MODELING FOR INVERSION IN EXPLORATION GEOPHYSICS

A Dissertation
Presented to
The Academic Faculty

By

Mathias Louboutin

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of CSE in the College of Computing

Georgia Institute of Technology

May2020

MODELING FOR INVERSION IN EXPLORATION GEOPHYSICS

Approved by:

Dr. Felix J. Herrmann, Advisor
School Computational Science and Engineering
Georgia Institute of Technology

Dr. Umit Catalyurek
School Computational Science and Engineering
Georgia Institute of Technology

Dr. Edmond Chow
School Computational Science and Engineering
Georgia Institute of Technology

Dr. Tobin Isaac
School of Computer Science
Georgia Institute of Technology

Dr. Zhigang Peng
School of Earth and Atmospheric Sciences
Georgia Institute of Technology

Date Approved: February 26, 2020

ACKNOWLEDGEMENTS

Before anything else, I would like to thank my supervisor Dr Felix J. Herrmann for giving me the opportunity to work with him. Thanks to his leadership I had the opportunity to work and scientifically challenging problems in a collaborative and motivating atmosphere.

I would also like to thank Professor Gerard Gorman at Imperial college. And large part of my research was kick-started by a visit at Imperial College and Dr. Gorman's support and guidance made me achieve my research objective.

I would like to thank Professor Umit Catalyurek, Professor Edmond Chow, Professor Tobin Isaac and Professor Zhigang Peng for agreeing to be on my Ph.D. committee at Georgia Tech, for reviewing my thesis, for making time for my proposal and defense and for your valuable input on my work. I would also like to thank my former Ph.D. committee at the University of British Columbia, Professor Michael Bostock, Professor Ozgur Yilmaz who oversaw me during the first years of my Ph.D. and throughout my candidacy.

I am very thankful to the collaborators I had, including my coworkers at SLIM and at Imperial college. I would specifically like to thank Philipp Witte at SLIM, with whom I collaborated closely over the years and who greatly helped me to improve my writing. I would also like to thank Dr F. Luporini who has been my greatest motivation to be a better programmer. Our collaboration allowed Devito to grow to what it is today.

Warm thanks to Henryk Modzelewski at UBC and to our former administrative assistant Miranda Joyce that helped me navigate graduate studies and welcomed me and helped be part of the SLIM team.

May thanks to Sverre Brandsberg-Dahl and his colleagues at Microsoft (Alexander Morris, Steve Roach, Fred Park) who made our project in the Cloud feasible and provided us support and visibility.

I would like to acknowledge the sponsors of SLIM and SINDAB and Georgia Institute of Technology that provided funding form y research over the course fo my PhD.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Seismic inversion, and more generally geophysical exploration, aims at better understanding the earth's subsurface, which is one of today's most important challenges. Firstly, it contains natural resources that are critical to our technologies such as water, minerals and oil and gas. Secondly, monitoring the subsurface in the context of CO2 sequestration, earthquake detection and global seismology are of major interests with regard to safety and the environment hazards. However, the technologies to monitor the subsurface or find resources are scientifically extremely challenging. Seismic inversion can be formulated as a mathematical optimization problem that minimizes the difference between field recorded data and numerically modeled synthetic data. The process of solving this optimization problem then requires to numerically model, thousands of times, wave-propagation in large three-dimensional representations of part of the earth subsurface. The mathematical and computational complexity of this problem, therefore, calls for software design that abstracts these requirements and facilitates algorithm and software development.

My thesis addresses some of the challenges that arise from these problems; mainly the computational cost and access to the right software for research and development. In the first part, I will discuss a performance metric that improves the current runtime-only benchmarks in exploration geophysics. This metric, the roofline model, first provides insight at the hardware level of the performance of a given implementation relative to the maximum achievable performance. Second, this study demonstrates that the choice of numerical discretization has a major impact on the achievable performance depending on the hardware at hand and shows that a flexible framework with respect to the discretization parameters is necessary. In the second part, I will introduce and describe Devito, a symbolic finite-difference DSL that provides a high-level interface to the definition of partial differential equations (PDE) such as the wave equation. Devito, from the symbolic definition of PDEs, then generates and compiles highly optimized C code on-the-fly to compute the solution

of the PDE. The combination of the high-level abstractions and the just-in-time compiler enable research for geophysical exploration and PDE-constrainted optimization based on the paradigm of separation of concerns. This allows researchers to concentrate on their respective field of study while having access to computationally performant solvers with a flexible and easy to use interface to successfully implement complex representations of the physics. The second part of my thesis will be split into two sub-parts; first describing the symbolic application programming interface (API), before describing and benchmarking the just-in-time compiler. I will end my thesis with concluding remarks, the latest developments and a brief description of projects that were enabled by Devito.

CHAPTER 1
INTRODUCTION

1.1 Introduction and Background

Understanding the physics of our surrounding has driven science and technology for a long time and while some part of the earth is well know and understood, its subsurface is mostly unknown. The subsurface is one of today's most important challenges. Firstly, it contains natural resources that are critical to our technologies such as water, minerals, gas and oil. Secondly monitoring of the subsurface such as CO2 sequestration [1, 2], earthquake monitoring and prediction [3] and global seismology [4, 5] are major problems for safety and environments. However, the technologies to monitor or find these resources are scientifically extremely challenging. My thesis addresses some of the challenges that arise from these problems, mainly the computational cost and the access to the right software for research and development.

Seismic imaging estimates subsurface parameters such as the velocity of sound waves or the rock's density from pressure measurements recorded at the surface of the earth or the ocean. This parameter estimation problem can be formulated as a mathematical optimization problem that is usually the minimization of a data misfit between the field recorded data, and numerically generated synthetic data [6]. The optimization problem and its minimization algorithm therefore involves solving the wave-equation, a partial differential equation (PDE), either in the frequency domain via iterative solvers [7, 8, 9, 10] or in the time domain via time-steppers [11, 12, 13]. In practice, the type of seismic sources used in the field to record the data are bandwidth-limited, and, unlike other fields such as Medical Imaging, seismic measurement can only be recorded at the surface and/or at very limited number of well locations. These two physical constraints render the mathematical data fitting problem ill-posed and non-convex in most cases. As a consequence, extensive research has been directed towards finding algorithms to solve seismic inversion and imaging problems with good or less good success depending on the situation and the quality of available datasets, e.g. [6, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]. For these reasons, this is a field that is still in active development.

While the mathematical difficulty to solve the data-fitting optimization is complex, one of its main requirement is to have access to computationally efficient wave-equation solvers. The problem formulation requires solutions of thousands of wave-equations in order to achieve an acceptable (inversion) results for large scale domains. A standard seismic problem involves an unknown (discrete representation of the subsurface) typically with up to a billion unknowns:

$$\underset{x}{\text{minimize}} \quad \sum_{i=1}^{\mathcal{O}(1e4)} f_i(x), \quad x \in \mathbb{R}^{1e9}. \tag{1.1}$$

However, this computational complexity and cost cannot become a burden to mathematicians or geophysicist whose domains of expertise and interest are optimization algorithms, subsurface parameters estimation, and imaging and not high-performance computing to carry out the wave simulations. For these reasons, well designed software including an interface to PDE solvers are necessary to provide a workflow that allows separation of concerns and rapid innovation. In addition to the computational demand of having to solve the wave-equation many times for thousands of times steps, gradient calculations of inversion algorithms often require solutions of the adjoint equation. As I will explain below, the derivation of the adjoint wave equation itself including its interaction with the forward wavefield is challenging since it is generally impossible to store this state variable in memory. In practice, storage of this variable may require terabytes of memory because of our model size and number of time steps ($\mathcal{O}(1e9)$ grid points and $\mathcal{O}(1e4)$ time-steps). This model size calls for advanced methods such as optimal checkpointing [26, 27, 28, 29] or compression methods [30].It is clear that this type of additional requirements add complexity to the wave-equation solver and requires special care in the design of the interface so that these additional requirements can be accommodated and implemented by domain specialists.

The concept of separation of concerns in computational physics has led to numerous projects that motivated my core contribution: Devito [13]. Devito is a finite-differences domain-specific language (DSL) that provides a symbolic interface to define partial differential equations (PDE) and implements its own just-int-time compiler. High-level interfaces such as symbolic DSLs and just-in-time compilers are gaining attention and earlier work in computational fluid dynamics (CFD) provided a strong basis and justification for it. The need for a high-level interface for CFD led to the design of symbolic DSLs such as FEniCS [31] or Firedrake [32] that provide a symbolic interface to define the weak form of variational problems. Both of these two frameworks implement the same DSL known as UFL [33]. The success of these DSL laid the ground for Devito's high-level user interface. Moreover, Devito also relies on just-in-time code generation and compilation to provide state-of-the-art computational performance. A thorough overview of the literature is detailed in each of the Chapters constituting my thesis.

The main aim of my thesis is to find an answer to the extreme computational challenges of seismic inversion while providing an interface that enables rapid code development, with a carefully designed DSL, and performance tools such as automatic roofline performance benchmark [34]. My introduction is organized as follows. First, I introduce the seismic inversion problem in more detail including its mathematical formulation. Next, I provide a motivating example that highlights the complexities that arise when dealing with realistic physical models that describe wave motion in the Earth subsurface. After discussing this example, I will define the objectives of my thesis and detail my contributions and conclude with an outline of the three main chapters of my thesis.

1.2 Wave-equation based geophysical exploration

Wave-equation based seismic inversion, including Full-Waveform Inversion (FWI), aims to estimate one or more physical properties of the Earth subsurface by minimizing the misfit between multiple observed shot records (field recorded data) of a seismic survey and their numerically modeled counterparts. To model these predicted shot records, I solve the wave-equation for each individual source location. These simulated shot records themselves depend on the parametrization $\mathbf{m}$ of the wave propagator in terms of physical rock parameters that include the compressional wavespeed, density of mass and potentially other parameters. During seismic inversion, we are interested in obtaining estimates for gridded spatial distributions of these parameters.

