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

The implementation of the Helmholtz problem on a Maxeler machine

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

for the degree

MASTER OF SCIENCE
in
APPLIED MATHEMATICS

by

Onno Leon Meijers

Delft, the Netherlands
May 2015

MSc THESIS APPLIED MATHEMATICS

“The implementation of the Helmholtz problem on a Maxeler 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

...	...
...	...

May 2015

Delft, the Netherlands

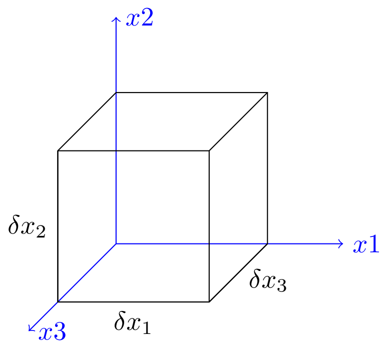

Figure 1: infinitesimally small element V

1 Introduction of the problem

This section will derive the problem with the boundary conditions. Then the numerical approximation will be given and the numerical solver.

1.1 Helmholtz equation

Suppose there is an infinitesimally small fluid element with a volume V in a domain $\Omega \in \mathbb{R}^3$ as showed in Figure 1. Assume zero viscosity, 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:

$$F = m\frac{\partial \mathbf{v}}{\partial t}, \tag{1.1}$$

with m the element 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 (1) and rewriting gives:

$$\nabla p = -\frac{m}{V}\frac{\partial \mathbf{v}}{\partial t} = -\rho_0 \frac{\partial \mathbf{v}}{\partial t}, \tag{1.2}$$

with ρ_0 the static density.

For fluids, Hooke's law reads

$$\frac{\mathrm{d}p}{\mathrm{d}t} = -K\frac{\frac{\mathrm{d}V}{\mathrm{d}t}}{V}, \tag{1.3}$$

with K the compression modulus of the media. For small spatial variations we assume that the cube remains a cube,

$$\begin{aligned}
\frac{\frac{\mathrm{d}V}{\mathrm{d}t}}{V} &= \frac{\frac{\mathrm{d}\delta x_1}{\mathrm{d}t}}{\delta x_1} + \frac{\frac{\mathrm{d}\delta x_2}{\mathrm{d}t}}{\delta x_2} + \frac{\frac{\mathrm{d}\delta x_3}{\mathrm{d}t}}{\delta x_3} \\
&= \frac{(v_1)_{x_1+\delta x_1} - (v_1)_{x_1}}{\delta x_1} + \frac{(v_2)_{x_2+\delta x_2} - (v_2)_{x_2}}{\delta x_2} + \frac{(v_3)_{x_3+\delta x_3} - (v_3)_{x_3}}{\delta x_3} \\
&= \left(\frac{\partial v_1}{\partial x_1} + \frac{\partial v_2}{\partial x_2} + \frac{\partial v_3}{\partial x_3}\right) \\
&= (\nabla \cdot \mathbf{v}) .
\end{aligned} \tag{1.4}$$

By using (1.4) and since $\frac{\mathrm{d}p}{\mathrm{d}t} = \frac{\partial p}{\partial t}$, (1.3) becomes

$$(\nabla \cdot \mathbf{v}) = -\frac{1}{K}\frac{\partial p}{\partial t}. \tag{1.5}$$

Applying the gradient operator to (1.2) gives the equation

$$\nabla \cdot \left(-\frac{1}{\rho_0} \nabla p \right) = \frac{\partial}{\partial t} \nabla \cdot \mathbf{v} \tag{1.6}$$

Substitution of (1.5) into (1.6) and assuming that the gradient density ρ_0 is infinitesimally small results in

$$\Delta p = \frac{1}{c^2} \frac{\partial^2 p}{\partial t^2}, \tag{1.7}$$

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

We are concerned with time-harmonic waves of time-dependent pressure of the form

$$p(\mathbf{x}, t) = u(\mathbf{x}) \exp(-\mathbf{i}\omega_w t) \tag{1.8}$$

where $\omega_w > 0$ and $\mathbf{i}$ denote the angular frequency and the imaginary unit, respectively. Substituting (1.8) into (1.7) results in

$$-\Delta u(\mathbf{x}) - k^2(\mathbf{x}) u(\mathbf{x}) =: \mathcal{A} u(\mathbf{x}) = 0, \tag{1.9}$$

With $\mathcal{A}$ the Helmholtz operator. In (1.9) 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, we also find 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 (1.9) is known as the Helmholtz equation for the pressure.

When a source term is introduced and the assumption is made that this source term is also time-harmonic, a more general formulation of the Helmholtz equation is obtained:

$$\mathcal{A} u(\mathbf{x}) := -\Delta u(\mathbf{x}) - k^2(\mathbf{x}) u(\mathbf{x}) = g(\mathbf{x}), \tag{1.10}$$

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

$$\mathcal{A} u(\mathbf{x}) := -\Delta u(\mathbf{x}) - (1 - \alpha \mathbf{i}) k^2(\mathbf{x}) u(\mathbf{x}) = g(\mathbf{x}), \tag{1.11}$$

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 (1.10).

