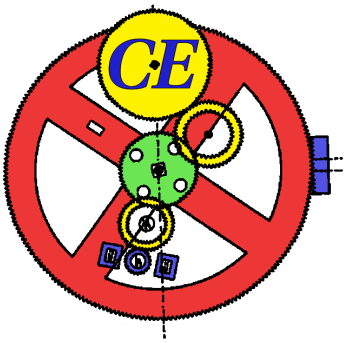

Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2009

MSc THESIS

A Hardware Accelerator for the OpenFoam Sparse Matrix-Vector Product

Mottaqiallah Taouil

CE-MS-2009-09

Abstract

One of the key kernels in scientific applications is the Sparse Matrix Vector Multiplication (SMVM). Profiling OpenFOAM, a sophisticated scientific Computational Fluid Dynamics tool, proved the SMVM to be its most computational intensive kernel. A traditional way to solve such computationally intensive problems in scientific applications is to employ supercomputing power. This approach, however, provides performance efficiency at a high hardware cost. Another approach for high performance scientific computing is based on reconfigurable hardware. Recently, it is becoming more popular due to the increasing On-Chip memory, bandwidth and abundant reasonable cheaper hardware resources. The SGI Reconfigurable Application Specific Computing (RASC) library combines both approaches as it couples traditional supercomputer nodes with reconfigurable hardware. It supports the execution of computational intensive kernels on Customized Computing Units (CCU) in Field Programmable Gate Arrays (FPGA). This thesis presents the architectural design and implementation of the SMVM product for the OpenFOAM toolbox on an FPGA-enabled supercomputer. The SMVM is targeted to be a Custom Computing Unit (CCU) within the RASC machine. The proposed CCU comprises multiple Processing Elements (PE) for IEEE-754 compliant floating point double precision data. Accurate equations are developed that describe the relation between the number of PEs and the available bandwidth. With two PEs and an input bandwidth of 4.8 GB/s the hardware unit can outperform execution in pure software. Simulations suggest speedups between 2.7 and 7.3 for the SMVM kernel considering four PEs. The performance increase at the kernel level is nearly linear to the number of available PEs. The SMVM kernel has been synthesized and verified for the Virtex-4 LX200 FPGA and a hardware counter is integrated in the design to obtain the accurate performance results per CCU. Although the synthesis tool reports higher frequencies, the design has been routed and executed on the Altix 450 machine at 100 MHz. Based on our experimental results we can safely conclude that the proposed approach, using FPGAs as accelerator, has potential for application speedup for the SMVM kernel against traditional supercomputing approaches.

TU Delft

Faculty of Electrical Engineering, Mathematics and Computer Science

A Hardware Accelerator for the OpenFoam Sparse Matrix-Vector Product

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Mottaqiallah Taouil
born in Al Hoceima, Morocco

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Delft, The Netherlands

A Hardware Accelerator for the OpenFoam Sparse Matrix-Vector Product

by Mottaqiallah Taouil

Abstract

One of the key kernels in scientific applications is the Sparse Matrix Vector Multiplication (SMVM). Profiling OpenFOAM, a sophisticated scientific Computational Fluid Dynamics tool, proved the SMVM to be its most computational intensive kernel. A traditional way to solve such computationally intensive problems in scientific applications is to employ supercomputing power. This approach, however, provides performance efficiency at a high hardware cost. Another approach for high performance scientific computing is based on reconfigurable hardware. Recently, it is becoming more popular due to the increasing On-Chip memory, bandwidth and abundant reasonable cheaper hardware resources. The SGI Reconfigurable Application Specific Computing (RASC) library combines both approaches as it couples traditional supercomputer nodes with reconfigurable hardware. It supports the execution of computational intensive kernels on Customized Computing Units (CCU) in Field Programmable Gate Arrays (FPGA). This thesis presents the architectural design and implementation of the SMVM product for the OpenFOAM toolbox on an FPGA-enabled supercomputer. The SMVM is targeted to be a Custom Computing Unit (CCU) within the RASC machine. The proposed CCU comprises multiple Processing Elements (PE) for IEEE-754 compliant floating point double precision data. Accurate equations are developed that describe the relation between the number of PEs and the available bandwidth. With two PEs and an input bandwidth of 4.8 GB/s the hardware unit can outperform execution in pure software. Simulations suggest speedups between 2.7 and 7.3 for the SMVM kernel considering four PEs. The performance increase at the kernel level is nearly linear to the number of available PEs. The SMVM kernel has been synthesized and verified for the Virtex-4 LX200 FPGA and a hardware counter is integrated in the design to obtain the accurate performance results per CCU. Although the synthesis tool reports higher frequencies, the design has been routed and executed on the Altix 450 machine at 100 MHz. Based on our experimental results we can safely conclude that the proposed approach, using FPGAs as accelerator, has potential for application speedup for the SMVM kernel against traditional supercomputing approaches.

Laboratory : Computer Engineering
Codenumber : CE-MS-2009-09

Committee Members :

Advisor:	Georgi Kuzmanov, CE, TU Delft
Chairperson:	Koen Bertels, CE, TU Delft
Member:	Todor Stefanov, Leiden

To all my friends and family for their support

Contents

List of Figures

List of Tables

Acknowledgements

First of all, I would like to thank my advisors Georgi Kuzmanov and Koen Bertels for guiding me through my MSc. thesis. I would like to thank Wouter van Oijen for the thesis he presented. Some work has been reused from it. Next, I would like to express my gratitude to the system administrator of the Computer Engineering's department for keeping up the systems running and assisting with the bugs found in the reconfigurable hardware unit of the SGI Altix machine. Further, I would like to thank Vlad for his contributions in assisting and helping me with the installation of the OpenFOAM software, and identifying the problems integrating the hardware unit to the OpenFOAM tool. Last but not least, I would like to thank my friends and family for the support during this thesis.

Mottaqiallah Taouil
Delft, The Netherlands
July 2, 2009

1 Introduction

To keep up with the increasing demand for computing power in the area of Scientific Computing complex machines are fabricated. These machines consist of multiple processing units coupled through high speed interconnections. The performance can be increased to higher levels, if Customized Computing Units (CCU) replace partitions of the program that would be normally executed on the Central Processing Unit (CPU) of the general purpose computer. The CCUs usually implement computational intensive parts of the program and can be mapped on hardware. Depending on the content of the kernels, orders of magnitudes speed up can be gained. The CCUs can be mapped on An Application Specific Integrated Circuit (ASIC) can be targeted or on an Field Programmable Gate Array (FPGA). It has been identified that the Sparse Matrix times Dense Vector multiplication (SMVM) is a key component in scientific applications. This thesis presents an CCU for the Double Precision Floating Point SMVM for the OpenFOAM Computation Fluid Dynamic (CFD) tool. Section 1.1 present an introduction and motivation behind the presented work. Subsequently, Section 1.2 defines the thesis goals. Section 1.3 concludes this chapter and gives an overview of the organization of the remaining part of this thesis.

1.1 Problem Statement

With an Computation Fluid Dynamic (CFD) tool the movement and dynamics of flows can be examined. By turning real world object into mathematical models representing the system or environment, the flow of fluids can be simulated and predicted by means of solving physical equations that apply for the particular fluid.

The OpenFOAM (Open Field Operation and Manipulation) CFD toolbox is a scientific application that can be used to simulate a variety of industrial processes. Examples of these processes include but are not limited to complex fluid flows involving chemical reactions, turbulence and heat transfer, solid dynamics, electromagnetics and the pricing of financial options.

ActiFlow [1], a Dutch-based company is interested in OpenFOAM for their modeling activities in the field of the automotive, aerospace, oil and gas industry, the medical industry and the construction industry. The time to finish the simulation processes using traditional approaches is in the order of weeks and any kind of optimization, to decrease the execution time is necessary. The CFD tool, allows fast design verification and prototype-building. One can build a model of the system or device that must be analyzed. By applying real-world physics and chemistry to the model, the software will provide images and data, which predict the behavior of that system.

Predicting the behavior of the models requires enormous computational power. In the past years, the computational power of a single processor is reaching its technological

limitations. Currently, performance increase is obtained by executing the simulations on multiple CPU nodes which run in parallel, that divide the computational load of the modeled environment or system among the CPUs. The communication between the processing nodes is controlled through the Message Passing Interface (MPI) library. Although this approach accelerates the execution of the program, we believe that by utilizing FPGA, the performance can be brought to higher levels.

In this thesis, the approach is taken to accelerate the OpenFOAM solvers beyond the traditional achievements employing reconfigurable hardware. The design of computer architectures on reconfigurable hardware is becoming more popular now that classical drawbacks are diminishing. Field-Programmable Gate Arrays (FPGAs) are constantly improving in terms of IO-bandwidth and area, and they provide a technology platform that allows fast and complex reconfigurable designs.

Silicon Graphics Inc. (SGI) builds supercomputers applicable for scientific applications. By integrating and connecting multiple state of the art CPU cores into one machine supercomputers can be created. The Altix series allow integration of multiple Itanium 2 cores with the optionality to include multiple FPGAs. Each FPGA, a Xilinx Virtex 4(XC4VLX200-FF1513-10) has a direct bandwidth of 6.2 GB/s to the main memory and a total bandwidth up to 22.4 GB/s, when off-chip memory is included.

1.2 Project Goals

1.2.1 Scope

The main goal is to increase the performance of the SimpleFoam solver, by moving time critical software kernels to a hardware CCU unit, compared to the traditional way of accelerating SimpleFOAM using multiple CPU nodes. Moving these kernels to hardware allows to exploit data level parallelism efficiently. The OpenFOAM tool contains hundreds of Mega Bytes of source code and we focus on the most important kernels only.

1.2.2 Objectives

In Chapter 2.7, the Double Precision floating point SMVM is identified as one of the main kernels potentially to be executed in hardware. The hardware design will be mapped on the FPGAs available on the Altix machine. The main thesis goals are:

- Profile the SimpleFoam solver and identify the time critical kernels.
- Design hardware CCU units supporting these kernels.
- Integrate the CCU kernels into the SGI RASC system and accelerate OpenFOAM.
- Integrate the sparse matrix vector product, being one of the identified critical kernels, with the dense matrix vector product implementations proposed in [25].

1.2.3 Related Work

Different proposals for the SMVM designs based on FPGAs are proposed. In [7], a striping algorithm is proposed for matrices that contain limited number of stripes in the Finite Element Method (FEM). In [29], the matrix is divided in vertical sections with low performance results for sparse matrices. In [4], the authors arrange PEs in a bidirectional ring to compute $\mathbf{y} = A^i\mathbf{b}$. The proposed design significantly saves I/O bandwidth due to local storage of the matrix and intermediate results, but the matrix sizes are limited by the FPGA On-Chip memory. In [24], the authors divide the matrix in vertical and horizontal sections. So far, they report the highest peak performances. In our design, we present a new approach which does not require adder trees. Moreover, we are planning to design the PEs of our design in such a way that they are capable of computing both dense and sparse matrix vector multiplications. Section 5.4 contains more details for the related work.

1.2.4 Main thesis contributions

The following contributions can be assigned to this thesis:

- Identification of the performance critical kernels of the OpenFOAM CFD tool. Precisely, the simpleFoam solver is used to profile a mesh containing over 6 millions cells, which is considered a realistic input.
- Analysis and selection of the best Sparse Matrix Format to represent typical matrices of the FVM method.
- The design of an SMVM capable of computing $\mathbf{c} = \alpha\mathrm{A}\mathbf{b} + \beta\mathbf{c}$. For α=$\{-1, 1\}$ and β=$\{-1, 0, 1\}$ in IEEE-754 compliant floating-point format.
- Verification of the SMVM design in simulation and hardware for one and two Processing Elements and verification in simulation for more (four and eight) Processing Elements.
- A set of equations is derived containing the relation between the number of Processing Elements, the required bandwith and the performance of the SMVM kernel.
- Modification of the Processing Element in [25], which computes a dense vector matrix multiplication. The design of the PE is modified in such a way that it is capable of computing sparse and dense vector multiplications. The new design with the combined Processing Elements is verified in simulations. A paramater allows users to select between dense and sparse matrix vector multiplication. Thus, $\mathbf{c} = \alpha\mathrm{A}\mathbf{b} + \beta\mathbf{c}$ can be computed for A being a dense or a sparse matrix, for α=$\{-1, 1\}$ and β=$\{-1, 0, 1\}$.
- A decision rule is derived to select between dense and sparse matrices. In case over 40% of the matrix is filled with non-zero elements, it is more performance efficient to employ the dense sparse matrix multiplication rather then the sparse one.
- A list of bugs for the RASC-core has been identified.

- Customized Computing Units that are integrated to the RASC environment suffer from complex routing issues. A small number of modifications improves significantly the routing issues. Moreover, by iteratively generating multiple bitstreams, in each iteration information can be extracted from the timing report and from it the VHDL design can be adjusted accordingly to improve the performance.

1.3 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 discusses one of the solvers of the OpenFOAM toolbox named SimpleFoam, which is used to simulate moving fluids. The profiling results of this solver are presented also in this chapter. Subsequently, the SMVM kernel is analyzed in Chapter 3. The cost of the matrix storage format, as it is currently in OpenFoam, is analyzed and compared against other storage schemes in terms of required memory bandwidth and memory space. Next, Chapter 4 discusses the hardware unit designed for the CCU SMVM. It includes a description of the hardware platform and the CCU unit that is presented, which is analyzed and compared against software execution time for the stand alone version. Besides software comparison, the hardware unit is also compared to related work in Chapter 5. Chapter 5 includes the interconnection between OpenFOAM and the SMVM and the analysis for the application performance increase due to the hardware acceleration of the SMVM kernel. Finally, this thesis is concluded in Chapter 6, where recommendations for future work can also be found.

Appendix A contains a brief explanation of the code inside the OpenFOAM solver. In Appendix B, a description of the profile commands included with the profiling reports. Appendix C reports the results of the *Metis* and *Simple* decomposition methods using four CPUs. Appendix D discusses a list of bugs and issues regarding the SGI RASC-library. Furthermore, recommendations to improve routing issues for the RASC-reconfigurable FPGA can be found here. Last, Appendix E describes the contents of the CD-ROM containing design files for this thesis.

2 SimpleFoam

OpenFOAM supports different solver applications to best fit the different models. SimpleFoam is a steady state solver used to simulate the movement of incompressible flows. In this chapter, a description of the SimpleFoam application solver will be given, in addition with certain input and output files describing the mesh, solvers etc. This gives an insight and understanding of the basic principles needed to use the OpenFOAM CFD toolbox. In the second part of this chapter, SimpleFoam is profiled. The profiling report is discussed and the most important kernels are briefly explained. Finally, this chapter ends with a conclusion.

2.1 The Navier Stokes equations

SimpleFoam, implements the Navier-Stokes equations. These equations for incompressible fluids contain the relation between pressure and velocity for moving fluids in time and space. The equations are:

$$\frac{\partial \mathbf{U}}{\partial t} + \nabla \cdot (\mathbf{U}\mathbf{U}) = \mathbf{g} - \nabla p + \nabla \cdot (\nu \nabla \mathbf{U}) \quad (2.1)$$

$$\nabla \cdot \mathbf{U} = 0 \quad (2.2)$$

where, $\mathbf{U}$ is the velocity, ν the kinematic viscosity (dynamic viscosity/density), p the kinematic pressure, $\mathbf{g}$ the external body force vector. The first equation, the momentum equation, describes the relation between the velocity, pressure and external forces of the incompressible fluid. The second equations, states that the divergence of the velocity must be zero, which means that the fluid can't expand (positive divergence) nor compress (negative divergence).

2.2 Discretizing continual operations

A computer can only process discrete quantities and therefore continual quantities like time and space must be discretized in order to be processed. A distinction of 3 different discretization processes must be made. Each one of them is explained in the subparagraphs below.

Spatial discretization Spatial discretization is the process of decomposing the mesh into cells which form the basis for the computational grid. Each cell is bounded by its faces, and the center of the cell should be inside the cell. The cells must be contiguous, meaning they have to fill up the whole computational domain without overlapping each

other. Accuracy of the solution can be increased by incrementing the cell account for the same grid size.

Temporal discretization Fluids naturally tend to flow in time depending on physical properties. To compute and approximate such flows in time, the time solution space must be divided into a finite number of intervals.

Equation discretization The partial differential equations that are appropriate for certain cases must be discretized in order to be processed. OpenFOAM solves certain fields, like pressure and speed, through a translation from the partial differential equations and the computational grid into a set of equations. Specifically, the SimpleFoam solver implements the Navier-Stokes equations added with *transport* and *turbulence* equations. The final goal is to calculate the movement of the fluids by solving the set of equations.

The discretized Navier-Stokes equations can be written into a block decomposition:

$$\begin{bmatrix} A & G \\ D & 0 \end{bmatrix} \begin{bmatrix} U^{n+1} \\ p^{n+1} \end{bmatrix} = \begin{bmatrix} r^n \\ 0 \end{bmatrix} + \begin{bmatrix} bc's \\ \end{bmatrix} \tag{2.3}$$

In this equation, matrix G equals the discrete gradient operator, matrix D the divergence operators and matrix A depends on spatial and temporal discretization (depends on mesh and on time interval step). The first row of 2.3 is the discretized version of the momentum equation of Eqn. (2.1) and the second row the descretized form of the second equation of the Navier Stokes equations, Eqn. (2.2). The vector with bc's contain the boundary conditions for the momentum and pressure equations. The pressure must be solved implicitly to ensure the compressibility constraints. The vector r^n is the explicit right hand side of the momentum equations.

Before going into more detail how SimpleFoam solves these equations, the interface to the OpenFOAM toolbox is described, meaning for the input the description of the meshes, select the desired models and solvers, and for the output, the results after solving the case and the post-processing thereof.

2.3 I/O-interface

The easiest way to describe the interface is probably to look at the input and output files and their ordering in folders. Each case is build up, out of 3 folders as depicted in Figure 2.2. Each of the folders, the *system*, *constant* and the *time* directories will be discussed now.

2.3.1 Input files

All the input files in the different folders have a common header, depicted in figure 2.1. Comments are preceded with //. The header consist of 9 entries that contain general information like the OpenFOAM version, location of the file, the directory path and

```
FoamFile
{
//    Keyword     Entry              Description
      version     1.4.1;             // OpenFoam version
      format      ascii;             // Datatype:  ascii or binary input

      root        "/home/FastBak";   // The root directory of the case
      case        "case04b";         // The case name
      instance    "system";          // The subdirectory in the case
      local       "";                // Any subdirectories in instance
                                     // e.g polyMesh in figure 2.2
      class       dictionary;        // Typical a dictionary or a field
      object      controlDict;       // The name of the filename
}
```

Figure 2.1: I/O header

```
. <case name >
|– system
      |– controlDict
      |– fvSchemes
      |– fvSolution
      |– decomposeParDict (optional)
|– constant
      |– ...Properties
      |– polyMesh
            |– points
            |– cells
            |– faces
            |– boundary
|– time directories
```

Figure 2.2: Organization of the directory structure of a case

the file class/type. In the next paragraphs the different files and there classes will be discussed.

2.3.1.1 System folder

The system folder contains at least 3 files that specify the chosen solvers (*fvSolution*), the selected discretization schemes (*fvSchemes*) and timing and storage information (*controlDict*). If the computational domain is divided, in order to solve it on multiple processing

nodes an additional file has to be included (*decomposeParDict*). An example of each of the files will be given.

fvSolution The *fvSolution* file contains for each physical quantity the selected solver scheme. An example of such a file is shown in figure 2.3. A difference must be made here with the application solver like SimpleFoam, which solves a particular problem by a set of equations and likely will use different solvers for its physical quantities. In contrast to the application solver, the solver methods mentioned here are techniques used to solve particular equations. Section 2.4 contains more information about the solvers.

fvSchemes The *fvSchemes* file allows users to select appropriate discretization schemes. Table 2.1 describes the keywords used in this file with their mathematical terms.

Keyword	description
interpolationSchemes	Point-to-point interpolations of values
snGradSchemes	Component of gradient normal to a cell face
gradSchemes	Gradient ∇
divSchemes	Divergence $\nabla\cdot$
laplacianSchemes	Laplacian ∇^2
timeScheme	First and second time derivatives $\frac{\partial}{\partial t}, \frac{\partial^2}{\partial^2 t}$
fluxRequired	Fields which require the generation of a flux

Table 2.1: Keywords in *fvSchemes* [19]

controlDict The *controlDict* file contains time related settings and options to store to the mesh calculated. Figure 2.5 shows an example of such a file.

decomposeParDict In the *decomposeParDict* file, the user can specify which decomposition technique will be used and how many processing nodes the mesh will be divided on.

2.3.1.2 Constant folder

The constant folder contains the description of the mesh in the polyMesh subfolder and some physical models depended on the application. SimpleFoam requires a selected *turbulence* model in the file turbulenceProperties and a *transport* model in the file *transportProperties* models, while for example a totally different solver application could use a *perfectGas* model.

```
// * * * * * * * * * * * * fvSolution file:  solver properties * * * * * * * * * * * * //

solvers
{
        p GAMG
        {
            tolerance                                   1e-06;
            relTol                                      0.1;

            smoother                                    GaussSeidel;
            nPreSweeps                                  0;
            nPostSweeps                                 2;

            cacheAgglomeration                          true;

            nCellsInCoarsestLevel                       10;
            agglomerator                                faceAreaPair;
            mergeLevels                                 1;
        };

        U smoothSolver
        {
            smoother                                    GaussSeidel;
            nSweeps                                     1;
            tolerance                                   1e-8;
            relTol                                      0.1;
        };

        k smoothSolver
        {
            smoother                                    GaussSeidel;
            nSweeps                                     1;
            tolerance                                   1e-8;
            relTol                                      0.1;
        };

        epsilon smoothSolver
        {
            smoother                                    GaussSeidel;
            nSweeps                                     1;
            tolerance                                   1e-8;
            relTol                                      0.1;
        };

        R smoothSolver
        {
            smoother                                    GaussSeidel;
            nSweeps                                     1;
            tolerance                                   1e-8;
            relTol                                      0.1;
        };

        nuTilda smoothSolver
        {
            smoother                                    GaussSeidel;
            nSweeps                                     1;
            tolerance                                   1e-8;
            relTol                                      0.1;
        };
        SIMPLE
        {
            nNonOrthogonalCorrectors                    0;
            pRefCell                                    0;
            pRefValue                                   0;
        }

        relaxationFactors
        {
            p                                           0.3;
            U                                           0.7;
            k                                           0.7;
            epsilon                                     0.7;
            R                                           0.7;
            nuTilda                                     0.7;
        } // ************************************************************************* //
```

Figure 2.3: The *fvSolution* file

```
/* * * * * * * * * * * * fvSchemes file:  Discretization properties * * * * * * * * * * * */

ddtSchemes
{
        default                                 steadyState;
}

gradSchemes
{
        default                                 Gauss linear;
        grad(p)                                 Gauss linear;
        // grad(p)                              cellLimited Gauss linear 1.0;
        grad(U)                                 cellLimited Gauss linear 1.0;
        grad(nuTilda)                           cellLimited Gauss linear 1.0;
}

divSchemes
{
        default                                 none;
        div(phi,U)                              Gauss linearUpwind cellLimited Gauss linear 1.0;
        div(phi,k)                              Gauss limitedLinear 1.0;
        div(phi,epsilon)                        Gauss limitedLinear 1.0;
        div(phi,R)                              Gauss limitedLinear 1.0;
        div(R)                                  Gauss limitedLinear 1.0;
        div(phi,nuTilda)                        Gauss limitedLinear 1.0;
        div((nuEff*dev(grad(U).T())))           Gauss linear;
}

laplacianSchemes
{
        default                                 none;
        laplacian(nuEff,U)                      Gauss linear limited 0.333;
        laplacian((1|A(U)),p)                   Gauss linear limited 0.333;
        laplacian(DkEff,k)                      Gauss linear limited 0.333;
        laplacian(DepsilonEff,epsilon)          Gauss linear limited 0.333;
        laplacian(DREff,R)                      Gauss linear limited 0.333;
        laplacian(DnuTildaEff,nuTilda)          Gauss linear limited 0.333;
}

interpolationSchemes
{
        default                                 linear;
        interpolate(U)                          linear;
}

snGradSchemes
{
        default                                 limited 0.333;
}

fluxRequired
{
        default                                 no;
        p;
}

// * * * * * * * * * * * * * * * * * * fvSchemes file * * * * * * * * * * * * * * * * * * //
```

Figure 2.4: The *fvSchemes* file

2.3.1.3 Time folder

Before solving a certain case, the time folder requires a folder, named after its first start time (according to the keyword *startTime* in the controlDict file and usually equals 0), with each physical property at that time. For the simpleFoam solver, expected files could be the pressure p, velocity U, and some other quantities which depend on the selected transport and turbulence model. Each file contains the initial values on each cell for the entire mesh.

```
// * * * * * * * * * * * * * * * * * * * * controlDict file:  time and data I/O control * * * * * * * * * * * * * * //

        application                             simpleFoam;
        startFrom                               startTime;
        startTime                               0;
        stopAt                                  endTime;
        endTime                                 60;
        deltaT                                  1;
        writeControl                            timeStep;
        writeInterval                           10;
        purgeWrite                              0;
        writeFormat                             binary;
        writePrecision                          6;
        writeCompression                        compressed;
        timeFormat                              general;
        timePrecision                           6;
        graphFormat                             xmgr;
        runTimeModifiable                       yes;


// * * * * * * * * * * * * * * * * * * * * * * * * * controlDict * * * * * * * * * * * * * * * * * * * * * * * * * //
```

Figure 2.5: The *controlDict* file

```
// * * * * * * * * * * * * * * * * decomposeParDict file:  mesh decomposition * * * * * * * * * * * * * * * * * * //

        arguments                               "" off off;

        numberOfSubdomains                      2;

        method                                  metis;

        simpleCoeffs
        {
            n                                   (1 1 1);
            delta                               0.001;
        }

        hierarchicalCoeffs
        {
            n                                   (1 1 1);
            delta                               0.001;
            order                               xyz;
        }

        manualCoeffs
        {
            dataFile                            "";
        }

// * * * * * * * * * * * * * * * * * * * * * * * * * controlDict * * * * * * * * * * * * * * * * * * * * * * * * * //
```

Figure 2.6: The *decomposeParDict* file

2.4 Solvers

OpenFOAM users have the option to select between different solver which suites them best. In figure 2.3 an example is given where 2 different solvers have been used. A Gauss-Seidel smooth solver and a Geometric Algebraic Multigrid solver. A full list of solvers can be found in the openFOAM userguide.

2.5 SimpleFoam

The previous sections concentrated on the Input Output of the CFD tool. In this section, we continue with the description of the algorithm behind the *SimpleFoam* solver. In subsection 2.5.1 an explanation of the discretized pressure equation that is solved in SimpleFoam will be given. The algorithm behind SimpleFoam, the SIMPLE algorithm,

is described in 2.5.2.

2.5.1 The Pressure Equation

The pressure equation can only be solved implicitly and must be solved from a semi-descretized form of the momentum equation, presented in Eqn. (2.4).

$$a_p \mathbf{U}_p = \mathbf{H}(\mathbf{U}) - \nabla p \tag{2.4}$$

$$\mathbf{H}(\mathbf{U}) = -\sum_f a_N \mathbf{U}_N + \frac{\mathbf{U}^0}{\Delta t} \tag{2.5}$$

$$\mathbf{U}_p = \frac{\mathbf{H}(\mathbf{U})}{a_p} - \frac{\nabla p}{a_p} \tag{2.6}$$

$$\mathbf{U}_f = (\frac{\mathbf{H}(\mathbf{U})}{a_p})_f - \frac{(\nabla p)_f}{(a_p)_f} \tag{2.7}$$

$$\nabla \cdot \mathbf{U} = \sum_f \mathbf{S} \cdot \mathbf{U}_f = 0 \tag{2.8}$$

$$\nabla \cdot \frac{\nabla p}{a_p} = \nabla \cdot (\frac{\mathbf{H}(\mathbf{U})}{a_p}) = \sum_f \mathbf{S} \cdot (\frac{\mathbf{H}(\mathbf{U})}{a_P})_f \tag{2.9}$$

The term $\mathbf{H}(\mathbf{U})$ in this equation consist of 2 parts, a transport part and a source part, rewritten in Eqn. 2.5. The transport part contains the matrix coefficients for all neighbours a_N multiplied by their corresponding velocities. The source part contains the transient term of the velocity. Eqn. (2.6) is obtained from Eqn. (2.4) by expressing for $\mathbf{U}_p$. The next equation, Eqn. (2.7) is derived from the previous one by face-interpolating the cell center values. By substituting Eqn. (2.6) into the left had side of Eqn. (2.8) and Eqn. (2.7) into the right handside of Eqn. (2.8), Eqn. (2.9) is obtained. Eqn. (2.9) is the expression used to solve for the pressure.

The speed and pressure equations, i.e. Eqn. (2.9) and Eqn. (2.6) are coupled. A segregated approach by Patankar deals with inter-equation coupling and solves them in sequence. The SIMPLE algorithm is explained in the coming section.

2.5.2 The SIMPLE algorithm

The Semi-Implicit Method for Pressure-Linked Equations allow to solve the Navier-Stokes equations with the additively transport equations to be solved in an iterative procedure. It consist of the following steps:

1. Set the boundary conditions.
2. Solve the discretized momentum predictor.
3. Compute the cell face fluxes.
4. Solve the pressure equation and apply under-relaxation

5. Correct and adjust the cell faces.
6. Correct the velocities on the basis of the new pressure field.
7. Update the boundary conditions
8. Repeat, until the convergence criteria are satisfied

The code is explained in more detail in Appendix A.

2.6 High Performance Computing

This section will contain information regarding parallel computing of the domain. To use the SimpleFoam solver in parallel, the computational grid must be divided in parts. In section 2.6.1 the decomposition of meshes will be described.

2.6.1 Mesh decomposition

The techniques that OpenFOAM uses are listed in the *decomposeParDict* file in Figure 2.6 and each one of them will be discussed on the following 2 properties:

- A fairly distributed work-load balance among the processors.
- Minimizing the communication cost.

2.6.1.1 Simple decomposition

In the simple decomposition technique, the mesh is divided into pieces along the xyz-axes, e.g. 1 piece in the x and z direction and 4 pieces in the y-direction. Using this technique, only the first metric, a fairly distributed work-load balance among the processors is guaranteed. However the algorithm behind this decomposition totally ignores communication, and as a consequence lower performance can be expected. The technique is simple and fast and can lead to good decomposition results for regular structured meshes.

The algorithm behind the Simple decomposition method, will be explained by means of an example. Consider a structured mesh with 32 nodes, with 4 nodes in the x- and y- direction and 2 nodes in the z-direction shown in Figure 2.7. And n=242, i.e. 2 processors in the x-direction denoted as n_{procx}, 4 in the y-direction denoted as n_{procy} and 2 in the z-direction denoted as n_{procz}.

The algorithm uses 2 lists, *pointIndices* and *finalDecomp*. The first list is a temporary list used to keep up the indexes of the nodes, sorted in a certain direction (x y or z). Each value at position i is initialized with i for the list (*pointIndices*[i]=i). For each node, finalDecomp list stores the index of the processor where the node will map to.

The first step is to reorder the pointIndices list, according to the sorting of the nodes along their x-coordinate. This means *pointIndices*[0] will hold the index of the node with the smallest x-coordinate and each next element will have an equal or large x-coordinate value. The next step is to map the sorted nodes on the number of processors in the x-direction. The result of this step is shown in figure 2.7.a. The numbers in the figure

represent the processor values where the node will be mapped on to, i.e., the values so far in *finalDecomp*. Since n_{procx} is 2, the former 16 values of pointIndices will be mapped to processor 0 and the later 16 to processor 1.

The same steps are repeated, but now for the y-coordinates. There are 4 processors in this direction, therefore the 8 nodes with the lowest y-coordinates will be mapped on processor 0, the next lowest 8 on processor 1 etc... This is drawn in figure 2.7.b.

