Delft University of Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft Institute of Applied Mathematics

Implementation of the BiCGSTAB Method for the Helmholtz Equation on a Maxeler Data Flow Machine.

A thesis submitted to the
Delft Institute of Applied Mathematics
in partial fulfilment of the requirements

for the degree

MASTER OF SCIENCE
in
APPLIED MATHEMATICS

by

Onno Leon Meijers
Delft, the Netherlands
Aug 2017

Copyright © 2017 by Onno Leon Meijers. All rights reserved.

MSc THESIS APPLIED MATHEMATICS

“Implementation of the BiCGSTAB Method for the Helmholtz Equation on a Maxeler Data Flow Machine.”

Onno Leon Meijers

Delft University of Technology

Daily supervisor

Prof.Dr.Ir. C. Vuik

Responsible professor

Prof.Dr.Ir. C. Vuik

Other thesis committee members

Prof.Dr.Ir. H.X. Lin

Prof.Dr.Ir. G.N. Gaydadjiev

Aug 2017

Delft, the Netherlands

Abstract

To get a clear picture of the earth crust the Helmholtz equation needs to be solved and a high wave number is needed in order to provide fine grain details. The Stabilised BiConjugate Gradient method needs to be implemented to solve the Helmholtz equation. This is done on a Maxeler data flow machine to improve calculation times. The algorithm is split into 3 parts and all of them use the same calculation kernel. An improvement in calculation time of 2.4 times faster than in the literature is achieved and a two times faster calculation is expected with more modern hardware. Maxeler's data flow machine shows great possibilities to decrease calculation times for high performance computing problems.

Contents

1 Introduction **11**

2 Introduction of the problem **13**
2.1 Helmholtz equation . . . 13
2.2 Boundary conditions . . . 15
2.3 Finite difference approximations . . . 16
2.4 Krylov subspace iterative methods . . . 17
2.5 Conjugate gradient method . . . 18
2.6 BiCGSTAB . . . 19

3 The Maxeler Data flow Machine **21**
3.1 FPGAs and their history . . . 21
3.2 Maxeler's Data flow Machine . . . 21
3.3 Data flow principle and Engine . . . 22
3.4 Flow diagram's different node types . . . 23
3.5 Implementation of an example on a Maxeler machine . . . 24
3.6 Ticks versus cycles . . . 28
3.7 The compiling process and programming . . . 30

4 Implementation info and background **33**
4.1 Maxeler machine . . . 33
4.1.1 CPU info . . . 33
4.1.2 Data flow engine info . . . 33
4.2 Implementation of the BiCGSTAB method on the CPU . . . 33
4.3 BiCGSTAB on the data flow machine . . . 34
4.4 Implementation differences and complications . . . 35
4.4.1 Multiple kernels . . . 35
4.4.2 Complex numbers . . . 36
4.4.3 Intermediate convergence test . . . 36
4.4.4 Last steps of algorithm . . . 36
4.4.5 Small improvements . . . 36
4.5 Preconditioning . . . 37
4.6 2D Parallelism . . . 38
4.7 Final algorithm . . . 39

5 Second order finite difference matrix-vector product **42**
5.1 Matrix-free operator versus matrix-vector multiplication . . . 42
5.1.1 Implementation details . . . 42
5.2 Tiling . . . 43
5.3 Experimental results of matrix-vector product . . . 44
5.3.1 Results for large systems . . . 45

6 Experimental results of the implementations **47**
6.1 Fast memory implementation . . . 47
6.1.1 Results of fmem implementation . . . 48
6.2 Split kernel implementation . . . 50
6.2.1 Split kernels results . . . 51
6.3 Parallel implementation . . . 52
6.3.1 Intermediate results . . . 52
6.4 Final real valued implementation . . . 53
6.5 Final complex valued implementation . . . 54
6.5.1 Comparison with results from the literature . . . 55
6.5.2 DFE resource usage . . . 55
6.6 Plots of the calculations . . . 56
6.7 Future performance estimation . . . 57

7 Conclusions **59**

8 Future research **59**

A Build-log of example **61**

B Ccode **63**

C Kernel Fast Memory **67**

List of abbreviations

BiCG	BiConjugate Gradient
BiCGSTAB	Stabilised BiCG
CG	Conjugate Gradient
CLB	Configurable Logic Block
CPU	Central Processing Unit
DFE	DataFlow Engine
EPROM	Erasable PROM
fmem	Fast memory of DFE
FPGA	Field Programmable Gate Array
GPU	Graphics Prossesing Unit
HPC	High Performance Computing
IO bound	Input Output bound
lmem	large memory of Data Flow Engine
LUT	LookUp Table
PLD	Programmable Logic Device
PROM	Programmable Read-Only Memory
SLiC interface	Simple Live CPU interface
SPD	Symmetric Positive Definite

1 Introduction

Astounding progress to solve three dimensional (3D) seismic problems has been made thanks to the availability of immense computing power. Seismic computations are done to get a clear picture of the earth crust. A clear picture of the earth crust can be useful for example earthquake predictions but also for example to predict oil reserves.

A prediction of the structure of layers within the earth crust is made. Then a calculation is made to compare this prediction with measurement data. Then a new prediction can be made and this process continues until the computations converge with the measurements. This thesis is limited to the computations. 3D simulations with a high wave number have to be performed in order to provide fine grain details. An iterative method is developed where the number of iterations scales linearly with the wave number. The required simulation times on the existing high-performance systems are too long. In order to do realistic simulations, new parallel hardware platforms could be used. The hardware devices used in this thesis are known as Field Programmable Gate Arrays (FPGAs) and are actually old devices used in a new way. Maxeler has developed a machine that uses FPGAs to solve large scientific problems. The machine is constructed in such a way that domain experts from different disciplines can use it without the need to understand the hardware design. The main building blocks of this machine are named Data Flow Engines (DFEs).

The purpose of this report is to present results of the BiCGSTAB method implementation for the Helmholtz equation on the Maxeler machine. The preconditioner is not yet implemented due to time constraints. However, the intention to implement a preconditioner has influenced the results of this project. For the BiCGSTAB algorithm an improvement in computation time was achieved compared to simulations with the Central Processing Unit (CPU) and the Graphics Processor Unit (GPU). A theoretical performance estimation was also derived for future DFE devices.

Firstly, the math problem is described in Section 2. Then the introduction to the Maxeler machine or DFE based on a FPGA including an example are presented in Section 3. Theoretical background of the implementation choices is presented in Section 4. In Section 5 the finite difference matrix vector multiplication is compared to the operator on the Maxeler machine. In Section 6 the results of the implementation are given. Sections 7, and 8 give the conclusions, and the recommendation on future research respectively.

2 Introduction of the problem

This section will derive the Helmholtz equation with the boundary conditions. Then the numerical approximation will be given together with the numerical solver.

2.1 Helmholtz equation

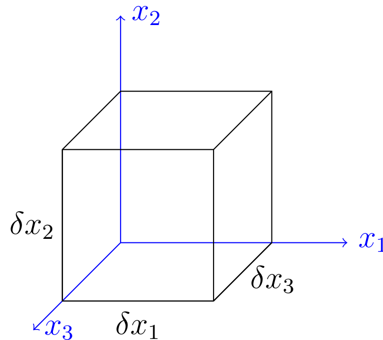


Figure 1: infinitesimally small element of volume V.

Suppose that there is an infinitesimally small element with volume V in a domain $\Omega \in \mathbb{R}^3$ as shown in Figure 1. Assuming zero deformation, then the spatial variation of the pressure $p = p(\mathbf{x}, t)$ on this element will generate a force F according to Newton's second law:

$$F = m\frac{\partial \mathbf{v}}{\partial t}, \tag{2.1}$$

with m the elements mass, $\mathbf{v} = \mathbf{v}(\mathbf{x}, t)$ the partial velocity, $\mathbf{x} = (x_1, x_2, x_3)$ and $F = -\left(\frac{\partial p}{\partial x_1}, \frac{\partial p}{\partial x_2}, \frac{\partial p}{\partial x_3}\right) V = -\nabla p V$. The operator ∇ is the gradient operator. Substituting F in Equation (2.1) and rewriting gives:

$$\nabla p = -\frac{m}{V}\frac{\partial \mathbf{v}}{\partial t} = -\rho_0 \frac{\partial \mathbf{v}}{\partial t}, \tag{2.2}$$

with ρ_0 the static density.

For solids, Hooke's law gives

$$\frac{\mathrm{d}p}{\mathrm{d}t} = -K\frac{\frac{\mathrm{d}V}{\mathrm{d}t}}{V}, \tag{2.3}$$

with K the compression modulus of the media. For small spatial variations we assume that the element holds it shape and stays a cube,

$$
\begin{aligned}
\frac{\frac{\mathrm{d}V}{\mathrm{d}t}}{V} &= \frac{\frac{\mathrm{d}\delta x_1}{\mathrm{d}t}}{\delta x_1} + \frac{\frac{\mathrm{d}\delta x_2}{\mathrm{d}t}}{\delta x_2} + \frac{\frac{\mathrm{d}\delta x_3}{\mathrm{d}t}}{\delta x_3}, \\
&= \frac{(v_1)_{x_1+\delta x_1} - (v_1)_{x_1}}{\delta x_1} + \frac{(v_2)_{x_2+\delta x_2} - (v_2)_{x_2}}{\delta x_2} + \frac{(v_3)_{x_3+\delta x_3} - (v_3)_{x_3}}{\delta x_3}, \\
&= \left(\frac{\partial v_1}{\partial x_1} + \frac{\partial v_2}{\partial x_2} + \frac{\partial v_3}{\partial x_3} \right), \\
&= (\nabla \cdot \mathbf{v}) .
\end{aligned} \tag{2.4}
$$

By using Equation (2.4) and since $\frac{\mathrm{d}p}{\mathrm{d}t} = \frac{\partial p}{\partial t}$, Equation (2.3) becomes

$$
(\nabla \cdot \mathbf{v}) = -\frac{1}{K} \frac{\partial p}{\partial t}. \tag{2.5}
$$

Applying the gradient operator to Equation (2.2) gives

$$
\nabla \cdot \left(-\frac{1}{\rho_0} \nabla p \right) = \frac{\partial}{\partial t} (\nabla \cdot \mathbf{v}) . \tag{2.6}
$$

Substitution of Equation (2.5) into Equation (2.6) with assumption that the gradient density ρ_0 is infinitesimally small results in

$$
\Delta p = \frac{1}{c^2} \frac{\partial^2 p}{\partial t^2}, \tag{2.7}
$$

which is the pressure wave equation for solids, with $\Delta \equiv \nabla^2$ the Laplace operator, and $c = \sqrt{\frac{K}{\rho_0}}$ the propagation speed of compressional waves in solids.

The concern is with the time-harmonic waves of time-dependent pressure of the form

$$
p(\mathbf{x}, t) = u(\mathbf{x}) \exp(-\mathbf{i} \omega_w t), \tag{2.8}
$$

where $\omega_w > 0$ and $\mathbf{i}$ denotes the angular frequency and the imaginary unit, respectively. Substituting Equation (2.8) into Equation (2.7) results in

$$
-\Delta u(\mathbf{x}) - k^2(\mathbf{x}) u(\mathbf{x}) =: \mathcal{A} u(\mathbf{x}) = 0, \tag{2.9}
$$

with $\mathcal{A}$ the Helmholtz operator. In Equation (2.9) k is the wave number, and $k(\mathbf{x}) = \frac{\omega_w}{c(\mathbf{x})}$. Because $\omega_w = 2\pi f$, where f is the wave frequency, it is found that $k(\mathbf{x}) = \frac{2\pi}{\lambda_w(\mathbf{x})}$, where $\lambda_w(\mathbf{x}) = \frac{c(\mathbf{x})}{f}$ is defined as the wavelength. Equation (2.9) is known as the Helmholtz equation for the pressure.

Introducing a source term with the assumption that it is time-harmonic, a more general formulation of the Helmholtz equation is obtained:

$$
\mathcal{A} u(\mathbf{x}) := -\Delta u(\mathbf{x}) - k^2(\mathbf{x}) u(\mathbf{x}) = g(\mathbf{x}), \tag{2.10}
$$

where $g(\mathbf{x})$ is the source term. The Helmholtz equation can be generalised even further taking into account the possibility of a barely attenuative medium. For this type of problem the Helmholtz equation becomes

$$\mathcal{A}u(\mathbf{x}) := -\Delta u(\mathbf{x}) - (1 - \alpha \mathbf{i})k^2(\mathbf{x})u(\mathbf{x}) = g(\mathbf{x}), \tag{2.11}$$

with $0 \leq \alpha \ll 1$ indicating the function of damping in the medium. In geophysical applications this damping can be set up to 5%. Note that $\alpha = 0$ gives the equation without damping, Equation (2.10).

2.2 Boundary conditions

For the Helmholtz equation proper boundary conditions are required to have a well-posed problem. A boundary condition at infinity can be derived by considering the physical situation at infinity. This situation can be viewed directly from the Helmholtz equation. Consider a domain Ω with a homogeneous medium and assume spherical symmetric waves propagating from a source or a scatterer in the domain. Then in most cases, close to the source/scatterer this assumption is easily violated; over there the waves are arbitrary and more complex than just spherical. However, we assume that at infinity these complex waves are disentangled and become spherical. Under this assumption, Equation (2.9) can be evaluated in a spherical coordinate system in which the Helmholtz equation transforms into

$$-(ru)'' - k^2(ru) = 0, \tag{2.12}$$

with a general solution of the form

$$u(r) = A\frac{\cos(kr)}{r} + B\frac{\sin(kr)}{r}. \tag{2.13}$$

Combining Equation (2.8) with Equation (2.13) results in

$$p(r,t) = A^* \frac{\exp(\mathbf{i}(kr - \omega_w t))}{r} + B^* \frac{\exp(-\mathbf{i}(kr - \omega_w t))}{r}. \tag{2.14}$$

In surfaces of constant phase, as $\omega_w t$ increases in time the first term on the right-hand side of Equation (2.14) describes the waves propagating away from the source/scatterer. The second term describes waves propagating inwards from infinity. If a region Ω is bounded by a spherical surface $\Gamma = \partial\Omega$, and contains a scatterer then the second term of the right-hand side of Equation (2.14) cannot be a physical solution. Therefore,

$$p(r,t) = A^* \frac{\exp(\mathbf{i}(kr - \omega_w t))}{r}. \tag{2.15}$$

Since $p(r,t)$ contains a factor r^{-1}, the amplitude of the wave must disappear at infinity. The vanishing condition ensuring $u(r) \to 0$ as $r \to \infty$ is given in [2]. Hence to have a well posed problem the following criterion is needed

$$\lim_{r\to\infty} (-u' - \mathbf{i}ku) \sim o(r^{-1}), \tag{2.16}$$

with "o" a Landau symbol, defined as

$$f_1(x) \sim o(f_2(x)) \Longrightarrow \frac{f_1(x)}{f_2(x)} \to 0.$$

Equation (2.16) is known as the Sommerfeld radiation condition for spherical coordinate systems [4, 3]. Alternately, Dirichlet boundary conditions can be used as a simplification.

2.3 Finite difference approximations

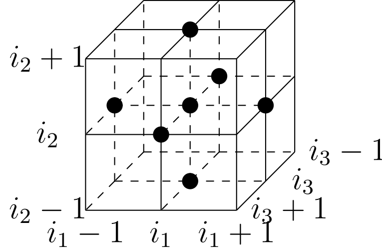


Figure 2: 3D finite difference 7-point stencil.

Let the sufficiently smooth Ω be discretised by an equidistant grid with grid size h. The discretised domain is denoted by Ω_h. The approximate solutions of the Helmholtz equation on Ω_h are computed. Considering the solution at the grid point $\mathbf{x} = (x_1, x_2, x_3) = \{(i_1 h, i_2 h, i_3 h) | i_1, i_2, i_3 = 0, 1, \ldots, N-1\}$ in Ω_h, with N the number of unknowns per coordinate direction. Introducing the standard lexicographical numbering and denoting the approximate solution $u(\mathbf{x}) = u(i_1 h, i_2 h, i_3 h)$ as u_{i_1,i_2,i_3}, see Figure 2, the central difference scheme for the second order differential term is

$$\frac{\partial^2 u}{\partial x_1^2} \approx \frac{1}{h^2}\left(u_{i_1+1,i_2,i_3} - 2u_{i_1,i_2,i_3} + u_{i_1-1,i_2,i_3}\right), \tag{2.17}$$

$$\frac{\partial^2 u}{\partial x_2^2} \approx \frac{1}{h^2}\left(u_{i_1,i_2+1,i_3} - 2u_{i_1,i_2,i_3} + u_{i_1,i_2-1,i_3}\right) \text{ and} \tag{2.18}$$

$$\frac{\partial^2 u}{\partial x_3^2} \approx \frac{1}{h^2}\left(u_{i_1,i_2,i_3+1} - 2u_{i_1,i_2,i_3} + u_{i_1,i_2,i_3-1}\right). \tag{2.19}$$

Equation (2.11) can now be approximated in Ω_h by

$$\begin{aligned} &-\frac{1}{h^2}\left(u_{i_1-1,i_2,i_3} + u_{i_1,i_2-1,i_3} + u_{i_1,i_2,i_3-1}\right. \\ &\left.- 6u_{i_1,i_2,i_3} + u_{i_1+1,i_2,i_3} + u_{i_1,i_2+1,i_3} + u_{i_1,i_2,i_3+1}\right) \\ &- (1-\alpha\mathbf{i})k^2 u_{i_1,i_2,i_3} = g_{i_1,i_2,i_3}, \text{ for } i_1, i_2, i_3 = 1, 2, \ldots, N. \end{aligned} \tag{2.20}$$

Equation (2.20) can be written in stencil notation as

$$\begin{aligned} &A_{h,7p} \widehat{=} \\ &-\frac{1}{h^2}\left(\begin{bmatrix} & 0 & \\ 0 & 1 & 0 \\ & 0 & \end{bmatrix}_{i_3-1} \begin{bmatrix} & 1 & \\ 1 & -6+(1-\alpha\mathbf{i})k^2h^2 & 1 \\ & 1 & \end{bmatrix}_{i_3} \begin{bmatrix} & 0 & \\ 0 & 1 & 0 \\ & 0 & \end{bmatrix}_{i_3+1} \right), \end{aligned} \tag{2.21}$$

with $h = \frac{1}{N}$ the grid size. For smooth solutions and uniform grids this approximation is of $O(h^2)$ accuracy.