1.2 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. If we 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; 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 (1.9) can be evaluated in a spherical coordinate system. In the spherical coordinate system, the Helmholtz equation transforms into

$$-(ru)'' - k^2(ru) = 0, \tag{1.12}$$

with a general solution of the form

$$u(r) = A \frac{\cos(kr)}{r} + B \frac{\sin(kr)}{r}. \tag{1.13}$$

Combining (1.8) with (1.13) results in

$$p(r, t) = A^* \frac{\exp(\mathbf{i}(kr - \omega_w t))}{r} + B^* \frac{\exp(-\mathbf{i}(kr - \omega_w t))}{r}. \tag{1.14}$$

Consider surfaces of constant phase. Then it easy to see that as $\omega_w t$ increases in time the first term of the right-hand side of (1.14) 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 (1.14) cannot be a physical solution. Therefore,

$$p(r,t) = A^* \frac{\exp(\mathbf{i}(kr - \omega_w t))}{r}. \tag{1.15}$$

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

$$\lim_{r\to\infty} (-u' - \mathbf{i}ku) \sim o(r^{-1}), \tag{1.16}$$

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 (1.16) is known as the Sommerfeld radiation condition for spherical coordinate system [4],[3].

1.3 Finite difference approximations

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. Consider 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 axis. We introduce the standard lexicographical numbering and denote 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. By using the central difference scheme for the second order differential term

$$\frac{\partial^2 u}{\partial x_1^2} \approx \frac{1}{h^2}\left(u_{i_1+1,i_2,i_3} - 2u_{i_1,i_2,i_3} + u_{i_1-1,i_2,i_3}\right), \tag{1.17}$$

$$\frac{\partial^2 u}{\partial x_2^2} \approx \frac{1}{h^2}\left(u_{i_1,i_2+1,i_3} - 2u_{i_1,i_2,i_3} + u_{i_1,i_2-1,i_3}\right) \text{ and} \tag{1.18}$$

$$\frac{\partial^2 u}{\partial x_3^2} \approx \frac{1}{h^2}\left(u_{i_1,i_2,i_3+1} - 2u_{i_1,i_2,i_3} + u_{i_1,i_2,i_3-1}\right), \tag{1.19}$$

(1.11) can now be approximated in Ω_h by the equation

$$\begin{aligned} &-\frac{1}{h^2}(u_{i_1-1,i_2,i_3} + u_{i_1,i_2-1,i_3} + u_{i_1,i_2,i_3-1} \\ &-6u_{i_1,i_2,i_3} + u_{i_1+1,i_2,i_3} + u_{i_1,i_2+1,i_3} + u_{i_1,i_2,i_3+1}) \\ &-(1-\alpha\mathbf{i})k^2 u_{i_1,i_2,i_3} = g_{i_1,i_2,i_3}, \text{ for } i_1, i_2, i_3 = 1, 2, \ldots, \sqrt[3]{N}. \end{aligned} \tag{1.20}$$

(1.20) can be written in stencil notation as

$$\begin{aligned} &A_{h,7p} \widehat{=} \\ &-\frac{1}{h^2}\left(\begin{bmatrix} & 0 & \\ 0 & 1 & 0 \\ & 0 & \end{bmatrix}_{i_3-1} \begin{bmatrix} & 1 & \\ 1 & -6+(1-\alpha\mathbf{i})k^2h^2 & 1 \\ & 1 & \end{bmatrix}_{i_3} \begin{bmatrix} & 0 & \\ 0 & 1 & 0 \\ & 0 & \end{bmatrix}_{i_3+1}\right), \end{aligned} \tag{1.21}$$

with $h = \frac{1}{\sqrt[3]{N}}$ the grid size. For smooth solutions in uniform grids this approximation is of $O(h^2)$ accuracy.

Left to do are the boundary conditions given in equation (1.16).Those boundary conditions become

$$\left(-\frac{\partial}{\partial \eta} - \mathbf{i}k\right) u = 0, \tag{1.22}$$

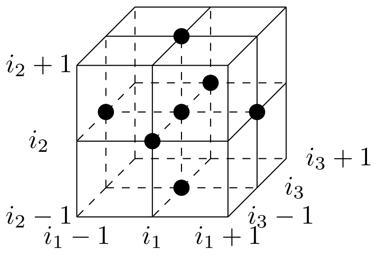

Figure 2: 3D finite difference 7-point stencil

where η is the outward unit normal component to the boundary. Note that 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

$$\begin{aligned} \left(\frac{\mathrm{d}}{\mathrm{d}x} - \mathbf{i}k\right) u_{0,i_2,i_3} &= 0 \\ \frac{u_{1,i_2,i_3} - u_{0,i_2,i_3}}{h} - \mathbf{i}k u_{0,i_2,i_3} &= 0 \\ (1+\mathbf{i}kh)\, u_{0,i_2,i_3} &= u_{1,i_2,i_3} \\ u_{0,i_2,i_3} &= \frac{u_{1,i_2,i_3}}{1+\mathbf{i}kh}. \end{aligned} \tag{1.23}$$

1.4 Krylov subspace iterative methods

In this section the method for solving the linear system will be explained for equation (1.20) which is of the form

$$Au = g. \tag{1.24}$$

A is a square, complex valued and in general not a symmetric nor positive definite matrix.