The results from figure 2.7.b and 2.7.c are now combined to get the resulting mesh in figure 2.7.d by the following formula:

$$finalDecomp_C = finalDecomp_A + n_{procx} * map_y \tag{2.10}$$

where $finalDecomp_A$ the mapping as in figure 2.7.b, map_y the mapping as in figure 2.7.c and $finalDecomp_C$ the resulting mapping in figure 2.7.d.

The final steps of the algorithm consist of including the processors in the z-direction to the *finalDecomp* list. Figure 2.7.e shows the mapping of the nodes to the processors in the z-direction. Since there are 2 processors along this direction, the 16 nodes with the smallest z-coordinates will be mapped to processor 0 and the others to processor 1. Figure 2.7.f contains the final decomposition by using the following formula:

$$finalDecomp_E = finalDecomp_C + n_{procx} * n_{procy} * map_z \tag{2.11}$$

where map_z the mapping shown in figure 2.7.e and $finalDecomp_E$ the final decomposition shown in figure 2.7.f.

The number of nodes for this mesh, 32, is a multiple of n_{procx}, n_{procy} and n_{procz}. Therefore the figures 2.7.b, 2.7.c and 2.7.d had all mappings where each processor got assigned an equal amount of nodes. Consider a mesh with 104 nodes, and where the processors in each direction are: $n_{procx} = 2$, $n_{procy} = 4$ and $n_{procz} = 5$. Since 104 is not a multiple of n_{procz} a non-equal node distribution will take place. Each processor in the z-direction gets assigned $\lfloor \frac{104}{n_{procz}} \rfloor = 5$ processors, except for the first $104\%n_{procz} = 4$ which have to process one node more.

The whole approach tries to fairly distribute the cells but disregards the connectivity of the mesh and therefore also the communication cost. While for regular meshes and cubes communication can be minimized, the opposite could be the case for unstructured meshes. Consider a fragment of a mesh in Figure 2.8 where the mesh could be divided at the vertical lines 1 and 2. Cutting the mesh on line 1 will lead to more communication traffic at runtime than the case where the mesh is cut at line 2, since more nabour cell information has to be exchanged.

2.6.1.2 Hierarchical decomposition

The hierarchical decomposition decomposes the mesh in the same way as in the simple decomposition method, but this method allows different ordering in the x,y and z direction [19]. Like in simple decomposition no edge connectivity is taken into account.

2.6.1.3 Metis decomposition

Metis is a software package that partitions large irregular graphs, large meshes, and computing fill-reducing orderings of sparse matrices. The Metis application is used to

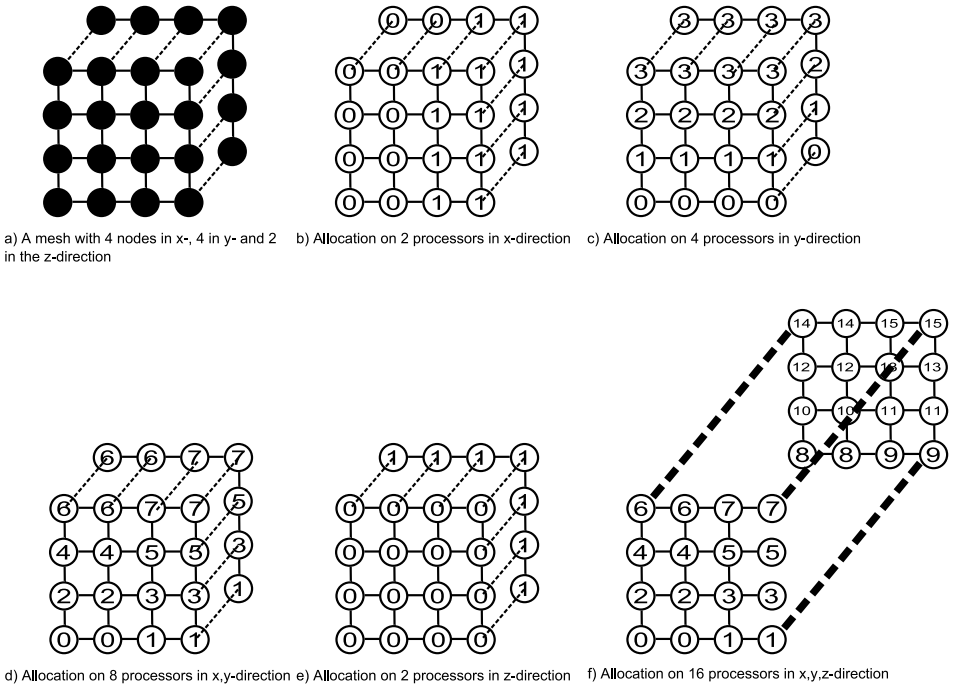

Figure 2.7: Decomposing meshes with simple decomposition, the numbers represent the processor where the nodes map on.

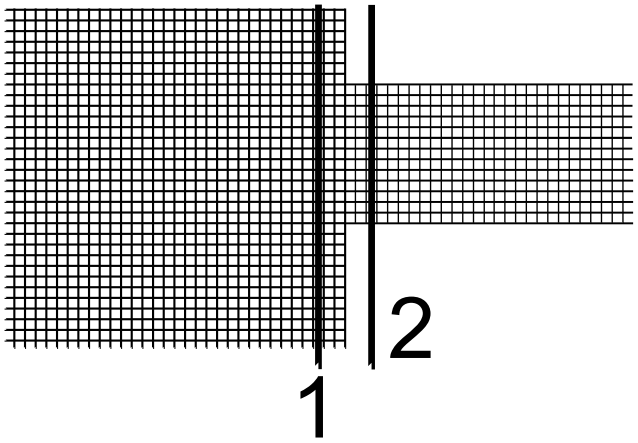

Figure 2.8: Dividing meshes on different places.

partition large meshes. The objective is to minimize the communication cost and to achieve a fair distributed load-balance. These objectives can be realized by computing a balanced k-way partitioning such that the numbers of edges shared between different processor boundaries is minimized.

Minimizing the Communication Volume Minimizing the edgecut, is an approximation of the true communication cost and the partitioning algorithm can minimize the true communication cost reasonably well. Given a graph G = (V, E), and let P be a vector with size —V—, such that P[i] stores the number of processor partitions which belongs to vertex i. The edgecut is defined as the number of edges, which its vertices belong to different processor partitions. That is, the number of edges (v,u) for which P(v) != P(u).

Let Vb $\subset$ V be the subset of the boarder vertices. That is, each vertex v in Vb, is connected to at least one vertex that belongs to a different partition. For each v in Vb let Nadj[v] be the number of Sub-domains that the vertices adjacent to v belong to, without its own partitioning P[v]. The total volume is now defined as the sum of all Nadj[v] for each v in Vb. During computations, each processor interface vertex must be sent to all of its Nadj[v] partitions, which means, that the sum corresponds to the total communication volume.

By minimizing the *totalv*, the overall communication cost will be directly minized, this in contrast to minimizing the edgecut which is an approximation for the same. The results are comparable for well-shaped finite elements, because the degrees of the vertices are similar and the different objectives (*totalv* and the edgecut) behave the same. In terms of the amount of time required by these 2 partitioning objectives, minimizing the edgecut is faster then minimizing the *totalv* [17].

Interface with Metis The various functionalities that Metis provides are accessible through the MetisLib library. The Metis decomposition utility used in OpenFOAM makes use of two different methods called: k-way and recursive. The algorithms used in the k-way and recursive methods are developed by G. Karypis and V. kumar [17]. The library functions that are called from the OpenFOAM decomposePar tool are *METIS_PartGraphRecursive* and *METIS_PartGraphKway*

Graph data structure Metis expects the graph in compressed storage format (CSR) format. Its a wildly used scheme for storing sparse graphs. Two arrays called xadj and adjncy are used to present the graph. The index element xadj[i] holds the startindex of the neighbours list of node i, in the array adjncy. The neighbours are stored in consecutive order in the adjncy matrix. The last neigbour belonging to node i can be found by xadj[i+1]-1, which says, take the start index of the next node and substract one from it. The size of xadj equals the amount of vertices +1, and the size of adjncy equals twice the amount of edges, since each edge is stored twice. If the processors have different computing capacities, a weight list can be added. This processor weight list allows for example a 2-way partitioning in such a way that first processor takes 70% of the weight and the latter only 30%. To use the same functionality, but included with a processor weight list, one of the following functions will be called: METIS_WPartGraphRecursive or METIS_WPartGraphKway.

The objectives of these functions are minimizing the edgecut. If one wants to minimize for the *totalv*, the call function must be changed to METIS_PartGraphVKway or METIS_WPartGraphVKway.

2.6.1.4 Manual decomposition

In manual decomposition, the user has to specify the allocation of the cells to the available processors. This to support decomposition methods other then the ones described above possibly generated with other tools.

Each decomposition method calculates the allocation of cells to the processors. After the allocation the cells and faces have to be distributed to each processor. The boundaries of each created submesh need be closed with processor patches. Each processor patch contains all the faces which 2 processors share.

2.7 Profiling the SimpleFoam solver

Identifying hot-spots in applications allow engineerings to mark functions as potential candidates that can be accelerated. Applying optimization techniques on these hot-spots results in a faster execution time of the program. This section focuses on the hot-spots for the SimpleFoam solver for a certain input parameter set which conclusions are made from. To do so, the platform, the profiling method and the results will be discussed in this chapter. SimpleFoam is one of the many solvers used in the OpenFOAM toolbox.

2.7.1 Profiling platform

The platform, the full hardware architecture on which the application runs on, influences the profiling report in 2 ways:

- The profiler can call specific hardware counter units, architecture depended, to assist in the profiling.
- Certain functions can execute faster on different platforms due to dedicated hardware execution units.

The platform that is used to profile the SimpleFoam solver is the Silicon Graphics, Inc. (SGI) Altix 450 midrange server. A blade architecture is used to connect dual-socket blades containing a dual-core Intel Itanium-II chip. The Blade-to-NUMAlink^{TM} architecture supports the integration of eight choices of blade servers, each with interchangeable memory, I/O and compute power. The latency over this NUMAlink^{TM} using Message Passing Interface (MPI) for a short message is 1 microsecond with a bandwidth for unidirectional throughput of maximum 3.2 GB/s. In addition, the NUMAflex^{TM} architecture makes memory global visible for each processor. The experiment for the profiling will take place on 1 CPU-core, 2 CPU-cores and 4 CPU-cores communicating through Message Passing Interface (MPI).

2.7.2 Compiling OpenFoam on the Altix 450

To compile OpenFOAM on the Itanium-II the following changes had to be made:

- The first problem encountered is that system GCC (version 4.1.2), pre-installed on the Altix by SGI, could not create any OpenFOAM libraries. Therefore a local GCC (4.3.0) was build in order to start the compilation.

- In the file /src/OpenFOAM/global/new.C *::abort()* was changed to *abort()*
- Copied pointPatchFieldFunctions.H from http://openfoam.cfd-online.com/forum/messages/126/5122.html?1187081100 to OpenFOAM-1.4.1/src/OpenFOAM/fields/pointPatchFields/pointPatchField/
- Copied /usr/include/c++/4.1.2/ia64-suse-linux/bits/c++locale.h (from the system GCC) to the corresponding location inside the local GCC folder.
- Modified the wmake rules in OpenFOAM-1.4.1/wmake/rules/linuxIA64I64. Each occurrence of -kPIC has been changed to -fPIC.

All these steps where necessary to successful compile the OpenFOAM CFD toolbox. The openMPI version included with the OpenFOAM is not used. We used the native MPI version of the Altix 450 machine, since it is already optimized for the Altix machine.

2.7.3 Profiling method

A large number of profilers exist. The task of the profiler is to report bottlenecks in terms of execution time so that designers can concentrate on accelerating them. From the report the exact functions or lines of code where most CPU cycles are spent can be seen. The report should contain accurate information, should be complete and preferably generated fast.

Extra effort, compared to normal execution, must be done in order to be able to profile the software and to find the bottlenecks. Different profiling techniques exist and the most common ones have been summarized here.

Source-level instrumentation Source-level instrumentation involves altering the source code that eventually becomes the application by inserting profiling code. A programmer can add explicit calls to the profiling API or this can be done automatically through a pre-processor.

Compile-time instrumentation The compiler itself can insert extra code which is needed for the profiling. This has the advantage above source-level instrumentation of being more convenient, but requires the source code to be recompiled.

Offline binary instrumentation Binary images that contain the text sections for shared libraries or applications can be rewritten to add instrumentation. This technique is complex to implement.

On-line binary instrumentation Mapped binary images are rewritten to add instrumentation.

Simulation A simulator can easily collect detailed data as part of the simulation run. Such techniques tend to be very reliable and slow, so they are used when the level of detail is critical.

Sampling In contrast to the software instrumentation methods above, sampling does not require any instrumentation. Sampling can be based on 3 different methods:

- Instruction Pointer (IP) Sampling. This is the most commonly used method of the 3.
- Hardware Event Sampling on application level or procedural level.

- Call Stack Sampling.

Itanium supports monitoring of more then 100 hardware events and can give precise information about the address and instruction pointer which can be used to generate the profiling report. The easiest and fastest method to profile SimpleFoam on the Itanium-II processor is to collect data received from the hardware Performance Monitoring Unit (PMU), since it has the advantage above the software instrumentation techniques that it does not require any binary of library to be recompiled for procedural profiling. In addition, the overhead in terms of execution time profiling-enabled will be minimal. The Performance Monitoring Unit (PMU) is integrated into the Linux kernel and available through the Perfmon API. Different profiling tools exist for the Itanium-II that are based on one or more of the sampling techniques. Some of these profiling tools are Vtune, Pfmon, profile.pl and histx.

Vtune Intels VTune performance counters monitor events inside Intel microprocessors to give a detailed view of application behavior. VTune provides time- and event-based sampling, call-graph profiling and hotspot analysis. It collects and analyzes software performance data from a systemwide view down to specific functions or instructions in the source application.

pfmon The pfmon tool is a performance monitoring tool designed for Linux. It uses the Itanium (PMU) to count and sample unmodified binaries. In addition, it can be used for the following tasks [12]:

- To monitor unmodified binaries in its per-CPU mode.
- To run system-wide monitoring sessions. Such sessions are active across all processes executing on a given CPU.
- Launch a system-wide session on a dedicated CPU or a set of CPUs in parallel.
- Monitor activities happening at the user level or at the kernel level.
- Collect basic hardware event counts.
- Sample program or system execution, monitoring up to four events at a time.

Profile.pl The Perl script profile.pl provides a simple way to do procedure-level profiling of a program running on the Altix system. It requires that symbol information is present in the program text. The profile.pl script handles the entire user program profiling process and is designed for Altix machines. Profile.pl is actually an interface to the pfmon, in a more user-friendly way. In addition it is very easy to use with MPI.

histx Like the profile.pl script, the histx module is also developed by SGI. The histx module is a set of tools used to assist in application performance analysis. It includes three data collection programs and three filters for performance data post-processing and display. The following sections describe this set of tools. The programs can be used to gather data for later profiling:

- histx: A profiling tool that can sample either the program counter or the call stack.
- lipfpm: Reports counts of desired events for the entire run of a program.

- samppm: Samples selected counter values at a rate specified by the user.

Like the profile.pl script, using mpirun with histx is very simple. Both are developed by SGI to make the profiling easy on the Altix machines. The profiler we selected to profile SimpleFoam, is the histx tool script for the simplicity of accessing hardware counters. The option is chosen to use the Instruction Pointer sample since the callstack for the application can be huge and a depth of it must be specified. Line level information is not included. Appendix B.1 contains all the commands used to start the profiling.

2.7.4 Profiling input parameters

The description of the input parameter set will be presented by means of the I/O files described in Section 2.3. Figure 2.3 and 2.4 contain the selected solvers for the physical quantities and the selected descretized schemes respectively. Figure 2.5 contains the file with the timing related information, such as the number of time steps taken. Profiling on multiple CPUs requires the mesh to be decomposed. Figure 2.6 shows a set-up for 2 subdomains (2 CPUs in this case) with the Metis decomposition technique selected. By changing the value of the keyword *method* to *simple* the simple decomposition technique will be selected with the options under *simpleCoeffs*.

Mesh stats	
points	2886846
edges	12400342
faces	16371921
internal faces	16029396
cells	6858424
boundary patches	10
point zones	0
face zones	0
cell zones	0

Table 2.2: Statistics of the properties of the mesh

Table 2.2 and Table 2.3 show certain properties of the mesh. The former table contains global information on the number of cells, faces etc. while the later table contains data of the numbers of types of cells inside the mesh. Each run will be executed for an amount of 60 time steps as stated in Figure 2.5.

2.7.5 Profiling results

This section gives an overview of the results of the profiling of SimpleFoam. Since C++ functions can have very long names, due to inheritances and namespaces the tables with the profiling results will obtain short references to their real names. The full names and the references to them can be found in Appendix B.2. Table B.1 contains the profiling report for the candidate functions that could be implemented in hardware. Information is gathered running the application on one CPU and two CPUs. In the case of two CPUs

Types of cells	
hexahedra	0
prisms	4952537
wedges	0
pyramids	15084
tet wedges	0
tetrahedra	1890803
polyhedra	0

Table 2.3: Statistics of the structures of the cells of the mesh

both Metis and simple decomposition techniques have been used. The profiling results using the configuration with 4 CPUs are shown in Table B.2 and B.3 for Metis and Simple decomposition respectively. The discussion of the potential hardware candidates continues in Section 2.7.6.

The execution time, during the profiling, for each test case is summarized in Table 2.4. The first column obtain the number of CPUs SimpleFoam is executed on and the selected decomposition method. The second column contains the user time and the last column the elapsed wall time. The elapsed wall time is the actual time taken by a computer to complete a task. It includes the CPU time, I/O time and the communication time between the multiple CPUs. The presented numbers in the table are directly retrieved from the application solver. The time needed to decompose and reconstruct the mesh are shown in Table 2.5 and 2.6 respectively. The table shows for each testcase the user time to decompose or reconstruct the mesh. The linux time command has been used here. The execution time, to decompose and reconstruct the mesh is for all test cases similar and negligible compared to the execution timing of the solver. However, the selected decomposition method does effect the execution time as can be seen from Table 2.4 and Table 2.7. Table 2.7 shows the achieved speed up by using multiple cores. The numbers presented in this table are directly calculated from Table 2.4 by dividing the execution time for each case with the execution time for one CPU. A number of statements regarding the selected decomposition method can be concluded:

- The number of cells shared between different processors is more balanced with Metis decomposition. The decomposition reports for four CPUs are included in Appendix C.
- The relative time spend in the kernels is approximately the same for all the CPU nodes when the Metis decomposition is select. There is more variation when the simple decomposition method is selected.
- The number of faces shared between the processors is more then 50% higher when simple decomposition is selected.
- For the given mesh, running the application on both two and four CPUs using Metis decomposition resulted in lower execution time.

#CPUs - Method	User time in seconds	Wall Time in seconds
1 - none	10330.6	10352
2 - Metis	4812.74	4838
2 - Simple	5057.15	5091
4 - Metis	3398.48	3428
4 - Simple	3267.72	3359

Table 2.4: Execution time running *SimpleFoam* for two decomposition methods for one, two and four CPUs

#CPUs-Method	User Time in seconds
2 - Metis	315.716
2 - Simple	307.885
4 - Metis	316.678
4 - Simple	310.951

Table 2.5: Execution time running *decomposePar*

2.7.6 Hardware candidates

The profiling shows the potential functions that could be implemented in hardware. First these functions must be analyzed whether they are suitable to be implemented in hardware. Potential bottlenecks are usually coming from 4 different types of processes: CPU-bound processes, Memory-bound processes, I/O bound processes and communication. Functions where the CPU is the bottleneck are favorite to be implemented in hardware as data level parallelism can be exploited in such cases. The following functions are worth to be investigated to be implemented in hardware:

- smooth
- residual
- Amul
- limitFace
- grad_1
- grad_3

Appendix B.2 contains the library, namespaces and classes in which the functions belong and what arguments they take. The grad functions are difficult to be implemented in hardware and are outside the scope of this thesis, since they are build on a deep-level of classes. The other kernels are basically written in C and are basically independent functions. Examining these functions, some kernels consist of a communication part in which data between processors is exchanged and some are pure computational intensive. Table 2.8 shows for these kernels the relative time that is spend in the computation and

#CPUs- Method	**User Time in seconds**
2 - Metis	284.370
2 - Simple	294.648
4 - Metis	291.381
4 - Simple	293.413

Table 2.6: Execution time running *reconstructPar*

#CPUs-Method	**Speed up** - Execution Time	**Speed up** - Wall Time
2 - Metis	2.147	2.140
2 - Simple	2.043	2.033
4 - Metis	3.040	3.020
4 - Simple	3.161	3.082

Table 2.7: Measured speed up using multiple CPUs compared to the execution time of 1 CPU

communication. Four CPUs are used and information for the Host is recorded only. From the table the conclusion can be made that all kernels are dominated by computations and that communication time is neglible small. A study of each of the kernels which are more or less implemented in C are given now. We do not focus on the grad functions, due to the complex C++ structures implemented in it.

4 CPU's. Recorded on Host CPU. Methis Decomposition used.		
Function	**Computation**	**Communication (and other)**
Smooth	91.20 %	8.80 %
residual	99.87 %	0.13 %
Amul	99.85 %	0.14 %

Table 2.8: Relative time spent in communication and computation

2.7.6.1 smooth

```
/***** Computational intensive part of the smooth function *****/

register scalar curPsi;
register label fStart;
register label fEnd = ownStartPtr[0];

for (register label cellI=0; cellI<nCells; cellI++)
{
        // Start and end of this row
        fStart = fEnd;
        fEnd = ownStartPtr[cellI + 1];

        // Get the accumulated neighbour side
        curPsi = bPrimePtr[cellI];

        // Accumulate the owner product side
        for (register label curFace=fStart; curFace<fEnd; curFace++)
        {
                curPsi -= upperPtr[curFace]*psiPtr[uPtr[curFace]];
        }

        // Finish current psi
        curPsi /= diagPtr[cellI];

        // Distribute the neighbour side using current psi
        for (register label curFace=fStart; curFace<fEnd; curFace++)
        {
                bPrimePtr[uPtr[curFace]] -= lowerPtr[curFace]*curPsi;
        }

        psiPtr[cellI] = curPsi;
}
```

The smooth function smoothes the error and residual norms over time and tries to improve the accuracy of the solver. The smooth function can be divided into 2 sections, a part that communicates with other processors to exchange boundary information with neighbour processing nodes, and a part that does the actual computation. The communication cannot be accelerated in hardware and therefore the concentration will be on the computational part. The box contains the computational part of the code part of the smooth function. It contains 3 loops that possibly could be unrolled. The 2 inners loops are similar in nature and can be unfolded to a three based model. The outerloop needs more investigation: the line curPsi = bPrimePtr[cellI] could depend on a previous iteration, since bPrimePtr is updated after its being read from.

2.7.6.2 limitFace

```
template<>
inline void cellLimitedGrad<scalar>::limitFace
(
    scalar& limiter,
    const scalar& maxDelta,
    const scalar& minDelta,
    const scalar& extrapolate
)
{
    if (extrapolate > maxDelta + VSMALL)
    {
        limiter = min(limiter, maxDelta/extrapolate);
    }
    else if (extrapolate < minDelta - VSMALL)
    {
        limiter = min(limiter, minDelta/extrapolate);
    }
}

template<class Type>
inline void cellLimitedGrad<Type>::limitFace

(
    Type& limiter,
    const Type& maxDelta,
    const Type& minDelta,
    const Type& extrapolate
)
{
    for(direction cmpt=0; cmpt<Type::nComponents; cmpt++)
    {
        cellLimitedGrad<scalar>::limitFace
        (
            limiter.component(cmpt),
            maxDelta.component(cmpt),
            minDelta.component(cmpt),
            extrapolate.component(cmpt)
        );
    }
}
```

In Appendix B.2, the specific type of the template class for the LimitFace is Vector. The Vector consist of 3 components each direction containing a double. The function can be parallelized by calculating all the upper and lower limits for each component of

the class Type in parallel.

2.7.6.3 Amul

```
register const label nCells = diag().size();
for (register label cell=0; cell<nCells; cell++)
{
    TpsiPtr[cell] = diagPtr[cell]*psiPtr[cell];
}

register const label nFaces = upper().size();
for (register label face=0; face<nFaces; face++)
{
    TpsiPtr[uPtr[face]] += upperPtr[face]*psiPtr[lPtr[face]];
    TpsiPtr[lPtr[face]] += lowerPtr[face]*psiPtr[uPtr[face]];
}
```

The Amul functions consist of a communicational and a computational part. The performance as a consequence of irregular memory pattern accesses results in low performance. On top of that, its current format allows a low degree of parallelism. Different sparse matrix formats will be analyzed in the next chapter.

2.7.6.4 residual

The residual function is similar to the Amul function, but instead of a matrix vector multiplication unit, a vector minus matrix vector multiply unit is performed here. Since it is nearly the same as the Amul, we do the analysis for the Amul function and extract this kernel from the Amul kernel.

2.8 Conclusion

In the first part of this chapter, a number of aspects of the OpenFOAM CFD tool are described. In particular, the focus is on the SimpleFoam solver. The Navier-Stokes equations describe motion of fluids in space. The mesh, representing the computational grid, is read by the CFD tool through files together with properties of the solvers, the discrete operators and timing options. The files presented here form the basis for the profiling in the next chapter. Finally, the decomposition methods are described that support multiple CPU nodes to decrease simulation execution time in solving the grid.

The second part described the profiling process. The best way to profile applications on the Itanium processor is to use specific additional hardware instrumentation for profiling, like Instruction Point (or Program Counter) sampling. The SimpleFoam solver has been profiled on 1, 2 and 4 CPUs with a mesh containing over 6 million cells. Two different decomposition methods have been analyzed and compared. The *Metis* decomposition method was found to be superior over the *Simple* method since it was able to reduce the communication cost and lead to faster execution times. The total percentage of the most important kernels found during profiling on 1, 2 and 4 CPUs are

# CPUs	Decomposition method	CPU-node	Total % of kernels
1	None	Host	36.852
2	Metis	Host	32.784
		Slave-1	34.214
2	Simple	Host	32.116
		Slave-1	34.880
4	Metis	Host	25.120
		Slave-1	25.618
		Slave-2	25.350
		Slave-3	25.549
4	Simple	Host	25.490
		Slave-1	27.254
		Slave-2	26.263
		Slave-3	26.014

Table 2.9: Summary of the most important kernels of Table 2.10 from the profiling report. The contribution of these kernels is reported in percentages compared to the total application time. Each active node, being the host or slave is included.

summarized in Table 2.9. The table contains for each active CPU, the host or slaves, the sum percentage of the main kernels reported in Tables B.1, B.2 and B.3.

Table 2.10 summarizes for each kernel the feasibility to be implemented in hardware. In the table arguments are given for the selected hardware candidates. The Amul, residual and smooth functions, are the most important kernels and as a consequence of timing issues, only the Amul and residual kernel will be investigated in the next chapter. We selected this kernels for the following reasons:

- The Amul and residual functions are general kernels in the whole OpenFOAM toolbox.
- Table 2.8 shows that the communication part in the Amul and residual functions is negligible.

Function	**Type of bottleneck**	**Description**	**Hardware implementation?**	**Implemented in this thesis in hardware?**
smooth	MEM-CPU	Smooth the error and residual norms over time and improving accuracy of the solver.	Yes, contains for loops with lots of multiply-add instructions that with unrolling could possibly lead to faster execution time in hardware.	No
grad_1	Not investigated	--	Possibly, complexity to understand the C++ structures and translate to the hardware for this function falls outside the time of this thesis.	No
limitFace	CPU	Limit Face values.	Yes, its a relative small function. The functions grad_1 and grad_2 include a call to this function.	No
grad_3	Not investigated	--	Possibly, complexity to understand and generate the hardware for this function falls outside the time of this thesis.	No
Amul	MEM-CPU	A sparse matrix vector multiplication with irregular memory access patterns.	Yes, Investigation of different storage formats, possibly allow performance increase.	Yes
residual	MEM-CPU	A sparse matrix vector multiply-add operation with irregular memory access patterns.	Yes, the same reasons as for the Amul.	Yes

Table 2.10: Candidates for potential hardware implementation

Analysis of the Sparse Matrix Dense Vector product

3

In the previous chapter, the kernels of the OpenFoam CFD toolbox have been summarized by executing the SimpleFoam solver. In this chapter, we are concentrating on one of these kernels, the sparse matrix dense vector multiplication. The associated functions are the Amul and residual functions. These functions are analyzed in order to select an appropriate format scheme suitable for hardware implementation. The remaining part of the organization of this chapter is as follows. Section 3.1 defines the SMVP. Subsequently, in Section 3.2 the sparse storage format in OpenFOAM is analyzed. Next, different formats are investigated in Section 3.3 to study for better formats suited for hardware. A conversion scheme from the OpenFOAM format to this new format is presented in Section 3.3.8. Finally, this chapter ends with a conclusion in Section 3.4.

3.1 Definition of Sparse Matrix Dense Vector product

Mathematically, the dense matrix vector product A$\mathbf{b}$ =$\mathbf{c}$ with sizes A $M \times N$, $\mathbf{b}$ and $\mathbf{c}$ $N \times 1$, can be written as:

$$c_i = \sum_{x=0}^{N-1} a_{i,x} \cdot b_x \tag{3.1}$$

where $a_{i,x}$ is the i-th column on row x of matrix A, b_x is the x-th index of the input vector $\mathbf{b}$ and c_i the i-th index of vector $\mathbf{c}$. The operation can be extended by initializing $\mathbf{c}$.

$$c_i = c_i + \sum_{x=0}^{N-1} a_{i,x} \cdot b_x \tag{3.2}$$

A dense matrix stored in its dense format can be transformed into a sparse format, when all its zeros are eliminated. However, this requires additional arrays that store the non-zero locations separately. In cases of extreme sparse matrices significant memory space can be saved and zero multiply instructions can be avoided.

Storing a matrix in a sparse format becomes more efficient compared to storing the whole matrix, when the overhead in accessing the index arrays to obtain the non-zero values included with these non-zero calculations is faster then calculating the full dense matrix vector multiplication. On top of that, memory can be saved in case the non-zero elements present a small fraction of the original matrix. Different common storage formats used for sparse matrices employ all the same basic techniques, a (multi-dimensional) array to store the non-zero values and additionally one or multiple arrays storing the locations of these non-zeros.

3.2 The OpenFoam Sparse Matrix Storage format

The computation space is divided by means of a mesh in small segments. To solve this computational grid, the physical equations need to be linked to the grid and translated into matrix form. In this section, a description of properties of the mesh, the matrix description and the relation between the two are described. A special relation between the mesh and the matrices used in the solvers exist and is described in Section 3.2.1. Section 3.2.2 describes the properties of the matrices created in OpenFOAM.

3.2.1 From Mesh to Matrix

Mesh Description All meshes in OpenFoam are considered 3D. A single layer of cells must be used to create 2D cells with an empty front and back. A mesh is defined by:

- List of vertices. Each vertex describes a point in 3D place, consisting out of an x, y and z-coordinate.
- List of faces. Each face is constructed out of vertices. The ordering of these faces define the face normal using the right hand rule.
- List of cells. Each cell is determined in terms of faces. Figure 3.1 contains an example cell. **P** denotes the owner cell center and **N** the neighbor cell center. The center of the face shared by these 2 cells is denoted by f and the face normal by $\mathbf{s}_f$. The distance vector $\mathbf{d}_f$ denotes the distance between the cell centers **P** and **N**.
- List of boundary patches. A patch contains a group of faces on the boundary of the mesh.

All internal faces are stored in a list, ordered by the cell index they belong to. Each face has an owner and a neighbour cell. The owner of the cell has the face normal pointing out of the cell for that particular face. The order of the faces in the list is stored in increasing neighbour cell label order, i.e. the face between cell 1 and 3 comes before the face between cell 1 and 5. The boundary faces, which do not have neighbour cells, are stored in the same list after the internal faces.

