CHALMERS
UNIVERSITY OF TECHNOLOGY

Investigation of possible acceleration of OpenFOAM

With the use of FPGA

Degree Project, BSc in Electrical Engineering

DANIEL JOHANNESSON
SIMON LINDAHL

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2014

Investigation of possible acceleration of OpenFOAM
With the use of FPGA

Daniel Johannesson, Simon Lindahl

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Sweden
Tel: +46-(0)31-772 1000
Fax: +46-(0)31-772 3663

Department of Computer Science and Engineering
Gothenburg, 2014

Foreword

This Bachelor thesis was carried out at the company Synective Labs, a company actively working with FPGA and GPU solutions. The project was performed within eleven weeks during the second semester of the year 2014 corresponding 15 credits. The authors of this work are taking a Bachelor of Science education with the specialization as Electrical engineers. Lars Svensson is the examiner of our project, which are supervised by Georgi Gaydadjiev and Catalin Ciobanu.

We want to thank the following persons: Thomas Luvö and Håkan Dahlbom at Synective Labs who started this project and for all the help on the way, Ivan Majdandzic and Tomas Frostensson who always answered our questions and Lars Svensson for his calming words.

Daniel Johannesson and Simon Lindahl

Abstract

Increasing software performance makes it possible to compute problems consisting larger data sets, which corresponds to more accurate real life simulations. However, there is always strive to achieve even faster computations is always desirable due to profiting from reduced time. By implementing hardware designed to perform a specific task can accelerate computations as well as it is power efficient. OpenFOAM is a computational fluid dynamics tool able to simulate real life problems such as thermodynamics. The authors got the task to investigate if OpenFOAM is suited to be accelerated with an FPGA. During the investigation a model was built from an existing model and after profiling the computations, the Gauss-Seidel smoother where chosen to be implemented on a FPGA. Several attempts were made to understand the arithmetic function of the smoother. The implementation made for it is actually slower than the regular CPU based software. One of the reasons is that the PCIe cannot transfer data fast enough to the FPGA, to resolve this, another smoother have to be implemented that get the same result as Gauss-Seidel. Such as Jacobi's method which can be even more parallelized. Other possibilities include finding different matrix compression method which is more beneficial for the task or even use GPU:s as accelerators

Table of Contents

Denotations

ALU - Arithmetic Logic Unit

CFD - Computational Fluid Dynamics

CLB - Configurable Logic Block

CPU - Central Processing Unit

CUDA – Compute Unified Device Architecture

DSP - Digital Signal Processing

FIFO - First in, First out

FPGA - Field-programmable gate array

FSM - Finite-State Machine

GPU - Graphics Processing Unit

HDL - Hardware Description Language

IEEE - Institute of Electrical and Electronics Engineers

IOB - Input/Output Block

OpenFOAM - Open Field Operation and Manipulation

PCIe - Peripheral Component Interconnect Express

PLD - Programmable Logic Device

RAM - Random Access Memory

SMVM - Sparse Matrix-vector Multiplication

VHDL - VHSIC Hardware Description Language

VHSIC – Very High Speed Integrated Circuit

1. Introduction

Since software today increase in performance, they are also applied with bigger tasks with heavier calculations. It is beneficial to reduce the computation time due to the simple reason that is saves time. To accomplish a reduction in computation time one option is to design hardware capable of performing computations regarding a specified task. FPGA:s consist of large arrays of configurable logic blocks (CLB), digital signal processing blocks (DSP), block RAM and input/output blocks (IOB). Usage of DSP:s and CLB:s are similar to a processors arithmetic logic unit (ALU) except the fact the CLB:s can be programmed solely with operations needed by an application, instead of being designed in a general-purpose manner. Because of this higher computation efficiency can be acquired at the same time as power efficiency is gained. The FPGA architecture enables application specific ALU:s to be constructed, which makes data-level parallelism possible. Due to direct data flow between operators data can efficiently be pipelined through a design [1]. When using FPGA and/or GPU to speed up the CPU, the performance can be increased substantially. For example:

- Computational speed increased up to 10 times – 100 times.
- System size reduced up to 10 times.
- Power consumption reduced up to 10 times [2].

These speedups are especially interesting for companies within fields with computing intense applications, such as OpenFOAM.

1.1. Background

The company Synective Labs is interested to know the possibility to accelerate the Computational Fluid Dynamic (CFD) software OpenFOAM with FPGA:s to increase its competitiveness. There exists a few other CFD-tools on the market that are faster but also expensive since they are licensed. But open source software are free and Synective Labs believes there is an opening in the market for OpenFOAM if it were to become faster.

1.2. Project goals

1.2.1. Objectives

The main goal of this project is to investigate the possibility to accelerate the open source software OpenFOAM with an Xilinx Virtex 6 FPGA. As guidelines some questions need to be stated:

- Is it possible to accelerate OpenFOAM using an FPGA?
- Which function or part of OpenFOAM is best suited for acceleration?
- Are there any limitations for the speedup gained? If so, what is the cause of the limitation?

1.2.2. Scope

Since the project is very extensive and the time is short some limitations need to be set. The model which Synective Labs requested can be simplified to get a working model faster. The focus will lie on a module within OpenFOAM software that solves heat transferring problems. Another limitation are that only one or a part of a function needs to be implemented since programming VHDL code is time consuming. Furthermore when coding VHDL, utilization of the LogiCORE IP Generator tool in Xilinx ISE is used to generate as much as possible, such as memory blocks and FIFO:s. Because writing floating-point arithmetic is very challenging for the time-schedule of this project.

1.2.3. Related work

A similar project where done by Taouil at Delft University of Technology [3] where he constructed a hardware accelerator for the OpenFOAM sparse matrix-vector product. His project covers a profiling and acceleration of a specific solver, `simpleFoam`. Furthermore he investigates different sparse matrix formats. This has been a great source for quickly learn and understand sparse matrices.

Mannakkara [4] wrote about implementing an asynchronous design on a FPGA to accelerate the Gauss-Seidel method. This design cannot be implemented in the present project since his design uses fixed-point, where this project needs floating-point. Furthermore his design implements an asynchronous design which is difficult to construct.

2. Method

To start the project, an investigation of OpenFOAM is carried out to get a greater understanding of the software and how to build a model. This will be a primary focus since a working dataset has to be acquired. When enough information is acquired about constructing a case in the software, construction of a case matching the specifications set by Synective Labs. When a working dataset is obtained an investigation process will be carried out, to find the most time consuming parts within the software and whether the time consuming parts are suitable for FPGA implementation or not. During this process a profiling tool will be used to assist in analyzing the function usage as well as time consumed within these functions. Profiling results will be evaluated where each of the top functions is examined to see what sort of computation is made. For the examination a great effort is put in to understand the C++ code and to see where heavy computations are done. This will make it easier to choose a function to accelerate. When a selection has been completed, the next step is to start planning the hardware architecture of the implementation before constructing a VHDL implementation. During the construction the code will be simulated to see if it is working correctly. This is an important step since the output of the hardware need to match the software output. Lastly when the output of the hardware matches the software output, an evaluation of the acceleration gained or lost is done.

The company has provided an Acer computer with 5.3 Gb DDR3 RAM, an AMD A6-3620 processor with four cores and 2.5 GHz speed. Which is used to run OpenFOAM version 2.3.0 installed on Ubuntu 13.10 64 bit. Additional hardware such as a FPGA made of Xilinx, model Virtex 6 XC6VHX380T where provided. The FPGA has 27 648 kb block RAM and 4 760 kb distributed RAM, which during the implementation of the VHDL code will be essential to see if everything fits on the FPGA. Additional tools used are Xilinx ISE 14.2 Design Suite to simulate and code VHDL and Perf, which is used for profiling software.

3. Technical background

The following chapters give a deeper insight into what an FPGA and OpenFOAM is. Here we also describes what the IEEE 754 Standard is and how rounding works for binary floating-point.

3.1. FPGA

FPGA is a field-programmable gate array, which is one of the most sophisticated programmable logic devices (PLD). Compared to a standard chip which have a fixed functionality, PLD:s are digital integrated circuits that consists of programmable blocks of logic which are connected with programmable interconnects [5]. This allows the circuitry to be configured for a wide range of different applications. The user can program a FPGA by using different languages, such as VHDL and Verilog HDL [6].

FPGA was created in the mid-1980s. It was mainly used for glue logic, such as simple logic functions and address decoding circuitry, and tasks with relatively limited data processing tasks. Since then FPGAs have been undergoing a massive evolution and are now containing millions of gates, embedded microprocessor cores and high-speed input/output devices. Today they can be used to almost anything. Another major development is the memory of the FPGA, which is called embedded RAM. This has grown in size and can now contain several thousand bytes and is split in two categories: distributed RAM and block RAM. Distributed RAM is memory that is allocated all over the chip of the FPGA and used when smaller memory blocks are needed, such as state machines. Block RAM on the other hand is big chunks of data for implementation that require a lot of memory, such as single- or dual-port RAM or first-in first-out (FIFO) functions [5].

3.2. IEEE 754 Standard for binary floating-point

Binary floating-point can be represented with single- or double-precision floating point. The two differs only in length and number presentation. Table 1 shows each format's parameters.

Table 1: Format parameters for floating point

Parameter	Format	
	Single	**Double**
p	32	53
E_{max}	+127	+1023
E_{min}	-126	-1022
Exponent *bias*	+127	+1023
Exponent with in bits	8	11
Format with in bits	32	64

The precision, `p`, is the number of significant bits. E_{max} and E_{min} is the maximum and minimum value for the exponent, `E`. The sum of the exponent and a constant (bias), which is chosen to make the biased exponent's range non-negative is called `Exponent bias`, i.e:

$$e = E + bias \qquad (1)$$

These parameters form a binary representation according to following formula:

$$(-1)^s 2^E (b_0 \cdot b_1 \cdot b_2 \ldots b_{p-1}) \qquad (2)$$

Where `s` in the formula above is either zero or one to implement if the number represented is negative or positive. b_p are the binary representation of the number, also called fraction, see functions 3.

$$f = b_0 \cdot b_1 \cdot b_2 \ldots b_{p-1} \qquad (3).$$

`e`, `s` and `f` is put together, to get the binary representation just as Figure 1 and 2 shows.

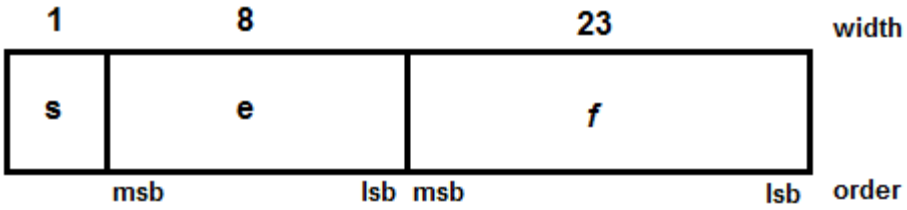

Figure 1: Binary representation of single precision

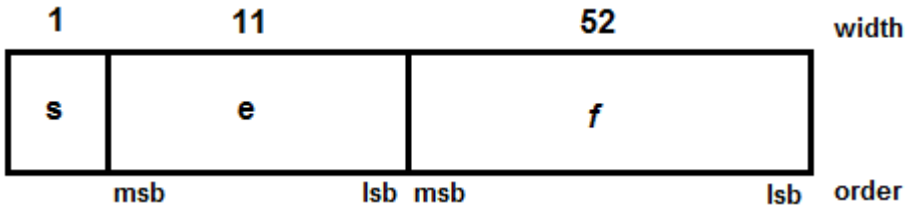

Figure 2: Binary representation of double precision

As seen in the figures above, `msb` mean most significant bit and `lsb` means least significant bit and width is the number of bits for each section [7].