In its most basic form, the misfit function used to compare measured and predicted data in the $\ell_2 - norm$ reads [35, 6]:

$$\underset{\mathbf{m}}{\text{minimize}} \quad f(\mathbf{m}) = \sum_{i=1}^{n_s} \frac{1}{2} \left\| \mathbf{d}_i^{\text{pred}}(\mathbf{m}, \mathbf{q}_i) - \mathbf{d}_i^{\text{obs}} \right\|_2^2, \tag{1.2}$$

where $f(\mathbf{m})$ is the objective function as a function of the discretized model parameters (slowness squared with slowness = velocity^{-1}) collected in the vector $\mathbf{m}$. The index i runs over the total number of shots n_s. The predicted data $\mathbf{d}_i^{\text{pred}}$ is represented by the solution of the wave equation as follows:

$$\begin{aligned} \mathbf{d}_i^{\text{pred}}(\mathbf{m}, \mathbf{q}_i) &= \mathbf{P}_r \mathbf{u}(\mathbf{m}) \\ \mathbf{u}(\mathbf{m}) &= \mathbf{A}(\mathbf{m})^{-1} \mathbf{P}_s^\top \mathbf{q}_i. \end{aligned} \tag{1.3}$$

In this expression, the matrix $\mathbf{A}(\mathbf{m})$ represents the discretized wave-equation parameterized by the unknown model vector $\mathbf{m}$. The vector $\mathbf{q}_i$ is the time-dependent source distribution for the ith shot record. This sourcetime function is injected into the grid of wave equation solver by the adjoint (denoted by the $\cdot^\top$ symbol) of the restriction operator $\mathbf{P}_s$.

This latter operator restricts the wavefield to the source location. After applying the inverse of the wave operator, we simulate observed data by restricting the wavefield to locations of the receivers via the restriction operator $\mathbf{P}_r$.

In this thesis, I concentrate on wave simulations with time-domain finite differences for the following reasons. First, I am interested in inversion in highly heterogeneous media. While mesh-based methods [36, 37, 38, 39] may be more accurate, including a mesh may make the inversion more complicated as this mesh needs to be updated during the iterative inversion. And, secondly, (spectral) finite element methods often require expensive implicit solvers, which rapidly becomes too expensive for the large models of interest in this thesis. I will motivate this choice more in each chapter in relation to the computational performance and the accuracy of the wave-equation solver for large scale seismic inverse problems.

Optimization problems of the form listed in Equation 1.2 are known as non-linear parametric least-squares problems, since the predicted data simulated with the forward modeling propagator depends nonlinearly on the unknown parameters $\mathbf{m}$. The aim is to minimize the objective function with respect to the model parameter $\mathbf{m}$. Because the dimensionality of $\mathbf{m}$ is high, we have to rely on local derivative based optimization methods to minimize Equation 1.2. We obtain the gradient by applying the chain rule and taking the partial derivative of the inverse wave-equation $\mathbf{A}(\mathbf{m})^{-1}$ with respect to$\mathbf{m}$. This yields the following expression for the gradient [40, 41, 17, 7]:

$$\nabla f(\mathbf{m}) = \mathbf{J}^\top(\mathbf{m})\Big[\mathbf{P}_r\mathbf{A}(\mathbf{m})^{-1}\mathbf{P}_s^\top\mathbf{q} - \mathbf{d}_{\text{obs}}\Big], \tag{1.4}$$

where $\mathbf{J}(\mathbf{m})$ is the Jacobian:

$$\begin{aligned} \mathbf{J}(\mathbf{m}) &= \frac{\partial}{\partial\mathbf{m}}\Bigg[\mathbf{P}_r\mathbf{A}(\mathbf{m})^{-1}\mathbf{P}_s^\top\mathbf{q}\Bigg] \\ &= \mathbf{P}_r\mathbf{A}(\mathbf{m})^{-1}\frac{\partial^2\mathbf{u}(\mathbf{m})}{\partial t^2} \end{aligned} \tag{1.5}$$

This gradient of the objective function in Equation 1.2 can be rewritten as

$$\nabla f(\mathbf{m}) = -\sum_{t=1}^{n_t} \mathbf{u}[t] \odot \ddot{\mathbf{v}}[t]. \tag{1.6}$$

In this expression for the gradient, the sum over the element-wise product (denoted by the symbol $\odot$) runs over the number of computational time-steps n_t. The vector $\ddot{\mathbf{v}}$ denotes the second time derivative of the adjoint wavefield $\mathbf{v}$. Physically, this sum corresponds to the zero-lag correlation between the forward and second time derivative of adjoint wavefield. While the above expression for the gradient looks simple, there is an important complication that stems from the fact that the adjoint wave equation is solved backwards in time. This means that one can not simply compute the above sum. Moreover, the need for an adjoint wave equation may complicate things in situations where the wave equation at hand is for physical and mathematical reasons, not self adjoint. The latter situation arises when dealing with unphysical but numerical feasible discretizations of wave motion in anisotropic media where the wavespeed depends on the propagation direction.

To be more specific, I will illustrate the main challenges of seismic imaging and why Devito is the right tool to tackle these challenges that include the fact that

- wave equations associated with realistic representations of the physics, such as anisotropy, are mathematically complex and computationally extremely demanding (Add footnote and say how many floats per grid point for TTI) [42, 43] (Chapter 2 and 4). Therefore, the implementation of anisotropic wave propagators in low level languages such as C or FORTRAN can take large amounts of human-time and typically results in monolithic codebases that are inflexible and difficult to maintain. Because Devito provides a high-level symbolic interface, it allows for automatic code generation that are computationally performant without relying on low-level manual coding. Instead, Devito derives its performance from modern state-of-the-art compiler technology and design that takes abstract symbolic expressions for wave equations as input and produces highly optimized low-level C code as output.

- derivations of realistic representations for the physics can also lead to non self-adjoint systems [44]. Unfortunately, this lack of being self-adjoint is often overlooked due to the sheer amount of work it takes to implement the forward wave-equation. Instead, systems are erroneously assumed to be self-adjoint. I will demonstrate what the consequences are of this wrong assumption regarding wave propagation itself and inversion, which relies on computing the gradient.

Throughout my thesis, I demonstrate that the above complexities can be handled by the right tools; modern automatic code generation in combination with proper abstractions in the form of a domain-specific language (DSL). Devito combines these two aspects into a finite-difference DSL with its own symbolic compiler and code generation. As I mentioned above, Devito implements a symbolic finite-difference DSL based on `sympy` [45] that allow to define PDEs in terms of mathematical expressions. This high level interface is designed to drastically shorten the turn around time for the implementation of a new wave equation to weeks (or days) rather than months, which are usually necessary for a by-hand implementation. The definition of the equation at a mathematical level also negates the complexity associated with the implementation of code associated with the adjoint PDE. Second, because Devito uses state-of-the art code generation tools [46, 47, 48, 49], the computational performance matches, and in some cases even surpasses, hand-tuned codes. With these two feature satisfied, Devito is a flexible and complete DSL that offers the right high-level interface for rapid development in the context of complex wave physics, and provides the computational performance needed for large scale inverse problems through its code generation and just-in-time compiler and ability to properly handle adjoints.

1.3 Motivational example[1]

As stated before, my main motivation is wave-equation based seismic inversion. In this section, I highlight why high-level interfaces are extremely important for easy and rapid development of simulation and inversion codes in exploration geophysics. The example I choose is an anisotropic representation of the physics called Transverse Tilted Isotropic (TTI) [50]. This representation for wave motion is one of the most widely used in exploration geophysics since it captures the leading order kinematics and dynamics of acoustic wave motion in highly heterogeneous elastic media where the medium properties vary more rapidly in the direction perpendicular to sedimentary strata [**baysal1983**, 51, 52, 53, 54, 55, 56, 57, 58, 44, 59, 60, 61, 62, 63, 64]. The TTI wave equation is an acoustic, low dimensional (4 parameters, 2 wavefields) simplification of the 21 parameter and 12 wavefields tensorial equations of motions [65]. This simplified representation is parametrized by the Thomsen parameters $\epsilon(x), \delta(x)$ that relate to the global (many wavelength propagation) difference in propagation speed in the vertical and horizontal directions, and the tilt and azimuth angles $\theta(x), \phi(x)$ that define the rotation of the vertical and horizontal axis around the cartesian directions.

However, unlike the scalar isotropic acoustic wave-equation itself, the TTI wave equation is extremely computationally costly to solve and it is also not self-adjoint. The TTI wave-equation reads as follows:

$$\begin{aligned} m(x)\frac{d^2p(x,t)}{dt^2} - (1+2\epsilon(x))H_{\bar{x}\bar{y}}p(x,t) - \sqrt{1+2\delta(x)}\, H_{\bar{z}}r(x,t) = q, \\ m(x)\frac{d^2r(x,t)}{dt^2} - \sqrt{1+2\delta(x)}\, H_{\bar{x}\bar{y}}p(x,t) - H_{\bar{z}}r(x,t) = q, \end{aligned} \tag{1.7}$$

where $p(x,t)$ and $r(x,t)$ are the two component of the anisotropic wavefield and $H_{\bar{z}}$ and $H_{\bar{x}\bar{y}} = G_{\bar{x}\bar{x}} + G_{\bar{y}\bar{y}}$ are the rotated second order differential operators that depend on the tilt,

[1]This introductory example is a detailed extension to the work I presented at the SEG annual conference in 2018 [44].

azimuth $(\theta(x), \phi(x))$ and the conventional (isotropic) cartesian spatial derivatives $\frac{d}{dx}$, $\frac{d}{dy}$ and $\frac{d}{dz}$.

After discretization, the TTI wave-equation can be rewritten in a linear algebra form:

$$\mathbf{m} \begin{bmatrix} \frac{\mathrm{d}^2\mathbf{p}}{\mathrm{d}t^2} \\ \frac{\mathrm{d}^2\mathbf{r}}{\mathrm{d}t^2} \end{bmatrix} = \begin{bmatrix} (1+2\epsilon)H_{\bar{x}\bar{y}} & \sqrt{1+2\delta}\, H_{\bar{z}} \\ \sqrt{1+2\delta}\, H_{\bar{x}\bar{y}} & H_{\bar{z}} \end{bmatrix} \begin{bmatrix} \mathbf{p} \\ \mathbf{r} \end{bmatrix} + \mathbf{P}_s^\top \mathbf{q} \tag{1.8}$$

where the bold font represents discretized version of the wavefield and physical parameters. With this expression, we can rewrite the solution of the anisotropic wave equation as the solution of a linear system $\mathbf{u}(\mathbf{m}) = \mathbf{A}(\mathbf{m})^{-1}\mathbf{P}_s^\top$ similarly to Equation 1.3 where in this case $\mathbf{u}(\mathbf{m})$ is a two component vector $(\mathbf{p}(\mathbf{m})^\top, \mathbf{r}(\mathbf{m})^\top)^\top$. Like before, the matrix $\mathbf{P}_s^\top$ injects the source in both wavefield components.