Matrix description OpenFoam uses a column row storage format for each non-zero entry of the matrix. Each matrix can be divided into 3 parts:

- A *diagonal* list of scalars (SP or DP floating point numbers), representing the diagonal values of the matrix.
- An *upper* list of scalars, representing the non-zero elements of the upper matrix.
- A *lower* list of scalars, representing the non-zero elements of the lower matrix.

And only the non-zero elements are stored. Consider as an example the following matrix:

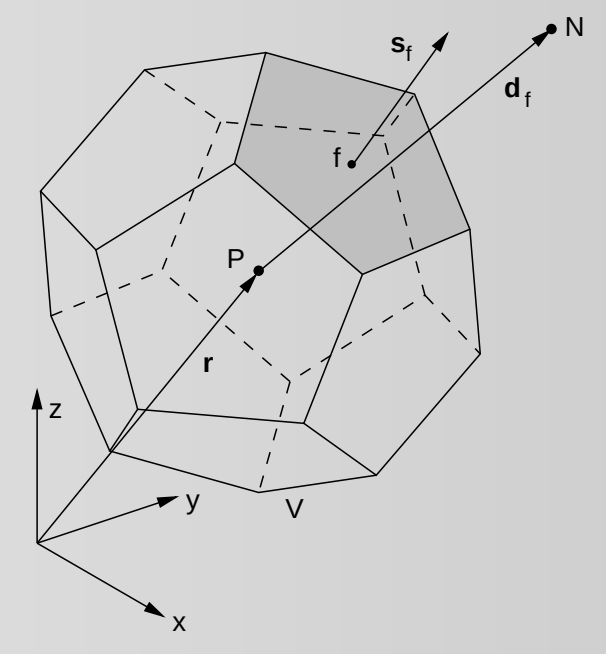

Figure 3.1: Definition of a cell in OpenFOAM [16]

$$\begin{bmatrix} d(0) & 0 & 0 & u(0) & 0 \\ 0 & d(1) & 0 & u(1) & 0 \\ 0 & 0 & d(2) & 0 & u(2) \\ l(0) & l(1) & 0 & d(3) & u(3) \\ 0 & 0 & l(2) & l(3) & d(4) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 6 & 0 \\ 0 & 2 & 0 & 7 & 0 \\ 0 & 0 & 3 & 0 & 8 \\ 10 & 11 & 0 & 4 & 9 \\ 0 & 0 & 12 & 13 & 5 \end{bmatrix} \tag{3.3}$$

In the left matrix, the values are represented by the diagonal, upper and lower entries list indices denoted by d, u and l respectively. The right side contains the values of these lists. The full matrix is not stored in OpenFOAM, but the compressed format is instead of it used. The diagonal entries are in general non-zero and are stored in the *diagonal* list. For the off-diagonal entries, the lists *upper* and *lower* have references to the indices by 2 additional coordinate lists called *upperAddr* and *lowerAddr*. The upper elements have the *lowerAddr* list as row-coordinates and *upperAddr* as column-coordinate, while for the lower elements the *upperAddr* list is stored as row-coordinate and *lowerAddr* as the column-coordinate. The Diagonal, Lower, Upper, vector **b** and **c** arrays are independent, i.e they have no common shared elements.

The matrix in Eqn. 3.3 is stored as follows:

$$\begin{aligned} Diagonal &= [1,2,3,4,5] \\ Upper &= [6,7,8,9] \\ Lower &= [10,11,12,13] \\ UpperAddr &= [3,3,4,4] \\ LowerAddr &= [0,1,2,3] \end{aligned}$$

Relation between Mesh and Matrices The size of the *Diagonal* list equals the number of cells the mesh consists of. The sizes of the *Upper* and *Lower* lists equal the number of internal faces of the mesh. The upper matrix contains values belonging to the owners of faces and the lower matrix contains values belonging to the neighbour cells. Off-diagonal matrix element contain a non-zero value if a face is shared between an owner and neighbour cell on that position (in the mesh). If for example cell 0 and cell 3 sharing a face, non-zero values will be created at the matrix coordinates (0,3) and (3,0). The owner cell must have a lower cell number then the neighbour cell.

All the solvers, except for the Algebraic Multigrid Solvers (AGM) involve matrix operations based on the owner-neighbour relation of cells. The AGM solvers create a hierarchy of coarse matrices and the sparse matrix product is calculated for all considered matrices. The residual kernel, which also computes a SMVM product involves matrices only based on the owner-neighbour relation, i.e. no SMVM is calculated in this kernel for the coarser matrices.

3.2.2 Matrix properties

The matrices created have the following properties:

- The matrices are square.
- A non-zero diagonal stored separately.
- Elemental positions are symmetric.
- Upper and lower entries have mutually transposed coordinates.
- The lowerAddr list is sorted.
- The matrices are very sparse.

The following formula expresses the density of the non-zeros of the matrix in terms of cells and number of faces.

$$\frac{diagonal_{size} + upper_{size} + lower_{size}}{diagonal_{size}^2} * 100\% = \frac{nCells + 2 * nFaces}{nCells^2} * 100\% \qquad (3.4)$$

where $diagonal_{size}$ equals the size of the Diagonal vector, $lower_{size}$ the size of the Lower vector, $upper_{size}$ the size of the Upper vector, $nCells$ the number of cells of the matrix and $nFaces$, the number of internal faces of the matrix. Given the mesh with the

properties summarized in Table 2.2 where $nCells = 6.858.424$ and $nFaces = 16.029.396$, the percentage of non-zero entries for the matrix belonging to this mesh equals $8,27 * 10^{-5}\%$.

Figure 3.2 shows the non-zero entries for a matrix belonging to a smaller mesh, more specific, the PitzDaily case described in the OpenFOAM Programmers Guide. The matrix has 12225 $nCells$ and 24170 $nFaces$.

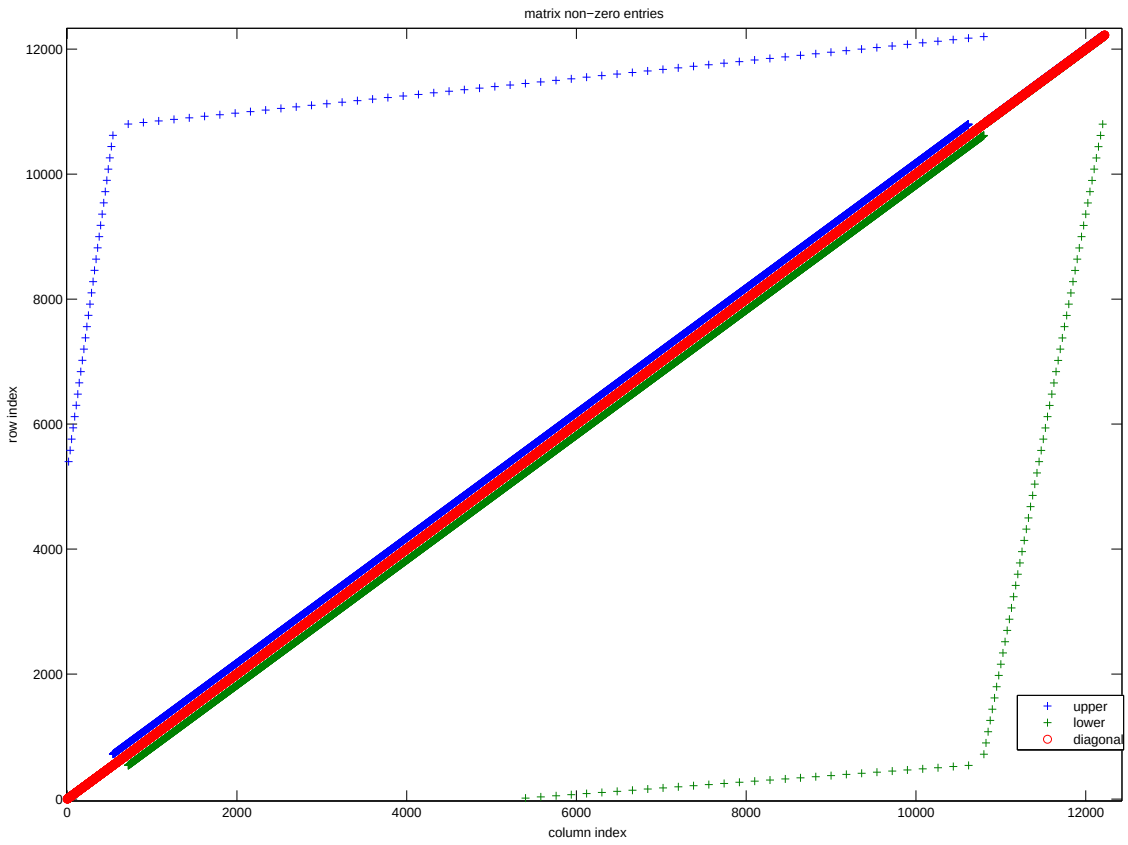

Figure 3.2: Non-zero entries for the pitzDaily case

3.2.3 Analysis of the sparse matrix calculation using the OpenFOAM matrix storage format

For convenience the matrix of Figure 3.3 is repeated here. The instructions that perform the sparse matrix vector products will be analyzed now.

$$\begin{bmatrix} 1 & 0 & 0 & 6 & 0 \\ 0 & 2 & 0 & 7 & 0 \\ 0 & 0 & 3 & 0 & 8 \\ 10 & 11 & 0 & 4 & 9 \\ 0 & 0 & 12 & 13 & 5 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} \tag{3.5}$$

The fragment in the box below represents the computational part of the Amul kernel, with a small modification in the symbol names.

```
for (register label cell=0; cell<nCells; cell++)
{
        c[cell] = Diagonal[cell]*b[cell];
}

for (register label face=0; face<nFaces; face++)
{
        c[upperAddr[face]] += Lower[face]*b[lowerAddr[face]];
        c[lowerAddr[face]] += Upper[face]*b[upperAddr[face]];
}
```

The SMVP consists of 2 loops, in the first loop the diagonal matrix of the matrix times the vector is calculated. The second loop calculates contribution of the SMVP for the non-diagonal elements and are summed to the values of the first loop.

Analyzing the first loop, the following instructions for the matrix of Eqn. 3.3 are obtained:

```
Iteration 0: c[0] = Diagonal[0]*b[0] => A[0,0] * b[0]
Iteration 1: c[1] = Diagonal[1]*b[1] => A[1,1] * b[1]
Iteration 2: c[2] = Diagonal[2]*b[2] => A[2,2] * b[2]
Iteration 3: c[3] = Diagonal[3]*b[3] => A[3,3] * b[3]
Iteration 4: c[4] = Diagonal[4]*b[4] => A[4,4] * b[4]
```

The hardware unit can execute several of these instructions in parallel, since the input vector **b**, the output vector **c** and *Diagonal* lists are not sharing any values. Depending on the area and bandwidth these type of instructions are easily scalable when more area and memory bandwidth are available.

Unrolling the second for-loop of the vector-matrix product of the same matrix A, results in:

```
Iteration 0:    c[3] += Lower[0]*b[0] => A[3,0] * b[0]
                c[0] += Upper[0]*b[3] => A[0,3] * b[3]
Iteration 1:    c[3] += Lower[1]*b[1] => A[3,1] * b[1]
                c[1] += Upper[1]*b[3] => A[1,3] * b[3]
Iteration 2:    c[4] += Lower[2]*b[2] => A[4,2] * b[2]
                c[2] += Upper[2]*b[4] => A[2,4] * b[4]
Iteration 3:    c[4] += Lower[3]*b[3] => A[4,3] * b[3]
                c[3] += Upper[3]*b[4] => A[3,4] * b[4]
```

For each *Upper* and *Lower* value, the coordinates in the original matrix are obtained through the *lowerAddr* and *upperAddr* lists to be able to acquire the correct elements of **b** and **c** and to store the result on the correct location **c**. The instructions cannot be executed all at once in parallel anymore due to read-write dependencies. The dependent instructions are summed in Figure 3.3 for this matrix.

```
Iteration 0: c[3] += Lower[0]*b[0] = A[3,0] * b[0]
Iteration 1: c[3] += Lower[1]*b[1] = A[3,1] * b[1]
Iteration 3: c[3] += Upper[3]*b[4] = A[3,4] * b[4]

Iteration 2: c[4] += Lower[2]*b[2] = A[4,2] * b[2]
Iteration 3: c[4] += Lower[3]*b[3] = A[4,3] * b[3]
```

Figure 3.3: Read-Write dependencies in the OpenFoam Spare Matrix Vector Product at c[3] and c[4]

As a consequence of the read-write dependencies we can conclude that this format is not suited for hardware execution, since it is not scalable with memory bandwidth. The dependent instructions need to be executed in order to ensure correct results. Therefore an investigation for a number of different sparse matrix formats will be analyzed in Section 3.3.

3.3 Analysis of different sparse matrix representations

In this section a conversion, from the currently addressing mode (lowerAddr and upperAddr), to a new matrix representation will be investigated. The OpenFOAM format is not sufficient enough to be accelerated on hardware, since the storage format can not be exploited for parallelism, as can be seen in Figure 3.3. A couple of different matrix formats are analyzed in this section. The criteria to compare the formats are scalability, the number of memory accesses and the required memory to store the matrix. Here scalability means the convenience of exploiting parallelism in such a way that when multiple hardware processing units are available, dividing the matrix A is trivial if more memory bandwidth is available, for example by dividing the matrix in rows. A distinction between storage needed for the indexing and values is made, since some schemes include storage of zero values. The most general formats can be found at [15] or [22]. In some specific cases, a reference is made to the new published formats.

3.3.1 Current Format: (A modification of COO)

The Coordinate (COO) storage format is perhaps the simplest format and most flexible format which can be used to store sparse matrices. Assuming we have an N x N matrix A = [a_{ij}] containing the non-zero entries, the COO format is created as follows: For each non-zero entry, the row and column coordinates are stored besides its value. All the arrays equal in length equal to the number of non-zeros of the matrix.

- **values** The array with values is created by taking all the non-zeros elements of the original matrix, there is no specific order required.
- **col** Each value **col**[k] holds the column coordinate of the non-zero element a_{ij} stored in **values**[k].

- **row** Each value **row**[k] holds the row coordinate of the non-zero element a_{ij} stored in **values**[k].

By taking advantage of the symmetric elemental positions, the diagonal entries can be stored in a separate array, the row and column indexes of the lower half or the upper half of the original matrix have to be stored. This modified storage format is used in the OpenFOAM toolbox. Figure 3.4 contains an example of a matrix in the COO format together with its OpenFoam format.

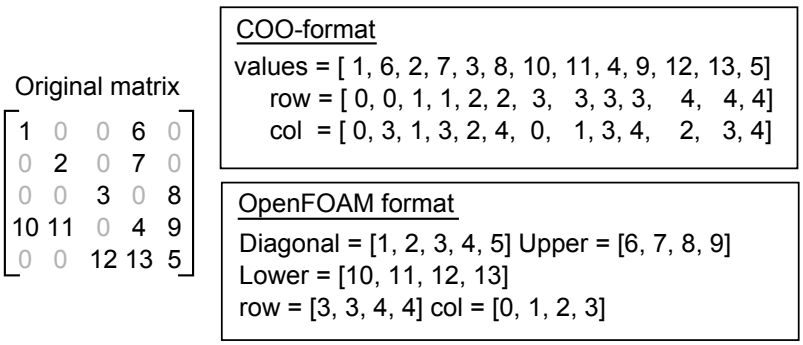

Figure 3.4: The COO and OpenFoam sparse storage format

Table 3.1 contains a summary of the criteria regarding the required minimal memory space and the number of memory instructions for the OpenFOAM format directly retrieved from the code on page 34.

Parallelism	No, memory accesses can be random
Storage Index	**col**, **row**: $2nFaces$
Storage Value	**values** : $nCells + 2nFaces$
Loads	$2nCells + 6nFaces$
Stores	$nCells + 2nFaces$
Memory accesses	$3nCells + 8nFaces$

Table 3.1: Properties of executing an SMVM with the OpenFOAM storage format

3.3.2 Compressed Row Storage

The CRS, also referred as Compressed Sparse Row (CSR) in the literature, is one of the most or the most used sparse matrix storage format. Its a general format which can be used to store any matrix. First, a 1-dimensional vector values is constructed that contains all the values of the non-zero elements a_{ij} of matrix A, taken in a row-wise fashion from the matrix. No specific ordering is required for the non-zero elements within a row, however we make the assumption here that in each row, column coordinates are sorted from low to high column index values.

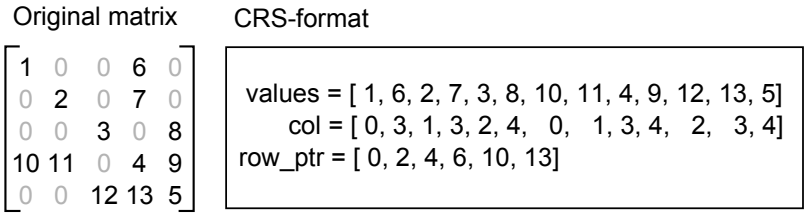

Figure 3.5: The CRS sparse storage format

- **values** = $[v_k]$ where 0 < k < N_z, $v_k = a_{ij}$ where a_{ij} is the k-th element of A for which holds a_{ij} != 0 and $0 \leq i, j < N$
- Secondly, a 1-dimensional vector **col** = $[c_k]$ of length equal to the length of values is constructed that contains the original column positions of the corresponding elements in values. Thus, c_k = j where 0 < k < N_z, j is the column position of the k-th element of matrix A for which holds a_{ij} != 0 and 0 < i, j < N
- Last, an array **row_ptr** = $[r_m]$ of length N+1 is constructed. The length of row m of the matrix A can be computed by r_{m+1}-r_m and the first row starts with index 0, $r_0 = 0$. Each element in vector **row_ptr** is therefore a pointer to the 1st non-zero element of each row in the vectors **values** and **col**.

Figure 3.5 shows the CRS format for the example matrix. The code to perform the SMVM for the CRS format is written out in the box below.

```
//  Compressed Row Storage format
int p = 0;
int q = 0;

for (i=0;i<N;i++)
{
    p = q;
    q = row_ptr[i+1];
    int c0 = 0;

    for (j=p; j<q; j++)
        c0 += values[j]*b[col[j]];

    c[i] = c0;
}
```

Analyzing the above code, Table 3.2 contains the memory cost for this storage format.

Memory accesses and memory space can be reduced if the diagonal is stored separately. This format is called the Modified Sparse Row (MSR) format. The code contains

Parallelism	Yes, row-wise
Storage Index	**row_ptr**, **col** : $2nCells + 2nFaces$
Storage Value	**values** : $nCells + 2nFaces$
Loads	$4nCells + 6nFaces$
Stores	$nCells$
Memory accesses	$5nCells + 6nFaces$

Table 3.2: Properties of executing an SMVM with the CRS format

a small modification which is the initializing of the temporary variable c0 used to accumulate the row sum. Figure 3.6 shows the MSR format for the example matrix. The memory cost analysis are similar to the CRS format and the results for this format can be found in Table 3.8.

```
// Modified Sparse Row format
int p = 0;
int q = 0;

for (i=0;i<N;i++)
{
    p = q;
    q = row_ptr[i+1];

    int c0 = diag[i]*b[i];

    for (j=p; j<q; j++)
        c0 += values[j]*b[col[j]];

    c[i] = c0;
}
```

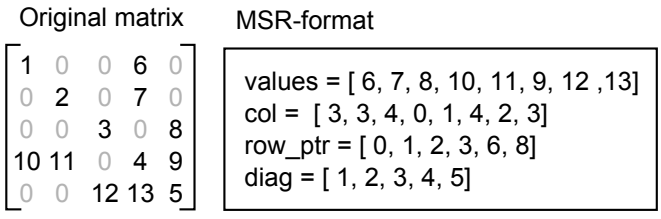

Figure 3.6: The MSR sparse storage format

Due to the symmetry of the matrix, it is only required to store the indexes of the strictly upper matrix or lower matrix. Storing the diagonal separately, and only the upper or lower matrix in CRS format is called the Symmetric Sparse Skyline format

(SSS). The SSS for the example matrix is shown in Figure 3.7. The memory cost for this format can be obtained in Table 3.8.

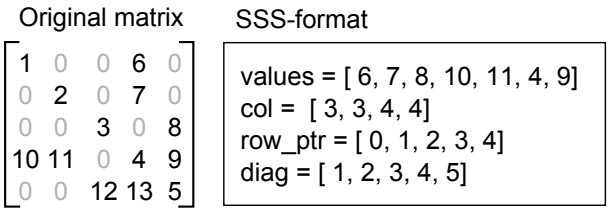

Figure 3.7: The SSS sparse storage format

```
// Symmetric Skyline format
for (i=0;i<nRows;i++)
{
c[i] = diag[i]*b[i];
}

for (i=0;i<nRows;i++)
{
    a = b;
    b = row_ptr[i+];

    for (j=a;j<b;j++)
    {
        c[i]  += Upper[j]*b[col[j]];
        c[col[j]] += Lower[j]*b[i];
    }
}
```

The advantage of the SSS scheme is its efficiency in required memory space, however at the same time it destroys the parallelism in a similar way as the OpenFOAM format.

3.3.3 Jagged Diagonal Storage

The JDS format is a very useful format when calculating matrix vector products on parallel and vector processors. The format can be constructed in the following way:

- First, all the nonzero elements are shifted to the left side of the matrix maintaining the order within a row, the zeros stay on the right side. While doing this, each original column coordinate is stored as well in a 1-dimensional array **col**. Figure 3.8 depicts this.

- Second, the rows are permuted in such a way that they lead to decreasing row sizes. The permutations are stored in **perm**.

- Finally, the resulting vertical vectors created in the second step are stored in a linear array **value** and and the indexes of the starting point of each vertical column are stored in **Ind**. The vertical vectors at the left side are larger or equal to the once the right side, the size of the **Ind** is denoted as $N_{z,col}$.

```
// Jagged Diagonal format
int p = ind[0];
int q = ind[0];

for (i=0; i < nz_col ; i++)
{
    q = p;
    p = ind[i+1];

    for ( j=0; j < p-q; j++)
    {
        c[j] = c[j] + value[q+i] * x[col[q+i]];
    }
}
```

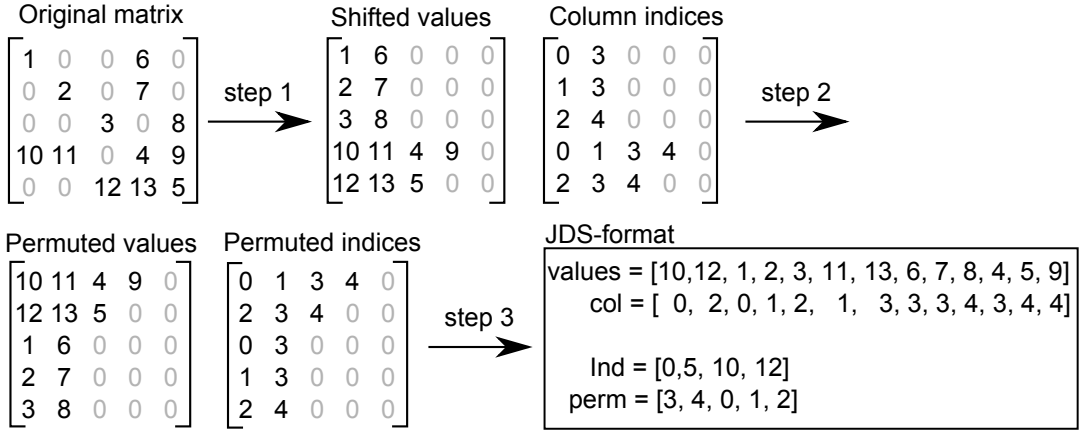

Figure 3.8: The JDS sparse storage format

Table 3.3 summarizes the cost executing the SMVP with the JDS format.

3.3.4 Diagonal Format, Diagonal Storage Format, Compressed Diagonal Storage (DIA ,DSF, CDS)

When a sparse matrix contains a lot of non-zero diagonals, the required storage space for the index arrays can be compressed very efficiently. The DIA format stores the non-zero diagonals only and can be constructed from the original matrix by creating a

Parallelism	Yes, vertical columns allow parallel execution
Storage Index	**perm**, **col**, **jd_ptr**: $2nCells + 2nFaces$
Storage Value	**values** : $nCells + 2nFaces$
Loads	$4nCells + 8nFace$
Stores	$nCells + 2nFaces$
Memory accesses	$5nCells + 10nFaces$

Table 3.3: Summary of the cost executing the SMVM with the JDS format

matrix **values** and an array **distance** and a parameter *ndiag*. The matrix values has size $ndiag \times N$ and stores all the values of the non-zero diagonal in the original matrix. The offset, measured from the main diagonal, for each non-zero diagonal is stored in the **distance** vector. Since the matrices are positional symmetric only the offsets for the upper (or lower) triangular part of the matrix are necessary to be stored.

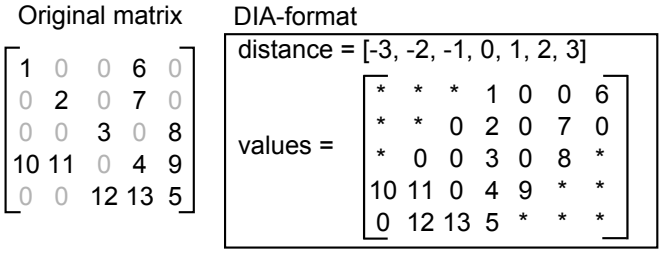

Figure 3.9: The DIA sparse storage format

Table 3.4 shows the contribution of the 15 most important diagonals. The table has to be read as follows: The first column specifies the offset, or distance from the main diagonal. Since the matrix is symmetric the upper half have positive and lower half negative offsets. The next 2 columns show for each diagonal, with the distance to the main diagonal given in the first column, the number of zeros and non-zeros on that diagonal. The third column includes the extra zeros to make the matrix values a rectangular matrix. The 4th column shows the cumulative sum of the non-zeros so far and the last column shows the cumulative sum in percentages of the total amount of non-zeros.

This format will not be analyzed further due to the following reasons:

- To many diagonals are needed to store this matrix, 36% of the non-zero of the whole matrix are stored for example in 15 diagonals.
- Each diagonal itself contains a lot of zero's which are included in the DIA format.
- A total of 2.452.421 diagonals (not shown in the table) are required to stored the whole matrix in DIA format, included with the main diagonal (with offset 0). The memory requirement to store the matrix **values** with size N by 2.452.421 is not an option to consider.

Distance	No. non-zeros	No of zeros	Cum Sum nz	Cum freq %
0	6.858.424	0	6.858.424	17.62
(-) 2	1.356.056	5.502.368	9.570.536	24.59
(-) 4	697.619	6.160.805	10.965.774	28.18
(-) 1	538.902	6.319.522	12.043.578	30.95
(-) 6	316.874	6.541.550	12.677.326	32.58
(-) 3	284.139	6.574.285	13.245.604	34.04
(-) 8	168.433	6.689.991	13.582.470	34.90
(-) 5	147.049	6.711.375	13.876.568	35.66

Table 3.4: Contribution of the main diagonals for the DIA format for the profiled mesh

The Ellpack-Itpack (ELL) is a generalization of the diagonal storage format and is intended for matrices with a limited maximum number of non-zeros per diagonal. Since this is not the case, this format will be discarded as well.

3.3.5 Block Sparse Row

In the BSR format the matrix is considered to consist of small dense square blocks. Zeros inside the dense block are treated as non-zero values with the value zero. The BSR format can be seen as a simple generalization of the Row Compressed Storage. The dimension of each small dense block is denoted by B_{dim}. The total of nonzero blocks equals $N_{z,block} = \frac{N_z}{N_{block}^2}$ and the matrix dimension becomes $N_{BSR} = \frac{N}{B_{dim}}$. The BSR format is constructed as follows:

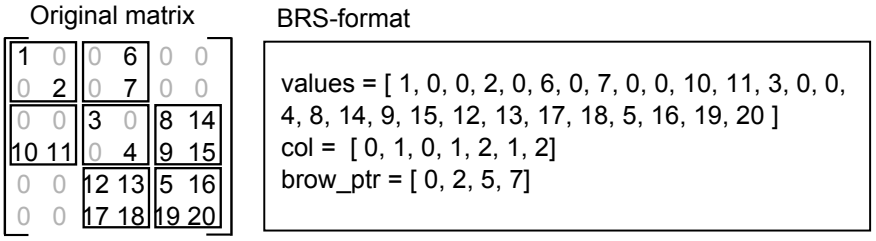

Figure 3.10: The BSR sparse storage format

- Each block of size $B_{dim} \times B_{dim}$ that contains a non-zero value is stored in a row wise fashion. Within a block, each element is again stored in a row wise fashion. All the values are stored in an array values of length $N_{z,block} \times B_{dim}^2$.

- Second, a 1-dimensional vector **col** of length equal to the length of $N_{z,block}$ is constructed that contains the column positions of the corresponding non-zero blocks in values. The column positions of the upper left coordinate of each block are stored.

- Last, a vector **brow_ptr** = $[r_m]$ of length N_{BSR}+1 is constructed. r_m+1-r_m denotes the length of row m of matrix and $r_0 = 0$. Each element in the array **brow_ptr** is therefore a pointer to the 1st non-zero block of each row in vectors values and col.

Figure 3.10 shows this format for $B_{dim} = 2$.

```
// BSR-format with block sizes of 2
q = brow_prt[0] // =0;

for (i=0;i<N_bsr;i++)
{
    p = q;
    q = brow_ptr[i+1];
    for (j=p;j<q;j++)
    {
        j0 = col[j];
        b0=b[j0<<1];
        b1=b[j0<<1 +1];

        j1 = j<<2;
        c0 += values[j1 + 0]*b0 + values[j1 + 2]*b1;
        c1 += values[j1 + 1]*b0 + values[j1 + 3]*b1;
    }

    y[2*i] = c0;
    y[2*i+1] = c1;
}
```

Ideally, if all blocks are filled with $B_{dim}^2 = 4$ non-zero elements, this format will lead to better results then for CRS. The two reasons for this are:

- Less index positions have to be loaded from the array **col**, at the slight cost of more run-time index calculations, which can be done in parallel.
- Less elements of vector **b** have to be loaded, to be precise, only half of the indexes, compared to CRS storage format.

The drawback is that possible extra zeros can be included and therefore multiplications with zeros. To analyze this cost, the BSR format for the profiled matrix is created with $N_{z,block}$. Table 3.5 shows the results for $B_{dim} = 2$.

The extra zero-values included to form small dense blocks is very bad, since 68% zero multiply-add instructions are included. The advantage or losses gained by the auxiliary position matrix can be calculated and compared to storage formats that do not include storage of zeros.

As a consequence of storing zero values, this scheme needs more loads then in the CRS/MSR format as can be seen from Table 3.6. The expectation is that using a

$N_{BSR} = \frac{N}{2}$	3 429 212
$N_{z,block}$	30 635 142
Average ratio zeros within a block	0.68
Average zeros within a block	2.72

Table 3.5: Matrix properties for $B_{dim} = 2$ for the BSR format

Parallelism	Yes, B_{dim} rows simultaneously
Storage Index	**brow_ptr**: $N_{z,block}$+1 = 30 635 143 **col**: => $N_{z,block}$ = 30 635 142
Storage Value	**values**: $B_{dim}^2 \times N_{z,block}$ = 122 540 568
Loads	$1 + N_{BSR} + 7 \times N_{BSR} \times N_{z,block}$ = 217 875 207
Stores	$nCells$ = 6 858 424
Memory accesses	224 733 631

Table 3.6: Summary of the cost executing the SMVM with the BSR format

larger B_{dim} more zeros will be included in the computations and the performance will only decrease. In a similar way like in the Skyline Sparse Skyline storage format, it is possible to store the blocks on the main diagonal separately, in addition with the block coordinates for the strict upper part of the original matrix. In this way memory storage space can be reduced.