3.2.1. Conversion between Binary and Decimal

Conversion between binary and decimal is possible, but the figure will always be rounded off. When implementing the IEEE 754 standard the representable value is rounded off to the closest most precise result. Such as if the two nearest representable values are equally near it will be rounded off to the value with the least significant bit zero. Rounding off to ∞ with no change in sign happens when the magnitude of the value extends to:

$$2^{[E_{max}]}(2 - 2^{-p}) \qquad (4)$$

In formula 4 E_{max} and `p` are determined by the wanted format, see table 1.

Consider two integers M and N, which represents a decimal value shown in formula 5 below.

$$\pm M \times 10^{\pm N} \qquad (5)$$

The maximum values for M and N when converting a decimal value to binary and vice versa is shown in the table below.

Table 2: Decimal conversion ranges

Format	**Decimal to Binary**		**Binary to Decimal**	
	Max *M*	**Max *N***	**Max *M***	**Max *N***
Single	10^9-1	99	10^9-1	53
Double	10^{17}-1	999	10^{17}-1	340

However for the rounding off in order to work as specified in the previous section the values for M and N needs to be within the limits as seen in table 3.

Table 3: Correctly rounded conversion range

Format	**Decimal to Binary**		**Binary to Decimal**	
	Max *M*	**Max *N***	**Max *M***	**Max *N***
Single	10^9-1	13	10^9-1	13
Double	10^{17}-1	27	10^{17}-1	27

Furthermore, conversions must be monotonic, which means that an increased value of a binary floating-point number shall not get a decreased value when converted to decimal, vice versa. This will occur as long as the digits for the decimal value is carried to the maximum precision specified in Table 2. Maximum M is 9 digits for single and 17 digits for double [7].

3.3. OpenFOAM

OpenFOAM was released by OpenCFD Ltd. in 2004 and is an open source software that is available for everyone to use. It is a Computational Fluid Dynamic (CFD) software that can handle much more than just complex fluids flows, for example solid dynamics, electromagnetics and chemical reactions. OpenFOAM is written in the program language C++ and all of its source code can be reached at OpenFOAM C++ Documentation web-page [8].

3.3.1. The mesh of a model

The most fundamental step when developing a 3D process is modeling. Most applications uses what is commonly called a mesh, a set of points describing coordinates as well as edges that arises from these points and faces that are surfaces enclosed by these edges. In OpenFOAM a model is built by a mesh which is defined by a set of vertices or set of points of which 3-dimensional models is described by. Within these set of vertices there can be an arbitrary number of cells, also called computational molecules, which represent coefficients connected to the properties of the specified area, where cells can take any shape. Cells are connected amongst each other by faces of the cell, for example two cells can share a face and will therefore directly affect each other. One cell can have an unlimited number of faces with an unlimited number of edges. Faces are defined by a list of points that are connected by edges that in turn are defined by points. These points are ordered in an anti-clockwise manner where the director of the normal vector can be acquired by using the right hand rule when looking towards a face [9].

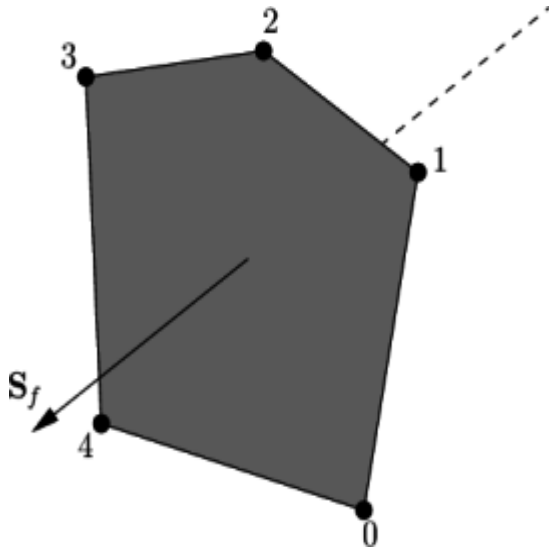

Figure 3: A face consisting of five points connected by edges and showing the face normal vector [9]

There are two types of faces in a mesh which are known as internal faces and boundary faces. Internal faces connects two cells, the face normal then points into an adjacent cell the direction being into the cell with the larger label for example the cell sorted later in a list. Boundary faces are faces connecting a cell to the edge of a defined space which means the face normal points outside the computational domain. Cells are defined by a list of faces [9].

Geometric shapes can be acquired by defining coordinates which encloses a space with a number of cells within this space. In addition to this boundary, conditions needs to be defined to wherever

there is a cell owning a face adjacent to the end of this defined space. Boundary conditions can be specified to represent physical attributes so that a finite space can be defined, this way a real-life problem can be delimited and simulated. As an example, consider a 3-dimensional cube; it needs to be assigned coordinates that represents the area of which it will exist. Let's say that the cube is 1×1×1, this is preset as meters in OpenFOAM, and it will be given those coordinates in file `blockmeshdict` as seen in Figure 4.

```
vertices
      (
            ( 0 0 0 )   // Vertex number 0
            ( 1 0 0 )   // Vertex number 1
            ( 1 1 0 )   // Vertex number 2
            ( 0 1 0 )   // Vertex number 3
            ( 0 0 1 )   // Vertex number 4
            ( 1 0 1 )   // Vertex number 5
            ( 1 1 1 )   // Vertex number 6
            ( 0 1 1 )   // Vertex number 7
      );
```

Figure 4 Vertices for the Mesh

These set of coordinates (x, y, z), are ordered in a Cartesian manner and by configuring this list of points, a geometric shape can be acquired. In this case we have a cube as mentioned above. Figure 5 defines a block, which in this case is a hexahedron. It also defines which vertices to use and gives the option to set number of cells within the block. [10]

```
blocks
(
      hex (0 1 2 3 4 5 6 7)   //Vertex numbers
      (100 100 1)             //Number of cells in each direction
      simpleGrading (1 1 1)   //Expansion ratio of the cell
);
```

Figure 5: Geometric properties of a block

4. Implementation

In a logic fashion this section describes how we constructed the model and evaluated results from a profiling of OpenFOAM. It also further explains which dependencies the selected function has and how that reflects the outcome of the investigation. Lastly, the thought process regarding the VHDL implementation is explained.

4.1. The model

Synective Labs requested a model for OpenFOAM which should simulate a tube enclosing a fluid flow with a certain temperature. The tube should also be contained by a substance, which can be a fluid as well, to make it simple, with a lower temperature as compared to the fluid inside the tube. The desired area of simulation is the change in temperature of the liquid inside the tube when the flow changes to standing still.

However this model can be simplified by looking only at an intersection of this tube showing the outer fluid and inner fluid parted by insulation, which will be the intersection of the tube wall. As shown in Figure 6.

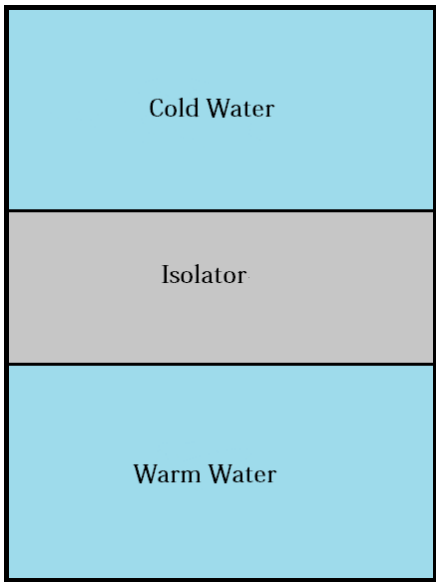

Figure 6: The model requested by Synective Labs

Containing three regions, where the lower region has the same properties of water with a flow equal to zero and a certain temperature, in our model the temperature is set to 300 degrees K. The middle region depicts the isolator, with the same properties as iron. Above the isolator there is the third region which has a constant temperature of 273 K that will act as a cooler affecting the other two regions.

An existing model, `planewall2D` [11] which is a recreation of an example from the book *Fundamentals of Heat and Mass Transfer* [12], was modified to the specifications of Synective Labs. In the following chapters there is a description of which files are changed to achieve this.

4.1.1. File structure

A model contains several folders where each one contains files and folders with different parameters to setup a model to a specific case. Figure 7 illustrates the file structure of *planewall2D*, to give a greater understanding when describing changes made to the *planewall2D* case in the following chapters.

```
.<planewall2D>
|- system folder
      |- bottomAir folder
            |- changeDictionaryDict
            |- decomposeParDict
            |- fvSchemes
            |- fvSolution
      |- topAir folder
            |- <same as bottomAir>
      |- wall folder
            |- <same as bottomAir>
      |- controlDict
      |- decomposeParDict
      |- fvSchemes
      |- fvSolution
      |- topoSetDict
|- constant folder
      |- bottomAir folder
            |- g
            |- radiationProperties
            |- RASproperties
            |- thermophsicalProperties
            |- turbulenceProperties
      |- polyMesh
            |- blockMeshDict
            |- boundry
      |- topAir folder
            |- <same as bottomAir>
      |- wall folder
            |- radiationProperties
            |- thermophysicalProperties
      |- regionProperties
|- time directories
```

Figure 7: The folder and file structure of planewall2D

4.1.2. Constant folder

The constant folder handles how the model should look like and the properties of each block in the model. Firstly *bottomAir* is renamed *bottomWater* and *topAir* is deleted. Furthermore the properties of air is not the same as water, this is accomplished by changing some constants in the *thermophysicalProperties* file. This is also done for the same file in the *wall* folder since the wall in planewall2D is concrete. However, in our model the isolator will be iron.

```
thermoType
{
    type            heRhoThermo;
    mixture         pureMixture;
    transport       const;
    thermo          hConst;
    equationOfState perfectGas;
    specie          specie;
    energy          sensibleEnthalpy;
}

mixture
{
    specie
    {
        nMoles          1;
        molWeight       28.9;
    }
    thermodynamics
    {
        Cp              1000;
        Hf              0;
    }
    transport
    {
        mu              1.8e-05;
        Pr              0.7;
    }
}
```

Figure 8: Code from thermophysicalProperties

Figure 8 is the thermophysical properties for air, to get water instead *molWeight* is set to 18 g/mol and *Cp*, liquid viscosity, is set to 4181. Furthermore the transport properties *mu* and *Pr* also needs to be changed. Reduced chemical potential, *mu*, is set to 659e-6 and Prandtl number, *Pr* [13], is set to 6.62. OpenFOAM now handles the *bottomWater* block as water and not air. When concrete is changed to iron, the appropriate constants need to be changed accordingly.

```
regions
(
    fluid       (bottomAir topAir)
    solid       (wall)
);
```

Figure 9: Code from regionProperties

Some changes needs to be done to *regionProperties* to get OpenFOAM to understand what each block is for state, to be more precise Figure 9 shows which block is fluid and which is solid. Here *topAir* and *bottomAir* is replaced by *bottomWater* in the fluid region and *wall* is unchanged.

Lastly, a lot of changes are done in the *blockMeshDict* file in the *polyMesh* folder where the mesh of the model is expressed. The changes made were for the model's boundary condition. Most of the existing walls had the wrong boundary condition and had to be deleted and exchanged for walls with correct boundary conditions. In *blockMeshDict* these are set as *types*. There are a lot of types for walls such as *empty*, *patch* and *wall*. In the model all of the walls where set to empty except *topWall* that has the type of wall. When a wall is set to the type empty, OpenFOAM handles that wall as if it does not exist but the type wall can be set to a constant value, which was done in chapter 3.1.3. To review the complete *blockMeshDict* see Appendix A Figure 21.

