Graduate thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Anders Solberg Pedersen

Seismic wave modelling on GPU clusters

Graduate thesis in Electronic systems design
Supervisor: Espen Birger Raknes
June 2019

NTNU
Norwegian University of
Science and Technology

Anders Solberg Pedersen

Seismic wave modelling on GPU clusters

Graduate thesis in Electronic systems design
Supervisor: Espen Birger Raknes
June 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

NTNU
Kunnskap for en bedre verden

Contents

List of Figures

List of Tables

List of Algorithms

List of Source Code

Abstract

We develop an application for simulating seismic waves on a GPU cluster using CUDA and MPI. By using the spectral element method to numerically solve the 3D elastic wave equation the problem can be paralellized and solved efficiently. The implementation was tested using up to 16 Nvidia P100 GPUs and the scaling behaviour shows a 14% increase in execution time between 1 and 16 GPUs. The numerical accuracy is examined by comparing to the analytical solution and with another implementation in an isotropic homogeneous medium when using a uniform mesh, both comparisons confirming the correctness of the implementation.

1 Introduction

Modelling seismic waves is important in several disciplines, such as hydrocarbon exploration, sub-surface imaging, and earthquake modelling. The problems encountered in these areas are often very computationally demanding as simulating wave propagation in an earth-size model can take days even when running on hundreds of CPU cores (S. Tsuboi and D. Komatitsch 2003). One method where the increased performance of the GPU would be beneficial is Full Waveform Inversion. This method can provide very accurate estimations of subsurface parameters, but it requires solving the wave equation several times. This method used to rely on simplifications to be computationally feasible, but the increased availability of powerful hardware including GPUs in computing clusters could allow this method to be used to model more complicated phenomena according to Raknes and Arntsen 2017. Because of this we are interested in developing an implementation with an even higher performance, allowing us to handle larger systems with even better accuracy. In order to do this we develop and implement a program for modelling seismic wave propagation using the Spectral Element Method (SEM) accelerated by multiple Graphics Processing Unit (GPU).

Several approaches are available when it comes to modelling seismic waves. The choices depend on what kind of information one is interested in and what kind of media are in question. Situations can include acoustic(fluid), elastic(solid), coupled acoustic/elastic or poroelastic, either isotropic or anisotropic. Naturally, a full implementation is able to cover all of these situations, but in this report we will implement just one variant: the elastic wave equation in isotropic media. The reason for this is its usefulness in modelling many real-world situations accurately.

Solving the elastic wave equation in an isotropic medium requires the application of some numerical method. These can be very computationally expensive which is why we use the Spectral Element Method. The reason for choosing the SEM is that it gives high-accuracy results and it can be used with unstructured meshes, like with Finite Element Methods (FEM). One significant advantage is that we can use numerical quadrature with the Gauss-Lobatto-Legendre points on hexahedral elements, resulting in a diagonal mass matrix in the linear system. This leads to a more efficient implementation than FEM, since no matrices need to be inverted in order to solve the system. A drawback of the method is that the mesh must consist of hexahedral elements, making the construction of meshes more difficult.

Since the method yields a diagonal linear system, we see a great opportunity for parallelization and potentially a large performance gain by using a computer architecture which is specialised on parallel problems, such as a Graphics Processing Unit. The GPU architecture consists of thousands of computing units, capable of executing thousands of threads concurrently. Using the CUDA language an implementation can be written which is capable of utilising the hardware of an Nvidia GPU (NVIDIA 2018[a]).

A single GPU is not enough to simulate most real-world problems, unfortunately. Therefore we seek a way to allow the simulation to run on systems with multiple GPUs. The tool of choice for this endeavour is the Message Passing Interface (*MPI: A Message-Passing Interface Standard* 2019). MPI provides a programming interface which can be used to scale a computation to run on multiple computers. In a reasonable implementation, the computation

scales to any number of computers without any overhead.

A lot of work has been done in this area and it has spawned a notable implementation called Specfem (D. Komatitsch et al. 2012). The Specfem package uses the theory behind spectral elements to implement a very efficient solver which also can run on GPU clusters (Dimitri Komatitsch, Erlebacher, et al. 2010). Our goal is not to compete with Specfem on performance, but rather to provide an alternative implementation in a more modern programming paradigm. The intent is that our implementation will will be easy to use for new developers and allow for extension with new methods and techniques as research in this field continues. It also provides the opportunity for validation of results by comparing the results from different implementations.

This report describes just a part of a larger package for seismic wave modelling. The intent is to implement a *first* version of a solver which runs on a GPU cluster, with a focus on how it can be implemented correctly, but without trying to maximise the performance. Therefore, we will focus on developing the mathematics of the spectral element method applied to the elastic wave equation. Then we look at how to solve the resulting equations on the GPU hardware in a reasonable but not necessarily optimal way. We will focus on ensuring the correctness of the implementation by comparing the results with analytical solutions as well as with Specfem.

2 Theory

In this chapter we present the theory needed to solve the elastic wave equation numerically using the Spectral Element Method (SEM). The theory comprises the strong and weak formulations of the equation, using the Galerkin projection, numerical quadrature, constructing a linear system and time marching using the Newmark Method.

2.1 The 3D Elastic Wave Equation

The 3D isotropic elastic wave equation is given by the equations for particle motion

$$\rho(\mathbf{x})\ddot{u}_x(\mathbf{x},t) = \partial_x\sigma_{xx}(\mathbf{x},t) + \partial_y\sigma_{xy}(\mathbf{x},t) + \partial_z\sigma_{xz}(\mathbf{x},t) + f_x(\mathbf{x},t) \tag{2.1}$$

$$\rho(\mathbf{x})\ddot{u}_y(\mathbf{x},t) = \partial_x\sigma_{xy}(\mathbf{x},t) + \partial_y\sigma_{yy}(\mathbf{x},t) + \partial_z\sigma_{yz}(\mathbf{x},t) + f_y(\mathbf{x},t) \tag{2.2}$$

$$\rho(\mathbf{x})\ddot{u}_z(\mathbf{x},t) = \partial_x\sigma_{xz}(\mathbf{x},t) + \partial_y\sigma_{yz}(\mathbf{x},t) + \partial_z\sigma_{zz}(\mathbf{x},t) + f_z(\mathbf{x},t) \tag{2.3}$$

and stress

$$\sigma_{xx}(\mathbf{x},t) = [\lambda(\mathbf{x}) + 2\mu(\mathbf{x})]\partial_x u_x(\mathbf{x},t) + \lambda\partial_y u_y(\mathbf{x},t) + \lambda\partial_z u_z(\mathbf{x},t) \tag{2.4}$$
$$\sigma_{yy}(\mathbf{x},t) = [\lambda(\mathbf{x}) + 2\mu(\mathbf{x})]\partial_y u_y(\mathbf{x},t) + \lambda\partial_x u_x(\mathbf{x},t) + \lambda\partial_z u_z(\mathbf{x},t) \tag{2.5}$$
$$\sigma_{zz}(\mathbf{x},t) = [\lambda(\mathbf{x}) + 2\mu(\mathbf{x})]\partial_z u_z(\mathbf{x},t) + \lambda\partial_x u_x(\mathbf{x},t) + \lambda\partial_y u_y(\mathbf{x},t) \tag{2.6}$$
$$\sigma_{xy}(\mathbf{x},t) = \mu(\mathbf{x})[\partial_x u_y(\mathbf{x},t) + \partial_y u_x(\mathbf{x},t)] \tag{2.7}$$
$$\sigma_{xz}(\mathbf{x},t) = \mu(\mathbf{x})[\partial_x u_z(\mathbf{x},t) + \partial_z u_x(\mathbf{x},t)] \tag{2.8}$$
$$\sigma_{yz}(\mathbf{x},t) = \mu(\mathbf{x})[\partial_y u_z(\mathbf{x},t) + \partial_z u_y(\mathbf{x},t)] \tag{2.9}$$

where $\mathbf{x} = (x, y, z)$ is any point in the domain Ω shown in Figure 1, ρ is the density of the medium, λ and μ are the Lamé parameters, u_p is the particle displacement and $\ddot{u}_p$ is the particle acceleration, σ_{pq} the stress and f_p the applied force in the direction p, q where p, q is either x, y or z direction (Ikelle and Amundsen 2005, ch. 2).

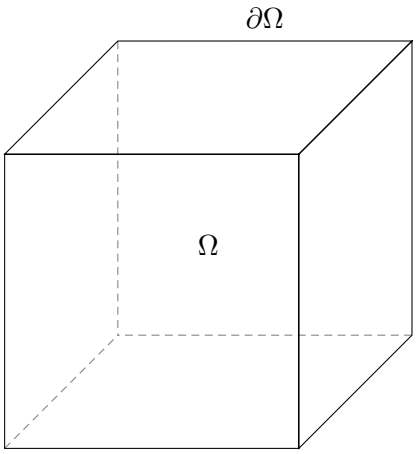


Figure 1: An example domain Ω on which the 3D elastic wave equation could be solved. The boundary $\partial\Omega$ is the entire surface enclosing the cube. This is just an example and in in theory the domain could have an arbitrary shape as long as it can be properly approximated by a mesh.

We also have the stress-free boundary conditions

$$\sigma(\mathbf{x},t) \cdot \mathbf{n} = 0, \mathbf{x} \in \partial\Omega \tag{2.10}$$

where $\mathbf{n}$ is the outward unit normal to the domain boundary $\partial\Omega$ and the initial condition

$$\mathbf{u}|_{t=0} = 0, \dot{\mathbf{u}}|_{t=0} = 0 \tag{2.11}$$

Equations (2.1) - (2.9) and the boundary conditions (2.10) and initial conditions (2.11) together make up the strong formulation of the 3D elastic wave equation. From this we will derive the weak formulation which has some advantageous properties (Quarteroni 2017, chs. 3,4), mainly that it permits the solution of partial differential equations by the methods of linear algebra.

2.2 Weak formulation of the 3D elastic wave equation

Rather than using the strong form of the equations (2.1) - (2.9) we can use the so-called weak formulation of the equations. By using the weak formulation we can relax the restrictions on the second derivative, allowing for solutions where the second derivative is not continuous. The weak formulation also incorporates the boundary conditions so that they are implicitly satisfied.

We find the weak formulation by multiplying both sides of each equation by a test function $\varphi(\mathbf{x})$ and integrating over the domain Ω. The test function φ is an arbitrary time-independent function $\varphi : \Omega \to \mathbb{R}$. For Equation (2.1) we get this weak formulation

$$\int_\Omega \rho \ddot{u}_x \varphi d\Omega = \int_\Omega \partial_x \sigma_{xx} \varphi + \partial_y \sigma_{xy} \varphi + \partial_z \sigma_{xz} \varphi + f_x \varphi d\Omega.$$

In order to be terse we drop all function arguments where they are obvious for the rest of this section.

By performing integration by parts we can move the spatial derivatives from the stress functions $\sigma_{xx}, \sigma_{xy}, \sigma_{xz}$ to the test functions φ:

$$\int_\Omega \partial_x \sigma_{xx} \varphi d\Omega = \int_{\partial\Omega} \sigma_{xx} \varphi d\partial\Omega - \int_\Omega \sigma_{xx} \partial_x \varphi d\Omega.$$

Applying the stress-free boundary condition from Equation (2.10) the integral over the boundary $\partial\Omega$ disappears so we are left with

$$\int_\Omega \partial_x \sigma_{xx} d\Omega = -\int_\Omega \sigma_{xx} \partial_x \varphi d\Omega.$$

Applying this to the other two terms we get

$$\int_\Omega \rho \ddot{u}_x \varphi d\Omega = -\int_\Omega \sigma_{xx} \partial_x \varphi + \sigma_{xy} \partial_y \varphi + \sigma_{xz} \partial_z \varphi d\Omega + \int_\Omega f_x \varphi d\Omega$$

and rewriting we arrive at

$$\int_\Omega \rho \ddot{u}_x \varphi d\Omega + \int_\Omega \sigma_{xx} \partial_x \varphi + \sigma_{xy} \partial_y \varphi + \sigma_{xz} \partial_z \varphi d\Omega = \int_\Omega f_x \varphi d\Omega.$$

Doing the same procedure for Equations (2.2)-(2.3) gives the following weak formulations for the equations of motion

$$\int_\Omega \rho \ddot{u}_x \varphi d\Omega + \int_\Omega \sigma_{xx}\partial_x\varphi + \sigma_{xy}\partial_y\varphi + \sigma_{xz}\partial_z\varphi d\Omega = \int_\Omega f_x\varphi d\Omega \tag{2.12}$$

$$\int_\Omega \rho \ddot{u}_y \varphi d\Omega + \int_\Omega \sigma_{xy}\partial_x\varphi + \sigma_{yy}\partial_y\varphi + \sigma_{yz}\partial_z\varphi d\Omega = \int_\Omega f_y\varphi d\Omega \tag{2.13}$$

$$\int_\Omega \rho \ddot{u}_z \varphi d\Omega + \int_\Omega \sigma_{xz}\partial_x\varphi + \sigma_{yz}\partial_y\varphi + \sigma_{zz}\partial_z\varphi d\Omega = \int_\Omega f_z\varphi d\Omega. \tag{2.14}$$

For the stress Equations (2.4)-(2.9) we get

$$\int_\Omega \sigma_{xx}\varphi d\Omega = \int_\Omega (\lambda + 2\mu)\partial_x u_x\varphi + \lambda\partial_y u_y\varphi + \lambda\partial_z u_z\varphi d\Omega \tag{2.15}$$

$$\int_\Omega \sigma_{yy}\varphi d\Omega = \int_\Omega (\lambda + 2\mu)\partial_y u_y\varphi + \lambda\partial_x u_x\varphi + \lambda\partial_z u_z\varphi d\Omega \tag{2.16}$$

$$\int_\Omega \sigma_{zz}\varphi d\Omega = \int_\Omega (\lambda + 2\mu)\partial_z u_z\varphi + \lambda\partial_y u_y\varphi + \lambda\partial_x u_x\varphi d\Omega \tag{2.17}$$

$$\int_\Omega \sigma_{xy}\varphi d\Omega = \int_\Omega \mu(\partial_x u_y + \partial_y u_x)\varphi d\Omega \tag{2.18}$$

$$\int_\Omega \sigma_{xz}\varphi d\Omega = \int_\Omega \mu(\partial_x u_z + \partial_z u_x)\varphi d\Omega \tag{2.19}$$

$$\int_\Omega \sigma_{yz}\varphi d\Omega = \int_\Omega \mu(\partial_y u_z + \partial_z u_y)\varphi d\Omega. \tag{2.20}$$

We must also transform the initial conditions from Equation (2.11)

$$\int_\Omega \varphi \mathbf{u}|_{t=0} d\Omega = 0, \int_\Omega \varphi \dot{\mathbf{u}}|_{t=0} d\Omega = 0.$$

By using the weak formulation instead of the strong one we are able to solve the wave equation using linear algebra. As we will see in the following sections we can apply several techniques which allow us to find an approximate solution which is stable, consistent and approaches the continuous solution.

2.3 Galerkin projection

Before we can solve these equations we must approximate the solutions in a finite-dimensional vector space, thus making it possible to compute the solution numerically (Quarteroni 2017, ch. 4). This is called the Galerkin method.

Note that from now on and for the rest of the entire theory chapter we will only use the first equation for particle motion, Equation (2.12), and the first of the stress Equations (2.15). This is because the other equations are so similar that performing the exact same steps for them as well would be redundant. We only intend to demonstrate how the equations can be solved, not give full expressions for each part of the system. The remaining equations can easily be arrived at by following the methods presented in this chapter.

We define a set of N_b basis functions $\varphi_i(\mathbf{x})$ with which we can approximate, i.e. project, a function from an infinite-dimensional vector space V^∞ to a finite-dimensional vector space V^n. We can approximate a function $f(\mathbf{x}, t) \in V^\infty$ by decomposing it in the following way

$$f(\mathbf{x}, t) \approx \bar{f}(\mathbf{x}, t) = \sum_{i=0}^{N_b-1} f_i(t)\varphi_i(\mathbf{x}).$$

Here $\bar{f}(\mathbf{x}, t) \in V^n$ is the approximation to $f(\mathbf{x}, t)$ defined by the decomposition using time-dependent coefficients $f_i(t)$ and space-dependent basis-functions $\varphi_i(\mathbf{x})$. The functions φ_i form a basis of the approximation space, so the accuracy of the approximation depends on the choice of basis functions.