As discussed in [63] and [56], I choose a finite-difference discretization of the three differential operators $H_{\bar{z}}, G_{\bar{x}\bar{x}}, G_{\bar{y}\bar{y}}$ that is self-adjoint to ensure numerical stability. For example, we define $G_{\bar{x}\bar{x}}$ as a function of the discretized tilt θ and azimuth ϕ as:

$$\begin{aligned} G_{\bar{x}\bar{x}} &= D_{\bar{x}}^T D_{\bar{x}} \\ D_{\bar{x}} &= \cos(\theta)\cos(\phi)\frac{\mathrm{d}}{\mathrm{d}x} + \cos(\theta)\sin(\phi)\frac{\mathrm{d}}{\mathrm{d}y} - \sin(\theta)\frac{\mathrm{d}}{\mathrm{d}z}. \end{aligned} \tag{1.9}$$

Because of the very high number of floating-point operations (FLOP) needed per grid point for the weighted rotated Laplacian, this anisotropic wave-equation is extremely challenging to implement. As I show in Chapter 2, an estimate of the computational cost with high-order finite-difference is in the order of thousands of FLOPs per grid point. Consequently, the implementation of a solver for this wave-equation can be time-consuming and can lead to thousands of lines of code and the verification of its result becomes challenging as any small error is effectively untrackable and any change to the finite-difference scheme or to the time-stepper is nearly impossible to achieve without substantial re-coding. Another complication stems from the fact that practitioners of seismic inversion are often geoscientists and not computer scientists/programmers. Unfortunately, this background often either results in poorly written low performant codes or it leads to complications when research codes are handed off to computer scientists who know how to write fast codes but who often miss the necessary geophysical domain knowledge. Neither situation is conducive to addressing the complexities that come with implementing codes based on the latest geophysical insights in geophysics and high-performance computing. Devito with its high level interface and state-of-the art just-in-time compiler addresses these complications by enabling geophysical domain experts to express themselves while offering sufficient flexibility to make the code suitable for industrial applications. Aside from these practical industrial considerations, obtaining correct and numerically stable implementations for the adjoint TTI wave-equation also has proven to be challenging in the past. Add refs. In the next section, we will demonstrate the importance of having correct adjoints and how Devito enables the correct implementation.

1.4 Modeling for inversion

Simulation of wave motion is only one aspect of solving problems in seismology. During wave-equation based imaging, we also need to compute sensitivities (gradient) with respect to the quantities of interest. This imposes additional constraints on the design and implementations of our simulation codes as outlined in [17]. Among several factors, such as fast setup time etc., we will focus mainly on correct and testable implementations for the adjoint wave equation and the gradient (action of the adjoint Jacobian).

1.4.1 Adjoint modeling

While the true physics of tensorial wave motion in elastic is self-adjoint, its numerical implementation on realistic model sizes is unfeasible because it requires a parameterization in terms of 21 spatially varying elastic constants and 12 wavefields [66, 67, 68, 65]. To make the problem computationally feasible, different approximate formulations have been proposed that are computationally feasible but unfortunately often no longer self-adjoint. Consequently, time reversing the solution of the forward problem, which is often common practice, may lead to erroneous results because the time-reversed wavefield is no longer equivalent to the solution of the adjoint equation. While this practice may have a physical justification, it can lead to less accurate or even incorrect results in certain cases.

To illustrate potential pitfalls associated with the use of wrong adjoints, let us consider the above TTI formulation (cf. Equation 1.9) where the spatially varying finite-difference operators G are self-adjoint by construction but the overall system itself is not because these operators are multiplied by terms that contain the spatially varying Thomsen parameters. The provably correct (see adjoint test below) adjoint system of equations corresponding to the TTI forward wave-equations (Equation 1.7) is given by [44]:

$$\mathbf{m}\begin{bmatrix}\frac{\mathrm{d}^2\mathbf{p}_a}{\mathrm{d}t^2}\\ \frac{\mathrm{d}^2\mathbf{r}_a}{\mathrm{d}t^2}\end{bmatrix} = \begin{bmatrix}H_{\bar{x}\bar{y}}(1+2\epsilon) & H_{\bar{x}\bar{y}}\sqrt{1+2\delta}\\ H_{\bar{z}}\sqrt{1+2\delta} & H_{\bar{z}}\end{bmatrix}\begin{bmatrix}\mathbf{p}_a\\ \mathbf{r}_a\end{bmatrix} + \mathbf{P}_s^\top\mathbf{q}_a \tag{1.10}$$

where $\mathbf{p}_a, \mathbf{r}_a$ are the two components of the adjoint anisotropic wavefield. The vector q_a is the adjoint source. For seismic imaging, a form of computing the gradient for a good starting model for the slowness squared $\mathbf{m}$, this adjoint source is given by the residual (difference between the recorded field and numerically modeled data). Compared to the forward TTI wave-equation, the adjoint system differs fundamentally. Contrary to the forward system, which consists of two coupled PDEs made of two rotated and weighted acoustic wave-equations, the adjoint system consists of two fully decoupled equations where the horizontal and vertical derivatives appear separately. To illustrate the differences between solutions yielded by solving the true adjoint wave-equations (cf. Equation 1.7) or by time reversing the solution of the forward equation (cf. Equation 1.10). I include in Figure 1.1 the impulse responses for these two approaches for comparison.

As we can see from these plots, the impulse responses differ significantly in the dynamics (amplitudes) and in some instances even in sign. Because the adjoint wavefield consists

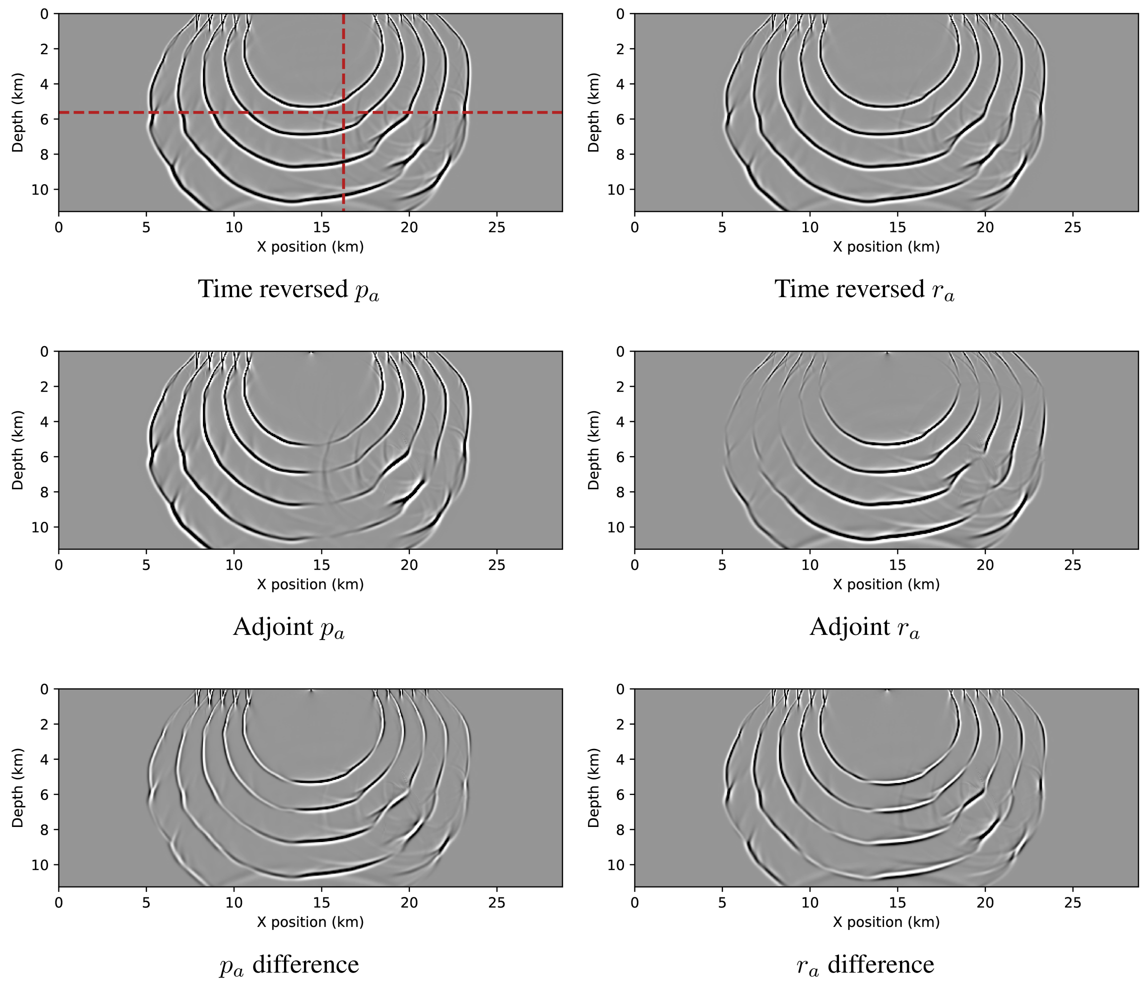


Figure 1.1: Impulse response of the time-reversed and adjoint wave-equation in a TTI medium (BP synthetic model 2007). The top row shows the wavefields of the time-reversed forward wave-equations, while the wavefields of the true adjoint equations are shown in the centre row. The bottom row shows the component-wise difference between the time-reversed and adjoint wavefields. All snapshots and differences are displayed at the same scale.

of purely vertical and horizontal components while the time-reversed wavefields is made of a combination of both the amplitudes yielded by the adjoint and time-reversed wave equations differ as we can observe from the bottom row of plots in Figure 1.1. Aside these amplitude differences, which can be significant as we can see from the plots in Figure 1.2 extracted at the location of the red cross in Figure 1.1, there are occasional sign flips, which means that the phase is modeled incorrectly. Both phase [41, 40, 17] and amplitudes need to be modeled correctly if we want to correctly implement sensitivities that involve multi-dimensional cross correlations between the solution of the forward wave equation acting on the source and the adjoint wave equation acting on the residual wavefield. While at first sight the kinematics may be modeled accurately by time reversal, the aforementioned differences in amplitude and sign can result in significant differences in the sensitivities and therefore in the resulting image as I will show below.