4.1.3. System folder

In the system folder, the properties of the regions of the model is set, which as is also how OpenFOAM perform its calculations for the case. In the *controlDict* file the properties of which solver and the time between calculations can be set. This is shown in Figure 10. Most of the lines were left unchanged but some had to be changed to get a proper result in the simulation.

```
application     chtMultiRegionSimpleFoam;
startFrom       startTime;
startTime       0;
stopAt          endTime;
endTime         50000;
deltaT          1;
writeControl    timeStep;
writeInterval   100;
purgeWrite      5;
writeFormat     ascii;
writePrecision  7;
writeCompression uncompressed;
timeFormat      general;
timePrecision   6;
runTimeModifiable true;
```

Figure 10: Code in the controlDict file

To get a proper result when simulating, *deltaT* was changed to 10, this means that the timesteps are 10 time units. To make OpenFOAM write data every second timestep during the execution, *writeInterval* is changed to 2. Further changes include changing *purgeWrite* to 0 to show all of the data written by OpenFOAM. If *purgeWrite* is set to, for example 1, then only one time folder will be created and every new data will overwrite the old data, see time directories in Figure 7. Lastly *endTime* is changed to 30000, because to keep the amount of data at a reasonable amount. The application is the solver for the case. It can be changed but in this case it is unchanged. More information on the *controlDict* file can be found at [14].

Just like the constant folder *bottomAir*, is renamed to *bottomWater*, and *topAir* are deleted for the system folder. Only one file needs to be changed in the new folder *bottomWater*, and that

is `changeDictionaryDict`. In this file all the start values for the different properties are set, such as temperature and pressure. The velocity of the water is changed to zero; this is because it was specified by Synective Labs. The same changes is done with the `changeDictionaryDict` file in the *wall* folder with one exception, the wall on the top is set to a constant temperature of 273 K. This is to simplify the model and to achieve a constant temperature of the water above the isolation. See Appendix A Figure 22, 23 and 24 for the changes in `changeDictionaryDict` file for `bottomWater`.

Furthermore is `topoSetDict` an important file, which illustrated in Appendix A Figure 25. Here is where the blocks `bottomWater` and `wall` is located in the mesh. As seen in Figure 25 in Appendix A the block *wall* start at the coordinates (0 0.8 0) and ends at the coordinates (1 1 0.1) in the mesh. Meanwhile `bottomWater` start at the coordinates (0 0 0) and ends at the coordinates (1 0.8 0.1). This means that the model contains two separate blocks, `wall` and `bottomWater`. Now OpenFOAM can calculate the heat exchange between both of these blocks, by choosing suitable boundary conditions which allows adjacent regions to affect each other.

4.2. OpenFOAM sparse matrix format

Data structure is of great importance when it comes to hardware implementation, since an FPGA is well suited for performing high speed calculations. But will only be able to do so efficiently when data can be directly streamed to the hardware. Preferably data should be preprocessed before performing computations with the FPGA and to investigate this, the compressed matrix format that OpenFOAM currently uses will be looked into.

Matrices in OpenFOAM are sparse, meaning matrices that consist of a significantly higher number of zero elements in comparison to the number of non-zero elements. Usually when this is the case it is profitable, in terms of memory allocation, to remove all zero elements. Since for example a SMVM otherwise will result in a lot of unnecessary multiplications by zero. By removing all zero elements, a way of indexing for all non-zero elements must be introduced. There are a number of ways to accomplish this. Below depicts how this is done in the OpenFOAM software. Let's consider an example matrix as follows.

$$\begin{bmatrix} d_1 & u_1 & 0 & u_2 & 0 \\ l_1 & d_2 & u_3 & 0 & u_4 \\ 0 & l_3 & d_3 & u_5 & 0 \\ l_2 & 0 & l_5 & d_4 & u_6 \\ 0 & l_4 & 0 & l_6 & d_5 \end{bmatrix}$$

All non-zero elements are stored in three separate arrays; one array containing of all upper contributions, `upperPtr`, a second array containing diagonal values, `diagPtr`, and lastly, all lower contributions are stored in a third array called `lowerPtr`. These three arrays together represent the full matrix, such as $A = L + D + U$.

$$\texttt{UpperPtr} \quad \begin{bmatrix} l_1, & l_2, & l_3, & l_4, & l_5, & l_6 \end{bmatrix}$$

$$\texttt{LowerPtr} \quad \begin{bmatrix} u_1, & u_2, & u_3, & u_4, & u_5, & u_6 \end{bmatrix}$$

$$\texttt{DiagPtr} \quad \begin{bmatrix} d_1, & d_2, & d_3, & d_4, & d_5 \end{bmatrix}$$

Since all zero elements are excluded, the addressing, coordinates to positions within the full matrix, has to be introduced. OpenFOAM uses addressing by column combined with row to locate elements with arrays, *uPtr* and *lPtr*. When looking at U*pperPtr*, values in *uPtr* expresses column coordinates and *lPtr* row coordinates. While addressing elements within L*owerPtr*, *lPtr* combined with *uPtr* now expresses coordinates in reverse. Meaning that *lPtr* acts as column index, while *uPtr* acts as row index. *uPtr* and *lPtr* below is shown with values corresponding to the example matrix [15].

$$\texttt{uPtr} \quad \begin{bmatrix} 1, & 3, & 2, & 4, & 3, & 4 \end{bmatrix}$$

$$\texttt{lPtr} \quad \begin{bmatrix} 0, & 0, & 1, & 1, & 2, & 3 \end{bmatrix}$$

4.3. Investigate functions

To obtain enough information about the functions used, a profiling tool, Perf, was of great assistance. It gives a better insight of how the program is distributing its workload in terms of function usage. From that information it is possible to search for a candidate for acceleration. Optimally a function will be found that consisting of simple arithmetic which is executed over and over, of which several iterations can be executed simultaneously. This is the type of function that is suited for FPGA implementation. P*erf* produces an extensive list of functions which are ordered by the number of samples taken within each function. Although Perf does not consider the amount of time in each function, but it conveys helpful information and is of guidance to find where the most time is consumed when executing the case. Therefore further analysis of the code need to be done to find the bottleneck for each function. The following list consists of the result when using Perf while executing the case with a mesh of (100 100 1) and 8.000 cells:

- 2.25% inv
- 2.17% GaussSeidelSmoother
- 1,90% multiply
- 1,88% Amul
- 1,85% List

Notice that the functions *GaussSeidelSmoother* and *Amul* are in the top of the list among with *inv*, *multiply* and *List*. However when increasing number of cells in the mesh, a scaling in usage by the *GaussSeidelSmoother* and *Amul* is observed. This information is acquired by an additional profiling, result listed below. When increasing the number of cells which corresponds to matrix size, since matrices are N×N where N equals to the number of diagonal cells, the profiling indicates the *GaussSeidelSmoother* and *Amul* functions to be more computational heavy then the rest. This is shown when the mesh is increased to (300 300 1) in the *blockmeshDict* file. This also increases the number of cells to 72.000.

- 36,39% GaussSeidelSmoother
- 21,82% Amul

With an increased mesh, both GaussSeidelSmoother and Amul increased in usage. This is excellent for acceleration, since the speedup over the serial case increases when a model increases

in size. Other functions such as `inv`, `multiply` and `List` where not chosen because of their complexity in code and none of them consisted of any heavy computational loops.

4.3.1. GaussSeidelSmoother

Figure 26 in Appendix A, `GaussSeidelSmoother.c`, consists of several intertwined loops. Starting at the outer scope, an update of coefficients is performed before any kind of computations are made. This communicational part updates coefficients used for the computation, which is controlled by number of sweeps. Perhaps it is possible to include number of sweeps within the FPGA as well; further investigation will be needed to determine this.

`GaussSeidelSmoother.c` operates with the iterative technique known as GaussSeidel algorithm [16]. This method is used to solve square systems of *n* linear equations with an unknown x .

$$Ax = b \qquad (6)$$

Where x is initiated with a primary guess, $x^{(0)}$, which will converge into a better approximation after a number of sweeps. Gauss Seidel method is defined by the iteration.

$$Lx^{k+1} = b - Ux^{k} \qquad (7)$$

Matrix A in formula 6 is decomposed into a lower triagonal and a strictly upper triagonal. L now represents strictly lower elements within the matrix A as well as diagonal elements and U consists of the strictly upper contributions of matrix A .

By multiplying the lower triagonals inverse, x^{k+1} can be computed by using the previous x with the following iteration:

$$x^{k+1} = L^{-1}\left(b - Ux^{k}\right) \qquad (8)$$

Since L includes the diagonal of the matrix it can be taken advantage of, resulting in sequential computation of x^{k+1} as follows.

$$x_i^{k+1} = \frac{1}{a_{ii}}\left(b_i - \sum_{j<i} a_{ij} x_j^{k+1} - \sum_{j>i} a_{ij} x_j^{k}\right) \qquad (9)$$

As seen above there is a dependency when computing the next x value, due to the previous *x* will have to be calculated in order to obtain the lower product sum. However, upper product sum are not affected by this. This dependency will limit the level of parallelism that can be achieved, since some arithmetic operations will have to be executed sequentially in order to obtain every x_i^{k+1} . Perhaps this limitation can be diminished by isolating the dependency further since it only lies within calculating the lower sum. Lets consider the following example:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \qquad (10)$$

$$x_1^{k+1} = \frac{1}{a_{11}}\left(b_1 - \left(a_{12}x_2^k + a_{13}x_3^k\right)\right) \quad \Rightarrow \quad \begin{bmatrix} x_1^{k+1} \\ x_2^k \\ x_3^k \end{bmatrix} \qquad (11)$$

$$x_2^{k+1} = \frac{1}{a_{22}}\left(b_2 - a_{21}x_1^{k+1} - a_{23}x_3^k\right) \quad \Rightarrow \quad \begin{bmatrix} x_1^{k+1} \\ x_2^{k+1} \\ x_3^k \end{bmatrix} \qquad (12)$$

$$x_3^{k+1} = \frac{1}{a_{33}}\left(b_3 - \left(a_{31}x_1^{k+1} + a_{32}x_2^{k+1}\right)\right) \quad \Rightarrow \quad \begin{bmatrix} x_1^{k+1} \\ x_2^{k+1} \\ x_3^{k+1} \end{bmatrix} \qquad (13)$$

It is shown from the formulas above that x_i^{k+1} will, iteration by iteration, overwrite elements in vector x row by row. This will have an advantage in terms of amount of memory required when successfully overwrite a memory containing vector x , while reading values simultaneously. By the example above, a conclusion can be drawn that it is possible to compute the lower product sum column wise. While simultaneously, compute row by row matrix vector multiplication, since all elements located in $column_i$ will be multiplied with x_i^{k+1} . Lower triagonal part multiplications with rows further down, will have to be stored and subtracted to get the full x_i^{k+1} result. These pre-calculated products will need some sort of index, to know if the actual row being calculated have a pre-calculated product, which needs to be subtracted.