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

$$\mathcal{K}^j(A; r^o) = \text{span}\{r^0, Ar^0, \dots, A^{j-1}r^0\}, \tag{1.25}$$

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

$$u^j \in u^0 + \mathcal{K}^j(A; r^0), \;\; j > 1. \tag{1.26}$$

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

$$V^j = [v^1, v^2, \dots, v^j] \in \mathcal{K}^j. \tag{1.27}$$

Combining (1.26) with (1.27) gives the expression

$$u^j = u^0 + V^j y^j, \tag{1.28}$$

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

$$\begin{aligned} r^j &= g - Au^j \\ &= g - Au^0 + AV^j y^j \\ &= r^0 - AV^j y^j. \end{aligned} \tag{1.29}$$

From equation (1.29) is deduced that a Krylov subspace method relies on constructing a basis V^j and the vector y^j. First the conjugate gradient(CG) method will be explained then the Bi-CGSTAB method

1.5 Conjugate gradient method

For the CG method the matrix A needs to be symmetric, positive definite(SPD). Note that the matrix in our problem does not have these properties in general. In CG the new vector $u^j \in \mathcal{K}^j(A; r^0)$ is constructed such that $||u - u^j||_A$ is minimal, where $||u||_A = (Au, u)^{\frac{1}{2}}$ and (a, b) the standard Hermitian inner product. Therefore, u^{j+1} is expressed as

$$u^{j+1} = u^j + \alpha^j p^j, \tag{1.30}$$

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

$$r^{j+1} = r^j - \alpha^j A p^j. \tag{1.31}$$

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

$$(r^j - \alpha^j A p^j, r^j) = 0, \tag{1.32}$$

which gives

$$\alpha^j = \frac{(r^j, r^j)}{(Ap^j, r^j)}. \tag{1.33}$$

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

$$p^{j+1} = r^{j+1} + \beta^j p^j, \tag{1.34}$$

such that p^{j+1} is A-orthogonal to p^j, i.e. $(Ap^{j+1}, p^j) = 0$. Then the denominator in (1.33) can be written as $(Ap^j, p^j - \beta^{j-1} p^{j-1}) = (Ap^j, p^j)$. Also

$$\begin{aligned}
(Ap^{j+1}, p^j) &= 0 \\
(A(r^{j+1} + \beta^j p^j), p^j) &= 0 \\
(\beta^j A p^j, p^j) &= -(Ar^{j+1}, p^j) \\
\beta^j &= -\frac{(Ar^{j+1}, p^j)}{(Ap^j, p^j)}
\end{aligned} \tag{1.35}$$

Symmetry of A and orthogonality of the r^j's results in

$$\begin{aligned}
\beta^j &= -\frac{(Ap^j, r^{j+1})}{(Ap^j, p^j)} \\
&= -\frac{(\frac{r^j - r^{j+1}}{\alpha^j}, r^{j+1})}{(Ap^j, p^j)} \\
&= \frac{1}{\alpha^j} \frac{(r^{j+1}, r^{j+1})}{(Ap^j, p^j)} \\
&= \frac{(Ap^j, p^j)}{(r^j, r^j)} \frac{(r^{j+1}, r^{j+1})}{(Ap^j, p^j)} \\
&= \frac{(r^{j+1}, r^{j+1})}{(r^j, r^j)}.
\end{aligned} \tag{1.36}$$

The summary of the CG method is presented in Algorithm 1.1.

Algorithm 1.1 has the good properties that it only requires short recurrences and only one matrix-vector multiplication and a few vector updates have to be calculated per iteration. However, this algorithm may not converge for the Helmholtz problem because SPD is not guaranteed. Therefore, the Bi-CGSTAB method will be presented.

Algorithm 1.1 Conjugate gradient, CG

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

1.6 Bi-CGSTAB

For non-symmetric matrices an iterative method is developed based on the bi-Lanczos method. Bi-CGSTAB [5] is based on the following observation. Iterates u^i are generated so that $r^i = \tilde{P}_i(A)P_i(A)r^0$ with an i^{th} degree polynomial $\tilde{P}_i(A)$. A possibility is to take for $\tilde{P}_i(A)$ a polynomial of the form

$$Q_i(x) = (1 - \omega_1 x)(1 - \omega_2 x)(1 - \omega_i x), \tag{1.37}$$

and to select suitable constants $\omega_i \in \mathbb{R}$. This expression leads to an almost trivial recurrence relation for the Q_i's. In Bi-CGSTAB, ω_i in the i^{th} iteration step is chosen as to minimise r^i, with respect to ω_i, for residual that can be written as $r^i = Q_i(A)P_i(A)r^0$[7].

The preconditioned Bi-CGSTAB algorithm for solving the linear system $Au = b$, with preconditioner M is presented in Algorithm 1.2[7].