The observed differences between the impulse response of the time-reversed forward and adjoint TTI wave equation underline the importance of having access to correct a forward/adjoint pair when computing sensitivities to the parameters of the wave equation, which are our object of interest. When the gradients of our data misfit objectives contain errors, we can not expect gradient based inversion algorithms to converge [69, 70, 71, 35]. Getting these sensitivities correct is challenging certainly when dealing with involved wave physics requiring complex parametrization such as the in the TTI wave equation. Below, I will discuss techniques I have used to deal with these complexities.

1.4.2 Verification of sensitivities

Without having accurate and verifiable access to how the solution of the wave equation changes to an infinitesimal change in it parameterization we are not in a position to minimize our data-misfit objectives. I use the so-called adjoint or dottest to verify that I am computing the correct adjoint anisotropic wavefield. This test is designed to numerically guarantee that the forward-adjoint pair of PDEs are numerically adjoints [10]. The adjoint

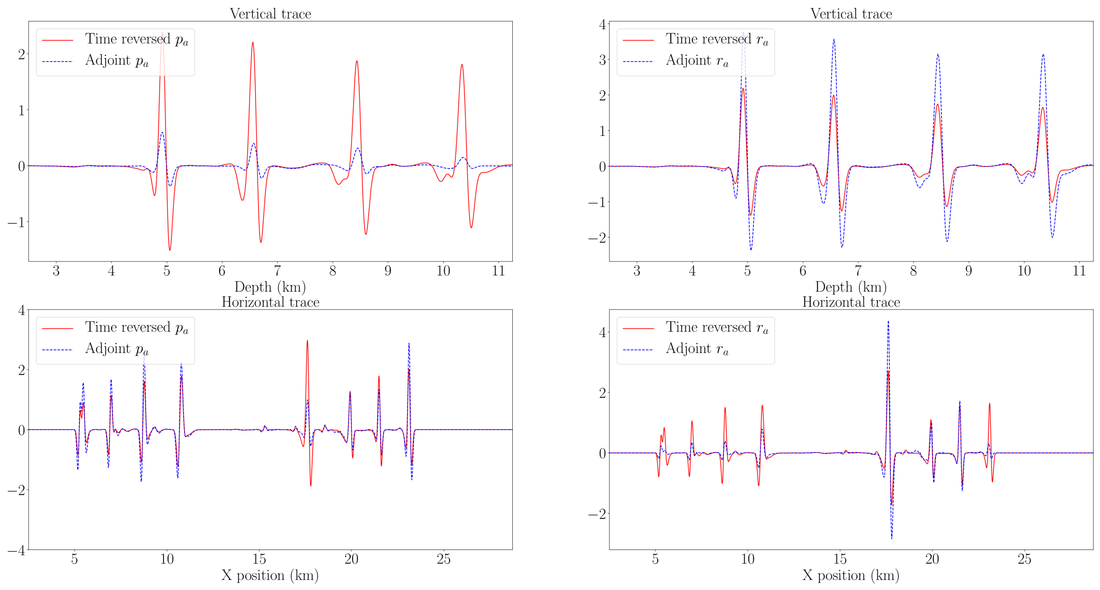


Figure 1.2: Vertical and horizontal traces, extracted from the two-dimensional adjoint and time-reversed wavefields.

test verifies that the following equality stands:

$$\langle \mathbf{A}(\mathbf{m})^{-1}\mathbf{P}_s^{*}\mathbf{q}, (\mathbf{p}^{\top}, \mathbf{r}^{\top})^{\top} \rangle = \langle \mathbf{q}, \mathbf{P}_s\mathbf{A}(\mathbf{m})^{-*}(\mathbf{p}^{\top}, \mathbf{r}^{\top})^{\top} \rangle. \tag{1.11}$$

The term on the left computes the inner product between the wavefield everywhere whereas the inner product on the right involves the source wavefield that lives at the source locations alone. It is important that we consider the wavefield everywhere, rather than restricted to the receiver positions alone, because our gradient calculations involve the wavefield everywhere. To arrive at a practical test, I recast Equation 1.11 as:

$$\text{adjoint-error} = 100\left(\frac{\langle \mathbf{A}(\mathbf{m})^{-1}\mathbf{P}_s^{*}\mathbf{q}, (\mathbf{p}^{\top}, \mathbf{r}^{\top})^{\top} \rangle}{\langle \mathbf{q}, \mathbf{P}_s\mathbf{A}(\mathbf{m})^{-*}(\mathbf{p}^{\top}, \mathbf{r}^{\top})^{\top} \rangle} - 1\right). \tag{1.12}$$

This equation measures the relative error between the left hand-side and right-hand side of Equation 1.11. Ideally, the error (difference between the two terms) should be exactly zero. Due to numerical discretization errors and boundary effects, I measure the relative percentage error between the two inner products. This error should be close to zero for

Table 1.1: Adjoint test for the different wave-equation and their respective adjoint/time-reverse. We show the normalized error, i.e the error w.r.t the wave-equation. The results for the two layer model are as expected good for all kernel as it only measures the isotropic first layer while the transmission experiment and the 2007 BP model [72] demonstrates that only the true adjoint passes the dot-test for a strongly anisotropic media.

Wave-equation	Two layer	Transmission	BP2007
Adjoint	.16%	1.6%	.5%
Time reversed	2.97%	2.1%	11.6%

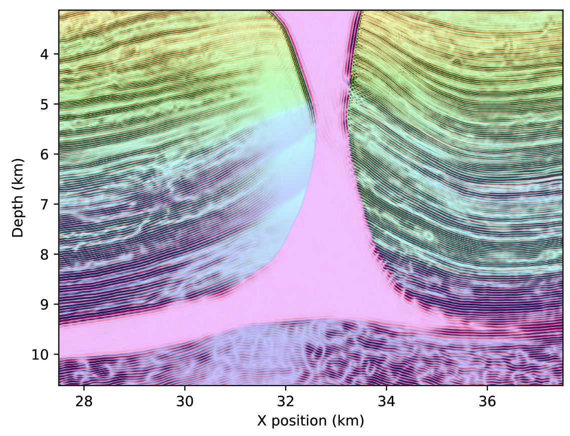
a correct adjoint. Table 1.1 list the adjoint-errors according to Equation 1.12 for three different anisotropic models, namely a simple two layers model with limited anisotropy variations, a model with smoothly varying anisotropy parameters, and finally a section of the highly complex and realistic 2007 BP TTI model [72]. By design, these three models are increasingly more anisotropic illustrating the increased need for exact adjoints as the TTI wave-equation 1.7 can only be considered a self-adjoint when the medium is close to homogeneous. As the errors in Table 1.1 confirms, we can no longer consider the TTI equation as self adjoint when the medium properties vary realistically and we need in that case to rely on the true adjoint to control the error in the above adjoint test. If this error becomes too large, the gradient of the data misfit objective will not pass the gradient test and we can expect to produce inaccurate images as I show below.

1.4.3 Imaging

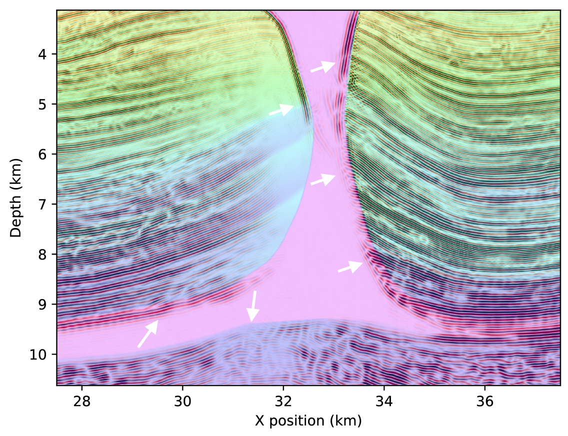
As I mentioned above, the error incurred by the time-reversed waver-equation carries through the inversion and this can lead to incorrect images where reflectors are mispositioned, blurry, and of wrong amplitude. These effects can be explained because the time-reverse wave equation leads to wavefields that are of wrong amplitude with events that may have to wrong sign. Because the gradients for each source experiment themselves are based on multi-dimensional cross-correlations between the forward and adjoint wavefields (reduced adjoint state gradient from Equation 1.4), these errors can lead to errors in the image as illustrated in the enclosed images of parts of the BP 2007 TTI model [72] depicted in Figure 1.3. These three images show the difference between imaging with correct adjoint (left column) and the incorrect time-reversed "adjoint" (right column). When comparing the images obtained with these two methods, we find that overall the images for the correct adjoint are crisper while the corresponding images with time-reversal are less focused and blurry. This effect is most prominent for the plots at the bottom. More importantly, we observe major mispositioning of the imaged reflectors compared to the ground truth velocity model plotted in color. These errors are most prominent on the boundary between the high-velocity salt (depicted in pink) and the sediments. Reflectors that do not exist in reality appear in the salt when we use the incorrect time reversal. While these mispositionings may seem minor, they correspond to errors in the order of the wavelength that can be detrimental when delineating oil & gas reservoirs. We observe these errors in areas where the model is strongly anisotropic with strong tilt angles.

It is clear that at all times we want to avoid making errors of the kind I just described. For that reason, we need to implement correct pairs of forward and adjoint wave equations effectively doubling the amount of programming if these equations are implemented by hand. This explains why the industry seldomly implements these equations with the possible detrimental consequences as I outlined above.

