven PEs. The band of the matrix may be divided over the PEs as indicated by figure 59 and figure 60.

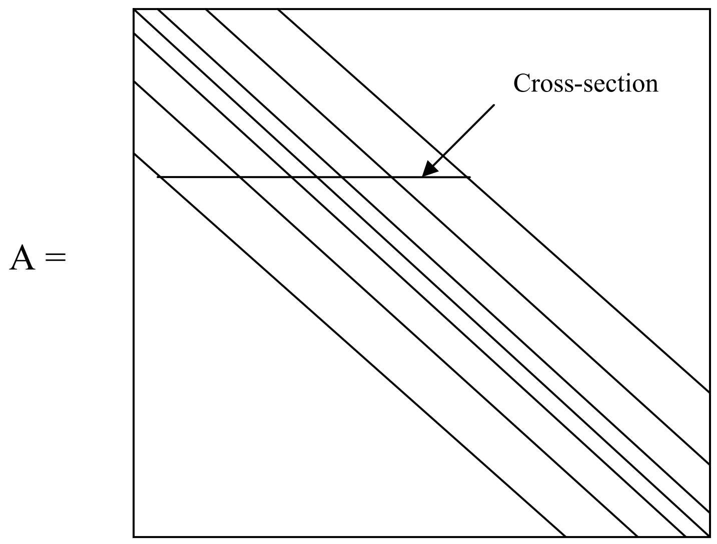


Figure 59: Dividing large band into smaller bands

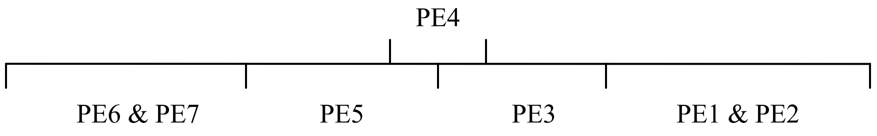


Figure 60: Cross-section of band defined at figure 59.

The system can process at most seven elements per row because a PE may only process one element per row. Because the bandwidth is divided into smaller parts, the seven elements must lie withinin certain areas. The restrictions are that there must be two elements lying in the band of PE6 and PE7, two elements in the band of PE1 and PE2, and one in each band of PE3 till PE5. If these restrictions are not met, the row cannot be processed. The position and size of the band thus have to be chosen carefully. Rows with less than seven elements have similar restrictions.

6.6. Problems

As already mentioned in chapter 6.5.1 not every row can be processed with the proposed design. Rows with more non-zero elements than the number of PEs or rows with multiple non-zero elements in the same region cannot be processed. This chapter proposes a solution to solve this problem.

The reason why a PE may only process one non-zero element per row is because of the adder latency of the MAC unit. To process a second non-zero element of a row, the result of the first non-zero element must be available. If the two elements are processed after each other, the PE has to wait a number of cycles between the start of processing the first and the second element of the row. The number of cycles to wait is equal to the adder latency of the MAC unit. The MAC unit used in [7] has an adder latency of eight.

To process every row, regardless of the number and position of the non-zero elements it is necessary to give up the restriction that a PE may only process one element per row.

Suppose a system has to process a matrix, where for example row r_{46} has four non-zero elements e_{80} till e_{83}. Suppose the system consists of only two PEs, P1 and P2. The non-zero elements e_{80} and e_{81} can only be processed by P1, e_{82} and e_{83} can only be processed by P2. Suppose the number of pipeline stages is twelve for both PEs.

Element:	Column index:	Row index:
e_{80}	30	46
e_{81}	32	46
e_{82}	48	46
e_{83}	50	46

Table 7: Elements of row 46

In the normal case the results that are produced by P2 are the final values of the result vector **y**. To achieve this P1 has to send one partial result to P2. This partial result is $y_{46}' = e_{80} * x_{30} + e_{81} * x_{32}$. The PEs can only execute the instruction: A*B + C. Thus at a certain time P1 executes $y_{46}^1 = e_{80} * x_{30} + 0$. The result of this operation is available after twelve clock cycles. This result is required to compute $y_{46}^2 = e_{81} * x_{32} + y_{46}^1$. Thus the two operations cannot be executed after each other. This dependency causes a low utilization of the PE if the time between the dependent operations is not used.
A possible solution might be to schedule operations of other rows between dependent operations. This scheduling is probably hard to accomplish and implement.
Another solution is that the system does not have to produce final results. It may also produce partial results, which have to be added by additional logic in a later stadium. For this case it would mean that P1 produces two partial results for y_{46}. Each partial result is then added to a partial result of P2. P2 thus also produces two partial results. To compute the final result these two partial results have to be added by additional logic.

P1:

Instruction:	Start time:	End time:
$y_{46}^1 = e_{80} * x_{30} + 0$	1	13
$y_{46}^2 = e_{81} * x_{32} + 0$	2	14

Table 8: Instructions carried out by P1

P2

Instruction:	Start time:	End time:
$y_{46}^3 = e_{82} * x_{48} + y_{46}^1$	14	26
$y_{46}^4 = e_{83} * x_{50} + y_{46}^2$	15	27

Table 9: Instructions carried out by P2

Additional logic:

Instruction:	Start time:
$y_{46} = y_{46}^3 + y_{46}^4$	28

Table 10: Instructions carried out by additional logic

The advantage of this solution is that the PEs do not have internal dependencies anymore. Therefore the utilization of the PEs can be very high. The disadvantage is that additional logic is required for the accumulation of the partial results. The additional logic will be referred as Partial Result Adder.

6.7. Partial Result Adder

The task of the additional logic is to accumulate the partial results of the system. The accumulation can be done with the same two-operand adder used in the MAC unit of the PEs.

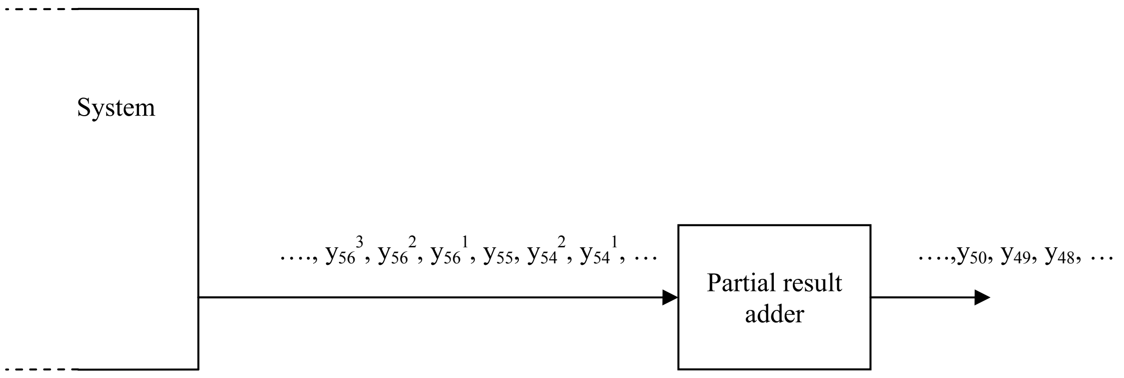

System
…., y_{56}^3, y_{56}^2, y_{56}^1, y_{55}, y_{54}^2, y_{54}^1, …
Partial result adder
….,y_{50}, y_{49}, y_{48}, …

Figure 61: System with Partial Result Adder

With this solution the PEs do not have internal dependencies anymore. Unfortunately the dependencies are at the partial result adder. In [14] an implementation of a partial result adder is proposed.

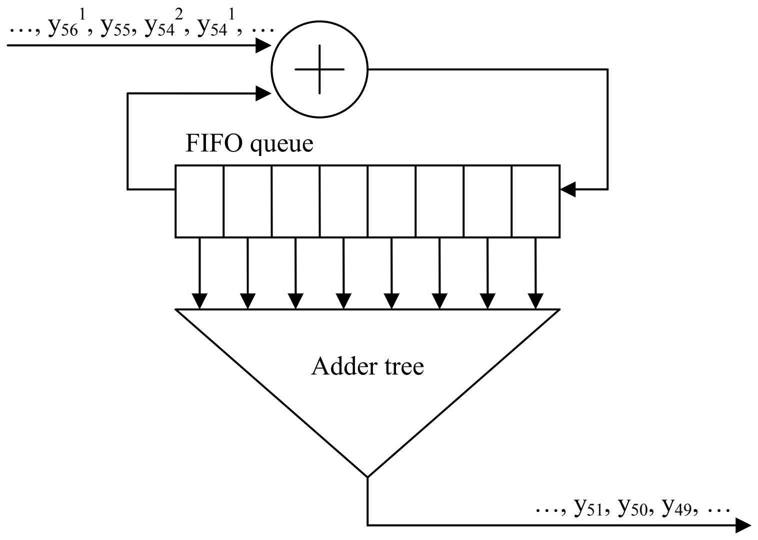

…, y_{56}^1, y_{55}, y_{54}^2, y_{54}^1, …
FIFO queue
Adder tree
…, y_{51}, y_{50}, y_{49}, …

Figure 62: Example of partial result adder

Figure 62 plots a part of the partial result adder as presented in [14]. To compensate for the adder pipeline, there is a FIFO queue with length equal to the number of pipeline stages. As long as the partial results have the same row index, the partial results flow through FIFO queue. The advantage of this design is that the number of partial results is reduced to the number of pipeline stages (eight in case of the MAC unit of [7]) regardless of the number of partial results produced by the PEs. An adder tree is used to add the final eight partial results into the final result. The FIFO queue in the design of [14] is implemented as a two dimensional array. The number of rows is equal to the number of columns which is equal to the number of pipeline stages (assumed to be eight). The extra dimension is required to process rows with less than eight partial results. The access pattern of the write pointer is the same as the access pattern of the read pointer but has a delay of eight cycles.