3.3.6 Block Based Compression Storage (BBCS)

The last sparse format scheme discussed here is the Block Based Compression Storage format. The BBCS is a relative new format and proposed in [23]. The BBCS format can be obtained as follows:

- The $N \times N$ matrix is vertically cut in $\lceil \frac{n}{s} \rceil$ vertical blocks, denoted by A^m, where $0 < m < \lceil \frac{n}{s} \rceil - 1$
- Each non-zero element a_{ij}, $s \times m < j < s \times (m + 1)$ and $0 \leq i < N$, is stored in a row wise fashion within its vertical block.

The advantage of this scheme, in computing A**b**, is that the values of A and vector **b** can both be streamed. Each A^m block can be multiplied with a fraction of vector **b**, namely $\mathbf{b}[s \times m : s \times (m + 1)]$. These values can be kept close to the executional units. Each value of A is needed only once and therefore they can be streamed as well.

The Am blocks are stored as a sequence of 6-tuple bits in the following way [23]:

1. **Value:** specifies the value of a non-zero a_{ij} matrix element if the Zero-Row Flag equals 0. Otherwise it denotes the number of subsequent block rows within a vertical block with zero matrix elements.

2. **Column position:** Specifies the matrix element column number within the block. Thus for a matrix element a_{ij} within a vertical block A^m it is computed as $j \ mod \ m$.

3. **End-Of-Row Flag (EOR):** is 1 when the current data entry describes the last non-zero element of the current row block and 0 otherwise.

4. **Zero-Row Flag (ZR):** is 1 when the current block row contains no nonzero value and 0 otherwise. When this flag is set the $Value$ field denotes the number of subsequent block rows that have no non-zero values.

5. **End-Of-Block Flag (EOB):** when 1 it indicates that the current matrix element is the last non-zero one within the VB.

6. **End-Of-Matrix Flag (EOM):** is only one at the last entry of the last vertical block of the matrix.

The width of a 6-tuple entry equals $log_2(s) + 5$ where $log_2(s)$ equals the size of each column coordinate and where the 5 remaining flags require all 1 bit. To map the tuples efficiently on memory, we investigate this format for $s = 8$ and $s = 2048$ so that the tuples can be stored in a *char* and *short integer* respectively.

The indexes needed to store the $Values$ array and 6-tuples within a vertical block equals on average $c + avg\_col \times s = c + avg\_row * s$ where avg_col the average number of non-zeros per column for the entire matrix, which equals the average number of non-zeros per row. The value c represent the average additional entries for each block where the Zero-Flag is non-zero. The storage required for the full matrix can be calculated now as: $\frac{n}{s} \times \{c + avg\_row \times s\}$.

The loads and store that are required are:

- To operate in parallel, the start index each vertical block must be known in advance, this requires an addition amount of loads, $\lceil \frac{n}{s} \rceil$.

- Each element of A^m, has to be multiplied with an element of $\mathbf{b}$. For the matrix N_z (equal to nCells+2nFaces) values are required to be accessed, while for $\mathbf{b}$ nCells loads are necessary.

- Each result of a row×vector multiplication within a vertical block has to update results in loading and storing in y. The worst case scenario would be $2 \times length(\mathbf{y}) \times nblocks = 2 \times n \times \frac{n}{s}$, which occurs when all blocks at least have a non-zero on each row. However this is certainly not the case. We try to analyze it by summing up all the non-zero rows for each block and multiplying it with 2, shown in Table 3.7.

- Last, for each block c 6-tuples and c offsets entries of load instruction are required to determine the next non-zero row within a block. These are the values where ZR For the total matrix $\frac{n}{s} \times c$ entries are required.

Table 3.7 summarizes all the properties for the matrix in its BBCS form.

Memory Accesses		
Size of vertical block A^m	$s = 8$	$s = 2048$
Loads for block-pointers	857.303	3.349
Loads for all a_{ij} and x[j]	77.834.432	77.834.432
Load-Store for temporay y	$2 \times 33.072.553 = 66.145.106$	$2 \times 26.916.017 = 53.832.034$
Value of c	19.16	3349
Load c ($ZR = 1$)	$2 \times c \times \lceil \frac{n}{s} \rceil = 32.851.851$	$2 \times c \times \lceil \frac{n}{s} \rceil = 22.430.530$
Total Memory accesses	177.688.692	154.100.345
Storage		
Storage type of 6-tuple	char (8 bits)	short int (16 bits)
Amount of 6-tuples	55.377.434	50.165.032
Amount of $Values$	55.377.434	50.165.032
Block pointers	$\lceil \frac{n}{s} \rceil + 1 = 857.304$	$\lceil \frac{n}{s} \rceil = 3350$
Other		
percentage of $ZR = 1$	29.7%	28.8%

Table 3.7: Matrix properties for the BBCS format for $s = 8$ and $s = 2048$

3.3.7 Comparing the formats - Choosing the best Hardware Candidate format

Table 3.8 summarizes the results of the different formats. From the table the conclusion can be made that the SSS storage is the most efficient storage format, however difficult to implement in hardware to avoid dependencies and hardly scalable with bandwidth and area increase.

The MRS format requires the least amount of memory operations and storage considering a format which is scalable. Therefore this format will be chosen to be implemented in hardware.

3.3.8 Sparse matrix conversion: From OpenFoam format to MRS/CRS

The best solution is to rewrite the toolbox for MRS support. In this case no convertions between the matrices is required. But for fast proto-typing and verification we convert the matrix during run-time. An algorithm to convert the current OpenFOAM format to the MRS format in $O(n)$ time, where n the matrix dimension will be presented in this section. To obtain the MRS/CRS format the following steps have to be executed.

- **Step 1** The **row_ptr** array is initialized with 0 for MRS and 1 for CRS.
- **Step 2** each index of **row_ptr** will be filled with its size.
- **Step 3** The **row_ptr** is constructed by summing up previous row sizes to the current index.
- **Step 4** Filling the **col** array with the correct coordinates.
- **Step 5** Restoring the **row_ptr** array.

Format	**Parallelism**	**Memory Accesses**	**Storage - Values** **- Indexes**
Current	No	3 nCells + 7 nFaces	- Minimal (nCells + 2 nFaces) - 2 nFaces
CRS	Yes, row-wise	5 nCells + 6 nFaces	- Minimal - 2 nCells + 2 nFaces
MRS	**Yes, row-wise**	**4 nCells + 6 nFaces**	**- Minimal** **- nCells + 2 nFaces**
SSS	No	4 nCells + 9 nFaces	- Minimal - nCells + nFaces
JDS	Yes, vertical columns	5 nCells + 10 nFaces	- Minimal - 5 nCells + 10 nFaces
DIA	Yes, diagonal-wise	Too many zeros included	- Too many zeros included
BSR	Yes, multiple row-wise	+79.2% compared to CRS	- 170% compared to CRS - 133.8% compared to CRS
BBCS	Yes, vertical blocks	For s = 8 +17% compared to MRS For s = 2048 +6.5 % compared to MRS	For s = 8 - 142% compared to MRS - 37.8% of MRS For s = 2048 - 129% compared to MRS - 64.5% of MRS

Table 3.8: Analysis of several Sparse Matrix Storage Formats

Figure 3.11 shows these intermediate steps. On the left side code is written to obtain the values and on the right side the values in the arrays of these steps are applied.

The CRS format is also included. In the next chapter its explained why we implemented this format instead of the MRS format.

3.4 Conclusion

In this chapter, the OpenFOAM SMVP has been extensively examined. The original format was not appropriate for parallel extension and therefore different formats were analyzed. The criteria to compare them were scalability with increasing bandwidth, the number of memory accesses and the required storage space. The MSR was found to be the best. A conversion algorithm in O(n) time between the OpenFOAM format and the MSR format is also given. In the next chapter a description of the start of the hardware unit for the MSR format is given.

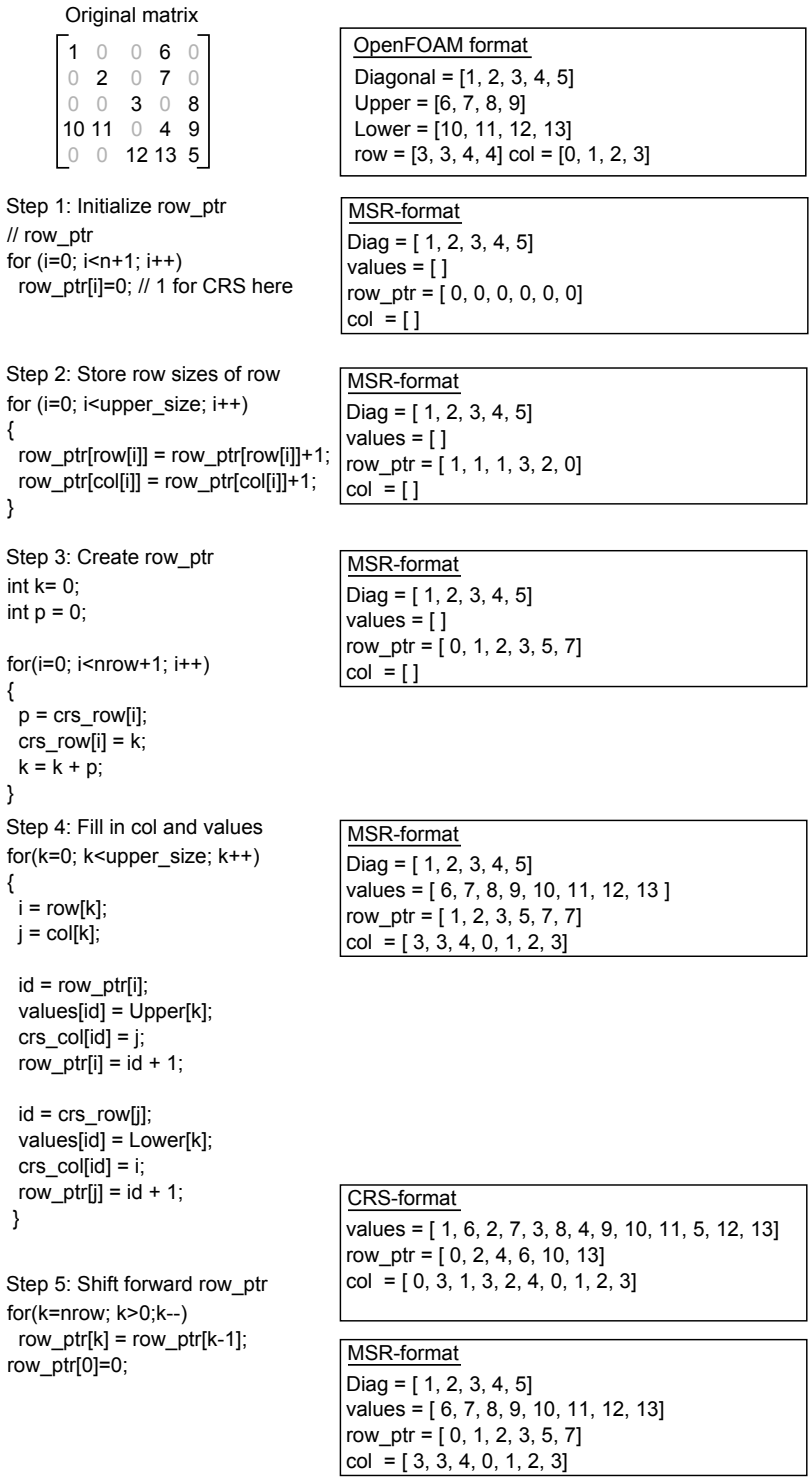

Figure 3.11: Conversion between OpenFoam format and MRS/CRS

A hardware Accelerator for the SMVM

4

In the previous chapter analysis showed that the Modified Sparse Row format is probably the most suited format to accelerate FVM matrices in hardware. This chapter continues with the development of a hardware accelerator for the SMVP suited for matrices in the FVM area. First, an overview of the RASC reconfigurable hardware platform is given. Next, the hardware unit is described. Finally, this chapter ends with a conclusion regarding the designed hardware unit.

4.1 The Hardware Platform

The hardware platform that is targeted is the Altix-450 machine. The same system environment that was used to profile the SimpleFoam solver. The Altix supports multiple reconfigurable hardware units which can be integrated into the server with low latency cost and high bandwidth. Section 4.1.1 introduces the RASC-core. Next, Section 4.1.2 describes the design options that the RASC-core provide to integrate the Customized Computing Unit into the system environment.

4.1.1 Introduction to the RASC-core

The Altix machines have the ability to be extended with RASC-blades. RASC stands for Reconfigurable Application-Specific Computing. Each RASC-blade uses FPGA technology as co-processor of the CPU. The RASC hardware module is based on an Application-Specific Integrated Circuit (ASIC) called TIO. TIO attaches to the Altix system NUMA-link interconnect directly. Figure 4.1 depicts the organization of the RASC Hardware Blade.

To be able to reach high performances, the FPGAs are connected with a NUMAlink fabric making them a peer to the CPU with high bandwidth and low latency. Using the Direct Memory Access (DMA) streams, a bandwidth from main memory to the FPGAs up to 6.4 GB/s per FPGA can be reached. The 6.4 GB/s bandwidth consist out of 3.2 GB/s in both directions, from main memory to FPGA and vice versa. These features enable both extreme performance and scalability.

The RASC algorithm FPGA is a Xilinx Virtex 4 LX200 part (XC4VLX200-FF1513-10). It is connected to an SGI Altix system via the Scalable System Port (SSP) on the TIO ASIC and configured with a bitstream through the FPGA loader.

4.1.2 Implementation Options

The RASC Core Services allow users to configure a number of options. These options are:

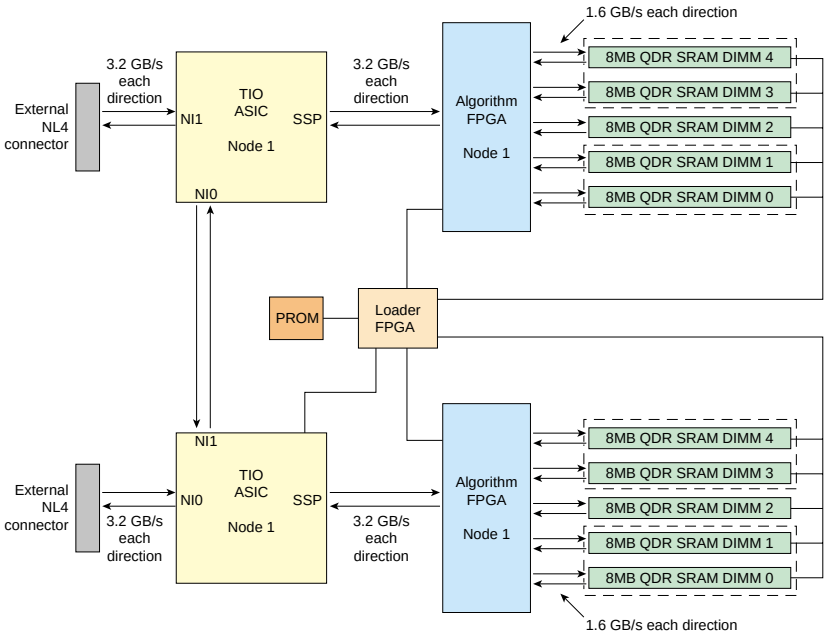

Figure 4.1: RASC Hardware Blade [13]

- Algorithm clock rate
- Optional supplemental clock
- Stream DMA
- Memory configurations
- Algorithmic Defined Registers
- Algorithmic Debug Registers
- Synthesis tool option

The algorithm clock rate can be set to 50 MHz, 100 MHz or 200 MHz. The last mentioned frequency is the clock rate at which the Altix Core Services outside the FPGA run. An optional supplemental clock can be used to allow the CCU running on a different frequency then the algorithmic clock. This clock is generated by a Digital Clock Manager (DCM). The designer is responsible to transfer data among 2 different clock domains. A maximum of 4 DMA streams in each direction can be picked, from FPGA to main memory and vice versa. The streams directly transfer data from memory to the FPGA. The bandwidth of 3.2 GB/s in each direction is shared among the streams. A selection between three setups for the memory configuration of the external FPGA memory can be chosen:

- SRAMS disabled, no external memory can be used.

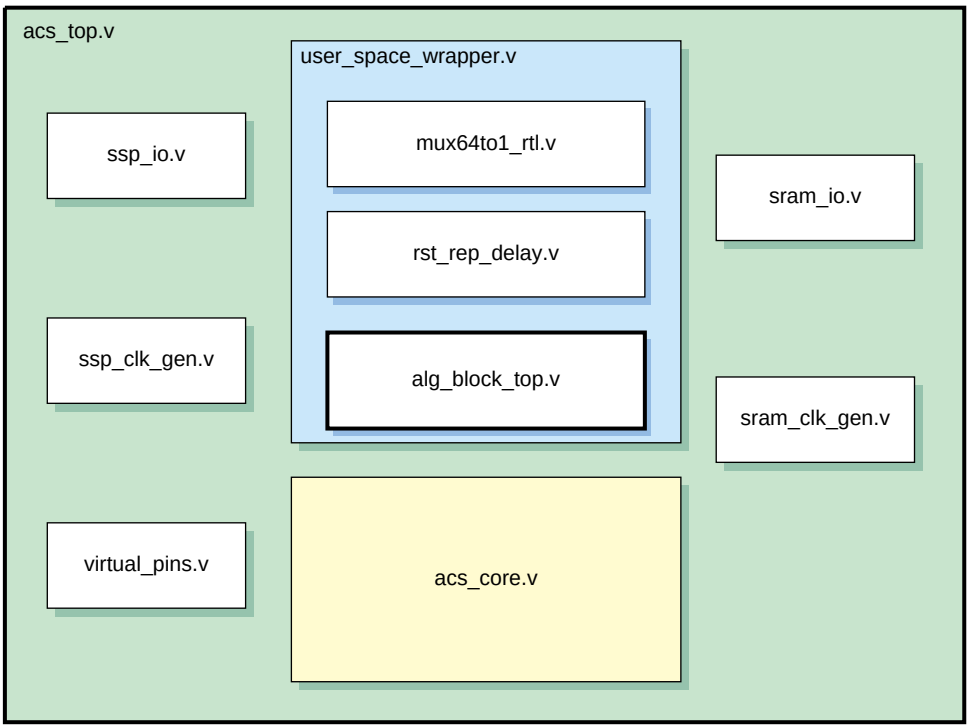

Figure 4.2: RASC file overview [13]

- A configuration with 3 logical SRAMS, in which the first two banks contain 128 bits wide data and the latter 64 bits data.
- A configuration with 5 physical SRAMS. In this configuration all banks contain 64 bits equal bandwidth.

There are up to 64 Algorithmic Defined Registers (ADR) available and there are registers available for debugging purposes. The ADRs are used to store function arguments inside the FPGA. The Algorithmic Debug Registers are used to debug the application in real time with gdbfpga. The gdbfpga tool supports monitoring up to 64 signals which can be traced in real hardware. In the last option, selection the synthesis tool, Xilinx or Synplify must be specified. In Figure 4.2 the RASC file overview is depicted. The user needs to connect and add his hardware algorithm in the file alg_block_top.v. The other files are already designed by the SGI support team and should not be modified.

4.2 Hardware Design

The hardware-unit is build out of two key components called the Host and Processing Element. The Host is responsible for loading data from memory and providing this data in a correct way to the PE(s). The task of the processing element is to perform the

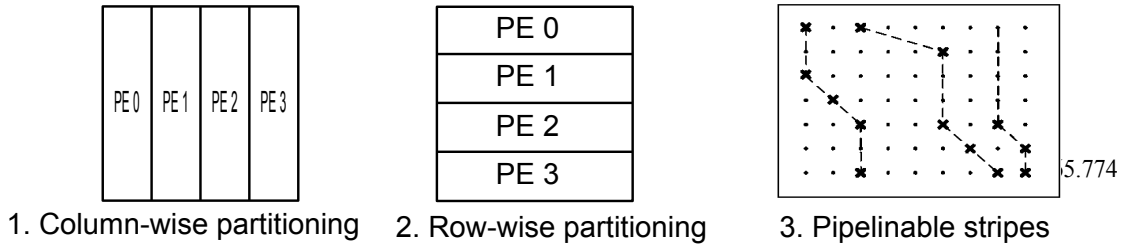

Figure 4.3: Matrix divide strategies

calculation on the data provided by the host. This section is continued by explaining the designs for the Host and the PE components.

4.2.1 Strategy of dividing work among processing Elements

Before presenting the hardware unit, this section explains the approach how the work is divided among the Processing Elements. The ability exist to group the data of matrix A into the following ways:

1. Vertical slicing of the matrix. A vertical block will be assigned to each processing element that becomes free.

2. Horizontal slicing of the matrix. Assigning the work of one or multiple rows to a Processing Element.

3. Creating stripes. By grouping multiple non-zeros into stripes.

Figure 4.3 contains the dividing methods. The first method is implemented in [29]. The last method is implemented in [7]. Creating stripes is not applicable for the matrices generated from meshes constructed out of prisms, pyramids and similar structures. Section 5.4 explains this in more detail. Due to the extremely sparse matrices, we believe that option one or two is the best method to proceed on. In the next chapter, the different methods and results are compared more in detail.

When more bandwidth is available more processing elements can be placed. The data from matrix A must be divided and streamed into the different processing elements. To accomplish this, matrix A can be divided vertically (in groups of columns) or horizontally (in groups of rows). To complete the analysis, the striping method is also included.

- Pre-processing
  - Horizontal sectioning does not require any changes to the CRS/MRS format.
  - Vertical sectioning requires extra preprocessing to contain the format for the vertical slices. For each vertical slice the new CRS/MSR format must be calculated.

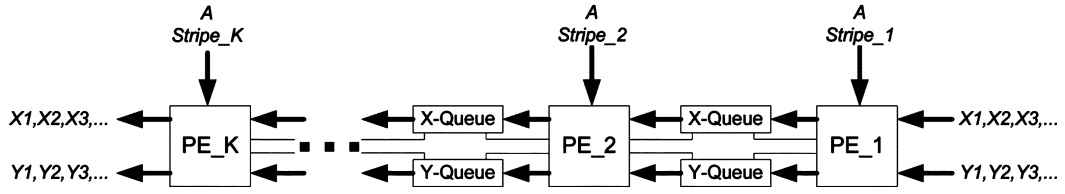

Figure 4.4: SMVM pipeline of the design in [7]

  - The striping methods requires more pre-processing then the previous two methods. The matrix must be divided in stripes which can be pipelined and feed through the execution unit. Figure 4.4 depicts the method described in [7]. In a linear array pipeline architecture, matrix elements are fed to the PEs from the top, while the vectors are horizontally fed in to the pipeline.

- Size of **b**
  - The horizontal sectioning requires that the whole vector **b** should be available to all processing elements.
  - Vertical sectioning requires only subparts of **b** available to the processing elements.
  - In the striping method only the active elements are fed into the pipelines.

- Final result
  - With row-sectioning each row that is processed contains immediately its final result.
  - With Vertical sectioning, rows results are obtained by summing up all the resulting temporary values of the vertical slices. Since the matrix sizes can be huge (over 1 million), the FPGA does not contain enough memory to store the temporary results for the slices. Accessing high-latency off-chip memory and doing additions with high latency adder is a complex task.
  - For the pipeline array, the temporary row results are streamed through the PEs and the full cost for the SMVM is obtained.

A similar implementation with vertical slices, can be found in [29]. In general it could be said that the sparser the matrix the lower the peak performance. The sparsest matrix tested by the authors contained a non-zero ratio of 0,04% with a peak performance of 20% only. A performance of 350 MFLOPS using a bandwidth of 8 GB/s. In addition, the authors did not include the final addition of the temporary vectors. They assume that these additions are summed up on the GPP CPU. To include full cost, we implement the row-wise partitioning strategy.

4.2.2 Design strategy

The following design strategy is taken to design the SMVM:

- Design of smaller components of the CCU (The SMVM kernel).
- Verification of each subcomponent by testing it separately.
- Integration of the subcomponents into larger units.
- Verification of the CCU unit in simulation.
- Automated test generation to test Modelsim simulations and real hardware execution.
- Connecting the CCU to the RASC-services environment.
- Including specific signals from the design to the RASC debug interface for debug and verification purposes.
- Verification of the design with 1 and 2 PEs in real hardware.
- Automated test generation for multiple PEs for simulation verification and performance estimations.

4.2.3 Processing Element(s)

The computations that are involved in a sparse matrix vector product are similar as in the dense full matrix vector product, with the exception that large parts are zero and therefore the zero computations are excluded. The non-zero computations from any sparse matrix format are similar and the Processing Element (PE) is designed in a way capable of accepting any pairs of values of A and **b**.

The core of the processing element is the multiply-add unit. To compute the rows, many designs [29], [24] implement adder trees to accumulate row results. The task of this circuit is to sum up all partial products and to reduce it into one final value. The adder reduction trees consume a huge amount of slices on the FPGA. In our design, the adder trees are avoided by accumulating the sum of each row in time. Since there is a latency of 8 cycles for the addition, different rows utilize the same resource unit simultaneously. By creating 8 slots, each cycle 1 slot have access to the multiply add unit. Thus, the multiply-add unit is used to calculate different rows simultaneously. We assume significant resources can be saved if this approach is used. To handle the complexity of the design, temporary results will be stored in BRAM (FPGA on chip memory) for the different rows that are active at the same time, to avoid multiplexers required for switching between the inputs to the multiply add unit. The multiply add core is being reused from the design in [25]. The total latency for the multiply add unit is 11 cycles.

Considering the Processing Elements as a black box, the inputs and outputs of this component are shown in figure 4.5. There are 2 global signals, the clock (clk) and the reset (rst) signal. Further, 3 group of signals can be seen on the left side, where the first

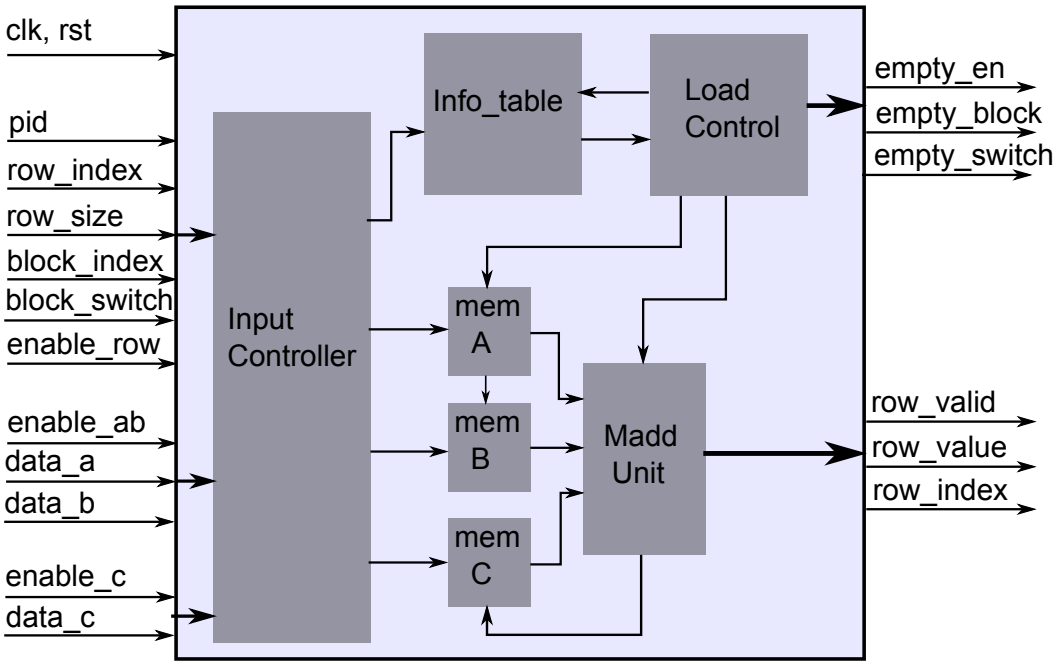

Figure 4.5: Overview of the Organization inside a PE

group is related to the row information, the second group to data from the matrix A and vector **b** and the last group accepts data from vector **c**, if enabled. Data from the matrix A and vector **b** must enter in pairs and be valid at the same time.

Row information includes the row size, its index, the pid (processing ID) to which the row belongs too, and for which slot (denoted by block_index) its targeted. To hide memory latency, the memory operations overlap with data computations. This requires a second memory and the block_switch signals denotes which of the 2 memories is used to store data on and which is being used for computation.

On the right side of the figure two groups of outputs can be seen. The first group handles the slots that came free, so new rows can be loaded in that spot. The second group returns the final row result and its index. The latency to push data in the pipeline is 11 cycles, so we take the 8 slots.

The data feed to this unit comes from 3 memories. The first memory contains the elements of A, the second memory the elements of **b**, and the third memory the elements of vector **c**. To handle the complexity, 3 units called Input_controller, Info_Table and Load_controller are responsible for filling data to the multiply-add unit. Each of them will be discussed now.

The Input Controller: The tasks of the Input Controller is to accept the rows, update the information of this row in the component Info_Table by storing the row size, the index of the current row and to state that the slot is valid. In addition, it writes the accepted values of A, **b** and **c** in the correct locations. The memory addresses are

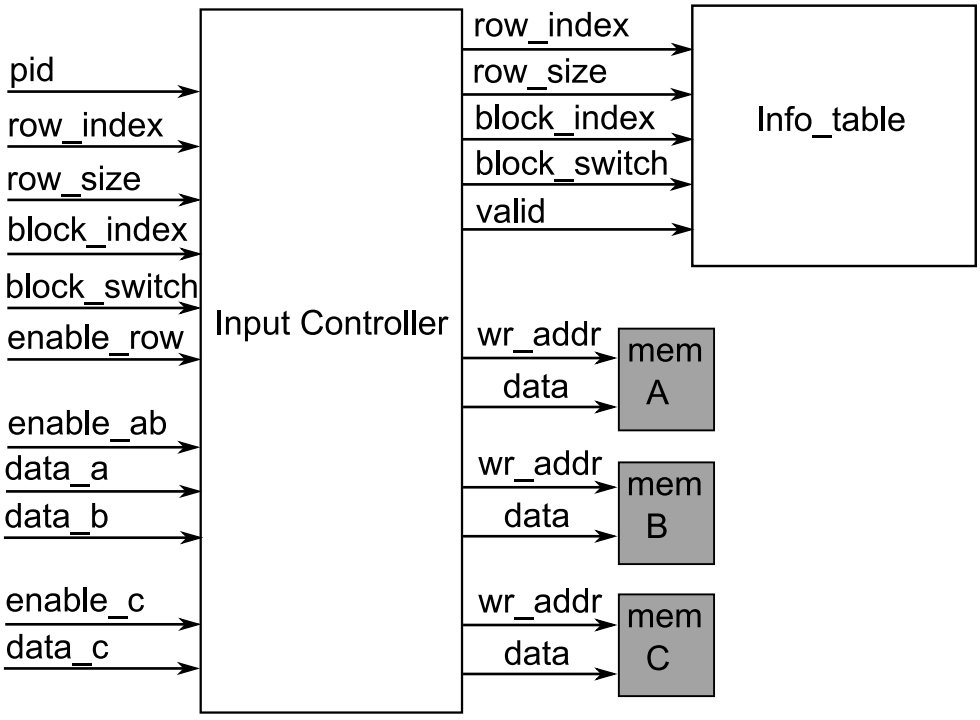

Figure 4.6: IO ports of the component Input Controller

generated by combining the slot index, the offset of the current element with respect to the current row. The addresses of these locations are composed by the block_index and a counter that keeps track of the number of elements of the current row currently written to memory. Figure 4.6 contains the Input and Output ports of the Input Controller and Figure 4.7 shows a timing diagram for the Input Controller.