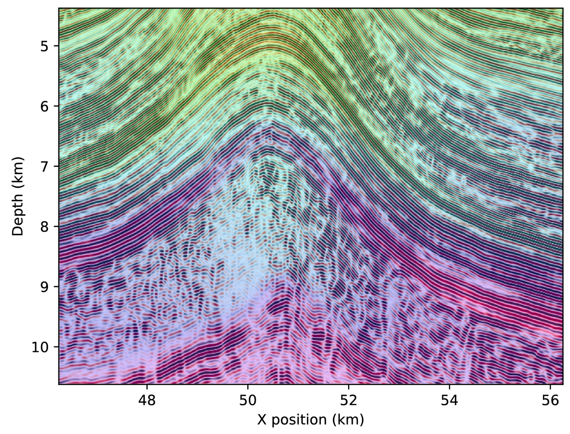
However, when using automatic code generation we overcome these problems and this provides the main motivation for my thesis. The above example clearly highlights the need for a sophisticated computational and development framework designed to carry out simulations for inversion which included modeling of complex (anisotropic) wave motion, correct adjoints, and sensitivities. Devito allowed me to implement this introductory anisotropic example in a matter of weeks while the implementation and comparison of different TTI discretization could have taken up months or even years if done by hand in a low level language.



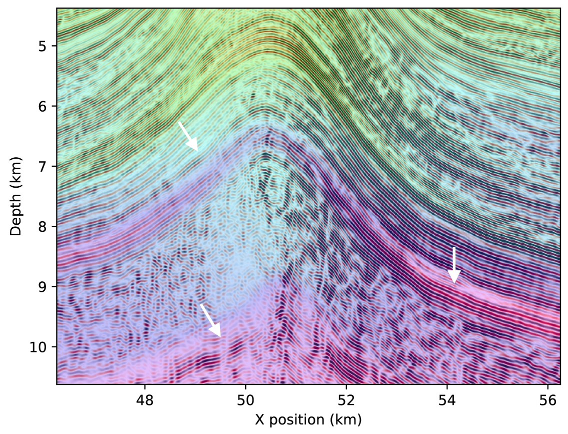
Adjoint



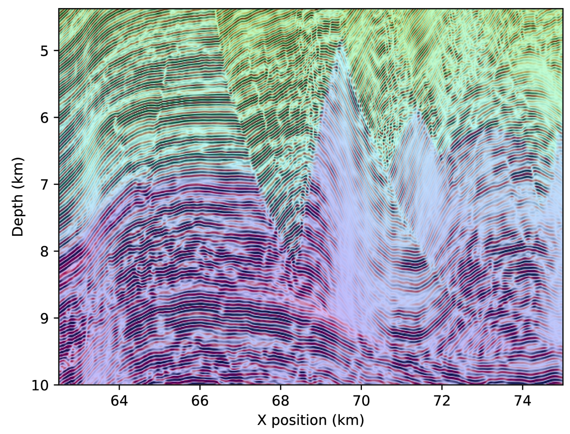
Time-reverse



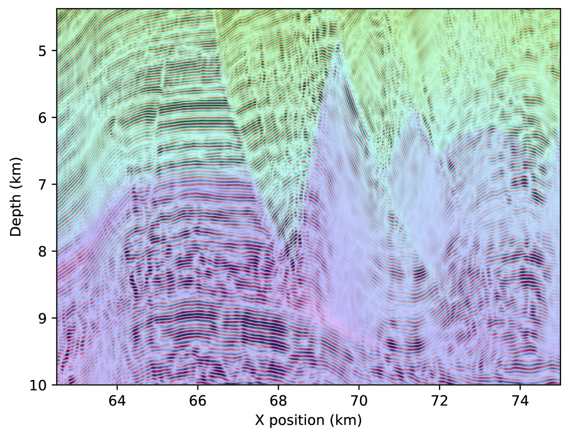
Adjoint



Time-reverse



Adjoint



Time-reverse

Figure 1.3: Selected areas of the RTM image of the BP model overlaid with the background velocity model. The left column is the image with the true adjoint wavefield and the right column with the time-reversed as the adjoint. These selected areas are the areas of the model that are the most complicated to image due to strong anisotropy and strong anisotropy variations.

1.5 Objectives

In light of the above motivational example in conjunction with the scientific problems I described, the objectives of my research are oriented towards computational software design for seismic inverse problems, and more generally PDE constrained optimization.

The first objective is to design a performance metric that is portable and does not rely on relative measures or self-comparison. To do so, that performance metric requires not to be associated with code-to-code comparison but with an absolute measure of performance. This measure is the hardware usage that is representable on a roofline model plot [34, 73]. This measure provides two main outputs. First, the design of finite-difference software can be driven by the roofline model as it provides theoretical estimates of the maximum achievable performance based on the choice of hardware and discretization. Moreover, the use of roofline models avoids poor or non-improvable performance after months of coding due to early choices. Finally, benchmarks obtained with the roofline model are absolute, portable and unbiased.

With a performance metric in place, the second objective is to develop software that enables research for domain specialists. The turnaround time for research and development is conventionally hindered by the lack of computational software targeting research and development rather than production. As the implementation of new PDE solver can be extremely time-consuming, research and development needs to have access to high-level interfaces to computationally performant finite-difference propagators.

The second objective is to provide an interface at user level that allows easy implementation of complex PDEs so that domain specialists such as geophysicists and mathematicians have the tools necessary to focus on algorithms, geophysical modeling, inversion, and acquisition design. As I show in Chapter 3, dedicated to Devito's Application Programming Interface (API), high-level interfaces already exist or have been in development for years. However, these frameworks usually focus only on finite-difference discretization, which is not enough for measurement-based inverse problem such as seismic inversion that involve the minimization of data objectives using local derivative information (sensitivities).The objective is to enable not only finite-difference propagators but to also provide a high-level interface to non-standard finite-difference operations such as source injection, measurement interpolation, on the fly Fourier transform and more generally any stencil operations on a structured grid or part of it. Moreover, adjoint based inverse problems usually require to store the history of the forward wavefield. For large scale problems, such as seismic inversion, the size of this wavefield can reach Terabytes and requires advanced methods to store it such as checkpointing [29] or Fourier compression [30]. The high-level interface is also meant to be general enough to allow the definition and implementation at a symbolic level of these methods.

The third and final objective is to provide a framework that, while implementing a high-level user interface, gives state-of-the-art computational performance. The performance optimization is enabled through code generation and just-in-time compilation. The idea of code generation is fairly new but has gained a lot of attention thanks to recent improvements in the performance of automatically generated code. These improvements come from the automation of modern high performance computational methods such as vectorization, memory padding, cache-blocking and shared/distributed memory computation [46, 48, 49]. Code-generation also enables portability in the sense that the same code can be executed on multiple computer platforms as the code-generation framework and compiler takes care of the translation to code for the new hardware. While conventional in house hand-coded solvers may be efficient on one specific computer architecture, its portability is very limited as the low-level code is usually hand-tune for the specific hardware at hand (on top of being hard to modify as pointed out earlier). The objective is to implement a code generation framework under the high-level use interface that implements and automates all the methods known to improve the computational performance of the generated code. This automation also contains platform specific details for portability. This framework renders the high-level interface usable for any users for academic teaching, research and development and production through a high-level user interface to large scale production that still benefits from low-level optimized C-code from the code-generation framework and compiler.

As I will show in each chapter, my work also focuses on improving verifiability and reproducibility of research. Reproducibility is one of the main requirements for the dissemination of knowledge. Low level and hand-coded software tend to not be portable as the code is platform specific or not easily installable and executable. Even though some framework (such as [12]) went through the massive effort to make hand-coded software reproducible, the hardware specific implementation limits its potential portability to similar computer architectures. Devito, with its high-level interface and code-generation framework, provides a fully reproducible documentation and set of example that only requires the installation of standard packages such as `Python`. Second, Devito uses the roofline model to benchmark its performance (and automatically generates the necessary runtimes and parameters) and does not rely on a comparison against a reference homemade code and allows users to easily benchmark Devito and its generated code on any hardware in hand. the high-level interface and the roofline together provide a reproducible and portable software for both examples and performance benchmark.

1.6 My contributions

I now describe my scientific contribution with regards to the computational and software challenges related to wave-equation based geophysical exploration, and more generally computational modeling.

- My first contribution is theoretical performance estimation of finite-difference solvers with the roofline model [43, 34]. This work is presented in Chapter 2 of the thesis. In this work, I looked at how finite-differences stencils can be theoretically studied to infer the optimal achievable performance associated with its computational implementation on specific hardware. First, I show that the choice of discretization can be decided before implementation based on the hardware available and the estimate of arithmetic and operational intensity [34]. Doing so allows me to make informed decisions rather than post-implementation benchmarking of a code that may not be a fit for the available hardware architecture. Secondly, I show that traditional runtime based benchmarking does not necessarily reflect the efficiency of a code nor how much improvement can be made, while the roofline model provides an absolute measure of computational performance that is not artificially boosted by comparing against a known slow code. The roofline model also provides insights on how much improvement can be achieved.

- My second contribution is Devito, and more specifically its symbolic API, presented in Chapter 3. Devito is a generic domain-specific language (DSL) for finite-difference based on explicit time stepping. At its core, Devito implements a generic structured grid stencil DSL. It uses `sympy`[45] to provide a high-level symbolic API, which allows users to mathematically, by basically writing out the PDE, define finite-difference propagators. The original design and motivation behind Devito is seismic inverse problems, but is extends to more general time-dependent PDE-constrained optimization problems. Because Devito provides a symbolic interface that allows to write complex mathematical representations of the physics such as TTI in a simple way, my work enables and improves turn around times of research and development for domain-specialists. Even though other high-level symbolic interfaces exist, such as `sympy` [45], these approaches do not necessarily provide adequate computational performance to be appealing for users. For example, `sympy` offers a wide variety of tools for symbolic manipulation but the (numerical) evaluation of a symbolic expression relies on computationally inefficient symbol replacement rules that scale exponentially with the number of symbols in the expression. Devito, on the other hand, possesses a code generation framework with its own just-in-time compiler that comes with computational performance in par with hand-coded low-level propagators. I designed and mostly implemented the symbolic API, which is a central part, but the larger Devito project has contributions from other researchers and different institutions, principally Imperial College London.