4.3.2. Amul

```
// Initialise the update of interfaced interfaces
initMatrixInterfaces
(
      interfaceBouCoeffs,
      interfaces,
      psi,
      Apsi,
      cmpt
);

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

register const label nFaces = upper().size();

for (register label face=0; face<nFaces; face++)
{
      ApsiPtr[uPtr[face]] += lowerPtr[face]*psiPtr[lPtr[face]];
      ApsiPtr[lPtr[face]] += upperPtr[face]*psiPtr[uPtr[face]];
}

// Update interface interfaces
updateMatrixInterfaces
(
      interfaceBouCoeffs,
      interfaces,
      psi,
      Apsi,
      cmpt
);

tpsi.clear();
```

Figure 11: Code from Amul.C

`Amul`, Figure 11, consists of a pre-computational part where interfaces are initialized, which is followed by an update of interfaces. The computational part, which will be in focus, consists of two loop performing several multiplication accumulate instructions. Initially a matrix diagonal is multiplied by a vector, which accumulates into a new vector. By multiplying lower and upper triagonal by a vector accumulated in a new vector. This is how matrix by vector multiplication is done. According to Taouil [3], there are dependencies for the computational part. The dependency lies within the second for-loop, which cannot be parallelized because of it. This is because all iterations depend on the previous iteration. Taouil [3] continues to say that a change in the OpenFOAM matrix format needs to be done to get rid of this dependency.

4.3.3. Final words

After investigating functions `GaussSeidelSmoother` and `Amul`, the conclusion drawn is that `GaussSeidelSmoother` are a more suitable candidate for implementation on the Virtex 6 FPGA. Because of its scaling in usage due to heavy computations. Repetitive in terms of computations, in addition changing sparse matrix format might not be necessary. Changing the matrix format will be to complex and too time consuming for this thesis. However it might be good for optimizing an implementation to reformat the data structure. The dependency will however be a limitation in the level of parallelization of this function since only one row can be computed at a time. Therefore the parallelizing arithmetic instructions which is not affected directly by the dependency will be focused on.

4.4. VHDL architecture

The VHDL code is structured by several blocks where each has its own function. Top-level is the first block which controls everything. Under top-level there are a block for storing all vectors from the CPU to the memory and another block which calculates the upper triagonal. One final block calculates lower triagonal and finalizing the calculation for each row. This chapter gives an overview of how these blocks work.

In the top level of the architecture, a state machine is controlling whether calculations shall be computed or if the FPGA shall transfer data from the software. It is constructed as follows; the state machine gets a start signal and begins to collect all the data needed for the calculations. When all of the data is collected calculations are initiated. This is the slowest way, since all calculations has to wait for the data to be stored in memories. In a private communication with Tomas Frostensson, System designer at Synective Labs, he mentioned that the PCIe can stream 256 bits every clock cycle which can be distributed in either four 64 bit or eight 32 bit channels. Since the construction needs values for Upper, Lower, X and B and addresses for the Upper-values and upper-values column indexes. It also needs ownerPtr which indicates how many values there are in each row for Upper and in each column for Lower. This means six vectors need to be transferred to the FPGA. To get the fastest transfer, all vectors need to be 32 bit in size which means single-precision floating point. One problem is that OpenFOAM uses double-precision floating point, 64 bit. This means some errors occur since rounding is a problem for values converted between single-precision to double-precision.

4.4.1. Output error

The output error was tested and controlled beforehand, using C++ for a standalone GaussSeidel method. This test was done by letting a 32 bit and a 64 bit version of GaussSeidel run in parallel, which allows to complete a comparison between the results. The formulas used to calculate error and average error are as follows:

$$Error: \left| \frac{(result_{32bit} - result_{64bit})}{result_{64bit}} \cdot 100 \right| \qquad (14)$$

$$Average\ Error: \frac{\sum Error}{N} \qquad (15)$$

From the test the average error was calculated to $3.59223 \cdot 10^{-6}$ % and maximum error was calculated to $2.44492 \cdot 10^{-5}$ %. This is a very low error meaning that the output from the 32 bit version is almost identical with the output from the 64 bit version.

4.4.2. Top-Level FSM

As mentioned the FSM in the top level controls input of data and the calculations. It also controls which signals that reaches the memories.

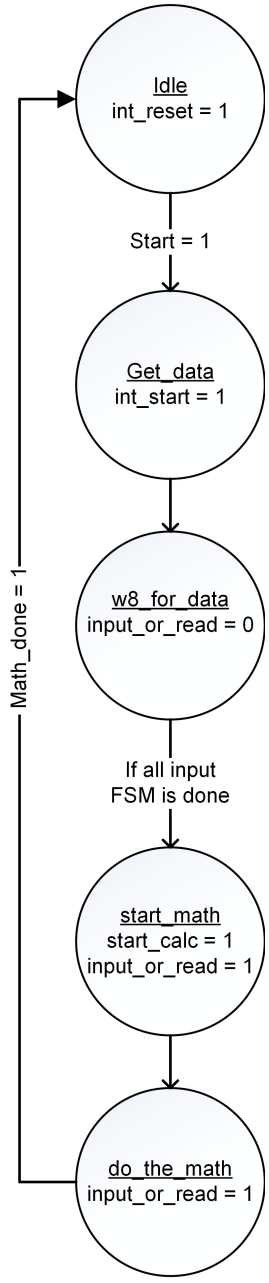

Figure 12: Flow chart of top-level FSM

As seen in Figure 12, the FSM sets *input_or_write* signal to zero when input of data to the memories are needed, and one when calculations are ongoing. This is needed because the memories only have two inputs and outputs. Since there are memories that need to be able to deliver two data-values at once, the top level FSM controls so there are no collisions in data requests for the memories. In addition the amount of memories needed will be reduced.

The top-level FSM has five different states. In *idle* everything is resetted and the FSM keeps waiting for a start signal from the software. *Get_data* starts the FSMs handling the input of data, while all of the data is being stored in the memories, the top-level FSM waits in *w8_for_data*. When all data has been stored it jumps to *start_math* which starts the calculations and sets the *input_or_read* to one. This is done to enable reading from both ports of the memory at the same time. Last step is to wait for the calculations to be done, when this occurs a jump to *idle* is performed.

4.4.3. Memory and FIFO

In the architecture several memories and FIFO:s are needed. FIFO is an acronym for first in, first out. This is a sort of data queue. Oldest data that is put into it is also the first one to leave it. It is used to quickly transfer data without accidentally losing any. All of the FIFO:s in the design has been generated by LogiCORE IP Generator tool in Xilinx ISE, same with the memories.

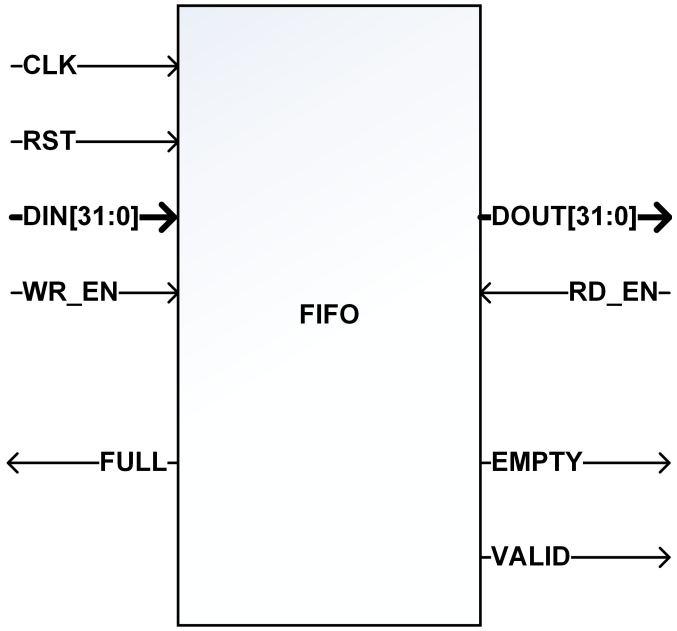

Figure 13: In- and Out-ports of a FIFO

As seen in Figure 13 it has several inputs and outputs. The signals *FULL* and *EMPTY* indicate if it is full or empty, just to prevent data loss for example when trying to write to a full FIFO. *WR_EN* enables writing to the FIFO while *RD_EN* requesting data from it. To clarify, when *WR_EN* is set to one the FIFO stores the value that are available on the *din* signal, vice versa. The last signal, *VALID*, is set to one when valid data is represented on the *DOUT* port, which will only occur when *RD_EN* is set to one to request data from the FIFO.

Memories for the project are block-RAM and there are two different memory sizes used in the architecture. Though all of them have elements which are 32 bit of size but the depth differs depending on which vector that is stored. For example the vector for the X-values has a memory with a depth of 8000 elements, same for the `B` vector and `ownerPtr`. `Upper-`, `Lower-` and `UpperAddr` vector all needs a memory with a depth of 16000 elements.

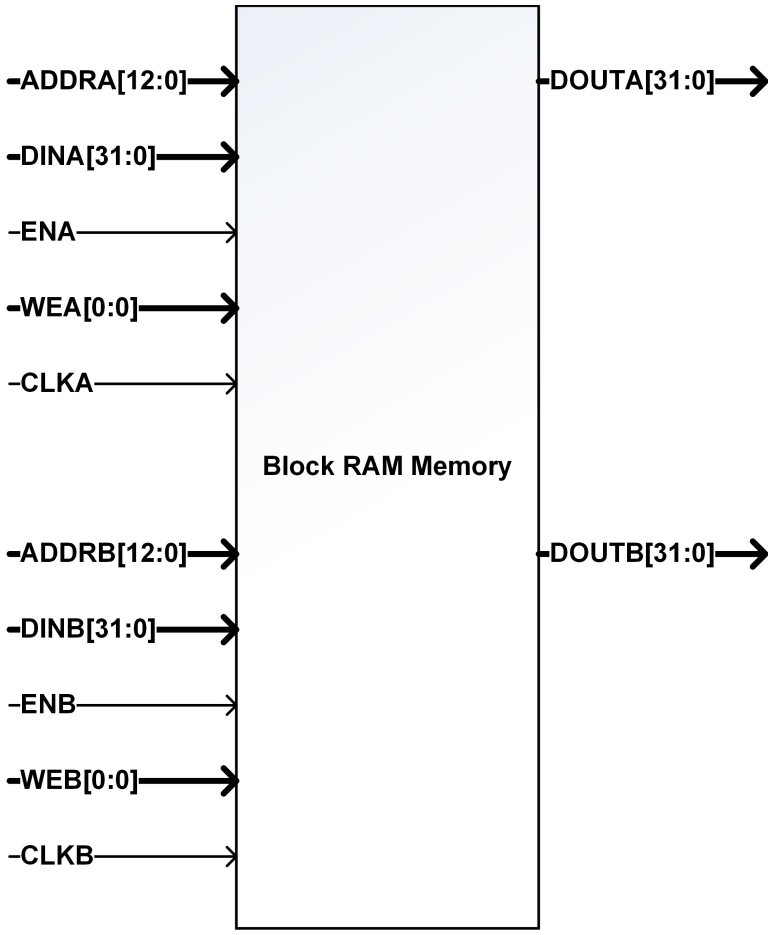

Figure 14: In- and Out-ports of a 8000 depth memory

Figure 14 visualizes the memory. All memories are dual port which means they can read and write at the same time, only limitation is that both ports cannot write to the same address and writing to an address while reading from it is also prohibited. *ADDR* is the address for an element in the memory. A memory of 8000 elements needs 13 bits to address all of the elements, on the other hand a memory with 16000 elements needs 14 bits. The signal `EN` enables the `CLK` for the port wanted to either write or read from. The *WE* signal tells the memory whether it shall store the data from the *din* port on the element with the address of *ADDR* or if it shall fetch data from the memory and send it to *DOUT*.