When *enable_row* is high, all related row information like the *row_index*, *row_size*, *block_index* and the *block_switch* is registered inside this component. Each cycle when the value of enable_ab is high, the pairs data_a and data_b are stored in memory A and B. Later on, the address indices are explained. A counter keeps track when all the row elements are read in and compares this with the *row_size*. When the element of vector $\mathbf{c}$ is read in, a boolean is raised to store this. In case A$\mathbf{b}$ is calculated ($\beta = 0$), the value zero must be written for each new row, to clean up row results of previous calculations. When all elements of the row are read in, the valid signal to the component Info Table is raised.

The Info Table: This component can be seen as a table that stores all the information related to the rows. It keeps track of the row size in each slot, the current element of the row that is processed and forwards this data to the Load Controller. Further, it administrates whether the slots are used to fill with data (communication) or processing (computation). Each memory used to realize this is explained below:

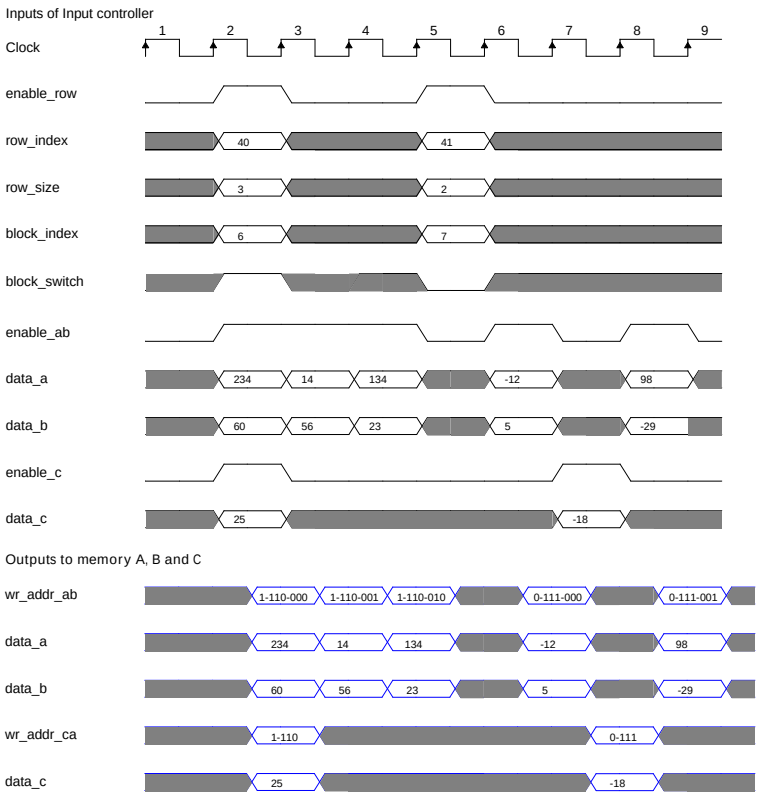

Figure 4.7: Timing Diagram of the Input Controller of the Processing Element

- **row index** Two Simple Dual Port memories 32 bits wide (width can be specified in a configuration file), both 8 entries, store the row index of the matrix. The write port to these memories are controlled by the Input Controller which updates old values when new rows are accepted. The read requests to this memories are coming from the Load Control, which needs the values for block that has access to the Floating Point unit. Only 1 of the 2 memories is forwarded to the output, depending on the active signal for the current requested block.

- **row size** The memories which contain the row_sizes for each slot are constructed in the same way as for the row index. Requests are coming from the Load Control unit and new values are updated by the Input Control unit when new rows are accepted. The active bit for the requested block selects between the output value of 2 Dual Port SRAMs, the one that is currently used for computations. The widths of the memory depend on the largest row size specified in the configuration file.

- **current_index** Since each row starts at a different time, due to different row lengths, we need to know how many elements already are processed of this row. The *current_index* keeps tracks of this. Only 1 Simple Dual port memory is required for this case to store the current_index 32 bits wide, consisting of 8 entries. The

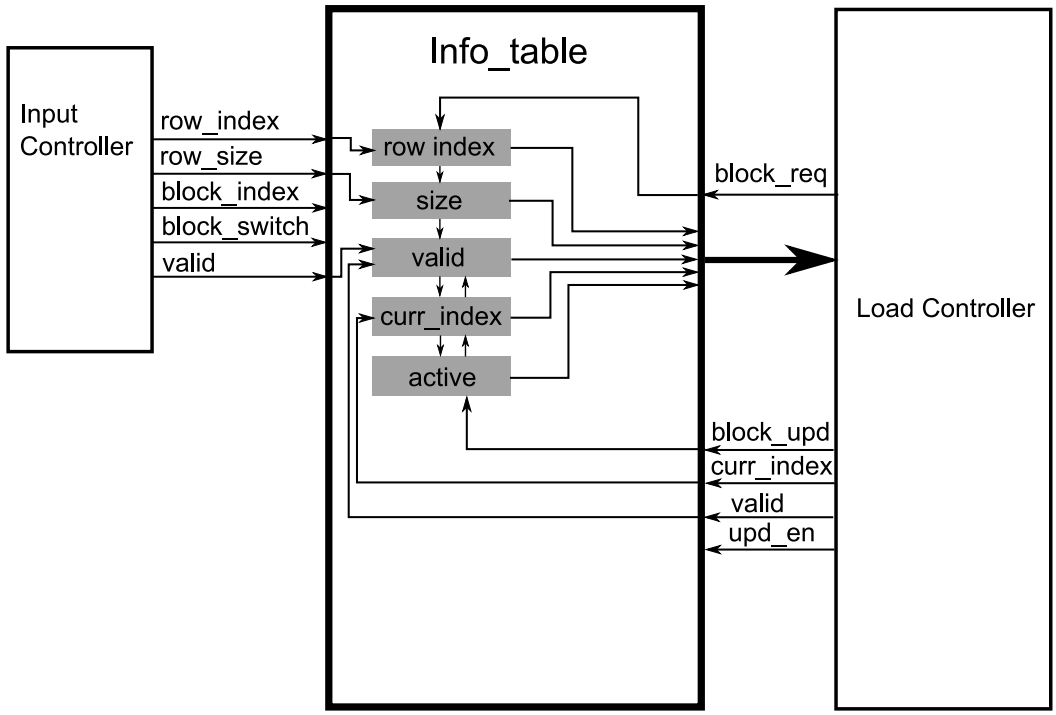

Figure 4.8: Organization of the component Info Table

request (*block_rqst*) and updates (*block_upd*) signals are both connected to the Load controller. The memory is initially reset to zero and the load controller is responsible for the correct usage and resetting the value.

- **valid** For each slot, a valid bit is stored that administrates if that particular slot contains valid row data. The Input Control unit sets the valid values high when new rows are accepted and the load_controller invalidates them when a row is finished with its computations. Two Simple Dual Port memories of 1 bit wide containing 8 entries store the valid bits for each slot. The valid bit is updated when a new row is accepted from the input controller or when its invalidated (denoted by the *valid* signal) by the load_controller.

- **active** An active bit for each slot is stored to keep track of which part of the memory is doing computation and which part is being filled by the Host controller. The size of the register is 8 bits. Initially, all the slots with *block_switch* value zero are active and the valid bit for each slot is initially set to zero.

Figure 4.8 depicts a somewhat simplified version of the organization of the Info_Table. The control signals from the Input Controller and the Load Controller are also included. The total cost generating hardware for this component is using only 162 slices for the Virtex-4 LX200 device running at a maximum frequency of 250 MHz. The memory address widths that are used are 32 bits wide. In a configuration file this address width

can be adjusted. This component can be improved further by combining memories that are split in 2 Simple Dual ports for the mentioned memories into 1 Simple Dual port. This saves some address wires and multiplexers. The approach to take is similar as storing the values of matrix A and vector **b** in the memories A and B.

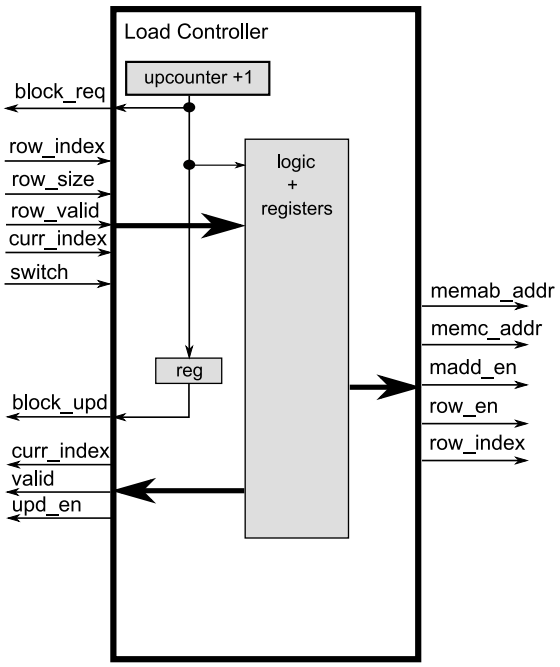

Figure 4.9: Organization of the component Load Control

The Load Controller: The load controller is based on 1 bit up counter that generates the addresses for the Info_Table unit. If a non-valid row has access to the multiply add unit, the Floating Point unit will be disabled. If a valid row is returned, a row address for the Memories A, B and C will be generated, based on the switch value for the current block, its address and the value of the current_index. The multiply add unit will be enabled (madd_en). If the current_index equals the size of the current active row, also row_en will be set high, which signals that the computations for the row are finished. This signal is passed together with the index of the row (row_index) through the Floating Point unit to the output of the Processing Element. Since, its determined here, when a row finished its computations, this component also controls the signals to state which slots became free. In Figure 4.9 the contents of the Load Controller component can be seen.

Memory A, B, C: Memory A is used to store the active row elements of matrix A, while memory B does this for elements of vector **b**. Last, Memory C is used to store initial values of the vector **c** and stores also the temporary row results. Rows of the matrix are assigned to empty slots.

Memory A is depicted in Figure 4.10. The Input Controller stores elements on the correct addresses by combining the active signal (block_switch), slot id (block_index) and the row offset. The load controller generates the address to read the data and the output of the memory is forwarded directly to the Multiply-Add unit.

Memory A, consist out of 2**row_length*8 entries. The number 2 represents here one memory for buffering and the other one for active computations for each slot. The factor 8 here is due to 8 available slots. The *row_length* can be specified in a configuration file and denotes the maximal row length of each slot. If the maximal row length equals 6 as in the profiled mesh for example, the total memory size for A will be 96 DP floating numbers. Memory B is composed exactly in the same way as memory A.

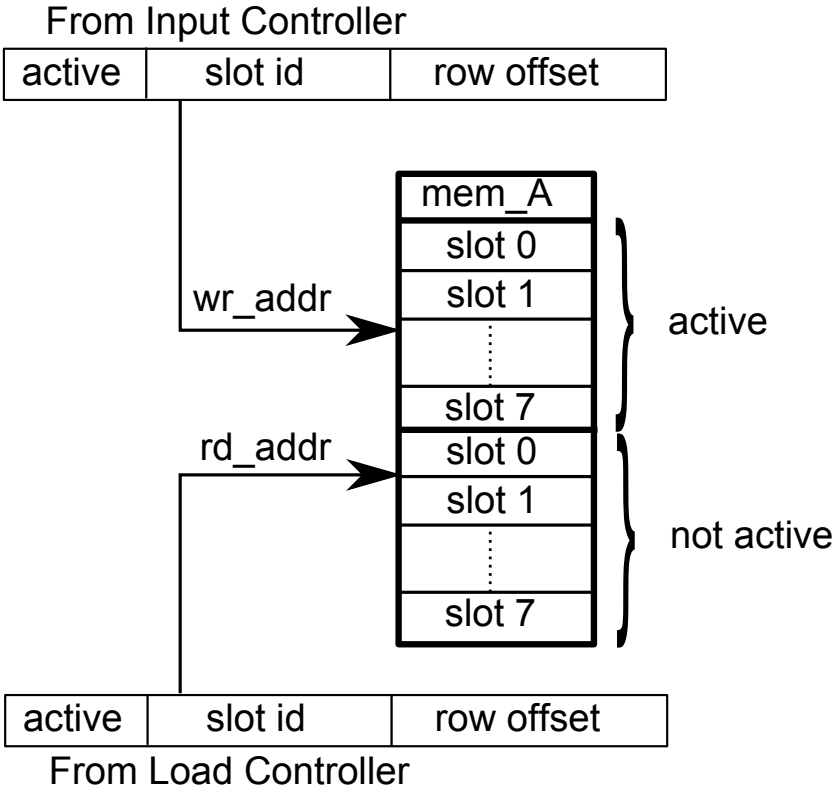

Figure 4.10: Organization of Memory A inside the PE

Memory C contains 2 True Dual Port memories as can be seen in Figure 4.11. The host stores a new element of **c** in the correct memory, according to the value of the active (block_switch) signal and slot id (block_switch). The Multiply Add unit is reading temporary results from the correct memory address and stores the new temporary result in it. A simple Look Up Table (LUT) forwards the wr_address to the correct memories with its writes control signals (not shown in the figure). The output multiplexor that selects output data_c between the 2 memories is controlled by the rd_addr signal from the Multiply Add unit. Each slot contains 1 element only for this component. It is

possible that both *wr_addr* coming from the Input Controller and Load Controller are active at the same time in the same memory, but it is not possible that they write to the same address. This memory can be improved by removing one of the memories. Only one memory is required that stores data from the Input Controller. The second memory, used by the Multiply-Add core can be removed, if intermediate results are directly looped back into the multiply-add unit. Extra logic must be designed for this to select between the initial value from the memory and the intermediate results. This signal can be provided by the Load Control, since it can easily determine if the first element of the row is processed.

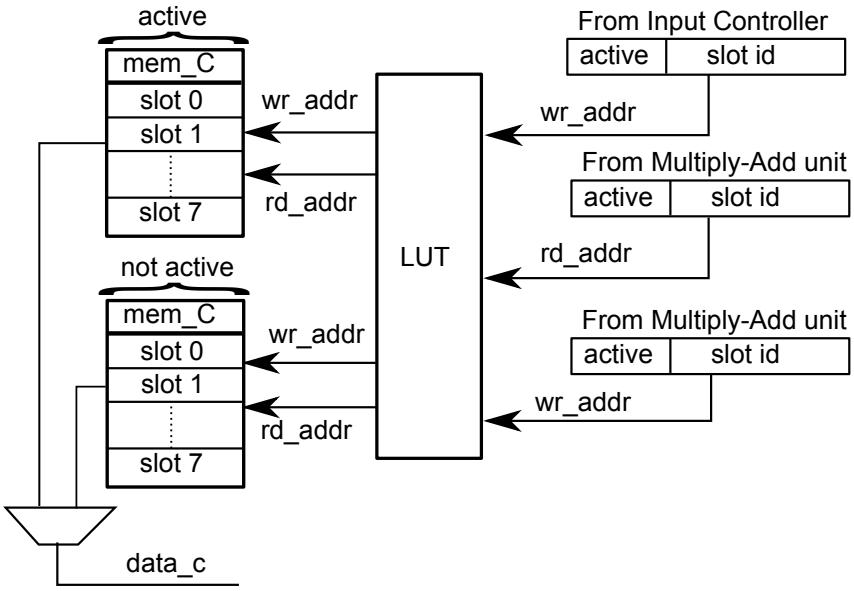

Figure 4.11: Organization of Memory C inside the PE

The multiply add unit: The input and output ports and the connections to other components of the Multiply Add unit is shown in Figure 4.12. The Multiply-add unit operates on data from the Memories A, B and C. The load_controller enables the multiply add unit with the *enable* signal each time when valid data is fed into the multiply add pipeline. The multiply add unit operates in the first 3 cycles on data_a and data_b only. Data_c is not required yet, and therefore addr_c is also forwarded into the pipeline. At the 3rd cycle an access to memory C is requested, to load the correct element of initial vector **c** or a temporary row result. The outputs of the multiply add unit is the result after 11 cycles. This result is stored back in the Memory C or forwarded to the output of the PE if *row_valid* is high. The signal *row_valid* equals the *row_enable* signal delayed with 11 cycles. Similarly, *row_value* is the 11 cycle latency signal of *id_row*. The data path of the multiply-add unit of [25] has been modified to include index support and valid row values (the final value of a row).

Table 4.1 summarizes the frequencies and required number of slices for each of the

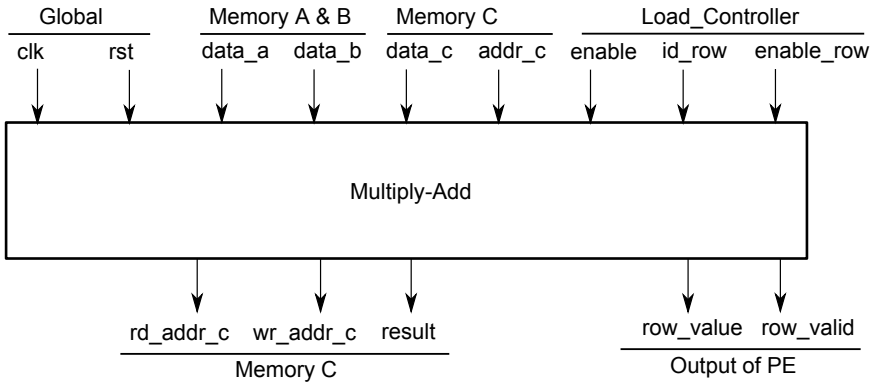

Figure 4.12: Input and output ports of the Multiply Add unit and its connections to other components

units inside the PE. The slices are the occupied slices, which contain logic and routing. The table does not include the cost of the 9 DSP48 18 $\times$ 18 bit multipliers, the DSP48 multiplier are basic blocks of the Floating Point multiplier. The multiply-add core contributes most the total cost of the PE.

Component	**slices**	**frequency (MHZ)**
Input Controller	95	265
Info Table	162	252
Load Controller	84	449
Multiply Add	1,414	159
Processing Element	2,140	156

Table 4.1: Area cost and frequency analysis of each component of the Processing Element

4.2.4 Host

The host contains the memory controller and is responsible for providing the data to the processing elements in the correct way (see input of Input controller). The Host is constructed from 2 smaller components, one responsible for generating memory addresses and the other to provide the data to the processing elements.

An implementation of the Host with 5 memory banks In this design each array that is required to do the CRS SMVM is stored in a separate memory, as depicted in Figure 4.13. By doing this, higher bandwidth can be achieved compared to using 1 memory. We implemented here the CRS scheme instead of the MSR scheme for the following reasons:

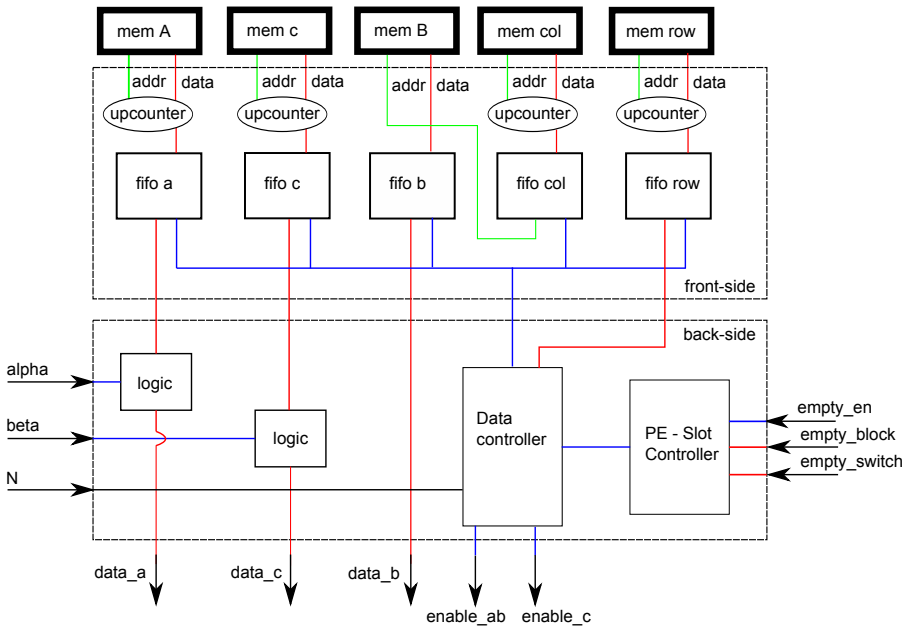

Figure 4.13: Organization of the front-side and back-side of the Host controller

- The MSR scheme complicates the memory controller, if for example both the diagonal and non-diagonal are stored int he same memory bank.
- The MSR requires theoretically less bandwith then the CRS, however practically no advantage can be gained. The theoretical bandwith reduction is a consequence of storing the diagonals seperately which do not require load instructions for the column coordinate. Pratically, thye memory bank with the column coordinates will stays idle, if a diagonal element would be loaded. This means that in the practical case, for this implementation, no bandwith reduction can be realized due to actual mapping to the memory banks.

The design contains two subcomponents: called the front-side and back-side.

front side The front side is responsible for generating the addresses and storing the data in FIFOs. Data is stored in FIFOs for the following reasons:

- 13 cycles latency from the FPGA to main memory (SRAM) on the RASC core.
- The convenience to control the data from and to the FIFOs is simpler.

All the addresses to the main memory are generated with counters except for the loading vector **b**. Loading elements from **b** is determined by its column coordinate.

The buffer control is designed to prevent buffer overflow. The FIFO sizes are 16 units large and the memory loads are requested each cycle when the number of elements inside the FIFO equals 3 or less. An additional requirement is implemented when entries

of vector **b** are requested, which is that the FIFO holding the column coordinates must have valid data. The latency to travel data from the input to output of the FIFOs is 1 cycle. Memory request start as soon as the *start* signal goes high.

back-side The back-side has 4 tasks:

- A Finite State Machine (FSM) controlling data coming out of the FIFOs from the front side. It provides data to the input of the PE compliant to the inferface of the Input Controller. Each time a new row is read in, the *enable_row* signal is asserted. Elements of matrix A and vector **b** must enter in pairs. The FSM assures no buffer underflow for the data inside the FIFOs, to avoid data loss.
- Modifying the values of A and **c**, according to the values of α and β. With them, $\alpha A\mathbf{b}+\beta\mathbf{c}$ is computed in the right way. The supported values for α are $\{-1, 1\}$ and for β $\{-1, 0, 1\}$. Several bits of the floating point numbers α and β are inspected, and accordingly, the sign bits of data_a and data_c could be reversed. In case β is zero, it is not required to load entries of the memory for elements of **c**. The value of data_c will stay zero.
- Keep track when the *done* signal must be raised. The done signal is raised when all the results are written back to memory.
- An FSM controlling the empty slots of the PE. Initially, 16 slots are free since no computations are started yet. These first 16 slots are assigned manually from the host by a simple FSM. The ordering of slots that became free coming from the PE, which depend on the row length, are stored into a FIFO. When this FIFO contains valid data and a new row is detected, data can be forwarded to the next empty slot, residing inside the PE.

An implementation of the Host with 1 memory bank In this design all the arrays, A , **b**, **c**, col, row are stored in 1 memory. The design is similar to the one with 5 memories, except that the front side contains an arbiter to arbitrate which values are loaded.

4.3 Connecting the design to the RASC-services

Besides external memory, the SRAMS, the RASC-core provides the ability to use streams. These streams have been included to increase the bandwidth to obtain higher performance. The design is targeted to achieve a maximal input bandwidth of 11.2 GB/s at a frequency of 200 MHz. Since the design is not routing on 200 MHz, a supplemental clock for the CCU of 100 MHz is required. This is not implemented in this thesis. Currently, 2 PEs are routed as depicted in Figure 4.14. It inclused 2 DMA streams and uses an input bandwidth of 5.6 GB/s. Due to limitations of the SRAM sizes (8 MB per SRAM), the following conditions must hold, $N < 1000000$ and $N_z < 2000000$, for successive computations. The number of processing elements can be doubled to 4 running at a frequency of 100 MHz if the algorithmic clock is set to 200 MHz. This is future

work. The figure does not contain the controllers that control the bandwidth from the memory banks to the PEs. Different components are designed to control this with the following functionality:

- mem_load_a: The elements of matrix A are streamed through the DMA streams directly into the FPGA from main memory. This component interfaces between the host of the PE and the RASC-core. The data_width of the stream equals 128 bits and is converted to a 64 bit output. The streams behave in a similar way as FIFOs.

- mem_load_b: This component is responsible for loading the elements of vector **b**. When new column coordinates are available, new elements of vector **b** are requested. Further, this unit is responsible for the buffer overflow control.

- mem_load_c: This component is responsible for loading the elements of vector **c**. Since both PEs share the same memory (SRAM 4) an arbitration is taken place between request from both PEs. Further, this unit controls the FIFO buffer overflow for both PEs and keeps track which received data belongs to which PE.

- mem_load_row: This unit does similar work as the mem_load_c component, but with extra control converting two row values stored in 64 bit format, into to separate 32 bit values. Since addresses are 32 bits, it might be possible that the first element of the the PE could reside in the upper or lower half of the 64 value. This unit selects the correct output depending on the start address.

- mem_load_col: This unit does exactly the same work as the mem_load_row unit.

An overview of how the RASC software layers is depicted if Figure 4.15. The application communicates with the FPGA from the RASC Core Services library space. The Core Services include all the functions required to communicate from the application to the FPGA, which includes data transfer from main memory to the SRAMS in both directions, streaming of data to the FPGA and sending and receiving function arguments through the Algorithmic Defined Registers and Algorithmic Debug Registers. The limitations and a more detail descriptions of the library functions are described in Appendix D.

4.4 Combining Dense with Sparse Matrix Vector multiplication

In this section, we combine 2 processing elements of the following designs:

- The processing element of the dense matrix vector multiplication design in [25].

- The processing element of the sparse matrix vector multiplication design presented in Section 4.2.3.

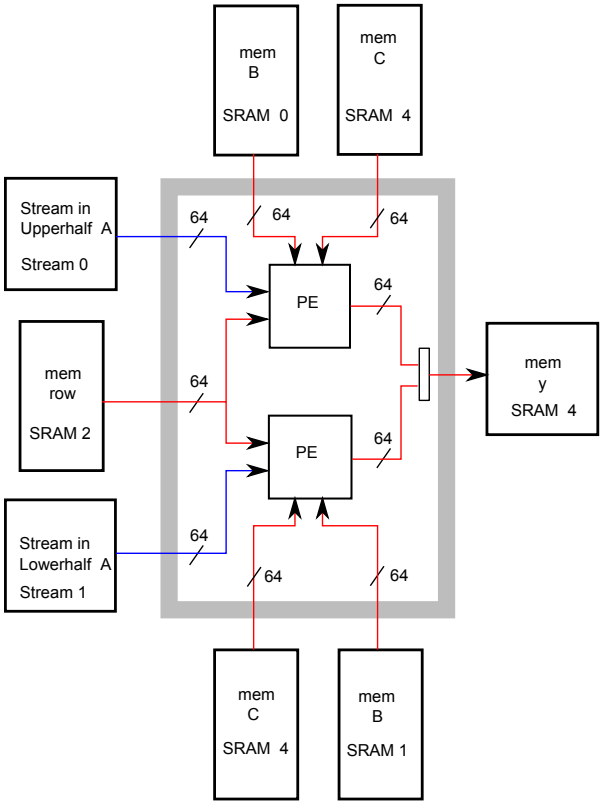

Figure 4.14: Design of the SMVP including 2 DMA-streams

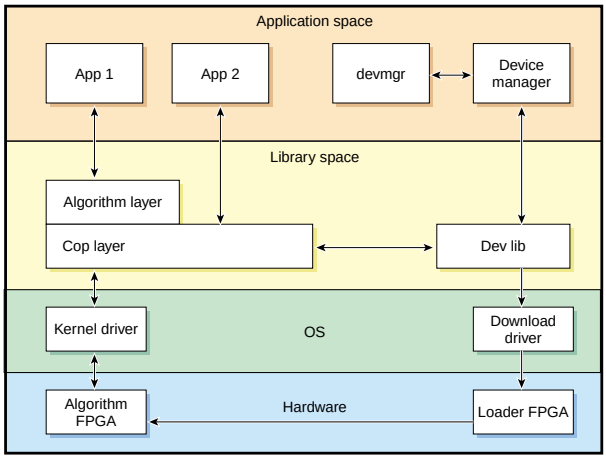

Figure 4.15: RASC Software Overview [13]

Figure 4.16 shows a design that contains the integration of the 2 PEs. Memories A of both designs are combined into 1 memory in which the size is specified in the config-

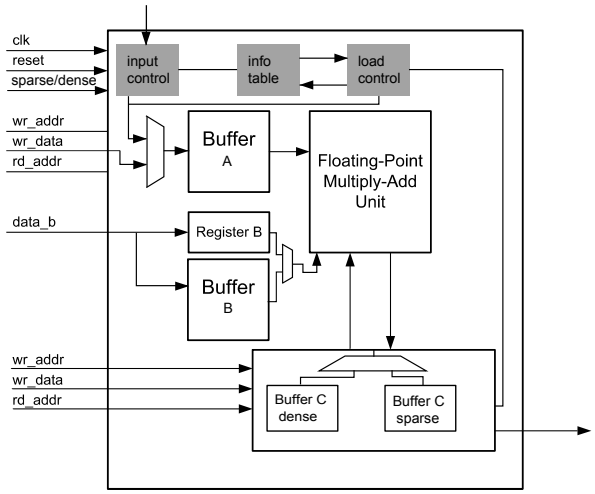

Figure 4.16: Processing element for the combined dense/sparse design

uration file. The maximum value between the two values is selected. The address and data inputs between both designs are multiplexed and controlled with the sparse/dense input signal.

The dense design does not require memory B, but a register is used to store values of vector **b**. Currently, data_b is both forwarded to this register (dense case) and the memory B (sparse case) and a multiplexers selects the desired signal. Later, this can be improved by storing this element at index 0 of memory B. This saves the register and avoids multiplexing between 2 64 bit signals (output of memory and register). A smaller multiplexer is required to select between address 0 for the dense case and the address normally used for the sparse case.

So far, Memories C of both designs have not been integrated together. Efforts have been focused to integrate the controllers (the LUT in figure 4.11) for the sparse and dense case, which turns to be a complex task. In Figure 4.16 it will be simpler by multiplexing all in and outputs of the combined memory, in a similar way as for the combined Memory A.

The dense multiply-add unit has been modified for the sparse multiply-add unit by appending signals specifically used for row identification and final result identification for the sparse design. The dense case does not require this signals and in the combined PE only sparse related components like the Load Controller are connected to it.

So far, we have verified the combined processing elements using 1 mixed PE capable of computing both the Sparse and Dense Vector matrix product. The combined PE requires the same amount of cycles as for the case when the sparse or the dense case is executed separately. The CCU_Core component of the dense design has been modified in such a way that the host_controller has been included to provide the data for the sparse_design. Depending whether a sparse or dense case matrix vector multiplication is used, one of the memory controllers, host_control for the sparse design and the Load_ctrl and Store_ctrl for the dense design, gain access to the main memory. Fundamentally,

there is no difference for the PE that compute both dense and sparse matrix vector products. But accessing the non-zero elements is different in both designs.

4.5 Conclusions

In this chapter, the design methodology is described developing the SMVM hardware unit. The platform and architectural choices have been discussed. The Processing Elements of the SMVM design are low cost in terms of area. Next, the Processing Elements of the presented SMVM unit and the PE of the dense design in [25] have been integrated together. In the next chapter performance analysis for both units is performed.

Design evaluation: Performance Analysis and test results

5

The previous chapter described the SMVM kernel. This chapter continues with performance analysis for the hardware accelerator. The performance of the hardware unit will be compared with current literature and software performance. Finally, conclusions for the efficiency of the design and the impact of the overall performance of the SMVM unit for the OpenFOAM toolbox are discussed.

5.1 Symbols

The symbols used in this chapter are summarized in Table 5.1.