- My third contribution involves work I have done on the Devito compiler itself and is presented in Chapter 4. While I have designed the high-level symbolic core API and implemented the majority of it, with Dr. Luporini providing me substantial feedback and contribution, I only contributed to the development of the Devito compiler that was designed, and largely implemented, by Dr Luporini. I provided substantial feedback on what needed to be supported and partially implemented and designed how to interface it with high-level API. The Devito compiler is designed to be portable and therefore supports multiple backends(targeted hardware and/or compiler). My main contributions to this part of the development of Devito involved development and implementation of the `core` backend that targets CPUs (Intel, AMD, ARM). I mainly contributed to the part of Devito responsible for the symbolic manipulations of the generated finite-difference stencils and to the overall integration of Devito and `sympy`. I made sure that the compiler would always understand and process the symbolic expressions to generate efficient code. Finally, I ensured that the generated code always produces the correct result and wrote a major part of the extensive automated testing framework designed to make sure that results produced by Devito are correct. As can be seen from Devito's contributors list, I am one of the two main contributors of Devito with Dr Luporini (and the recent addition of R. Nelson that made major contributions to the MPI APIs and 'numpy'' interface), and the commit history highlights my involvement in the compiler side and design and implementation of the symbolic API.

My first contribution is highlighted in Chapter 2 while Chapter 3 and 4 present Devito's API, compiler, its testing framework and details the benchmarking results on realistic problems.

1.7 Outline

My thesis is organized as follows:

The first chapter presents my work on the roofline model for finite-difference solver for different wave-equations [43]. This work is published in Computer and Geoscience. The roofline model is a broadly used tool in the computer science community [34, 74, 75, 73] and offers hardware level performance measurement and prediction. In this work, I provide a theoretical study of the optimal performance achievable for finite-difference solver for a range of wave-equation. This theoretical study highlights the computational complexities and potential of these wave-equations and shows that careful design choices have to be made depending on the problem. This work allows to extend the conventional runtime only performance in geophysics to a more portable and absolute measure. I also link in this work the roofline performance metric to the conventional runtime performance based on the theoretical estimates of hardware usage.

The second chapter presents Devito and its API [13]. This chapter is published in Geoscientific Model Development (GMD) and was selected as one of their highlight papers. In this chapter, the core design principle and implementation of a domain-specific language for finite-difference solver are presented. I present in detail the implementation and specifics of Devito's symbolic interface and how I inherited `sympy` symbolic capabilities to design a user interface that allows the mathematical definition of finite-difference propagators. This work describes step-by-step the API and how I designed its testing framework. I will show that every part of Devito is tested such as a complete verification of the numerical accuracy of the generated code. I also detail its usage over a range of seismic and computational fluid dynamics examples. Two hands-in tutorials of Devito in the context of seismic inversion have been published in The Leading Edge (TLE) as part of a tutorial series on full-waveform inversion (FWI) [76, 77, 78].Devito, now in use in research and production, provides the high-level interface necessary to allow domain specialists to concentrate on research and development.

The third chapter, published in TOMS(ACM Transactions on Mathematical Software), details the Devito compiler [42] that makes Devito a computationally efficient solution for large scale and industry size problems instead of a small scale development tool. This chapter describes each layer of the compiler, from the symbolic definition of the stencil to the compilation of the generated C-code with all the layers of symbolic and C-level optimizations. The computational performance of the generated code on realistic problems is studied as well and its portability is demonstrated with the benchmark of Intel accelerators (Xeon Phi).

Finally, the conclusions give a summary of my work, as well as a brief outlook of the research that was enabled by my work and my contributions to it. I also briefly describe my latest developments in Devito, more specifically the support for vectorial equations, and finally discuss future work and remaining open questions.

CHAPTER 2
PERFORMANCE PREDICTION OF FINITE-DIFFERENCE SOLVERS FOR DIFFERENT COMPUTER ARCHITECTURES

2.1 Introduction

The increasing complexity of modern computer architectures means that developers are having to work much harder at implementing and optimizing scientific modelling codes for the software performance to keep pace with the increase in performance of the hardware. This trend is driving a further specialization in skills such that the geophysicist, numerical analyst and software developer are increasingly unlikely to be the same person. One problem this creates is that the numerical analyst makes algorithmic choices at the mathematical level that define the scope of possible software implementations and optimizations available to the software developer. Additionally, even for an expert software developer it can be difficult to know what are the right kind of optimizations that should be considered, or even when an implementation is ”good enough” and optimization work should stop. It is common that performance results are presented relative to a previously existing implementation, but such a relative measure of performance is wholly inadequate as the reference implementation might well be truly terrible. One way to mitigate this issue is to establish a reliable performance model that allows a numerical analyst to make reliable predictions of how well a numerical method would perform on a given computer architecture, before embarking upon potentially long and expensive implementation and optimization phases. The availability of a reliable performance model also saves developer effort as it both informs the developer on what kind of optimizations are beneficial, and when the maximum expected performance has been reached and optimization work should stop.

Performance models such as the roofline model by [74] help establish statistics for best case performance — to evaluate the performance of a code in terms of hardware utilization (e.g. percentage of peak floating point performance) instead of a relative speed-up. Performance models that establish algorithmic optimality and provide a measure of hardware utilization are increasingly used to determine effective algorithmic changes that reliably increase performance across a wide variety of algorithms [75]. However, for many scientific codes used in practice, wholesale algorithmic changes, such as changing the spatial discretization or the governing equations themselves, are often highly invasive and require a costly software re-write. Establishing a detailed and predictive performance model for the various algorithmic choices is therefore imperative when designing the next-generation of industry scale codes.

We establish a theoretical performance model for explicit wave-equation solvers as used in full waveform inversion (FWI) and reverse time migration (RTM). We focus on a set of widely used equations and establish lower bounds on the degree of the spatial discretization required to achieve optimal hardware utilization on a set of well known modern computer architectures. Our theoretical prediction of performance limitations may then be used to inform algorithmic choice of future implementations and provides an absolute measure of realizable performance against which implementations may be compared to demonstrate their computational efficiency.

For the purpose of this paper we will only consider explicit time stepping algorithms based on a second order time discretization. Extension to higher order time stepping scheme will be briefly discussed at the end. The reason we only consider explicit time stepping is that it does not involve any matrix inversion, but only scalar product and additions making the theoretical computation of the performance bounds possible. The performance of other classical algorithm such as matrix vector products or FFT as described by [34] has been included for illustrative purposes.

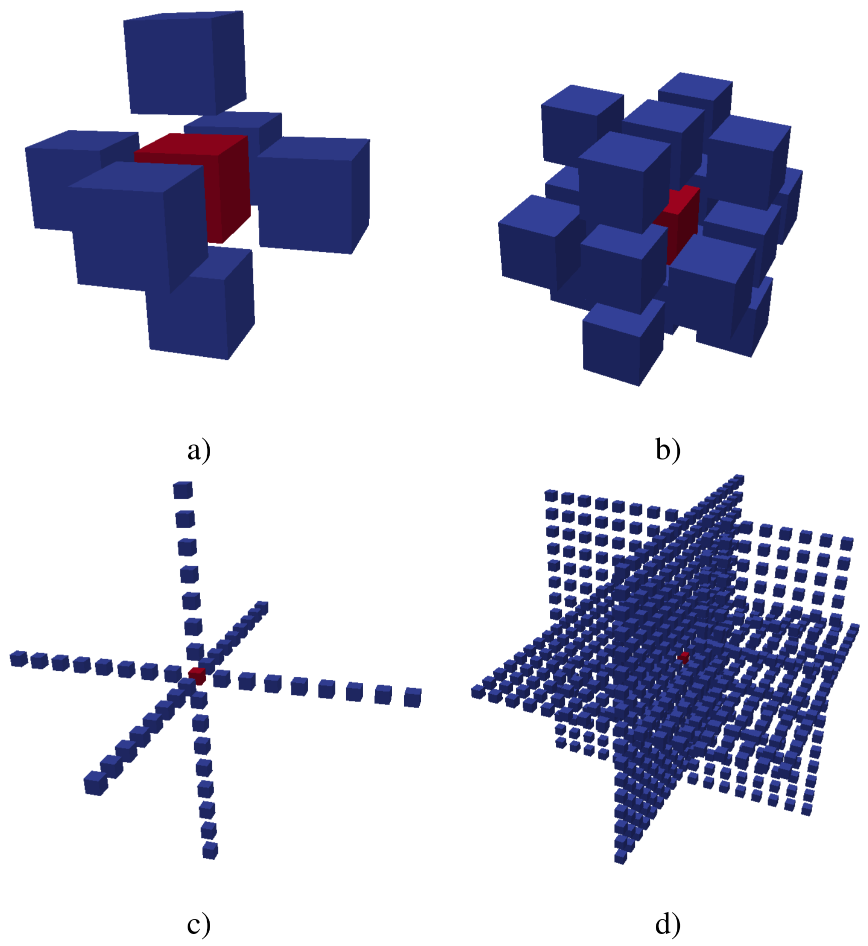
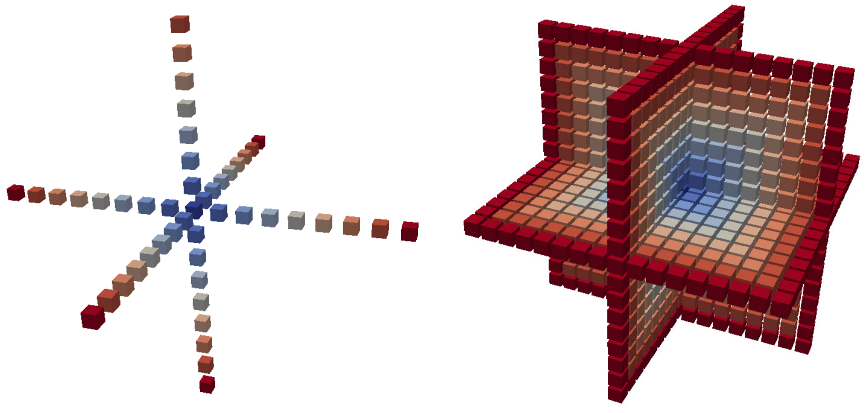


Figure 2.1: Stencil for the acoustic and anisotropic wave-equation for different orders of discretization. A new value for the centre point (red) is obtained by weighted sum of the values in all the neighbor points (blue). a) 2nd order laplacian, b) second order rotated Laplacian, c) 16th order Laplacian, d) 16th order rotated Laplacian

2.2 Introduction to stencil computation

A stencil algorithm is designed to update or compute the value of a field in one spatial location according to the neighboring ones. In the context of wave-equation solver, the stencil is defined by the support (grid locations) and the coefficients of the finite-difference scheme. We illustrate the stencil for the Laplacian, defining the stencil of the acoustic wave-equation (Equation 2.8), and for the rotated Laplacian used in the anisotropic wave-equation (Equation 2.10, 2.11) on Figure 4.4 - 2.2. The points colored in blue are the value loaded while the point colored in red correspond to a written value.



a) b)

Figure 2.2: Stencil for the 16th order acoustic and anisotropic wave-equation with distance to centre highlighting a) Laplacian, b) rotated Laplacian

The implementation of a time stepping algorithm for a wavefield u, solution of the acoustic wave-equation (Equation 2.8) is straightforward from the representation of the stencil. We do not include the absorbing boundary conditions (ABC) as depending on the choice of implementation it will either be part of the stencil or be decoupled and treated separately.