Left to do are the boundary conditions given in equation (2.16). Those boundary conditions become

$$\left(-\frac{\partial}{\partial \eta} - \mathbf{i}k\right) u = 0, \tag{2.22}$$

where η is the outward unit normal component to the boundary. Note that in 2D for a 5-point stencil only the boundary conditions at the faces are needed. Next the derivation of the discretised boundary condition of the left face is presented with a one-sided scheme and for an equidistant grid.

Hence, for the left face, the outward unit normal component is then $-\frac{\mathrm{d}}{\mathrm{d}x}$ and the discretisation becomes

$$\begin{aligned}
\left(\frac{\mathrm{d}}{\mathrm{d}x} - \mathbf{i}k\right) u_{0,i_2,i_3} &= 0, \\
\frac{u_{1,i_2,i_3} - u_{0,i_2,i_3}}{h} - \mathbf{i}k u_{0,i_2,i_3} &= 0, \\
(1 + \mathbf{i}kh)\, u_{0,i_2,i_3} &= u_{1,i_2,i_3}, \\
u_{0,i_2,i_3} &= \frac{u_{1,i_2,i_3}}{1 + \mathbf{i}kh}.
\end{aligned} \tag{2.23}$$

Note that these were the intended boundary conditions. However, due to time constraints only the Dirichlet boundary conditions were used in this project. Then $\mathbf{u} = 0$ on the boundary. This results in a simple matrix and no corrections for the boundary are needed.

2.4 Krylov subspace iterative methods

In this section the method for solving the linear system will be explained for Equation (2.20) which is of the form

$$A\mathbf{u} = \mathbf{g}. \tag{2.24}$$

A is a square, complex valued and in general not a symmetric positive definite (SPD) matrix .

The Krylov subspace iteration methods are based on consecutive iterants in a Krylov subspace, i.e. a subspace of the form

$$\mathcal{K}^j(A; \mathbf{r}^o) = \operatorname{span}\{\mathbf{r}^0, A\mathbf{r}^0, \ldots, A^{j-1}\mathbf{r}^0\}, \tag{2.25}$$

where $\mathbf{r}^0 := \mathbf{g} - A\mathbf{u}^0$ is the initial residual, with $\mathbf{u}^0$ the initial solution. The idea of Krylov subspace methods can be explained as follows. For an initial solution $\mathbf{u}^0$, approximations $\mathbf{u}^j$ to the solution $\mathbf{u}$ are computed every step by iterants $\mathbf{u}^j$ of the form

$$\mathbf{u}^j \in \mathbf{u}^0 + \mathcal{K}^j(A; \mathbf{r}^0), \quad j > 1. \tag{2.26}$$

The Krylov subspace $\mathcal{K}^j$ is constructed by the basis $\mathbf{v}^1, \mathbf{v}^2, \ldots, \mathbf{v}^j$, where

$$V^j = [\mathbf{v}^1, \mathbf{v}^2, \ldots, \mathbf{v}^j] \in \mathcal{K}^j. \tag{2.27}$$

Combining Equation (2.26) with Equation (2.27) gives the expression

$$\mathbf{u}^j = \mathbf{u}^0 + \mathbf{V}^j \mathbf{y}^j, \tag{2.28}$$

for some $\mathbf{y}^j \in \mathbb{C}^N$. The residual now becomes

$$\begin{aligned}\mathbf{r}^j &= \mathbf{g} - A\mathbf{u}^j,\\ &= \mathbf{g} - A\mathbf{u}^0 + AV^j\mathbf{y}^j,\\ &= \mathbf{r}^0 - AV^j\mathbf{y}^j. \end{aligned} \tag{2.29}$$

From equation (2.29) is deduced that a Krylov subspace method relies on constructing a basis V^j and the vector $\mathbf{y}^j$. First the conjugate gradient (CG) method will be explained then the BiCGSTAB method.

2.5 Conjugate gradient method

For the CG method the matrix A needs to be SPD. Note that the matrix in the problem of this thesis does not have these properties in general. From the CG method the BiCGSTAB method will be derived. This method does not require a matrix to be SPD. Therefore, the CG method will be derived first.

In the CG method the new vector $\mathbf{u}^j \in \mathcal{K}^j(A;\mathbf{r}^0)$ is constructed such that $||\mathbf{u} - \mathbf{u}^j||_A$ is minimal, where $||\mathbf{u}||_A = (A\mathbf{u},\mathbf{u})^{\frac{1}{2}}$ and $(\mathbf{a},\mathbf{b})$ denotes the standard Hermitian inner-product. Therefore, $\mathbf{u}^{j+1}$ is expressed as

$$\mathbf{u}^{j+1} = \mathbf{u}^j + \alpha^j\mathbf{p}^j, \tag{2.30}$$

with $\mathbf{p}^j$ the search direction. The next residual $\mathbf{r}^{j+1}$ then becomes

$$\mathbf{r}^{j+1} = \mathbf{r}^j - \alpha^j A\mathbf{p}^j. \tag{2.31}$$

All $\mathbf{r}^j$'s need to be orthogonal to each other: $(\mathbf{r}^{j+1},\mathbf{r}^j) = 0$. Hence,

$$(\mathbf{r}^j - \alpha^j A\mathbf{p}^j, \mathbf{r}^j) = 0, \tag{2.32}$$

which gives

$$\alpha^j = \frac{(\mathbf{r}^j,\mathbf{r}^j)}{(A\mathbf{p}^j,\mathbf{r}^j)}. \tag{2.33}$$

The next search direction $\mathbf{p}^{j+1}$ is a linear combination of $\mathbf{r}^{j+1}$ and $\mathbf{p}^j$

$$\mathbf{p}^{j+1} = \mathbf{r}^{j+1} + \beta^j\mathbf{p}^j, \tag{2.34}$$

such that $\mathbf{p}^{j+1}$ is A-orthogonal to $\mathbf{p}^j$, i.e. $(A\mathbf{p}^{j+1},\mathbf{p}^j) = 0$. Then the denominator in (2.33) can be written as $(A\mathbf{p}^j, \mathbf{p}^j - \beta^{j-1}\mathbf{p}^{j-1}) = (A\mathbf{p}^j,\mathbf{p}^j)$. Also

$$\begin{aligned}(A\mathbf{p}^{j+1},\mathbf{p}^j) &= 0,\\ \Rightarrow (A(\mathbf{r}^{j+1} + \beta^j\mathbf{p}^j),\mathbf{p}^j) &= 0,\\ \Rightarrow (\beta^j A\mathbf{p}^j,\mathbf{p}^j) &= -(A\mathbf{r}^{j+1},\mathbf{p}^j),\\ \Rightarrow \beta^j &= -\frac{(A\mathbf{r}^{j+1},\mathbf{p}^j)}{(A\mathbf{p}^j,\mathbf{p}^j)}. \end{aligned} \tag{2.35}$$

Symmetry of A and orthogonality of the $\mathbf{r}^j$'s results in

$$\begin{aligned}\beta^j &= -\frac{(A\mathbf{p}^j, \mathbf{r}^{j+1})}{(A\mathbf{p}^j, \mathbf{p}^j)}, \\ &= -\frac{(\frac{\mathbf{r}^j - \mathbf{r}^{j+1}}{\alpha^j}, \mathbf{r}^{j+1})}{(A\mathbf{p}^j, \mathbf{p}^j)}, \\ &= \frac{1}{\alpha^j}\frac{(\mathbf{r}^{j+1}, \mathbf{r}^{j+1})}{(A\mathbf{p}^j, \mathbf{p}^j)}, \\ &= \frac{(A\mathbf{p}^j, \mathbf{p}^j)}{(\mathbf{r}^j, \mathbf{r}^j)}\frac{(\mathbf{r}^{j+1}, \mathbf{r}^{j+1})}{(A\mathbf{p}^j, \mathbf{p}^j)}, \\ &= \frac{(\mathbf{r}^{j+1}, \mathbf{r}^{j+1})}{(\mathbf{r}^j, \mathbf{r}^j)}. \end{aligned} \tag{2.36}$$

The summary of the CG method is presented in Algorithm 2.1.

Algorithm 2.1 CG.

1: Set initial guess: $\mathbf{u}^0$. Compute $\mathbf{r}^0 = \mathbf{g} - A\mathbf{u}^0$. Set $\mathbf{p}^0 = \mathbf{r}^0$.
2: **for** $k = 0, 1, \ldots$ **do**
3: $\quad \alpha^k = \frac{(\mathbf{r}^k, \mathbf{r}^k)}{(A\mathbf{p}^k, \mathbf{p}^k)}$.
4: $\quad \mathbf{u}^{k+1} = \mathbf{u}^k + \alpha^k \mathbf{p}^k$. If accurate then quit.
5: $\quad \mathbf{r}^{k+1} = \mathbf{r}^k - \alpha^k A\mathbf{p}^k$.
6: $\quad \beta^k = \frac{(\mathbf{r}^{k+1}, \mathbf{r}^{k+1})}{(\mathbf{r}^k, \mathbf{r}^k)}$.
7: $\quad \mathbf{p}^{k+1} = \mathbf{r}^{k+1} + \beta^k \mathbf{p}^k$.
8: **end for**

Algorithm 2.1 has the good properties that it only requires short recurrences and only one matrix-vector multiplication and only a few vector updates have to be calculated per iteration. However, this algorithm may not converge for the Helmholtz equation because A is not guaranteed SPD. Therefore, the BiCGSTAB method will be presented.

2.6 BiCGSTAB

For non-symmetric or non-SPD matrices an iterative method is developed based on the bi-Lanczos method. BiCGSTAB [8] is based on the following observation. Iterates $\mathbf{u}^i$ are generated so that $\mathbf{r}^i = \tilde{P}_i(A)P_i(A)\mathbf{r}^0$ with an ith degree polynomial $\tilde{P}_i(A)$. One possibility is to take $\tilde{P}_i(A)$ as a polynomial in the form

$$Q_i(x) = (1 - \omega_1 x)(1 - \omega_2 x)(1 - \omega_i x), \tag{2.37}$$

and to select suitable constants $\omega_i \in \mathbb{R}$. An almost trivial recurrence relation for the Q_i's is a result of this expression. In BiCGSTAB, ω_i in the ith iteration step is chosen to minimise $\mathbf{r}^i$, with respect to ω_i, for residuals that can be written as $\mathbf{r}^i = Q_i(A)P_i(A)\mathbf{r}^0$ [10].

The preconditioned BiCGSTAB method for solving the linear system $A\mathbf{u} = \mathbf{b}$, with preconditioner M is presented in Algorithm 2.2.[10]

Algorithm 2.2 BiCGSTAB.

1: Set initial guess: $\mathbf{u}^0$. Compute $\mathbf{r}^0 = \mathbf{g} - A\mathbf{u}^0$;
2: $\bar{\mathbf{r}}^0$ is an arbitrary vector, such that $(\mathbf{r}^0, \bar{\mathbf{r}}^0) \neq 0$, e.g. $\bar{\mathbf{r}}^0 = \mathbf{r}^0$;
3: $\rho_{-1} = \alpha_{-1} = \omega_{-1} = 1$;
4: $\mathbf{v}^{-1} = \mathbf{p}^{-1} = \mathbf{0}$;
5: **for** $i = 0, 1, 2, \ldots$ **do**
6: $\quad \rho_i = (\bar{\mathbf{r}}^0, \mathbf{r}^i)$; $\beta_{i-1} = \frac{\rho_i}{\rho_{i-1}} \frac{\alpha_{i-1}}{\omega_{i-1}}$;
7: $\quad \mathbf{p}^i = \mathbf{r}^i + \beta_{i-1}(\mathbf{p}^{i-1} - \omega_{i-1}\mathbf{v}^{i-1})$;
8: $\quad \hat{\mathbf{p}} = M^{-1}\mathbf{p}^i$;
9: $\quad \mathbf{v}^i = A\hat{\mathbf{p}}$;
10: $\quad \alpha_i = \frac{\rho_i}{(\bar{\mathbf{r}}^0, \mathbf{v}^i)}$;
11: $\quad \mathbf{s} = \mathbf{r}^i - \alpha_i \mathbf{v}^i$;
12: $\quad$ **if** $||\mathbf{s}||$ small enough **then**
13: $\quad\quad \mathbf{u}^{i+1} = \mathbf{u}^i + \alpha_i \hat{\mathbf{p}}$; quit;
14: $\quad$ **end if**
15: $\quad \mathbf{z} = M^{-1}\mathbf{s}$;
16: $\quad \mathbf{t} = A\mathbf{z}$;
17: $\quad \omega_i = \frac{(\mathbf{t}, \mathbf{s})}{(\mathbf{t}, \mathbf{t})}$;
18: $\quad \mathbf{u}^{i+1} = \mathbf{u}^i + \alpha_i \hat{\mathbf{p}} + \omega_i \mathbf{z}$;
19: $\quad$ **if** $\mathbf{u}^{i+1}$ is accurate enough **then**
20: $\quad\quad$ quit;
21: $\quad$ **end if**
22: $\quad \mathbf{r}^{i+1} = \mathbf{s} - \omega_i \mathbf{t}$;
23: **end for**

The matrix M in Algorithm 2.2 represents the preconditioning matrix and the way of preconditioning from [8]. This algorithm in fact carries out the BiCGSTAB procedure for the explicitly post-conditioned linear system

$$AM^{-1}\mathbf{y} = \mathbf{g}. \tag{2.38}$$

However,the vectors $\mathbf{y}^i$ have been transformed back to the vectors $\mathbf{u}^i$ corresponding to the original system $A\mathbf{u} = \mathbf{g}$. This solver will be used for our problem due to the spectral properties of our problem.[9]

3 The Maxeler Data flow Machine

This section presents the history and the background of Field Programmable Gate Arrays (FPGAs). Second, the data flow principle is given. Third, the flow diagram node types are presented. An example of an implementation in the Maxeler machine is given. Then the difference is explained between a tick and a cycle. Finally, the compiling process is described from the users point of view.

3.1 FPGAs and their history

FPGAs are hardware devices containing a reconfigurable fabric of arithmetic units that can be organized in arbitrary configurations. This means that the same hardware device can be repurposed for different calculations. This will make them cheaper than custom chip implementations and they are also updatable if they are still only used for one purpose.

The FPGA industry started from Programmable Read-Only Memory (PROM) and Programmable Logic Devices (PLDs). PROMs and PLDs were both programmable. However, the programmable logic was hard-wired between logic gates.

Altera introduced the EP300 in 1984. This was the industry's first reprogrammable logic device. If users would shine an UV lamp through a quartz window in the package on the die, then it erased the Erasable PROM (EPROM) cells that held the device configuration. Hence, after this the chip could be programmed again. It seemed like a minor convenience, however it proved to be a major influence in the industry.[1]

Xilinx introduced the XC2064 in 1985. This was the first commercially available FPGA . It had programmable gates and programmable interconnects between gates. It was the beginning of a new technology and market. It had 64 Configurable Logic Blocks (CLBs), with two three-input lookup tables (LUTs)[7]. A LUT stores the results of an expensive calculation in an array and connects an input of the calculation to a precalculated output.

In the nineties, FPGAs developed both in their sophistication and production volumes. FPGAs were primarily used in telecommunications and networking. By the end of the decade, FPGAs found their way into consumer, automotive, and industrial applications.

Most High Performance Computing (HPC) applications in Scientific Computing implemented on CPUs and GPUs are memory bounded. This means that the computer is faster in calculating compared to the time it has to wait for data from memory to arrive at the processors. Therefore the Maxeler Data Flow Engines (DFEs) offer a new way of dealing with this problem by benefiting from the streaming execution model.

3.2 Maxeler's Data flow Machine

This section explains the separation between a Maxeler's machine, a FPGA and a DFE. Maxeler has built their own data flow computer that has FPGAs integrated in them, this will be called the Maxeler data flow machine. The basic building blocks are called data flow Engines (DFEs) because the analogy with the steam engines at the beginning of the industrial revolution and their ability to implementation large-scale, high-throughput

[1] https://www.altera.com/solutions/technology/system-design/articles/_2013/in-the-beginning.html

Figure 3: A MaxWorkstation.

data flow accelerators directly in hardware. Maxeler develops software for their data flow technology so that a wider community of domain experts can use them. With this the users do not have to know about the mapping details on the hardware device. The users only have to understand the data flow programming and execution models. These principles will be explained in Section 3.3 and an example will be given in Section 3.5.

To introduce programming on the Maxeler machine a few concepts will be explained. The environment to program in is called MaxIDE. It is the Eclipse[2] IDE for Maxeler which is an open source environment to develop for example Java software. The CPU-side is programmed in for example C. On the DFE side, the extension of the file names are .maxj and this is similar to Java (Maxeler Java).

3.3 Data flow principle and Engine

In this section a data flow machine will be explained.

On a CPU, in a language like Matlab, an array is declared and then an operator on the array is defined and used. In C, for-loops need to be built and then for each step a calculation is defined.

In a data flow engine the operations on each unit are defined. Instead of building for-loops, the data flows through the operations constituting the computation kernel graph.

In a data flow engine a kernel is made to define all operations per input element sets. The input vector is sent from the memory through the kernel. In a kernel a large number of consecutive operations are performed systolically. All arithmetic and logic operations are done on the available arithmetic units and then the final results flow back to the memory.

Of course, not all operators can be done consecutively. A simple multiplication and addition can be done after each other. However, other operations require storage. For example, take the average over every two consecutive elements, then there are 2 possibilities. First, the vector can be split before it enters the kernel and added by having two

[2]eclipse.org

streams into the kernel. Second, one unit is stored each time, and then added to the next one to be divided by 2.

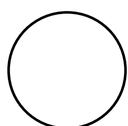
The kernel holds the information whether or not it accepts data. For example, in the average over two consecutive elements with storage, for each 2 data elements in, 1 flows out. Therefore, for the example, a counter is needed that swaps between true and false for each tick and set as boolean for the outflow.