4.4.4. Incoming data

As mentioned before, there are six different streams of incoming data to the FPGA. This data need to be placed in the right memory and on the right place in the memory. To manage this FSM is needed, below is the flowchart of this type of FSM.

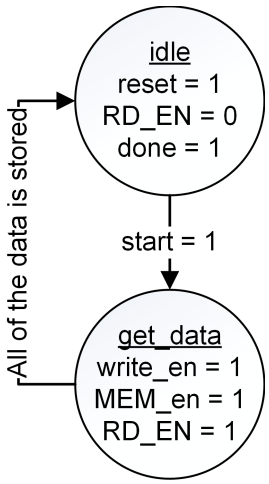

Figure 15: Flowchart for a input FSM

The FSM has several outputs and inputs. A start signal is received from the top-level FSM to begin to store data and *RD_EN* is set to one, to request data from the FIFO. *Valid_data* is received from the FIFO which is set to one when valid data is on the output of it, which is directly transferred to the memory which is enabled with the signals *MEM_en* and *write_en*. To keep track of which address to put the data in the memory, the FSM counts every valid signal. First valid signal the value stored in address zero is received, next valid signal the value stored in address one and so on. When the counter for the signal has reached either 8000 or 16000 depending on which memory used, the FSM resets and tells the top level FSM that it is ready with data transfer. For every vector that has to be stored on the FPGA is a FSM of this type.

4.4.5. Calculations

There are three operands needed to do all calculations: multiplication, addition and subtraction. The diagonal will be divided from all constant values on the CPU as a pre-process, which is why the quotient is not calculated on the FPGA. In the LogiCORE IP Generator tool all of operands can be generated and all of them can perform their designated operand with floating point numbers. A drawback is that each block have a latency, which means that when for example multiplication get data on the in-ports, it takes a certain amount of clock cycles to get a result on the out-port. The latency can be tweaked depending on the frequency of the clock. A higher clock frequency results in a higher latency while a lower frequency results in a lower latency. For this case the clock frequency of 400 MHz where assumed, because of the arithmetic blocks maximum frequency is 400 MHz.

In chapter 4.4 a dependency is mentioned that slows down the calculations. For every row in the matrix the previous calculated value is needed to complete the calculations of the current row. Further studies of the Gauss-Seidel smoother reveals that the upper triagonal is not dependent on the previous row, only lower the triagonal are. So to get the fastest possible calculations on the

FPGA all of the upper triagonal calculations can be done in parallel to the lower triagonal calculations. This means that for every row is a minimum of latency of one multiplication, one addition and one subtraction. This is the wait time to calculate the dependency, though the first row has a latency of one multiplication, one subtraction and one addition.

4.4.6. Calculation of Upper triagonal

From the example in chapter 4.4 the first row is calculated using this formula:

$$x_1^{k+1} = \frac{1}{a_{11}}\left(b_1 - \left(a_{12}x_2^k + a_{13}x_3^k\right)\right) \qquad (16)$$

As mentioned the division with the diagonal, in this case a_{11} , is already precalculated and scaled from b_1 , a_{12} and a_{13} on the CPU. The new formula is as follows:

$$x_1^{k+1} = b_1 - \left(a_{12}x_2^k + a_{13}x_3^k\right) \Rightarrow b_i - \sum_{j>i} a_{ij}x_j^k \qquad (17)$$

This can be expressed in the following binary tree:

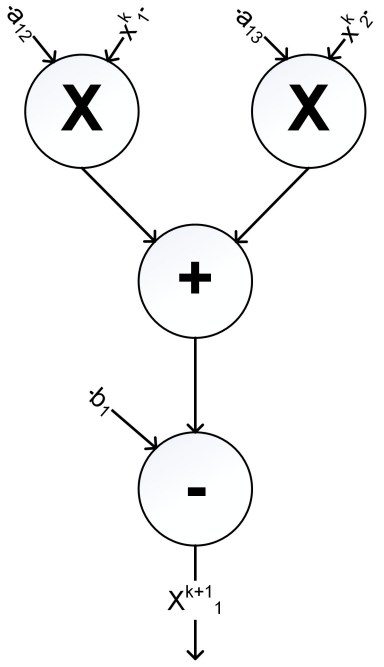

Figure 16: Binary tree for the first row

The binary tree in Figure 18 can be used for whole upper triagonal and illustrates function 16.

$$b_i - \sum_{j>i} a_{ij}x_j^k \qquad (18)$$

Input data can be streamed to the in-ports, which are fed with new values every clock cycle. This will result in a good throughput and latencies are concealed since a result value is acquired every clock cycle after the first result value is received, this method is known as pipelining. Each value is stored in a memory to be used later when calculating a whole row. The Figure 17 gives a greater understanding on how calculation of the upper triagonal is competed.

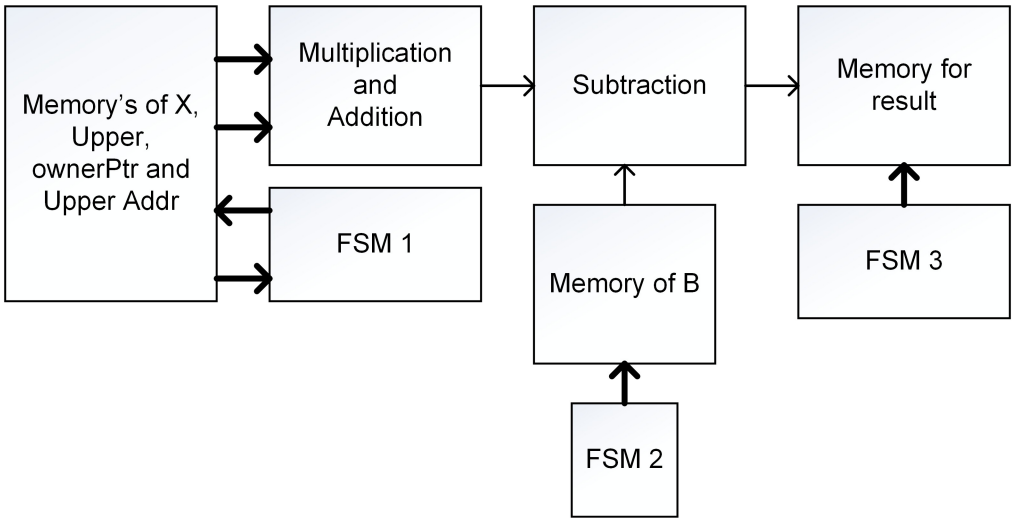

Figure 17: Function blocks for upper triagonal

When the top-level FSM signals to start the calculations, FSM 1 in Figure 17 begins to collect data needed to put into the pipeline of multiplication and addition of Upper-values and X-values. This process takes four clock cycles, since `ownerPtr` contains of how many X- and Upper-values there are in each row and `UpperAddr` contains the address for the X-values needed. To further explain, the first clock cycle is used get how many X- and Upper-values needed from the `ownerPtr` memory, let's say two. Afterward it needs to get two values from `UpperAddr` which is addresses to get the X-values needed. This takes one additional clock cycle. Thereafter another clock cycle is used to gather all the correct values of Upper and X from respective memory. On the last cycle, transfer them to the pipeline.

FSM 2 waits for the first value from the multiplication and addition block. This is known to arrive after the latency of the pipeline however it only need to wait for the latency the very first time. Plus three clock cycles since every fourth cycle a new value is put into the pipeline, which means a new value is presented every fourth clock cycle. A clock cycle before a new value from the pipeline is presented FSM 2 fetches the correct value from the B memory which is subtracted with the value from the pipeline.

Lastly FSM 3 main purpose is to store the result of upper triagonal. Just like the previous FSM it waits for a value to be presented from the subtraction. This means it has to wait for latency of the pipeline and subtraction plus four clock cycles. These values stored in the memory is used later to solve rows together with the lower triagonal.

4.4.7. Calculation of Lower triagonal

Lower triagonal is more complex compared to the upper triagonal. As mentioned in chapter 4.4 there is a dependency for the Gauss-Seidel smoother where the lower part of the matrix needs the previous calculated value for a row. Another difficult problem is how the Lower values are stored in the vector. As an example consider this example matrix:

$$\begin{bmatrix} d_1 & u_1 & 0 & u_2 & 0 \\ l_1 & d_2 & u_3 & 0 & u_4 \\ 0 & l_3 & d_3 & u_5 & 0 \\ l_2 & 0 & l_5 & d_4 & u_6 \\ 0 & l_4 & 0 & l_6 & d_5 \end{bmatrix}$$

For this matrix there are six values in the lower triagonal, l_1 to l_6 . The ownerPtr vector states how many values there are in each column instead of row for Lower. Which means that for the Lower vector the values are in this order: $[l_1, l_2, l_3, l_4, l_5, l_6]$. So when fetching l_1 from the memory to multiply with the correct X-value, the l_2 value can be fetched at the same time because it has to be multiplied with the same X-value. The figures below corresponds to the architecture of the lower triagonal.

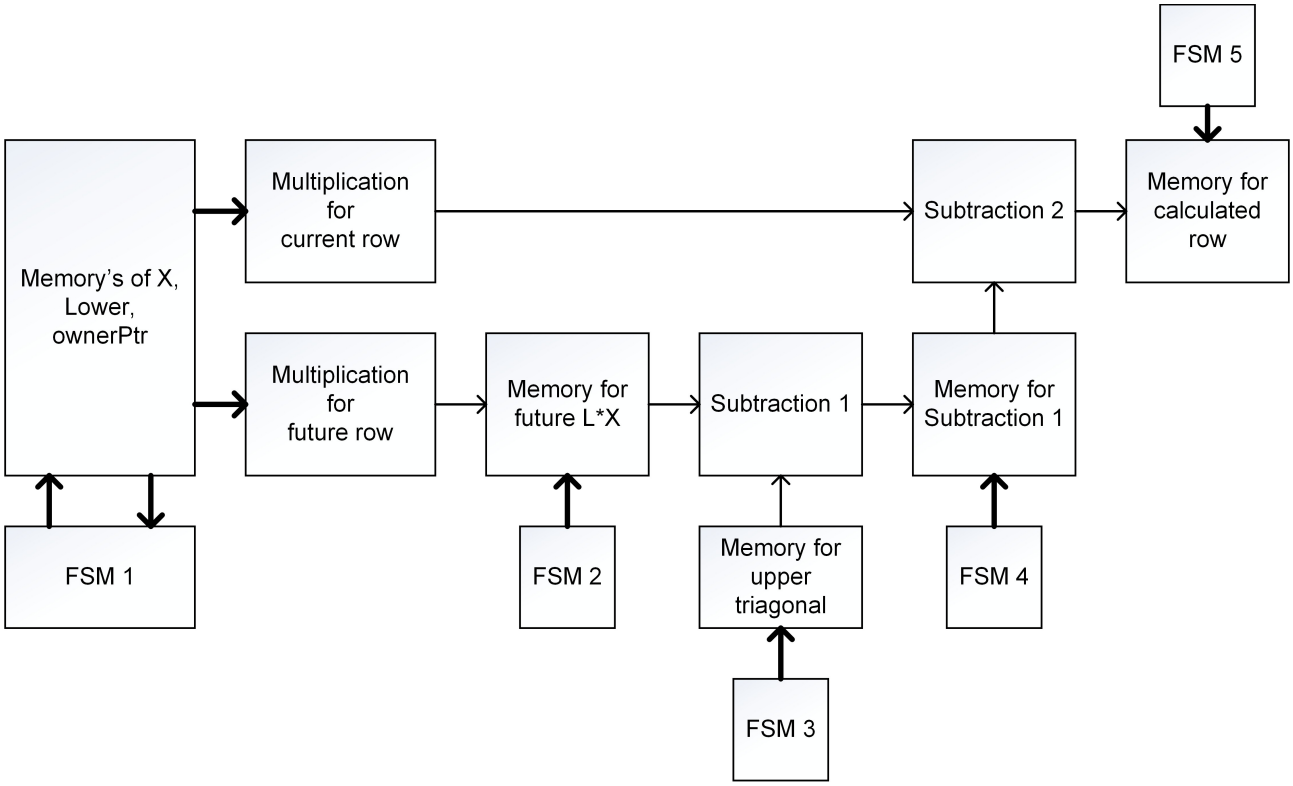

Figure 18: Function blocks for lower triagonal

There are a lot of similarities compared to the upper triagonal. For example FSM 1 for lower triagonal works exactly the same as for upper. It gathers correct values for X and Lower depending on ownerPtr. See previous chapter for more information. The only difference is that there are two

multiplications. Lets take the same matrix as example. Multiplication for current row block receives l_1 and x_0^{k+1} while multiplications for future row block receives l_2 and x_0^{k+1} .