Algorithm 1.2 Bi-CGSTAB

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

The matrix M in Algorithm 1.2 represents the preconditioning matrix and the way of preconditioning[5]. Algorithm 1.2 in fact carries out the Bi-CGSTAB procedure for the explicitly post-conditioned linear system

$$AM^{-1}y = g, \tag{1.38}$$

but the vectors y^i have been transformed back to the vectors u^i corresponding to the original system $Au = g$. This solver will be used for our problem due to the spectral properties of our problem [6].

1.7 Shifted Laplacian Preconditioner

Because the convergence of the Bi-CGSTAB method is too slow a preconditioner is needed. The original system can be replaced by an equivalent preconditioned system

$$AM^{-1}y = g, \; y = Mu, \tag{1.39}$$

where the system $y = Mu$ is easy to solve. The matrix AM^{-1} is well-conditioned, so that the convergence of Bi-CGSTAB (and any other Krylov subspace methods) is improved. As the preconditioner for Bi-CGSTAB we consider the Shifted Laplace preconditioner introduced by Erlangga, see [4], which is based on the following operator

$$\mathscr{M}_{\beta_1,\beta_2} = -\Delta - (\beta_1 - \beta_2\mathbf{i})k^2, \tag{1.40}$$

with the same boundary conditions as $\mathscr{A}$ in equation (1.11). The precondition steps are calculated with a standard multigrid V-cycle.

2 Dataflow computing on Maxeler machine

In this section the theoretical implementation will be explained. First an introduction will be given about the Field-Programmable Gate Array (FPGA) and Maxeler. Then the functions that are needed to implement the preconditioned Bi-CGSTAB are presented. Finally the implementation ideas for the algorithm are given.

2.1 FPGA and Maxeler introduction

A field-programmable gate array (FPGA) is an integrated circuit (IC) that can be programmed after manufacture and after it has been installed in the field. A Maxeler machine is an example of an FPGA. Maxeler has developed a machine and software so that a lot of people can use the benefits of using FPGAs. The software handles for the user the layout of the calculation on the FPGA. The FPGA consist of a lot of calculation and storage units. Each unit is connected with multiple other units. The software calculates which connections to use to get the best performance. Maxelers software works using the dataflow model. The user needs to implement the kernel and the data streams and the software handles the rest. The implementation will be explained in Section 2.2. The FPGA is also called a Dataflow Engine (DFE).

2.2 Implementation of an example on a Maxeler machine

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 problem.

When you want to implement your problem on a dataflow computer, then as a CPU programmer you must first change the idea how the problem is programmed. Instead of thinking what has to happen about the whole vector you need to think about what happens per element when it gets through a kernel and follow the data. The building of a small program will be explained by the example presented in Listing 1. For a vector, every element is multiplied by two and the element before that is added and the result is saved in a vector.

Listing 1: Example in C

```
for(int i=1;i<N;i++){
  result[i] =  2*input[i]+input[i-1];
}
```

This example will be build from the beginning. Hence, we start with the most simple kernel which does not do anything with the data and sends it through. The following code represents the kernel of this.

```
package examplereport;

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

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

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

This kernel stores the input in a DFEVar. DFEVar is the class of variables that can take all the primitive instances like single and double precision. The input needs the stream-name so that it can be guided properly from for example CPU to DFE and the type of the element send, which is single precision in this case. Then the input needs to be send back with a stream-name, the variable and the data type. The parameters object also needs to passed on because it is used internally within the Kernel class and this is done in java with the super call.

Then the result is stored and multiplied by 2.

```
DFEVar input = io.input("input", dfeFloat(8, 24));
DFEVar result = 2*input;
io.output("output", result, dfeFloat(8, 24));
```

Next the previous element is needed. Therefore, the function stream.offset is used. This function needs two arguments, the variable and the integer how big the offset must be. If the offset is negative, then the previous number is used.

```
DFEVar input = io.input("input", dfeFloat(8, 24));
DFEVar prev = stream.offset(input, -1);
DFEVar result = prev + 2*input;
io.output("output", result, dfeFloat(8, 24));
```

Note that for the first element of the stream, stream.offset is not defined and it does not return a value.

For convenience the data type is also stored.

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

Then consider the boundary term . For this a counter is needed to know if this tick something has to happen.

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

A counter is build up from 2 steps. First the parameters 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 511. Then properties can be add like the maximum which is 384 in this case or link counters together. Then the counter is defined.

Now the definition of the previous argument needs to be zero when the counter is zero.