A tick is when the output is calculated for an input. It is different from a CPU clock cycle and will be explained in Section 3.5. However, the FPGA also has cycles. For example, when the first elements of the two streams flow into one kernel, they need to be multiplied with a different constant and then added. In the first cycle, an element of each of the flows will be multiplied by the respective constant. Then a cycle happens and every element is pushed a step further. The two elements are only then added. In the next cycle, the new element flows out. The second element of the flows are following the first ones. Hence, when the first elements are pushed to the addition, the second elements are pushed into the multiplication by a constant. This will happen during the whole flow, until there are no more elements. However, still 2 cycles need to happen for the last elements of the vector to flow through the operations and into the memory.

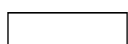
The basics of a kernel are mentioned above, but the data streams need to be managed and that is done by the software module describing the "manager". So the kernel has inflows and constants that need to be set, such as the weight for the weighted average. This constant could change every time the kernel is invoked but not during the process and the manager sets this constant in the kernel. The manager connects the kernel with the memory in this case. The manager needs to know how many bits will flow from which location. Note that when every result is not taken like in the weighted average example, the outflow connects the units that do flow.

3.4 Flow diagram's different node types

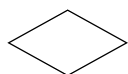
In this section the different nodes of flow diagrams will be presented. A node is an operation that needs to be executed in the kernel. Hence, it contains the addition and or multiplication operators, but also contains the in- and outflow operators. Following diagrams can give an easy representation of the implementation. The data will flow through these nodes. All these operation will be applied onto the data and the result will flow out.



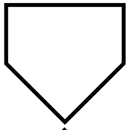
Computation nodes perform arithmetic and logic operations(e.g., +,*,<,&) as well as type casts to convert between the floating point, fixed point, and integer variables. Usually two streams flow in this node, however in the graph, those flows might come together earlier above the node. The type of operator will be written in the middle.



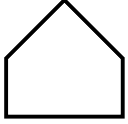
Value nodes provide parameters which are either constant or set by the CPU application on run-time. The name of the constant stands in the middle.



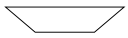
Stream offsets allows access to past and future elements of data streams. The amount of offset will be written in the middle.



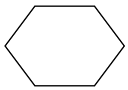
Input node connects data streams between the Kernel and Manager. The name of the input variable is in the middle.



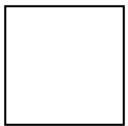
Output node is the same as the Input node excepts the data flows out instead of in. The picture is an upside down input node.



Multiplexer (mux) nodes for taking decisions. A boolean on the left decides which of the two flows from the top it takes.



Counter to keep track of individual actions on the elements. The properties will flow into the counter.



The true or false square. Everything that is build is always active. However, the calculation is not always needed for the algorithm, because the kernel is used a number of times for different calculations. This square is filled with a part number in the middle for when it is true. It will stand on the left of the in- output and the multiplexer nodes. This determines which results are used and connected to the memory.

3.5 Implementation of an example on a Maxeler machine

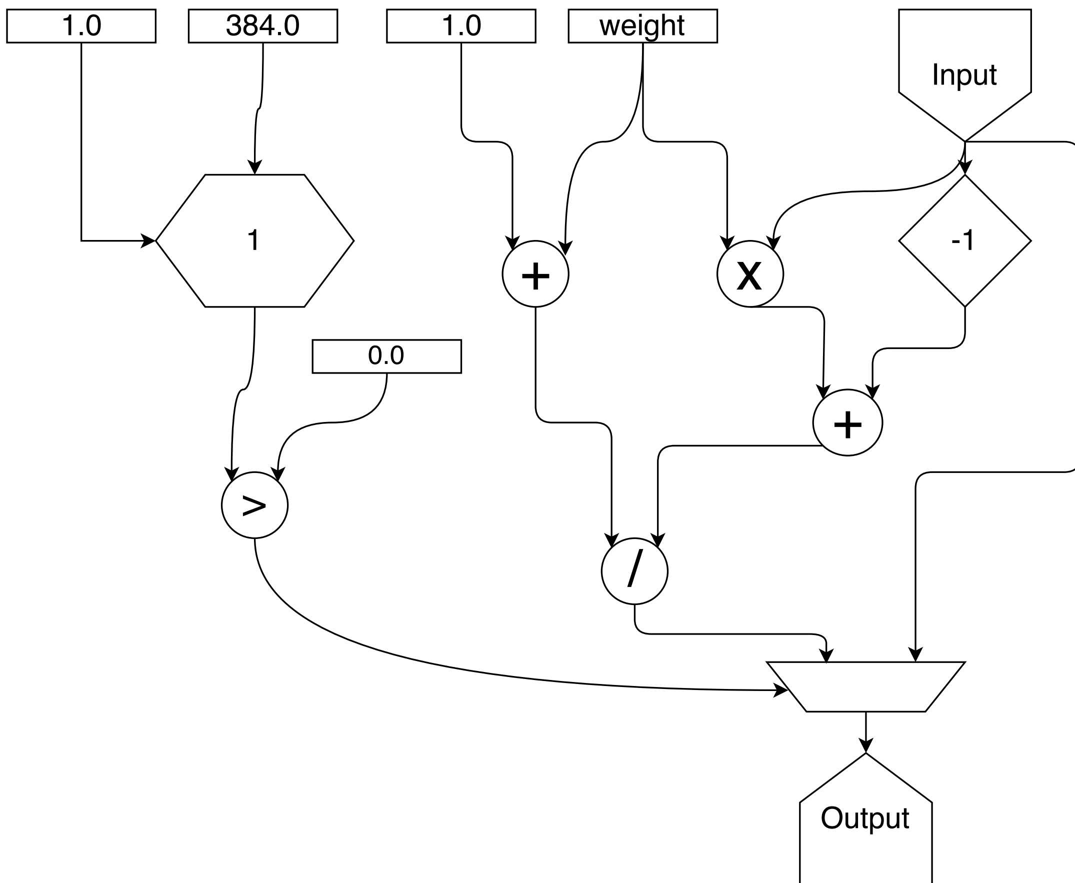


Figure 4: Flow diagram of the example.

This section will explain how to program a small example on the Maxeler machine. This will show all the basics from the manual[1] needed to implement the Helmholtz equation.

For this example, the weighted average over 2 elements will be calculated. Figure 4 shows the flow diagram of the example. In C the calculation step is the following for-loop:

```
for(int i=1;i<N;i++){
  result[i] =  (weight*input[i]+input[i-1])/(weight+1);
}
```

The example is explained from a kernel that does nothing up to this small example. Note that there is also a manager needed which controls the flow and that will be explained after. However, keep in mind that there is also a flow of bits into this kernel. The following code represents the kernel.

```
package movingaverageoftwo;

import com.Maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.Maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
import com.Maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;

class MovingAverageOfTwoKernel extends Kernel {
  MovingAverageOfTwoKernel(KernelParameters parameters) {
    super(parameters);

        <USER CODE GOES HERE>
  }
}
```

Each line will be explained separately. The first line in the listing above defines the package and is used so that all functions defined in these files can be used by the other files within the package. The 3 imports are for the class objects. Then, there is Java code defining the class that extends the kernel class, which is inheriting those properties and functions. In the next line the constructor is defined with the incoming parameters. By the super statement, the parameters inherit their properties from above. These were all the constructions around the calculations. The user code with the calculations will be defined next.

```
DFEVar input = io.input("input", dfeFloat(8, 24));
io.output("output", input, dfeFloat(8, 24));
```

This kernel links the input in a DFEVar. DFEVar is the class of variables that can take any arbitrary data type and size. This includes all the primitive instances such as single and double precision. For example, when floating point data is considered, users can choose the amount of bits that will be used for the float exponent and for the mantissa. The link looks like it is stored as in a C-program. However, it is used to link the input to other places as in this case when it directly links the input to the output.

The input needs the stream-name, which is ”input” in this case, so that it can be guided from the memory to the kernel. This is controlled in the manager and will be explained later. The input needs the type of the element that is sent, which is in this case single precision, IEEE floating point(dfeFloat(8,24)). DfeFloat is the float class for

a floating point number of the DFE with exponent and mantissa. Then the kernel knows that for each tick, it needs to take 32 bits in the kernel and do dfeFloat(8,24) operations on those 32 bits. Afterwards, the input needs to be send back with a stream-name, the variable and the data type.

Next, the result needs to be multiplied by a constant that is set by the CPU.

```
DFEVar weight = io.scalarInput("weight",dfeFloat(8,24));
DFEVar input = io.input("input", dfeFloat(8, 24));
DFEVar result = weight*input;
io.output("output", result, dfeFloat(8, 24));
```

The first line gets the input during runtime. This means that when the kernel is called the CPU sets the constant. The kernel is given the type of the constant and multiplies it with the input. Note that the result is a link again as explained earlier.

Next, the previous element is needed. For this purpose the function stream.offset is used. This function needs two arguments; the stream name and the signed integer of how big the offset must be. If the offset is negative, then the previous data element is used.

```
DFEVar weight = io.scalarInput("weight",dfeFloat(8,24));
DFEVar input = io.input("input", dfeFloat(8, 24));
DFEVar prev = stream.offset(input, -1);
DFEVar result = (prev + weight*input)/(weight+1);
io.output("output", result, dfeFloat(8, 24));
```

The stream.offset uses 1 storage element in this case. Each tick it stores the value of "input" and gives it on the next tick while storing the input for this tick. If the number is larger it stores more inputs and provides the required one accordingly. This all is done for the user by the software. Finally the average is calculated. Note that for the first element of the stream, stream.offset is not defined and it does not return a good value.

Therefore, the boundary term needs to be considered. For this case, a counter could be used to know if in this tick something has to happen. In this case it is only in the first tick and there can be other ways to note this. But for the example it will be used.

```
    Params params = control.count.makeParams(9).withMax(384);
    Counter counter = control.count.makeCounter(params);
```

A counter is built up from 2 steps. First, the parameters of the counter are defined. The first argument is the amount of bits needed to store the counter. In this case the number of bits is 9 so the counter can count from 0 up to $2^9 - 1$. The properties can then be added as the maximum which is 384 in this case. Other counter options are explained in the DFE programming manual[1]. The counter is defined with the parameters. Note that this maximum is not used, it is just an example.

The definition of the previous argument needs to be zero when the counter is zero. This will be done using the ternary operator that implements a multiplexer. ?4

```
    DFEVar result = (counter.getCount()>0 ? (prev + weight*input)/(
       weight+1):input);
```

Hence, when the counter is larger than zero the result is taken. For the boundary, only the input is taken, which is the average of one point. Note that for an alternative method a zero constant for the previous vector could be taken, but the result also needs a multiplexer for the boundary case. Hence, this is the more efficient way.

For convenience the data type is also stored so that the code is more efficient and is better presented. Note that DFEType is also imported into the kernel. This can be seen in Listing 1. However this is suggested by MaxIDE automatically so the user does not have to worry about that.

```
DFEType singleType   = dfeFloat(8, 24);
```

All this, will result in the kernel to solve the weighted average example, see Listing 1.

Listing 1: Kernel of the example.

```
package movingaverageoftwo;

import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.Count.Counter;
import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.Count.Params;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEType;
import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;

public class MovingAverageOfTwoKernel extends Kernel {
  private static final DFEType singleType = dfeFloat(8, 24);
  MovingAverageOfTwoKernel(KernelParameters parameters) {
    super(parameters);
    Params params = control.count.makeParams(9).withMax(384);
    Counter counter = control.count.makeCounter(params);
    DFEVar weight = io.scalarInput("weight",singl);
    DFEVar input = io.input("input", singleType);
    DFEVar prev = stream.offset(input, -1);
    DFEVar result =  (counter.getCount()>0 ? (prev + weight*input)/(weight+1):input);
    io.output("output", result, singleType);
  }
}
```

The kernel is built and it expects streams with the proper stream names. Those streams are provided and orchestrated by the manager. The simplest manager will be used for this example. This manager will connect all the streams and the constants with the CPU.

Listing 2: Manager of the example.

```
package movingaverageoftwo;

import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
import com.maxeler.maxcompiler.v2.managers.standard.Manager;
import com.maxeler.maxcompiler.v2.managers.standard.Manager.IOType;

public class MovingAverageOfTwoManager {
  public static void main(String[] args) {
    MovingAverageOfTwoEngineParameters params = new
        MovingAverageOfTwoEngineParameters(args);
    Manager manager = new Manager(params);
    Kernel kernel   = new MovingAverageOfTwoKernel(manager.
        makeKernelParameters("MovingAverageOfTwoKernel"));
    manager.setKernel(kernel);
    manager.setIO(IOType.ALL_CPU);
    manager.createSLiCinterface();
    manager.build();
  }
```

```
}
```

In Listing 2 the manager of the weighted average is presented. As in the kernel, the package and imports of the needed classes are given in the first few lines. In the main, the parameters are set and put in a new manager. Then the kernel is built and added to the manager. Then the data streams are defined by "setIO". In this case all streams are from and back to the CPU. This is done with the call:

```
    manager.setIO(IOType.ALL_CPU);
```

Note that the DFE also has a large memory. However to keep the example simple the CPU main memory is used.

The Simple Live CPU(SLiC) interface creates the interface so that CPU can call the functions programmed on the DFE. Using the Basic Static SLiC interface level is the simplest method, yet adequate for this example. This creates the function that the CPU's C program calls to perform the computation on the DFE. Finally, the function is built. Note that when a new project is built, the user can start with the kernel and manager templates predefined when building a MaxCompiler Project in MaxIDE.

Listing 3 displays the CPU code that builds the vector and calls the DFE function.

Listing 3: Example in C.

```
#include "Maxfiles.h"

int main(void){
        const int size = 384;
        int weight = 3;
        int sizeBytes = size* sizeof(float);
        float *x = malloc(sizeBytes);
        float *s = malloc(sizeBytes);

        for(int i = 0; i<size; ++i) {
                x[i]= random();
        }
        MovingAverageOfTwo(size, weight, x, s);
        return 0;
}
```

Maxfiles.h includes the functions prototypes defined in the manager. First, the weight and size are set in the main part. The memory is allocated using the malloc function. Then some random numbers are generated. The function executed on the DFE is

```
        MovingAverageOfTwo(size, weight, x, s);
```

After that, there can be some actions on the result **s**. However, the example will not be further elaborated.

3.6 Ticks versus cycles

This section will explain the difference between a tick and a cycle with the weighted average of Section 3.5 as the example. Figure 4 shows the data flow structure of the

example in the previous subsection. As a reminder, a tick is the result for an input. A cycle is what happens when the device sends every element one place further.

A flow needs to have at least 384 bits for the PCIe interface. To keep the example small, 6 elements of double precision will be used. Hence, the type will be dfeFloat(11,53). Six times 64 bits makes a full burst of 384 bits. The input vector is chosen to be {0.1,2.1,1.3,4.6,0.5,3.8}.

Assume that there is a result after each operator in Figure 4. Each of these points have a heading in the tables except for the addition of the constant plus one. Since its value will be the same for the entire run. Figure 4 shows this step for clarity because the example was built up this way. However, because it is always the same, it is more efficient to calculate it on the CPU and send as a second constant.

tick	counter	input	multiplication	previous	addition on the right	divide	multiplexer	output
1	0	0.1	0.2	?	?	?	0.1	0.1
2	1	2.1	4.2	0.1	4.3	1.43	1.43	1.43
3	2	1.3	2.6	2.1	4.7	1.57	1.57	1.57
4	3	4.6	9.2	1.3	10.5	3.5	3.5	3.5
5	4	0.5	1.0	4.6	5.6	1.87	1.87	1.87
6	5	3.8	7.6	0.5	8.1	2.7	2.7	2.7

Table 1: Table of tick diagram

cycle	counter	input	multiplication	previous	addition	divide	multiplexer	output
1	0	0.1	?	?	?	?	?	?
2	0	2.1	0.2	?	?	?	?	?
3	0	1.3	4.2	0.1	?	?	?	?
4	0	4.6	2.6	2.1	4.3	?	?	?
5	0	0.5	9.2	1.3	4.7	1.43	0.10	0.10
6	1	3.8	1.0	4.6	10.5	1.57	1.43	1.43
7	2	?	7.6	0.5	5.6	3.50	1.57	1.57
8	3	?	?	3.8	8.1	1.87	3.50	3.50
9	4	?	?	?	?	2.70	1.87	1.87
10	5	?	?	?	?	?	2.70	2.70

Table 2: Table of cycle diagram

Table 1 shows the results for each tick. Hence, for each input the result is calculated. Therefore, for tick 1 the input is 0. The value of the previous input is unknown and it could be anything. That value is never used due to the multiplexer in the end. The others are just the calculations for the inputs. Hence, for tick 3 the addition of the previous element, which is 2.1, plus 2 times this input, which is 2.6, is 4.7.

Table 2 shows the results for cycles and what the value behind each operation is. On the first cycle the first data arrives at the input. It gets multiplied by the constant. In the next cycle it gets added to the previous constant which is unknown for the first data. It is divided by the other calculated constant. Finally, the multiplexer chooses the result.

From Table 2, it can be learned that storage is needed. Because there is a direct flow from the input into the previous and the multiplexer. However, the table shows that they are not directly needed. Hence, they have to be stored until they are called upon. For the previous operator, there actually needs to be a buffer to store an element for 2 cycles. It is in order to align with the input and be ready for the addition at the same time as the multiplication is done. For the addition this can be done by storing two elements. The multiplexer needs to store 4 elements to align with the input.

Note that the output column is not written a tick later. This is because the output is after the multiplexer. Then the output is buffered in a batch until enough bits are stored to send a full batch to the memory or another location.

3.7 The compiling process and programming

In this section the compiling process, which are the main steps that the user has to do and can see when he/she is programming, will be explained in short. An example of the build-log will be presented in Appendix A. A new MaxCompiler Project will be made in MaxIDE, which is the editor. This will give all the necessary surrounding for a project, like the build file.

The programmer begins by changing at least the kernel, manager, and the CPU-side, see Section 3.5 for an example. With only these three, a program can be run on the CPU while using the DFE as an accelerator. The programming language on the CPU-side will be assumed C. However, other interfaces are possible such as Matlab, python, R and more.

After the program is built, the user can use simulation to check whether it has mistakes or not. These simulations are done on the CPU and of course they take a lot of computer power, so the input vectors should be kept small.