We can apply this decomposition to the acceleration

$$\ddot{u}_x(\mathbf{x}, t) \approx \ddot{\bar{u}}_x(\mathbf{x}, t) = \sum_{i=0}^{N_b-1} \ddot{u}_x^i(t)\varphi_i(\mathbf{x}) \tag{2.21}$$

and to the stress

$$\sigma_{xx} \approx \bar{\sigma}_{xx}(\mathbf{x}, t) = \sum_{i=0}^{N_b-1} \sigma_{xx}^i(t)\varphi_i(\mathbf{x}). \tag{2.22}$$

Using this approximation we move from our exact weak Equation (2.12) to an approximation by choosing the test functions φ to be the basis functions φ_q of the approximation space. Note that we use the index i in the Galerkin approximation, but index q for the test functions to indicate that these functions should be counted separately. We then get

$$\int_\Omega \rho\ddot{\bar{u}}_x\varphi_q d\Omega + \int_\Omega \bar{\sigma}_{xx}\partial_x\varphi_q + \bar{\sigma}_{xy}\partial_y\varphi_q + \bar{\sigma}_{xz}\partial_z\varphi_q d\Omega = \int_\Omega f_x\varphi_q d\Omega.$$

Expanding the approximations gives

$$\sum_{i=0}^{N_b-1}\left[\int_\Omega \rho\ddot{u}_x^i\varphi_i\varphi_q d\Omega + \int_\Omega \sigma_{xx}^i\varphi_i\partial_x\varphi_q + \sigma_{xy}^i\varphi_i\partial_y\varphi_q + \sigma_{xz}^i\varphi_i\partial_z\varphi_q d\Omega\right] = \int_\Omega f_x\varphi_q d\Omega. \tag{2.23}$$

Even though the weak formulation (2.12) has the requirement that u_x is a solution for *any* function φ it is enough for us to require that it is a solution for φ_q since they form a basis for the approximation space (Quarteroni 2017, p. 61).

For the stress Equation (2.15) we get

$$\sum_i^{N_b-1} \int_\Omega \sigma_{xx}^i\varphi_q d\Omega = \sum_i^{N_b-1} \int_\Omega (\lambda + 2\mu)u_x^i\partial_x\varphi_i\varphi_q + \lambda u_y^i\partial_y\varphi_i\varphi_q + \lambda u_z^i\partial_z\varphi_i\varphi_q d\Omega. \tag{2.24}$$

2.4 Discretizing the domain

Computation of a numerical solution requires decomposition of the domain into simpler geometrical entities called elements. In the SEM in three dimensions these elements are hexahedra. We divide a domain Ω into n_e non-overlapping elements Ω_e such that

$$\bigcup_{e=0}^{n_e-1} \Omega_e = \Omega$$

as shown in Figure 2. Now we can separate an integral of a function $f(\mathbf{x})$ over a domain Ω into

$$\int_\Omega f(\mathbf{x}) d\Omega = \sum_{e}^{n_e=0} \int_{\Omega_e} f(\mathbf{x}) d\Omega_e$$

so we can rewrite Equation (2.23) as

$$\sum_{e=0}^{n_e-1} \sum_{i=0}^{N_b-1} \left[\int_{\Omega_e} \rho \ddot{u}_x^i \varphi_i \varphi_q d\Omega_e + \int_{\Omega_e} \sigma_{xx}^i \varphi_i \partial_x \varphi_q + \sigma_{xy}^i \varphi_i \partial_y \varphi_q + \sigma_{xz}^i \varphi_i \partial_z \varphi_q d\Omega_e \right] = \sum_{e=0}^{n_e-1} \int_{\Omega_e} f_x \varphi_q d\Omega_e.$$

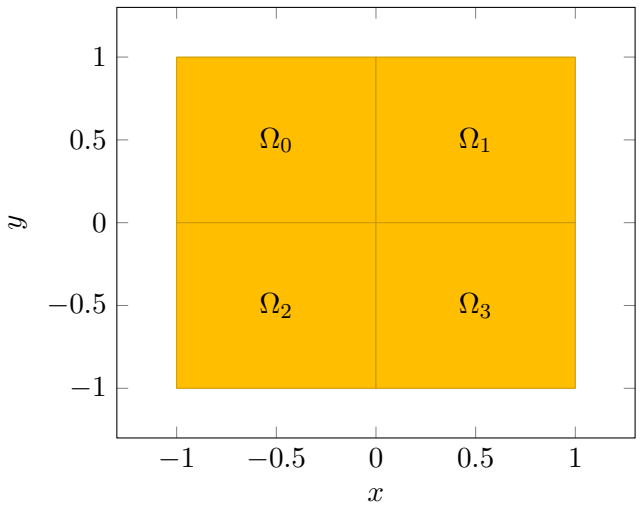


Figure 2: To make computations feasible, the domain Ω is divided into non-overlapping elements $\Omega_0 \dots \Omega_3$. In 3D we would use hexahedra instead of quadrilaterals as we see here.

The issue with this formulation is that the coefficients $\ddot{u}_x^i$ depend on all the integrals over every element. To remedy this we choose local basis functions φ_{ei} which are zero everywhere except on element Ω_e (they maybe zero inside the element also, but not everywhere). Note that ei are two indices, first the number of the element $e \in [0, 1, \dots, n_e - 1]$ and then the number of the basis function on that element $i \in [0, 1, \dots, N_b - 1]$.

This means that we can write the approximation of the acceleration $\ddot{\bar{u}}_x$ from Equation (2.21) on a per-element basis as

$$\ddot{u}_x(\mathbf{x},t)|_{\mathbf{x}\in\Omega_e} \approx \ddot{\bar{u}}_x(\mathbf{x},t)|_{\mathbf{x}\in\Omega_e} = \sum_{i=0}^{n_b-1} \ddot{u}_x^{ei}(t)\varphi_{ei}(\mathbf{x})$$

Now we can rewrite Equation (2.23) as a set of equations, with one for each element

$$\begin{aligned}
&\sum_{i=0}^{N_b-1} \int_{\Omega_e} \rho \ddot{u}_x^{ei}\varphi_{ei}\varphi_{eq} d\Omega_e \\
&+ \int_{\Omega_e} \sigma_{xx}^{ei}\varphi_{ei}\partial_x\varphi_{eq} + \sigma_{xy}^{ei}\varphi_{ei}\partial_y\varphi_{eq} + \sigma_{xz}^{ei}\varphi_{ei}\partial_z\varphi_{eq} d\Omega_e \\
&= \int_{\Omega_e} f_x\varphi_{eq} d\Omega_e
\end{aligned} \tag{2.25}$$

which means that we can solve the equations for each element independently of each other. Note that we also change the naming from i to ei so it includes the element e since we now have a set of different basis functions for each element, which means a total of $n_e N_b$ basis functions.

Now we can perform the same treatment on the stress Equation (2.24) and get

$$\sum_{i}^{N_b-1} \int_{\Omega_e} \sigma_{xx}^{ei}\varphi_{eq} d\Omega = \sum_{i}^{N_b-1} \int_{\Omega_e} (\lambda+2\mu)u_x^{ei}\partial_x\varphi_{ei}\varphi_{eq} + \lambda u_y^{ei}\partial_y\varphi_{ei}\varphi_{eq} + \lambda u_z^{ei}\partial_z\varphi_{ei}\varphi_{eq} d\Omega. \tag{2.26}$$

These expressions quickly become impractical to compute since we would have to compute and store every single basis function at multiple points in order to compute the integrals. This situation can be improved by introducing a *reference element* which we can map all elements to and perform computations.

2.5 Mapping to the reference element

We define a reference element Λ and an invertible mapping $\mathbf{F}_e(\xi)$ between coordinates of any mesh element Ω_e and the reference element. We do this so that we can treat all elements the same, only changing the mapping $\mathbf{F}_e$, which will simplify the computations of the integrals. Let any point in the reference element be given by $\xi = \{\xi_1, \xi_2, \xi_3\}$ and a point in element Ω_e be given by $\mathbf{x} = \{x, y, z\}$. We then define a mapping from a coordinate in the reference element to a coordinate in another element

$$\mathbf{x}_e(\xi) = \mathbf{F_e}(\xi). \tag{2.27}$$

We express this mapping as a sum of n_a shape functions $N^a(\xi)$ multiplied by the n_a corners (anchor nodes, 8 in total for hexahedra) $\mathbf{x}_e^a$ of the element e

$$\mathbf{F}_e(\xi) = \sum_{a=0}^{n_a-1} N^a(\xi)\mathbf{x}_e^a.$$

We choose our reference element Λ to be the unit cube ($[-1,1]^3$) with corners

$$\xi^0 = (-1,-1,-1), \xi^2 = (1,-1,-1), \ldots, \xi^7 = (1,1,1)$$

and the shape functions as products of three Lagrange polynomials of order 1

$$N^a(\xi) = l_{1,a}(\xi_1)l_{2,a}(\xi_2)l_{3,a}(\xi_3).$$

with the property that a shape function N^a is 0 on all corners ξ^b except ξ^a

$$N^a(\xi^b) = \delta_{ab}$$

where δ_{ab} is the Kronecker delta. This gives the following 8 shape functions

$$N^0(\xi) = \frac{1}{2}(1-\xi_1)\frac{1}{2}(1-\xi_2)\frac{1}{2}(1-\xi_3) \tag{2.28}$$
$$N^1(\xi) = \frac{1}{2}(1+\xi_1)\frac{1}{2}(1-\xi_2)\frac{1}{2}(1-\xi_3) \tag{2.29}$$
$$N^2(\xi) = \frac{1}{2}(1-\xi_1)\frac{1}{2}(1+\xi_2)\frac{1}{2}(1-\xi_3) \tag{2.30}$$
$$N^3(\xi) = \frac{1}{2}(1+\xi_1)\frac{1}{2}(1+\xi_2)\frac{1}{2}(1-\xi_3) \tag{2.31}$$
$$N^4(\xi) = \frac{1}{2}(1-\xi_1)\frac{1}{2}(1-\xi_2)\frac{1}{2}(1+\xi_3) \tag{2.32}$$
$$N^5(\xi) = \frac{1}{2}(1+\xi_1)\frac{1}{2}(1-\xi_2)\frac{1}{2}(1+\xi_3) \tag{2.33}$$
$$N^6(\xi) = \frac{1}{2}(1-\xi_1)\frac{1}{2}(1+\xi_2)\frac{1}{2}(1+\xi_3) \tag{2.34}$$
$$N^7(\xi) = \frac{1}{2}(1+\xi_1)\frac{1}{2}(1+\xi_2)\frac{1}{2}(1+\xi_3). \tag{2.35}$$

A property of the mapping $\mathbf{F}_e(\xi)$ is that corners ξ^a in the reference element map to the corresponding corner $\mathbf{x}^a$ in another element

$$\mathbf{F}_e(\xi^a) = \mathbf{x}_e^a.$$

Using this mapping we can rewrite Equation (2.25) on the domain Λ, greatly simplifying computation. The expression becomes rather long so we perform the procedure for each term instead the whole equation at once. The first term of Equation (2.25) gives

$$\sum_{i=0}^{N_b-1} \int_{\Omega_e} \rho \ddot{u}_x^{ei} \varphi_{ei} \varphi_{eq} d\Omega_e = \sum_{i=0}^{N_b-1} \int_{\Lambda} \rho(\mathbf{x}_e(\xi)) \ddot{u}_x^{ei} \varphi_{ei}(\mathbf{x}_e(\xi)) \varphi_{eq}(\mathbf{x}_e(\xi)) J^e(\xi) d\Lambda$$

where J^e is the Jacobian determinant of the transformation $\mathbf{F}_e$ which is given by

$$J^e(\xi) = \begin{vmatrix} \frac{\partial x_e}{\partial \xi_1} & \frac{\partial x_e}{\partial \xi_2} & \frac{\partial x_e}{\partial \xi_3} \\ \frac{\partial y_e}{\partial \xi_1} & \frac{\partial y_e}{\partial \xi_2} & \frac{\partial y_e}{\partial \xi_3} \\ \frac{\partial z_e}{\partial \xi_1} & \frac{\partial z_e}{\partial \xi_2} & \frac{\partial z_e}{\partial \xi_3} \end{vmatrix}.$$

To simplify a bit we rewrite the mapped variables as

$$\rho(\mathbf{x}_e(\xi)) = \hat{\rho}^e(\xi)$$
$$\varphi(\mathbf{x}_e(\xi)) = \hat{\varphi}(\xi).$$

We can also note that all of the elemental basis functions φ_{ei} are the same after mapping to the reference element so we can actually drop the e index.

We then get

$$\int_{\Omega_e} \rho \ddot{u}_x^{ei} \varphi_{ei} \varphi_{eq} d\Omega_e = \int_{\Lambda} \hat{\rho}^e(\xi) \ddot{u}_x^{ei} \hat{\varphi}_i(\xi) \hat{\varphi}_q(\xi) J^e(\xi) d\Lambda$$

for the first term. The second term of Equation (2.25) includes a spatial derivative which must be dealt with. This is done by using the chain rule for multiple variables

$$\partial_x \varphi = \frac{\partial \varphi}{\partial \xi_1}\frac{\partial \xi_1}{\partial x} + \frac{\partial \varphi}{\partial \xi_2}\frac{\partial \xi_2}{\partial x} + \frac{\partial \varphi}{\partial \xi_3}\frac{\partial \xi_3}{\partial x} = \frac{\partial \xi_1}{\partial x}\partial_{\xi_1}\varphi + \frac{\partial \xi_2}{\partial x}\partial_{\xi_2}\varphi + \frac{\partial \xi_3}{\partial x}\partial_{\xi_3}\varphi.$$

So for the second term we get

$$\int_{\Omega_e} \sigma_{xx}^{ei} \varphi_{ei} \partial_x \varphi_{eq} d\Lambda = \int_{\Lambda} \sigma_{xx}^{ei} \hat{\varphi}_i(\xi) (\frac{\partial \xi_1^e}{\partial x}(\xi)\partial_{\xi_1} + \frac{\partial \xi_2^e}{\partial x}(\xi)\partial_{\xi_2} + \frac{\partial \xi_3^e}{\partial x}(\xi)\partial_{\xi_3}) \hat{\varphi}_q(\xi) J^e(\xi) d\Lambda$$

while the third and fourth terms give similar expressions. The right hand side of Equation (2.25) gives the following

$$\int_{\Omega_e} f_x(\mathbf{x}, t) \varphi_q(\mathbf{x}) d\Omega = \int_{\Lambda} f_x(\mathbf{x}_e(\xi), t) \hat{\varphi}_q(\xi) J^e(\xi) d\Lambda$$

After mapping to the reference element the entire Equation (2.25) thus becomes

$$\begin{aligned}
&\sum_{i=0}^{n_b-1} \int_\Lambda \hat{\rho}^e(\xi)\ddot{u}_x^{ei}\hat{\varphi}_i(\xi)\hat{\varphi}_q(\xi)J^e(\xi)d\Lambda \\
&\quad + \int_\Lambda \sigma_{xx}^{ei}\hat{\varphi}_i(\xi)\left(\frac{\partial\xi_1^e}{\partial x}(\xi)\partial_{\xi_1} + \frac{\partial\xi_2^e}{\partial x}(\xi)\partial_{\xi_2} + \frac{\partial\xi_3^e}{\partial x}(\xi)\partial_{\xi_3}\right)\hat{\varphi}_q(\xi)J^e(\xi)d\Lambda \\
&\quad + \int_\Lambda \sigma_{xy}^{ei}\hat{\varphi}_i(\xi)\left(\frac{\partial\xi_1^e}{\partial y}(\xi)\partial_{\xi_1} + \frac{\partial\xi_2^e}{\partial y}(\xi)\partial_{\xi_2} + \frac{\partial\xi_3^e}{\partial y}(\xi)\partial_{\xi_3}\right)\hat{\varphi}_q(\xi)J^e(\xi)d\Lambda \\
&\quad + \int_\Lambda \sigma_{xz}^{ei}\hat{\varphi}_i(\xi)\left(\frac{\partial\xi_1^e}{\partial z}(\xi)\partial_{\xi_1} + \frac{\partial\xi_2^e}{\partial z}(\xi)\partial_{\xi_2} + \frac{\partial\xi_3^e}{\partial z}(\xi)\partial_{\xi_3}\right)\hat{\varphi}_q(\xi)J^e(\xi)d\Lambda \\
&\quad = \int_\Lambda \hat{\varphi}_q(\xi)f_x(\mathbf{x}_e(\xi),t)J^e(\xi)d\Lambda.
\end{aligned} \tag{2.36}$$

We perform the same mapping to the stress in Equation (2.26) and get

$$\begin{aligned}
&\sum_{i}^{N_b-1} \int_\Lambda \sigma_{xx}^{ei}\hat{\varphi}_q(\xi)J^e(\xi)d\Lambda \\
&= \sum_{i}^{N_b-1} \int_\Lambda [\hat{\lambda}^e(\xi) + 2\hat{\mu}^e(\xi)]u_x^{ei}\left(\frac{\partial\xi_1^e}{\partial x}(\xi)\partial_{\xi_1} + \frac{\partial\xi_2^e}{\partial x}(\xi)\partial_{\xi_2} + \frac{\partial\xi_3^e}{\partial x}(\xi)\partial_{\xi_3}\right)\hat{\varphi}_i(\xi)\hat{\varphi}_q(\xi)J^e(\xi)d\Lambda \\
&\quad + \int_\Lambda \hat{\lambda}^e(\xi)u_y^{ei}\left(\frac{\partial\xi_1^e}{\partial y}(\xi)\partial_{\xi_1} + \frac{\partial\xi_2^e}{\partial y}(\xi)\partial_{\xi_2} + \frac{\partial\xi_3^e}{\partial y}(\xi)\partial_{\xi_3}\right)\hat{\varphi}_i(\xi)\hat{\varphi}_q(\xi)J^e(\xi)d\Lambda \\
&\quad + \int_\Lambda \hat{\lambda}^e(\xi)u_z^{ei}\left(\frac{\partial\xi_1^e}{\partial z}(\xi)\partial_{\xi_1} + \frac{\partial\xi_2^e}{\partial z}(\xi)\partial_{\xi_2} + \frac{\partial\xi_3^e}{\partial z}(\xi)\partial_{\xi_3}\right)\hat{\varphi}_i(\xi)\hat{\varphi}_q(\xi)J^e(\xi)d\Lambda.
\end{aligned} \tag{2.37}$$

What is needed now is a way to numerically evaluate the integrals on the reference element.