The disadvantage of the proposed partial result adder is that the utilization of the adder tree will be low. The maximum rate of the partial results is one result every clock cycle but the adder tree is capable of adding eight partial results in one clock cycle.
The advantage of the partial result adder is that the control is relatively simple. The control does not have to schedule computation.

6.8. Preprocessing

Each PE covers a part of the band of the matrix. This means that a PE cannot process every non-zero element. This requires that the non-zero elements have to be divided over the PEs such that they can be processed. This dividing can be done off-line with a preprocessor or on the fly. The disadvantage of an off-line solution is that it can produce extra information. This extra information has to be send from external memory to the FPGA besides the matrix. The bottleneck of the SMVM is always the memory bandwidth. The communication between the FPGA and the memories must be kept to a minimum. Extra information of the preprocessor thus has a negative affect on the performance of the implementation.
The disadvantage of an on-the-fly scheduler is the complexity. The performance of such a scheduler must be very high. It has to be able to assign a non-zero element to each PE in one clock cycle. When the number of PEs increases a sequential implementation will not be fast enough and a parallel implementation has to be used. A parallel implementation will add additional complexity to the scheduler. Because of the complexity a preprocessing solution might be a good choice.

In the most optimal case, only the system matrix and the vector **x** are sent from memory to the FPGA once. Any additional information is seen as overhead.

Each PE is able to process a matrix stored in CSR format. The main matrix has to be divided into multiple matrices referred as matrix slices. A matrix slice has the same size as the original main matrix but has less non-zero elements. A matrix slice is thus a subset of the original matrix and will be processed by one PE. Notice that a non-zero element can only be in one of the matrix slices.
The non-zero elements within a matrix slice lie in the small band covered by the PE that will process it.

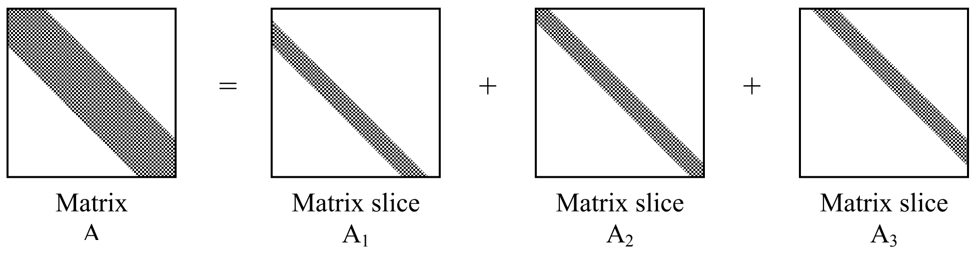


Figure 63: Division of matrix into matrix slices

6.8.1. Overhead

Dividing the system matrix into matrix slices results in some storage and memory bandwidth overhead.

The system matrix of the Volume Reconstruction algorithm has 138,324 rows and columns. In the normal case, to index the columns, at least 18 bits (2^18 = 262,144) are required. But since a PE can only index within the band (22,393 elements wide) only 15 bits (2^15 = 32,768) are required. These indices have thus become relative to the main diagonal. Because words are 64 bits, four column indexes could be transferred in one word (4*15 = 60) (because of alignment 16 bits will be used). The values are represented with 64 bits. Thus five 64-bit words are needed to represent four non-zero elements with their column index.
In the CSR format the size of the array containing the row pointers is equal to the number of rows plus one. In this case that is 138,325. The row pointers must point within the 'val' and 'col' arrays. The length of these arrays is equal to the number of non-zeros. For the Volume Reconstruction matrix the number of non-zeros is 2,460,562. To index the non-zeros 22 bits are needed (2^22 = 4,194,304).
Because the rows are processed in consecutive order it is sufficient to store the number of non-zeros for a row instead of a pointer for each row. The maximum number of non-zeros on one row for the Volume Reconstruction Algorithm is 49. Because of alignment, 8 bits will be used to store the number of non-zeros of a row. Thus instead of using the CSR format a slightly modified version is used.

Modifications of the CSR format:

- Instead of storing the normal column index, store relative indices.
- Instead of storing pointers for each row, store number of non-zeros.

With this information the size of the arrays can be determined.

Size of Val: (64/8)*2,460,562/(1024*1024) = 18.8 MB.
Size of Col: (64/8)*2,460,562/(4*1024*1024) = 4.7 MB.
Size of Row: (64/8)*138,324/(8*1024*1024) = 0.13 MB.

	Size in MB	Percentage
Val	18.8	80%
Col	4.7	20%
Row	0.13	0.56%
Total	23.6	100%

Table 11: Storage requirements for Volume Reconstruction matrix

The extra information of the matrix slices comes from the row array. Fortunately the size of this array is small compared to the other arrays.

Nr. of matrix slices	Total Size in MB	Extra information
2	23,73	0,56%
3	23,86	1,12%
4	23,99	1,68%
5	24,13	2,24%
6	24,26	2,80%
8	24,52	3,91%
10	24,79	5,03%

Table 12: Extra information for Volume Reconstruction matrix slices

6.8.2. Division in matrix slices

The task of the preprocessor is to divide the matrix into matrix slices. To achieve high utilizations of the PEs, the matrix slices have to contain about the same number of non-zeros.

Each PE will compute the SMVM of a matrix slice. The problem is that every PE has to produce the same number of partial results for a row to produce the correct result. This means that the number of partial results for a row is equal to the maximum number of non-zeros for a row of all the matrix slices.
One matrix slice might for example cover three non-zeros of a row while another matrix slice covers for example five non-zeros.

An example of two matrix slices with the modified CSR format:

$$A_1 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 4 & 6 & 0 \\ 2 & 5 & 7 & 8 \end{pmatrix}$$

$$\begin{aligned} \text{val} &= (1\ 3\ 4\ 6\ 2\ \ 5\ \ 7\ \ 8) \\ \text{relative col} &= (0\ 0\ -1\ 0\ -3\ -2\ -1\ 0) \\ \text{nnz row} &= (1\ 1\ 2\ 4) \end{aligned}$$

$$A_2 = \begin{pmatrix} 0 & 0 & 0 & 2 \\ 0 & 0 & 4 & 5 \\ 0 & 0 & 0 & 6 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\begin{aligned} \text{val} &= (2\ 4\ 5\ 6) \\ \text{relative col} &= (3\ 1\ 2\ 1) \\ \text{nnz row} &= (1\ 2\ 1\ 0) \end{aligned}$$

Figure 64: Small matrix one **Figure 65: Small matrix two**

Matrix slice A_1 will be processed by PE P_1 and matrix slice A_2 will be processed by PE P_2. In case P_1 executes the matrix slice in a straight forward way it produces eight partial results. The problem is that P_1 will only produce one partial result for row two while P_2 needs two partial results. This can be solved by P_1 by producing an extra partial result with value zero.

P_1 will execute the following instructions:

$y_1^1 = 1 * x_1$
$y_2^1 = 3 * x_2$
$y_2^2 = 0$
$y_3^1 = 4 * x_2$
$y_3^2 = 6 * x_3$
$y_4^1 = 2 * x_1$
$y_4^2 = 5 * x_2$
$y_4^3 = 7 * x_3$
$y_4^4 = 3 * x_4$

Table 13: Instruction executed by P_1

P_2 will receive nine partial results from P_1. P_2 will either add a partial result to a received partial result of P_1 or it passes the partial result on. P_2 will execute the following instructions.

$y_1 = 2 * x_4 + y_1^1$
$y_2^3 = 4 * x_3 + y_2^1$
$y_2^4 = 5 * x_4 + y_2^2$
$y_3^3 = 6 * x_4 + y_3^1$
$y_3^4 = y_3^2$
$y_4^5 = y_4^1$
$y_4^6 = y_4^2$
$y_4^7 = y_4^3$
$y_4^8 = y_4^4$

Table 14: Instruction executed by P_2

Every PE will need a controller to schedule additional partial results which have a value of zero.
Note that the total utilization of the PEs is 12/18 = 0.67. For this example the total utilization could be higher if the non-zeros where more equally divided over the matrix slices.

The most ideal situation is when each matrix slice has about the same number of non-zeros on a row because this leads to the lowest number of partial results. In that case the total number of MAC slots used is equal to the number of partial results times the number of PEs.