When the program is verified, it can be built for the DFE. The engine compiles and builds an executable program for the C-code to call. The compilation process consists of 12 phases as shown in Appendix A after the date in the middle of the lines. The first 6 steps are the preparation and will result in a preliminary report. It will show the expected usage of the resources of the device. More explanation will be given later.

The real place and route of the data flow graph on the reconfigurable device will be done next. This will usually take a very long time. In step 8 the tool chain will run cost tables. It places the individual operations on specific physical locations on the device. If a cost table has a score of more than zero it means that the route it tried to make did not work out. The physical placement reached the end. However, some calculations did not fit on the device. Multiple cost tables can be run in parallel to save on design time.

When a cost table with score zero is found, all cost tables will stop and the last few steps will be executed. These steps are building reports and at last they build the Maxfile. These reports are presenting the timings and the final resource usage. In this, the user can see how much of the available compute and memory resources he/she is using. This can lead to instantiating more parallel paths if there are sufficient resources left. The Maxfile is the executable program that can be called by the CPU-program. The Maxfile contains all the physical locations of the DFE device. Hence, it contains every reconfigurable connection and buffer such that are used such that the calculation can be done correctly.

Finally, the C-file will be compiled as normal. The program is ready to run and the intended calculations can be performed on the DFE.

4 Implementation info and background

This section presents the theoretical background of the implementation and problems to be considered when implementing the BiCGSTAB method on the DFE for the discretised Helmholtz equation. First the implementation on the CPU is given as the benchmark. Then the idea is presented and after that the differences with the original idea are given. Finally, the optimized algorithm is explained.

4.1 Maxeler machine

4.1.1 CPU info

vendor id	:	GenuineIntel
CPU family	:	6
model	:	42
model name	:	Intel(R) Core(TM) i7-2600S CPU @ 2.80GHz
stepping	:	7
CPU MHz	:	1600.000
cache size	:	8192 KB

The computer has 8 cores. However, only one core will be used during this project for the calculations.

4.1.2 Data flow engine info

frequency	:	0-200 MHz
generation	:	max3
compiler version	:	2013.2.2m
lmem capacity	:	48GB
communication speed	:	
between CPU and DFE	:	3000 MB/s
lmem bandwidth on DFE	:	38 GB/s

4.2 Implementation of the BiCGSTAB method on the CPU

This section explains the implementation of the BiCGSTAB method on the CPU that is used to check the results. It is also used to compare the speedup of the implementations on the DFE.

The code of the implementation of the algorithm on the CPU is provided in Appendix B. The algorithm is using for-loops to address the DFE implementation stage. Hence, for each step of the algorithm where a vector needs to be updated a double for-loop is used as shown in the listing below.

```
for(int j=0;j<N-1;j++)
  for(int i=0;i<N-1;i++)
    point= i + N*j;
```

Note that the calculations are done such that the ghost points[3] are not calculated, see

[3] The ghost point is used as padding.

Section 4.5. Also note that these computations could have been done in parallel. However, only one core is used for the CPU implementation.

The matrix vector multiplication for the finite difference method is equal to the addition of five vectors. Each point is calculated with two for-loops. For the update, the finite difference constants are multiplied by the appropriate points from the vector and added to each other. For each point it is checked if the point is connected in the boundary. If not, the element is added. This method will be called the matrix-free method or finite difference operator, because the matrix is never constructed.

```
for (int j=0;j<N-1;j++)
  for (int i=0;i<N-1;i++)
    point =i+N*j;
    v[point]=((4.0L/(h * h))-(1-(alphaProblem * I)) * (k * k)) * p[
       point];
    if(j!=0)//south
      v[point] -= p[point-N]/(h * h);
    if(j!=N-2)//north
      v[point] -= p[point+N]/(h * h);
    if(i!=0)//west
      v[point] -= p[point-1]/(h * h);
    if(i!=N-2)//east
      v[point] -= p[point+1]/(h * h);
```

The implementation of the inner-product with the begin vector is only a read of the element at the source point. This is because the begin vector consists of zeros with only a one at the source point. This is different than the final implementation on the DFE. The implementation on the CPU is not changed to keep the same baseline. If the update would also be made then the CPU calculation time will become slightly larger.

The used norm is the 2-norm. This is also calculated with two for-loops. A sum variable is used and the square root is taken at the end.

```
for(int j=0;j<N-1;j++)
  for(int i=0;i<N-1;i++)
    point= i + N*j;
    norm += creal(s[point])*creal(s[point]) + cimag(s[point])*cimag(s
       [point]);
return sqrt(norm)
```

4.3 BiCGSTAB on the data flow machine

In this subsection the idea of the implementation will be explained. After this subsection the changes and improvements on the idea will be presented.

As a starting point, a few assumption will be made. The first assumption is that $\bar{\mathbf{r}}^0 = \mathbf{r}^0$. Second is that the start-solution $\mathbf{u}^0$ is a zero-vector. This will result in an error in the setup step which is $r^0 = g$ and therefore $\bar{\mathbf{r}}^0 = \mathbf{r}^0 = \mathbf{g}$. Also k from Equation 2.11 will be assumed to be constant. The last assumption is that the source function is a point-source. This means that $\mathbf{g}(x, y, z) = \delta_{x_s,y_s,z_s}(x, y, z)$ with δ the Dirac delta function at point x_s, y_s, z_s. For this project the source is located at the centre of the domain. The final implementation will not need the last assumption. All these assumptions and

some renaming of the variables result in Algorithm 4.1, where all variables in bold represent vectors. Note that $M = I$, with I the identity matrix, is the algorithm without preconditioner.

Algorithm 4.1 BiCGSTAB for implementation.

1: $\mathbf{u} = \mathbf{0}$; $\bar{\mathbf{r}} = \mathbf{r} = \mathbf{g} = \delta_{x_s,y_s,z_s}$; $\rho_{old} = \alpha = \omega = 1$; $\mathbf{v} = \mathbf{p} = \mathbf{0}$;
2: **for** $i = 0, 1, 2, \ldots, maxit$ **do**
3: $\quad \rho_{new} = \mathbf{r}(x_s, y_s, z_s)$; $\beta = \frac{\rho_{new}}{\rho_{old}} \frac{\alpha}{\omega}$; $\rho_{old} = \rho_{new}$;
4: $\quad \mathbf{p} = \mathbf{r} + \beta(\mathbf{p} - \omega \mathbf{v})$;
5: $\quad$ Solve $M\hat{\mathbf{p}} = \mathbf{p}$;
6: $\quad \mathbf{v} = A\hat{\mathbf{p}}$;
7: $\quad \alpha = \frac{\rho_{old}}{\mathbf{v}(x_s,y_s,z_s)}$;
8: $\quad \mathbf{s} = \mathbf{r} - \alpha \mathbf{v}$;
9: $\quad$ **if** $||\mathbf{s}||$ small enough **then**
10: $\quad\quad \mathbf{u} = \mathbf{u} + \alpha\hat{\mathbf{p}}$; quit;
11: $\quad$ **end if**
12: $\quad$ Solve $M\mathbf{z} = \mathbf{s}$;
13: $\quad \mathbf{t} = A\mathbf{z}$;
14: $\quad \omega = \frac{(\mathbf{t},\mathbf{s})}{(\mathbf{t},\mathbf{t})}$;
15: $\quad \mathbf{u} = \mathbf{u} + \alpha\hat{\mathbf{p}} + \omega \mathbf{z}$;
16: $\quad \mathbf{r} = \mathbf{s} - \omega \mathbf{t}$;
17: $\quad$ **if** $||\mathbf{r}||$ is small enough **then**
18: $\quad\quad$ quit;
19: $\quad$ **end if**
20: **end for**

After the literature study [6] the following plan was derived. Three kernels are needed to be implemented, because three different kinds of calculations have to be done. The first kernel is the convolve kernel, which calculates the matrix-free finite difference operator. The second kernel is an inner-product kernel. The third kernel has three streams of data and two constants and multiplies two of the streams by the two constants and adds the results up by the first stream. This will be called the addition kernel. The manager connects these kernels.

4.4 Implementation differences and complications

After the literature study, a plan was formed on how to implement the BiCGSTAB algorithm on the DFE, as described in Section 4.3. However, this creates difficulties which were not obvious for the user. This subsection will explain those difficulties and how they are addressed.

4.4.1 Multiple kernels

The original plan was to build 3 different kernels or more and connect them all up with the manager. This way one calculation can flow into the other. However, this implementation used many streams and the Maxeler tools simultaneous streams limit is 15. The usage

of a flow means that the manager connects a stream of data to 2 parts of the program. Also, because the implementation used the streams limit, the build times were long. This happens because one of the resources of the machine was used to the maximum. However, sometimes it was possible to build a Maxfile and sometimes the build failed after the maximum amount of attempts. This means that no cost table with zero score was found (see Section 3.7).

The final implementation is all built in a single file. Because some parts are combined, less streams are needed and therefore build times are shorter.

4.4.2 Complex numbers

The Helmholtz equation is using complex numbers. However, implementing complex numbers is not fully supported by the Maxeler tools. Only real constants can be set in the program. The workaround here is to put the real and complex constants in there, and make a complex constant inside the kernel. This works fine, but leads to more code.

The Maxeler machine expects a flow of doubles and when they are complex doubles, the flow is twice as long with real and imaginary part alternated. Therefore, a conversion is needed from complex numbers to a vector of doubles, before sending to the DFE.

Finally, the error is calculated using the complex conjugate. This function does not exist for DFEVector<DFEComplex>. However, it does exist for scalar DFEComplex values. Hence, a new DFEVector<DFEComplex> is created and for each element of the original vector the complex conjugate is connected to the new vector.

4.4.3 Intermediate convergence test

In the BiCGSTAB algorithm in Algorithm 4.1 at steps 9-11 a convergence test is performed. However, this step is sacrificed to save a stream back and forth from the device. This would also cost extra calculations. This was not beneficial to save half an iteration on some occasions.

4.4.4 Last steps of algorithm

In Algorithm 4.1 ρ_{new} is calculated at the first step of a new iteration. This can also be done at the end, after $\mathbf{r}$ is calculated. This is needed because constants of the kernel have to be set before a kernel is called upon by the CPU. Therefore, ρ_{new} will be calculated at the last step of an iteration instead of the first.

4.4.5 Small improvements

The first small improvement to save a cycle is done when 4 numbers are added. $((a + b) + c)+ d$ costs 3 cycles while $(a + b) + (c + d)$ costs only 2 cycles. Both implementations use the same amount of resources.

Next, 2 multiplications could be saved by the following process. Consider the calculations for the inner-products:

```
(part1?r0*output:(part3?r0*between:between*output));
```

This is a straightforward implementation of required results. Each part has its multiplication and the multiplexers choose the calculations that are needed per part. The following implements the same calculation:

```
(part2?between:r0)*(part3?between:output);
```

However, this implementation costs 2 multiplications less.

Everything that is programmed in the kernel is calculated on the DFE. This also holds for constants. Sometimes these constants are the same for each tick. It would be more efficient if the constant is calculated beforehand and is sent to the kernel as a new constant.

The last small improvement has a bigger influence. At first to combine line 15 and 16 of Algorithm 4.1 into part 3, an additional stream is added to the kernel and connected by the manager. However, in part 3 the result of the matrix-vector multiplication is not used. Hence, this stream is also not used. This improvement adds a multiplexer before the stream connection of the matrix-vector product. This multiplexer chooses the result of the matrix-vector multiplication for the first 2 parts and U in the third part. This improvement saves a stream connection and a lot of resources with it. This improvement is only implemented in the complex implementation and makes a higher frequency easier accessible.

4.5 Preconditioning

In the original plan a preconditioner was intended. The preconditioner is the shifted Laplacian preconditioner introduced by Erlangga (see [3]) which is based on the operator

$$\mathscr{M}_{\beta_1,\beta_2} = -\Delta - (\beta_1 - \beta_2 \mathbf{i})k^2, \tag{4.1}$$

with the same boundary conditions as the original equation. The precondition steps will be calculated with a standard multi-grid V-cycle. A damped-Jacobi pre- and post-smoother will be used.

The grid has to be aligned with the burst-size of the DFE. This burst-size is from the PCIe interface between the CPU and the DFE. Hence, per dimension a multiple of 384 bits should be used, which is 3 double complex numbers or 6 doubles. Each level is a multiple of this. However, in the multi-grid V-cycle, a ghost point is needed in the grid to align the grids equidistantly. Figure 5 shows a 1D grid for 6 points. The first 5 are spaced out between 0 and 1. Point 6, or 5 if you count from 0, is the ghost point. Then in the V-cycle, when the grid halves the next grid will become the following; point u_1 will become $\bar{u}_0$, u_3 will become $\bar{u}_1$ and u_5 will become $\bar{u}_2$. The next level is also equidistantly spaced out and can be easily calculated. If this was done differently the matrix operator will be different, the points do not lay on the same place in the grid. For example, take 6 points, then in the next level 3 points remain. If they need to be equidistant the restriction operator will be harder, that is why this grid is chosen. However, due to time constraints, the multi-grid pre-conditioner was not implemented, but the grid was already prepared and is as described in Figure 5.

The difficulty with the multi-grid is with the solve step. When the grid is small enough on the CPU, then the system is solved with a quick algorithm. However, for the DFE

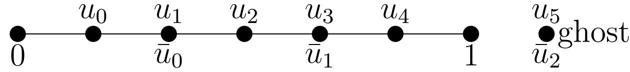


Figure 5: 1D discretisation for $h = \frac{1}{6}$.

there are no algorithms developed yet. Hence, an idea was to bring the grid back to 1 point and then expand back again. Another idea was to solve the effective 2×2, or 3×3 system with the ghost points, directly. This can be streamed when it were complex numbers to comply the burst-size. The last step for doubles has to be done at once from one level higher with a 6×6 system..

Finally, note that all these calculations cost space on the DFE that will only be used in the multi-grid step. Therefore, the efficiency of the implementation will be lower, less units will be working effectively and less parallel paths can be implemented.

4.6 2D Parallelism

When the algorithm was implemented only a small amount of the arithmetic resources were used. Therefore, additional parallel paths could be implemented. The parallel paths are implemented using DFEVectors. Suppose there is a stream of data flowing in a kernel. Then for each element an operation is applied and it is sent out. With DFEVector, the kernel accepts the amount of elements of the vector, splits them and applies the operation in parallel at the same time on all of them at once. Then it combines the result into a new vector with the original places of data and sends the data to the manager. Hence, the kernel will process a vector of elements instead of a single data element per tick.

Parallel paths will not result into any problems for a multiplication and an addition, because the same elements will be added to each other and multiplied with the constants. The first difference that occurs is in the stream.offset function of the Matrix-free operator. However, Maxeler has a function called stream.offsetStriped for this. This does what stream.offset does for normal DFEVars.

Processing the boundary results into some difficulties. A counter is kept in the non-parallel case to calculate when the next points of the matrix are on the boundary and are set zero due to the Dirichlet boundary conditions. This is done with a multiplexer connecting a zero-vector when the boundary condition is met, otherwise it takes the stream.offset. The stream.offsetStriped function works fine for the Y-direction. Then it just takes a whole zero vector the way it should. However, for the X-direction only the first value of the vector should be zero. A new vector is made to solve this and it connects the original value to all points except for the boundary. For the boundary it connects a zero element.

Finally, the inner-product also needs a small extra step. The implemented inner-product function needs scalar values to add the partial sums. Hence, the elements of the vectors are added up to become scalar and are then added to a partial sum.

4.7 Final algorithm

Algorithm 4.2 is divided into 3 parts. In each part an update is done to a vector with an addition kernel. Then a matrix vector product is calculated with the result of the addition. At last, a constant is calculated with 1 or 2 inner-products. These steps are separated so that the new constants can be set before the kernel is called upon. Hence, in Algorithm 4.2, steps 4,5 and 6 are part 1, steps 7, 8 and 9 are part 2. Part 3 is slightly different. There are 2 additions left with step 10 and 11, an inner-product in step 12 and an inner-product in step 13. Therefore, an addition is added to each part. However, the result is only read in part 3. Also note that there is no matrix-vector product in part 3 of the algorithm. The result of the matrix-vector product is still calculated, but not used.

Algorithm 4.2 BiCGSTAB after implementation

1: $\mathbf{u} = \mathbf{0};\ \mathbf{r_0} = \mathbf{r} = \mathbf{g} = \delta_{x_s,y_s,z_s};\ \rho_{old} = \alpha = \omega = \rho_{new} = 1;\ \mathbf{v} = \mathbf{p} = \mathbf{0};$
2: **for** $i = 0, 1, 2, \ldots, maxit$ **do**
3: Part 0: $\beta = \frac{\rho_{new}}{\rho_{old}} \frac{\alpha}{\omega};\ \rho_{old} = \rho_{new};$
4: Part 1: $\mathbf{p} = \mathbf{r} + \beta(\mathbf{p} - \omega\mathbf{v});$
5: Part 1: $\mathbf{v} = A\mathbf{p};$
6: Part 1: $\alpha = \frac{\rho_{old}}{(\mathbf{v},\mathbf{r_0})};$
7: Part 2: $\mathbf{s} = \mathbf{r} - \alpha\mathbf{v};$
8: Part 2: $\mathbf{t} = A\mathbf{s};$
9: Part 2: $\omega = \frac{(\mathbf{t},\mathbf{s})}{(\mathbf{t},\mathbf{t})};$
10: Part 3: $\mathbf{u} = \mathbf{u} + \alpha\mathbf{p} + \omega\mathbf{s};$
11: Part 3: $\mathbf{r} = \mathbf{s} - \omega\mathbf{t};$
12: Part 3: $\rho_{new} = (\mathbf{r}, \mathbf{r}_0);$
13: **if** $||\mathbf{r}||$ is small enough **then**
14: quit;
15: **end if**
16: **end for**

Figure 6 shows a simple representation of the algorithm with the flow diagram with the nodes of Section 3.4. A lot of details are left out here to get a general overview of the implementation. The total graph is unreadable. First, the matrix vector operation is visualized as an addition of the 5 stencil points. The boundary is not taken into account nor the ghost point. Therefore, the counters are also not shown. Second, parallel paths are also not displayed. This figure represents the flow of a single path. The changes that are needed for a parallel path described in Section 4.6 are not displayed. Finally, per iteration this kernel is used 3 times. When the different parts of the kernel are used, they are displayed with the square blocks. However, these calculations are always done, whether or not they are used. Input 3 for example is always used but has no stream in part 2. Then a zero vector and zero constant are used, but this is not displayed.