2.6 Evaluating the integrals

Having mapped the equations to the reference element Λ we now need a way to compute the integrals. To do this we use Gauss-Lobatto-Legendre (GLL) quadrature (Fichtner 2011, appendix A.3). This choice has several desirable properties, one of which is that the collocation points include the ends of the integration interval, which is not the case in Gaussian quadrature, and that this method leads to a diagonal system of equations, which is trivial to solve.

Like Gaussian quadrature, GLL quadrature is performed by sampling the integrand at the N GLL points ξ^l and multiplying by weights w_l

$$\int_a^b f(\xi)d\xi \approx \sum_{l=0}^{N-1} w_l f(\xi^l).$$

The GLL points and weights are given in Fichtner 2011, appendix A.3.2 for some polynomial degrees on the interval $[-1, 1]$.

Since we are dealing with a three dimensional domain, namely the reference cube $\Lambda = [-1, 1]^3$, we apply the quadrature three times, once for each spatial direction. We then get

$$\int_\Lambda f(\xi) d\Lambda \approx \sum_{l,m,n=0}^{N-1} w_l w_m w_n f(\xi^{lmn})$$

where ξ^{lmn} is the GLL-point $(\xi_1^l, \xi_2^m, \xi_3^n)$.

Now we have enough information to choose our basis functions φ. We will use a product of three Lagrange polynomials $l_i^{(n_b-1)}$ of order $n_b - 1$ collocated at the n_b GLL points on the interval $[-1, 1]$, one polynomial for each spatial direction. Fichtner 2011, appendix A.3.2 gives several reasons to motivate the choice of Lagrange polynomials collocated at the GLL points as basis functions. It is important to keep in mind that when using n_b points for GLL quadrature it is exact for polynomials of degree $2(n_b - 1) - 1 = 2n_b - 3$ or lower (Fichtner 2011, appendix A.3.2). However the polynomials in our expressions are of order $2n_b - 2$ so the quadrature is not exact.

The basis functions thus become, dropping the $(n_b - 1)$ superscript for simpler notation,

$$\hat{\varphi}_{ijk} = l_i(\xi_1) l_j(\xi_2) l_k(\xi_3)$$

and to reflect the fact that the basis function is a product of three polynomials we change the indexing from $i \in \{0, 1, \ldots, N_b - 1\}$ to the triplet $i, j, k \in \{0, 1, \ldots, n_b - 1\}$ so that the total number of basis functions in the reference element is $N_b = n_b^3$. We will also start writing sums as

$$\sum_{i=0}^{N_b-1} \hat{\varphi}_i = \sum_{ijk} \hat{\varphi}_{ijk}$$

using the sum shorthand

$$\sum_{ijk} \hat{\varphi}_{ijk} = \sum_{i=0}^{n_b-1} \sum_{j=0}^{n_b-1} \sum_{k=0}^{n_b-1} \hat{\varphi}_{ijk}$$

since all the sums from now on will have this shape.

Let us consider the first term of Equation (2.36) and apply GLL quadrature to compute the integral

$$
\begin{aligned}
&\sum_{i=0}^{N_b-1} \int_\Lambda \hat{\rho}^e(\xi)\ddot{u}_x^{ei}(t)\hat{\varphi}_i(\xi)\hat{\varphi}_q(\xi)J^e(\xi)d\Lambda \\
\approx &\sum_{ijk}\sum_{lmn} w_l w_m w_n \hat{\rho}^e(\xi^{lmn})u_x^{eijk}(t)\hat{\varphi}_{ijk}(\xi^{lmn})\hat{\varphi}_{qrs}(\xi^{lmn})J^e(\xi^{lmn}) \\
= &\sum_{ijk}\sum_{lmn} w_l w_m w_n \hat{\rho}^e(\xi^{lmn})u_x^{eijk}(t)l_i(\xi_1^l)l_j(\xi_2^m)l_k(\xi_3^n)l_q(\xi_1^l)l_r(\xi_2^m)l_s(\xi_3^n)J^e(\xi^{lmn}).
\end{aligned}
$$

Since the Lagrange polynomials satisfy the cardinal interpolation property (Fichtner 2011, appendix A.2.2)

$$
l_i(\xi^j) = \delta_{ij}
$$

we can simplify the expression a lot. The expression consists of a product of several Lagrange polynomials and we can see that it is zero whenever $i \neq l \neq q$, $j \neq m \neq r$ and $k \neq n \neq s$. So we must have $i = l = q$ which gives

$$
\begin{aligned}
&\sum_{ijk}\sum_{lmn} w_l w_m w_n \hat{\rho}^e(\xi^{lmn})u_x^{eijk}(t)l_i(\xi_1^l)l_j(\xi_2^m)l_k(\xi_3^n)l_q(\xi_1^l)l_r(\xi_2^m)l_s(\xi_3^n)J^e(\xi^{lmn}) \\
= &\sum_{jk}\sum_{mn} w_q w_m w_n \hat{\rho}^e(\xi^{qmn})u_x^{eqjk}(t)l_j(\xi_2^m)l_k(\xi_3^n)l_r(\xi_2^m)l_s(\xi_3^n)J^e(\xi^{qmn})
\end{aligned}
$$

and $j = m = r$

$$
= \sum_{k}\sum_{n} w_q w_r w_n \hat{\rho}^e(\xi^{qrn})u_x^{eqrk}(t)l_k(\xi_3^n)l_s(\xi_3^n)J^e(\xi^{qrn})
$$

as well as $k = n = s$ giving the greatly simplified expression

$$
= w_q w_r w_s \hat{\rho}^e(\xi^{qrs})u_x^{eqrs}(t)J^e(\xi^{qrs}).
$$

This is the expression for the elemental mass matrix $\mathbf{M}^e$

$$
\left[M_{qrs}^e\right] = \left[w_q w_r w_s \hat{\rho}^e(\xi^{qrs})J^e(\xi^{qrs})\right]. \tag{2.38}
$$

Now we have arrived at an expression for the first term of (2.12) which can be computed numerically. We then perform the same procedure on the second, third and fourth terms which all follow the same pattern. The only difference is the spatial derivative of the stress functions, but we saw how to deal with this in the previous Section 2.5 arriving at Equation (2.36). Starting with Equation (2.36) we apply GLL quadrature to the second term of this equation

$$
\begin{aligned}
&\sum_{ijk}\int_{\Lambda}\sigma_{xx}^{eijk}(t)\hat{\varphi}_{ijk}(\xi)(\frac{\partial\xi_1^e}{\partial x}(\xi)\partial_{\xi_1}+\frac{\partial\xi_2^e}{\partial x}(\xi)\partial_{\xi_2}+\frac{\partial\xi_3^e}{\partial x}(\xi)\partial_{\xi_3})\hat{\varphi}_{qrs}(\xi)J^e(\xi)d\Lambda\\
&\approx\sum_{ijk}\sum_{lmn}w_lw_mw_n\sigma_{xx}^{eijk}(t)l_i(\xi_1^l)l_j(\xi_2^m)l_k(\xi_3^n)l'_q(\xi_1^l)l_r(\xi_2^m)l_s(\xi_3^n)\frac{\partial\xi_1^e}{\partial x}(\xi^{lmn})J^e(\xi^{lmn})\\
&+\sum_{ijk}\sum_{lmn}w_lw_mw_n\sigma_{xx}^{eijk}(t)l_i(\xi_1^l)l_j(\xi_2^m)l_k(\xi_3^n)l_q(\xi_1^l)l'_r(\xi_2^m)l_s(\xi_3^n)\frac{\partial\xi_2^e}{\partial x}(\xi^{lmn})J^e(\xi^{lmn})\\
&+\sum_{ijk}\sum_{lmn}w_lw_mw_n\sigma_{xx}^{eijk}(t)l_i(\xi_1^l)l_j(\xi_2^m)l_k(\xi_3^n)l_q(\xi_1^l)l_r(\xi_2^m)l'_s(\xi_3^n)\frac{\partial\xi_3^e}{\partial x}(\xi^{lmn})J^e(\xi^{lmn}),
\end{aligned}
$$

Since the expressions have become very long we split them into one term for each partial derivative, so that we can write them on multiple lines easily. Because of the spatial derivatives this expression does not simplify as much as the first term but a much shorter expression can still be arrived at

$$
\begin{aligned}
&\sum_{ijk}\int_{\Lambda}\sigma_{xx}^{eijk}(t)\hat{\varphi}_{ijk}(\xi)(\frac{\partial\xi_1^e}{\partial x}(\xi)\partial_{\xi_1}+\frac{\partial\xi_2^e}{\partial x}(\xi)\partial_{\xi_2}+\frac{\partial\xi_3^e}{\partial x}(\xi)\partial_{\xi_3})\hat{\varphi}_{qrs}(\xi)J^e(\xi)d\Lambda\\
&\approx\sum_{l}w_lw_rw_s\sigma_{xx}^{elrs}(t)\frac{\partial\xi_1^e}{\partial x}(\xi^{lrs})l'_q(\xi_1^l)J^e(\xi^{lrs})\\
&+\sum_{m}w_qw_mw_s\sigma_{xx}^{eqms}(t)\frac{\partial\xi_2^e}{\partial x}(\xi^{qms})l'_r(\xi_2^m)J^e(\xi^{qms})\\
&+\sum_{n}w_qw_rw_n\sigma_{xx}^{eqrn}(t)\frac{\partial\xi_3^e}{\partial x}(\xi^{qrn})l'_s(\xi_3^n)J^e(\xi^{qrn}).
\end{aligned}
$$

This procedure must also be applied to the third and fourth terms but the steps will not be given here as they are exactly the same. Adding the second, third and fourth terms together gives the elemental vector

$$
\begin{aligned}
[k^e_{qrs}] = &\sum_l w_l w_r w_s \sigma^{elrs}_{xx}(t) l'_q(\xi^l_1) \frac{\partial \xi^e_1}{\partial x}(\xi^{lrs}) J^e(\xi^{lrs}) \\
&+ \sum_m w_q w_m w_s \sigma^{eqms}_{xx}(t) l'_r(\xi^m_2) \frac{\partial \xi^e_2}{\partial x}(\xi^{qms}) J^e(\xi^{qms}) \\
&+ \sum_n w_q w_r w_n \sigma^{eqrn}_{xx}(t) l'_s(\xi^n_3) \frac{\partial \xi^e_3}{\partial x}(\xi^{qrn}) J^e(\xi^{qrn}) \\
&+ \sum_l w_l w_r w_s \sigma^{elrs}_{xy}(t) l'_q(\xi^l_1) \frac{\partial \xi^e_1}{\partial y}(\xi^{lrs}) J^e(\xi^{lrs}) \\
&+ \sum_m w_q w_m w_s \sigma^{eqms}_{xy}(t) l'_r(\xi^m_2) \frac{\partial \xi^e_2}{\partial y}(\xi^{qms}) J^e(\xi^{qms}) \\
&+ \sum_n w_q w_r w_n \sigma^{eqrn}_{xy}(t) l'_s(\xi^n_3) \frac{\partial \xi^e_3}{\partial y}(\xi^{qrn}) J^e(\xi^{qrn}) \\
&+ \sum_l w_l w_r w_s \sigma^{elrs}_{xz}(t) l'_q(\xi^l_1) \frac{\partial \xi^e_1}{\partial z}(\xi^{lrs}) J^e(\xi^{lrs}) \\
&+ \sum_m w_q w_m w_s \sigma^{eqms}_{xz}(t) l'_r(\xi^m_2) \frac{\partial \xi^e_2}{\partial z}(\xi^{qms}) J^e(\xi^{qms}) \\
&+ \sum_n w_q w_r w_n \sigma^{eqrn}_{xz}(t) l'_s(\xi^n_3) \frac{\partial \xi^e_3}{\partial z}(\xi^{qrn}) J^e(\xi^{qrn})
\end{aligned}
\tag{2.39}
$$

To finish with Equation (2.12) we must also integrate the right hand side. Starting with the right hand side of Equation (2.36) we perform the same treatment as for the other terms

$$
\begin{aligned}
&\int_{\Omega_e} \varphi_{qrs}(\mathbf{x}) f_x(\mathbf{x}) d\Omega \\
=&\int_\Lambda \hat{\varphi}_{qrs}(\xi) f_x(\mathbf{x}_e(\xi)) J^e(\xi) d\Lambda \\
=&\int_\Lambda l_q(\xi_1) l_r(\xi_2) l_s(\xi_3) f_x(\mathbf{x}_e(\xi)) J^e(\xi) d\Lambda \\
=&\sum_{lmn} w_l w_m w_n l_q(\xi^l_1) l_r(\xi^m_2) l_s(\xi^n_3) f_x(\mathbf{x}_e(\xi^{lmn})) J^e(\xi^{lmn}) \\
=&w_q w_r w_s f_x(\mathbf{x}_e(\xi^{qrs}), t) J^e(\xi^{qrs})
\end{aligned}
$$

which we can write as a vector

$$
[f^e_{qrs}] = w_q w_r w_s f_x(\mathbf{x}_e(\xi^{qrs}), t) J^e(\xi^{qrs}). \tag{2.40}
$$

It is important to keep in mind that the approximation of the right hand side by GLL quadrature may not be appropriate since it is only valid for polynomials up a certain degree. If the source function f_x is far from a polynomial of degree $2(n_b - 1) - 1$, like the common case of a point source, then other methods might be more appropriate such as exact integration of the point source (Fichtner 2011, ch. 4.2.4).

Now we can write Equation (2.12) as a system of equations by using the expressions for the mass matrix $\mathbf{M}^e$ from Equation (2.38), the vector $\mathbf{k}^e$ from Equation (2.39) and the vector $\mathbf{f}^e$ from Equation (2.40)

$$\mathbf{M}^e \cdot \ddot{\mathbf{u}}_x^e(t) + \mathbf{k}^e(t) = \mathbf{f}^e(t).$$

We could solve this equation for each element, but the solution would not be continuous at the boundaries between elements. In Section 2.7 we will deal with this issue, but first we must compute the stress σ which is needed by Equation (2.39).