The task of the scheduler is thus to divide the bandwidth of the matrix over the PEs such that the matrix can be divided into matrix slices such that the number of partial results is low. The complexity lies in the memory usage for the storage of the vector **x**. The most optimal partition into matrix slices can be achieved when every PE stores the complete bandwidth but that's a very resource inefficient solution. This means that there is a trade off between memory usage and utilization. This complex task has to be fulfilled by the preprocessor.

6.9. Controller

As mentioned in chapter 6.8, a PE might have to schedule additional partial results which are zero. Each PE will need a controller to schedule these additional partial results.

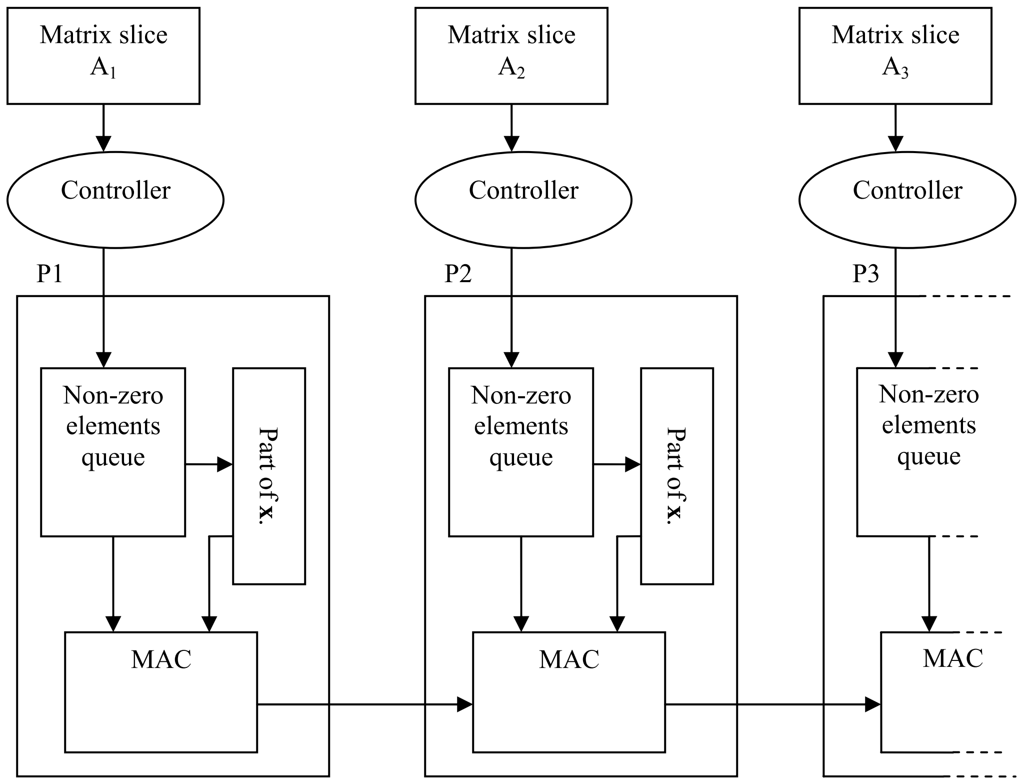


Figure 66: Part of system

A controller has to schedule additional partial results if the number of non-zeros of the row is less than the maximum number of non-zeros for that row. This information can be determined by the preprocessor or it can be determined on the fly. The preprocessor solution will result in an additional array with the same size as the row array of the modified CSR format. As already seen in chapter 6.8, this array is very small.

6.10. Upper bound Utilization

As already indicated the highest utilization of the MAC units can be achieved when each PE covers the complete band of the matrix. This results in a large window for each PE. Each PE stores thus the same elements of **x** which is recourse inefficient. Because each PE covers the complete band, the most optimal partition in matrix slices can be determined. This is an upper bound on the utilization of the PEs.

For the system matrix of the Volume Reconstruction algorithm the upper bound on the utilization is plotted in figure 67.

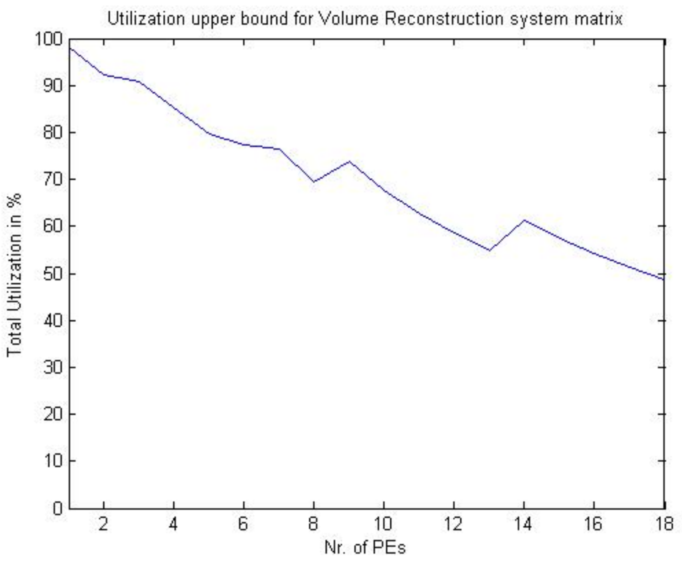


Figure 67: Upper bound of utilization of Volume Reconstruction system matrix

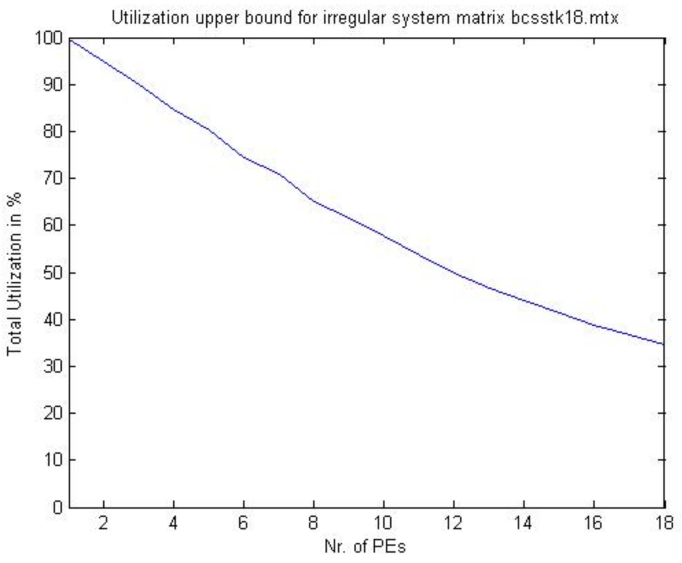


Figure 68: Upper bound of utilization of irregular system matrix bcsstk18

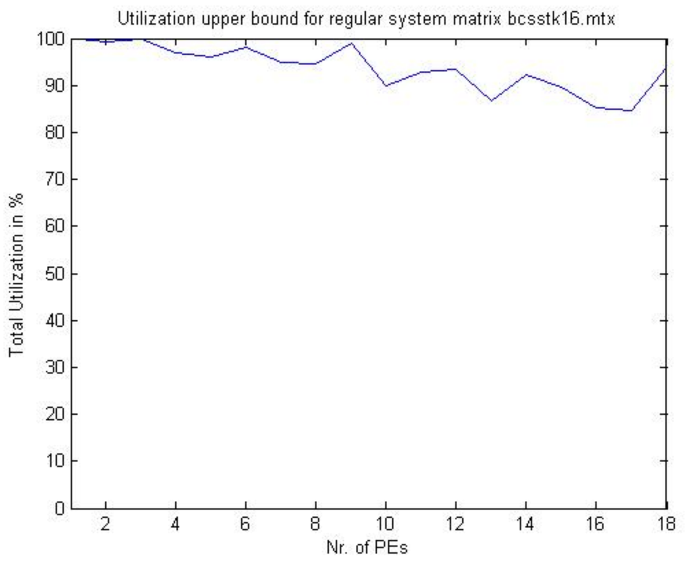


Figure 69: Upper bound of utilization of regular system matrix bcsstk16

As mentioned the upper bound can be achieved if all the PEs cover the complete band. Unfortunately this is very memory inefficient and might not be possible at all because of limited FPGA memory.

6.11. Preprocessor

As mentioned in chapter 6.8 the preprocessor has to divide the bandwidth of the matrix over the PEs and to divide the matrix in multiple “equal” matrix slices.
To achieve the highest utilization with a given memory size, multiple iterations of these two task are required. Besides the multiple iterations, the tasks themselves are also very complex.
Questions that the preprocessor has to answer are for example:
“How large should the window of PE 1 be?”
“Is the utilization higher if a part of the window of PE 1 is moved to PE 2?”
“Should this non-zero element be processed by PE 3 or by PE 4?”
Without proving it, answering these questions is a NP-complete problem. With a heuristic function it might be possible to get close to the optimal solution while having a fast implementation.