Figure 6 gives an overview for how the data flows in the DFE. 4 Input-flows enter into the kernel and are multiplied by constants and added in the higher part of the graph. These are the addition kernels. Then the outflows of "U" on the left and "between" on the right are done in the middle as is the matrix-free operator. The result of matrix-free operation and the result of "U" are sent to "output". The multiplexer on the left above

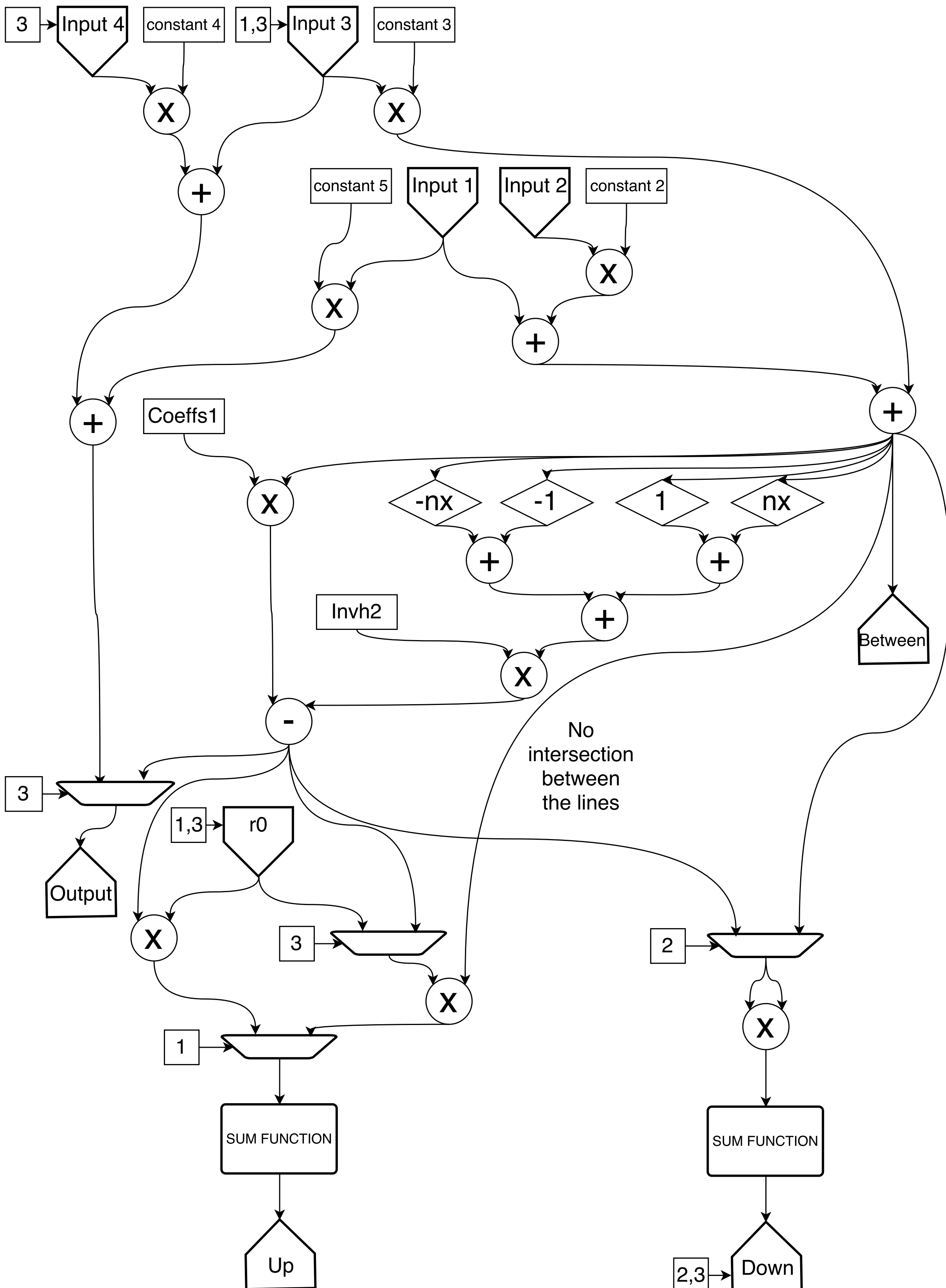


Figure 6: Simple representation of the data flow graph of the Kernel.

"output" chooses the stream.

The bottom part of the graph displays the inner-products. These are filled with multiplexers, which are filled with "Between", "output" and the last inflow of $r0$ to calculate the respective constants per part. The inner-products are displayed with the sum functions to present the adders. This is again a simplification.

5 Second order finite difference matrix-vector product

In this section the difference between matrix-free operator and matrix-vector multiplication implementations will be explained. First the difference and second the tiling will be presented. Finally, the results comparing all these implementations will be given.

5.1 Matrix-free operator versus matrix-vector multiplication

The finite difference operator has only dependencies with the points next to it. The matrix has 5 non-zero diagonals in 2D or 7 in 3D (see the stencil in Equation 2.21 in Section 2.3). Hence, for the matrix-free implementation, the stream.offset method will be used. Then the Matrix-free operator is a simple addition of 5 elements in 2D stored for the right amount of time. The bigger the sizes of the N_x, N_y and N_z dimensions are the more storage will be needed. In 2D $N = 2 \cdot N_x$ and in 3D $N = 2 \cdot N_x \cdot N_y$ elements, need to be stored before they can be added.

In the matrix-vector multiplication the vector x has to be multiplied with the matrix A. The result is an inner-product for each element. This means that each element on the same position is multiplied with each other and then all partial results are added up. This is usually done on a CPU with a temporary sum and then each new number is added to it.

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

The DFE operates on streams, and in order to calculate a temporary sum, some special function is needed. This is done on the Data Flow Engine (DFE) with a number of partial sums. In this project 16 adders are used. When a new number that has to be added is calculated, the program adds the number to one of the partial sums. The program creates a new number by adding that partial sums and the new number and connects it to the old place of the partial sum. This leads to 16 numbers that together hold the information. Therefore those numbers should be added. This is done with stream.offset and all the 16 adders are delayed one by one and added up. Of course this process needs to be repeated for each row of the matrix. Therefore, the vector will be stored in the fast memory (fmem). A limit of the DFE for this method might be the amount of fmem for the vector. However, this operator cannot be used for systems larger than the amount of fmem available. The storage of the full matrix in the memory is the limitation of this method.

5.1.1 Implementation details

Matrix-free operator

The matrix-free implementation will be done the same as explained in Section 4.2. Hence, 4 stream.offset functions will be used for all bordering points. They are added and sent back. With a tiled implementation the Dirichlet boundary is implemented due to the zero halo around it. However, for the untiled implementation the boundary has to be taken into account, see Section 5.2 for more information on tiling.

For the implementation using the DFE large memory (lmem), nothing changes. Only that it first has to be send to the lmem and then read from there. Obviously this will impact the system performance.

Matrix-vector product
A matrix-vector multiplication can be separated into an inner-product per row with the vector. Hence, the vector is needed for each row and therefore will be stored into the fmem. The inner-product will be done with partial sums and for each tick the new product will be added to one of these sums as explained in Section 5.1. Then the sums are added with stream.offset function to respect the timing.

The large memory implementation is again not very different. The difference is that the data first has to be sent to the DFE lmem. Then the data will be streamed from the lmem to the kernel.

5.2 Tiling

To keep the use of the fmem manageable, tiling will be done. Tiling is the process of splitting the domain into multiple tiles. In Section 2.3 the lexicographical numbering is introduced. In Table 3 a small 4 by 4 grid is shown as motivational example.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Table 3: Small 4x4 grid lexicographical numbered.

With tiling, this numbering is different. The first element remains the same and horizontal is counted first. However, the counting does not count until the end of the row but until the end of the tile. Then in a row lower, the numbering continues. This is done until the last line of the tile and then the next tile will begin in the same order as within the first tile. Table 4 shows this.

1	2	5	6
3	4	7	8
9	10	13	14
11	12	15	16

Table 4: Small 4x4 grid numbered in 2x2 tiles.

To determine the matrix-free operator, for each point, the points next to it are needed. With the numbering system explained above, the points on the boundary of each tile are much further away from each other. In Table 4, the distance between point 3 and 9 is 1.5 times larger than in Table 3 between points 5 and 9. This can be bigger for other tile sizes. Hence, a halo is created. The halo is a ring of all data points around a tile. Hence,

for the tile 1,2,3,4, or originally 1,2,5,6 all points left, right, above, and below are added. With the Dirichlet boundary conditions, this leads to Table 5.

0	0	0	0
0	1	2	5
0	3	4	7
0	9	10	13

Table 5: Tile 1 with halo.

Note that tiling also can be done for storing a smaller matrix. In this case the matrix of the tile and the original is the the same, 16 by 16. However, with tiling, any original grid can be tiled up till the maximal tile size. When the matrix-vector product has been calculated for tile one, then only the original tile has been calculated correctly and has to be de-tiled.

Large matrix-vector products will be compared. Therefore, large matrices have to be stored. This is of course very inefficient, however it is needed for these tests. This storage will be the limitation.

In 2D without tiling, $2 \cdot Nx$ fmem units will be needed for the matrix-free operator. However, if the x-axis is split into 3 and a halo is used then only $2 \cdot (Nx/3 + 2)$ fmem units are needed. Tiling obviously costs extra time because more data needs to be send in total and the reordering of the data also costs time. However, if the fast memory is the limiting factor, tiling is a valid option. On the other hand, this will be a bigger problem for 3D cases, because the bandwidth of the matrix will be much bigger than in 2D.

5.3 Experimental results of matrix-vector product

96x96	100MHz	150MHz
matrix-vector CPU mem	0.90	0.65
DFE matrix-vector lmem	0.85	0.57

Table 6: Matrix-vector product of different memory for different frequencies, times in seconds.

96x96	CPU	DFE 150MHz lmem
matrix-vector	0.33	0.57
matrix-free	0.00015	0.015

Table 7: Matrix-vector product vs matrix-free for the CPU versus the DFE, times in seconds.

Tables 6 and 7 present the results of the tests over an average of 10 runs for a grid of 96 by 96. This grid is almost as huge as what the CPU can store. The full matrix cannot

be bigger because the zero's in the matrix are also stored as doubles and the CPU does not have more memory. Table 6 compares the difference between using the large memory (lmem)of the DFE versus the memory of the CPU. It also compares the difference between using the DFE on 100 MHz versus 150 MHz. Table 7 compares the CPU implementation with the lmem implementation on 150 MHz. It also compares the matrix-free operator versus the matrix-vector product.

To analyse the experiments two bounds will be used. The first bound is the Input Output (IO) bound, this bound is the time it takes to transport all data that is needed for the calculation. This is al the data divided by the bandwidth of the used connections. The second bound is the compute bound, this bound is the time it takes the calculations. This bound will be the amount of ticks divided by the time it takes the DFE to calculate a tick which is the frequency it is run on. This will lead to a minimal amount of time that will be needed for the calculations and when the experimental times are close it means that there is little overhead.

Calculating the matrix-vector product takes the DFE 0.65 seconds on 150 MHz. The improvement of the use of the lmem is only 10% in this case, that only takes 0.57 seconds. The higher frequency of the DFE, the faster it is in this case. For the lmem case, the calculation time decreases 1.5 times while frequency increases 1.5 times. Therefore, it can be concluded that this implementation is compute bounded. The CPU memory implementation does not decrease its time by a factor 1.5. Hence, the memory flow has an influence here. This of course makes sense, because the lmem bandwidth is higher than the bandwidth between the CPU and the DFE. However, writing the data on the large memory also costs 0.65 seconds.

In Table 7 is shown that the matrix-vector product takes 0.33 seconds on the CPU, while it takes 0.57 seconds on the DFE on 150 MHz with the lmem implementation. If this number is compared with the matrix-free implementations then the difference is huge. A matrix-free implementation on the CPU only takes 0.00015 seconds and for the DFE 0.015 seconds. The CPU is this much faster because the problem size is small and it can use its closest caches while running at high frequency. The DFE runs at only 150 MHz. Matrix-free is much better than a full matrix-vector multiplication. This of course makes sense, due to the sparseness of the matrix.

A sparse matrix-vector multiplication kernel will be an improvement over the full matrix-vector multiplication. However, if it will be faster than the matrix-free implementation will be a project for the future.

5.3.1 Results for large systems

The large system that will be used is a 6000×6000 grid. This means that 36 million complex numbers are used on this grid. With a communication speed between the CPU and DFE of 3GB per second, only transferring this data will take $\frac{2 \cdot 6000^2}{3000 \cdot 2^{20}} = 0.366$ seconds. With a non-parallel implementation, the compute bound will be $\frac{6000^2}{100 \cdot 10^6} = 0.36$ seconds on 100MHz.

The results are presented in Table 8. The calculation times are not close to the bounds. This is because making the complex vector in a double precision and back takes twice 0.1 seconds. The kernel only takes 0.44 seconds. Hence, the DFE calculation in total takes 0.67 seconds to calculate the matrix-vector product and the CPU only 0.38 seconds.

CPU	DFE 100MHz	speedup	tile size	efficiency	tiled	speedup
0.38	0.67	0.58	6002×6002	99.93%	0.98	0.39

Table 8: 2nd Order finite difference matrix-vector of 6000×6000 grid, times in seconds.

However, the computed time of the CPU is as fast as the memory bound. Because the IO-bound is the same as the calculation time of the CPU, a DFE accelerated implementation can never be achieved for the PCIe connection.

Tiling and de-tiling both take about 0.15 seconds for this system when the tile is 6002×6002. Hence, tiling costs about 30% of the time in this case. However, larger problems can be calculated then.

6 Experimental results of the implementations

This section presents the results of the implementations. The first 5 subsections will start to specify the implementation and then give the results of that implementation. First, the fast memory(fmem) implementation and second, the split kernel implementation is showed. Third, the parallel implementation is presented and as fourth the final implementation of the real valued problem is given. The optimized complex problem is explained as fifth. Finally, the results are compared with literature and an analytical indication is given for newer hardware.

The following constants will be the same for all problems unless otherwise specified:

$$h = \frac{1}{N}, \tag{6.1}$$

$$k = \frac{0.625}{h}, \tag{6.2}$$

$$\alpha = 0.05, \tag{6.3}$$

with N the grid size per dimension. The constants are derived in Section 2.3. The distance between grid points is h. The wave number is k. The last constant α is the damping factor.

6.1 Fast memory implementation

This subsection presents the results of the first complete implementation of the BiCGSTAB algorithm on the DFE. This implementation was discontinued because it used too many stream connections and the maximal grid size was only 180×180. First the implementation will be described and then these problems will be explained. The code of the kernel is presented in Appendix C. Algorithm 2.2 contains the referred steps. Note that this code is not optimised with the later insights.

This implementation runs the entire algorithm at once. This means that the Data Flow Engine (DFE) is only called once using the SLiC function prototype. However, not all steps can be calculated at the same time. Therefore, counters and booleans are needed to trigger calculations at the right times. In total 6 counters are used. Two Counters for step 4 and step 8 to keep track of the position of the grid in the x- and y-axis. One Counter for step 14 and one counter for step 15. Note that these counters are similar to part 1,2,3 from Section 4.7. The boolean of each step is true when the counter of the same step is ticking.

This implementation is called the "Fast memory" or "fmem implementation" because everything that is reused is saved in the fast memory of the DFE. For example, in Algorithm 2.2 step 4 and step 8, both of them need $\mathbf{r}$. However, they cannot be calculated at the same time. Hence, $\mathbf{r}$ is written in the fmem.

```
Memory<DFEVar> rInput = mem.alloc(dataType, dataSize);
rInput.write(point4, r, step4Bool);
```

Writing in the fmem needs two steps. First, the memory is allocated with mem.alloc and second, the data is written at "point4" when "step4Bool" is true. The "4" refers to step

4 of the algorithm. In total 5 vectors are stored on the fast memory. These vectors all have the size of the grid times the amount of bits of the precision. Therefore, the grid is limited at 180×180 when using double precision data values, because almost all the fmem will be used in this case.

There are two alternative implementations that will be presented in this subsection. The difference is in where the data is stored. For the first implementation, the CPU memory is used and for the second, the large memory(lmem) of the Data Flow Engine(DFE) is used. The data communication from the lmem to the DFE is faster than the communication between the CPU memory and the DFE.

Finally, note that the error is calculated on the CPU and not on the DFE. This does not result to extra streams for the implementation on the CPU memory. However, for the lmem implementation the CPU needs $\mathbf{r}$. Also $\mathbf{u}$ is send to the CPU memory and this results in extra calculation time, which is not needed. The result only needs to be send once at the end.

6.1.1 Results of fmem implementation

Table 9 and Table 10 present the results of the fmem implementation with the data stored on the CPU memory. All the data is real valued. In Table 9 in the first column the

grid	CPU	DFE 100MHz	DFE 125MHz	DFE 150MHz
4×4	$2.30 \cdot 10^{-6}$	$1.23 \cdot 10^{-3}$	$1.23 \cdot 10^{-3}$	$1.25 \cdot 10^{-3}$
8×8	$1.66 \cdot 10^{-5}$	$3.74 \cdot 10^{-3}$	$3.03 \cdot 10^{-3}$	$3.15 \cdot 10^{-3}$
16×16	$2.73 \cdot 10^{-4}$	$1.34 \cdot 10^{-2}$	$2.37 \cdot 10^{-2}$	$1.79 \cdot 10^{-2}$
32×32	$5.37 \cdot 10^{-3}$	$4.60 \cdot 10^{-2}$	$4.39 \cdot 10^{-2}$	$4.41 \cdot 10^{-2}(3 \cdot 10^{-5})$
64×64	$8.40 \cdot 10^{-2}$	$5.90 \cdot 10^{-1}$	$3.06 \cdot 10^{-1}$	$2.94 \cdot 10^{-1}$
128×128	1.08	2.55	2.35	2.20

Table 9: Results CPU mem fmem implementation in seconds per grid size. Stop criterion is 10^{-6}, times are in seconds.

grid sizes are presented. In the second column, the calculation time of the BiCGSTAB algorithm on the CPU is presented. Column 3-5 presents the calculation time of the BiCGSTAB algorithm on the DFE for different frequencies. These frequencies are the clock speed of the DFE. A higher frequency can result in faster calculation times.