Before we can compute the vector $\mathbf{k}$ given by Equation (2.39) we need to know the stresses σ_{xx}, σ_{xy} and σ_{xz}. So now we apply the same ideas to Equation (2.37) in order to arrive at an expression for σ_{xx} which can be computed (expressions for the other stress components can be found in exactly the same way so it will not be shown here, as mentioned previously). For the first term of the stress equation we easily find

$$\int_\Lambda \sigma_{xx}^{eqrs}(t)\hat{\varphi}_{qrs}d\Lambda \approx w_q w_r w_s \sigma_{xx}^{eqrs}(t) J^e(\xi^{qrs}).$$

The terms on the right hand side of Equation (2.37) which include partial derivatives are slightly longer and when applying GLL quadrature and simplifying we get

$$\begin{aligned}
&\sum_i^{N_b-1} \int_\Lambda [\hat{\lambda}^e(\xi) + 2\hat{\mu}^e(\xi)] u_x^{ei}(t) \left(\frac{\partial \xi_1^e}{\partial x}(\xi)\partial_{\xi_1} + \frac{\partial \xi_2^e}{\partial x}(\xi)\partial_{\xi_2} + \frac{\partial \xi_3^e}{\partial x}(\xi)\partial_{\xi_3} \right) \hat{\varphi}_i(\xi)\hat{\varphi}_q(\xi) J^e(\xi) d\Lambda \\
&= w_q w_r w_s J^e(\xi^{qrs})[\hat{\lambda}^e(\xi^{qrs}) + 2\hat{\mu}^e(\xi^{qrs})] \Bigg[\sum_i u_x^{eirs}(t) l_i'(\xi_1^q) \frac{\partial \xi_1^e}{\partial x}(\xi^{irs}) \\
&\qquad + \sum_j u_x^{eqjs}(t) l_j'(\xi_2^r) \frac{\partial \xi_2^e}{\partial x}(\xi^{qjs}) \\
&\qquad + \sum_k u_x^{eqrk}(t) l_k'(\xi_3^s) \frac{\partial \xi_3^e}{\partial x}(\xi^{qrk}) \Bigg].
\end{aligned}$$

The expressions for the rest of the right-hand side terms are equivalent. Putting all the terms together we get the whole equation

$$
\begin{aligned}
w_q w_r w_s \sigma_{xx}^{eqrs}(t) J(\xi^{qrs}) = w_q w_r w_s J^e(\xi^{qrs})[\hat{\lambda}^e(\xi^{qrs}) + 2\hat{\mu}^e(\xi^{qrs})] \Bigg[\sum_i u_x^{eirs}(t) l_i'(\xi_1^q) \frac{\partial \xi_1^e}{\partial x}(\xi^{irs}) \\
+ \sum_j u_x^{eqjs}(t) l_j'(\xi_2^r) \frac{\partial \xi_2^e}{\partial x}(\xi^{qjs}) + \sum_k u_x^{eqrk}(t) l_k'(\xi_3^s) \frac{\partial \xi_3^e}{\partial x}(\xi^{qrk}) \Bigg] \\
+ w_q w_r w_s J^e(\xi^{qrs}) \hat{\mu}^e(\xi^{qrs}) \Bigg[\sum_i u_x^{eirs}(t) l_i'(\xi_1^q) \frac{\partial \xi_1^e}{\partial y}(\xi^{irs}) \\
+ \sum_j u_x^{eqjs}(t) l_j'(\xi_2^r) \frac{\partial \xi_2^e}{\partial y}(\xi^{qjs}) + \sum_k u_x^{eqrk}(t) l_k'(\xi_3^s) \frac{\partial \xi_3^e}{\partial y}(\xi^{qrk}) \Bigg] \\
+ w_q w_r w_s J^e(\xi^{qrs}) \hat{\mu}^e(\xi^{qrs}) \Bigg[\sum_i u_x^{eirs}(t) l_i'(\xi_1^q) \frac{\partial \xi_1^e}{\partial z}(\xi^{irs}) \\
+ \sum_j u_x^{eqjs}(t) l_j'(\xi_2^r) \frac{\partial \xi_2^e}{\partial z}(\xi^{qjs}) + \sum_k u_x^{eqrk}(t) l_k'(\xi_3^s) \frac{\partial \xi_3^e}{\partial z}(\xi^{qrk}) \Bigg]
\end{aligned}
$$

We see that the equation is solved for $\sigma_{xx}^{eqrs}(t)$ simply by eliminating the factor $w_q w_r w_s J^e(\xi^{qrs})$ which gives

$$
\begin{aligned}
\sigma_{xx}^{eqrs}(t) = [\hat{\lambda}^e(\xi^{qrs}) + 2\hat{\mu}^e(\xi^{qrs})] \Bigg[\sum_i u_x^{eirs}(t) l_i'(\xi_1^q) \frac{\partial \xi_1^e}{\partial x}(\xi^{irs}) \\
+ \sum_j u_x^{eqjs}(t) l_j'(\xi_2^r) \frac{\partial \xi_2^e}{\partial x}(\xi^{qjs}) + \sum_k u_x^{eqrk}(t) l_k'(\xi_3^s) \frac{\partial \xi_3^e}{\partial x}(\xi^{qrk}) \Bigg] \\
+ \hat{\lambda}^e(\xi^{qrs}) \Bigg[\sum_i u_x^{eirs}(t) l_i'(\xi_1^q) \frac{\partial \xi_1^e}{\partial y}(\xi^{irs}) \\
+ \sum_j u_x^{eqjs}(t) l_j'(\xi_2^r) \frac{\partial \xi_2^e}{\partial y}(\xi^{qjs}) + \sum_k u_x^{eqrk}(t) l_k'(\xi_3^s) \frac{\partial \xi_3^e}{\partial y}(\xi^{qrk}) \Bigg] \\
+ \hat{\lambda}^e(\xi^{qrs}) \Bigg[\sum_i u_x^{eirs}(t) l_i'(\xi_1^q) \frac{\partial \xi_1^e}{\partial z}(\xi^{irs}) \\
+ \sum_j u_x^{eqjs}(t) l_j'(\xi_2^r) \frac{\partial \xi_2^e}{\partial z}(\xi^{qjs}) + \sum_k u_x^{eqrk}(t) l_k'(\xi_3^s) \frac{\partial \xi_3^e}{\partial z}(\xi^{qrk}) \Bigg]
\end{aligned}
\tag{2.41}
$$

Now we have all the pieces needed to compute a solution for each element. To get the solution for the entire mesh we must assemble the contributions from each element.

2.7 Assembling the global matrices

We now have several local linear systems for each element Ω_e

$$\mathbf{M}^e \cdot \ddot{\mathbf{u}}_x^e(t) + \mathbf{k}_x^e(t) = \mathbf{f}_x^e(t)$$

but solving them in isolation would give a discontinuous solution at points that lie on the boundary between elements. This is because when we discretized the domain in Section 2.4 we split the integrals over the domain Ω into a sum of integrals over each element Ω_e. Now we must add those integrals back together. Points that lie on the boundary between elements are included in multiple integrals and contributions must therefore be counted once for each element which shares that point. We must therefore assemble the contributions from the local systems into a global system

$$\mathbf{M} \cdot \ddot{\mathbf{u}}_x(t) + \mathbf{k}_x(t) = \mathbf{f}_x(t).$$

which can then be solved to give a continuous solution.

In Figure 3 we see a mesh of 4 elements and 25 nodes. It has the associated global mass matrix

$$\mathbf{M} = \begin{bmatrix} M_0 & 0 & \dots & 0 \\ 0 & M_1 & \dots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & M_{24} \end{bmatrix}$$

with 25 entries along the diagonal, one for each node. Each of the elements in the mesh $\Omega_0 \dots \Omega_3$ is represented by the reference element shown in Figure 4 and have local mass matrices

$$\mathbf{M}^e = \begin{bmatrix} M_0^e & 0 & \dots & 0 \\ 0 & M_1^e & \dots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & M_8^e \end{bmatrix}$$

with 9 entries, one for each node. In order to assemble the global mass matrix, we must take the entries of each local mass matrix and add them to the corresponding entry in the global mass matrix. In order to find out which local node maps to which global node we use the mapping $\mathbf{F}_e(\xi)$ which we introduced in section 2.5. Applying this mapping to element Ω_0 we would get the mapping shown in Figure 5. Using the mapping for each element to assemble the global mass matrix gives

$$\mathbf{M} = \begin{bmatrix} M_0^0 + M_6^2 & 0 & \dots & 0 \\ 0 & M_2^0 + M_0^1 + M_8^2 + M_6^3 & \dots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & M_4^3 \end{bmatrix}.$$

This procedure can then by applied to the vectors $\mathbf{k}$ and $\mathbf{f}$ as well.

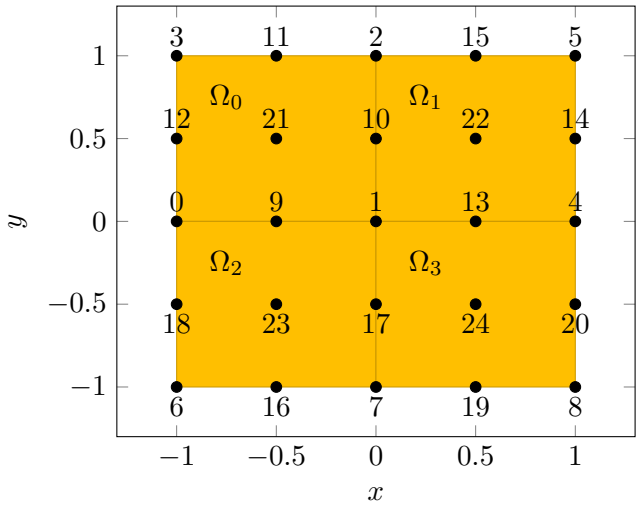


Figure 3: A mesh Ω consisting of 4 elements $\Omega_0 \dots \Omega_3$. We have used GLL quadrature of degree 2, resulting in $(2+1)^2 = 9$ nodes per element (see Section 2.6) and a total of 25 nodes in the mesh. Many of the nodes fall on the edge between the elements and contributions from the elements sharing the node must be added together when assembling the global system.

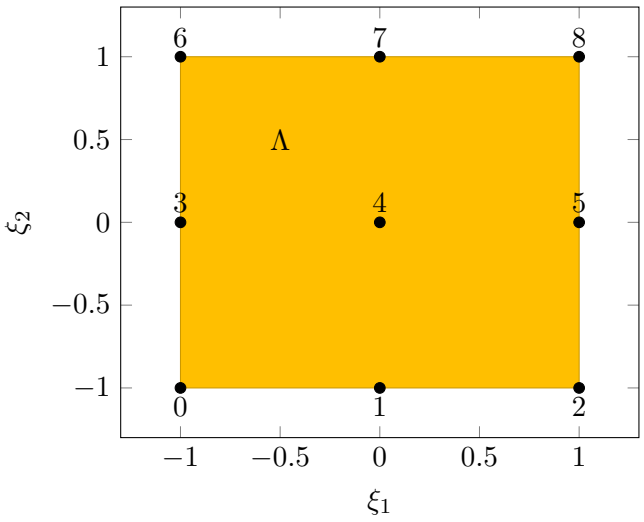


Figure 4: The reference element Λ to which all elements in a mesh are mapped for computation. The nodes are the GLL points of order 2. The integrals are computed on this domain and the result for each local node is mapped to the corresponding global node in Figure 3 using the element-specific mapping $\mathbf{F}_e(\xi)$.

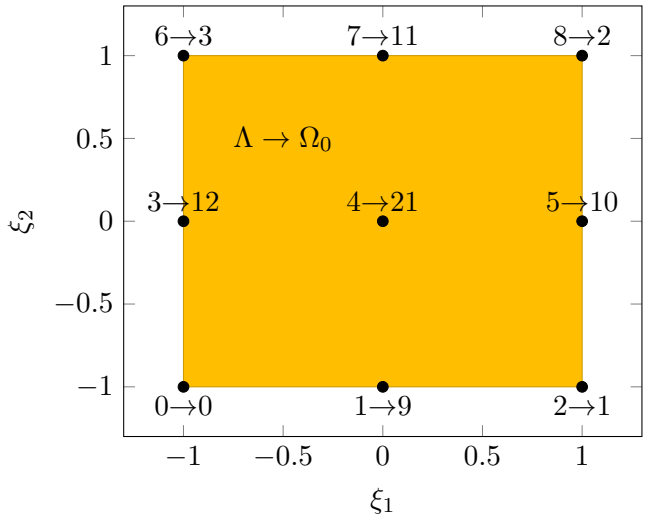


Figure 5: The mapping of nodes between the reference element Λ from Figure 4 to element Ω_0 in the mesh in Figure 3. This shows how the local nodes of element Ω_0 are mapped to the global nodes of the mesh Ω.

2.8 Solving the system

What we have now is a diagonal linear system given by

$$\mathbf{M} \cdot \ddot{\mathbf{u}}_x(t) + \mathbf{k}_x(t) = \mathbf{f}_x(t) \tag{2.42}$$

with one equation for each of the nodes in the mesh. Solving it is trivial since it is diagonal. We now discretize the time axis and introduce a method for solving the ODE in Equation (2.42) for the displacement $\mathbf{u}_x(t)$ and to march the system forward in time. We discretize the time axis by introducing a constant time step Δt. This gives time $t[n] = n\Delta t + t_0$ for any integer n where n is the time step number and t_0 is the initial time when $n = 0$. Now we can rewrite Equation (2.42)

$$\mathbf{M} \cdot \ddot{\mathbf{u}}_x(t[n]) + \mathbf{k}_x(t[n]) = \mathbf{f}_x(t[n]).$$

We can make it a little more succinct by writing the time-dependent vectors as $\ddot{\mathbf{u}}_x(t[n]) = \ddot{\mathbf{u}}_x^n$

$$\mathbf{M} \cdot \ddot{\mathbf{u}}_x^n + \mathbf{k}_x^n = \mathbf{f}_x^n. \tag{2.43}$$

To solve the ODE and march the system forward in time we will use the Newmark Method (Fichtner 2011, ch. 2.5.1) which we can express with the following two equations

$$\dot{u}^{n+1} \approx \dot{u}^n + (1-\gamma)\Delta t\ddot{u}^n + \gamma\Delta t\ddot{u}^{n+1}$$
$$u^{n+1} \approx u^n + \Delta t\dot{u}^n + \frac{1-2\beta}{2}\Delta t^2\ddot{u}^n + \beta\Delta t^2\ddot{u}^{n+1}$$

with a time step Δt and configurable parameters $\beta \in [0, \frac{1}{2}]$ and $\gamma \in [0, 1]$. We choose $\gamma = 0.5$ which is the only value that gives second-order accuracy while choosing $\beta = 0$ at the same time yields a second order explicit method (Fichtner 2011, ch. 2.5.1). We then get

$$\dot{\mathbf{u}}_x^{n+1} \approx \dot{\mathbf{u}}_x^n + \frac{1}{2}\Delta t \ddot{\mathbf{u}}_x^n + \frac{1}{2}\Delta t \ddot{\mathbf{u}}_x^{n+1} \tag{2.44}$$

$$\mathbf{u}_x^{n+1} \approx \mathbf{u}_x^n + \Delta t \dot{\mathbf{u}}^n + \frac{1}{2}\Delta t^2 \ddot{\mathbf{u}}_x^n. \tag{2.45}$$

Finding the solution for time step $n + 1$ amounts to a few simple steps. First solve Equation (2.43) for the acceleration $\ddot{\mathbf{u}}_x^{n+1}$. Then use (2.44) to compute the velocity $\dot{\mathbf{u}}_x^{n+1}$ and Equation (2.45) to compute the displacement $\mathbf{u}_x^{n+1}$.

We choose this method because it is simple to implement, efficient, has a small memory footprint and has second order accuracy. The reason that it's simple and efficient is that it is explicit, so each row of the vector can be evaluated independently with only a few operations. It is also readily paralellized since there are no dependencies between elements of the vectors. It does depend on the acceleration of the previous iteration $\ddot{\mathbf{u}}_x^n$ as well as the current $\ddot{\mathbf{u}}_x^{n+1}$, but this can actually be improved during implementation so that no extra data needs to be stored.

3 Architecture

Now that we have developed expressions that can be used to solve the elastic wave equation, we need to investigate how we can use the parallel hardware of a Graphics Processing Unit (GPU) to solve the equation quickly.

To explain how something can be implemented on the GPU we take a look at the Nvidia GPU architecture and how the CUDA language allows us to write programs that utilise the parallelism of a GPU. Refer to the CUDA Programming Guide, NVIDIA 2018(a), for a more detailed explanation. The topics discussed in this section applies to most GPUs from other vendors, but because of the confusion often caused by the different terms used by each vendor we stick with the jargon used by Nvidia. For an explanation of how these different terms compare, please see Hennessy and Patterson 2011, ch. 4.4.