Algorithm 1 Time-stepping

```
for $t = 0$ to $t = n_t$ do
    for $(x, y, z) \in (X, Y, Z)$ do
        $$u(t, x, y, z) = 2u(t-1, x, y, z) - u(t-2, x, y, z) + \sum_{i \in stencil} a_i u(t-1, x_i, y_i, z_i)$$
    end for
    Add Source : $u(t, ., ., .) = u(t, ., ., .) + q$
end for
```

In Algorithm 1, (X, Y, Z) is the set of all grid positions in the computational domain, (x, y, z) are the local indices ,(x_i, y_i, z_i) are the indices of the stencil positions for the centre position (x, y, z) and n_t is the number of time steps and q is the source term decoupled from the stencil. In the following we will concentrate on the stencil itself, as the loops in space and time do not impact the theoretical performance model we introduce. The roofline model is solely based on the amount of input/output (blue/red in the stencils) and arithmetic operations (number of sums and multiplication) required to update one grid point, and we will prove that the optimal reference performance is independent of the size of the domain (number of grid points) and of the number of time steps.

2.2.1 Notes on parallelization:

Using a parallel framework to improve an existing code is one of the most used tool in the current stencil computation community. It is however crucial to understand that this is not an algorithmic improvement from the operational intensity. We will prove that the algorithmic efficiency of a stencil code is independent of the size of the model, and will therefore not be impacted by a domain-decomposition like parallelization via OpenMP or MPI. The results shown in the following are purely dedicated to help the design of a code from an algorithmic point of view, while parallelization will only impact the performance of the implemented code by improving the hardware usage.

2.3 Roofline Performance Analysis

The roofline model is a performance analysis framework designed to evaluate the floating point performance of an algorithm by relating it to memory bandwidth usage [74]. It has proved to be very popular because it provides a readily comprehensible performance metric to interpret runtime performance of a particular implementation according to the achievable optimal hardware utilization for a given architecture [79]. This model has been applied to real-life codes in the past to analyze and report performance including oceanic climate models [80], combustion modeling [81] and even seismic imaging [82]. It has also been used to evaluate the effectiveness of implementation-time optimizations like autotuning [83], or cache-blocking on specific hardware platforms like vector processors [84] and GPUs [85]. Tools are available to plot the machine-specific parameters of the roofline model automatically [86]. When more information about the target hardware is available, it is possible to refine the roofline model into the cache-aware roofline model which gives more accurate predictions of performance [87]. The analysis presented here can be extended to the cache-aware roofline model but in order to keep it general, we restrict it to the general roofline model.

The roofline model has also been used to compare different types of basic numerical operations to predict their performance and feasibility on future systems [88], quite similar to this paper. However, in this paper, instead of comparing stencil computation to other numerical methods, we carry out a similar comparison between numerical implementations using different stencil sizes. This provides an upper-bound of performance on any hardware platform at a purely conceptual stage, long before the implementation of the algorithm.

Other theoretical models to predict upper-bound performance of generic code on hypothetical hardware have been built [89, 90, 91, 92] but being too broad in scope, can not be used to drive algorithmic choice like choosing the right discretization order. Some of these models have also been applied to stencil codes [93, 94], however the analysis was of a specific implementation and could not be applied in general. There are many tools to perform performance prediction at the code-level [95, 96, 97, 98]. However, any tool that predicts performance based on a code is analyzing the implementation and not the algorithm in general. Although performance modeling is a deep and mature field, most work is restricted to modeling the performance of specific implementations in code.Hofmann, Fey, Riedmann, Eitzinger, Hager, and Wellein makes a comparison quite similar to the one we do here where two algorithmic choices for the same problem are being compared with a performance model.

In this section we demonstrate how one creates a roofline model for a given computer architecture, and derives the operational intensity for a given numerical algorithm. This establishes the theoretical upper-bound for the performance of a specific algorithm on that architecture. A general roofline performance analysis consists of three steps:

- The memory bandwidth, bytes per second, and the peak number of floating point operations per second (FLOPS) of the computer architecture are established either from the manufacturers specification or through measurement using standard benchmarks.

- The operational intensity (OI) of the algorithm is established by calculating the ratio of the number of floating point operations performed to memory traffic, FLOPs per byte. This number characterizes the algorithmic choices that affect performance on a computer system. In combination with the measured memory bandwidth and peak performance of a computer architecture, this provides a reliable estimate of the maximum achievable performance.
- The solver is benchmarked in order to establish the achieved performance. A roofline plot can be created to illustrate how the achieved performance compares to the maximum performance predicted by the roofline for the algorithms OI. This establishes a measure of optimality of the implementation, or alternatively the maximum possible gain from further optimization of the software.

2.3.1 Establishing the Roofline

The roofline model characterizes a computer architecture using two parameters: the maximum memory bandwidth, B_{peak}, in units of $bytes/s$; and the peak FLOPS achievable by the hardware, F_{peak}. The maximally achievable performance F_{ac} is modelled as:

$$F_{ac} = \min\left(\mathcal{I}B_{peak}, F_{peak}\right), \tag{2.1}$$

where $\mathcal{I}$ is the OI.

As illustrated in Figure 2.3 this limitation defines two distinct regions:

- **Memory-bound**: The region left of the ridge point constitutes algorithms that are limited by the amount of data coming into the CPU from memory. Memory-bound codes typically prioritize caching optimizations, such as data reordering and cache blocking.

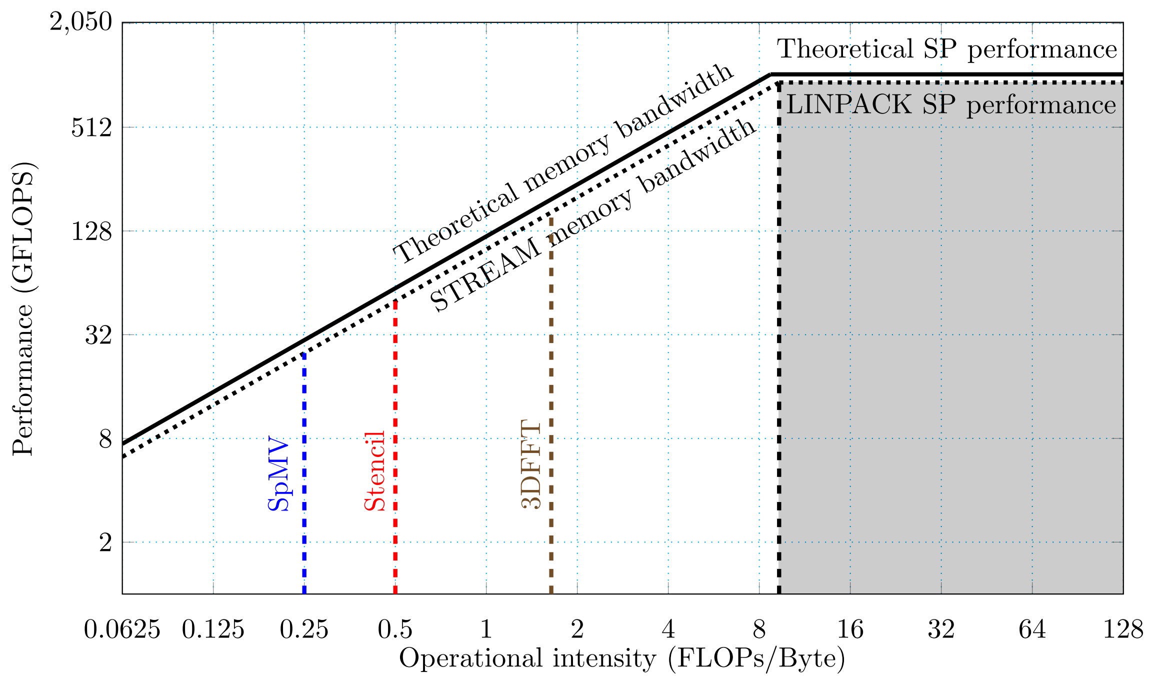


Figure 2.3: Roofline diagram showing the operational intensity of three well-known algorithms as reported by [74]: sparse matrix-vector multiplication (SpMV), stencil computation and 3D Fast Fourier Transform (3DFFT). The hardware limits are taken from [100] and the compute-limited area is highlighted through shading.

- **Compute-bound**: The region right of the ridge point contains algorithms that are limited by the maximum performance of the arithmetic units in the CPU and thus defines the maximum achievable performance of the given architecture. Compute-bound codes typically prioritize vectorization to increase throughput.

It is worth noting that changing from single to double-precision arithmetic halves the OI because the volume of memory that must be transferred between the main memory and the CPU is doubled. The peak performance will be impacted as well, since the volume of data and the number of concurrently used floating point units (FPU) changes. As commonly employed by industry, we assume single precision arithmetic for the examples presented here, but it is straightforward to extend to double precision.

Andreolli, Thierry, Borges, Skinner, and Yount illustrates an example of deriving the theoretical performance for a system that consists of two Intel Xeon E5-2697 v2 (2S-E5) with 12 cores per CPU each running at 2.7 Ghz without turbo mode. Since these processors support 256-bit SIMD instructions they can process eight single-precision operations per clock-cycle (SP FP). Further, taking into account the use of Fused Multiply-Add (FMA) operations (two per cycle), this yields

$$\begin{aligned}F_{peak} &= 8(SPFP) \times 2(FMA) \times 12(cores) \times 2(CPUs) \times 2.7\text{Ghz} \\ &= 1036.8 \text{ GFLOPS.}\end{aligned}$$

Clearly, this assumes full utilization of two parallel pipelines for Add and Multiply operations.