FSM 2 stores the values from the multiplication for future row block at corresponding row. For this example it is stored on memory address three. Since l_2 is located on row three Meanwhile, FSM 3 checks if there are a calculated value on row one, same example as section above. Since there is not a previous calculated value from Memory of future $L \cdot X$ block, it simply bypass Subtraction 1 and stores it in the memory for Subtraction 1. If for example FSM 3 where looking at row three, it will subtract the value from the upper triagonal with $l_2 \cdot x_0^{k+1}$. FSM 4 constantly sends correct value to the Subtract 2 block to complete the final value for a row. While FSM 5 only stores each result from Subtraction 2 block.

4.4.8. Final words

At this point, only upper triagonal calculations are working as intended. Lower triagonal has all its building blocks completed but there are several timing issues that have to be resolved before the whole VHDL implementation is working. With this in mind, the only thing limiting the speed of this implementation is the lower triagonal and the calculation of the first row. Calculation of the first can be seen in Figure 19, which has a latency of 4 clock cycles and one multiplication, one addition and lastly one subtraction. Latency of the lower triagonal can be seen in Figure 21, which also has a latency of four clock cycles but only one multiplication and one subtraction.

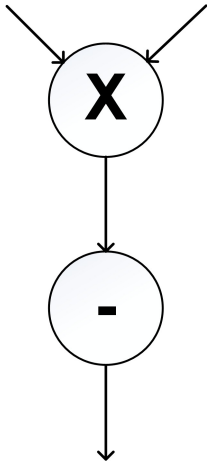

Figure 19: Lower triagonal binary tree

For this model with a mesh of (100 100 1) every vector fits into the memory of the Virtex 6 VHX380T FPGA. As the implementation is structured, the number of Block RAM used is 148 of 768, which is 19 %. This means that an even bigger model will fit into the FPGA. This is also positive for future acceleration of bigger models.

5. Result

OpenFOAM version 2.3.0 where installed on Ubuntu 13.10 64 bit on a Acer computer with 5.3 Gb DDR3 RAM, a AMD A6-3620 processor with four cores and 2.5 GHz speed. The model as seen in chapter 4.1.4 executes in 442,32 s. Gauss-Seidel smoother is called 113859 times which adds up to an execution time of 11,7628 s, which is 2,6469 % and approximately 0,10331 ms per call. By using Amdahl's law [17] the speedup can be calculated. If Gauss-Seidel smoother where minimized to zero seconds, the execution of the speedup would be 1,03 times faster, according to function 19.

$$\frac{1}{(1-P)}=\frac{1}{(1-0{,}026593417)}=\frac{1}{0{,}973406583}\approx 1{,}03 \qquad (19)$$

To get any acceleration on the FPGA, the implementation needs to be faster than 0,10331 ms.

The initial assumption of a clock frequency at 400 MHz cannot be implemented in our design, because the PCIe cannot deliver eight 32 bit elements every clock cycle at that speed. For the design to work with eight 32 bit elements every clock cycle, a frequency closer to 125 MHz will be more accurate. Following paragraphs describes the calculations for our case.

Each iteration of Gauss-Seidel does not contain the same amount of elements, sweeps and cells. For the model with a mesh of (100 100 1) they changes accordingly:

Table 4: Changes in parameters with each iteration

Iteration	Cells	Sweeps	Max Elements
1	31	4	54
2	62	4	110
3	125	4	232
4	250	4	466
5	500	4	956
6	1000	4	1931
7	2000	3	3911
8	4000	2	7861
9	8000	2	15821

Table 4 shows that each time the function is called, number of cells, etc., increases. This continues up to the ninth call, the tenth call restart the process. Since Gauss-Seidel is called 113859 times for the model, the number of recurring patterns is:

$$\frac{113859}{9}=12651 \qquad (20)$$

Knowing this, calculating the total time for the VHDL implementation is performed by calculate the time for the data transfer to and from the FPGA. Thereafter calculating the time for the FPGA to perform the calculations needed, which is later added to the time for the data transfer. The following section shows how to perform the calculations of the ninth call with 8000 cells, 2 sweeps and 15821 elements.

When transferring data to the FPGA via PCIe, a new element is presented every clock cycle. This means it takes time to transfer a lot of data. To transfer all data to the FPGA it takes:

$$\frac{number\,of\,elements}{clock\,frequency} \Rightarrow \frac{15821}{125\,Mhz} \approx 0{,}127\,ms \qquad (21)$$

Number of elements is the maximum elements in a vector. All vectors from the CPU transfers at the same time since PCIe can stream eight 32 bit elements simultaneously as mentioned in chapter 4.4. Additional data needs to be transferred back to the CPU from the FPGA. Since all memories on the hardware have two ports, it can send two elements at the same time to the CPU. The vector that is transferred back contains 8000 elements, so it only needs 4000 clock cycles to transfer all elements. This takes:

$$\frac{4000}{125\,MHz} = 0{,}032\,ms \qquad (22)$$

Total time for data transfer:

$$0{,}032 + 0{,}126568 \approx 0{,}159\,ms \qquad (23)$$

As mentioned in chapter 4.4.8 there is wait time for every row when performing calculations for Gauss-Seidel smoother. Firstly every arithmetic block generated with LogiCORE IP Generator tool has a latency from valid data is put into it and to valid data is presented on the output. Following table illustrates the latency for each arithmetic block when the clock frequency is set to 125 MHz.

Table 5: Latency for arithmetic blocks

Block	**Latency**
Multiplication	2
Addition	3
Subtraction	3

When calculating the first row, see chapter 4.4.6, a total latency of $2+3+3=8$ and four clock cycles gives a total of 12 clock cycles in latency. Thereafter for every row, a latency when calculating lower triagonal of $3+2=5$ plus additional four clock cycles gives a latency of 9. Total cycles for all calculations are:

$$12 + (9 \cdot 7999) = 72003\,cycles \qquad (24)$$

$$\Rightarrow \frac{72003\,cycles}{125\,MHz} \approx 0{,}576\,ms \qquad (25)$$

Since Gauss-Seidel smoother calculate the same matrix twice, set by the number of sweeps, this value has to be doubled to get total time for all calculations, which adds up to a total time of

1,152048 ms. Adding the time for transfer data, calculations and internal transfer of data gives a total time of:

$$0{,}158568\,ms + 1{,}1152048\,ms \approx 1{,}274\,ms \qquad (26)$$

These calculations can be done for every iteration and are summarized in the table below.

Table 6: Time for each iteration of the execution

Iteration	Data Transfer (ms)	Calculations (ms)	Calculations with sweeps (ms)
1	0,00056	0,002256	0,009024
2	0,00128	0,004488	0,017952
3	0,00236	0,009024	0,036096
4	0,004728	0,018024	0,072096
5	0,007648	0,036024	0,144096
6	0,019448	0,072024	0,288096
7	0,039288	0,144024	0,432072
8	0,078888	0,288024	0,576048
9	0,158568	0,576024	1,152048
Total	0,312768		2,727528

Table 6 shows that total time for data transfer for nine iterations is 0,312768 s and total calculation time is 2,727528 s. To get the total time for the whole execution:

$$(0{,}312768\,ms + 2{,}727528\,ms) \cdot 12651 \approx 38{,}463\,s \qquad (27)$$

That is 3,27 times longer compared to the CPU, which only took 11,7628 s. Also, one iteration takes according to function 28 approximately 0,339 ms, compared to 0,10331 ms per call for the CPU.

$$\frac{38{,}462784696\,s}{113859} \approx 0{,}339\,ms \qquad (28)$$

Furthermore calculations for how many GB/s is achieved can be done by summarize all elements that needs to be transferred and divide by the amount of time it takes to transfer them. For this case there are six vectors, each with different amount of elements, needs to be transferred to the FPGA. All elements consists of 32 bits.

Table 7: Total amount of elements

Iteration	**Elements**
1	246
2	516
3	1071
4	2148
5	4368
6	8793
7	17733
8	35583
9	71463
Total	141921

Total amount of elements as shown in Table 7 is 141921, which can be used to calculate GB/s.

$$141921\,elements \Rightarrow 141921 \cdot 32\,bit = 4541472\,bit \Rightarrow \frac{4541472}{8} = 567684\,Byte \qquad (29)$$

$$\Rightarrow 567684\,Byte \cdot 12651 = 7181770284\,B \Rightarrow 7{,}181770284\,GB \qquad (30)$$

$$\Rightarrow \frac{total\,amount\,of\,GB}{total\,time\,for\,data\,transfer} \Rightarrow \frac{7{,}181770284\,GB}{0{,}312768\,ms \cdot 12651} \Rightarrow \frac{7{,}181770284\,GB}{2{,}814912\,s} = 2{,}55\,GB/s \qquad (31)$$

2,55 GB/s is very plausible to achieve since Synective Labs uses PCIe2.0 x8 which can achieve a theoretical bandwidth of 4 GB/s [18].

One of the reasons for the slowdown is that the FPGA needs to have a low frequency on the clock. It needs to be low because of the bandwidth of the PCIe. With a faster clock on the FPGA the PCIe would only be able to send four 32 bit elements at a time, while it is needed to have six 32 bit elements at the same time for Gauss-Seidel smoother. Additional delays are when the FPGA waits for all the values before beginning its calculations. If able to use the correct values every clock cycle it arrives on the FPGA it can save up to 2,815 s, which still is not enough. But yes it is possible to accelerate OpenFOAM but the acceleration is limited to the bandwidth of the transfer between the CPU and FPGA. Furthermore the time it takes to evaluate if a certain function is accelerate-able and implementing it on an FPGA is considerable large.

6. Conclusion and Discussion

With a model constructed out of planewall2d, a function called Gauss-Seidel smoother where chosen to be accelerated and implemented with VHDL. Though only half of the implementation works as intended, calculations can be done to see whether it will be faster than the CPU. The execution time for the model where 442,32 s in which Gauss-Seidel smoother takes 11,763 s, corresponding to 2,647 %. Our implementation takes approximate 38,463 s and compared to the CPU it is 3,27 times longer.