3.1 Nvidia GPU Architecture

The Nvidia architecture is based on the parallel execution of so-called *CUDA threads*. This is not the same as a traditional CPU thread because it is not independent, instead these threads are executed in groups of 32 called *warps* that all perform the same operations, but on different data. The part of the hardware which executes the warps is called a *Streaming Multiprocessor* (SM). In the Nvidia Pascal architecture (which we will be using) each SM can execute two warps or 64 CUDA threads. So a GP100 GPU (NVIDIA 2018[b]) with 60 SMs can execute a maximum of $60 \cdot 64 = 3840$ CUDA threads concurrently. This means that it can

execute one instruction on 3840 different numbers in one clock cycle. With a clock frequency of 1.48 GHz it has a theoretical performance of 3840 FLOP $\cdot 1.48\,\text{GHz} = 5.3$ TFLOPS for double precision numbers and 10.6 TFLOPS for single precision. Compared to a server with two Intel Xeon E5-2630 v4 with 10 cores running at 3.1 GHz (*Intel Xeon Processor E5-2630 v4 Datasheet* 2018) with a theoretical performance of 20 FLOP $\cdot 3.1\,\text{GHz} = 62$ GFLOPS, it seems to be much better. This example is of course incredibly simplified, but it does illustrate the potential of using a GPU instead of a CPU.

Figure 6 shows the floor plan of the GP100 GPU. As we have mentioned, we see that it has 60 SMs each capable of executing 64 *CUDA Threads* on the single precision execution units represented by the small green squares, or 32 *CUDA Threads* with double precision (yellow squares). We will not look at the hardware in detail, but rather how to use it through the CUDA programming language.

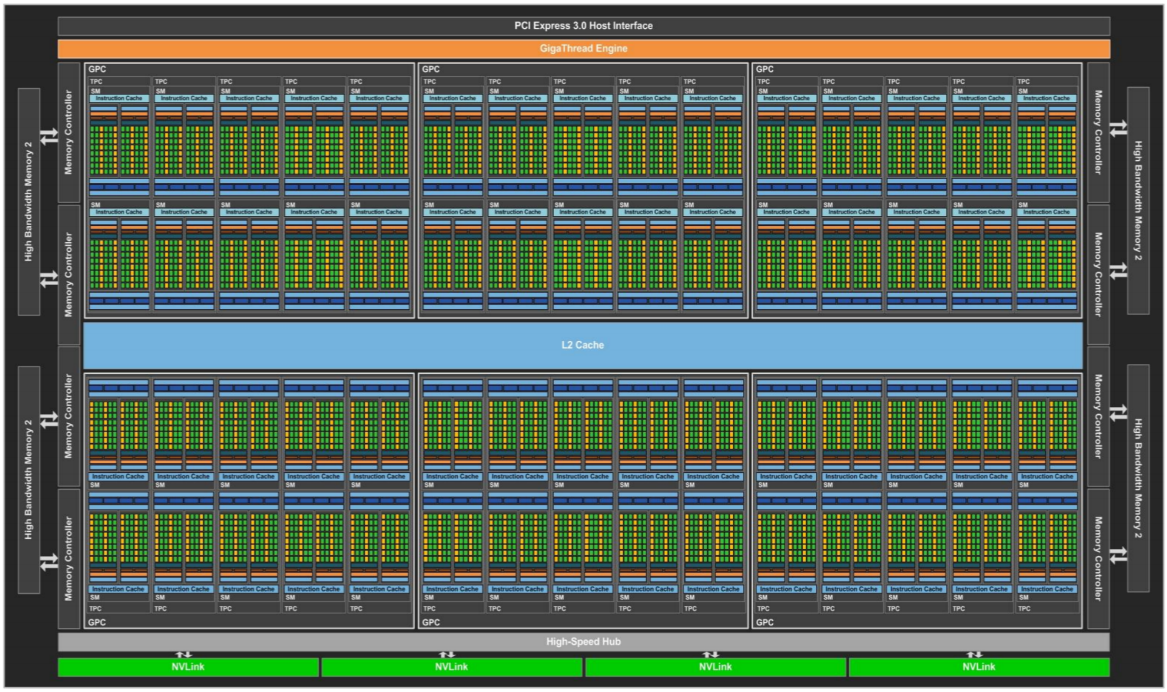


Figure 6: Floor plan of a GP100 GPU taken from NVIDIA 2018(b). We see the 60 SMs, each with 64 single precision execution units and 32 double precision execution units. When a function is run on the GPU it is divided in to thread blocks which are assigned to each SM by a block scheduler (GigaThread Engine). Each SM has a warp scheduler which is capable scheduling two *Warps* to utilise the 64 single precision execution units represented by the green squares. The 32 double precision execution units are represented by yellow squares.

3.2 CUDA programming

Listing 1 shows a simple program written in CUDA C++. The function defined with the prefix `__global__` is called a *kernel* which is a function that runs on the GPU. It can be called from normal CPU code using the syntax shown on line 31 in the code listing. The numbers in the angled brackets are the number of threads per thread block and the number

of thread blocks respectively. In the code a total of $\lceil \frac{N}{num\_threads\_per\_block} \rceil = 4096$ thread blocks are created with 256 threads each resulting in exactly one thread per entry in the array `x`. In the kernel a unique index `i` is computed for each thread using the block `blockIdx`, which identifies which block a thread belongs to, `blockDim`, the number of threads in a block, and `threadIdx`, the the id of the thread within the block. This way, each thread `i` concurrently computes one entry of the array `x[i]` each. A GPU has a dedicated memory called *global memory*, 16 GB for a P100 accelerator (NVIDIA 2018[c]), so all data needed for computation should be stored there if possible since accessing host memory (RAM) is much slower. We do this by allocating space in the global memory by using `cudaMalloc` on line 14. The data is then copied from the host to the device using `cudaMemcpy`. Launching a CUDA kernel does not make the CPU wait until it is finished, allowing the CPU to do concurrent work. We therefore have to use `cudaDeviceSynchronise` in order to ensure that the GPU has finished before we access the data.

```
1  __global__ void Square(float * x, int n) {
2    int i = blockIdx.x * blockDim.x + threadIdx.x;
3    if ( i < n ) {
4      x[i] *= x[i];
5    }
6  }
7
8  int main() {
9    constexpr int N = 1 << 20;
10   float x[N];
11   float * d_x;
12
13   // Allocate the array in the global memory of the device
14   cudaMalloc(&x, N*sizeof(float));
15
16   // initialize x the array on the host
17   for (int i = 0; i < N; i++) {
18     x[i] = i;
19   }
20
21   // Copy the array to the device
22   cudaMemcpy(d_x, x, N*(sizeof(float), cudaMemcpyHostToDevice);
23
24   // Run kernel on the GPU
25   int num_threads_per_block = 256;
26   // We must use ceil() to round the number of blocks up
27   // to ensure that we do not create one block too little
28   int num_blocks = ceil((double)N / num_threads_per_block);
29   // The total number of threads is num_blocks * num_threads_per_blocks
30   // which is greater than N
31   Square<<<num_threads_per_block, num_blocks>>>(x, N);
32
33   // Do some work on the CPU in parallel
34   // ...
35
36   // Wait for GPU to finish before accessing on host
37   cudaDeviceSynchronize();
38
39   // Copy the data back to the host
40   cudaMemcpy(x, d_x, N*sizeof(float), cudaMemcpyDeviceToHost);
41
```

```
42    // Free GPU memory
43    cudaFree(x);
44
45    // Do something with the data
46    // ...
47
48    return 0;
49 }
```

Listing 1: Sample CUDA code adapted from Harris 2018. It shows the definition and usage of a simple CUDA kernel function. Only a small amount of extra code, mostly memory allocation and synchronisation, is required to run code on the GPU.

4 Implementation

In this chapter we describe how to implement a solution to the 3D elastic wave equation based on the theory developed in Chapter 2 using the Graphics Processing Unit (GPU) to accelerate the most costly computations. Everything is written in C++ while using CUDA to implement the parts that run on the GPU and MPI for communication between nodes in the cluster. Of course any implementation might also include mesh processing or mesh creation as well as post processing of data, but we will only look at solving the wave equation.

4.1 Program structure

The implementation consists of several functions shown in the pseudo-code in Algorithm 1 which summarises the most important steps of the algorithm. The loop runs from simulated time $t = 0$ until $t = T$ in steps of Δt for a total of $n_t = \frac{T}{\Delta t}$ iterations. `ComputeStress()` implements Equation (2.41) in order to compute all the stresses $\sigma_{xx}, \sigma_{xy} \dots \sigma_{zz}$ for each of the n_b^3 GLL points in each of the n_e elements. `ComputeLocalK()` implements Equation (2.39) also for each point of each element. `AssembleGlobalK()` used the ideas behind assembling explained in Chapter 2.7 in order to take the contributions from each elemental vector $\mathbf{k}^e$ and add them all together in the global vector $\mathbf{k}$. Finally the system is solved and marched forward one timestep in `SolveSystem()` by using the Newmark Method as explained in Chapter 2.8.

```
Solve() begin
    while t < T do
        ComputeStress()
        ComputeLocalK()
        AssembleGlobalK()
        SolveSystem()
    end
end
```

Algorithm 1: Pseudo-code showing the computations performed in the programs time loop. The loop runs until the simulated time t reaches the total simulation time T. In each timestep computations are first performed on each element in `ComputeStress()` and `ComputeLocalK()` before the global system is assembled in `AssembleGlobalK()` and solved in `SolveSystem()`.

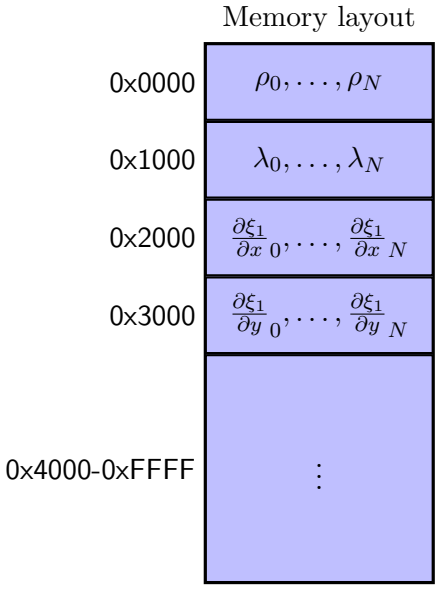


Figure 7: All data is stored in contiguous arrays in GPU memory. Each array has $N = n_e n_b^3$ entries which is the total number of local nodes.

The most time consuming steps are the functions `ComputeStress()` and `AssembleLocalK()` which implement Equations (2.41) and (2.39) respectively. Both of these expressions need many operations to compute just a single element and they must read data from many different places in memory which make them the most time consuming operations and the best candidates for optimisation. All the other operations are also accelerated by the GPU, but they are much simpler computations and leave less to be gained by optimisation so we will not focus on them.

4.2 Single GPU acceleration with CUDA

Before we start implementing the algorithms, it is important that the data is laid out in memory in a way which facilitates fast computations. There is one property which we will seek to obtain and that is locality which means that we want data which is used during a short period of time to be close in memory. This allows for automatic optimisations by the hardware such as caching and prefetching so that the computation will be faster with relatively little work required for it. In order to achieve this we organise all the data $(\rho, \lambda, \frac{\partial \xi}{\partial x_i}, \ldots)$ into separate arrays which are contiguous in memory as shown in Figure 7. We shall see that by having such a memory layout we can order computations in a way which takes advantage of it, thus improving speed.

To compute the stress we implement equation (2.41) while trying to make good use of the memory layout in Figure 7 to improve performance. In the equation we see that for each index $eqrs$ a total of 15 different arrays must be accessed (that's $\sigma, \lambda, \mu, (3x)u_i, (9x)\frac{\partial \xi}{\partial x_i}$, since l' is constant and only has values at the n_b^3 local GLL points). In order to improve the performance, we split up the computations such that as few arrays as possible are accessed

simultaneously. Since the equation consists of 9 sums we can compute each of these sums one at a time for each node and then add together the contributions, which will allow for much better utilisation of the cache since only a few arrays are accessed simultaneously for each computation. This leads us to an implementation of `ComputeStress()` shown in Algorithm 2.

`ComputeStress(`σ_{xx}`)` **begin**

$\sigma_{xx}^{eqrs} = \sum_i u_x^{eirs} l_i'(\xi_1^q) \frac{\partial \xi_1}{\partial x}$

$\sigma_{xx}^{eqrs} += \sum_j u_x^{eqjs} l_j'(\xi_2^r) \frac{\partial \xi_2}{\partial x}$

$\sigma_{xx}^{eqrs} += \sum_k u_x^{eqrk} l_k'(\xi_3^s) \frac{\partial \xi_3}{\partial x}$

$\sigma_{xx}^{eqrs} *= (\lambda^{eqrs} + 2\mu^{eqrs})$

$\texttt{temp}^{eqrs} = \sum_i u_y^{eirs} l_i'(\xi_1^q) \frac{\partial \xi_1}{\partial y}$

$\texttt{temp}^{eqrs} += \sum_j u_y^{eqjs} l_j'(\xi_2^r) \frac{\partial \xi_2}{\partial y}$

$\texttt{temp}^{eqrs} += \sum_k u_y^{eqrk} l_k'(\xi_3^s) \frac{\partial \xi_3}{\partial y}$

$\texttt{temp}^{eqrs} += \sum_i u_z^{eirs} l_i'(\xi_1^q) \frac{\partial \xi_1}{\partial z}$

$\texttt{temp}^{eqrs} += \sum_j u_z^{eqjs} l_j'(\xi_2^r) \frac{\partial \xi_2}{\partial z}$

$\texttt{temp}^{eqrs} += \sum_k u_z^{eqrk} l_k'(\xi_3^s) \frac{\partial \xi_3}{\partial z}$

$\sigma_{xx}^{eqrs} += \lambda^{eqrs} \texttt{temp}^{eqrs}$

end

Algorithm 2: The algorithm for computing σ_{xx} given by Equation (2.41). Since the algorithm is supposed to run on a GPU, there is no need for loops, because there is a single thread allocated for each index $eqrs$ and each line is executed once for each thread, and thus once for each index. If implemented with proper synchronisation, the threads in a block (which share a cache) execute only a single line at a time, which means that it only accesses the arrays needed by that line and that the cache is not polluted by other memory accesses. The index $eqrs$ corresponds to the array index $i = en^3 + qn^2 + rn + s$.

In Algorithm 2 we see a way of computing Equation (2.41) which attempts to minimise the number of arrays that are accessed by threads simultaneously. It utilises a temporary array `temp` in order to store the partial results, before multiplying by λ and μ and adding to σ_{xx}. As we can see, it adds together the individual sums in of each part of Equation (2.41) before multiplying by the factor $\lambda + 2\mu$ and adding to σ_{xx} it then adds together the remaining 6 sums, which are then multiplied by λ and added into σ_{xx}.

In order to assemble the local k-vector given by (2.39), we can use the same approach as for the stress. The only difference is the addition of the integration weights w and the Jacobian determinant J. The integration weights do not affect memory accessing, since there are only n_b different values for them which are known at compile-time. The Jacobian determinant, however, means that an additional array is accessed in each step. On the other hand, arrays λ and μ are not needed for this computation, and the `temp` array is also not needed, since the sums are not multiplied by anything, so they can be computed and then added directly,

which gives Algorithm 3.

`ComputeLocalA(`A_x`)` **begin**

$$
\begin{aligned}
A_x^{eqrs} &= w_r w_s \textstyle\sum_l w_l \sigma_{xx}^{elrs}(t) l_q'(\xi_1^l) \frac{\partial \xi_1}{\partial x} J(\xi^{lrs}) \\
A_x^{eqrs} &+= w_q w_s \textstyle\sum_m w_m \sigma_{xx}^{eqms}(t) l_r'(\xi_2^m) \frac{\partial \xi_2}{\partial x} J(\xi^{qms}) \\
A_x^{eqrs} &+= w_q w_r \textstyle\sum_n w_n \sigma_{xx}^{eqrn}(t) l_s'(\xi_3^n) \frac{\partial \xi_3}{\partial x} J(\xi^{qrn}) \\
A_x^{eqrs} &+= w_r w_s \textstyle\sum_l w_l \sigma_{xy}^{elrs}(t) l_q'(\xi_1^l) \frac{\partial \xi_1}{\partial y} J(\xi^{lrs}) \\
A_x^{eqrs} &+= w_q w_s \textstyle\sum_m w_m \sigma_{xy}^{eqms}(t) l_r'(\xi_2^m) \frac{\partial \xi_2}{\partial y} J(\xi^{qms}) \\
A_x^{eqrs} &+= w_q w_r \textstyle\sum_n w_n \sigma_{xy}^{eqrn}(t) l_s'(\xi_3^n) \frac{\partial \xi_3}{\partial y} J(\xi^{qrn}) \\
A_x^{eqrs} &+= w_r w_s \textstyle\sum_l w_l \sigma_{xz}^{elrs}(t) l_q'(\xi_1^l) \frac{\partial \xi_1}{\partial z} J(\xi^{lrs}) \\
A_x^{eqrs} &+= w_q w_s \textstyle\sum_m w_m \sigma_{xz}^{eqms}(t) l_r'(\xi_2^m) \frac{\partial \xi_2}{\partial z} J(\xi^{qms}) \\
A_x^{eqrs} &+= w_q w_r \textstyle\sum_n w_n \sigma_{xz}^{eqrn}(t) l_s'(\xi_3^n) \frac{\partial \xi_3}{\partial z} J(\xi^{qrn})
\end{aligned}
$$

end

Algorithm 3: The Algorithm used to compute the local k-vector given by Equation (2.39). It is very similar to Algorithm 2 but does not need a temporary array.