First, note that for the grid 32×32 and the DFE frequency of 150 MHz the problem did not converge. At some point truncation errors will add up and make the value at the source point zero and therefore α infinite. The result that is presented had an error of $3 \cdot 10^{-5}$ and was stopped there. The BiCGSTAB algorithm is sensitive to truncation errors. Especially when the algorithm does not converge fast enough these errors can add up. Hence, for the algorithm without preconditioner this will happen more often.

At last note that in the last column the first and second calculation times are longer than their lower frequencies counterparts. This is probably a result of timing errors. However, it could also be some overhead.

Table 10 presents the difference for different stop criteria of the BiCGSTAB algorithm of a grid of 128×128 between the CPU and the 100 MHz DFE implementation. Column

Stopcriterion	#iterationsCPU	CPU	#iterationsDFE	DFE 100MHz
10^{-1}	186	$1.07 \cdot 10^{-1}$	188	$2.01 \cdot 10^{-1}$
10^{-2}	1321	$6.32 \cdot 10^{-1}$	1179	1.27
10^{-3}	2095	$9.87 \cdot 10^{-1}$	2013	2.17
10^{-4}	2188	1.01	2108	2.29
10^{-5}	2240	1.04	2126	2.29
10^{-6}	2342	1.08	2356	2.55
10^{-7}	2359	1.08	2387	2.57

Table 10: Results CPU mem fmem implementation in seconds per precision for grid size of 128×128, times are in seconds.

one sets the minimal stop criterion that needs to be reached. The second and fourth column presents the amount of iterations that the CPU and DFE needed respectively to converge. Column 3 and 5 present the calculation times of the CPU and the DFE. Note that the number of iterations needed for the DFE are different than the CPU due to truncation errors that add up. In Table 11 the results are presented of a brief analysis

level	l	7
N	$\left(2^l\right)^2$	16384
IO Bound	$\frac{(4+4)\cdot N\cdot 8}{3000\cdot 2^{20}}$	$\frac{1}{4000}s$
Compute Bound	$\frac{4\cdot N}{100\cdot 10^9}$	$\approx \frac{1}{1520}s$

Table 11: Brief analysis of the compute and IO bound.

of the problem. The Maxeler machine that we are using has a PCI-express gen 2 with 8 lanes. This results that effectively 3000 MB per seconds can be transported. In total 4 input vectors and 4 output vectors have to be transported and the size of a double precision data elements is 8 bytes. This results in an IO bound implementation $\frac{N}{3000\cdot 2^{14}}$s. The compute bound is the number of ticks divided by the amount in Hertz that the machine is running, with a standard value of 100 MHz. This results in a bound of $\frac{4N}{100\cdot 10^6}$s. Therefore the problem is compute bound. We compare the results of Table 9 with the expectations for level 7. Levels were implemented to be able to implement the V-cycle of the multi-grid preconditioner(see Section 4.5). Hence, one level higher is twice as big. 2300 iterations were needed, therefore 1.5 seconds is the expected calculation time. The calculation time is 2.5 seconds. Hence, a loss of less than a factor 2 compare to the theoretical bound. When we compare the theoretical bound with the results of the CPU then implementation on the DFE is still slower than the CPU.

Table 12 presents the results of the fmem implementation that stores the data on the lmem. The first column states the grid size. The second and third column present the calculation time of the BiCGSTAB algorithm on respectively the CPU and the DFE on 100 MHz. The fourth column presents the speedup from the CPU to the DFE. This is

size	CPU	DFE 100MHz	speedup
12×12	$1.75 \cdot 10^{-4}$	$9.81 \cdot 10^{-3}$	0.02
24×24	$4.10 \cdot 10^{-3}$	$4.52 \cdot 10^{-2}$	0.09
36×36	$1.87 \cdot 10^{-2}$	$1.02 \cdot 10^{-1}$	0.18
48×48	$5.11 \cdot 10^{-2}$	$2.01 \cdot 10^{-1}$	0.25
60×60	$1.05 \cdot 10^{-1}$	$3.62 \cdot 10^{-1}$	0.29
72×72	$1.88 \cdot 10^{-1}$	$5.31 \cdot 10^{-1}$	0.35
84×84	$3.73 \cdot 10^{-1}$	$7.09 \cdot 10^{-1}$	0.53
96×96	$5.80 \cdot 10^{-1}$	1.04	0.56
108×108	1.02	1.41	0.72
120×120	1.64	2.54	0.65
132×132	2.08	2.99	0.70
144×144	$---$	$---$	$---$
156×156	5.25	5.58	0.94
168×168	5.39	6.78	0.79
180×180	7.88	10.71	0.73

Table 12: Results of fmem implementation on lmem in seconds per grid size, stop criterion is 10^{-6}, times are in seconds.

lower than one and thus the CPU is faster than the DFE.

The stop criterion is 10^{-6}. However, the amount of iterations that both implementations need to converge are not the same. This is due to differences in the value truncation process. Because this algorithm needs a preconditioner to converge within a reasonable amount of iterations, this truncation error accumulates and lengthens the calculation. When the process converge can be considered a bit of lucky with the truncation errors. Note that the problem did not converge at all for the grid of 144×144.

Table 12 shows that even though this code is not optimized that for the larger grids in this table it is almost as fast as the CPU. Also that the DFE scales better than the CPU. This means that the speedup for larger problems is better than for smaller ones. However, 180×180 is the largest grid possible. Therefore, another implementation is needed that uses less fmem.

6.2 Split kernel implementation

This section will explain the split kernel implementation. The fmem implementation was limited to a grid size of 180×180. A new implementation was made with split kernels. Hence, for each process of the algorithm a new kernel was made. For example, the matrix vector operator had a kernel that accepted a vector v and calculated Av. Then this vector was send back to the manager. The manager connected this result to the memory. Some streams were connected between kernels by the manager and flows were copied. This resulted in too many streams for the manager to connect and in very long build times. This implementation was also not optimized. The error is still calculated on the CPU. This means that the CPU has to read the residual vector from the large memory of the

DFE and then calculate the norm of the error.

Algorithm 6.1 presents the calculations needed on the DFE per iteration. Note that the other constants are calculated on the CPU and for α only the $v(sourcepoint)$ is calculated on the DFE. Step 2 is an addition kernel. Step 3 is a matrix-free operator kernel. Step 4 is a source-point kernel. These 3 kernels can all be done for each point after each other and therefore can be combined into part one. Note that this is similar to the description in Section 4.7. Then step 5 and 6 are similar and therefore these two are calculated exactly as part one and the overhead of the source-point is only in computations not in time.

Step 7 is two dot products and it makes part 3 of the algorithm. Step 8 and 9 are both done by a new addition kernel. However, in this implementation they are done after each other to make part 4 and 5. Note that this is not optimal. These steps can be calculated at the same time and this will be done in the next subsection. At last, after all the iterations the results will be read from lmem.

Algorithm 6.1 Bi-CGSTAB steps for the DFE for the split kernel implementation.

1: **for** $i = 0, 1, 2, \ldots$ **do**
2: $p = r + \beta(p - \omega v)$;
3: $v = Ap$;
4: $\alpha = \frac{\rho}{v(sourcepoint)}$;
5: $s = r - \alpha v$;
6: $t = As$;
7: $\omega = \frac{(t,s)}{(t,t)}$;
8: $u = u + \alpha p + \omega s$;
9: $r = s - \omega t$;
10: **end for**

6.2.1 Split kernels results

In Table 13 the results of the split kernel implementation of 10 iterations over 10 times average are presented. In the first column the grid size is given. In the second column the time it took for the CPU to calculate one iteration is presented. In the third column the time it took for the DFE and the CPU to calculate one iteration is shown. In the last column the speedup is reported.

gridsize	CPU	DFE 50MHz	speedup
180×180	$1.37 \cdot 10^{-3}$	$7.00 \cdot 10^{-3}$	0.19
240×240	$2.17 \cdot 10^{-3}$	$1.00 \cdot 10^{-2}$	0.22
6000×6000	1.13	4.20	0.27

Table 13: Split kernel, times in seconds.

Table 13 shows that this implementation is a lot slower than fmem implementation of the previous subsection. It shows this by the lower speedup factor in Table 12 compare to the CPU implementation. This was to be expected because instead of the fmem the

lmem is used to store the data. However, now larger systems can be calculated. Also the data transfer is divided over 10 iterations.

In total 5 times N^2 ticks are needed. With 50 MHz the expected time needed for the DFE to calculate the 6000×6000 would be $\frac{5 \cdot 6000^2}{50 \cdot 10^6} = 3.6$ seconds. This is exactly the time that was measured for the calculation. The other 0.6 seconds of the time are the writing and reading the data from and to the DFE and the calculations of the constants. The implementation works as expected but is not an improvement over the CPU. The resources of the DFE is only used for a small bit. Parallel computations can be done and will be introduced in the next section.

6.3 Parallel implementation

In this section the results after the first parallelisation will be presented. Hence, for each tick 2 calculations are done at the same time. Compared to the last section a number of other improvements have been implemented as well.

The first improvement is that instead of separate kernels, everything is built into a single kernel. This results in less streams for the manager to control.

The second improvement is that instead of 5 parts, only 4 parts are implemented. Part 3 and 4 were merged together. Only this merge action saves N^2 ticks of the execution time. Now, these parts are all instances of the same kernel with an integer to control the part active at any moment in time. With this integer, various data flow graph links are turned on and off when required. However, the part in isolation is of course still preformed. However, its results are just not used.

The last big difference between the proposal in Section 6.2 and this one is the addition of the partial sums of the inner-products. In this implementation the partial sums are sent to the CPU and added up there. This is a complication of implementing the parallel paths.

6.3.1 Intermediate results

Note that build times were much shorter than in the fmem implementation. The build times was days for the fmem implementation and this implementation was done within 2 hours. Therefore, higher frequencies could be tested. Hence, in this section the frequency of the DFE will be 150 MHz.

gridsize	CPU	DFE 150MHz	speedup
180×180	$1.37 \cdot 10^{-3}$	$1.53 \cdot 10^{-3}$	0.90
6000×6000	1.13	0.48	2.35

Table 14: Parallel kernel per iteration, times in seconds.

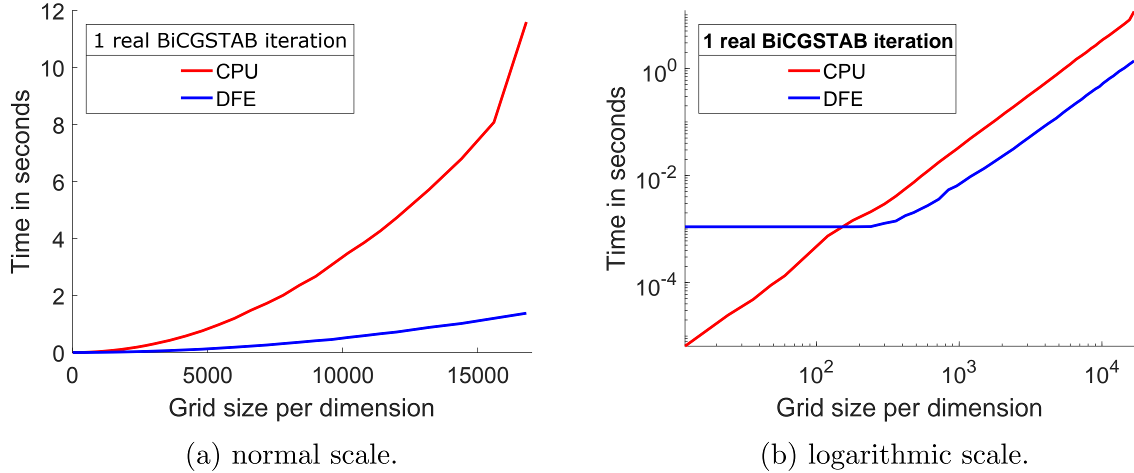
Table 14 compares the implementations of the CPU and DFE per iteration. Hence, the writing and reading is not taken into account. For the grid of 6000×6000, this costs 0.9 seconds. This means that in total the DFE is slower if only 1 iteration is executed.

The calculation time of the grid 6000×6000 was 3.6 seconds for the lmemsplit implementation of section 6.2. Table 14 shows that this implementation takes 0.48 seconds per

iteration. This is 7.5 times faster than the result of the previous implementation of Table 13. This number is achieved because the machine runs 3 times faster, the parallelisation makes it 2 times faster and the removal of 1 out of 5 parts makes it 1.25 times faster. This makes it 7.5 times faster. In the next section the last set of improvements will be presented.

6.4 Final real valued implementation

In this section the result are presented of the final implementation for the real valued problem. The last improvements are implemented as explained in Section 4.4.5, a number of constants are pre calculated. As explained in Section 4.7 the algorithm is split into 3 parts instead of 4. Part 4 is integrated into part 3. This means that again N^2 ticks are saved for the cost of some arithmetic space and an additional stream of data flows into the kernel. Also this algorithm is parallelised into six parallel paths.



(a) normal scale.

(b) logarithmic scale.

Figure 7: Calculation times of 1 real valued BiCGSTAB iteration.

Figures 7a and 7b show the computation time of 1 BiCGSTAB iteration for different grid dimensions, in red it is the time it took the CPU and in blue it is the time it took using the DFE. Figure 7b shows the small grid size for which the CPU is faster. Figure 7a shows that the DFE implementation scales much better than the CPU.

gridsize	CPU	DFE 150MHz	speedup
180×180	$1.45 \cdot 10^{-3}$	$1.10 \cdot 10^{-3}$	1.32
6000×6000	1.13	$1.87 \cdot 10^{-1}$	6.04
16800×16800	11.6	1.38	8.41

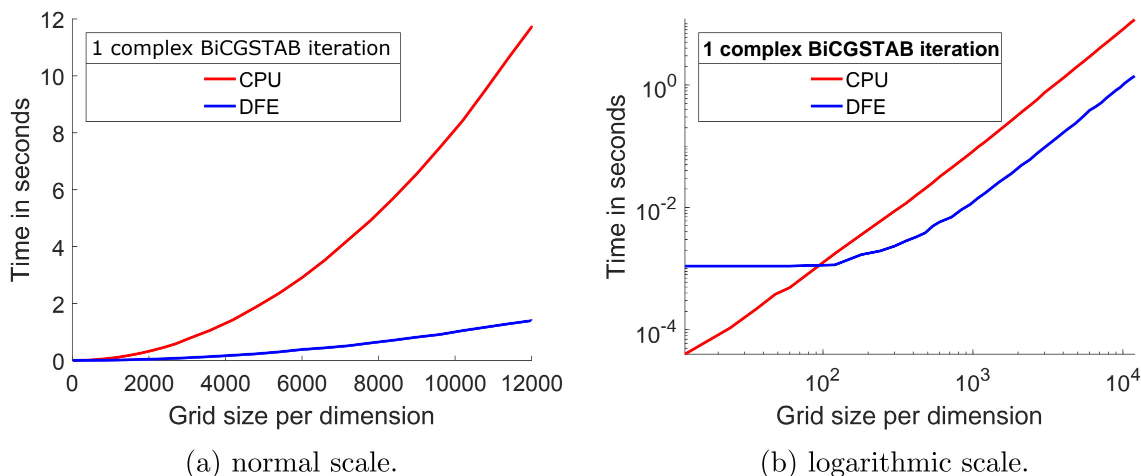
Table 15: Finished real kernel per iteration, times in seconds.

The DFE has a communication speed of 38 GB per second, see Section 4.1.2. The precision is 8 bytes and there are 17 large in- output streams in the 3 parts. This means that the IO bound is at least $\frac{17 \cdot 8 \cdot N^2}{38 \times 10^9}$s. For $N = 6000$ this will result in an IO bound of 0.129

seconds. The results from Table 15 is 0.187 seconds and therefore the last implementation is close to the IO bound. The computation bound is the amount of ticks divided by the frequency times the amount of parallel paths $\frac{3 \cdot N^2}{6 \cdot 150 \cdot 10^6}$s. This compute bound for $N = 6000$ is 0.12. However, one bound for the whole implementation is not correct. Each part has a different IO bound due to the different amount of large streams of date that the calculation require. The first part has 6 streams, the second art has 4 streams and the third part has 7 streams. Hence, each part has an IO bound of $\frac{S \cdot 8 \cdot N^2}{38 \times 10^9}$s, with S the amount of streams per part. The first and third part are IO-bounded. However, the second part is compute bounded. Therefore, the overall bound is $\frac{13 \cdot 8 \cdot N^2}{38 \times 10^9} + \frac{\cdot N^2}{6 \cdot 150 \cdot 10^6}$s. For $N = 6000$ this is 0.138 seconds.

6.5 Final complex valued implementation

In this section the result of the final implementation for the complex valued problem are presented. As explained in Section 4.4.2 some minor changes were needed to implement the complex numbers. Hence, complex constants were created and the complex conjugate was added. The implementation has three parallel paths and also the small improvements as explained in Section 4.4.5.



(a) normal scale.

(b) logarithmic scale.

Figure 8: Calculation times of 1 complex valued BiCGSTAB iteration.

Figures 8a and 8b show the computation time of 1 BiCGSTAB iteration for different grids, in red it is the time it took the CPU and in blue it is the time it took using the DFE. Figure 8b shows that the CPU is better for small grid sizes. Figure 8a shows that the DFE implementation scales much better than the CPU. Hence, for bigger problems the DFE would perform even better. However, bigger problems cannot be computed on the CPU due to main memory capacity limitations. Therefore, it cannot be tested if the results are correct but the DFE could handle much bigger problems. The DFE has 48 GB of memory(see Section 4.1.2) and therefore it can store 7 vectors of 20000×20000. Hence. 20000×20000 is the largest problem in 2D that a MAX3 DFE can compute.