```
    DFEVar prev = (counter.getCount()>0 ? stream.offset(input, -1):constant.var(
        singleType,0));
```

If the counter is larger than zero then the offset is taken, otherwise zero is taken. Note that in this statement both values are calculated and in run-time the correct value is taken.

This all will result in the kernel to solve this small example.

Listing 2: Example in maxj

```
package examplereport;
```

```
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;

class examplereportKernel extends Kernel {
  private static final DFEType singleType = dfeFloat(8, 24);
  protected examplereportKernel(KernelParameters parameters) {
    super(parameters);
    Params params = control.count.makeParams(9).withMax(384);
    Counter counter = control.count.makeCounter(params);
    DFEVar input = io.input("input", singleType);
    DFEVar prev = (counter.getCount()>0 ? stream.offset(input, -1):constant.var(
        singleType,0));
    DFEVar result = prev + 2*input;
    io.output("output", result, singleType);
  }
}
```

Now the kernel is build and the graph of this kernel is given in Figure 2.2. The green rectangles contain numbers. The operators are clear. The rhombus is the offset operator, taken for the input. The hexagon is the counter. The counter takes two number in this case, the maximum and the step-size. The trapezoid is the multiplexer node for taking decisions. In this case it choose between the offset when the larger then operator is true, otherwise zero is taken.

Next the data needs to be streamed through the kernel. This is done with the manager.

Listing 3: Manager of example

```
package examplereport;

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 examplereportManager {

  private static final String s_kernelName = "examplereportKernel";

  public static void main(String[] args) {
    examplereportEngineParameters params = new examplereportEngineParameters(args);
    Manager manager = new Manager(params);
    Kernel kernel   = new examplereportKernel(manager.makeKernelParameters(
        s_kernelName));
    manager.setKernel(kernel);
    manager.setIO(IOType.ALL_CPU); // Connect all kernel ports to the CPU
    manager.createSLiCinterface();
    manager.build();
  }
}
```

In Listing 3 the manager is displayed. First the parameters are set and put in a new manager. Then the kernel is build and added to the manager. Then the data streams are defined. In this case all streams are from and back to the CPU. this is done with

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

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 most simple and enough for this example. Then the manager is build.

2.3 Bi-CGSTAB on the dataflow machine

In this subsection the Bi-CGSTAB Algorithm 1.2 will be explained step by step. To start a few assumption will be made. First is that the $\bar{r}^0 = r^0$. Second is that the begin-solution u^0 is a zero-vector. This will

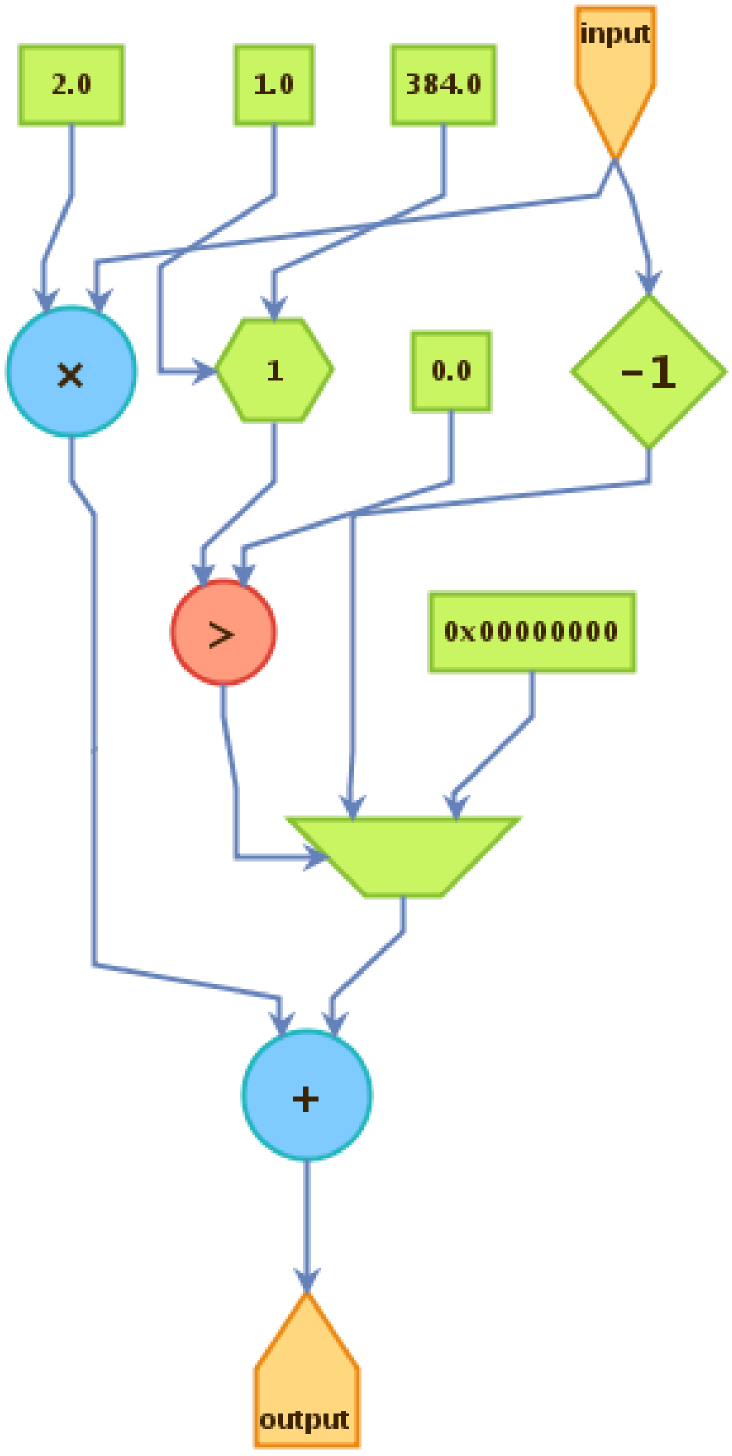

Figure 3: Kernel of example

result in that the error in the setup step will be $r^0 = g$ and that $\bar{r}^0 = r^0 = g$. Also k from Equation 1.11 will be assumed constant for now. The last assumption is that the source function is a point-source. This means that $g(x, y, z) = \delta_{x_s,y_s,z_s}(x, y, z)$ with δ the Dirac delta function at point x_s, y_s, z_s. This source is located at the centre for example. All these assumptions and some renaming of the variables result in Algorithm 2.1. Bold variables are vectors. Note that the intention is to use the Maxeler machine as a replacement for the CPU.

The calculation have to be laid down in the space and 3 large kernels need to be implemented. Then the manager needs to make sure the streams flow accordingly. Every step will be explained what needs to happen. But first the 3 kernels shall be explained. The first kernel is the convolve kernel. This calculated the matrix-vector multiplication and is a optimised calculation by Maxeler in the MaxGenFD package. The second kernel is an inproduct kernel. For 2 stream every element needs to be multiplied and added. The third kernel has 3 streams of data and 2 constants like in Figure 4. This will be called the addition kernel. Note that it has to be tested if we also need an addition kernel with 2 streams and 1 constant or that this will be implemented in the addition kernel with a dummy stream and zero constant.

Algorithm 2.1 Bi-CGSTAB 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:    $\rho_{new} = \mathbf{r}(x_s, y_s, z_s)$; $\beta = \frac{\rho_{new}}{\rho_{old}} \frac{\alpha}{\omega}$; $\rho_{old} = \rho_{new}$;
4:    $\mathbf{p} = \mathbf{r} + \beta(\mathbf{p} - \omega\mathbf{v})$;
5:    Solve $M\hat{\mathbf{p}} = \mathbf{p}$;
6:    $\mathbf{v} = A\hat{\mathbf{p}}$;
7:    $\alpha = \frac{\rho_{old}}{\mathbf{v}(x_s,y_s,z_s)}$;
8:    $\mathbf{s} = \mathbf{r} - \alpha\mathbf{v}$;
9:    if $||\mathbf{s}||$ small enough then
10:     $\mathbf{u} = \mathbf{u} + \alpha\hat{\mathbf{p}}$; quit;
11:   end if
12:   Solve $M\mathbf{z} = \mathbf{s}$;
13:   $\mathbf{t} = A\mathbf{z}$;
14:   $\omega = \frac{(\mathbf{t},\mathbf{s})}{(\mathbf{t},\mathbf{t})}$;
15:   $\mathbf{u} = \mathbf{u} + \alpha\hat{\mathbf{p}} + \omega\mathbf{z}$;
16:   $\mathbf{r} = \mathbf{s} - \omega\mathbf{t}$;
17:   if $||\mathbf{r}||$ is small enough then
18:     quit;
19:   end if
20: end for
```