Most of the work done in these algorithms lies in the sums. Therefore it is also important that the computation of the sums happens in an efficient manner. To compute the sums, we use the implementation given in Listing 2, which is a rather straight-forward CUDA-C++-implementation of a sum with a variable stride, however it uses loop unrolling through a compile-time constant `N` which avoids the overhead of looping. The implementation would have been trivial, if it wasn't for the fact the stride is different for each array in each sum. Let us look at the example of computing the sum

$$\sum_i u_x^{eirs} l_i'(\xi_1^q) \frac{\partial \xi_1}{\partial x}^{eirs}$$

using `SumStride`. Since the elements of the arrays u_x and $\frac{\partial \xi_1}{\partial x}$ are adjacent in memory with indices given by `index` $= en_b^3 + qn_b^2 + rn_b + s$, the stride of this sum is `stride` $= n_b^2$ since the loop variable i replaces q in the index expression. The sum is thus computed by calling `SumStride(`u_x^{e0rs}, $\frac{\partial \xi_1}{\partial x}^{e0rs}$, $l_i'(\xi_1^q), n_b^2$`)`.

```
template <int N>
__device__ float SumStride(float u[], float xi[], float l[], int stride)
{
    float sum = 0;
    #pragma unroll
    for (int i = 0; i < N; ++i) {
        sum += u[i*stride] * xi[i*stride] * l[i*N];
    }
    return sum;
}
```

Listing 2: Implementation of sum with variable stride. The stride of the array `l`, which represents the spatial derivatives of the basis functions $l_i'(\xi)$, is always `N`. The `__device__` keyword is needed to make the code a function that can run on the GPU and be called from a kernel. Since the number of iterations in the loop `N` is the same as the number of GLL points n_b, we can actually use C++ templates in our CUDA code to generate one function for each `N` and force loop unrolling using the `#pragma unroll` directive to improve performance.

We similarly implement a summing function to be used when computing the k-vector. It is uses the same idea, but is a bit longer since more parameters are involved in the same, as shown in Listing 3.

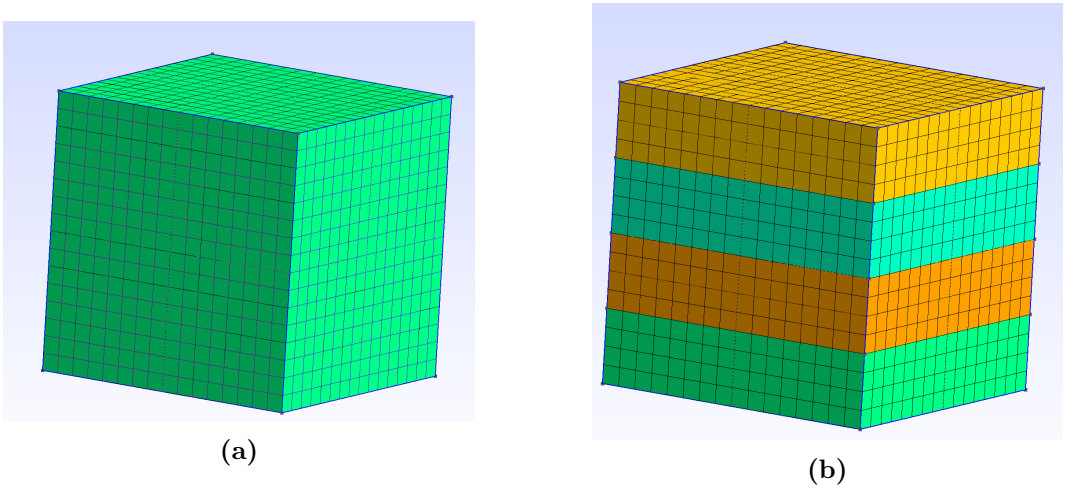
```
template <int N>
__device__ float SumStride(float s[], float xi[], float j[], float w[],
        float l[], int stride)
{
    float sum = 0;
    #pragma unroll
    for (int i = 0; i < N; ++i) {
        sum += s[i*stride] * xi[i*stride] * j[i*stride] * l[i] * w[i];
    }
    return sum;
}
```

Listing 3: Implementation of sum with variable stride to be used when computing the k-vector. Uses templates to enable loop unrolling, as described in the caption of Listing 2.

4.3 Multiple GPU acceleration with MPI

The implementation presented in the previous section is enough to compute a solution using a single GPU but if we want to solve larger systems we have to use more GPUs. Using the Message-Passing Interface (MPI) we can implement the algorithm in such a way that it can run in parallel on any number of computers. An excerpt from the MPI standard *MPI: A Message-Passing Interface Standard* 2019 says "MPI addresses primarily the message-passing parallel programming model, in which data is moved from the address space of one process to that of another process through cooperative operations on each process". This means that MPI allows for communication between processes, and these processes may very well run on physically separate hardware. MPI is therefore a way for us to split computations over multiple GPUs running on different machines on a network, share the results between the GPUs and consolidate contributions from each GPU into a single solution for the whole system. The question is then how the computations should be distributed between the GPUs. A sensible approach to this is to attempt to minimise the amount of communication between the GPUs since it is often memory bandwidth that is the bottleneck.

Domain decomposition is a method of solving a PDE by splitting the domain into multiple non-overlapping regions. In our case the domain will be some mesh, like the one shown in Figure 8a, and it can be decomposed into multiple partitions as shown in Figure 8b. When decomposing the domain this way, each GPU is assigned one partition and it needs only communicate with the adjacent partitions. So in this example each GPU does two communications (since they only have two adjacent partitions) and it only needs to communicate data for the nodes which lie in the interface of each partition. This leads to a network topology as shown in Figure 9. One can of course imagine a different decomposition which attempts to make the interfaces smaller by partitioning into cubes instead of slices, though this means each partition is adjacent to more than two other partitions and will need to perform more communication while less data can be transferred in communication. This could be more efficient since these communications can be performed in parallel, but it is not as simple to



(a) **(b)**

Figure 8: A mesh (a) decomposed into 4 partitions (b). Only the surface elements are shown. Each partition shares only its upper and lower faces with another partition, thus information about the inner elements does not need to be communicated.

implement. Using MPI we can implement a function which communicates the data between each partition using the functions defined in the MPI standard. Listing 4 shows how the data in the global K-vector is sent and received between adjacent partitions.