6.11.1. Dividing the bandwidth

For each row the non-zero elements have to be divided over the PEs. The goal of the preprocessor is to divide the bandwidth such that it is possible to get close to the upper bound utilization with a limited amount of memory. For each row there is a minimum number of partial results, this minimum is:

$$PR_{\min} = \left\lceil \frac{\#\,\text{elements within a row}}{\#\,\text{PEs}} \right\rceil$$

To get close to the upper bound this minimum must be achieved, which depends on the bandwidth coverage.

If a row cannot be scheduled perfectly (number of non-zeros of a row modulo the number of PEs is zero) than it is hard to determine the bandwidth of the PEs. The preprocessor has in that case multiple choices to schedule the non-zero elements.
If a row can be scheduled perfectly than each non-zero element has to be processed by a certain PE. This situation gives thus information about the bandwidth requirements of the PEs.

For each PE, the non-zero elements that have to be processed by that PE are collected. Each non-zero element has a relative column index. Figure 70 is a histogram of the non-zero elements of the Volume Reconstruction matrix that should be processed by PE 1 to get close to the upper bound utilization in case of a system of nine PEs. Most of the non-zero elements have a relative column index of -4000.

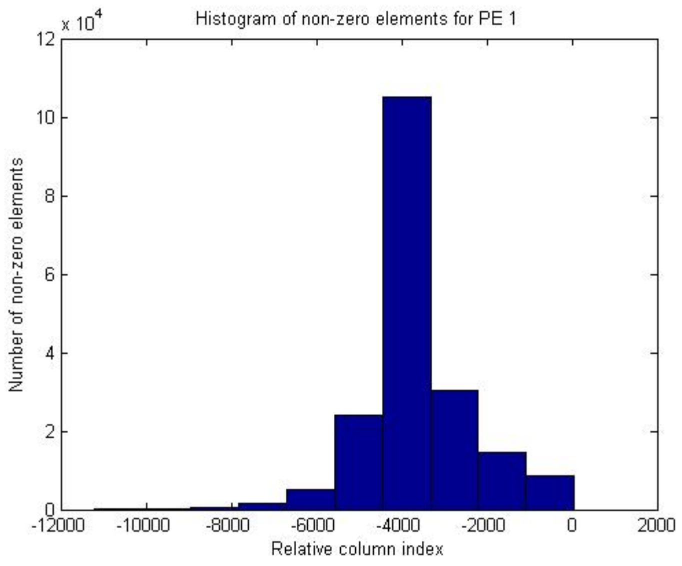


Figure 70: Histogram of the non-zero elements that should be processed by PE 1

To get close to the upper bound utilization, PE 1 should cover the bandwidth of -11,000 till 0. The same plot can be made for PE 2.

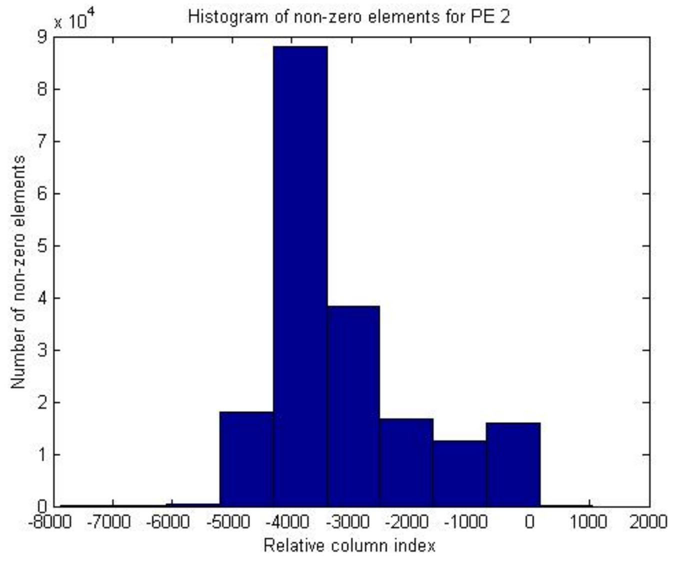


Figure 71: Histogram of the non-zero elements that should be processed by PE 2

PE 2 should cover the bandwidth from -8,000 till 1,000. Figure 72 plots the bandwidth requirements of the nine PEs. The black parts are the parts of the bandwidth that should be covered by a PE.

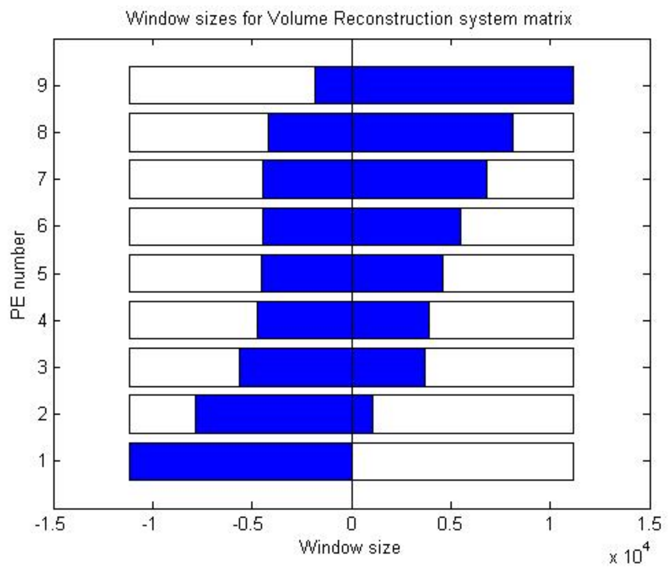


Figure 72: Bandwidth requirements of the nine PEs.

As already seen in figure 70, PE 1 should cover the complete part below the main diagonal. PE 9 should cover the complete part above the main diagonal plus a small part below the main diagonal. With the division of the bandwidth indicated by figure 72 the upper bound utilization (73.8%) can be realized. Unfortunately the memory requirements are quite high. The bandwidth of the matrix in this particular case has to be stored 4.2 times.

An optimization to limit the amount of memory is to let every PE cover not all of its non-zero elements, but to let a PE cover only for example 95% of its non-zero elements. Consider figure 71, only a few non-zero elements have a relative column index between -8,000 and -5,000 or between 0 and 1,000. Covering these elements by PE 2 is very inefficient; it requires almost twice the amount of memory to cover about 2% more non-zero elements.

If the memory usage of the PEs is limited between 1,9 and 2,0 times the bandwidth, the following division is made:

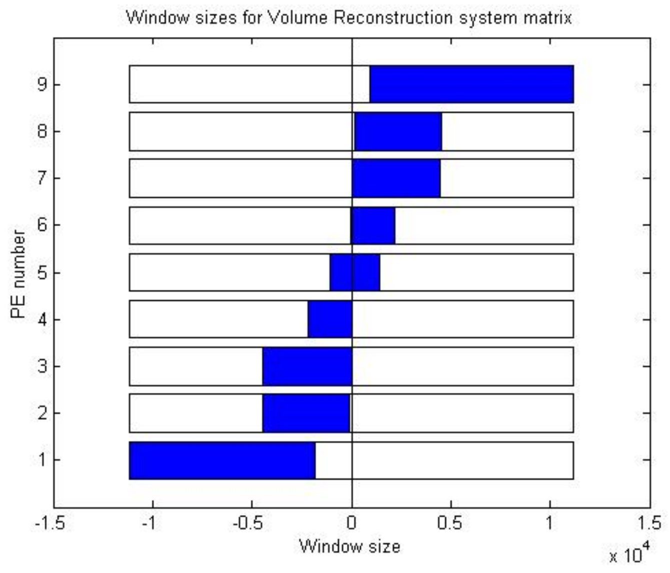


Figure 73: Bandwidth requirements of the nine PEs with limited amount of memory.

The utilization that could be achieved with the bandwidth division indicated by figure 73 is still 62.8%.

6.11.2. Scheduling the non-zero elements over the PEs

After a bandwidth division is made, the matrix slices have to be made. The utilization numbers in chapter 6.11.1 were already computed with a heuristic implementation for the scheduling of the non-zero elements. This chapter explains how the non-zero elements can be scheduled over the matrix slices (PEs).

The PEs with lowest number cover the lowest relative column indices, the PEs with the highest number cover the highest relative column indices.
The matrix is stored in the CSR-format. Scheduling the non-zero elements is done by traversing through the arrays linearly. This means that the non-zero elements will be scheduled row wise. Within a row the elements are scheduled from the lowest column index to the highest.

For every row there is minimum number of partial results as indicated in chapter 6.11.1. This means that each PE can process the same number of non-zero elements as the minimum number of partial results without consequences. The preprocessor uses this information to assign the non-zero elements. The minimum number of partial results is referred as PR_{min}. At the start of a new row, the preprocessor tries to assign the first PR_{min} non-zero elements to the first PE. In case one of these non-zero elements cannot be covered by the first PE, the preprocessor continues with the second PE, etcetera.
In case a non-zero element originally intended to be processed by a PE has a lower column index than the PE covers the non-zero element has to be processed by the previous PE.