The DFE has a communication speed of 38 GB per second, see Section 4.1.2. The precision is 8 bytes for both real and complex values and there are 17 large in- output

gridsize	CPU	DFE 150MHz	speedup
180×180	$3.58 \cdot 10^{-3}$	$1.20 \cdot 10^{-3}$	2.12
6000×6000	2.913	0.368	7.55
12000×12000	11.7	1.39	8.42

Table 16: Finished complex kernel per iteration, times in seconds.

streams in the 3 parts. However, as in the final real implementation part 2 is compute bounded while part 1 and part 3 are IO bounded. This means that the overall bound is $\frac{13 \cdot 16 \cdot N^2}{38 \times 10^9} + \frac{\cdot N^2}{3 \cdot 150 \cdot 10^6}$s. For $N = 6000$ this will result in an overall bound of 0.277s.

6.5.1 Comparison with results from the literature

In this subsection results will be compared with the work of others. There are only 2 comparisons because no preconditioner was implemented and this implementation is only in 2D.

The first problem is from [5, p. 25]. In this problem $k = 40$, $N = 256$ and the stop criterion is 10^{-6} while the data elements are in single precision. This will be compared to double precision. This is the worst case, a single precision will lead to an improvement in calculation time. However, the grid is chosen to be 252 because the grid needs to be a multiple of 12 for this implementation. Different truncation errors will lead to different amount of iterations. The CPU implementation does not converge and therefore will be compared with the same amount of iterations as in [5] which is 6047. That took the CPU 46.8 seconds to run, while it took the CPU in [5] 370.6 seconds. Hence, the CPU used in this thesis is much faster than the CPU used in [5], but that is not interesting. The DFE implementation took 11860 iterations and 22.8 seconds. This means, the DFE was almost 2 times faster than the CPU while performing twice as many iterations. The GPU implementation of [5] used a similar amount of iterations, 11000, and took 27.1 seconds. Therefore, the DFE is slightly faster than the GPU for this problem.

Compared to the problem of [5, p. 27], which is a grid of 1024 and k=40, a better improvement can be seen. The GPU implementation took 3.2 seconds and the DFE implementation took 1.32 seconds for a grid of 1020×1020. The reason of the small deviation in grid sizes is because the DFE implementation expects grid sizes to be a multiple of 12. Note that 0.08 seconds of the 1.32 seconds are used for copying the data onto the large memory. Both implementations scale well with large grid sizes.

6.5.2 DFE resource usage

Listing 4: Use of Maxeler machine.

```
FINAL RESOURCE USAGE
Logic utilization:      206977 / 297600 (69.55%)
  LUTs:                 166760 / 297600 (56.03%)
  Primary FFs:          182218 / 297600 (61.23%)
  Secondary FFs:         44991 / 297600 (15.12%)
Multipliers (25x18):       963 / 2016   (47.77%)
```

```
  DSP blocks:                 963 / 2016    (47.77%)
Block memory (BRAM18):        834 / 2128    (39.19%)
```

Listing 4 presents the final resource usage of the executable. This report is presented when the implementation is run and an executable is built. More precise usage reports can be found in the files. Listing 4 shows that not every hardware resource unit is used. This means that additional calculations can be placed on the device. In general, to optimally use the reconfigurable device, every unit should be used and the data communication buses should be in use all the time. Earlier in this section, it was shown that the implementation is memory bounded. An other parallel path might fit on the device in terms of resources. However, it will not be any faster because of the memory bandwidth. The available free arithmetic space can be used in the future for implementing the multi-grid preconditioner.

Note that the last line in Listing 4 shows how much fast memory is used. In the implementation presented in Section 6.1, this resource was completely used. However, fmem is now only used 39%. Therefore, an implementation that uses more of the fmem can be an improvement over this implementation. This memory could also be used for the multi-grid preconditioner.

Finally, because the implementation is memory bound, the build has to buffer partial results so that they can be made available at the right time. This leads to a more difficult build process and hence longer build times. An implementation that uses almost all of the DFE resources can also take a long time, because there is much less freedom for the synthetic tools to place the calculations.

6.6 Plots of the calculations

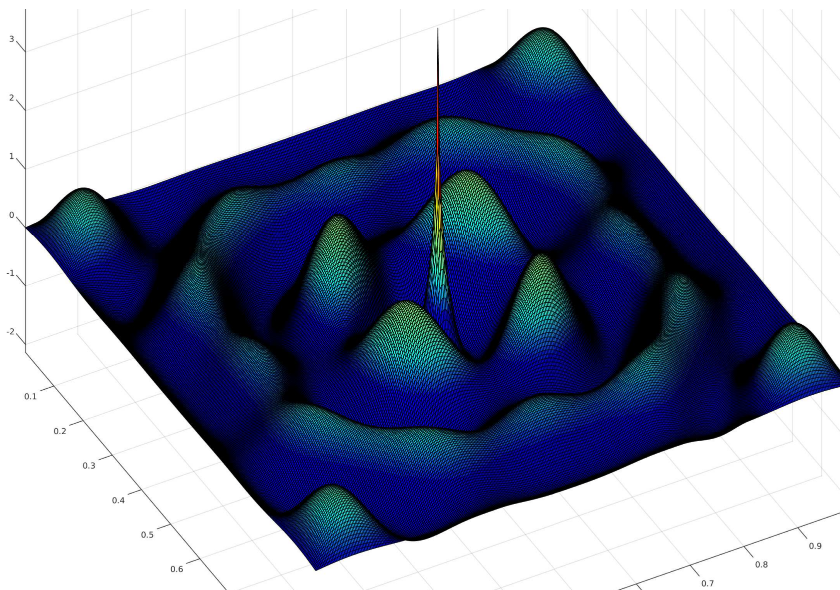


Figure 9: Solution of the Helmholtz equation with Dirichlet Boundary conditions.

Figure 9 presents the result of the Helmholtz equation with Dirichlet Boundary conditions. In the middle, a big red spike can be seen. This is because b is 1 at the source

point and 0 elsewhere. The expected result is a nice circular pattern around the middle. This is because the wave is travelling equally fast in all directions. However, the result in Figure 9 shows circles with interruptions. This is because the wave is bouncing back from the boundary due to the Dirichlet boundary conditions. Therefore other boundary conditions would perform better, because in reality there is no boundary but only a small part of the domain is modelled. This is why in Section 2.2 the Sommerfeld radiation condition is presented. This is not implemented in this thesis due to time constraints. However, instead of correcting the matrix-free operator with zero's from the Dirichlet boundary condition now the stream times a constant have to be placed. This leads to 4 more stream inner interactions.

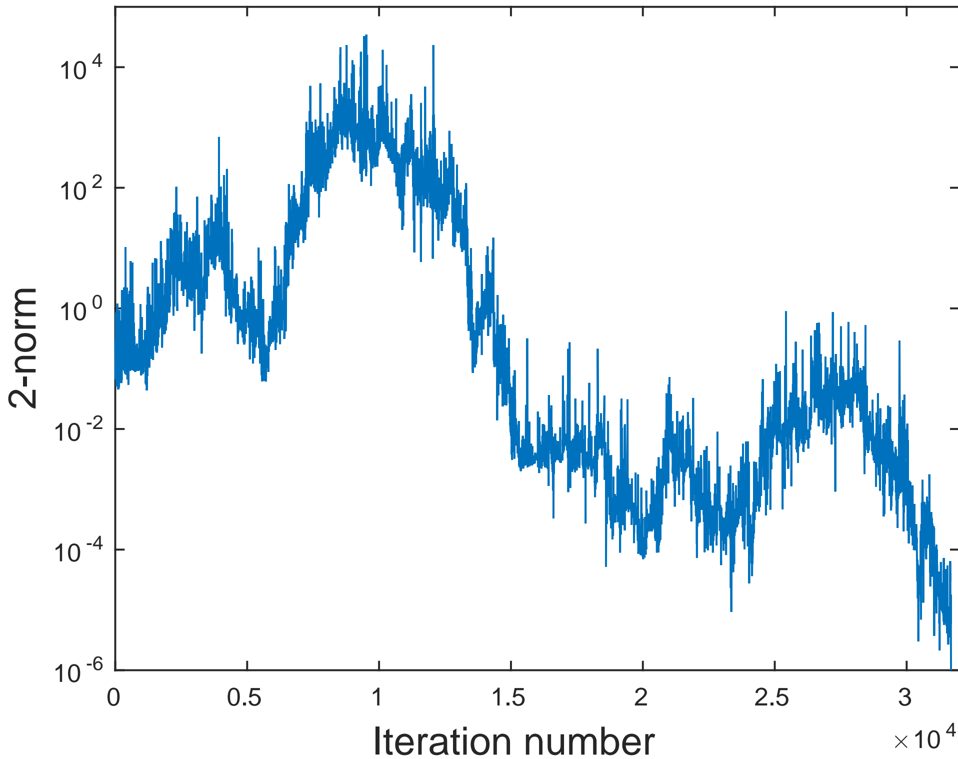


Figure 10: Convergence progress of a grid of 324×324.

Figure 10 presents the convergence progress of the BiCGSTAB algorithm. It shows the overall convergence, however it is not monotone convergence.

6.7 Future performance estimation

Table 16 presents the calculation time of the complex implementation, which is 1.39 seconds for the largest grid of $N = 12000 \times 12000$. The overall IO bound is $\frac{17 \cdot 16 \cdot N^2}{38 \times 10^9} s = 1.03s$. This means a loss of 35%. However, the 3 parts of calculation have their own IO bound. Then a new prediction can be made, because part 2 of the calculation is compute bounded. Part 1 has 6 large flows, part 2 has 4 large flows, and part 3 has 7 large flows.

This means that the IO bound of part 1 is 0.36s, part 2 is 0.24s and part 3 is 0.42s. The compute bound of all parts is $\frac{N^2}{3 \cdot 150 \cdot 10^6} s = 0.32s$. This leads to an overall bound of 1.1 seconds. Hence, the overhead is 26%. However, the experimental calculation times of the 12000×12000 grid are 0.488s for part 1, 0.333s for part 2, and 0.567s for part 3. From this it can be deducted that it takes about 0.081 seconds per stream. This means that the effective communication speed between the DFE and the lmem is about 28.4 GB/s for this implementation. This is about 10 GB/s below the theoretical communication speed. However, the above shows that an analytical model can be set up for other devices to predict performances.

The current implementation is IO bounded, because the frequency could be higher if the data would be available. The effective communication speed between the DFE and the large memory is 28.4 GB/s, call it c in GB/s. This means that an improvement in bandwidth will increase the performance.

However, the compute bound must also be considered. This implementation has a compute bound of $\frac{3 \cdot N^2}{p \cdot f \cdot 10^6}$s, where p is the amount of parallel paths and f is the frequency in MHz on which the device can run.

Therefore, the maximal performance for this device can be predicted when the communication speed is increased. This device can run at most 4 parallel paths at 200 MHz. Therefore, the lowest compute bound is $3.75 \cdot 10^{-9} N^2 s$. In order to match the compute bound, the IO bound is the same when the communication speed between the DFE and the lmem is 72.5 GB/s. For the grid of 12000×12000 this bound is 0.54 seconds.

$$\text{Compute bound} = \frac{3 \cdot N^2}{p \cdot f \cdot 10^6} \text{s} \tag{6.4}$$

$$\text{IO bound} = \frac{17 \cdot 16 \cdot N^2}{c \times 10^9} s \tag{6.5}$$

7 Conclusions

The results and discussions from the previous sections lead to the following conclusions:

- As expected if only the matrix vector multiplication is implemented on the DFE, no acceleration can be achieved as shown in Section 5.3.1. The PCIe communication speed between the CPU and the DFE is the limiting factor.

- The BiCGSTAB method can be implemented on the DFE. This is done by splitting the algorithm in 3 parts, see Section 4.7.

- In Section 5.3.1 results are compared with the literature. An improvement can be seen over the GPU implementation of [5]. This DFE implementation is 2.4 times faster for the BiCGSTAB method without preconditioner.

- Finally, an increase in communication speed between the large memory and the DFE will result in better performance of this BiCGSTAB implementation. When the communication speed is doubled the calculation time will be halved. See Section 6.7 for estimated performance of other hardware.

8 Future research

- In Section 6.3, it is explained that the partial sums from the inner products are sent to and added on the CPU. However, the advantages or disadvantages are not obvious. This can be implemented easily with stream.offset as it was in Section 6.2. However, the build times to get the same performance as the final implementation of this thesis are long.

- Sparse Matrix vector products are also something that can be investigated in the future. In Section 5.3, an obvious answer was achieved for the difference between a full matrix-vector product versus a matrix-free implementation. However, matrices of the finite difference approximations are sparse. Hence, a sparse matrix-vector implementation will lead to an improvement for sparse matrices compared to full matrices. The matrix-free method is expected to remain the fastest choice. However, this is still uncertain.

- Section 4.2 explains that the source point kernel was turned into an inner-product kernel. This is done to make the algorithm more general. However, this is not optimal for the seismic application. Therefore, taking the value at the source point instead of using the inner-product with an additional stream of $\mathbf{r}0$ will lead to a lower IO bound. Therefore an improvement of 0.16 seconds for the grid of 12000×12000 is expected. This is a 13% improvement in calculation time. Another idea for the source-point kernel is to either produce $\mathbf{r}0$ on the fly or store it in the fast memory. This will lead to the same results.

- The goal of this thesis was to implement the preconditioned BiCGSTAB method as described in [3]. The multi-grid preconditioner still needs to be implemented. Some

ideas are presented in Section 4.5. This would increase convergence significantly and therefore decrease the influence of the rounding errors.

- In this thesis the wave number k is constant. To implement the general case, an additional stream of $\mathbf{k}$ needs to be sent to the kernel. The costs in calculation time are expected to be 13% higher because it adds one stream in the first two parts of the algorithm.
- The last suggestion for future research is the implementation of the 3D problem. This may lead to a slower reading of data. Data management will be important and tiling needs to be considered. Multiple DFEs can be used for the tiles.

A Build-log of example

Listing 5: Build-log of example from Section 3.5.

```
Wed 15:41:  MaxCompiler version: 2013.2.2m
Wed 15:41:  Build "examplereport" start time: Wed Jan 25 15:41:41 CET
    2017
Wed 15:41:  Main build process running as user onno on host dutifi.ws
   .tudelft.net
Wed 15:41:  Build location: "/home/location"
Wed 15:41:  Detailed build log available in "_build.log"
Wed 15:41:  Instantiating manager
Wed 15:41:  Instantiating kernel "examplereportKernel"
Wed 15:41:  Compiling manager (CPU I/O Only)
Wed 15:41:
Wed 15:41:  Compiling kernel "examplereportKernel"
Wed 15:42:  Generating input files (VHDL, netlists, MegaWizard/
   CoreGen)
Wed 15:42:  Running back-end  build (12 phases)
Wed 15:42:  (1/12) - Prepare MaxFile Data (GenerateMaxFileDataFile)
Wed 15:42:  (2/12) - Synthesize DFE Modules (XST)
Wed 15:42:  (3/12) - Link DFE Modules (NGCBuild)
Wed 15:42:  (4/12) - Prepare for Resource Analysis (
   EDIF2MxruBuildPass)
Wed 15:42:  (5/12) - Generate Preliminary Annotated Source Code (
   PreliminaryResourceAnnotationBuildPass)
Wed 15:42:  (6/12) - Report Resource Usage (
   XilinxPreliminaryResourceSummary)
Wed 15:42:
Wed 15:42:  PRELIMINARY RESOURCE USAGE
Wed 15:42:  Logic utilization:      12119 / 297600 (4.07%)
Wed 15:42:    LUTs:                  8349 / 297600 (2.81%)
Wed 15:42:    Primary FFs:           9689 / 297600 (3.26%)
Wed 15:42:  Multipliers (25x18):        2 / 2016   (0.10%)
Wed 15:42:    DSP blocks:               2 / 2016   (0.10%)
Wed 15:42:  Block memory (BRAM18):     22 / 2128   (1.03%)
Wed 15:42:
Wed 15:42:  About to start chip vendor Map/Place/Route toolflow. This
    will take some time.
Wed 15:42:  For this compile, we estimate this process may take up to
    30 minutes.
Wed 15:42:  We recommend running in simulation to verify correctness
   before building a DFE configuration.
Wed 15:42:
Wed 15:42:  (7/12) - Prepare for Placement (NGDBuild)
Wed 15:42:  (8/12) - Place and Route DFE (XilinxMPPR)
Wed 15:42:  Executing MPPR with 1 cost table and 1 thread.
Wed 15:42:  MPPR: Starting 1 cost table
Wed 15:42:  MPPR: Cost table 1 met timing with score 0 (best score 0)
Wed 15:42:  (9/12) - Prepare for Resource Analysis (XDLBuild)
Wed 15:42:  (10/12) - Generate Resource Report (
   XilinxResourceUsageBuildPass)
Wed 15:42:  (11/12) - Generate Annotated Source Code (
   XilinxResourceAnnotationBuildPass)
```

```
Wed 15:42:  (12/12) - Generate MaxFile (GenerateMaxFileXilinx)
Wed 15:43:
Wed 15:43:  FINAL RESOURCE USAGE
Wed 15:43:  Logic utilization:        9444 / 297600 (3.17%)
Wed 15:43:    LUTs:                   7752 / 297600 (2.60%)
Wed 15:43:    Primary FFs:            7484 / 297600 (2.51%)
Wed 15:43:    Secondary FFs:          1845 / 297600 (0.62%)
Wed 15:43:  Multipliers (25x18):         2 / 2016   (0.10%)
Wed 15:43:    DSP blocks:                2 / 2016   (0.10%)
Wed 15:43:  Block memory (BRAM18):      24 / 2128   (1.13%)
Wed 15:43:
Wed 15:43:  MaxFile: "location.max" (MD5Sum:
    a377c2826c92a4f430052775e3aa0517)
Wed 15:43:  Build completed: Wed Jan 25 15:43:42 CET 2017 (took 2
    mins, 0 secs)
```