Step 3 calculates new constants. In this step read the correct value of the vector $\mathbf{r}$. Then calculate the constant β, for this read the other constants. Write the value of ρ_{old} back.

Step 4 is an addition kernel with the streams $\mathbf{r}, \mathbf{p}, \mathbf{v}$ and constants β and $\beta \cdot -\omega$.

Step 5 is the precondition step explained in Section 2.4 or left out without preconditioning.

Step 6 is a convolve kernel that convolves $\hat{\mathbf{p}}$ into $\mathbf{v}$. Note that this step will become more difficult when k is not constant any more.

Step 7 need to calculate the new value of α like in step 3 it needs to read the proper value from the vector $\mathbf{v}$ and divide ρ_{old} by it.

Step 8 is an addition kernel with streams $\mathbf{r}, \mathbf{v}$, a dummy stream and the constants α and zero.

Step 9 needs to calculate the norm of $\mathbf{s}$. This is an inproduct kernel. Note that it will be tested if this step is worth it.

Step 12 is the precondition step and the same as step 5 except with different input vector $\mathbf{s}$.

Step 13 is the same as Step 6, but this time $\mathbf{z}$ is convolved into $\mathbf{t}$.

Step 14 contains the calculation of 2 inner-products.

Step 15 is an addition kernel with streams $\mathbf{u}, \hat{\mathbf{p}}, \mathbf{z}$, and constants, α, ω,

Step 16 is an addition kernel with streams $\mathbf{s}, \mathbf{t}$, a dummy stream and the constants $-\omega$ and zero.

Step 17 calculates a norm like in step 9 and then stops the algorithm if the precision has been achieved.

2.4 Multigrid on the dataflow machine

The multigrid method will be presented in Algorithm 2.2 with a damped-Jacobi pre- and post-smoother.

The whole algorithm is recursive, however the level is known. Therefore, how many times this algorithm has to repeat itself and all the vectors it requires, is known.

Step 1 is an addition kernel combined with the convolve operator of the MaxGenFD package.