Symbol	*Meaning*
N	The dimension of the matrix N by N, and vector **b c** both N by 1
N_z	The number of non-zero entries in the given matrix
B	Bandwidth in $\frac{words}{sec}$ (1 word equals 8 Bytes)
P	Performance in $\frac{operations}{sec}$
PE	Processing Element
N_{PE}	Number of PEs
f_{op}	The operating frequency in MHz
γ	The ratio between non-zeros and total entries of matrix

Table 5.1: List of symbols used in the analysis

5.2 Theoretical limitations

5.2.1 Number of processing elements

The quantity of PEs that can be placed and routed depends on the availability of the amount of hardware resources and bandwidth. Current FPGA technology, contain abundant hardare resources. Therefore, it is very likely that the number of PEs is limited by the available bandwidth. The more bandwidth available, the more processing elements can be placed in order to keep the bandwidth utilized. Since each PE is working independently, meaning incoming data from main memory is not shared between the PEs, due to row splitting. We calculate the bandwidth for 1 PE and scale this up to N_{PE}.

To obtain high resource utilization, the multiply-add unit inside the PE must be occupied as much as possible in each cycle. Considering 1 PE, the total amount of matrix operations equals $2N_z$ operations and can be finished in N_z cycles if 1 PE is used. This requires that all the data from main memory must be loaded in this amount

of cycles. We assume here that computations and communications overlap and that the computation time is zero. Assuming memory addresses of 32 bits width, (5.1) describes the relation between number of processing elements minimal required to use the complete bandwidth. Given the number of PEs, (5.2) describes the minimal bandwidth required to keep all PEs working continuously. The assumption is made in these formulas that the work is divided equally among the Processing Elements.

$$N_{P_{E,sparse}} = \left\lceil \frac{N_z B}{f_{op}\left(\frac{5}{2}N_z + \frac{5}{2}N\right)} \right\rceil = \left\lceil \frac{2N_z B}{5 f_{op}\left(N_z + N\right)} \right\rceil \tag{5.1}$$

$$B_{sparse} = \left\lceil \frac{f_{op} N_{PE}(\frac{5}{2}N_z + \frac{5}{2}N)}{N_z} \right\rceil = \left\lceil \frac{5 f_{op} N_{PE}(N_z + N)}{2N_z} \right\rceil \tag{5.2}$$

The formulas can be obtained by analyzing the cost of the memory accesses to the arrays in Table 5.2. The table contains for each array in the first column, the data width and the total memory accesses to main memory for this array. Each element of vector **c** is accessed twice, the first time to load the initial value, and second time to store the row result in.

Array	data width (words)	Number of accesses
A	1	N_z
b	1	N_z
c	1	$2N$
row	0.5	$N+1$
col	0.5	N_z
Total	1	$\frac{5}{2}N_z + \frac{5}{2}N$

Table 5.2: Load cost analysis of arrays performing a SMVM

5.2.2 Performance of the SMVM hardware unit

The performance for the hardware sparse matrix vector multiply unit can be limited by:

- The amount of available computational resources, the number of PEs.
- The bandwidth, the higher the bandwidth the more data can be processed in parallel.

Equation (5.3) contains the formula in which the performance is limited by the number of PEs. Equation(5.4) shows this formula when the performance is limited by the bandwidth.

$$P_{sparse} = 2N_{P_E} f_{op} \tag{5.3}$$

$$P_{sparse} = \frac{2N_z B}{\frac{5}{2}N_z + \frac{5}{2}N} = \frac{4N_z B}{5\left(N_z + N\right)} \tag{5.4}$$

The former formula can be explained as follows: Each PE has 1 multiply-add unit and therefore can execute 2 operation per cycle. Multiplying this with the operating frequency results in the maximum performance.

In the case when the bandwidth limits the performance, the performance is limited by 5.4. Here, the assumption is made that enough processing elements are available. The formula is calculated by $\frac{T_{oper}}{T_{load}}$, where T_{oper} the total amount of operations equal to $2N_z$ and T_{load} the total time needed to load all data from main memory, which equals $\frac{5(N_z+N)}{2B}$.

The performance in (5.4) is an approximation of the performance limitation. In reality, the performance will be lower due to the following overheads:

- There are some clock cycles latency between the arrival of the data from main memory and the arrival of the data to the multiply-add unit. There is also some latency between the outputs of the multiply-add unit and the main memory.
- After the last memory element is loaded, at least the last row still need to be fully computed.

To verify equations (5.3) and (5.4), two designs, one with 1-memory bank and one with 5-memory banks are tested. The designs are explained in Section 4.2.4. Since the design with 5-memory banks is also verified in hardware, a hardware counter is included to verify the numbers. The design with 1 memory, tests (5.4) since bandwidth limits the performance. The required bandwidth for 1 PE is slightly overdimensioned with 5 memory banks and is liable to (5.3).

N	N_z	(5.4)	1-mem	Efficiency (%)
1000	3000	10000	10055	99.5
1000	30000	77500	77775	99.6
10000	30000	100000	100055	99.9
10000	49997	149993	150065	≈100.0

Table 5.3: Measured vs analytical number of execution cycles for different cases in simulation, with a bandwidth limitation, predicted by (5.4) using 1 memory bank.

N	N_z	(5.3)	5-mem (HW)	Efficiency (%)
1000	3000	3000	3064	97.9
1000	30000	30000	30280	99.1
10000	30000	30000	30064	99.8
10000	300000	300000	300280	99.9
10000	49997	49997	50303	99.4

Table 5.4: Measured vs analytical number of execution cycles for different cases in hardware, with a resource limitation, predicted by (5.4) using 5 memory banks.

Tables 5.3 and 5.4 summarize simulation and hardware results for some matrices which non-zeros are generated in a random pattern. The first two columns obtain for

both tables the matrix sizes. For Table 5.3, the 3^{rd} column contains the number of cycles derived from (5.4). The 4^{th} column contains simulation result using a bandwidth of 8 Bytes/cycle = 1 word/cycle. The estimated cycles can be calculated by:

$$cycles = \frac{2N_z}{P} \tag{5.5}$$

with P the performance, in which the bandwidth B specified in words/cycle in (5.4). The last column contains the efficiency of the design, which is calculated by ratio of the expected cycles and the measured number of cycles. For Table 5.4, the 3^{rd} column contains the number of cycles derived from (5.3). The 4^{th} column contains hardware cycle results. The last column shows the efficiency of the design.

5.2.3 Dense vs Sparse Matrix Vector Multiplication

The integrated dense and sparse matrix-multiply unit allows users to select between a Dense Matrix Matrix Multiplication, a Sparse Matrix Vector Multiplication and a Dense Matrix Vector Multiplication using the same hardware, without (complete) reconfiguration. The question rises, under what conditions we should perform a Dense Matrix Vector Multiplication or a Sparse Matrix Vector Multiplication. In this section, an estimation is given for this. Moreover, a discussing of the number of processing elements given a fixed bandwidth for both designs is also provided.

The decision rule can be made by comparing the execution time for both designs. Assuming both designs have enough processing elements and performance is limited by the bandwidth, the following equations determine the executing time for the sparse and dense matrices given N and N_z.

$$t_{execution} = \frac{total_{operations}}{P} \tag{5.6}$$

$$t_{exec,sparse} = \frac{5(N_z + N)}{2B} \tag{5.7}$$

$$t_{exec,dense} = \frac{N^2}{B} \tag{5.8}$$

The performance for the dense matrix vector product is limited by $2B$ [25] and the number of operations required are $2N^2$. By combining (5.7) and (5.8), we derive the following relation for N and N_z:

$$N_z = 0,4N(N - 2,5) \tag{5.9}$$

$$N = 1,25 + \sqrt{\frac{5}{2}N_z + 1,25^2} \tag{5.10}$$

Equation (5.11) states the ratio of non-zeros in a matrix. By substituting (5.9) for N_z in (5.11) we obtain (5.12).

$$\gamma = \frac{N_z}{N^2} \tag{5.11}$$

$$\gamma = 0,4 - \frac{1}{N} \tag{5.12}$$

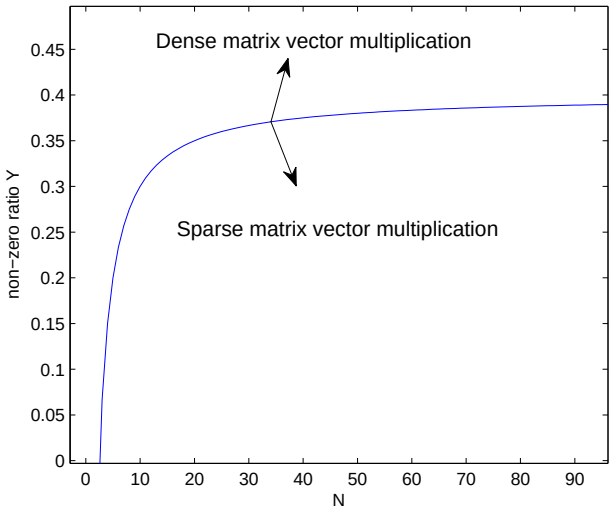

Figure 5.1: Boundary condition, where the performance for the SMVM equals the DMVM.

Figure 5.1 depicts the boundary condition in which the sparse and dense case equal in performance, given equal bandwidth for increasing N (matrix size) on the x-axis. The y-axis contains the density of the matrix. We conclude the following from our analysis:

```
% Considering large matrices, the execution time will be faster for
the dense matrix vector design when 40% or more of the matrix
is filled.
```

The previous discussion assumed enough processing power to keep up with the bandwidth. The discussion continues by considering the number of processing elements required to do the SMVM and DMVM matrix multiplications.

The number of processing elements required for the sparse case is written in (5.1). The number of processing elements for the dense matrix vector multiply hardware unit is harder to calculate since it depends for example on the local memory sizes [5]. Therefore we assume the circumstance where the bandwidth is matched with the number of processing elements for the dense hardware unit. That is:

$$N_{PE_{dense}} \leq 2B \quad = \quad 2f_{op}P \tag{5.13}$$

From this formula, the number of processing elements can be calculated by

$$N_{PE_{dense}} \quad = \quad \frac{B}{f} \tag{5.14}$$

Combining (5.1) with (5.14) and assuming equal bandwidth for both designs, the ratio between the number of processing elements for sparse and dense equals

$$\frac{N_{PE_{sparse}}}{N_{PE_{dense}}} \quad = \quad \frac{\frac{5}{2}N_z + \frac{5}{2}N}{N_z} = \frac{5(N_z + N)}{2N_z} \tag{5.15}$$

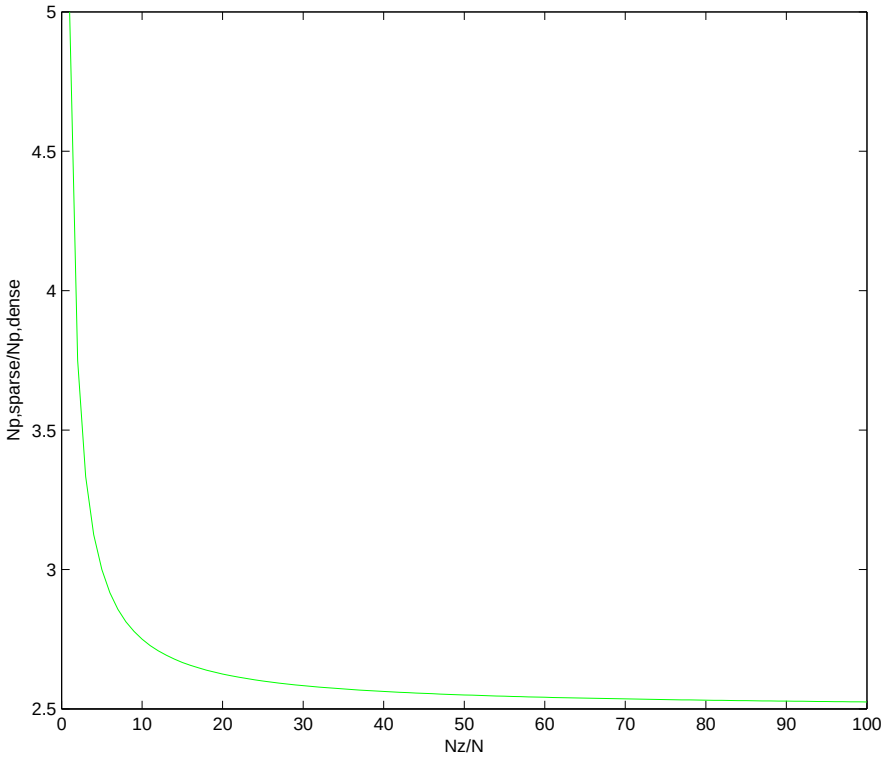

Figure 5.2: Ratio of the number of processing elements for dense and sparse increasing the average row length and considering equal bandwidth

Figure 5.2 shows graphically the formula in (5.15). Two observations can be made from this figure:

- $N_{PE_{sparse}} < N_{PE_{dense}}$. The Dense Matrix Vector Multiplication (DMVM), does not include auxillary index arrays, that need to be loaded to compute the product. This means that given an equal bandwidth, more data can be processed in the DMVM relatively to the SMVM. Therefore, the DMVM always requires equal or more PEs for the same bandwidth.

- By increasing $\frac{N_z}{N}$, the average row length, the overhead of loading the index vectors becomes less. This explains the decreasing line in Figure 5.2.

Equation (5.12) assumed equal bandwidth, and enough processing elements. It does not include however the available number of PEs. The discussion above proved that the dense case needs more PEs considering equal and fully used bandwidth. If we define D as the bandwidth required to match the work for $N_{PE,dense}$ processing element and S the for the $N_{PE,sparse}$, we can separate the following three cases for the condition were $N_{PE} = N_{PE,dense} = N_{PE,sparse}$:

- $B < D < S$

- $P_{sparse} = (5.4) = \frac{4N_zB}{5(N_z+N)}$ (limited by bandwidth)
- $P_{dense} = 2B$ (limited by bandwidth)
- $\gamma = (5.12) = 0.4 - \frac{1}{N}$

- $D \leq B < S$
  - $P_{sparse} = (5.4) = \frac{4N_zB}{5(N_z+N)}$ (limited by bandwidth)
  - $P_{dense} = (5.14) = 2f_{op}N_{PE}$ (limited by N_{PE})
  - $\gamma = \frac{2BN_2}{5f_{op}N_{PE}} - \frac{1}{N}$
- $B \geq S > D$
  - $P_{sparse} = (5.3) = 2f_{op}N_{PE}$ (limited by N_{PE})
  - $P_{dense} = (5.14) = 2f_{op}N_{PE}$ (limited by N_{PE})
  - $\gamma = 1$

From the three different cases we can conclude the following. In case the performance is limited by the bandwidth for both designs, it is more performance efficient to perform a DMVM in case approximately 40% or more of the matrix is filled. In the case where the bandwidth for the DMVM design is machted or overdimensioned and the bandwidth for the SMVM is underdimensioned, the equation $\frac{2BN_2}{5f_{op}N_{PE}} - \frac{1}{N}$ determines the boundary where the sparse and dense design will equal perform. Last, when both designs are overdimensioned, its always more attractive to perform a SMVM, unless $N_z = N^2$. In this special case, when the matrix is a dense matrix, the designs perform equally.

5.3 Experimental results

In this section the performances in software and in hardware are compared. The performance for the Amul and Residual kernels in both software and hardware are measured. Next, the performance for the hardware design using benchmark matrices from different fields is presented in Section 5.3.2.

5.3.1 Software Performance measurements

Measuring the performance for the Amul and Residual kernel Figure 5.3 is obtained. The plot shows the performance for a different number of matrices obtained from the profiled mesh in section 2.7.4 by assigning different weight factors to the decomposition process. The x-axis show the number of cells for the decomposed subparts of the original mesh, which equals N. Except for the mesh containing 12225 cells, which is the PitzDaily case. Since the performance also depends on N_z, Table 5.5 contains a list of all N, N_z pairs that have been used for the measurements in addition with the average row length. The Itanium-2 9130M is used for the software experiments and contains 16 kB L1 data cache, 256 kB L2 data cache and 4 MB L3 data cache. The machine operates on a frequency of 1669 MHz, with the Front Side Bus (FSB) operating on a frequency of 667 MHz.

N	N_z	$\frac{N_z}{N}$
12225	60565	4,95
47432	265608	5,60
95980	533754	5,56
192543	1099929	5,71
479463	2742493	5,72
958962	5487204	5,72
1920905	10959649	5,70
3429833	19328609	5,70
3428591	19569045	5,71
3499094	19697870	5,62
6522469	37007189	5,67

Table 5.5: A list of the different matrix sizes which are measured for performance in software.

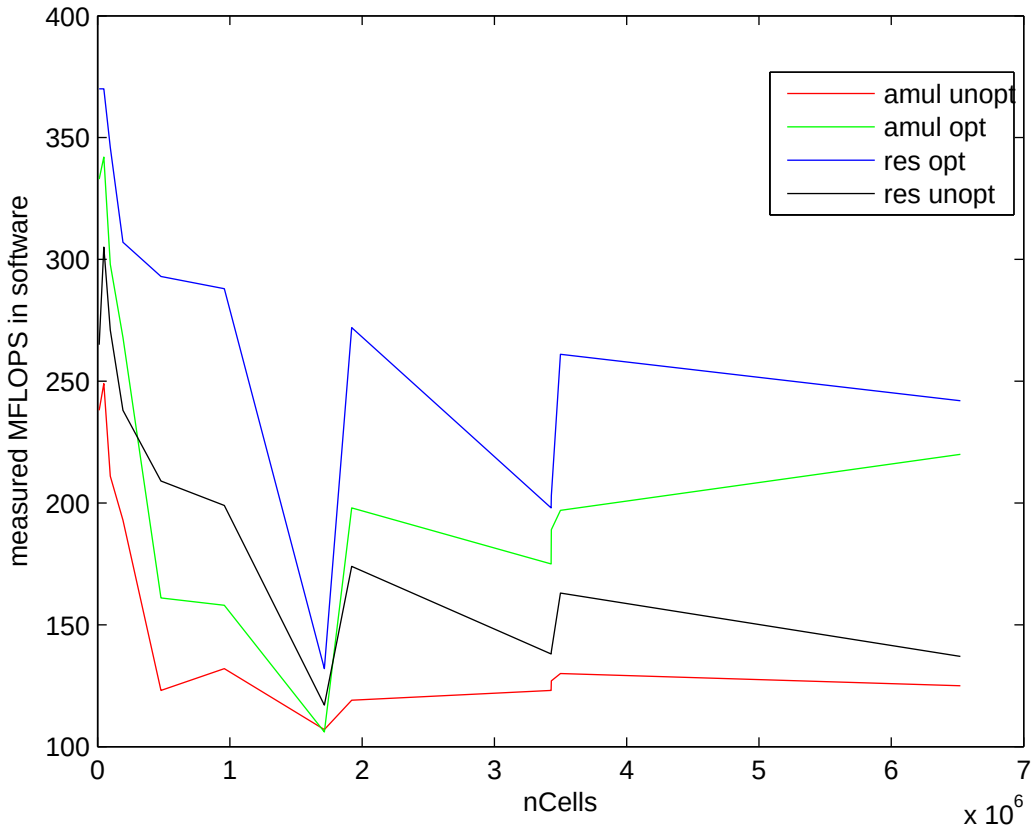

Figure 5.3: The performance in software measured on the Itanium-2 processor

5.3.2 Hardware Performance measurements

Three matrices out of the list of Table 5.5 are also profiled in hardware for the designs with 1 and 2 PE elements. Further the design with 2 PEs has been extended to CCUs containing 4 and 8 PEs elements by instantiating multiple components of the 2 PEs. Table 5.6 reports the performances measured in hardware and software for the different cases for one and two PEs and Table 5.7 for four and eight PEs.

matrix			Number of Processing elements			
			1		2	
N	N_z	$\frac{N_z}{N}$	HW_cycles	P	SIM_cycles	P
12225	60565	4.95	60640	199.8	30359	399.0
47432	265608	5.60	265704	199.9	132905	399.7
958962	5487204	5.72	5487438	200.0	2743733	400.0

Table 5.6: Hardware performance measurements, with P the performance in MFLOPS, for 1 and 2 PEs.

matrix			Number of Processing elements			
			4		8	
N	N_z	$\frac{N_z}{N}$	SIM_cycles	P	SIM_cycles	P
12225	60565	4.95	15216	796	7647	1584
47432	265608	5.60	66494	798.9	33285	1596
958962	5487204	5.72	131933	799.9	686009	1599.7

Table 5.7: Hardware performance measurements, with P the performance in MFLOPS, for 4 and 8 PEs.

In Tables 5.6 and 5.7, HW_cycles denote the cycles required to finish the full computation directly retrieved from hardware. Since there is no bitstream generated for all the different cases, also software simulations are included. Simulation results are denoted with SIM_cycles. For the hardware the design is routed at a frequency of 100 MHz and The peak performance for the cases are 200 MFLOPS, 400 MFLOPS, 800 MFLOPS and 1600 MFLOPS for the CCUs respectively with 1, 2, 4 and 8 PEs. The bandwidth for the benchmarks in Table 5.10 are as follows:

- The design with 1 PE is provided with an Input bandwidth of 2.4 GB/s and Output bandwidth of 800 MB/s.
- The design with 2 PEs Input Bandwidth of 4.8 GB/s and Output Bandwidth of 800 MB/s.
- The design with 4 PEs Input Bandwidth of 9.6 GB/s and Output Bandwidth of 1.6 GB/s.
- The design with 8 PEs Input Bandwidth of 19.2 GB/s and Output Bandwidth of 3.2 GB/s.

These bandwidths are used to calculate $\mathbf{c} = \mathrm{A}\mathbf{b}$. In case $\mathbf{c} = \mathrm{A}\mathbf{b} + \mathbf{c}$ is computed an additional bandwidth for the residual kernel of 400 MB/s is used per processing element, without additional cost in terms of cycles. The Virtex FPGA inside the RASC-core is connected to a maximal input and output bandwidth of 11.2 GB/s. Using this FPGAs in the RASC architecture, at least 4 PEs can be placed on the FPGA running at full speed, since bandwidth is not limiting the performance. When the design is operating at a frequency of 100 MHz, a performance of 800 MFLOPS can be expected in hardware. In software, the performance is fluctuating between 370 MFLOPS and 109 MFLOPS. Thus, a kernel speedup for the SMVM kernel between 2.16 and 7.3 can be achieved per FPGA. Since the rows are divided horizontally the matrices can be easily divided and extended to more FPGAs.

5.3.3 General Benchmarks for the SMVP unit

To verify and measure the performance of our unit against matrices from other domains, we include in this section benchmarks outside the OpenFOAM application. The matrices are from different application fields and can be found in Table 5.8. In the table, eight matrices from the University of Florida Sparse Matrix Collection [3] are included with its dimension, number of non-zeros and the minimal and maximal elements per row. The performance in software is measured with the tool Optimized Sparse Kernel Interface (OKSI) for these matrices and the results are obtained in Table 5.9. OSKI [26], is a collection of low-level C primitives, that provide automatically tuned computational kernels on sparse matrices. OSKI has a BLAS-style interface, providing basic kernels like sparse matrix-vector multiply. The current implementation targets cache-based superscalar uniprocessor machines. The tool has been compiled with the Intel 10.1.015 release with the optimization level -O3. This optimization flag enables aggressive optimization for code speed. The performance results obtained in hardware are summarized in Table 5.10, for 1, 2, 4 and 8 processing elements respectively.

Number	Kind of matrix	Matrix	N	N_z	min	max
1	Structural problem	dwt_162	162	1182	2	9
2	subsequent 2D/3D problem	fs_680_2	680	2424	1	8
3	Power Network problem	gemat12	4929	33044	2	44
4	acoustics problem	k3plates	11107	378927	15	58
5	semiconductor device problem	wang3	26064	177168	4	7
6	optimization problem	jnlbrng1	40000	199200	4	5
7	Thermal problem	epb3	84617	463625	3	6
8	Optimization problem	cont-300	180895	988195	2	6

Table 5.8: Properties of benchmarked matrices[3]

The software results heavily depend on the cache. The results for the sparsest matrices, like jnlbrng1, epb3 and cont-300 report similar performances as for the optimized cache improvement instructions for the OpenFOAM benchmarks. However, OSKI reports low performances for small cases and high results for denser matrices like k3plates.

matrix	P
1	75
2	99
3	291
4	465
5	315
6	267
7	271
8	278

Table 5.9: Software Benchmark results for the matrices in Table 5.8, with P the performance in MFLOPS

matrix	Number of Processing elements							
	1		2		4		8	
	HW_cycles	P	SIM_cycles	P	SIM_cycles	P	SIM_cycles	P
1	734	183	734	322	-	-	-	-
2	2534	191	1368	352	723	671	-	-
3	34485	192	17565	376	8881	744	4642	1425
4	379825	199.5	190956	397	95553	793	47973	1580
5	177253	199.9	88678	399.6	44384	798	22243	1593
6	199416	199.8	99838	399	50072	797	25023	1592
7	463910	199.9	231937	399.8	116016	799	58065	1597
8	988908	199.9	494807	399.4	247747	798	124128	1592

Table 5.10: Hardware Benchmark results in MFLOPS for the matrices in Table 5.8

For this matrix a performance of 465 MFLOPS is achieved, but this matrix contains at least 15 non-zeros per row is much denser then the matrices that are targeted in the OpenFOAM CFD toolbox. Figure 5.4 shows a graphical representations of the benchmark results. On the x-axis the matrix number are displayed. The y-axis contain the performances in MFLOPS for all the measured cases. For the software measurements we used the same machine described in Section 5.3.1.

5.4 Related Work

In this section, a summary is given of the literature of the research so far on the SMVM product. Section 5.4.1 presents techniques to accelerate the kernel on General Purpose Processors, while Section 5.4.2 is focused on the research for FPGA based designs. Finally, this section ends with a conclusion in Section 5.4.3.

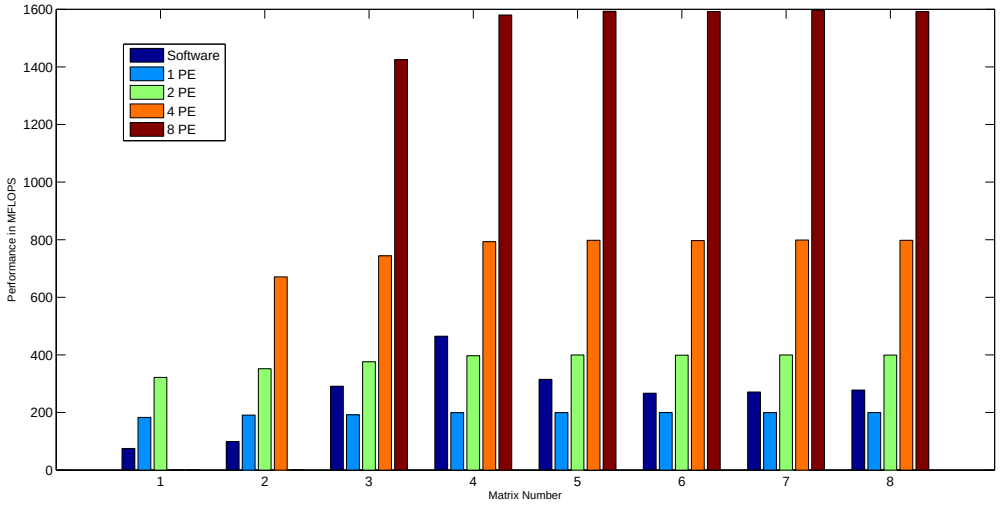

Figure 5.4: Benchmark results using, 1, 2, 4 and 8 PEs and software results

5.4.1 General Purpose Sparse matrix products

Different proposals to improve the sparse matrix-vector multiplication on general purpose processors have been made. Some researchers focus on the exploitation of parallelism of SMVM using multiple machines, while others focus on code and data structure reorganization.

Exploitation of parallelism In [27], a hybrid programming model is proposed to parallel execution of the SMVM on clusters of multiprocessor shared-memory nodes. The sparse matrix format that is investigated is the Jagged Diagonal storage. However, it is shown that this is not the best format to be implemented in hardware. In [20] the average communication cost is calculated in hypercubic networks for the SMVM. Routing the hypercubes on FPGA-based designs is complex. In [8] the authors present a design based on a linear array of processing elements based on a special cover of the non-zeros of the matrix, the staircase.

Code and data structure reorganization The authors of [21] propose a method to pack contiguous nonzero elements into dense blocks to reduce the number of load instructions. A reordering algorithm to increase the sizes of the dense blocks within the matrix is also proposed by them. Sparsity [9] tries to improve performance by means of loop transforming, register and cache blocking. However, little improvement is achieved for very irregular structures. OSKI [26], is a collection of low-level C primitives that provide automatically tuned computational kernels on sparse matrices, for use in solver libraries and applications.

5.4.2 FPGA-based designs

Due to increasing bandwidth, computational resource and available On-Chip memory on FPGAs, Floating Point designs become more popular for hardware implementations. In [28] and [5], a floating-point dense matrix multiplication is implemented. In [7] a sparse matrix vector multiplication is performed in which data from the matrix is converted to "pipelinable vertical nonzero stripes". The input vector is streamed in and the stripes are multiplied by the corresponding vector elements. The design performs well if the number of stripes are bounded. Moreover, the peak performance is high when the ordering of the pipelined stripes and the vector stream are matched. The matrices in OpenFOAM are not bounded by a fixed number of stripes and therefore the design in [7] is not suited for it. On top of that matrices constructed from meshes build out of prisms, pyramids and tetrahedrons will lead to many vertical strips. The vertical strips will lead to a low resource utilization, since the streaming vector has to be stalled each time the same element of the vector is required. In [4] the authors arrange PEs in a bidirectional ring to compute $\mathbf{y} = A^i\mathbf{b}$. The proposed design significantly saves I/O bandwidth due to local storage of the matrix and intermediate results. This limits the input matrix sizes.

So far, to our best knowledge, the closest work regarding the SMVM kernel for matrices in the Final Volume Method (FVM) found are the floating point sparse matrix vector multiplication designs implemented in [29] and [24].

The design in [29] has the following drawbacks:

- Currently, only simulation of the design are made. Our design is also verified in hardware.
- The sparser the matrices, the lower the peak performance (to 20% of peak). The FVM method involves matrices that are much sparser then the test matrices used there.
- Requires high on chip memory for huge matrices. It is possible to section the matrix in vertical slices as they propose and implement. However, they do not include the cost of the additions of the final partial sums.
- The design contains huge adder reduction trees, which depend on different parameters. For a banwidth of 8 GB/s the total number of slices equals 16931 (51% of the device Virtex 2 Pro XC2VP125) and for a bandwidth of 14.4 GB/s the required number of slices for the reduction trees only equals 23856 (73% of the device). The full cost of our design for 8 PEs in terms of area equals 22774 slices (25% of total) for the Virtex-4 LX200 device.

The design in [24] has the following disadvantages:

- Currently, only simulation of the design are made.
- The authors have an improved version of the adder tree reduction circuit by removing some of the adders and inserting FIFOs. The adder tree does not depend on parameters. Our design does not require adder trees.

- The authors section the matrix in groups of rows, and each group of row is also sectioned vertically. This enables them to use efficiently the bandwidth and to compute the full SMVM cost.

Proposal	(Dis)-advantage	Solution
[29] [24]	Simulation	Only our design is verified in hardware.
[29]	Partial result	Our design and [24] obtain full SMVM results.
[29] [24]	Bandwidth requirement	Currently, our design is not using the bandwidth optimally, while the designs in [29] [24] do.
[29] [24]	Peak Performance	Our design always performs close to peak performance (high efficiency). The design in [29] reports low efficiency in terms of performance. The design in [24] reports higher efficiency in terms of performance, but lower then ours Table 5.12 contains more details.
[29] [24]	Adder trees	Our design does not require adder trees

Table 5.11: Advantages and disadvantages for our closest work regarding the SMVM unit

Table 5.11 shows all the advantageous and disadvantages of our closest work compared to our design. Table 5.12 shows performance results for all related work. Elements in the table that contain a N.A (not available) were not found in the presented papers. In the table, for each design, the peak performance in MFLOPS, actual performance in MFLOPS, required bandwidth in GB/s, area in terms of slices, the maximal frequency and the actual frequency in MHz are summarized. The design in [4] is the peak performance per FPGA. For 1 FPGA they obtain an average of 66% of the peak performance. They also simulated 16 FPGAs to calculate the SMVM kernel. The average peak performance for 16 FPGA is 33% of the peak performance. By including more FPGAs the total On-Chip memory increases and larger matrices can be simulated. The authors in this paper implemented an iterative SMVM solver, which consist of multiple matrix vector products.