Example:
Suppose the SMVM of a given matrix and vector has to be computed with three PEs with the following bandwidth distribution:
PE P_1: -10 till 0
PE P_2: -5 till 5
PE P_3: 0 till 10

Suppose, the following rows have to be scheduled:

Row 13:	Non-zero element:	A	B	C	D	E			
PR_{min}= 2	Relative column index:	-4	1	2	6	8			
Row 14:	Non-zero element:	F	G	H	I	J	K		
PR_{min}= 2	Relative column index:	-8	-7	-6	4	7	9		
Row 15:	Non-zero element:	L	M	N	O	P	Q		
PR_{min}= 2	Relative column index:	-10	1	3	6	7	8		

Table 15: Example of three rows

The minimum number of partial results for the rows is two.
Because of PR_{min} of row one, the preprocessor tries to assign two elements (A and B) to PE 1. The relative column index of element B is not in the range of the bandwidth coverage of P_1 thus it cannot be processed by P_1. For the second PE, the preprocessor also tries to assign two elements, in this case elements B and C. Both elements can be processed because the relative column indexes are within the bandwidth rage of P_2. Elements D and E are assigned to and processed by P_3.
A more advanced example is row two. PR_{min} is two thus the preprocessor assigns element F and G to the first PE. In the next step element H and I would be assigned to the second PE but because element H has a lower relative column index than the bandwidth range of PE 2 it has to be processed by the first PE. Instead of having PR_{min} (two) partial results, three partial results are required to process row two. The third row also requires three partial results.

Result of scheduling of the three rows:

Elements processed by P_1	Elements processed by P_2	Elements processed by P_3
A	B	D
F	C	E
G	I	J
H	M	K
L	N	O
		P
		Q

Table 16: Scheduling of the three rows

For this example the preprocessor computed an optimal schedule using a simple heuristic as proposed. The main advantage of this simple heuristic is that it is fast. The preprocessor does not need to consider all possibilities.

Suppose a fourth row has to be scheduled with the same system as proposed above.

Row 16:	Non-zero element:	R	S	T	U				
PR_{min}= 2	Relative column index:	-4	-2	2	4				

Table 17: Example of a row

For this small example there are already six possible optimal schedules:

Possibility	Elements processed by PE 1	Elements processed by PE 2	Elements processed by PE 3
1	R,S	T,U	-
2	R,S	T	U
3	R,S	-	T,U
4	R	S,T	U
5	R	S	T,U
6	-	R,S	T,U

Table 18: Multiple optimal schedules

Evaluating all the possibilities results in a very slow preprocessor.

Because a heuristic is used to schedule the non-zero elements it is not guaranteed that an optimal schedule is computed. A situation that doesn't result in an optimal schedule is the following.

Row 5:	Non-zero element:	V	W	X	Y	Z	AA		
PR_{min}= 2	Relative column index:	1	2	3	6	7	8		

Table 19: Example row five

The preprocessor will assign elements V and W to P_2. The other four elements will be assigned to P_3. This results in four partial products while an optimal schedule results in three partial products.

6.12. Multi vector design

For the Volume Reconstruction algorithm the system A **x** = **b** has to be solved 51,000 times [1], a large part is independent of each other. This means that it is possible to solve multiple systems in parallel. The main advantage is that because matrix A is constant, it is possible to load matrix A once while multiplying it with multiple vectors. Figure 74 shows an example of a two vector design.

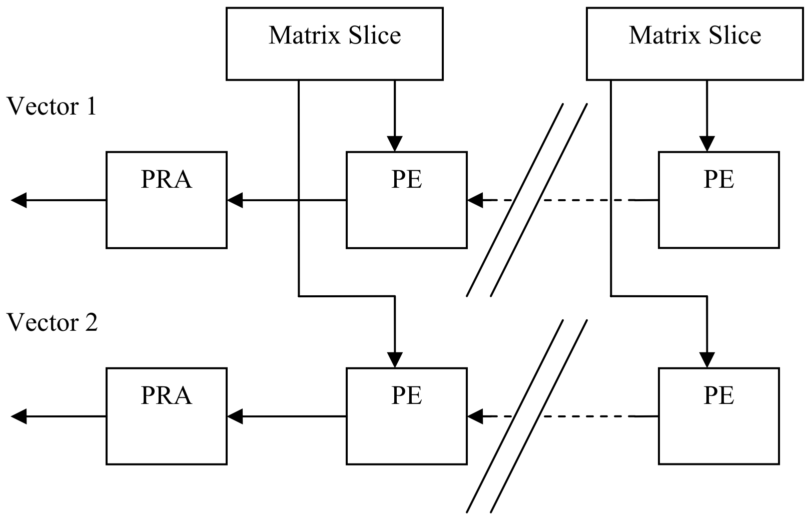


Figure 74: Example of a two vector design

The memory bandwidth requirements are almost half of a single vector design if the number of PEs is equal to the number of PEs in a single vector design. The disadvantage of the multi vector design is that for each SMVM system a partial result adder is required. In the current design it is assumed that the partial result adder consists of eight MAC units (see section 6.7). Because the maximum number of MAC units for the target FPGA is 31 the overhead of extra partial result adders is significant. As explained in chapter 6.7 the utilization of the PRA is very low. Further research is required to limit the resources occupied by the partial result adders. Perhaps it is possible to use one partial result adder with multiple SMVM systems.

6.13. Total System

Because of the limited amount of on chip memory the matrix slices and vectors are stored in an external memory. A memory controller is used to control the external memory.
Each PE processes a matrix slice. This matrix slice has to be loaded from external memory. The normal operation would be to load for each PE a part of the matrix slice. A round-robin schedule could be used to load the parts of slices for each PE. Some PEs will process slightly more non-zero elements than other PEs. The round-robin schedule should be modified such that only data is loaded if a PE has a request for data. This task will be fulfilled by the data scheduler. Besides loading parts of matrix slices another task is to load elements of vector **x** and to write elements of result vector **y** to external memory.

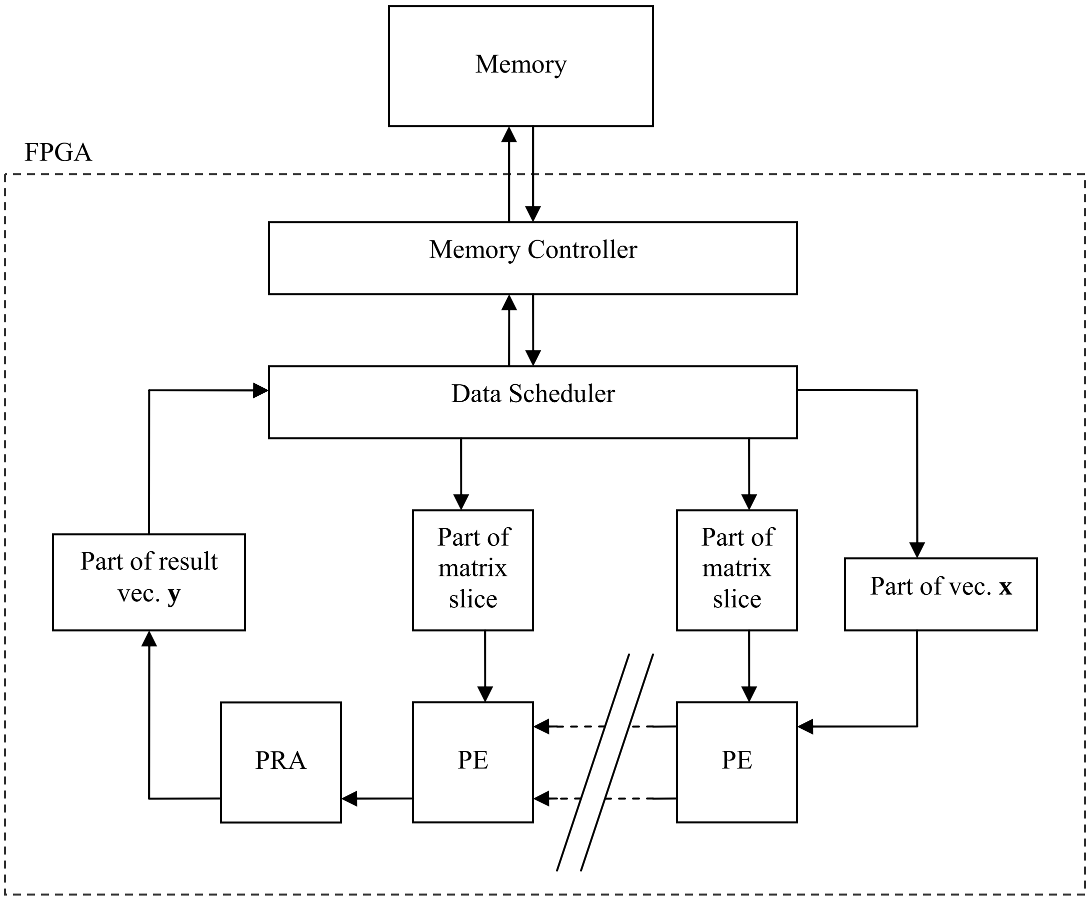


Figure 75 : Total system design

6.14. Conclusion

The design proposed will be analyzed with the largest Virtex-II Pro (XC2VP100) FPGA from Xilinx as a target. The 64-bit MAC unit proposed in [7] can run on the target FPGA at 170 MHz (speed grade: -6) while the maximum number of MAC units is 31. The assumption is that the MAC unit is by far the most complex part of the system. The assumption is that 90% of the resources of a PE is taken by the MAC unit.