A time plan where constructed before the start of this project, the Gantt-scheme can be viewed in Appendix C Figure 27. When constructing the time plan we had no idea how long everything would take. This can be seen in “Build the OpenFOAM model” which took three weeks instead of two. Because of the delay in building the model, the profiling of OpenFOAM and understanding each function where delayed. This also took longer than expected, five weeks instead of two. All this created a snowball-effect where everything got delayed and took longer time to execute than expected. This can be seen with the implementation of the VHDL code which is only half done, even though we had eleven week instead of ten weeks of working with the project. This project proved to be very challenging, especially when analyzing OpenFOAM source code, which was a bit more advanced than we are used to. Also, preparing and planning VHDL-design, considering the factors determining and affecting the result demanded a lot of investigation. Studying existing work more thoroughly prior to jumping in to this project would have led to a more structured approach, planning exactly what we needed to do every step of the way and might have saved us some time.

The result of this project, as described in the previous chapter, could not meet the preliminary expectations concerning acceleration level. The design, while not yet finished, is estimated to actually slow down the computations of the Gauss-Seidel Smoother. This is because of a bottleneck arises when transferring a lot of data to the FPGA through a PCIe where the bandwidth is a clear limitation. In addition, the design is not efficiently pipelined and will need to be optimized in order to result in acceleration. To be able to accomplish an efficiently pipelined FPGA-design of the Gauss-Seidel Smoother, the input data needs to be structured differently before streamed through the PCIe. Optimally, a structure where data can directly be fed into the arithmetic operators without have to index back and forth through memory blocks to obtain correct values to perform these operations. The assumptions that a clock frequency of 400Mhz could be reached, since such frequency is possible with IP core generated arithmetic block, is actually limited to about 125 MHz by the data transfer with PCIe. A clock speed of only 125 MHz will not be enough, with the VHDL-design intended. Since a critical point in the computations of the Gauss-Seidel smoother arises due to a delay on each row computed, because every row has to wait for previous row result. In addition, different matrices sizes are used for computation of the Gauss-Seidel throughout the execution of a case. Nine different matrix-sizes where found when analyzing the case calculations further. Previous assumption, that only one size of the matrix where used, which where the largest found where wrong, and it where used to estimate the time consumed by the Gauss-Seidel smoother. That result where not accurate. It is actually considerably less time consumed by the Gauss-Seidel smoother. Although, profiling results showed that usage does indeed scale with a larger mesh. Which means an acceleration of the Gauss-Seidel smoother should have a larger effect with larger matrices, directly corresponding to the size of the mesh.

Instead of using Gauss-Seidel method another linear solver might be implemented, one where there are no dependencies in each iteration. Let's consider the Jacobi method [19] of with the following iteration can be used to solve function 32.

$$Ax = b \qquad (32)$$

$$x_i^{k+1} = \frac{1}{a_{ii}} \cdot \left(b_i - \sum_{i \neq j} a_{ij} \cdot x_i^k \right) \qquad (33)$$

Where x_i^{k+1} is the result vector, a_{ii} the matrix diagonal, b_i a vector and a_{ij} symbolizes matrix elements excluding the diagonal, lastly x_i^k a vector given the initial guess. Since there are no dependencies in obtaining x_i^{k+1} for each row several rows can be computed simultaneously. However, the different convergence rates between the Gauss-Seidel method and Jacobi will lead to approximately twice as many iterations when using the Jacobi method before reaching the same accuracy [20]. To further investigate if Jacobi's method is a valid supplement for Gauss-Seidel we can do some calculations. Since Jacobi can be more pipelined a binary tree with four multiplications, two adders and one subtraction can be implemented. Which will have a latency of $2+3+3+3=11$ cycles. First value will be presented 11 clock cycles after first input, after that every clock cycle will present a new value. Since mentioned in chapter 5, there are different matrix sizes for every function call. To calculate the time for Jacobi's method, we are using the following formulas:

$$\frac{(Latency + (Cells - 1)) \cdot Sweeps}{frequency} = Time\ for\ calculation \qquad (34)$$

$$\frac{\frac{Cells}{2} \cdot (Sweeps - 1)}{frequency} = Time\ for\ internal\ data\ transfer \qquad (35)$$

$$\frac{\frac{Cells}{2}}{frequency} = Data\ transfer \rightarrow the\ CPU \qquad (36)$$

Table 8: Calculated times for the Jacobi method

Iteration	Cells	Sweeps	Calculations with sweeps (ms)	Internal data transfer (ms)	Data transfer to the CPU (ms)
1	31	8	0,002624	0,000896	0,000128
2	62	8	0,004608	0,001736	0,000248
3	125	8	0,00864	0,003528	0,000504
4	250	8	0,01664	0,007	0,001
5	500	8	0,03264	0,014	0,002
6	1000	8	0,06464	0,028	0,004
7	2000	6	0,09648	0,04	0,008
8	4000	4	0,12832	0,048	0,016
9	8000	4	0,25632	0,096	0,032
Total			0,610912	0,23916	0,06388

With the values from Table 8, the total time for the Jacobi's method can be calculated:

$$(0{,}610912 + 0{,}23916 + 0{,}06388) \cdot 12651 \approx 11{,}562\,s \qquad (37)$$

Even though Jacobi's method needs double the amount of iterations, it is faster than Gauss-Seidel. But other factors need to be addressed. Since it uses double amount of iterations, more data needs to be transferred between memories on the FPGA. In this example the function is fully pipelined with no delay between data input as our implementation has. If Jacobi can be fully pipelined it is better suited for FPGA implementation than Gauss-Seidel. Furthermore, since Jacobi's method does not have any dependencies, we can calculate several rows at the same time. For example, if five rows are calculated at the same time, the time needed to do all calculations is divided by five. On the other hand this means that the data has to be transferred before starting the calculations, since PCIe has a limited bandwidth.

A lot of problems arose when trying to build our own model in OpenFOAM. Luckily we found a similar model, `planewall2d` which we could modify. It was a challenge to understand the functions called by OpenFOAM since there where so many. When we finally chose Gauss-Seidel smoother as the function to implement, we got stuck trying to directly translate the C++ code to VHDL. This caused delays in our time plan. This was eventually solved by directly looking at the arithmetic’s for the Gauss-Seidel method and to construct an implementation which corresponds to it.

As conclusion, there is no definite best function to be accelerated for OpenFOAM. Since it is extensive software many functions need to be implemented on FPGA to get an acceleration worth of implementing. Our implementation was slower but shows that it is possible to use FPGA for acceleration but in this case the Gauss-Seidel smoother is not optimal since it has a dependency which makes it slow. In this case Jacobi's method would be faster if the data from the CPU can be streamed to the FPGA. Which mean that every element that are needed for every row is presented to the FPGA every clock cycle. If this is achieved Jacobi's method would be fully pipelined. But when matrices are growing the biggest problem is that the bandwidth of the data transfer becoming a bottleneck. Both Gauss-Seidel and Jacobi share this limitation for acceleration. Since bigger models needs bigger matrices, more data have to be transferred and stored this may make GPU:s a better choice for accelerating OpenFOAM. According to [21], GPU:s outperform FPGA in almost all their tests. Since GPU:s is more standardized it is easier to implement than FPGA which needs hardware developers and it has the disadvantage that it performs less floating-point operands per second than a GPU. When using FPGA:s and GPU:s to accelerate, a reduction in power consumption with 10 x to 100 x can be achieved [2]. Just because shorter computation time means that more computations can be done on the same period as before. This in the long run means less time sitting in front of the computer waiting for a result.

6.1. Future Work

Since this project did not lead to acceleration, due to an unfinished VHDL-design we have reached the conclusion that even a completed VHDL-design would still not have accelerated the linear solver. Partly because of the dependency in the thought VHDL architecture, a different approach might be to use an alternative linear solver. The linear solver should have better possibility to exploit its parallelism. OpenFOAM includes a variety of solvers, pre-conditioners and mesh manipulation, each with advantages and disadvantages regarding different problems. The computational workload seems to be distributed between functions, which means accelerating several functional will lead to a better result. In order to accomplish this, the sparse matrix format currently used by OpenFOAM might have to be altered. Data should be as compressed as possible and able to be streamed as well as calculated without dependencies, either using existing formats or perhaps a modified version of one. Since data transfer is limited by the PCIe bandwidth, a more compressed format can reduce the limitation of bandwidth, and result in a higher clock frequency. An idea might be to use run-length encoding (RLE) [22], where coordinates of non-zero elements in a matrix can be merged with the actual data. To further explain, n zero elements is presented as a zero followed by the number n . n represents the number of zeros in a row but will also determine where the next non-zero element in located in the full matrix, hence the known element in vector x can be found. In addition, if data corresponding to the number of elements per row can be merged with the actual data, the bandwidth will be reduced even further. Presently a current GPU implementation is already available through a plugin to OpenFOAM, SpeedIT, supplied by Vratis ltd. Including a library providing sparse linear solvers ready to be implemented in a CUDA supported NVIDIA-card [23]. With SpeedIT 2.4 they have achieved an acceleration up to 6 times. Further work like this can be the solution to accelerate OpenFOAM, to create general solutions for solvers and smoothers that easily can be implemented.

References

[1] P. S. Sandararajan, “*High Performance Computing Using FPGA's*”, [Online], Available: http://www.xilinx.com/support/documentation/white_papers/wp375_HPC_Using_FPGAs.pdf, [Accessed: 2 June 2014]

[2] Synective, “*Accelerating computations*”, [Online], Available: http://www.synective.se/index.php/acceleration/accelerating-computations/ [Accessed: May 27 2014]

[3] M. Taouil, “*A Hardware Accelerator for the OpenFoam Sparse Matrix-Vector Product*”, Delft University of Technology, Delft, The Netherlands, 2009.

[4] C. Mannakkara, “Asynchronous Pipeline Controller Based on Early Acknowledgement Protocol”, The Graduate University for Advanced Studies, Sokendai, Tokyo, 2010, page 75-90.

[5] C. M. Maxfield, *FPGAs: Instant Access*, Newspress, 2008, [E-book] Available: ScienceDirect.

[6] S. Brown, Z. Vranesic, “Fundamentals of Digital Logic with VHDL Desing”, Third Edition, New York: McGraw-Hill, 2009.

[7] American National Standards Intitute, *IEEE Standard for Binary Floating-Point Arithmetic: An American National Standard*, New York, The Institute of Electrical and Electronics Engineers Inc, 1985 [E-book] Avaible: IEEE Xplore.

[8] OpenFOAM, “*OpenFOAM C++ Documentation*”, [Online], Available: http://openfoam.org/docs/cpp/ [Accessed: April 20 2014].

[9] OpenFOAM, “Mesh description”, [Online], Available: http://openfoam.org/docs/user/mesh-description.php [Accessed: April 10, 2014]

[10] OpenFOAM, “*Mesh generation with the blockMesh utility*”, [Online], Available: http://www.openfoam.org/docs/user/blockMesh.php#x25-1430005.3.1 [Accessed: April 5 2014]

[11] OpenFOAMwiki contributors, (20 April 2014), Getting started with chtMultiRegionSimpleFoam: planeWall2D, [Online], Available: http://openfoamwiki.net/index.php?title=Getting_started_with_chtMultiRegionSimpleFoam_-_planeWall2D&oldid=15244

[12] F. P. Incropera, “Fundamentals of HEAT and MASS TRANSFER”, Seventh Edition, New York: John Wiley & Sons; 2011.

[13] F. M. White, “Viscous Fluid Flow”, Third Edition, New York: McGraw-Hill; 2006.