5.4.3 Conclusions Related work

Current proposals for sparse matrix vector multiply units focus primarily on matrices from the Finite Element Method. In the Finite Volume Method (FVM) the matrices that are involved are usually much sparser, which results in significant performance decrease. So far, no literature has been found that implemented a single SMVM multiplication design without reduction trees. In Section 5.4.4 the differences are explained. Besides the matrix vector multiplication kernel, OpenFOAM contains another similar kernel which computes $\mathbf{c} = -\mathbf{Ab} + \mathbf{c}$. The hardware unit should be able to compute this kernel as well. By controlling α (-1) and β (+1), we are able to do this for our kernel.

Ref	Peak MFLOPS	Actual MLOPS	B (GB/s)	Area	f_{max} - f_{act} (MHz)
[24]	1584	86%-98%	13.2	24129	N.A - 165
[29]	2880	30%-75%	14.4	*23856	200 - 165
[29]	1600	20%-79%	8.0	*16613	200 - 165
[4]	2240	33%-66%	N.A	N.A	140 - 140
[7]	1760	17.4%-86.4%	8.0	**	N.A - 110
Ours	1600	98.7%-100%	22.4	22700	156 - 100

* denotes area cost for reduction tree only.
** Area cost for this design equals 30% of logic resources and 40% internal RAM of the Stratix S80

Table 5.12: Performance results for the SMVM of Related Work

5.4.4 Reduction tree vs our implementation scheme

The reduction tree consist out of multiple adders ordered in a tree. Due the the multiple resources available the summation of a row can be calculated faster. The inputs to the adder tree are limited by the bandwidth. A simple unoptimized adder tree design can be seen in Figure 5.5.

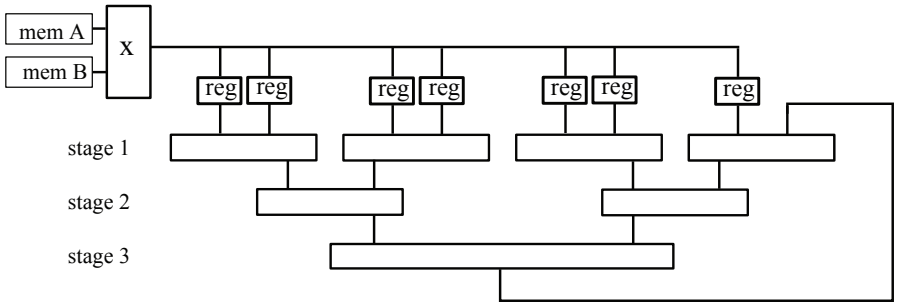

Figure 5.5: Floating Point Adder-tree with 3 stages

symbol	Meaning
L_{add}	adder latency in cycles
n_s	number of adder stages of the adder tree
r_l	current row length, the number of non-zeros on this row
init_{time}	time to initialize the loads from the memory before adder tree can start

Table 5.13: Symbols used in this section to describe performance of the adder tree

We discuss the drawbacks and advantages of the adder tree now by looking to different row lengths. We consider 3 cases:

- row length smaller then the number of inputs of first stage

- row length equal to the number of inputs of first stage
- row length larger then the number of inputs of first stage

For the first case, when the row length is smaller then the number of inputs. The total latency to compute one row equals: $init_{time}+n_s*L_{add}$. A next row can immediately start after all the registers all filled, the $init_{time}$ for the next row. The hardware resources are under utilized due to 2 reasons:

- Not all adders are used, since the row length is smaller then the number of inputs for the adders in the first stage.
- The adders are idle each time no data is filled into the pipelines, that is for each row, 1 out of $init_{time}$ cycles for that particular row. There are N rows, so in total N times the multiply add unit will be enabled out of at least N_z cycles.

The latency is not so important in this case, but the throughput limits the performance.

For the second case, when the row length is equal to the number of inputs to the the adder tree, only the second condition is true. But still, resources are underutilized.

The third case, where the row length is larger then the width of the adder tree could result in low performance results. In this specific case, the temporary row results need to be feed back back into the pipeline. The total latency to finish a row equals: $init_{time} + n_s * L_{add} * \lceil \frac{r_l}{2^{n_s}} \rceil$. A lot of cycles can be lost if the row lenghts are very long. New rows can not be forwarded to the pipeline, since data from the current row still needs to be processed. Our approach has no or minimal losses from large row lenghts, except for some high initialization, equal to the init_time.

For the first and second case, where the row size is equal or smaller to the numbers of input of the first stage of the adder, we expect similar performances, but our design utilizes the hardware much better. For the third case, our design can finish executions much faster, since we do not have to wait for the whole adder tree to be finished.

Lets provide an example. Assuming we have an adder tree based on 3 stages ($n_s = 3$) with 8 inputs for its first stage, a matrix with $N = 10000$, $N_z = 80000$ and that for each row $r_l = 8$. We assume also that the adder latency L_{add} equals 8 cycles. The total latency for the design based on the adder tree consist of the following contributions:

- $N * r_l = N_z = 80000$ cycles, to fill all the data into the pipelines
- $n_s * L_{add} = 24$ cycles, the latency to compute the last row
- We assume 8 cycles latency for the multiplier.

The total cycles equals 80032 cycles for the adder tree based design. In our design we have the following costs:

- $8 * r_l = 64$ cycles, to initialize in the first 8 slots.
- $N_z = 80000$ cycles, for all the computations. However, some of the computations are overlapped with the initialization of the first 8 slots. After the first 8 cycles of initializing the first row, the row is able to start already its computations. After

the first 16 cycles, when both row 1 and 2 are initialized, rows 1 and 2 can perform some computations. So for the 64 initialization cycles, 1+2+3+4+5+6+7+8 = 36 cycles are actualy already used for computation.

- 3 cycles latency for the multiplier.

The total cost equals 80031 cycles. The difference with the adder-tree is more or less neglible. The reason why our design has only 3 cycles for the multiplier is, is that its integrated with the adder unit. The total latency of this unit is 11 cycles, as explained in section 4.2.3.

Lets take now the same example but with $N_z = 90000$ and that for each row $r_l = 9$. The cost for our design is 8*9+90000-44+3 = 90031 cycles. Note here that the number 44 is obtained by the addition of 1+2+3+4+5+6+7+8+8, the last number is 8, since we have 8 slots only.

However the cost for the adder tree is much higher and consist of:

- $9 + n_s * L_{add} = 33$ cycles latency per row to fill data to the pipeline. Therefore, 330000 cycles to load all data to the pipeline.
- $n_s * L_{add} = 24$ cycles, the latency to compute the last row
- 8 cycles latency for the multiplier.

The total cost equals 330032 cycles for the adder tree design, while in our design this is only 90031 cycles. The problem with the adder tree is, that it stays long time idle when the row lenght is larger then the number of its inputs. Actually, its possible to already start a new row when you have to wait for the result for the current row, but this makes the controller very complicated.

A second approach to use the adder tree is depicted in Figure 5.6. Here, more memory banks are placed to feed the pipelines fasters, to utilize a higher resource utilization. However, this requires the matrix elements to be splitted among different memory banks, which is not very practical.

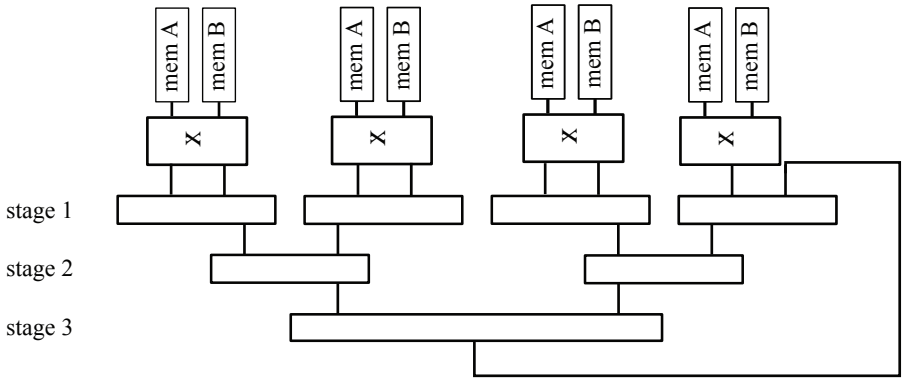

Figure 5.6: Floating Point Adder-tree with 3 stages, with high bandwidth

5.5 Design comparison

The presented design for the SMVM kernel in the previous chapter is very efficient and does not require huge adder trees. The performance efficiency for the presented design is close to peak performance if the matrix input sizes are reasonable large. For the closest work, the differences of the designs will be compared in depth. First we analyze design presented by L. Zhuo and V. Prasanna in [29]. They report a peak performance of 1.6 GFLOPS when the bandwidth equals 8 GB/s, and a performance of 2.88 GB/s when the bandwidth equals 14.4 GB/s. This bandwidth is the input bandwidth only for the matrix. The output bandwidth for this design is not included in their analysis. It is likely that the authors obtained these numbers by dividing the number of operations by the total load cost. The number of operations equals $2N_z$. Similar, as in Table 5.2, the total load cost equals $\frac{5}{4}N_z + \frac{5}{4}N$. the difference with our design is that they do not included the initial value **c**, they use 16 bits wide addresses and they load the vector **b** locally first. The performance for their design can be written as:

$$P = \frac{2N_z}{t_{load}} = \frac{2N_zB}{\frac{5}{4}N_z + \frac{5}{4}N} \leq 1.6B \tag{5.16}$$

Here, they made the assumption that $N_z >> N$ and the bandwidth is specified in words/s. This assumption does not hold when the matrices are very sparse. Considering a bandwidth of 8 GB/s (1 GWords/cycle) the 1.6 GFLOPS is obtained. The computation time is discarded since its overlapped with communication. However, the authors do not include the cost of the output bandwidth, since its negligible (given $N_z >> N$) compared to the bandwidth required to load the matrix A . The output bandwidth for the design depends on the number of elements and is at least higher compared our presented design. In [29] temporary results are streamed out, while in our design only final row results are streamed out. Comparing the input bandwidth only and assuming for both designs a peak performance of 1.6 GFLOPS, the design in [29] requires much less bandwidth, 8 GB/s versus 19.2 GB/s in our design. Our design always performs near optimal performances and for the design in [29] this depends on the type of matrix. Considering an input bandwidth of 8 GB/s, their performance is between 20%and 78% for the test matrices they presented. If we compare their design with a more fairly equal bandwidth: 9.2 GB/s using 4 PEs for our design and achieving a performance around 800 MFLOPS. We still outperform their design in case of 7 out of the 11 cases they presented. Their design reports lower performance results for sparser matrices, this can be explained by the adder trees that they use which must be filled with zero if no data is available. The sparsest matrix contained a nonzero density of 0.04% which is much higher then the matrices we target. The results for sparser matrices will lead to further lower performances in [29].

The second design which we want to compare is the design in [24]. It will be harder to compare with this design, since the authors did not include any absolute performance numbers. The authors take a similar approach as our design by dividing the matrix in horizontal stripes. They divide the horizontal stripes further into vertical stripes to reduce the address width to 16 bits. The vertical splitting allows to reduce the bandwidth. The matrices that they benchmarked reported peak performances between

86% for a density of non-zeros of 0.01% and a peak performance of 98% when the density is 0.5%. The reduction circuit in this design is improved compared to the design in [29] since they reduce the number of adders from 16 to 13 for each processing element. In our design no reduction circuit is used. In their paper they claim that they report similar performances as in the design of L. Zhuo and V. Prasanna [29].

Comparing our design with already available literature, the conclusion can be made that the design requires more bandwidth compared to other designs. In the next section we identify the causes for this more specifically and how to improve it. However, the design reports higher performances compared to similar bandwidths for extremely sparse matrices.

5.5.1 Future design improvements

To improve the design, the latency of the circuit must be improved as much as possible. Although the effect on large matrices is minimal, submatrices due to sectioning of the original matrix might be influenced more. The reasons why there is more bandwidth required for our design are the following:

1. We assume the worst case load, which is for example a diagonal matrix times a vector. This requires for each processing element a load operation from the row array. The matrices that we consider are so small that the inequality of the right hand side of (5.16) does not hold.

2. Vector **b** is considered Off-Chip. This requires a lot extra number of loads depending on the density of the matrix. Currently, the number of loads for matrix **b** equals N_z while the minimal could be N.

3. Currently, no vertical sectioning is implemented. Therefore, reducing the address widths to 16 bits is not possible, unless small matrices are considered.

If all the 3 issues are included in the design, the bandwidth can be reduced by a factor of: $\frac{\frac{5}{2}N_z+\frac{5}{2}N}{\frac{5}{4}N_z+\frac{13}{4}N} = \frac{10N_z+10N}{5N_z+13N}$. In case N_z equals N a factor decrease of 1.11 bandwidth can be achieved. In case $N_z >> N$ the bandwidth can be theoretically be reduced by a factor of 2. To improve the first issue, the memory which contains the row results can be shared with multiple processors. From typical matrices in the FVM method, in which the matrices contain at least 4 elements per row, 1 memory bank can be shared among 4 PEs, without lose of performance. The second and third issues can be improved by supporting vertical sectioning, next to row sectioning, similarly build in [24].

Another idea to reduce bandwidth is to consider huge On-Chip memory blocks which store vector **b**. By first streaming all the elements of the vector **b** prior to execution, a lot of bandwidth can be saved. However, additional cycles are wasted filling these matrices. Further the possibility exist to investigate whether cache is an option to store **b** in. Consecutive rows usually contain a similar pattern of non-zeros as in the previous rows.

5.6 Integrating the matrix vector multiply kernel into OpenFOAM

In this section, the hardware unit for the SMVM kernel will be integrated into the OpenFoam application. In Section 5.6.1, it is explained how OpenFOAM is connected to the hardware unit. Subsequently, Section 5.6.2 discusses the overall performance increase for the whole application.

5.6.1 Hardware connection to OpenFoam

To enable the hardware the user must have the ability to decide which CPU-nodes have an FPGA accelerator. This input can be provided in the new *fpgaDict* file located in the system folder as can be seen in Figure 2.2.

5.6.1.1 fpgaDict

An extra file is created to simplify the control to access the FPGAs. When SimpleFoam is executed, it reads out the content of this file. The user has the ability to specify for each kernel on each node for which kernel hardware is enabled.

An example file is shown in Figure 5.7. The first keyword which can be seen there is **fpga_enabled**. The values for **fpga_enabled** here must be *yes*, *y*, *no* or *n*. This is a global keyword and enables or disables all the FPGAs. The keyword **numberofProcs** tells how many CPU nodes are being used, this does not have to be the amount of available FPGAs. Next, the bitstream file names for each node is specified. If they are all equal, its sufficient to only put 1 element in the list. The rest of the files contain specific information for the kernels. In the current file only one kernel has been placed called matmul (Amul and residual kernels) and it is enabled for node 0 and node 2. The file could be extended by making a global list of CPU-nodes assuming they all have the same hardware kernels and we can give them the same bitstream names. Additional files have been included to the toolbox to support communication with the hardware platform, fpga_init.h is responsible for setting up the communication with the FPGAs on the specified nodes and fpga_close.h terminates the connections to the FPGAs.

Figure 5.8 shows the execution of the SimpleFOAM in software with new available hardware units. The application starts in the normal way, with additional procedures that initialize the hardware. If the hardware is not available or an error occurs during initialization, the hardware unit will be disabled. The kernels enabled for hardware will be executed on the hardware platform. When all the CPU nodes are finished in the computations, the connection to the FPGAs are terminated on the relevant nodes. Since node 0 and node 2 have hardware accelerators, these nodes can execute larger sparse matrix vector products in the same time for the nodes that do not contain an FPGA. In Section 5.6.2, we define the performance of the matrix vector products for the OpenFOAM application as 270 MFLOPS. Using 4 PEs, a speedup of a factor of 3 can be obtained. Since the design always reaches close to peak performance, nodes that contain an FPGA could have matrix sizes with 3 times more non-zeros, which approximates 3 times larger matrices, since the distribution of zeros per row is similar. With the Metis

```
// * * * * * * * * * * * * fpgaDict file:  hardware control * * * * * * *
* * * * //

fpga_enabled        y;
numberofProcs       4;

bit_stream_names
(
        sparsetest
        sparsetest2
);

matmul
{
                    enabled y;

                    cpu_nodes
                    (
                            0
                            2
                    );
};
// ********************************************************************* //
```

Figure 5.7: The *fpgaDict* file

decomposition process, exact weights can be specified to partition the original mesh in submeshes with desired sizes.

5.6.1.2 Problems with RASC-core services

A number of problems were encountered using and calling the SMVM kernel inside OpenFOAM. We were able to open and close the FPGAs on different nodes with MPI enabled. However, trying to execute the **matmul** kernel from inside OpenFOAM failed. Each time the matmul kernel was accessed in hardware the application crashed, while the kernel is working stand alone. With *strace*, a tool for tracing system calls and signals, the signals during runtime where intercepted and directed to the screen. At some point in time, once the hardware is being used 3 child threads. One of the child threads is abruptly terminated and kills it parent process as well. The comparison of the results of tracing the stand alone version and the version inside OpenFOAM did not resulted in any valuable knowledge. SGI provided some test examples that use the FPGA. Their example, alg6 crashes as well (for smaller sizes it works fine). We believe that the signal interventions between the OpenFOAM MPI library and RASC library are responsible for this problem.

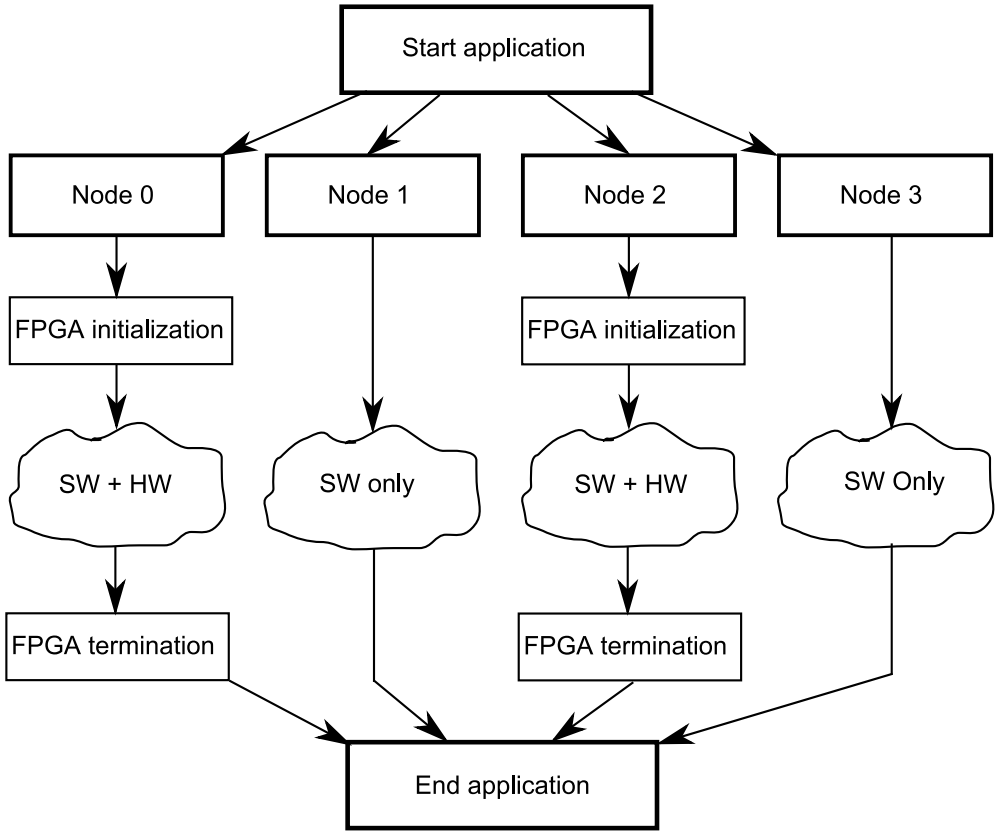

Figure 5.8: An example of how FPGAs are accessed on different CPU nodes using MPI

5.6.2 Applicational Speedup for the SimpleFoam solver using the hardware SMVM benefits from SMVM-kernel

The profiling results showed the sparse matrix vector multiplication contributed to 8% of the total execution time. According to Amdahls law: $\frac{1}{(1-p)+\frac{p}{N}}$, the speedup is limited by the serial part of the program. In the formula p presents the parallel fraction of the application, $1-p$ the serial part and N the number of processing nodes the program is accelerated over.

Considering a kernel speed up between 2.16 and 7.3 for 4 PEs and a parallel part p equal to 0.077 (sum of relative % of the residual and Amul kernels from the profiling report), according to Amdahls law the application speed up S is limited to: $1.043 \leq S \leq 1.071$. Thus, an overall application acceleration due to the SMVM between 4.3% and 7.1% can be expected. If we consider the Amul performance to be 210 MFLOPS and for the residual kernel 270 MFLOPS in Figure 5.3, the application speed up S will be: $\frac{1}{(1-p_0-p_1)+\frac{p_0}{N_0}+\frac{p_1}{N_1}} = \frac{1}{(1-0.075)+p0\frac{270}{800}+p1\frac{210}{800}}$=1.055. Table 5.14 provides the details of the exact fractions for each p_i and N_i. Table 5.15 shows the efficiency of application speed up, equal to 97.7%, compared to the theoretical limitation. When more PEs are considered, for example by using multiple FPGAs this efficiency can be increased more.

kernel name	i	p_i	N_i
Residual	0	0.041	800/270 = 2.96
Amul	1	0.036	800/210 = 3.81

Table 5.14: Fraction of the execution time of the SMVM kernels and the kernel speedup in hardware.

	Estimated speedup	Theoretical limitation	Efficiency
formula	S = $\frac{1}{(1-\sum_i p_i)+\sum_i \frac{p_i}{N}}$	T= $\frac{1}{1-\sum_i p_i}$	$\frac{S}{T} * 100\%$
value	1.055	1.08	97.7%

Table 5.15: Efficiency of the application speedup S relative to the theoretical limit

5.7 Conclusion

In this chapter, analysis for the hardware version of the SMVM kernel are presented. The design can be build out of multiple units scalable with the available bandwidth. Moreover, the design is integrated with the dense vector multiplication in [25] which reports the highest performance for dense sparse matrix vector multiplication. The processing elements are very area efficient by adding the accumulations of rows in time in stead of using adder trees. Besides software simulations, the design has been verified by automated test generations of over 1000 test matrices in hardware for 1 and 2 PE elements. Test results show that the performance reaches near peak performance as predicted with the formulas. However, the unit as it currently is requires too much bandwidth compared to other designs, since the elements of the vector **b** are loaded multiple times from main memory. Considering 4 PEs a speedup between 2.16 and 7.3 is achieved. Considering 8 PEs, simulation show a maximal speedup of 14.6 compared to the Itanium 2 processor. An application speed up can be reached between 4.3% and 7.1% due to the SMVM kernel only, using 4 PEs. If we estimat

6 Conclusions

This chapter presents the conclusions that could be drawn from the performed research. Section 6.1 presents a summary of this thesis, organized per chapter. The objectives and whether they have been met for this thesis are discussed in Section 6.2. Finally, recommendations for future work are proposed in Section 6.3.

6.1 Summary

In this thesis, a design for the double precision IEEE-754 Floating Point Sparse Matrix dense Vector Multiplication (SMVM) has been proposed. This kernel is not limited by assumptions on the input format and is designed for the Compressed Row Storage format. The kernel is targeted at the Silicon Graphics Inc. Altix 450 machine RASC-Core which contains a high throughput low-latency connection between traditional supercomputers and reconfigurable hardware. The Altix machine consist of multiple high performance GPPs, namely the Itanium 2 9130m .

In Chapter 2, various aspects of the OpenFOAM CFD tool were described. In particular, the focus was on the SimpleFoam solver, which is one of the many solvers OpenFOAM supports. The SimpleFoam application is based on the Navier-Stokes equations, which describe motion of fluids in space and time. The mesh, representing the computational grid, is read in through files into the CFD tool, along with the properties of the grid solvers, the descritized mathematical operations and several timing options. These input files presented in this chapter formed the basis for the profiling. Four different decomposition methods have been described that support solving grids using multiple CPU nodes, to decrease the modeling time. The second part of this chapter described the profiling of SimpleFoam. The best way to profile applications on the Itanium processor is to use specific additional hardware instrumentation for profiling, like Instruction Point (or Program Counter) sampling. The SimpleFoam solver has been profiled on 1, 2 and 4 CPUs with a mesh containing over 6 million cells, which is considered a realistic input. Two different decomposition methods have been analyzed and compared. The Metis decomposition method was found to be superior over the Simple method, since it was able to reduce the communication cost and lead to faster execution times. In this chapter the most important kernels have been identified and a small description of the key kernels was given.

In Chapter 3, the Openfoam Sparse Matrix Vector Multiplication (SMVM) kernel was extensively analyzed. The kernel seemed not appropriate for parallel extension and therefore different formats were analyzed. The criteria to compare them were scalability with increasing bandwidth, the number of memory accesses and the required storage space. The MSR was found to be the best. A conversion algorithm in O(n) time between the OpenFOAM format and the MSR format was also given.

In Chapter 4, the design methodology was described developing the SMVM hardware unit. The platform and architectural choices have been discussed. The area cost of the Processing Elements of the presented SMVM design is low. Last, the Processing Elements of the presented SMVM unit and the PE of the dense design in [25] have been integrated together.

In Chapter 5 analysis for the hardware version of the SMVM kernel were presented. The design contained multiple processing units scalable with the available bandwidth. More over, the design was integrated with the dense vector multiplication in [25] which reports the highest performance for dense sparse matrix vector multiplication. The processing elements are very area efficient by adding the accumulations of rows in time in stead of using adder trees. Besides software simulations, the design has been verified by automated random test generations using over 1000 test matrices in hardware for 1 and 2 PE elements. Test results showed that the performance reaches near peak performance, which verifies the presented formulas. However, the unit as it currently is, requires too much bandwidth compared to other designs, since the elements of the vector **b** are loaded multiple times from main memory. Considering 8 PEs, simulations suggest a maximal speedup of 14.6 compared to the Itanium 2 processor. An application speed up can be reached between 4.3% and 7.3% due to the SMVM kernel only, if 4 PEs are used. If we estimate the performance for the Amul kernel as 210 MFLOPS and for the residual kernel 270 MFLOPS an application speed up of approximately 5.5% can be obtained using four PEs only.

6.2 Objectives Coverage

The main thesis goals defined in Section 1.2.2 were:

1. Profile the SimpleFoam solver and identify the time critical kernels.
2. Design hardware CCU units supporting these kernels.
3. Integrate the CCU kernels into the SGI RASC system and accelerate OpenFOAM.
4. Integrate the sparse matrix vector product, being one of the identified critical kernels, with the dense matrix vector product implementations proposed in [25].

Objective 1: This objective was fully achieved. Profiling with the *histx* script developed by SGI identified the kernels of the OpenFOAM application for the SimpleFOAM solver accurately.
Objective 2: Currently, only a hardware unit is developed for the sparse matrix vector product. Different matrix formats were investigated and the best format found was the MSR format. Yet, the hardware unit supports the CRS format only, and this must be extended to the MRS format. The difference between the two formats its the storage of the main diagonal. The hardware unit performs close to peak performance for each CCU if reasonable input matrix sizes are considered.
Objective 3: This objective was not completely met. Although the stand alone version for the SMVM kernel is working in hardware, connecting the SMVM to OpenFOAM

failed due to problems mentioned in Section 5.6.1, very likely due to signal interference between OpenFOAM and the RASC-core unit.
Objective 4: For the last objective, we modified the dense PE in such a way that its also capable of computing the sparse matrix vector multiplication. Simulations verified the PEs working for both sparse and dense matrix vector multiplications. The unit is also capable of computing dense by dense matrices.

6.3 Future work

The following proposes some future research directions. The recommended future work for this thesis is divided in several sections.

Sparse Matrix Vector Multiplication:

- Considering the MSR format, besides the CRS format. This format can reduce the bandwidth. The MSR format stores the main diagonal separately and there for no column coordinates have to be loaded to obtain its column coordinate. In OpenFOAM, the diagonal is assumed to contain non-zero values and the MSR could lead to better performance results.

- Extending the SMVM design from two to four PEs and using more FPGAs. In hardware currently only two Processing Elements are included into the design. Due to routing issues a lot of effort is spend on routing these two PEs. By increasing the number of PEs higher performances can be achieved.

- The most important thing to be improved for the design is reducing the current bandwidth. If the matrix is sliced in vertical slices, the vector **b** can be stored in local memory, which allows using 16 bit wide addresses. The most bandwidth can be saved due to accesses to vector **b**.

Mixed Dense and Sparse Matrix Vector Multiplication:

- Improving and reducing the are cost further for this design. Area can be reduced if the register that stores the element of **b** in the dense PE is stored in the memory for the sparse PE at index zero. Further, memory **c** for both the sparse and dense design can be shared. This can also improve the frequency of the combined PE.

- Hardware verification of the design. So far, only software simulations in ModelSim have verified correct working.

OpenFOAM:

- Currently, problems occurred while connecting OpenFOAM to the RASC Core Services. The problems in the SGI design environment must be identified more specifically and a solution must be found for this.

- Integrating other kernels like the smooth function into hardware. The Gauss-Seidel smoother is the next important kernel for this application. It is already written in C-style coding and mapping the functions to hardware is straight forward.
- Due to problems with the RASC-core, only simulation performances are included in the analysis for application speed up. The impact and behavior of utilizing FPGAs for several computing nodes, but not all nodes, must be still investigated.

Bibliography

[1] Actiflow, www.actiflow.nl.

[2] F. Ambrosini, S. Raia, A. Funel, S. Podda, and S. Migliori, *Cross checking openfoam and fluent results of cfd simulations in enea-grid environment*, `http://www.cresco.enea.it/`.

[3] Timothy A. Davis, *University of florida sparse matrix collection*, NA Digest **92** (1994).

[4] M. deLorimier and A. DeHon, *Floating-point sparse matrix-vector multiply for fpgas*, Internation Symposium on Field-Programmable Gate Arrays, 2005.

[5] Yong Dou, S. Vassiliadis, G. K. Kuzmanov, and G. N. Gaydadjiev, *64-bit floating-point fpga matrix multiplication*, In ACM/SIGDA Field-Programmable Gate Arrays, ACM Press, 2005, pp. 86–95.

[6] Yousef El-Kurdi, Warren J. Gross, and Dennis, *Hardware acceleration for finite element electromagnetics: Efficient sparse matrix floating-point computations with field programmable gate arrays*, 2006.

[7] Yousef El-Kurdi, Warren J. Gross, and Dennis Giannacopoulos, *Sparse matrix-vector multiplication for finite element method matrices on FPGAs*, in *FCCM* [6], pp. 293–294.

[8] Lenwood S. Heath, Sriram V. Pemmaraju, and Calvin J. Ribbens, *Sparse matrix-vector multiplication on a small linear array*, Tech. report, 1993.

[9] Eun-Jin Im and Katherine Yelick, *Optimizing sparse matrix computations for register reuse in SPARSITY*, Lecture Notes in Computer Science **2073** (2001), 127–??

[10] Cray Inc., `www.cray.com`.

[11] Silicon Graphics Inc, *Sgi altix 450 system user's guide*, 2007.

[12] Silicon Graphics Inc., *Linux application tuning guide*, `http://techpubs.sgi.com/library/manuals/`, 2008.

[13] Silicon Graphics Inc, *Reconfigurable application specific computer user's guide*, `http://techpubs.sgi.com/library/`, 2008.