Besides the logical resources there is a limited amount of on chip memory. The target FPGA has 7,992 kbit of BlockRAM memory. The on chip memory is used to store parts of vector **x**, result vector **y** and parts of matrix slices.
To use the memory bandwidth as efficient as possible, burst transfers have to be used. For DDR-SDRAM the largest burst transfer is eight 64-bit words. This means that for each PE and each array ('val', col and row array of CSR format) at least eight 64-bit registers are required. To make sure that these registers will not become a bottleneck, sixteen registers will be used for each array. Thus for each PE, 3*16 = 48 registers of 64-bit will be used. For the vectors slightly more registers will be used (24 registers).

As mentioned before, the FPGA has 7,992 kbit of memory; almost 128.000 64-bit words. The storage requirements for the matrix slices are thus very low compared to the total amount of on chip memory. For the storage of the elements of vector **x** the assumption is that 90% of the on chip memory can be used.

The bandwidth of the Volume Reconstruction matrix is about 22k elements. With the memory of the target FPGA the bandwidth could be saved about 5 times. The preprocessor will use this information to determine the maximum performance that could be achieved with the design proposed in this chapter. For the target FPGA with single vector design, the highest performance is achieved with 17 PEs which results in 2.95 GFLOPS. The required memory bandwidth is 15.5 GByte/s.
For a two vector design the maximum number of PEs per vector is five. Every vector needs an expensive partial result adder which consists of eight MAC-units. Besides less PEs per vector, the on chip memory has to be divided over the two systems. The maximum performance for one vector system consisting of five PEs is 1.35 GFLOPS. Two of these systems can achieve a performance of 2*1.35 = 2.71 GFLOPS with a memory bandwidth of 8.3 GByte/s.
In [1] the SMVM of the Volume Reconstruction system matrix is optimized for the Intel Bensley platform. The system in [1] used two dual-core Xeon 5140 Woodcrest running at 2.3 GHz and four FB-DIMMs to provide a total memory bandwidth of 21 GB/s. The performance achieved was almost one GFLOPS.

The system requirements defined in chapter 3 are met with SBC for the Volume Reconstruction matrix.

- The system has to keep up with the memory interface; it should not become the bottleneck.
  The maximum performance of SBC on the target FPGA is 2.95 GFLOPS with a memory bandwidth of 15.5 GByte/s. This means that the system can process data with a maximum of 15.5 GByte/s. In perspective, five DDR2-400 SDRAM modules have together a peak of 16 GByte/s.
- The overhead that might be needed to schedule the SMVM must be kept to a minimum.
  The maximum number of 64-bit MAC units on the target FPGA is 31. In the current design the partial result adder uses eight of these MAC units. The maximum number of PEs is thus 23. The storage and communication overhead with 23 PEs is only 11.3%
- The design has to be scalable in the available memory bandwidth.
  The maximum data rate of SBC for the Volume Reconstruction matrix on the target FPGA is 15.5 GByte/s. More performance can be achieved by using more FPGAs to split the matrix horizontally or if possible using a multi-vector implementation.
- The design has to compute the SMVM of regular and irregular system matrices evenly well
  The maximum performance for the regular system matrix bcsstk16 is 5.4 GFLOPS. For the irregular system matrix bcsstk18 the maximum performance is 2.1 GFLOPS.

7. Results

On the internet there are two large collections of sparse matrices available. One is the University of Florida Sparse Matrix collection [15] which contains over 1800 matrices, the other is the Matrix Market [11] collection. The matrices represent real problems that arise in different application areas. Examples of the problem domains are structural analysis, fluid dynamics, heat transport, electromagnetism, but also economic modeling.
Because of the large number of sparse matrices and application areas it is hard to find a proper subset to test performance. Another problem is that there is no fixed test bench to test Sparse Matrix Vector Multiplication methods. This makes it hard to test against other methods. Examples are [10] and [12] where they used a number of matrices but never specify them.

One of the application areas of the Finite Element Method is structural engineering. A large set of structural engineering matrices is given in [16]. The following matrices are tested.

Nr.	Name:	Description:	Dimension:	Non zeros:
1	PKUSTK01	Beijing botanical exhibition hall	22,044	979,380
2	PKUSTK03	Dalian group silo	63,336	3,130,416
3	PKUSTK04	Yunsan Plaza	55,590	4,218,660
4	PKUSTK05	Cofferdam (reduced model)	37,164	2,205,144
5	PKUSTK06	Cofferdam (reduced model)	43,164	2,571,768
6	PKUSTK08	Cubic 21 nodes solid, 11x11x11 mesh	22,209	3,226,671
7	PKUSTK09	Group silo	33,960	1,583,640
8	PKUSTK10	4 tower silo	80,676	4,308,984
9	PKUSTK12	Jijian Plaza, tall building	94,653	7,512,317
10	PKUSTK13	Machine element, 21 nodes solid	94,893	6,616,827
11	PKUSTK14	Tall building	151,926	14,836,504

Table 20: Structural engineering matrices from Peking University

Figure 76 plots the performance that could be achieved with SBC on the target FPGA.

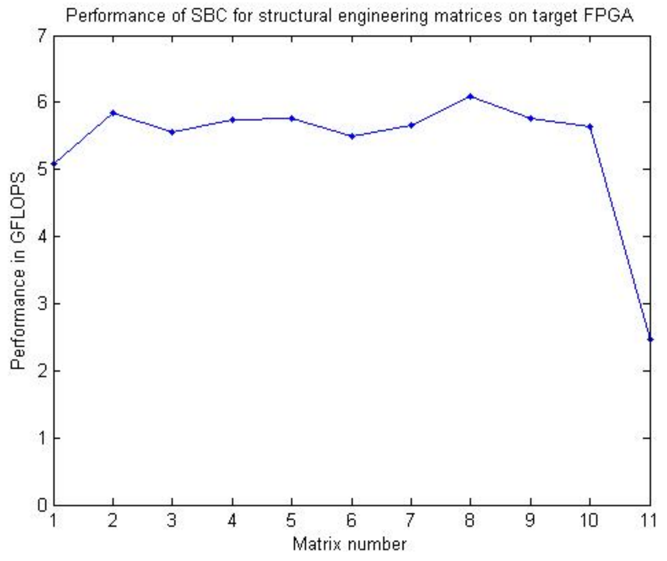


Figure 76: Performance of SBC for structural engineering matrices

In [17] a more advanced test set is defined. The test set has 44 matrices of which 16 are from a Finite Element Method. These FEM matrices are from all kinds of engineering areas.

Nr.	Name:	Description:	Dimension:	Non zeros:
1	raefsky3	Fluid structure interaction	21,200	1,488,768
2	olafu	Accuracy problem	16,146	1,015,156
3	bcsstk35	Stiff matrix automobile frame	30,237	1,450,163
4	venkat01	Flow simulation	62,424	1,717,792
5	crystk02	FEM Crystal free vibration	13,965	968,583
6	crystk03	FEM Crystal free vibration	24,696	1,751,178
7	nasasrb	Shuttle rocket booster	54,870	2,677,324
8	3dtube	3-D pressure tube	45,330	3,213,332
9	ct20stif	CT20 Engine block	52,329	2,698,463
10	af23560	Airfoil eigenvalue calculation	23,560	484,256
11	raefsky4	buckling problem	19,779	1,328,611
12	ex11	3D steady flow calculation	16,614	1,096,948
13	rdist1	Chemical process separation	4,134	94,408
14	av41092	2D PDE problem	41,092	1,683,902
15	orani678	Economic modeling	2,529	90,185
16	rim	FEM fluid mechanics problem	22,560	1,014,951

Table 21: FEM matrices from different application areas

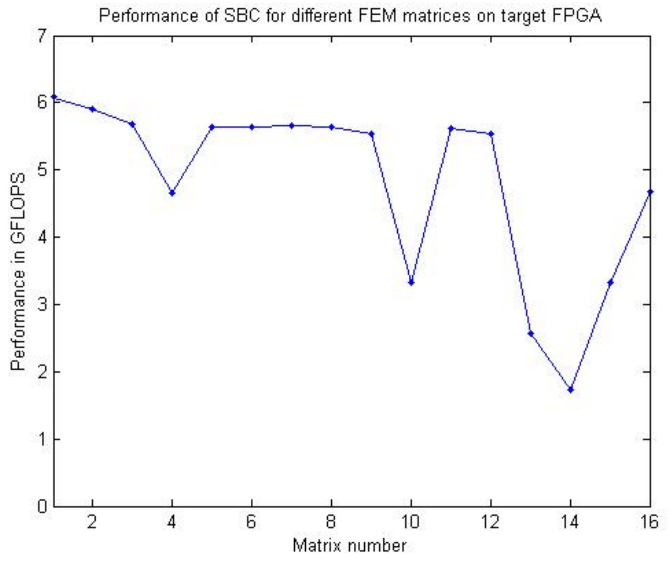


Figure 77: Performance of SBC for different FEM matrices

As can be seen in figure 76 and figure 77, the performance of SBC depends on the matrix used. There are two situations in which SBC performs poor:

- If the number of non-zero elements on a row is limited, SBC cannot fully utilize the PEs. With SBC every row has at least one partial result. If the number of non-zero elements is lower than the number of PEs the system is not fully utilized.
- If the bandwidth of the matrices is very large (around 40k elements) the bandwidth coverage of the PEs is far from optimal because of the limited amount of on chip memory.