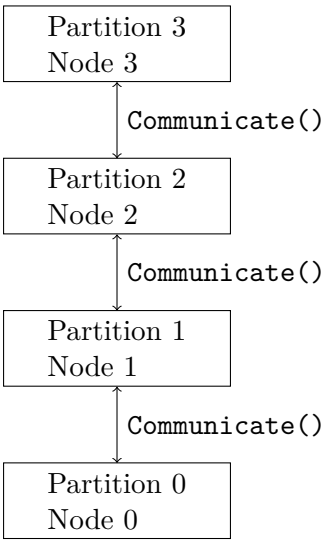


Figure 9: Network topology used with the partitioning scheme shown in Figure 8b. Each node calls `MPI_Sendrecv()` in order to exchange information about points which lie on the interface it shares with the nodes above and below it. The function `Communicate()` is described in Listing 4 and describes how each node uses MPI to communicate with the other nodes.

```
__device__ float Communicate(float send_above[], float receive_below[],
        float send_below[], float receive_above[], int n_nodes_per_face,
        int node_above, int node_below)
{
    MPI_Sendrecv(send_above, n_nodes_per_face, MPI_FLOAT, node_above, 0,
            receive_below, n_nodes_per_face, MPI_FLOAT, node_below, 0,
            MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Sendrecv(send_below, n_nodes_per_face, MPI_FLOAT, node_below, 0,
            receive_above, n_nodes_per_face, MPI_FLOAT, node_above, 0,
            MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

Listing 4: The function which performs communication between GPUs given the partitioning scheme shown in Figure 8b. The values at the n_nodes_per_face nodes of each interface is sent and received from the buffers send_above, receive_below and send_below, receive_above respectively. No special care must be taken for the uppermost and lowermost partitions, since the MPI implementation should do nothing if the values given for node_above and node_below do not exist.

5 Results

5.1 Comparison with the analytical solution

To ensure the correctness of the implementation we compare the results of a simulation with the analytical solution for a point force source in a homogeneous isotropic and unbounded

medium given by (Aki and Richards 2002, ch. 4)

$$\begin{aligned} u_i(\mathbf{x},t) =& \frac{1}{4\pi\rho}\left(3\gamma_i\gamma_j - \delta_{ij}\right)\frac{1}{r^3}\int_{\frac{r}{v_p}}^{\frac{r}{v_s}} \tau X_0(t-\tau)d\tau \\ &+ \frac{1}{4\pi\rho v_p^2}\gamma_i\gamma_j\frac{1}{r}X_0\left(t-\frac{r}{v_p}\right) - \frac{1}{4\pi\rho v_s^2}\left(\gamma_i\gamma_j - \delta_{ij}\right)\frac{1}{r}X_0\left(t-\frac{r}{v_s}\right). \end{aligned} \tag{5.1}$$

v_p and v_s are the primary (pressure) and secondary (shear) wave velocities, respectively, which are related to the Lamé parameters λ and μ and the density ρ (which we were using in Chapter 2) in the following way (Mavko, Mukerji, and Dvorkin 2009, ch. 3)

$$\begin{aligned} v_p &= \sqrt{\frac{\lambda+2\mu}{\rho}} \\ v_s &= \sqrt{\frac{\mu}{\rho}}. \end{aligned}$$

$u_i(\mathbf{x},t)$ is the displacement in spatial direction i (i.e. x, y or z) at position $\mathbf{x}$ and time t. $X_0(t)$ is the amplitude of the point force source at time t. We assume that the point source is situated in $(0,0,0)$ which means that the distance from the point source is $r = |\mathbf{x}|$. If a point $\mathbf{x}$ is written as $\mathbf{x} = (x_0, x_1, x_2)$ then the coefficients γ_i can be written as $\gamma_i = x_i/r$ where is either 0, 1 or 2. δ_{ij} is the Kronecker delta.

The mesh used is similar to the the cube shown in figure 8a, but consisting of $100 \cdot 100 \cdot 100 = 10^6$ cubic elements each $(10\,\mathrm{km})^3$ in size, making the mesh a cube of $(1000\,\mathrm{km})^3$. The density is $\rho = 1000\,\mathrm{kg\,m^{-3}}$ and the wave velocities are $v_p = 2000\,\mathrm{m\,s^{-1}}$ and $v_s = 1000\,\mathrm{m\,s^{-1}}$.

Since we are using the explicit Newmark time scheme there exists a maximum value for the time step Δt over which the simulation is not stable. To ensure the stability of the simulation we use the CFL stability condition (Fichtner 2011, ch. 2.5). This condition gives a relation between the timestep Δt, maximum wave velocity v_{pmax} and mesh resolution h_{min}

$$\frac{v_{pmax}\Delta t}{h_{min}} = C \leq C_{max}$$

where C is called the Courant number and C_{max} is an upper limit which depends on the time scheme and regularity of the mesh. It is important to choose C_{max} correctly, but according to Dimitri Komatitsch, Tsuboi, et al. 2005 there exists no exact method of determining the maximum Courant number for the SEM. In practice the value $C_{max} = 0.5$ works well for very regular meshes, but would need to be smaller for more irregular meshes.

Since all the elements in the mesh have the same size, 10 km, and they are of order 4 we know that the smallest distance between two GLL points in an element is approximately (Fichtner 2011, appendix A.3.2)

$$h_{min} \approx 0.3454 \cdot 5\,\mathrm{km} \approx 1727\,\mathrm{m}.$$

This means that the time step must be no larger than

$$\Delta t \leq \frac{C_{max} h_{min}}{v_{pmax}} = \frac{0.5 \cdot 1727\,\mathrm{m}}{2000\,\mathrm{m\,s^{-1}}} = 0.432\,\mathrm{s}.$$

To be on the safe side, we choose $\Delta t = 0.25\,\mathrm{s}$. To avoid reflections we simulate only until the waves reach the edge of the mesh, so we get a total simulation time T

$$T = \frac{500\,\mathrm{km}}{2\,\mathrm{km\,s^{-1}}} = 250\,\mathrm{s}$$

which means the number of timesteps n_t must be

$$n_t = \frac{T}{\Delta t} = \frac{250\,\mathrm{s}}{0.25\,\mathrm{s}} = 1000.$$

We let the point force source be a Ricker wavelet

$$X_0(t) = (1 - 2\pi^2 f_0^2 (t - t_0)^2) e^{-\pi^2 f_0^2 (t-t0)^2} \tag{5.2}$$

with dominant frequency $f_0 = 1/50\,\mathrm{Hz}$ and time shift $t_0 = 60\,\mathrm{s}$ in direction z placed in the middle of the mesh, at the coordinate $\mathbf{x}_{source} = (500\,\mathrm{km}, 500\,\mathrm{km}, 500\,\mathrm{km})$ and we record the z-component of the displacement at the position $\mathbf{x}_{receiver} = (500\,\mathrm{km}, 500\,\mathrm{km}, 600\,\mathrm{km})$.

A comparison of the simulated solution with the analytical solution is shown in Figure 10. We see that they are mostly similar, but the error is larger when the derivative is larger, likely caused by aliasing of high frequency components of the wave.

To investigate the source of the error in Figure 10 we do multiple tests with the same mesh, but with sources with different dominant frequencies f_0. We compute the error using the following formula

$$error = \sum |u_z - \bar{u}_z|^2$$

where u_z is the analytical solution and $\bar{u}_z$ is the simulated solution. The results shown in Figure 11 show that the error increases with the frequency, as expected.

To verify that the implementation also works when accelerated by multiple GPUs we partition the mesh as shown in Figure 8b and assign each partition to a GPU. The results shown in Figure 12 show it works just as well as on a multiple GPUs as on a single GPU.

5.2 Comparison with Specfem

While we have shown that the implementation produces a solution that is close to the analytical solution there is still an error. We wish to know if this error is just a result of the

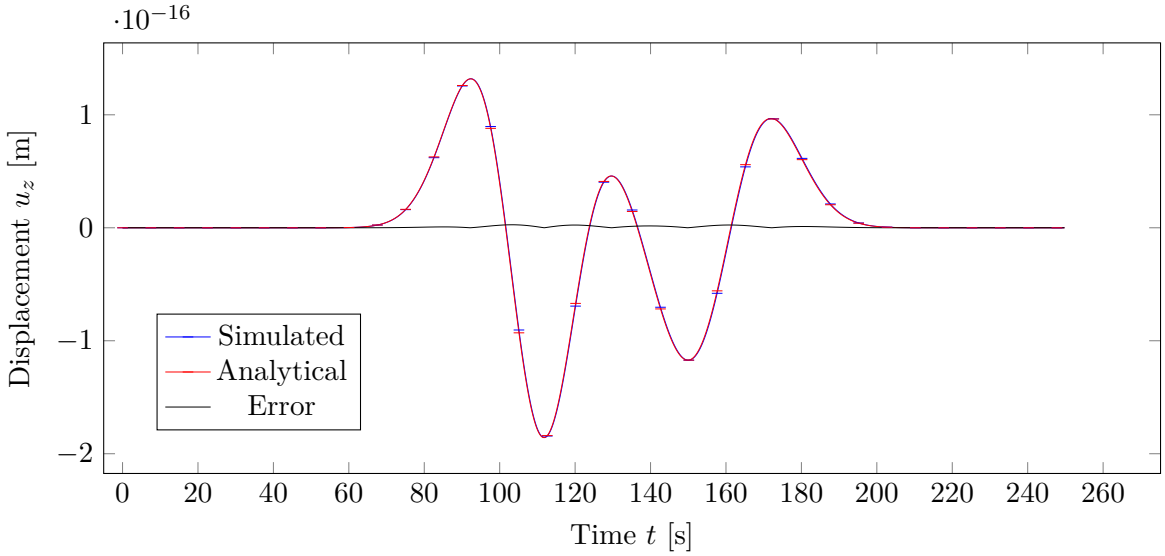


Figure 10: Comparison of a simulated Ricker wavelet in an isotropic, homogeneous mesh with the analytical solution. The simulation time is restricted to 250 s to avoid reflections, since the mesh is not unbounded whereas the analytical solutions assumes an unbounded medium. The source is at the centre of mesh in the point (500 km, 500 km, 500 km) and the receiver is in (500 km, 500 km, 600 km), at a distance of 100 km from the source in the z-direction. The lines across the graphs highlight the values of each graph at certain times. Where there is a larger gap between the lines, the error is larger. The error is the largest when the displacement is changing rapidly(larger derivative) because of more aliasing of high frequency components.

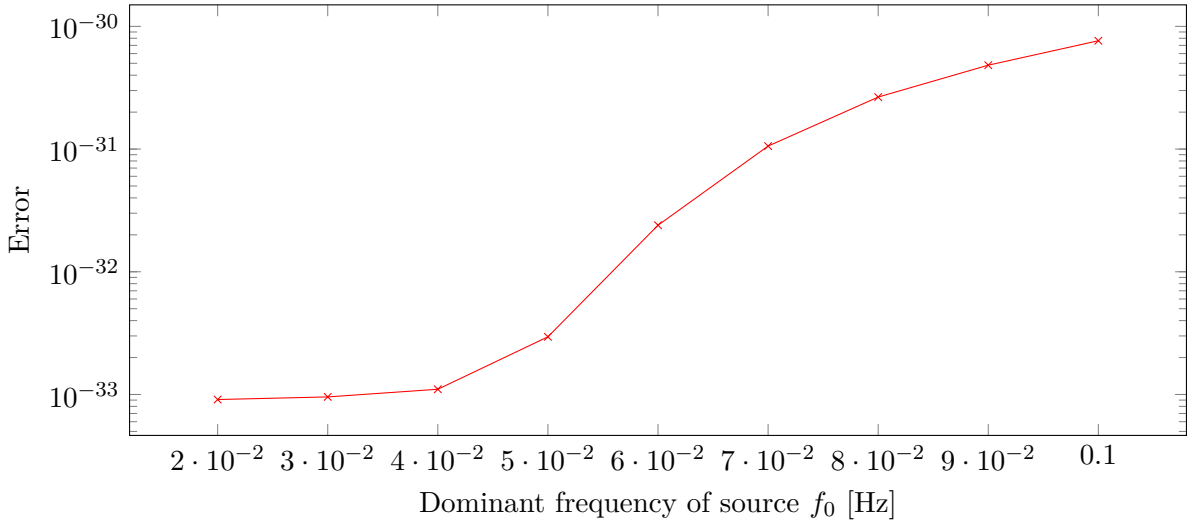


Figure 11: The error of the simulation compared to the analytical solution for different dominant frequencies f_0 of the Ricker wavelet source. Since the Ricker wavelet with larger f_0 contains more high-frequency components, we expected the error increase with f_0 because of aliasing. We see that expectation confirmed here. The error does not change much until 0.04 Hz, and at 0.05 Hz it quickly increases to 3 times that of 0.02 Hz.

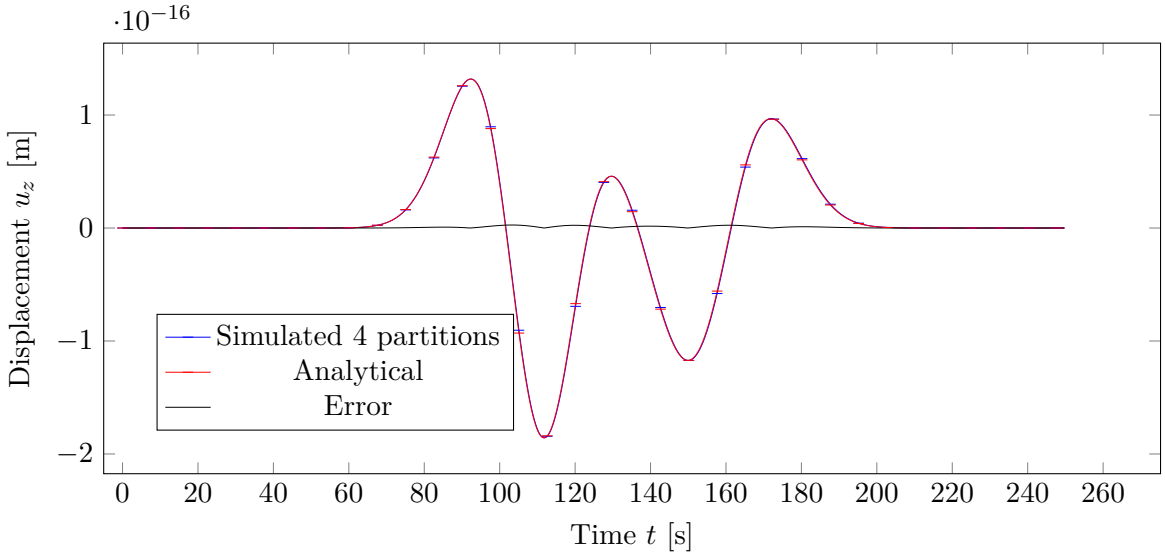


Figure 12: Comparison of distributed solver with analytical solution. The simulation was performed with the same parameters as in Figure 10 but on a mesh divided into 4 partitions as shown in Figure 8b. This should give exactly the same result and with perfect scaling the simulation would run 4 times faster. We see that the results is exactly the same as for the single-GPU simulation from Figure 10.

method, or whether the implementation has a fault. To check this we compare the results with another established implementation called Specfem3D (D. Komatitsch et al. 2012).

We run a simulation using Specfem3D with the parameters described in Chapter 5.1 and compare it with the result of our implementation. The results shown in Figure 13 show that the results are nearly exactly the same. The error when magnified 10000 times looks like white noise and is probably caused by floating point arithmetic which has limited precision and is also not associative.

5.3 Wavefields in a non-uniform mesh

We have verified that the implementation is correct by comparing it to analytical solution and to another, well-established implementation. However, it would still be interesting to see the wavefields on the whole mesh, not just in single points, in order to verify the correctness using our intuition and knowledge of how a propagating wave should look like. It might seem a little unscientific but it allows us see that the results make sense on a larger scale.

We use the non-uniform mesh shown in Figure 14 which is a smaller version of the one used previously in Figure 8a which has fewer elements and the area of the top surface has been reduced, creating a pyramid-like mesh. The reason for choosing a different mesh is that we want to show that the implementation works for non-uniform meshes as well.

We place a Ricker wavelet force source in the z-direction in the centre of the mesh $\mathbf{x}_{source} = (200\,\text{km}, 200\,\text{km}, 200\,\text{km})$ with a central frequency of $f_0 = 1/50\,\text{Hz}$ and offset $t_0 = 60\,\text{s}$. To honour the CFL-stability condition we choose a timestep $\Delta t = 0.2\,\text{s}$. In order to see reflections

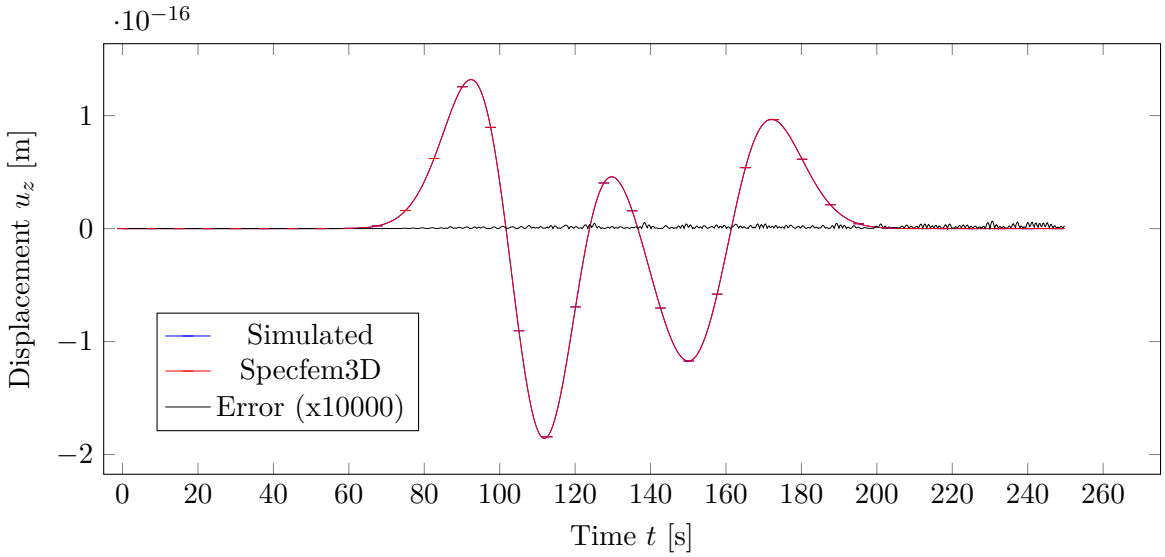


Figure 13: Comparison with Specfem3D. The simulation is the same as the one in Figure 10 which shows that both this and Specfem3D agree with the analytical solution. Both simulations give the exact same answer, save for very small differences which we ascribe to properties of floating point arithmetic such as non-associativity and limited precision.

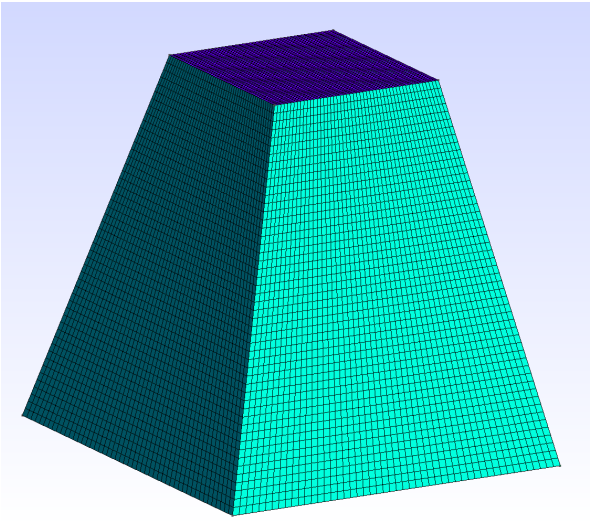


Figure 14: A non-uniform mesh where all elements are not the same size. The size of the bottom of the mesh is 400 km × 400 km while the size of the top is 200 km × 200 km and the height is 400 km. This means that the elements get narrower higher up so they are not all the same size. There are 50 elements in each direction for a total of $50^3 = 125000$ elements. The elements at the top are the smallest with a size of approximately 4 km × 4 km × 8 km.

Number of GPUs	Total execution time	Overhead
1	13m42s	0%
2	13m47s	3%
4	13m55s	4%
8	14m	5%
16	15m37	14%

Table 1: The total execution time of the tests used to make Figure 16. Over the course of 1000 time steps, the 15% increase in execution times per timestep becomes two minutes longer than what would be the ideal time, if there were perfect scaling.

and surface waves we run a longer simulation than previously with $n_t = 1450$ timesteps, resulting in 290 s total simulated time.

In Figure 15 we see a few visualisations of the wavefields in the mesh from Figure 14 at different times. We can see the waves propagate out from the source location, hit the boundaries, reflect and become surface waves.

5.4 Scaling behaviour

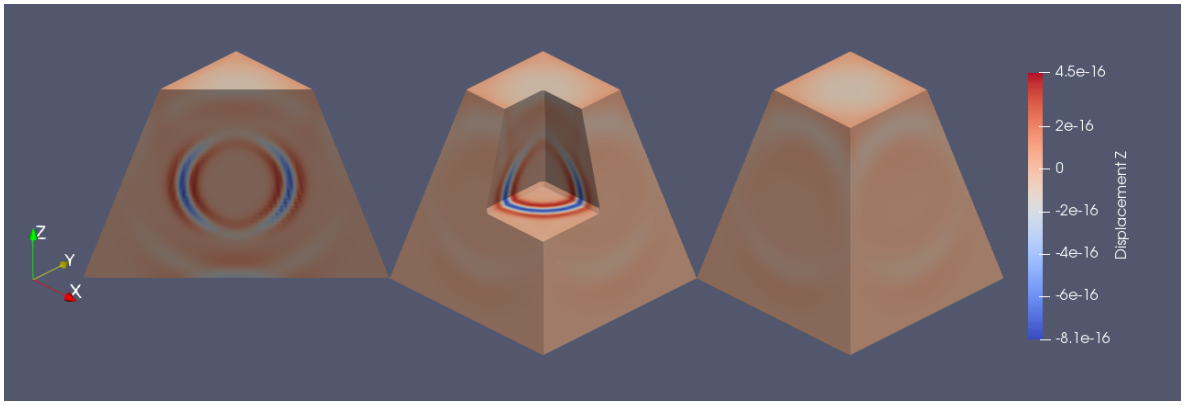
All simulations were run on the EPIC cluster at Norwegian University of Science and Technology(NTNU) in Trondheim, Norway. The cluster boasts a number of nodes each containing two Nvidia P100 GPUs which has a total of 56 streaming multiprocessors and 16GB of memory (NVIDIA 2018[c]).

Since the implementation is intended for use in large-scale simulations utilising multiple GPUs it is interesting to see how the execution time is affected by the number of GPUs used, when each GPU is using all of its available memory. Ideally, adding more GPUs does not affect the execution time since each GPU is working in parallel using the same amount of memory but in reality there will be some overhead caused by communication between the GPUs. Ideally this overhead is so small that the execution time is not impacted when adding more GPUs even though more data is processed. This would mean that solving larger systems is simply a matter of adding more hardware, since the implementation will support an arbitrary number of GPUs.

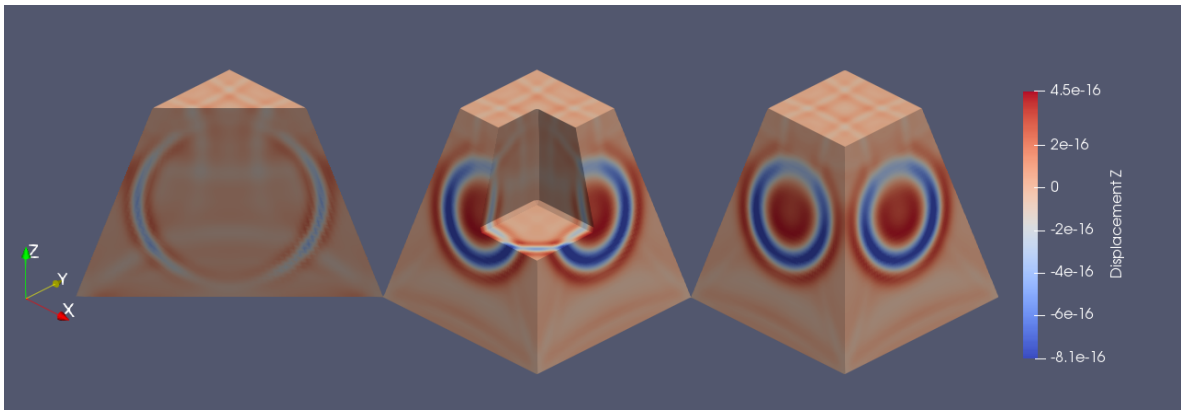
In figure 16 we see the average execution time per time step for a different number of GPUs. We observe that the execution time increases when the number of GPUs is increased due to the overhead introduced by communication between the GPUs when assembling the system. The total execution time and overhead for the tests are given in Table 1.