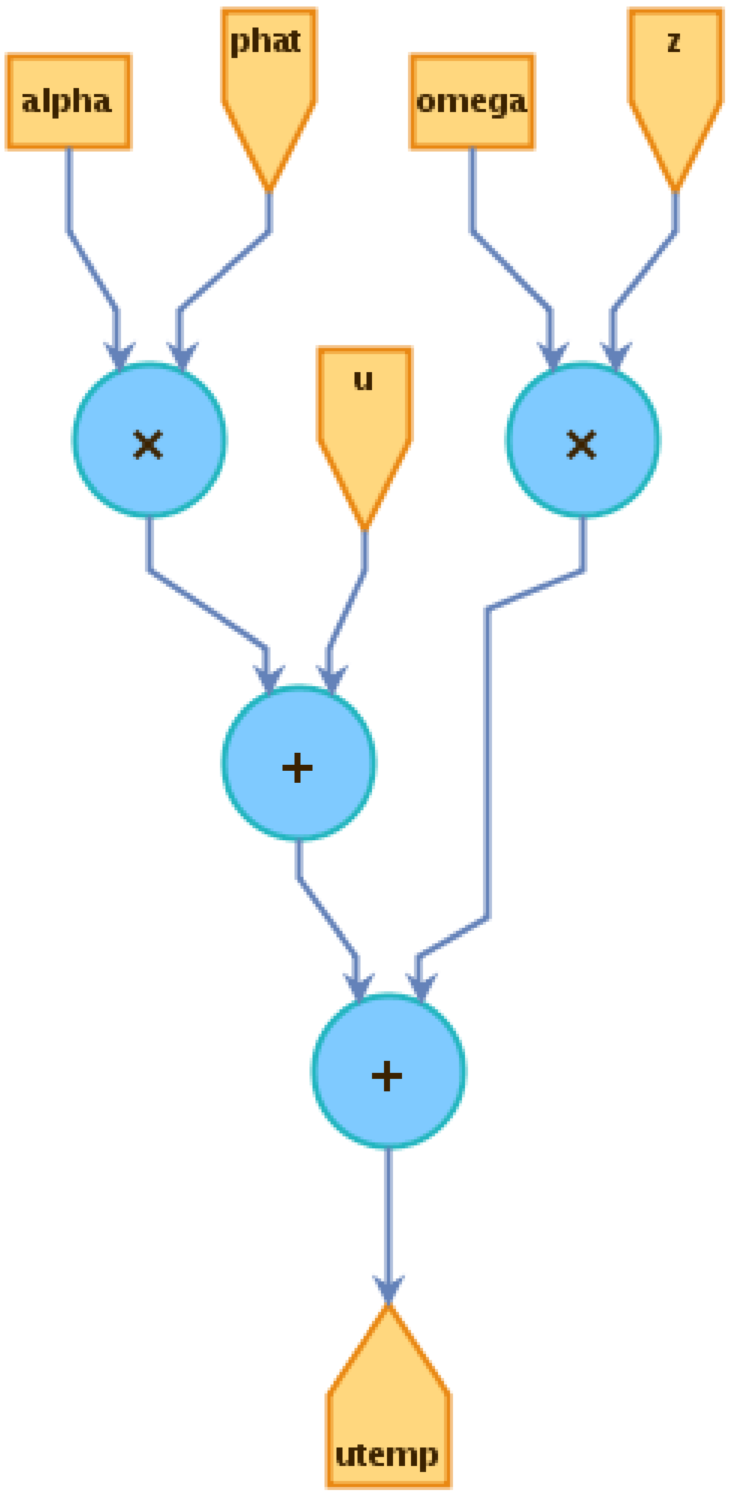

Figure 4: Kernel graph diagram for Bi-CGSTAB step 15

Step 2 to 5 is in most cases for $c_1 \in \{0, 1\}$. Therefore the for-loop is only 1 statement.

Step 3 is an addition kernel with the streams x and r_{level} and constant $\omega \cdot \text{diag}(M)^{-1}$. Note that the diagonal of M is not constant when the boundary conditions change to Sommerfeld Boundary conditions.

Step 4 is the same as step 1.

Step 6 is a stream with 2^d, with d the dimension, elements per output. And those inputs are multiplied

Algorithm 2.2 Multigrid preconditioning, solve $Mx = b$

```
MG($level, x, b$):
1:  $r_{level} = b - Mx$;
2:  for $i = 1, .., c_1$ do
3:      $x = x + \omega \text{diag}(M)^{-1} r_{level}$;
4:      $r_{level} = b - Mx$;
5:  end for
6:  $r_{level-1} = \text{restrict}(r_{level})$;
7:  $\hat{v}_{level-1} = 0$;
8:  if $level - 1 == 1$ then
9:      Solve $M_1 \hat{v} = r_1$;
10: else
11:     MG($level - 1, \hat{v}_{level-1}, r_{level-1}$);
12: end if
13: $\hat{v}_{level} = \text{interpolate}(\hat{v}_{level-1})$;
14: $x = x + \hat{v}_{level}$;
15: for $i = 1, .., c_2$ do
16:     $x = x + \omega \text{diag}(M)^{-1}(b - Mx)$;
17: end for
```

with a stencil, which is set from the beginning.