8. Future Work

8.1. Improving the partial result adder

The proposed design of the partial result adder has a very low utilization. The adder tree uses seven adders while at most only one adder is used effectively.

8.2. Symmetry

The bottleneck in the performance of the SMVM is the memory bandwidth. A possible optimization not covered in this project is the use of the symmetry of the system matrix.
If the symmetry of the system matrix could be used, every non-zero value (except values on the main diagonal) could be used twice. Instead of loading the complete matrix only the values on the main diagonal and above (or below) the main diagonal have to be loaded. This optimization could save almost 50% loads on the system matrix.

Example:

$$\begin{pmatrix} 1 & 0 & 2 & 3 \\ 0 & 4 & 5 & 0 \\ 2 & 5 & 6 & 7 \\ 3 & 0 & 7 & 8 \end{pmatrix}$$

Figure 78: Example of a symmetric matrix

The values 2,3,5,7 occur twice in the matrix of figure 78, one time above and one time below the main diagonal. The solution presented in chapter 6 computes the SMVM from the CSR format.
Instead of storing the complete matrix in the CSR format only the part above the main diagonal is stored in the CSR format. The solution of chapter 6 can compute the SMVM of the upper part the same way as presented in this report. Only a small adjustment on the system is needed to handle the main diagonal.

$$\begin{aligned} \text{val} &= \begin{pmatrix} 2 & 3 & 5 & 7 \end{pmatrix} \\ \text{col} &= \begin{pmatrix} 3 & 4 & 3 & 4 \end{pmatrix} \\ \text{row} &= \begin{pmatrix} 1 & 3 & 4 & 5 \end{pmatrix} \end{aligned}$$

Figure 79: Upper part of matrix defined in figure 78 in CSR format.

Notice that the upper part stored in CSR format corresponds with the lower part stored in Compressed Sparse Column (CSC) format.

$$\begin{aligned} \text{val} &= \begin{pmatrix} 2 & 3 & 5 & 7 \end{pmatrix} \\ \text{row} &= \begin{pmatrix} 3 & 4 & 3 & 4 \end{pmatrix} \\ \text{col} &= \begin{pmatrix} 1 & 3 & 4 & 5 \end{pmatrix} \end{aligned}$$

Figure 80: Lower part of matrix defined in figure 78 in CSC format.

To use the symmetry the information in the upper part of the matrix must be used such that also the partial results of the lower part are computed. These partial results then have to be added to the partial results of the upper part and the main diagonal to produce the final values. There are two possibilities to accomplish this.

8.2.1. Store partial results in local memory

The system described in chapter 6 computes the SMVM by traversing through the CSR format element by element. The row indexes of the partial results a PE produces are thus in ascending order. If all the partial results of a row are computed, they can be added and the result stored in external memory. The main advantage is that the number of partial result on the FPGA is low. Once they are added to produce a final result they are not used anymore. Another advantage is that the final values are in ascending order.

Instructions of a PE executing the parts thr SMVM above the main diagonal of figure 79:

$$y_1' = 2 \times x_3$$
$$y_1' = 3 \times x_4$$
$$y_2' = 5 \times x_3$$
$$y_3' = 7 \times x_4$$

Figure 81: Instructions to compute partial results of upper part

To compute the SMVM of the lower part, the instructions of figure 79 are different. The column indexes of the upper part are the row indexes of the lower part and the row indexes of the upper part are the column indexes of the lower part. The arrays that represent the upper part in the CSR format also represent the lower part but in CSC format. A PE that produces the partial results of the lower part executes the following instructions if it traverses through the arrays linearly:

$$y_3' = 2 \times x_1$$
$$y_4' = 3 \times x_1$$
$$y_3' = 5 \times x_2$$
$$y_4' = 7 \times x_3$$

Figure 82: Instructions to compute partial results of lower part

Unfortunately the row indexes of these partial results are not in ascending order anymore.

The strength of the system described in chapter 6 was that multiple PEs can compute partial results of one row at the same time. This leads to a system with multiple PEs and only one partial result adder instead of a partial result adder for each PE. The PEs that compute the partial results of the lower part do not have that property anymore if they traverse through the arrays linearly. This means that each PE has to store its partial result.

The partial result can be stored by the PEs based on its row index. The problem is that a PE might compute multiple partial results for one row as the case is in figure 82.
Suppose a PE has previously computed a partial result pr_1 for row 46 and has a new partial result pr_2 for that row. The PE should load pr_1 from memory and add pr_2 to it and store the result. Because of the adder pipeline the result is stored after for example eight cycles. It might be the case that the PE has another partial result pr_3 for row 46 before the result is stored. In that case it would load an old value from memory. This effect is known as Write After Read (WAR) hazard.

Besides the WAR hazard there is another disadvantage. The width of the band determines the number of partial results that have to be saved. The bandwidth of the Volume Reconstruction system matrix is more than 22k elements (chapter 2.4.3). This requires almost 20% of the internal memory capacity.

8.2.2. Convert CSC format into CSR format

As explained, the system can handle matrices in the CSR format very well. A solution would be if the lower part in the CSC format could be converted on the fly into the CSR format.

$$\begin{array}{ll} \text{val} & = (2\ 3\ 5\ 7) \\ \text{row} & = (3\ 4\ 3\ 4) \\ \text{col} & = (1\ 3\ 4\ 5) \end{array}$$

Figure 83: Lower part of matrix CSC format.

To covert a matrix in the CSC format into CSR format, the row array must be sorted. But that would destroy the information of the col array. To maintain the column indexes, another col array is needed (col2). The new array should index for each value the column index.

$$\begin{array}{ll} \text{val} & = (2\ 3\ 5\ 7) \\ \text{row} & = (3\ 4\ 3\ 4) \\ \text{col} & = (1\ 3\ 4\ 5) \\ \text{col2} & = (1\ 1\ 2\ 3) \end{array}$$

Figure 84: Lower part of matrix in extended CSC format.

If the row array is sorted an “extended” form of the CSR format is produced.

$$\begin{array}{ll} \text{val} & = (2\ 5\ 3\ 7) \\ \text{col} & = (1\ 2\ 1\ 3) \\ \text{row2} & = (3\ 3\ 4\ 4) \end{array}$$

Figure 85: Lower part of matrix in extended CSR format.

With some minor adjustments the system can compute the SMVM with the “extended” CSR format.

Sorting is very expensive in hardware, especially for large sets of numbers.
Because of the bandwidth, not all the non-zero elements have to be sorted to get all the non-zeros lined up for a particular row. There is a certain window in which the non-zero elements for one row lie. The size of the window depends on the matrix and can be determined beforehand. For the system matrix of the Volume Reconstruction algorithm this window has to be larger than 100K elements. The storage of 100K elements in 64 bit would already require 80% of the internal memory of the FPGA. Converting the complete upper part into the lower part is thus impractical.
The proposed solution divided the system matrix into matrix slices. The number of matrix slices is equal to the number of PEs. Because of the symmetry the goal is to divide only the upper part over the PEs. For each PE that computes the SMVM of a part of the upper part there has to be a counter PE. This counter PE has the same part of the matrix. Each counter PE has to sort his part of the matrix. Because the size of the matrix slices is smaller than the complete upper part, the sorting window will be smaller. Figure 86 indicates a possible design of such a system.