6 Discussion

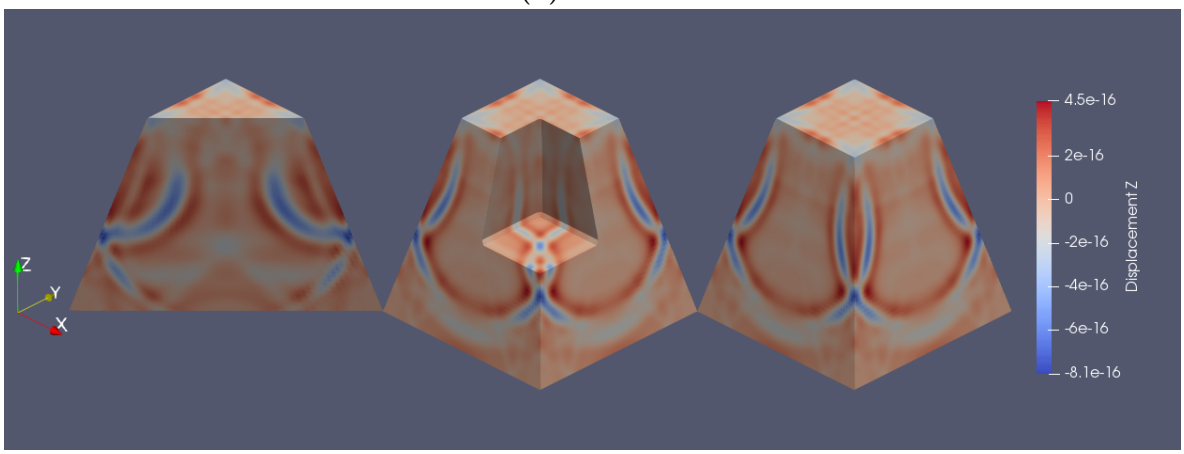
The aim of this project was to make an application which can simulate seismic wave propagation in elastic media. In order to verify the correctness of the implementation we compared



(a) T=160 s



(b) T=230 s



(c) T=290 s

Figure 15: Wavefields in the non-uniform mesh in Figure 14 after 160 seconds (a), 230 seconds (b) and 290 seconds (c). The leftmost figure shows a slice through the centre of the pyramid to show the waves as they propagate out from the centre of the mesh. In the middle figure we can see how the waves propagate differently in the z-direction compared to the x and y-direction. The figure to the right shows the surface waves.

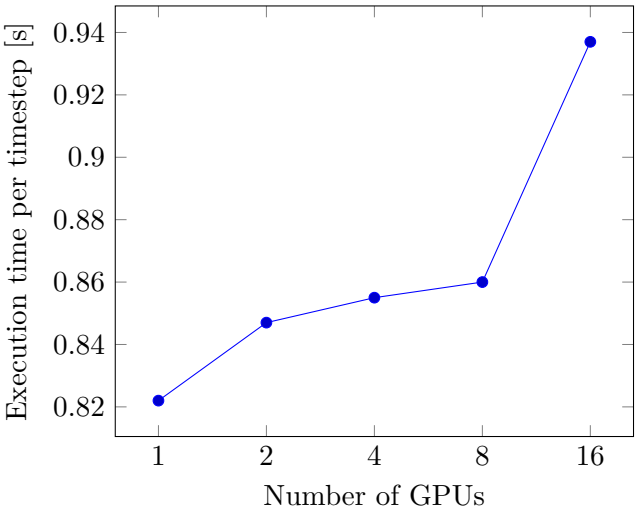


Figure 16: Average execution time of the timestep of the algorithm on between 1 and 16 GPUs. Each GPU performs the same amount of computations, thus the difference in execution time is caused by communication.

it to the analytical solution in an unbounded, isotropic and homogeneous medium. The results in Figure 10 show that the simulated wave closely resembles the analytical solution. We performed the same test with a mesh divided into 4 partitions running on 4 GPUs which gave the same results shown in Figure 12. One would rightly note that the placement of the receiver (100 km offset from the source along the z-axis) will not record the S-wave, only the P-wave, we can however clearly see the S-wave and P-wave in the wavefields in Chapter 5.3 which are discussed below. It would have been interesting to also test the implementation on partitions of different shapes, but it only supports partitioning into slices as shown in Figure 8b.

The error in the simulation could be caused by a bug in the implementation or it could be a result of approximations and discretization used in the spectral element method. To eliminate the latter we performed the same comparison against another implementation of the same method. The results in Figure 13 show a very small error, indicating that the implementations give exactly the same results, save for errors caused by the non-associativity and limited precision of floating point arithmetic. In order to support the conclusion that the error in Figure 10 is indeed caused by the numerical method, we performed the same test with sources with different frequency content. If discretization was causing the error we would see in increase in error with frequency because of more aliasing of the high frequency components. The results seem to support this, so we are inclined to believe that any error is caused by the numerical method and not by a bug in the implementation. Discretization in time also causes aliasing, but in this case the timestep is sufficiently small at 0.2 s giving a Nyquist frequency of $F_s = 5\,\text{Hz}$ such that a Ricker wave with dominant frequency $f_0 = 0.02\,\text{Hz}$ causes much less aliasing than the discretization in space.

Note that the mesh we used from Figure 8a is completely regular so all the elements are cubes of the same size. It was necessary to impose this limitation in order to compare the result with Specfem because of difficulties faced when attempting to create more complicated

meshes with Specfem. We can at least confirm that the implementation is correct for a simple regular, structured mesh. We would however also like to know that the implementation is correct for other types of meshes too.

In order to verify that the implementation also works meshes that are non-uniform we performed the same test with the mesh shown in Figure 14. Instead of comparing the result to the analytical solution, we plot the wavefields on different cross-sections of the mesh at different times as shown in Figure 15. The wave springs from a point force source in the centre of the mesh and we clearly see the difference between the P-wave propagating rapidly in the z-direction and the S-wave in the x and y directions moving more slowly. As expected we see reflections of the waves as they hit the surface and create surface waves. While this mesh is also rather regular, it is at least non-uniform, meaning that the elements are not all the same size. We have not confirmed that the implementation works in the general case of an unstructured mesh where any point may be shared by any number of elements and the elements may be of arbitrary (hexahedral) shape. While this case should work, we did not have the time to test it. Furthermore a full implementation should support meshes representing media with varying wave speeds v_s, v_p and density ρ, but this has also not been tested.

Another important factor of the implementation is how well it scales with larger problems. As we see in Figure 16 and Table 1 the execution time does increase when more GPUs are added up to 14% between 1 and 16 GPUs, and it will probably be larger for more GPUs which could be significant for very long simulations. This is certainly caused by the communication between the GPUs, which must exchange information about nodes in the interfaces between the mesh partitions. This means that it can be eliminated by overlapping communication and computation, by first computing the nodes that lie in the interfaces and starting the communication which can run in parallel with the computation of the remaining nodes. This has previously been demonstrated to work well by Dimitri Komatitsch, Erlebacher, et al. 2010. As mentioned in Chapter 4 the implementation could be more efficient, and we need only take a look at the source code of Specfem (D. Komatitsch et al. 2012) in order to see how. However, as mentioned in the introduction, we were not attempting to create the fastest possible implementation though optimisation will certainly be happening in the future.

7 Conclusion

Because of the significant computational complexity of simulating seismic wave propagation it is interesting to explore how to use existing hardware to accelerate computations. Because the SEM allows the elastic wave equation to be easily solved in parallel, the parallel architecture of a GPU arises as a competitor to the traditional CPU implementation. Furthermore it is possible to use multiple GPUs in a cluster, just as it is with CPUs, by using MPI. This method has been used successfully before by Dimitri Komatitsch, Erlebacher, et al. 2010 (Specfem) and can lead to significant speedups over CPU implementations. We have done the same in an attempt to create an alternative to Specfem.

We validated the implementation by comparing the results of a simulation first with the analytical solution and then with Specfem, both comparisons indicating that the results were correct. We also visualized the wavefields, confirming that the simulated waves propagate the

same way a real wave would. Furthermore, we investigated how the implementation scales when using more GPUs which revealed that there is an overhead caused by the communication between the GPUs, resulting in a 14% increase in execution time when using 16 GPUs. Speedup can still be acquired, but will suffer diminishing returns if the number of GPUs becomes too large.

The main limitation on the implementation is that it only works for simple meshes. However we have shown that it is correct in those cases and extension to more general meshes is already supported by the implementation, it just needs to be tested. The main part that which is not implemented, is support for general partitioning of a mesh. However, the results indicate that this implementation is correct in the simple cases which we tested. While these cases may not be interesting for real-world problems, they show that what we have is the groundwork necessary for a fully functional implementation.

8 Future work

While the implementation works for simple meshes it certainly leaves a few things to be desired. The most important work to be done is to extend the implementation to work with more general hexahedral mesh. So far it only works with a cubic mesh which can be divided into simple partitions which share only two interfaces with other partitions and where each interface has the same number of elements. The plan is to make the implementation compatible with the .msh v4 file format used by the GMSH software (*Gmsh homepage* 2019). GMSH is capable of creating and partitioning meshes and exporting the partitions into separate files which could work well with an implementation intended to run on a cluster.

Once the implementation supports a more general type of mesh and partition, it is important to compare it to other existing implementations, to check both numerical accuracy and performance. It is especially interesting to see if it performs better than existing CPU implementation, to prove that GPUs are indeed a viable option. It is not obvious how to compare CPU and GPU performance on clusters, since it depends on the amount of hardware available. It might instead be interesting to compare based on cost or power consumption, instead of just speedup.

Performance improvements can be made in several areas. One important improvement is to eliminate the overhead caused by communication, which has been done by Dimitri Komatitsch, Erlebacher, et al. 2010. This is important because it will allow implementation to scale indefinitely so that, ideally, the only limitation is the hardware which it runs on. This is desirable because it might be simpler to solve a larger problem by adding more hardware instead of writing better software. Furthermore, optimizations can be made to the current implementation by make better use of the GPU memory by using shared memory and coalescing memory access. An interesting application of a high-performance implementation is full waveform inversion which requires an effecient solution to the wave equation according to Raknes and Arntsen 2017.

If the implementation were to scale well, it would be interesting to see how it runs on more, but cheaper GPUs. We have tested the implementation on a rather small number (16) P100

GPUs. This is very expensive hardware so it would be interesting to see how the cost and performance compares to a larger number of cheaper GPUs, such as the Nvidia TITAN X.

References

Aki, K. and P.G. Richards (2002). *Quantitative Seismology*. Geology (University Science Books).: Seismology. University Science Books, p. 72. ISBN: 9780935702965.

Fichtner, Andreas (2011). *Full Seismic Waveform Modelling and Inversion*. Springer-Verlag Berlin Heidelberg. ISBN: 978-3-642-15807-0.

Gmsh homepage (2019). URL: http://gmsh.info/ (visited on 06/04/2019).

Harris, Mark (2018). *An Even Easier Introduction to CUDA*. URL: https://devblogs.nvidia.com/even-easier-introduction-cuda/ (visited on 12/09/2018).

Hennessy, John L and David A Patterson (2011). *Computer Architecture: A Quantitative Approach*. United States: Morgan Kaufmann Publishers Inc. ISBN: 9780123838728.

Ikelle, Luc T. and Lasse Amundsen (2005). *Introduction to Petroleum Seismology*. SEG (Society of Exploration Geophysicists). ISBN: 978-1-5231-1612-6.

Intel Xeon Processor E5-2630 v4 Datasheet (2018). URL: https://ark.intel.com/products/92981 (visited on 12/09/2018).

Komatitsch, Dimitri, Gordon Erlebacher, et al. (2010). "High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster". In: *Journal of Computational Physics* 229.20, pp. 7692–7714. ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2010.06.024. URL: http://www.sciencedirect.com/science/article/pii/S0021999110003396.

Komatitsch, Dimitri, Seiji Tsuboi, et al. (2005). "The spectral-element method in seismology". In: *GEOPHYSICAL MONOGRAPH-AMERICAN GEOPHYSICAL UNION* 157, p. 205.

Komatitsch, D. et al. (2012). *SPECFEM3D Cartesian v2.0.2 [software]*. Computational Infrastructure for Geodynamics. URL: https://geodynamics.org/cig/software/specfem3d/.

Mavko, Gary, Tapan Mukerji, and Jack Dvorkin (2009). *Seismic wave propagation*. 2nd ed. Cambridge University Press. DOI: 10.1017/CBO9780511626753.004.

MPI: A Message-Passing Interface Standard (2019). URL: https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf (visited on 04/14/2019).

NVIDIA (2018[a]). *CUDA C Programming Guide*. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/ (visited on 12/09/2018).

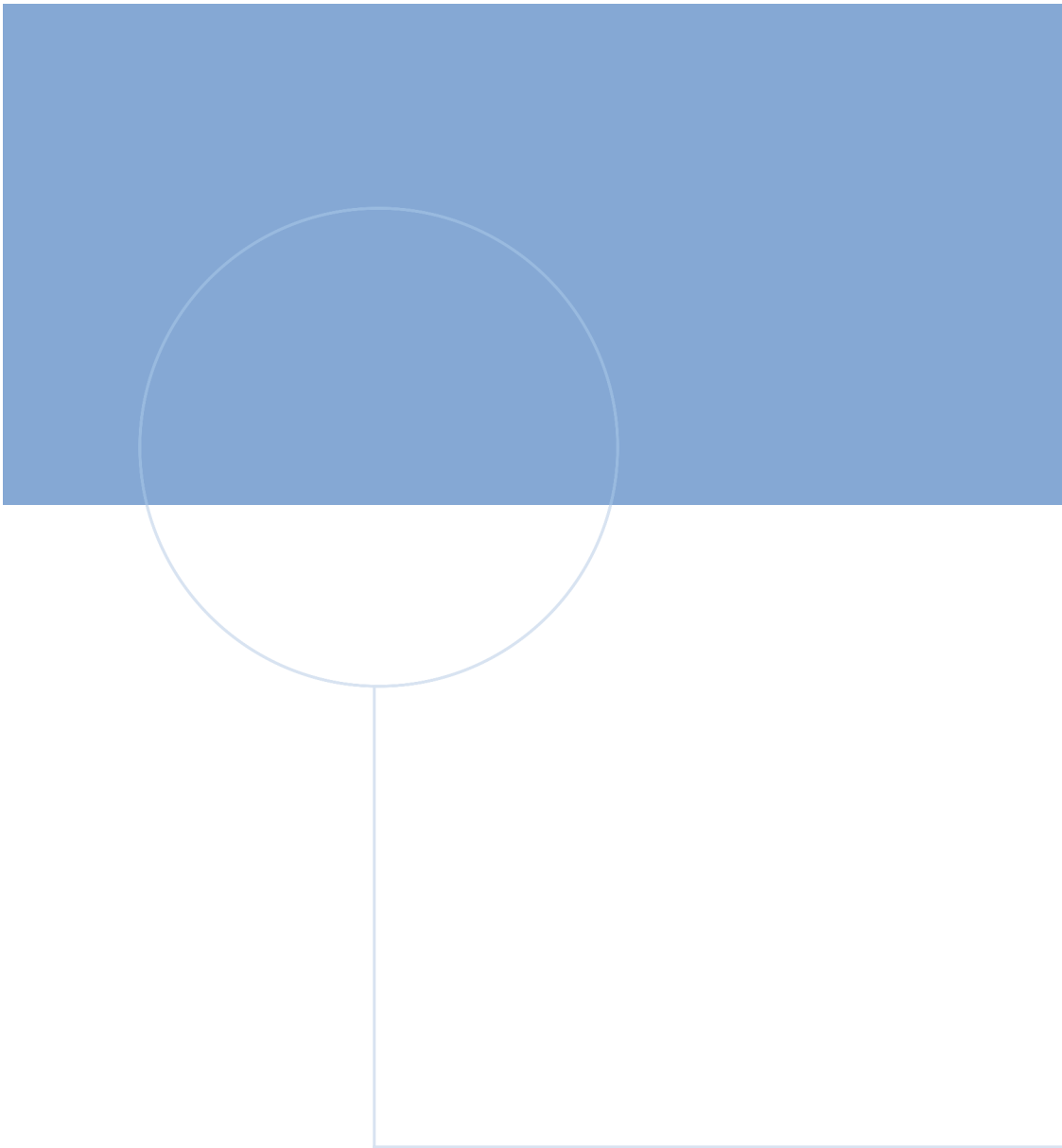
– (2018[b]). *NVIDIA Tesla P100 Whitepaper*. URL: https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf (visited on 12/09/2018).

– (2018[c]). *P100 Datasheet*. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-p100/pdf/nvidia-tesla-p100-datasheet.pdf (visited on 12/09/2018).

Quarteroni, Alfio (2017). *Numerical Models for Differential Problems*. eng. 3rd ed. 2017. Vol. 16. MS&A, Modeling, Simulation and Applications. Cham. ISBN: 3-319-49316-7.

Raknes, Espen Birger and Børge Arntsen (2017). "Challenges and solutions for performing 3D time-domain elastic full-waveform inversion". In: *The leading edge* 36.

S. Tsuboi J. Tromp, C. Ji and D. Komatitsch (2003). "A 14.6 billion degrees of freedom, 5 teraflops, 2.5 terabyte earthquake simulation on the Earth Simulator". In: *SC Conference(SC), Phoenix, Arizona*, p. 4. DOI: 10.1109/SC.2003.10023.



NTNU
Norwegian University of
Science and Technology