Step 7 is allocating space which is to be done in the setup-phase or maybe when the current level is small enough, saved locally.

Step 8 to 12 solves the 8 by 8 system by a simple multiplication if the algorithm is run until $level-1 = 1$. This is because there is only one point in the discretisation, the rest are ghost points. When the algorithm only decreases the system a number of levels a kernel is needed that solves the small system. If this level is not reached yet the algorithm is recursively called with the new vectors.

Step 13 is the reverse of step 5 with only a different stencil. For every element received, 2^d, with d the dimension, output elements have to be streamed back.

Step 14 streams the result and the update vector, adds every element and store them back.

Step 15 to 17 is the same as step 1 to 5, without step of r_{level} in between.

For the MG method to work nicely the following discretisation will be used.

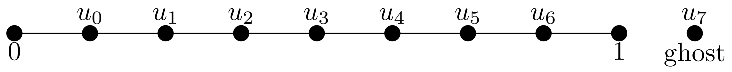

Figure 5: 1D discretisation for $h = \frac{1}{8}$

With this discretisation every level will align perfectly and a normal prolongation and restriction operator can be used. Note that also a ghost element is added. This is because the Maxeler machine can only send streams of a multiple of 16 bytes.

3 Research questions

Implementation plan:

1. The first thing to do is to be able to find out if our implementation is correct. Correctness is within a certain order, because it will never be the same due to different truncation protocols. Because the algorithm is already implemented in MATLAB, a program is needed to check the values of the solution. Because only levels will be checked it is easy to save the solutions in files and check against them every time. Also a visual check will be needed. Therefore, it is needed to write the data in a file and let that file be read by a MATLAB program and build a plot from there.

2. Implement a working matrix vector multiplication with a stencil. Note that the Dirichlet boundary conditions will be used for simplicity.

3. The first full problem to implement will be the 2D-Helmholtz problem with Bi-CGSTAB and Dirichlet boundary conditions. Note that the problem here is to have a working Bi-CGSTAB algorithm, because the Helmholtz problem with Dirichlet boundary conditions is in the stencil. The Bi-CGSTAB algorithm can be separated in a number of kernels. Therefore, the steps in Section 2.3 should be implemented and checked for every step.

4. Then the preconditioner needs to be implemented like in Item 3. Section 2.4 needs to be implemented step by step. There are 2 implementations possible; Galerkin method with a matrix and the matrix free implementation with a direct result from the discretisation. These implementations are not equal. Therefore it needs to be checked which works better in terms of performance and number of iterations.

5. Then the next step will be a small adaptation to change the boundary conditions to radiation boundary conditions.

6. The last constant problem to implement will be the preconditioned 3D-Helmholtz problem with Bi-CGSTAB and radiation boundary conditions.

7. The variable "$k = k(x, y(, z))''$. Then the 2D-problem with non-constant k needs to be implemented first.

8. Finally the 3D-Helmholtz problem with non-constant k can be implemented.

After each implementation step, except for step 1, the following questions need to be answered for the implementation on the Maxeler machine:

- Does the method converge approximately in the same amount of iterations than on the CPU?
- How many of the calculation and storage units of the Maxeler machine are used and can this be improved?
- Is the implementation on the Maxeler machine faster than on the CPU?

This results in a number of overall research questions:

- Can the Bi-CGSTAB algorithm with preconditioning be implemented on the Maxeler machine?
- What is the speedup achieved by implementing the Bi-CGSTAB algorithm on the Maxeler machine?
- What is the speedup achieved by implementing the preconditioned Bi-CGSTAB algorithm on the Maxeler machine?
- What is faster a matrix implementation or a matrix-free implementation?
- Is it faster to do not check the intermediate error?
- What precision is needed in the algorithm and what precision is needed for the intermediate steps? Is there an optimal choice for precision versus usage of the units of the Maxeler machine?
- How big can the grid-size become and still converge in reasonable time?
- Can an analytic model be developed that is able to optimise the algorithm layout onto a given spatial computing substrate, carefully balancing its computing, I/O and memory resources to the workload at hand? Is this model able to give insight in the performance increase that is to be expected?
- What speedup can be expected for various large three-dimensional problems on the spatial computing substrate compared to other best-in-class approaches?

References

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

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

[3] Yogi A Erlangga, Cornelis W Oosterlee, and Cornelis Vuik. A novel multigrid based preconditioner for heterogeneous helmholtz problems. *SIAM Journal on Scientific Computing*, 27(4):1471–1492, 2006.

[4] Yogi Ahmad Erlangga. *A robust and efficient iterative method for the numerical solution of the Helmholtz equation.* TU Delft, Delft University of Technology, 2005.

[5] Henk A 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.

[6] Martin B van Gijzen, Yogi A Erlangga, and Cornelis Vuik. Spectral analysis of the discrete helmholtz operator preconditioned with a shifted laplacian. *SIAM Journal on Scientific Computing*, 29(5):1942–1958, 2007.

[7] C Vuik and DJP Lahaye. Lecture notes scientific computing (wi4201).