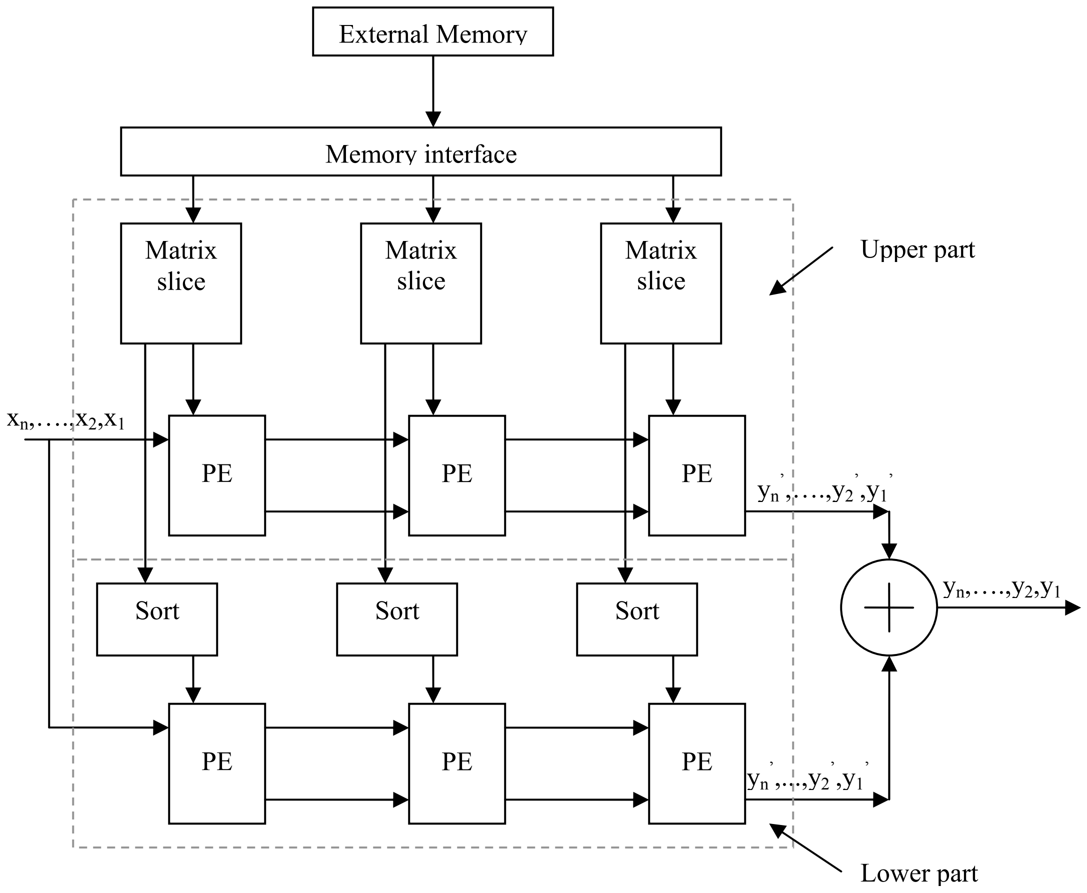


Figure 86: Possible design to use the symmetry of the system matrix

If the upper part of the system matrix of the Volume Reconstruction algorithm is divided into three equal parts, the size of the sorting window has to be more than 35K elements. Some small experiments showed that if the sorting window is kept on 1024 elements, the upper part has to be divided over about twenty PEs.

8.2.3. Conclusion

This proposed solutions to use the symmetry of the system matrix seems to be very difficult to implement and it may be impossible to implement them at all. Further research has to be done to find other solutions to use the symmetry of the system matrix

8.3. Implementation

To proof the concepts presented in chapter 6 an implementation has to be made. Besides the SMVM system, other operations of the Conjugate Gradient algorithm have to be implemented. The other operations are inner products and vector additions / subtractions.

9. Conclusion

The target of this research was to design a method to compute a Sparse Matrix Vector Multiplication (SMVM) on a FPGA. The project mainly focused on the Volume Reconstruction system matrix because of the time requirements described in chapter 1.
Within the research field of SMVM on FPGAs, two proposed implementations were evaluated for the Volume Reconstruction system matrix. Both solutions had large disadvantages.

One of the solutions was the stripe method which is discussed in chapter 4.4. For the Volume Reconstruction system matrix the stripe method has a very low utilization. In chapter 6 a new design is developed using two major modifications on the stripe method. These modifications result in a completely new design called Small Bandwidth Coverage (SBC). The performance of SBC is, compared to the other two evaluated solutions, quite high for the Volume Reconstruction system matrix. The maximum performance on one FPGA with SBC is about a factor six larger than the stripe method.

Within the FPGA research field SBC performs very well. Compared to the performance of General Purpose Processors, SBC gives good performances as well. For the Volume Reconstruction algorithm SBC is a factor three faster than the Intel Bensley platform used in [1]. The platform consisted of two dual-core Xeon 5140 Woodcrest running at 2.3 GHz and four FB-DIMMs to provide a total memory bandwidth of 21 GB/s.

References

[1] “Architecture for Volume Reconstruction in Diffuse Optical Tomography”, M.Sc thesis, Wouter Wiggers, University of Twente, March 2007

[2] http://www.topcrunch.org/

[3] “An introduction to the conjugate gradient method without the agonizing pain”, Jonathan Richard Shewchuk, School of Computer Science, Carnegie Mellon University, Pittsburgh, August 4, 1994

[4] “Matrix computations” (3rd ed.), Gene H. Golub, Charles F. van Loan, Johns Hopkins University Press, Baltimore, 1996

[5] “Fast sparse matrix-vector multiplication by exploiting variable block structure”, Richard W. Vuduc and Hyun-Jin Moon, Lawrence Livermore National Laboratory, University of California, Los Angeles

[6] “Sparse matrix-vector multiplication on a small linear array”, Lenwood S. Heath, Sriram V. Pemmaraju, Calvin J. Ribbens, October 27, 1993

[7] “64-bit Floating-Point FPGA Matrix Multiplication”, Yong Dou, S. Vassiliadis, G. K. Kuzmanov, G. N. Gaydadjiev, *FPGA’05,* February 20–22, 2005, Monterey, California, USA.

[8] “Systolic arrays for VLSI”, H.T. Kung and C.E. Leiserson, *Introduction to VLSI Systems* (Addison-Wesley, Reading, MA, 1980).

[9] “Parallel solution of linear systems with striped sparse matrices”, Rami Melhem, Department of Mathematics and Statistics, University of Pittsburgh, Pittsburgh, PA 15260, U.S.A.

[10] “Hardware Acceleration for Finite-Element Electromagnetics: Efficient Sparse Matrix Floating-Point Computations With FPGAs”, Yousef El-Kurdi, Dennis Giannacopoulos, and Warren J. Gross, Department of Electrical and Computer Engineering, McGill University, Montreal, QC H3A 2A7, Canada, IEEE TRANSACTIONS ON MAGNETICS, VOL. 43, NO. 4, APRIL 2007

[11] Matrix Market, http://math.nist.gov/MatrixMarket/

[12] “Sparse Matrix-Vector Multiplication for Finite Element Method Matrices on FPGAs”, Yousef El-Kurdi, Warren J. Gross, Dennis Giannacopoulos, Department of Electrical and Computer Engineering, McGill University Montreal (QC), H3A 2A7, Canada

[13] “Floating Point Sparse Matrix Vector Multiply for FPGAs”, Michael deLorimier, André DeHon, California Institute of Technology, Pasadena

[14] “Sparse Matrix Computations on Reconfigurable Hardware”, Viktor K. Prasanna, University of Southern California, Gerald R. Morris, US Army Engineer Research and Develpoment Center

[15] University of Florida Sparse Matrix Collection, T. Davis, www.cise.ufl.edu/research/sparse/matrices/

[16] Structural engineering matrices from Peking University, http://www.cise.ufl.edu/research/sparse/matrices/Chen/index.html

[17] “Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply”, Richard Vuduc, James W. Demmel, Katherine A. Yelick, Shoaib Kamil, Rajesh Nishtala, Benjamin Lee, Computer Science Division, University of California, Berkeley

Original assignment:

M.Sc. thesis assignment

Sparse Matrix-Vector multiplication on reconfigurable hardware

Background

Matrix inversion is an operation which is often used in signal processing applications. In this assignment the focus lies on the inversion of large sparse matrices. There are several algorithms to implement this, for example, QRdecomposition, Steepest Descent, Strassens Inversion or the Conjugate Gradient algorithm. In case of sparse matrices the Conjugate Gradient is the most used method.

The conjugate gradient is an iterative method to calculate the vector **x** in the system A***x** = **b** where matrix A and vector b are given. This method can be used to calculate the inverse of A by calculating multiple **x** vectors. These **x** vectors will become the columns of the inverse of A. To calculate these **x** vectors (columns of A), vector **b** will slide over the columns of the identity matrix. This because the property $A*A^{-1} = I$ holds.

The number of iterations to calculate one **x** vector is variable. In each iteration several operations are needed. One of them, sparse matrix vector multiplication, highly dominates the amount of computations. Because of that, this assignment will focus on the sparse matrix vector multiplication. The problem with sparse matrix vector multiplication is that the memory bandwidth limits the performance of different hardware architectures in general. This is in contrast with dense matrices where mainly the processing power limits the performance.

Applications require the inversion of very large sparse matrices within reasonable time. Therefore a lot of processing power is needed, this cannot be achieved efficiently with General Purpose Processors. Reconfigurable architectures have a lot of processing power in potential. As concluded before, this will be limited by the memory bandwidth. Also on that point reconfigurable hardware can be of value.

Goals

Goals of the assignment:

- To develop an efficient sparse matrix vector algorithm for reconfigurable hardware, with in mind the application it is needed for (Conjugate Gradient and Matrix inversion).
- Development of an efficient method to limit the use of off chip memory and thus avoiding the bottleneck that usually arises.
- Simulation of these two methods (the two above points) on at least one specific type of hardware.

The reconfigurable hardware under consideration is a FPGA and the Montium tile processor.