[14] OpenFOAM, “*Time and data input/output control*”, [Online], Available: http://www.openfoam.org/docs/user/controlDict.php [Accessed: April 14, 2014]

[15] OpenFOAMwiki contibutors, (17 October 2012), Matrices in OpenFOAM, [Online], Available: http://openfoamwiki.net/index.php?title=OpenFOAM_guide/Matrices_in_OpenFOAM&oldid=12806

[16] Wikipedia contributors, (11 May 2014), Gauss-Seidel method, [Online], Available: http://en.wikipedia.org/w/index.php?title=Gauss–Seidel_method&oldid=608006220

[17] Wikipedia contributors, (2 May 2014), Amdahl's law, Xilinx, [Online], Available: http://en.wikipedia.org/w/index.php?title=Amdahl%27s_law&oldid=606808850

[18] Tested. “*Theoretical vs actual bandwidth – PCI Express and Thunderbolt*” tested.com. [Online]. Available: http://www.tested.com/tech/457440-theoretical-vs-actual-bandwidth-pci-express-and-thunderbolt/ [Accessed: June 2, 2014]

[19] Y. Saad, "Iterative Methods for Sparse Linear Systems", Second Edition. Phiadelphia: Society for Industrial and Applied Mathematics, 2003. [E-book] Available: books24x7.

[20] D.M. Strong, (2005, July). "*Iterative methods of solving Ax=b – Analysis of Jacobi and Gauss-seidel methods" MAA* [Online]. Available: http://www.maa.org/publications/periodicals/loci/joma/iterative-methods-for-solving-iaxi-ibi-analysis-of-jacobi-and-gauss-seidel-methods [Accessed: June 2, 2014]

[21] D. H. Jones, A. Powell, C-S. Bouganis and P. Y. K. Cheung, "GPU versus FPGA for high productivity computing," in *International Conference on Field Programmable Logic and Applications*, Italy, Milano, 31 August – 2 September, 2010, London, Imperial College London, 2010.

[22] Wikipedia contributors, (21 May 2014), Run-length encoding, [Online], Available: http://en.wikipedia.org/wiki/Run-length_encoding [Accessed: June 4, 2014]

[23] Vratis. "*What is SpeedIT?*" speedit.vratis.com. [Online]. Available: http://speedit.vratis.com/ [Accessed: June 2, 2014]

Appendix A: Code for the Model

Figure 20 is the finished *blockMeshDict:*

```
convertToMeters 1;

vertices
(
    (0 0 0)
    (1 0 0)
    (1 1 0)
    (0 1 0)
    (0 0 0.1)
    (1 0 0.1)
    (1 1 0.1)
    (0 1 0.1)
);
blocks
(
    hex (0 1 2 3 4 5 6 7) (100 100 1) simpleGrading (1 1 1)
);
edges
(
);
boundary
(
    topWall
    {
        type wall;
        faces
        (
            (3 7 6 2)
        );
    }
    fixedWalls
    {
        type empty;
        faces
        (
            (0 4 7 3)
            (2 6 5 1)
            (1 5 4 0)
        );
    }
    frontAndBack
    {
        type empty;
        faces
        (
            (0 3 2 1)
            (4 5 6 7)
        );
    }
);
```

Figure 20: Code from the finished blockMeshDict

Figure 21 specifies how the code in *changeDictionaryDoct* of Wall is structured.

```
dictionaryReplacement
{
    T
    {
        internalField   uniform 285;

        boundaryField
        {
            topWall
            {
                  type  fixedValue;
                  value uniform 273;
            }
            frontAndBack
            {
                  type  empty;
            }
            fixedWalls
            {
                  type  empty;
            }
            "wall_to_.*"
            {
                  type compressible::turbulentTemperatureCoupledBaffleMixed;
                  Tnbr         T;
                  kappa        solidThermo;
                  kappaName    none;
                  value        uniform 285;
            }
        }
    }
}
```

Figure 21: Code from changeDictionaryDict of Wall

The code of *changeDictionaryDict* of bottomWater is separated in three parts due to it is so long, Figure 22 is the first part.

```
dictionaryReplacement
{
    boundary
    {
        fixedWalls
        {
            type            zeroGradient;
        }
        frontAndBack
        {
            type      zeroGradient;
        }
    }
    U
    {
        internalField   uniform (0.0 0 0);

        boundaryField
        {
            fixedWalls
            {
                type            fixedValue;
                value           uniform ( 0 0 0 );
            }

            frontAndBack
            {
                type            fixedValue;
                value           uniform ( 0 0 0 );
            }

            "bottomWater_to_.*"
            {
                type            fixedValue;
                value           uniform (0 0 0);
            }
        }
    }
```

Figure 22: Part 1 of the code from changeDictionaryDict of bottomWater

Figure 23 is the second part of the *changeDictionaryDict* of bottomWater.

```
T
    {
        internalField   uniform 300;

        boundaryField
        {
            fixedWalls
            {
                  type  zeroGradient;
            }

            frontAndBack
            {
                  type  zeroGradient;
            }

            "bottomWater_to_.*"
            {
                  type compressible::turbulentTemperatureCoupledBaffleMixed;
                  Tnbr        T;
                  kappa       fluidThermo;
                  kappaName   none;
                  value       uniform 300;
            }
        }
    }
    epsilon
    {
        internalField   uniform 0.01;

        boundaryField
        {
            fixedWalls
            {
                type            zeroGradient;
            }

            "bottomWater_to_.*"
            {
                type            compressible::epsilonWallFunction;
                value           uniform 0.01;
            }
        }
    }
```

Figure 23: Part 2 of the code for changeDictionaryDict of bottomWater

Figure 24 is the third and final part of the *changeDictionaryDict* of bottomWater.

```
k
    {
        internalField   uniform 0.1;
        boundaryField
        {
            fixedWalls
            {
                type            zeroGradient;
            }
            "bottomWater_to_.*"
            {
                type            compressible::kqRWallFunction;
                value           uniform 0.1;
            }
        }
    }
    p_rgh
    {
        internalField   uniform 1e5;
        boundaryField
        {
            ".*"
            {
                type          fixedFluxPressure;
                value         uniform 1e5;
            }
            fixedWalls
            {
                type          zeroGradient;
            }
            frontAndBack
            {
                type          zeroGradient;
            }
        }
    }
    p
    {
        internalField   uniform 1e5;
        boundaryField
        {
            ".*"
            {
                type          calculated;
                value         uniform 1e5;
            }
        }
    }
}
```

Figure 24: Part 3 of the code for changeDictionaryDict for bottomWater

Figure 25 illustrates the code from the `topoSetDict` file.

```
actions
(
    // wall
    {
        name    wall;
        type    cellSet;
        action  new;
        source  boxToCell;
        sourceInfo
        {
            box (0 0.8 0 )(1 1 0.1);
        }
    }
    {
        name    wall;
        type    cellZoneSet;
        action  new;
        source  setToCellZone;
        sourceInfo
        {
            set wall;
        }
    }

    // bottomWater
    {
        name    bottomWater;
        type    cellSet;
        action  new;
        source  boxToCell;
        sourceInfo
        {
            box (0 0 0 )(1 0.8 0.1);
        }
    }
    {
        name    bottomWater;
        type    cellZoneSet;
        action  new;
        source  setToCellZone;
        sourceInfo
        {
            set bottomWater;
        }
    }

);
```

Figure 25: Code from the topoSetDict file

Figure 26 illustrates the code from the GaussSeidelSmoother.C file.

```
for (label sweep=0; sweep<nSweeps; sweep++)
    {
        bPrime = source;

        matrix_.initMatrixInterfaces
        (
            mBouCoeffs,
            interfaces_,
            psi,
            bPrime,
            cmpt
        );

        matrix_.updateMatrixInterfaces
        (
            mBouCoeffs,
            interfaces_,
            psi,
            bPrime,
            cmpt
        );

        register scalar psii;
        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
            psii = bPrimePtr[celli];

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

            // Finish psi for this cell
            psii /= diagPtr[celli];

            // Distribute the neighbour side using psi for this cell
            for (register label facei=fStart; facei<fEnd; facei++)
            {
                bPrimePtr[uPtr[facei]] -= lowerPtr[facei]*psii;
            }

            psiPtr[celli] = psii;

        }

    }
```

Figure 26: Code for GaussSeidelSmoother.C

Appendix B: Function names

The full function names for a mesh of (100 100 1):

GaussSeidelSmooter: Foam::GaussSeidelSmoother::smooth(Foam::word const&, Foam::Field<double>&, Foam::lduMatrix const&, Foam::Field<double> const&, Foam::FieldField<Foam::Field, double> const&, Foam::UPtrList<Foam::ldu

inv: Foam::inv(Foam::Field<Foam::Tensor<double> >&, Foam::UList<Foam::Tensor<double> > const&)

multiply: Foam::multiply(Foam::Field<double>&, Foam::UList<double> const&, Foam::UList<double> const&)

Amul: Foam::lduMatrix::Amul(Foam::Field<double>&, Foam::tmp<Foam::Field<double> > const&, Foam::FieldField<Foam::Field, double> const&, Foam::UPtrList<Foam::lduInterfaceField const> const&, unsigned char

List: Foam::List<double>::operator=(Foam::UList<double> const&)

Appendix C: Time Plan

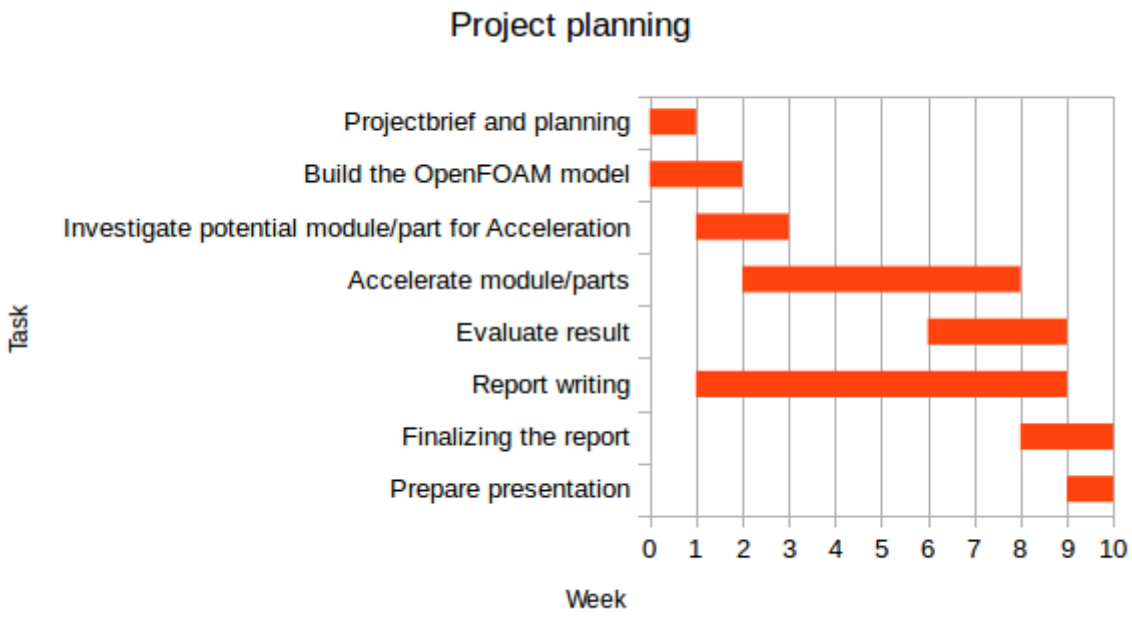

Figure 27: Gantt-scheme for the time plan