[14] Xilinx Inc., `www.xilinx.com`.

[15] Intel, *Intel math kernel library*, `www.intel.com`.

[16] H. Jasak, *Polyhedral mesh handling in openfoam*, `http://powerlab.fsb.hr/ped/kturbo/OpenFOAM/WorkshopZagrebJan2006/`, 2007.

[17] George Karypis and Vipin Kumar, *Methis, a software package for partitioning unstructured graphs, partitioning of meshes, and computing fill-reducing orderings of sparse matrices. version 4.0*, 2008.

[18] OpenCFD Ltd, *Openfoam programmer's guide version 1.4*, `http://foam.sourceforge.net/doc/Guides-a4/ProgrammersGuide.pdf`, 2007.

[19] OpenCFD Ltd., *Openfoam userguide version 1.4*, `http://foam.sourceforge.net/doc/Guides-a4/UserGuide.pdf`, 2007.

[20] Giovanni Manzini, *Lower bounds for sparse matrix vector multiplication on hypercubic networks*, Discrete Mathematics and Theoretical Computer Science, 1998.

[21] Ali Pnar and Michael T. Heath, *Improving performance of sparse matrix-vector multiplication*, In Proceedings of Supercomputing, 1999.

[22] Y. Saad, *Sparsekit: A basic tool kit for sparse matrix computations*, `www-users.cs.umn.edu/ saad/software/SPARSKIT/sparskit.html`, 1994.

[23] P.T. Stathis, *Sparse matrix vector processing formats*, PhD in Computer Engineering, Delft University of Technology, 2004.

[24] J. Sun, Gregory Peterson, and Olfa Storaasli, *Mapping sparse matrix-vector multiplication on fpgas*, In Proceedings of Supercomputing, 1999.

[25] W. van Oijen, *Implementation of a polymorphic floating-point matrix multiplication unit*, MSc in Computer Engineering, Delft University of Technology, 2007.

[26] Richard Vuduc, James W. Demmel, and Katherine A. Yelick, *Oski: A library of automatically tuned sparse matrix kernels*, J. Phys.: Conf. Ser. 16 (2005), 521–530.

[27] Gerhard Wellein, Georg Hager, Achim Basermann, and Holger Fehske, *Fast sparse matrix-vector multiplication for teraflop/s computers*, Springer Berlin / Heidelberg, 2003.

[28] Ling Zhuo and V. K. Prasanna, *Scalable and modular algorithms for floating-point matrix multiplication on fpgas*, In Proc. of The 18th International Parallel & Distributed Processing Symposium, 2004.

[29] Ling Zhuo and Viktor K. Prasanna, *Sparse matrix-vector multiplication on fpgas*, FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays (New York, NY, USA), ACM, 2005, pp. 63–74.

Abbreviations

ASIC	-	Application Specific Integrated Circuit
ADR	-	Algorithmic Defined Register
BBCS	-	Block Based Compression Storage
BSR	-	Block Sparse Row
CCU	-	Customized Computing Unit
CDS	-	Compressed Diagonal Storage
CRS	-	Compressed Row Storage
COO	-	Coordinate Format
CPU	-	Central Processing Unit
CFD	-	Computation Fluid Dynamics
DIA	-	Diagonal Format
DCM	-	Digital Clock Manager
DMA	-	Direct Memory Access
DMVM	-	Dense Matrix Vector Multiplication
DSF	-	Diagonal Storage Format
FEM	-	Finite Element Method
FIFO	-	First In First Out
FSB	-	Front Side Bus
FLOPS	-	Floating Point Operations per Second
FSM	-	Finate State Machine
FPGA	-	Field Programmable Gate Array
FVM	-	Finite Volume Method
GFLOPS	-	Giga Floating Point Operations per Second
GPP	-	General Purpose Processor
JDS	-	Jagged Diagonal Storage
MRS	-	Modified Row Sparse
MFLOPS	-	Mega Floating Point Operations per Second
OpenFOAM	-	Open Field Operation and Manipulation
RASC	-	Reconfigurable Application Specific Computing
SMVM	-	Sparse Matrix Vector Multiplication
SGI	-	Silicon Graphics Inc
VHDL	-	VHSIC Hardware Description Language
VHSIC	-	Very High Speed Integrated Circuit

SimpleFoam code

A

A.1 Imported headers

setrootcase: It verifies the arguments being parsed from the command line (MORE!).
createTime: This creates a time object called runTime, it keeps track of all timing information, e.g collapsed CPU Time, start time, stop time etc.
createMesh: Creates the mesh
createFields: Creates the pressure p, velocity U, the flux ϕ and constructs the turbulence and transport models
initContinuityErrs: Defines and initialize the scalar cumulativeContErr to 0.
continuityErrs.H: Calculates and prints the continuity errors.

A.2 Body code

Line 1: The scalar field p of the previous iteration must be stored, since it's needed for under-relaxation. Under-relaxation controls the changes of certain variables due to non-linearity. Given a under-relaxation factor α, and a general variable γ, the under-relaxation control can be described by

$$\gamma = \gamma_{\text{old}} + \alpha * \Delta\gamma$$

where $0 < \alpha \leq 1$ and γ could be the pressure, velocity, speed or any of the under-relaxation factors specified in *fvSolution* file.

line 2: The prediction equation for the momentum is set up. line 3: Under-relaxation adjustment for the speed line 4: Solve the momentum predictor

$$\nabla \cdot \mathbf{U} + \nabla \cdot \mathbf{R} = -\nabla p \tag{A.1}$$
$$\mathbf{R} = \nu_{eff} \nabla \mathbf{U} \tag{A.2}$$

line 4: Update the boundary conditions for p
The rest of the code computes the flux:

$$\phi = \mathbf{S} \cdot \mathbf{U}_f = \mathbf{S} \cdot \left\{ (\frac{\mathbf{H}(\mathbf{U})}{a_p})_f - \frac{(\nabla p)_f}{(a_p)_f} \right\} \tag{A.3}$$

Profiling Commands and Results

B

B.1 Commands

To run the SimpleFoam solver on 1 CPU the following command must be used: simpleFoam . <casename>
To profile the application using one CPU with the histx profiler, the following command can be used:

> histx -e timer@N -k -f -o <pattern> simpleFoam . case04m | tee <filedump>

Here histx takes the following arguments:

- -e, this option tells histx which source of events to use. A sample is recorded each time N additional user+system ticks have been accumulated.
- -k, tells histx to also record samples when running in the kernel at privilege level 0. Off by default. This option only has meaning when using the Itanium PMU event source.
- -f, tells histx to produce a report for any processes fork()ed from the initial process. This is required for profiling when multiple processing nodes are involved.
- -o, tells histx to write it's per-thread report to a file with path given by <pattern>.
- with the tee command everything is dumped to a the file <filedump>.

The commands to start the profiling on 2 CPUs is as follows: mpirun -np 2 histx -e timer@2 -k -f -o filename.log simpleFoam . case04m -parallel | tee screen.log

- here simpleFoam takes an extra argument parallel, to instruct the program telling that the mesh is being executed on multiple nodes
- np is an argument of mpirun and denotes the number of processors used.

By default, the histx report is written to a file named pattern.< *command* >.<PID>, in this case filename.log.simpleFoam.<PID>. Since 2 processors are used 2 files are created. The next step is to use iprep to create a useful report from the raw ip sampling reports produced by histx.

> iprep filename.log.simpleFoam.< PID > | c++filt | tee <filedump>

The c++filt command demanges the C++ symbols from the report and the profiling report is stored in filedump.

B.2 OpenFOAM functions

These functions here are the functions that are being identified as kernels by the profiler.

smooth src/OpenFOAM/matrices/lduMatrix/smoothers/GaussSeidel/GaussSeidelSmoother.C
$libOpenFOAM.so : Foam :: GaussSeidelSmoother :: smooth(Foam :: wordconst\&, Foam :: Field < double > \&, Foam :: lduMatrixconst\&, Foam :: Field < double > const\&, Foam :: FieldField < Foam :: Field, double > const\&, Foam :: UPtrList < Foam :: lduInterfaceFieldconst > const\&, unsignedchar, int)$

grad_1 /src/finiteVolume/finiteVolume/gradSchemes/limitedGradSchemes/cellLimitedGrad/cellLimitedGrad.C
$libfiniteVolume.so : Foam :: fv :: cellLimitedGrad < Foam :: Vector < double >>:: grad(Foam :: GeometricField < Foam :: Vector < double >, Foam :: fvPatchField, Foam :: volMesh > const\&)const$

limitFace src/finiteVolume/finiteVolume/gradSchemes/limitedGradSchemes/cellLimitedGrad/cellLimitedGrad.C
$libfiniteVolume.so : Foam :: fv :: cellLimitedGrad < Foam :: Vector < double >>:: limitFace(Foam :: Vector < double > \&, Foam :: Vector < double > const\&, Foam :: Vector < double > const\&, Foam :: Vector < double > const\&)$

grad_2 src/finiteVolume/finiteVolume/gradSchemes/limitedGradSchemes/cellLimitedGrad/cellLimitedGrad.C
$libfiniteVolume.so : Foam :: fv :: cellLimitedGrad < double >:: grad(Foam :: GeometricField < double, Foam :: fvPatchField, Foam :: volMesh > const\&)const$

residual /src/OpenFOAM/matrices/lduMatrix/lduMatrix/lduMatrixATmul.C
$libOpenFOAM.so : Foam :: lduMatrix :: residual(Foam :: Field < double > \&, Foam :: Field < double > const\&, Foam :: Field < double > const\&, Foam :: FieldField < Foam :: Field, double > const\&, Foam :: UPtrList < Foam :: lduInterfaceFieldconst > const\&, unsignedchar)const$

Amul /src/OpenFOAM/matrices/lduMatrix/lduMatrix/lduMatrixATmul.C
$libOpenFOAM.so : Foam :: lduMatrix :: Amul(Foam :: Field < double > \&, Foam :: tmp < Foam :: Field < double >> const\&, Foam :: FieldField < Foam :: Field, double > const\&, Foam :: UPtrList < Foam :: lduInterfaceFieldconst > const\&, unsignedchar)const$

grad_3 src/finiteVolume/finiteVolume/gradSchemes/gaussGrad/gaussGrad.C
$libfiniteVolume.so : Foam :: fv :: gaussGrad < double >:: grad(Foam :: GeometricField < double, Foam :: fvsPatchField, Foam :: surfaceMesh > const\&)$

grad_4 src/finiteVolume/finiteVolume/gradSchemes/gaussGrad/gaussGrad.C
$libfiniteVolume.so : Foam :: fv :: gaussGrad < Foam :: Vector < double >>:: grad(Foam :: GeometricField < Foam :: Vector < double >, Foam :: fvsPatchField, Foam :: surfaceMesh > const\&)$

MPI
$libmpi.so : MPI\_SGI\_shared\_progress$

interpolate src/finiteVolume/interpolation/surfaceInterpolation/surfaceInterpolationScheme/surfaceInterpolationScheme.C
$a.out : Foam :: surfaceInterpolationScheme < Foam :: Vector < double >>:: interpolate(Foam :: GeometricField < Foam :: Vector < double >, Foam :: fvPatchField, Foam :: volMesh > const&, Foam :: tmp < Foam :: GeometricField < double, Foam :: fvsPatchField, Foam :: surfaceMesh >> const&)$

B.3 Profiling Results

The application is profiled on 1, 2 and 4 CPUs. The results are summarized in the tables B.1 B.2 and B.3. In each table, depending on the number of CPUs used, we record information on the host and all the slaves. For the most important kernels, the CPU % is reported, sorted from high to low. Further, the cumulative CPU usage is reported, which is the sum of the total CPU usage for the considered kernels.

1 CPU. No Decomposition

Function	CPU usage%	cum. CPU usage %
smooth	23.111	23.111
grad_1	8.706	31.817
limitFace	6.385	38.202
grad_2	4.472	42.674
residual	3.877	46.550
Amul	3.479	50.029

2 CPU's. Methis Decomposition

Function	Host CPU usage%	Host cum. CPU usage%	Slave CPU usage%	Slave cum. CPU usage %
smooth	19.287	19.287	20.144	20.144
grad_1	9.272	28.559	9.941	30.086
limitFace	6.352	34.911	6.741	36.827
grad_2	4.064	38.975	4.346	41.173
grad_3	3.682	42.657	3.809	44.982
residual	3.649	46.307	3.742	48.724
Amul	3.496	49.803	3.587	52.311

2 CPU's. Simple Decomposition

Function	Host CPU usage%	Host cum. CPU usage%	Slave CPU usage%	Slave cum. CPU usage %
smooth	19.260	19.260	21.166	21.166
grad_1	8.804	28.064	9.677	30.793
limitFace	6.099	34.163	6.566	37.360
grad_2	3.837	38.000	4.381	41.741
grad_3	3.554	41.554	3.750	45.491
Amul	3.523	45.077	3.714	49.206
residual	3.234	48.311	3.434	52.639

Table B.1: Profiling using 1 and 2 CPUs, both with Metis and simple decomposition

4 CPU's. Methis Decomposition		
Function	**Host CPU usage%**	**Host cum. CPU usage%**
smooth	13.093	13.093
grad_1	6.826	19.919
MPI	5.111	25.031
limitFace	4.440	29.470
residual	4.097	33.567
grad_3	4.045	37.612
grad_4	3.522	41.134
Amul	3.490	44.624
Function	**Slave 1 CPU usage%**	**Slave 1 cum. CPU usage%**
smooth	13.134	13.134
grad_1	7.849	20.983
limitFace	4.848	25.830
residual	4.170	30.000
grad_3	4.104	34.105
grad_4	3.596	37.700
Amul	3.529	41.230
MPI	3.499	44.729
Function	**Slave 2 CPU usage%**	**Slave 2 cum. CPU usage%**
smooth	13.100	13.100
grad_1	7.380	20.481
limitFace	4.672	25.153
grad_3	4.272	29.425
residual	4.103	33.528
MPI	3.840	37.368
grad_4	3.560	40.928
Amul	3.520	44.448
Function	**Slave 3 CPU usage%**	**Slave 3 cum. CPU usage%**
smooth	13.211	13.211
grad_1	7.429	20.703
limitFace	4.648	25.388
grad_3	4.247	29.634
residual	4.172	33.807
grad_4	3.538	37.344
Amul	3.518	40.862
MPI	3.221	44.083

Table B.2: Profiling results, for using 4 CPUs. Metis decomposition

4 CPU's. Simple Decomposition		
Function	**Host CPU usage%**	**Host cum. CPU usage%**
smooth	13.341	13.341
grad_1	7.585	20.926
limitFace	4.635	25.561
grad_3	4.145	29.706
residual	3.985	33.691
grad_4	3.592	37.283
Amul	3.529	40.812
interpolate	3.099	43.911
Function	**Slave 1 CPU usage%**	**Slave 1 cum. CPU usage%**
smooth	14.122	14.122
grad_1	8.920	23.042
limitFace	5.275	28.318
residual	4.145	32.463
grad_3	4.130	36.593
Amul	3.712	40.305
grad_4	3.607	43.912
interpolate	3.501	47.413
Function	**Slave 2 CPU usage%**	**Slave 2 cum. CPU usage%**
smooth	13.712	13.712
grad_1	7.614	21.325
limitFace	4.782	26.108
grad_3	4.349	30.456
residual	4.135	34.591
grad_4	3.663	38.255
Amul	3.634	41.889
MPI	3.267	45.156
Function	**Slave 3 CPU usage%**	**Slave 3 cum. CPU usage%**
smooth	13.259	13.259
grad_1	8.459	21.718
limitFace	5.099	26.817
grad_3	4.244	31.061
residual	4.089	35.150
grad_4	3.720	38.870
Amul	3.567	42.446
interpolate	3.331	45.777

Table B.3: Profiling results, for using 4 CPUs. Simple decomposition

C Decomposition reports

This appendix chapter contains the decomposition reports for the *simple* and *Metis* decomposition methods for targetting 4 processors.

C.1 Metis report

```
/*---------------------------------------------------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  1.4.1                                 |
|   \\  /    A nd           | Web:      http://www.openfoam.org               |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/

Exec   : decomposePar . case04m
Date   : Aug 26 2008
Time   : 14:54:21
Host   : smaug
PID    : 25788
Root   : /opt/data/homes/mtaouil
Case   : case04m
Nprocs : 1
Create time

Time = 0
Create mesh


Calculating distribution of cells
Selecting decompositionMethod metis


Finished decomposition in 32.564 s

Calculating original mesh data

Distributing cells to processors
```

```
Distributing faces to processors

Calculating processor boundary addressing

Distributing points to processors

Constructing processor meshes

Processor 0
    Number of cells = 1715464
    Number of faces shared with processor 2 = 6885
    Number of faces shared with processor 1 = 5946
    Number of faces shared with processor 3 = 549
    Number of processor patches = 3
    Number of processor faces = 13380
    Number of boundary faces = 82738

Processor 1
    Number of cells = 1713283
    Number of faces shared with processor 0 = 5946
    Number of processor patches = 1
    Number of processor faces = 5946
    Number of boundary faces = 90936

Processor 2
    Number of cells = 1714303
    Number of faces shared with processor 0 = 6885
    Number of faces shared with processor 3 = 6981
    Number of processor patches = 2
    Number of processor faces = 13866
    Number of boundary faces = 83589

Processor 3
    Number of cells = 1715374
    Number of faces shared with processor 2 = 6981
    Number of faces shared with processor 0 = 549
    Number of processor patches = 2
    Number of processor faces = 7530
    Number of boundary faces = 85262

Number of processor faces = 20361
Max number of processor patches = 3
Max number of faces between processors = 13866
```

```
Processor 0: field transfer
Processor 1: field transfer
Processor 2: field transfer
Processor 3: field transfer

End.
```

C.2 Simple report

```
Exec   : decomposePar . case04m
Date   : Aug 27 2008
Time   : 13:26:59
Host   : smaug
PID    : 29331
Root   : /opt/data/homes/mtaouil
Case   : case04m
Nprocs : 1
Create time

Time = 0
Create mesh


Calculating distribution of cells
Selecting decompositionMethod simple

Finished decomposition in 32.672 s

Calculating original mesh data

Distributing cells to processors

Distributing faces to processors

Calculating processor boundary addressing

Distributing points to processors

Constructing processor meshes

Processor 0
    Number of cells = 1689323
    Number of faces shared with processor 2 = 6870
```

```
    Number of faces shared with processor 1 = 12659
    Number of faces shared with processor 3 = 15
    Number of processor patches = 3
    Number of processor faces = 19544
    Number of boundary faces = 81187

Processor 1
    Number of cells = 1739889
    Number of faces shared with processor 0 = 12659
    Number of faces shared with processor 3 = 1172
    Number of processor patches = 2
    Number of processor faces = 13831
    Number of boundary faces = 92545

Processor 2
    Number of cells = 1739889
    Number of faces shared with processor 3 = 14206
    Number of faces shared with processor 0 = 6870
    Number of processor patches = 2
    Number of processor faces = 21076
    Number of boundary faces = 79193

Processor 3
    Number of cells = 1689323
    Number of faces shared with processor 2 = 14206
    Number of faces shared with processor 1 = 1172
    Number of faces shared with processor 0 = 15
    Number of processor patches = 3
    Number of processor faces = 15393
    Number of boundary faces = 89600

Number of processor faces = 34922
Max number of processor patches = 3
Max number of faces between processors = 21076

Processor 0: field transfer
Processor 1: field transfer
Processor 2: field transfer
Processor 3: field transfer

End.
```

Problems, Bugs and other issues using the RASC Core Services

D

The latest RASC release, RASC 2.20 still contains bugs and other issues. In this chapter the problems faced during this thesis using the RASC-core are being described here.

D.1 Error in Configuration Tool script

SGI provided a Configuration Tool in which the user can specify which services are being used from the Core Services. Using this script (alg_def_tool.tcl) to generate 5 SRAMS results in an error in the ngdbuild phase. The error is probably due to a bug in ISE tool and is dependable of the order the design files are being compiled. To fix this, the alg_def_tool.tcl script must be replaced. SGI is already aware of this error and a fix is available for it (but officially not yet released).

D.2 Improper functionality of fpgastep of gbdfpga

The fpgastep for debugging purposes to stop the FPGA at certain points is not working correctly on 50 MHz and 100 MHz. The error can easily made visible if a counter is being debugged and the stepflag is set always high. SGI is contacted about this problem and a patch has been made available. This patch is not officially released.

D.3 Improper functionality of Debug tool with multi-buffering and streams

The debug tool SGI provided for the FPGA, gdbfpga seems not to work with the example *alg12_strm* they provided. Without multi-buffering the debug tool is working fine.

D.4 Claim of data transfer with dynamic memory sizes to SRAM is not working

In the RASC UserGuide SGI claims data can be transferred to the SRAMS as long as its 128 Byte aligned. However this is not the case. Only fixed sizes to SRAM can be send, with the size specified prior executing the application. This can decrease performance since at runtime zero padding might be necessary.

D.5 Transfer of data seems not working for all sizes

During testing the SMVM, the application seemed to crash when specific sizes for SRAM were allocated while for other sizes it worked. Testing a matrix with N=1024 and N_z=5110 failed in transferring the data when the SRAM size for A is taken 5110 words (1 word = 8 Bytes). This is a multiple of 128 Bytes and should work. However if the size overdimensioned to 8096 words the algorithm seems to work fine.

D.6 Error with algorithmic version number

This error seems not so important but this error influences the entire system. On top of that it was very hard to track this error. The user has to specify an algorithmic version with the design. If the algorithmic version contains a zero behind the . as in 10.0, the devmgr_server crashes. The FPGAs become invisible (using the devmgr -i option) and querying the algorithms in the registry (devmgr -q) results in a quantization packet error. The FPGAs can not be accessed anymore until the design is updated (devmgr -u) with a different version, like 10.1 for example. SGI might be not aware of this problem.

D.7 Incorrect memory allocation using DIRECT IO

If the design is using direct memory for faster transfer times, the allocation of is handled by the void *rasclib_huge_alloc(long size) function. According to the documentation, the size argument is rounded up to the next 128B cacheline. However it is found that this function is not allocating memory for small array sizes. Using gdb, allocating memories for different arrays resulted sometimes in the same pointer. The following allocations are requested:

```
#define N_size 32
#define Nz_size 32
mat_row = (unsigned long *) rasclib_huge_alloc ((N_size+1)* sizeof(unsigned long));
mat_col = (unsigned long *) rasclib_huge_alloc (Nz_size * sizeof(unsigned long));
mat_A = (double *) rasclib_huge_alloc (Nz_size * sizeof(double));
mat_B = (double *) rasclib_huge_alloc (N_size * sizeof(double));
mat_C = (double *) rasclib_huge_alloc (N_size * sizeof(double));
```

The requests result in the following addresses:

```
(gdb) print mat_row
$4 = (long unsigned int *) 0x8000000000000000
(gdb) print mat_col
$5 = (long unsigned int *) 0x8000000000004000
(gdb) print mat_A
$1 = (double *) 0x8000000000008000
(gdb) print mat_B
$2 = (double *) 0x8000000000008000
(gdb) print mat_C
```

```
$3 = (double *) 0x8000000000008000
```

It can be clearly seen that allocation of the addresses are not correct, since different arrays map to the same memory locations. If N_size and Nz_size are changed to a larger size, like 16384, the allocation is working properly.

D.8 Incorrect functionality using DMA streams with non-power of 2 stream sizes

When the DMA streams are used to transfer data from main memory to FPGA the applications fails execution for stream sizes non-equal to powers of 2. Currently, in the SMVM design this is avoided by sending additional zeros to fill next power of 2. The done signal (*alg_done*) for the SMVM design is not depending on this stream size. However, the behavior of the application is unknown when the alg_done flags is raised while data, the additional zeros, are still being transferred to the FPGA. Multi-buffering is not considered here.

D.9 Recommendation to improve routing issues

Without additional work, the Xilinx PAR requires a lot of effort to route the SGI wrappers. As a simple test case, a circuit has been synthesized containing 1 adder which add two values from SRAM 0 and SRAM 1 and store these in SRAM 2 using the memory configuration with 5 SRAMS at a frequency of 50 MHz. The 2 unused SRAMS have been disabled with the Configuration Tool. After the route phase, the trace tool reports that the frequency could not be met.

The following options can improve the routing of the design:

- Place a register for all the connections between the Custom Computing Unit (CCU) and the RASC-core (this is also being advised by the SGI Support Team)
- In the <project_name>.scr file in the implementation folder, change the value of the keyword equivalent_register_removal to NO. With equivalent_register_removal equivalent register at the RTL Level can be removed. Due to critical timing near the I/O boundary, the design routes better with this option. Since the global switch is changed, the affect of this may be unwanted on the CCU. To avoid this, a local override must be placed in the CCU (see Xilinx documentation to do this).
- The reset signal can suffer from high loads. To avoid this the reset signal can be routed through a global line.

Using these guidelines, the design of the Sparse Matrix Vector Product could be routed on 100 MHz using 5 SRAMS and 2 DMA Input Streams.

CD-ROM contents and Userguide

E

This appendix gives the reader an overview of the contents of the CD-ROM accompanying my thesis.

E.1 CD content

Figure E.1 contains the folders inside the CD-ROM.

. <Thesis >	
|– VHDL	
|– CCU_1PE	Contains the design files for CCU with 1 PE
|– CCU_2PE	Contains the design files for CCU with 2 PEs
|– CCU_mixed	Contains the design files for CCU with mixed PEs
|– CCU_2_4_8PE	Contains the design files for CCU with 2,4 and 8 PEs
|– OpenFOAM	
|– Profiling	Contains several subfolder with profiling results
|– SimpleFoam	Modification of SimpleFoam application to support hardware
|– lduMatrix	Files changed in lduMatrix folder in OpenFoam
|– Matrix Generation	Contains several files to generate automated tests and matrices.
|– BenchMarks	Includes files to automatic generate matrices from the UF Tim Davis collection for the OSKI tool
|– Altix Designs	CCU with 1 and 2 PE with automated makefiles for the RASC-core

Figure E.1: Organization of the file directory structure on the CD belonging to this thesis

In the remainder of this Appendix chapter we describe each folder briefly.

E.2 VHDL

The VHDL folder contains 4 subfolders with the developed designs. CCU_1PE contains the design for 1 PE. CCU_2PE contains the design for 2 PEs, specifically build for the RASC-core since it uses DMA streams. The CCU_mixed folder contains the file for the mixed Dense and Sparse Matrix Vector Multiplication. The last folder, CCU_2_4_8PE,

contains CCUs with 2, 4, and 8 PEs, but with optimized testbenches that do not require memories to simulate the memory banks on the RASC-Core. In stead of using memories, data is directly read in from files.

E.3 OpenFOAM

The OpenFOAM folder contains the 3 following subfolders:

- Profiling. The profiling folder contains all the profiling reports and timing measurements for the OpenFOAM tool running the simpleFoam solver, using Metis and Simple decomposition for 1, 2 or 4 CPUs on the Altix machine.

- SimpleFoam. This folder contains the files that reserves FPGA from the SimpleFoam solver application. The fpga_init.h reads the file fpgaDict depicted in Figure 5.7 and fpga_close.h disconnects the FPGA. Further the SimpleFoam.C file has been changed to include these files at the right time. All these files must be places into the ../OpenFOAM/OpenFOAM-1.4.1-taouil/applications/solvers/incompressible/simpleFoam folder.

- lduMatrix. The lduMatrix folder contains the adjustments made so far to support the hardware directly from OpenFoam. Further, it includes fpga.c and fpga.h used as the library to connect OpenFOAM to the RASC-Core. Last, cycles.h is a file that has been used for cycle time estimations in software to obtain the percentage of kernels that is spend in communication and computation (see Table 2.8).

E.4 Matrix Generation

In the folder Matrix Generation 3 files can be found:

- generate_test.c This file generates a matrix. The Number of rows can be specified by N and together with the minimal and maximal number of nonzero per rows. Each row length is generated randomly between the boundaries. Several files are created with random files for matrix A, vector **b** and **c** and the column coordinates.

- stream_2pe_test_gen.c This file does the same but automatic splits the mapping on the memory banks according to Figure 4.14. The outputs are generated for hardware testing purposes.

- divmat_generic_size.c This files generates test matrices suited for the CCU with multiple PEs for simulating multiple PEs from files. The input should be a valid matrix in CRS format, and the output will be a decomposed matrix in several sections depending on the number of PEs. The files can be read in simulation in the VHDL/CCU_2_4_8PE folder. Several options can be specified in this file like an automatic balanced partitioning.

E.5 Benchmarks

The folder Benchmarks contains 3 Files:

- crs_format_from_dia.m This file converts matrices from the OpenFOAM format to the CRS format.
- crs_format.m This file downloads a matrix from the The University of Florida Sparse Matrix Collection and converts this to the CRS format
- benchmarkOSKI.c This file uses the OSKI library to compute the Sparse Matrix Vector Multiplication in optimized software kernels.

E.6 Small UserGuide to use the Altix Designs

In this section a small user guide is given that describes the fast usage of the hardware design for the Altix machine. To be able to understand how everything works precisely, the user is adviced to read the SGI altix manual "Reconfigurable Application Specific Computer User's Guide" [13].

Currently, the latest RASC-software is called ia32_rc100_22_dev_env.tar.gz and can be found at the following path, /usr/share/rasc/ia32_dev_env/ on the Altix machine. This file must be extracted on the users personal computer (not on the Altix). Next, the RASC environment variable must point to the folder where the file is extracted. For a tcsh shell this can be done by the linux command:

```
$ setenv RASC <folder location>.
```

The next step is to download the contents of the Altix Designs on the CD-ROM to the correct locations where the file ia32_rc100_22_dev_env.tar.gz is extracted, the $(RASC) environment variable. In the Altix Designs two folders can be found, an implementations folder and an alg_core folder. The contents of the implementations folder must be copied to $(RASC)/implementations and the content of the alg_core folder must be copied to $(RASC)/design/alg_core folder. They both contain two folders called sparse and sparse_stream, where the first design contains the CCU for 1 PE and the latter for 2 PEs.

Now the folders are copied to its correct path, the Xilinx ISE tool and Python2.4 must be made visible on the command line. If they are included, go to the $(RASC)/implementations/<design name> folder, where design map is one of the present designs (sparse or sparse_stream), and run from their make. This will automatically compile all files and generate a bitstream. This can take several hours. Do this for both designs.

After the bitstream is generated it has to be included to the RASC registry on the Altix machine. The devmgr command does this. To upload the bitstream for the design with 1 PE enter the following command from the $(RASC)/implementations/sparse folder:

```
$ devmgr -a -n "sparse\_matrix" -b ./rev/sparse.bin -c user\_space.cfg
-s core\_services.cfg
```

This adds the file to the registery. Similar, go to the path $(RASC)/implementations/sparse_stream and type the command:

```
$ devmgr -a -n "sparse\_stream" -b ./rev/sparse\_stream.bin -c user\_space.cfg
-s core\_services.cfg
```

Now both streams are included to the registry with names sparse_stream and sparse_matrix. This can be verified with the command:

```
devmgr -q
```

The last step to do is to compile the C-file sparse_mat.c located in $(RASC)/implementations/<design name>. The following commands compile and execute the file:

```
$ cc -c sparse_mat.c
$ cc -o sparse_mat sparse_mat.o -lrasc -lm;
$ ./sparse_mat <N>  // used for 1 PE
$ ./sparse_mat <Ndiv> <N>    // used for 2 PEs
```

N represent here the matrix size, and Ndiv, the row index where the matrix is cut. In case the design with 2 PES (sparse_stream) is used, Ndiv must be specified. The first PE computes the first Ndiv rows and the second PE the last N-Ndiv rows. Inside the C-file, the user must specify the locations of the files that store the input matrix, what type of connection is used between CPU and FPGA (DIRECT_IO or BUFFERED_IO) and which arrays are transferred to the FPGA.In the folder Matrix Generation on the CD-ROM files are included, which can be used to generate random test examples for both 1 and 2 PEs.