B Ccode

Listing 6: BiCGSTAB on CPU.

```
1  void matrixVectorDirichlet2D( double _Complex *v, double _Complex *p,
       double h, double k,int N)
2  {
3  /*
4  * calculates v=Ap in 2D.
5  * A is the helmholtz matrix with Dirichlet boundary conditions.
6  * A= -laplace - (1-alpha*I)*k*k.
7  * For every point check if it is near the boundary or not.
8  */
9  int point;
10 for (int j=0;j<N-1;j++){
11         for (int i=0;i<N-1;i++){
12                 point =i+N*j;
13                 v[point]=((4.0L/(h * h))-(1-(alphaProblem * I)) * (k
                       * k)) * p[point];
14                 if(j!=0)
15                         v[point] += (-p[i+N * (j-1)]/(h*h));
16                 if(j!=N-2)
17                         v[point] += (-p[i+N * (j+1)]/(h*h));
18                 if(i!=0)
19                 v[point] += (-p[i-1+N * j]/(h * h));
20                         if(i!=N-2)
21                 v[point] += (-p[i+1+N * j]/(h * h));
22                 }
23         }
24 }
25
26 #ifdef CPU
27 struct Tuple Bi_CGSTABCPU(double  _Complex *u, int max_it, double h,
      double k, double tolerance,int N){
28 /*
29 * The Bi_CGSTAB algorithm solves Au=dirac(sourcePoint),
30 * which is at the center of the grid.
31 * On the CPU
32 */
33 //result variables
34 int iter=0;
35 int flag=0;
36 double error = 0.0L;
37
38 //The point where the source is located
39 int sourcePoint =(N/2-1)*(1+N);
40
41 //variables for the algorithm
42 double _Complex alpha,rho_old,rho_new,beta,omega;
43 rho_old=1.0L;
44 alpha=1.0L;
45 omega=1.0L;
46 double _Complex boven,beneden;
47 size_t sizeBytes = N*N*sizeof(  double _Complex);
```

```
48 double _Complex *r= malloc(sizeBytes);
49 double _Complex *t= malloc(sizeBytes);
50 double _Complex *v= malloc(sizeBytes);
51 double _Complex *p= malloc(sizeBytes);
52 double _Complex *s= malloc(sizeBytes);
53 for (int i =0;i<N*N;i++){
54 r[i]=0.0L;
55 v[i]=0.0L;
56 p[i]=0.0L;
57 }
58 //for-loop integer
59 int point;
60
61 //residu when u_0 = 0 and b=dirac(sourcePoint)
62 r[sourcePoint]=1.0;
63
64 //main part
65 for (iter=1;iter<=max_it;iter++){
66         //Step 3: constants
67         rho_new= r[sourcePoint];
68         printConstantComplex(rho_new,"rho_new",3);
69         if(rho_new==0.0){
70                 flag=-1;
71                 return (struct Tuple) {flag,iter,error};
72         }
73         beta = (rho_new/rho_old)*(alpha/omega);
74         rho_old=rho_new;
75         printConstantComplex(beta,"beta ",2);
76
77         //Step 4 p=r+beta(p-omega*v)
78         for(int j=0;j<N-1;j++){
79                 for(int i=0;i<N-1;i++){
80                         point= i + N*j;
81                         p[point]=r[point]+beta*(p[point]-omega*v[
                                point]);
82                 }
83         }
84         printVector(p,"p",4,N);
85
86         //Step 6 v=Ap
87         matrixVectorDirichlet2D(v,p,h,k,N);
88         printVector(v,"v",4,N);
89
90         //Step 7 alpha
91         alpha=(rho_old/v[sourcePoint]);
92         printConstantComplex(alpha,"alpha",2);
93
94
95         //Step 8 s=r-alpha*v
96         for(int j=0;j<N-1;j++){
97                 for(int i=0;i<N-1;i++){
98                         point= i + N*j;
99                         s[point]=r[point]-alpha*v[point];
100                }
101        }
```

```
102         printVector(s,"s",5,N);
103 
104         Step 9-11 check error
105         error = norm(s,N);
106         if ( error < tolerance ){// ||s|| small enough
107                 for(int j=0;j<N-1;j++){
108                         for(int i=0;i<N-1;i++){
109                                 point= i + N*j;
110                                 u[point] = u[point] + alpha*p[point];
111                         }
112                 }
113                 flag=0;
114                 if(debug>=1)
115                         printf("||s|| is small\n");
116                 return (struct Tuple) {flag,iter,error};
117         }
118 
119         //Step 13 t=As
120         matrixVectorDirichlet2D(t,s,h,k,N);
121         printVector(t,"t",5,N);
122 
123         //Step 14 omega
124         boven=0.0L;
125         beneden =0.0L;
126         for(int j=0;j<N-1;j++){
127                 for(int i=0;i<N-1;i++){
128                         point= i + N*j;
129                         boven+=(t[point]*s[point]);
130                         beneden+=(t[point]*t[point]);
131                 }
132         }
133         omega=boven/beneden;
134         printConstantComplex(omega,"omega",2);
135         if(omega==0.0){
136                 flag=-2;
137                 return (struct Tuple) {flag,iter,error};
138         }
139 
140         //Step 15 u=u+alpha*p+omega*s
141         for(int j=0;j<N-1;j++){
142                 for(int i=0;i<N-1;i++){
143                         point= i + N*j;
144                         u[point]=u[point]+alpha*p[point]+omega*s[
                                    point];
145                 }
146         }
147         printVector(u,"u",4,N);
148 
149         //Step 16 r=s-omega*t
150         for(int j=0;j<N-1;j++){
151                 for(int i=0;i<N-1;i++){
152                         point= i + N*j;
153                         r[point]=s[point]-omega*t[point];
154                 }
155         }
```

```
156         printVector(r,"r",4,N);
157
158         //Step 17-19 check error
159         error=norm(r,N);
160         printConstant(error,"relative error",2);
161
162         if(error<tolerance){                                        // ||r
               || small enough
163                 flag=0;
164                 if(debug>=1)
165                         printf("||r|| is small\n");
166                 return (struct Tuple) {flag,iter,error};
167         }
168 }
169 free(r);
170 free(s);
171 free(v);
172 free(p);
173 free(t);
174
175 //check if max_it is reached
176 if (iter==max_it+1)
177         flag=1;
178 return (struct Tuple) {flag,iter -1,error};
179 }
180 #endif
```

C Kernel Fast Memory

Listing 7: kernel of fmem.

```
1  package bi_cgstabdirichlet13;
2
3  import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;
4  import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;
5  import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.Count.
       Counter;
6  import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.Count.
       Params;
7  import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.Count.
       WrapMode;
8  import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.core.Stream.
       OffsetExpr;
9  import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.memory.Memory
       ;
10 import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEType;
11 import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;
12
13 class Bi_CGSTABDirichlet13Kernel extends Kernel {
14
15 private static final int     dataSize   = 32401;
16 private static final int     indexWidth = 15;
17
18 private static final int     counterWidth = 32;
19 private static final DFEType scalarType = dfeUInt(counterWidth);
20 Bi_CGSTABDirichlet13Kernel(KernelParameters parameters,
21 Bi_CGSTABDirichlet13EngineParameters params,
22 int nxMax,
23 int loopLength,
24 int minStreamLength) {
25 super(parameters);
26 DFEType dataType = params.getFloatingPointType();
27
28 //scalar inputs
29 DFEVar N = io.scalarInput("N", scalarType);
30 DFEVar N2=N*N;
                                                      //N^2
31 DFEVar NU = N.cast(dfeUInt(indexWidth));
                              //N Unsigned
32 DFEVar N2U = N2.cast(dfeUInt(indexWidth));
                              //N^2 Unsigned
33 DFEVar alpha   = io.scalarInput("alpha", dataType);
34 DFEVar omega   = io.scalarInput("omega", dataType);
35 DFEVar rho_old = io.scalarInput("rho_old", dataType);
36 DFEVar rho_new = io.scalarInput("rho_new", dataType);
37 OffsetExpr nx = stream.makeOffsetParam("nx", 4, nxMax);
38 DFEVar h = io.scalarInput("h", dataType);
39 DFEVar k = io.scalarInput("k", dataType);
40 DFEVar sourcePoint = (NU/2-1)*(1+NU);
41
42 //Step 3 calculating beta
```

```
43 DFEVar beta = (rho_new*alpha)/(rho_old*omega);
44
45 //Step 4: p=r+beta*(p-omega*v)
46 Counter step41Counter = count(NU,constant.var(true),WrapMode.
       COUNT_LT_MAX_THEN_WRAP);
47 Counter step42Counter = count(NU,step41Counter.getWrap(),WrapMode.
       STOP_AT_MAX);
48 DFEVar point4 = step41Counter.getCount()+NU*step42Counter.getCount();
49 DFEVar step4Bool = step42Counter.getCount() < NU ? constant.var(true)
        : constant.var(false);
50 //          debug.dfePrintf(step41Counter.getCount()===0&step42Counter.
       getCount()===0, "Tick %d:\n", step41Counter.getCount());
51 //          step4Bool.simWatch("step4");
52 DFEVar r = io.input("r", dataType,step4Bool);
53 DFEVar p = io.input("p", dataType,step4Bool);
54 DFEVar vIn = io.input("vIn", dataType,step4Bool);
55 DFEVar pOut = r + beta*(p-omega*vIn);
56 io.output("pOut", pOut, dataType,step4Bool);
57 Memory<DFEVar> pInput = mem.alloc(dataType, dataSize);
58 Memory<DFEVar> rInput = mem.alloc(dataType, dataSize);
59 pInput.write(point4, pOut, step4Bool);
60 rInput.write(point4, r, step4Bool);
61 //                pOut.simWatch("pOut");
62
63 //Step 6 v=Ap
64 DFEVar vOut= MV(pOut,h,k,nx,N,step41Counter, step42Counter,params);
65 io.output("vOut", vOut, dataType,step4Bool);
66 Memory<DFEVar> vInput = mem.alloc(dataType, dataSize);
67 vInput.write(step41Counter.getCount()+NU*step42Counter.getCount(),
       vOut, step4Bool);
68
69
70 //Step 7 alpha=rho_new/v(sourcepoint)
71 DFEVar setAlpha  = step41Counter.getCount()===(NU/2)-1&step42Counter.
       getCount()===(NU/2)-1;
72 Memory<DFEVar> alpha_new = mem.alloc(dataType, 2);
73 alpha_new.write(constant.var(dfeUInt(1),0), rho_new/vOut, setAlpha);
74 DFEVar alphaBool = point4 > sourcePoint - minStreamLength & point4 <=
        sourcePoint;
75 io.output("alpha_new",rho_new/vOut,dataType,alphaBool);
76
77 //Step 8 s=r-alpha*v
78 Counter step81Counter = count(NU,~step4Bool,WrapMode.
       COUNT_LT_MAX_THEN_WRAP);
79 Counter step82Counter = count(NU,step81Counter.getWrap(),WrapMode.
       STOP_AT_MAX);
80 DFEVar point8 = step81Counter.getCount()+NU*step82Counter.getCount();
81 DFEVar step8Bool = ~step4Bool & step82Counter.getCount() < NU ?
       constant.var(true) : constant.var(false);
82 //          step8Bool.simWatch("step8");
83 DFEVar s = rInput.read(point8)- alpha_new.read(constant.var(dfeUInt
       (1),0))*vInput.read(point8);
84 Memory<DFEVar> sInput = mem.alloc(dataType, dataSize);
85 sInput.write(point8, s, step8Bool);
86 //                s.simWatch("s");
```

```
87
88  //Step 13 t=As
89  DFEVar tOut= MV(s,h,k,nx,N,step81Counter, step82Counter,params);
90  Memory<DFEVar> tInput = mem.alloc(dataType, dataSize);
91  tInput.write(point8, tOut, step8Bool);
92
93
94  //Step 14 omega= <t,s>/<t,t>
95  Counter step14Counter = count(N2U,~step4Bool&~step8Bool,WrapMode.
        STOP_AT_MAX);
96  DFEVar step14Bool = ~step4Bool&~step8Bool&step14Counter.getCount() <
        N2U ? constant.var(true) : constant.var(false);
97  //        step14Bool.simWatch("step14");
98  DFEVar sRam  = sInput.read(step14Counter.getCount());
99  DFEVar tRam = tInput.read(step14Counter.getCount());
100 DFEVar omega_new = summation(tRam*sRam, N2, step14Counter.getCount().
        cast(dfeUInt(counterWidth)), loopLength,params)/
101 summation(tRam*tRam, N2, step14Counter.getCount().cast(dfeUInt(
        counterWidth)), loopLength,params);
102 //        omega_new.simWatch("omega_new");
103 //        debug.dfePrintf(step14Bool&step14Counter.getCount().cast(
        dfeUInt(counterWidth))>= N2 - minStreamLength, "Tick 14: %d: %e\n
        ", step14Counter.getCount(),omega_new);
104 io.output("omega_new", omega_new, dataType, step14Bool&step14Counter.
        getCount().cast(dfeUInt(counterWidth)) >= N2 - minStreamLength);
105
106
107 //Step 15 u=u+alpha*p+omega*s
108 Counter step15Counter = count(N2U,~step4Bool&~step8Bool&~step14Bool,
        WrapMode.STOP_AT_MAX);
109 DFEVar step15Bool = ~step4Bool&~step8Bool&~step14Bool & step15Counter
        .getCount() < N2U ? constant.var(true) : constant.var(false);
110 //        step15Bool.simWatch("step15");
111 DFEVar u = io.input("u", dataType,step15Bool);
112 DFEVar uOut = u+ alpha_new.read(constant.var(dfeUInt(1),0))*pInput.
        read(step15Counter.getCount())+omega_new*sInput.read(step15Counter
        .getCount());
113 io.output("uOut", uOut,dataType,step15Bool);
114 io.output("uOutCPU", uOut,dataType,step15Bool);
115 //        uOut.simWatch("uOut");
116
117
118 //Step 16 r=s-omega*t
119 DFEVar rOut = sInput.read(step15Counter.getCount())- omega_new*
        tInput.read(step15Counter.getCount());
120 io.output("rOut", rOut,dataType,step15Bool);
121 io.output("rOutCPU", rOut,dataType,step15Bool);
122 //        rOut.simWatch("rOut");
123
124 //Step rho_new=r(sourcePoint)
125 DFEVar rho_newBool = step15Counter.getCount() > sourcePoint -
        minStreamLength & step15Counter.getCount() <= sourcePoint;
126 io.output("rho_newOut",rOut,dataType,rho_newBool);
127 }
128 private Counter count(DFEVar size,DFEVar enable,WrapMode wrapMode){
```

```
129 Params coeffsCounterParams = control.count.makeParams(indexWidth)
130 .withMax(size)
131 .withEnable(enable)
132 .withWrapMode(wrapMode);
133
134 Counter coeffsCounter = control.count.makeCounter(coeffsCounterParams
        );
135 return coeffsCounter;
136 }
137 private DFEVar MV(DFEVar input,DFEVar h, DFEVar k, OffsetExpr nx,
        DFEVar N, Counter xCounter, Counter yCounter,
        Bi_CGSTABDirichlet13EngineParameters params){
138 DFEType dataType = params.getFloatingPointType();
139 //elements for sum
140 DFEVar prevX = (xCounter.getCount()>0)?
141 stream.offset(input, -1):       constant.var(dataType,0);
142 DFEVar nextX = (xCounter.getCount().cast(scalarType)<N-1)?
143 stream.offset(input, 1):                constant.var(dataType,0);
144 DFEVar prevY = (yCounter.getCount()>0)?
145 stream.offset(input, -nx):      constant.var(dataType,0);
146 DFEVar nextY = (yCounter.getCount().cast(scalarType)<N-1)?
147 stream.offset(input, nx):       constant.var(dataType,0);
148
149 //sum
150 DFEVar sum = (4/(h*h)-(k*k))*input-(1/(h*h))*prevX-(1/(h*h))
151 *nextX-(1/(h*h))*prevY-(1/(h*h))*nextY;//alpha=0
152
153 //For the ghost points
154 DFEVar number1 =xCounter.getCount().cast(scalarType);
155 DFEVar number2 =yCounter.getCount().cast(scalarType);
156 DFEVar bool =(number1===N-1)|(number2===N-1);
157 return (bool?constant.var(dataType, 0):sum);
158
159 //output
160
161 }
162 private DFEVar summation(DFEVar newTerm, DFEVar N, DFEVar termNum,
        int loopLength,Bi_CGSTABDirichlet13EngineParameters params)
163 {
164 DFEType dataType = params.getFloatingPointType();
165 DFEVar carriedSum = dataType.newInstance(this); // sourceless stream
166 DFEVar sum    = (termNum < loopLength) ? 0.0 : carriedSum;
167 DFEVar newSum = newTerm + sum;
168
169 carriedSum <== stream.offset(newSum, -loopLength);
170
171 DFEVar finalSum = summationTree(newSum, 0, loopLength-1);
172 return finalSum;
173 }
174
175
176 // Recursively produce binary tree of additions. Leaf nodes are
177 // offsets to source stream.
178 private DFEVar summationTree(DFEVar src, int left, int right)
179 {
```

```
180 if (left == right)
181 {
182 return stream.offset(src, -left);
183 }
184 int middle = left + (right - left)/2;
185 return summationTree(src, left, middle) + summationTree(src, middle +
        1, right);
186 }
187 }
```

References

[1] *Multiscale Dataflow Programming*, Version 2014.2.

[2] R Clayton and B Engquist. Absorbing boundary conditions for acoustic and elastic wave equations. *Bulletin of the Seismological Society of America*, 67(6):1529–1540, 1977.

[3] YA Erlangga. *A robust and efficient iterative method for the numerical solution of the Helmholtz equation*. PhD thesis, TU Delft, Delft University of Technology, 2005.

[4] YA Erlangga, CW Oosterlee, and C Vuik. A novel multigrid based preconditioner for heterogeneous Helmholtz problems. *SIAM Journal on Scientific Computing*, 27(4):1471–1492, 2006.

[5] HP Knibbe. *Reduction of computing time for seismic applications based on the Helmholtz equation by Graphics Processing Units*. PhD thesis, TU Delft, Delft University of Technology, 2015.

[6] OL Meijers. The implementation of the Helmholtz problem on a Maxeler machine. Literature study, TU Delft, Delft University of Technology, 2015.

[7] W Roelandts. 15 years of inovation. www.xilinx.com/publications/archives/xcell/Xcell32.pdf, 1999.

[8] HA Van der Vorst. "Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems". *SIAM Journal on scientific and Statistical Computing*, 13(2):631–644, 1992.

[9] MB van Gijzen, YA Erlangga, and C Vuik. Spectral analysis of the discrete helmholtz operator preconditioned with a shifted laplacian. *SIAM Journal on Scientific Computing*, 29(5):1942–1958, 2007.

[10] C Vuik and DJP Lahaye. Lecture notes, Scientific computing (wi4201), TU Delft. 2012.