A similar estimate for the peak memory bandwidth F_{peak} can be made from the memory frequency ($1866\ GHz$), the number of channels (4) and the number of bytes per channel (8) and the number of CPUs (2) to give $F_{peak} = 1866 \times 4 \times 8 \times 2 = 119\ GByte/s$.

It is important to note here that there is an instruction execution overhead that the above calculations did not take into account and therefore these theoretical peak numbers are not achievable ($\simeq 80\%$ is achievable in practice [100]). For this reason, two benchmark algorithms, STREAM TRIAD for memory bandwidth [101, 102] and LINPACK for floating point performance [103], are often used to measure the practical limits of a particular hardware platform. These algorithms are known to achieve a very high percentage of the peak values and are thus indicative of practical hardware limitations.

2.3.2 Performance Model

The key measure to using the roofline analysis as a guiding tool for algorithmic design decisions and implementation optimization is the operational intensity, $\mathcal{I}$, as it relates the number of FLOPs to the number of bytes moved to and from RAM. $\mathcal{I}$ clearly does not capture many important details about the implementation such as numerical accuracy or time to solution. Therefore, it is imperative to look at $\mathcal{I}$ in combination with these measures when making algorithmic choices.

Here we analyze the algorithmic bounds of a set of finite-difference discretizations of the wave-equation using different stencils and spatial orders. We therefore define algorithmic operational intensity $\mathcal{I}_{alg}$ in terms of the total number of FLOPs required to compute a solution, and we assume that our hypothetical system has a cache with infinite size and no latency inducing zero redundancy in memory traffic [79]. This acts as a theoretical upper bound for the performance of any conceivable implementation.

We furthermore limit our theoretical investigation to analyzing a single time step as an indicator of overall achievable performance. This assumption allows us to generalize the total number of bytes in terms of the number of spatially dependent variables (e.g. wavefields, physical properties) used in the discretized equation as $\mathcal{B}_{global} = 4N(l + 2s)$, where l is the number of variables whose value is being loaded, s is the number of variables whose value is being stored, N is the number of grid points and 4 is the number of bytes per single-precision floating point value. The term $2s$ arises from the fact that most computer architectures will load a cache line before it gets overwritten completely. However, some computer architectures, such as the Intel Xeon Phi, have support for stream stores, so that values can be written directly to memory without first loading the associated cache line, in which case the expression for the total data movement becomes $\mathcal{B}_{global} = 4N(l + s)$. It is important to note here that limiting the analysis to a single time step limits the scope of the infinite caching assumption above.

Since we have assumed a constant grid size N across all spatially dependent variables, we can now parametrize the number of FLOPs to be computed per time step as $\mathcal{F}_{total}(k) = N\mathcal{F}_{kernel}(k)$, where $\mathcal{F}_{kernel}(k)$ is a function that defines the number of flops performed to update one grid point in terms of the stencil size k used to discretize spatial derivatives. Additional terms can be added corresponding to source terms and boundary conditions but they are a small proportion of the time step in general and are neglected here for simplicity. This gives us the following expression for OI as a function of k, $\mathcal{I}_{alg}(k)$:

$$\mathcal{I}_{alg}(k) = \mathcal{F}_{total}(k)/\mathcal{B}_{global} = \frac{\mathcal{F}_{kernel}(k)}{4(l + s)}. \tag{2.2}$$

2.4 Operational intensity for finite-differences

We derive a general model for the operational intensity of wave-equation PDEs solvers with finite-difference discretizations using explicit time stepping and apply it to three different wave-equation formulations commonly used in the oil and gas exploration community: an acoustic anisotropic wave-equation; vertical transverse isotropic (VTI); and tilted transversely isotropic (TTI) [104]. The theoretical operational intensity for the 3D discretized equations will be calculated as a function of the finite-difference stencil size k, which allows us to make predictions about the minimum discretization order required for each algorithm to reach the compute-bound regime for a target computer architecture. For completeness we describe the equations in Appendix 2.A.1.

2.4.1 Stencil operators

As a baseline for the finite-difference discretization, we consider the use of a 1D symmetric stencil of size k, which uses k values of the discretized variable to compute any spatial derivatives enforcing a fixed support for all derivatives. Other choices of discretization are possible, such as choosing the stencil for the first derivative and applying it iteratively to obtain high order derivatives. Our analysis will still be valid but require a rewrite of the following atomic operation count. The number of FLOPs used for the three types of derivatives involved in our equation are calculated as:

- first order derivative with respect to x_i ($\frac{du}{dx_i}$): $(k+1)$ mult $+(k-1)$ add $= 2k \, FLOPs$
- second order derivative with respect to x_i ($\frac{d^2u}{dx_i^2}$): $(k+1)$ mult $+$ $(k-1)$ add $=$ $2k \, FLOPs$
- second order cross derivative with respect to x_i, x_j ($\frac{d^2u}{dx_i dx_j}$): (k^2-2k) mult $+ (k^2 - 2k - 1)$ add $= 2k^2 - 4k - 1 \, FLOPs$

where in 3D, x_i for $i = 1, 2, 3$ correspond to the three dimensions x, y, z and u is the discretized field.

Table 2.1: Derivation of $FLOPs$ per stencil invocation for each equation.

Equation	$\frac{du}{dx_i}$	$\frac{d^2u}{dx_i^2}$	$\frac{d^2u}{dx_i dx_j}$	mult	add	duplicates
Acoustic:	0	$3 \times 2k$	0	3	5	-4
VTI: $2 \times ($	0	$3 \times 2k$	0	5	5	$-2)$
TTI: $2 \times ($	0	$3 \times 2k$	$3 \times (2k^2 - 4k - 1)$	44	17	$-8)$

Computing the total wavefield memory volume B_{global} for each equation we have $4 \times 4N\ bytes$ for Acoustic (load velocity, two previous time steps and write the new time step), $9 \times 4N\ bytes$ for VTI (load velocity, two anisotropy parameters, two previous time steps for two wavefields and write the new time step for the two wavefields) and $15 \times 4N\ bytes$ for TTI (VTI plus 6 precomputed cos/sin of the tilt and dip angles). Equation 2.2 allows us to predict the increase of the operational intensity in terms of k by replacing B_{global} by its value. The OI $\mathcal{I}_{alg}(k)$ for the three wave-equations is given by:

- Acoustic anisotropic: $\mathcal{I}_{alg}(k) = \frac{3k}{8} + \frac{1}{4}$,
- VTI: $\mathcal{I}_{alg}(k) = \frac{k}{3} + \frac{4}{9}$,
- TTI: $\mathcal{I}_{alg}(k) = \frac{k^2}{5} - \frac{k}{5} + \frac{5}{3}$,

and plotted as a function of k on Figure 2.10. Using the derived formula for the algorithmic operational intensity in terms of stencil size, we can now analyze the optimal performance for each equation with respect to a specific computer architecture. We are using the theoretical and measured hardware limitations reported by [100] to demonstrate how the main algorithmic limitation shifts from being bandwidth-bound at low k to compute-bound at high k on a dual-socket Intel Xeon in Figure 2.4 - 2.6 and an Intel Xeon Phi in Figure 2.7 - 2.9.

It is of particular interest to note from Figure 2.4 that a 24^{th} order stencil with $k = 25$ provides just enough arithmetic load for the 3D acoustic equation solver to become

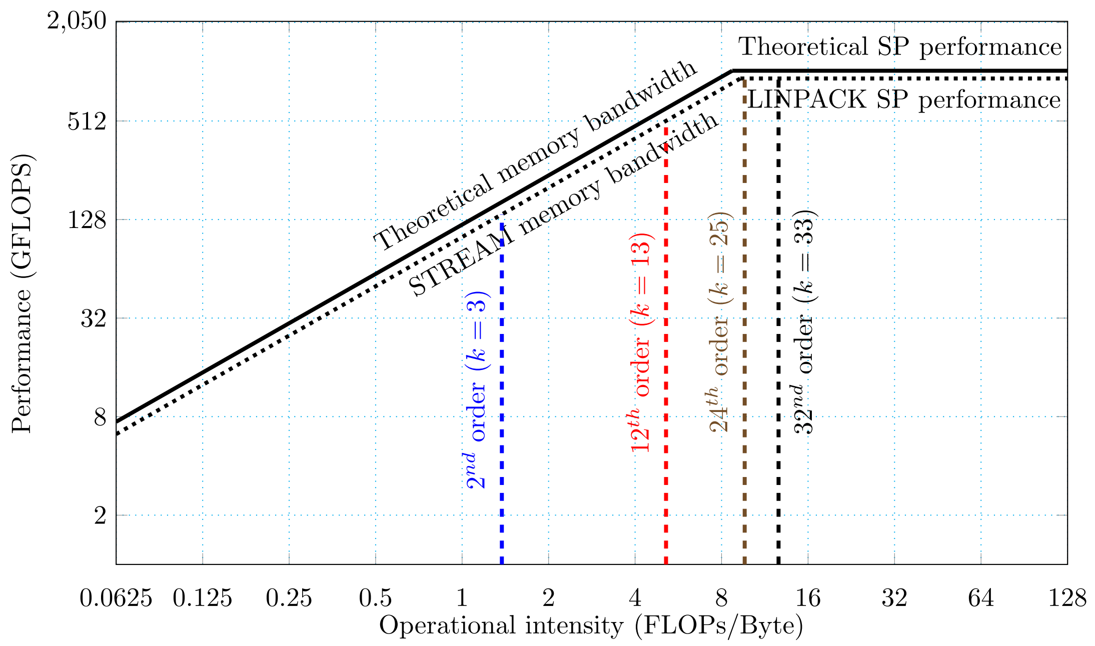


Figure 2.4: Increase in algorithmic OI with increasing stencil sizes on a dual-socket Intel Xeon E5-2697 v2 [100] for a 3D acoustic kernel. The 24^{th} order stencil is coincident with the ridge point — the transition point from memory-bound to compute-bound computation.

compute-bound, while $k = 25$ falls just short of the compute-bound region for the VTI algorithm. On the other hand a 6^{th} order stencil with $k = 7$ is enough for the TTI algorithm to become compute-bound due to having a quadratic slope with respect to k (Figure 2.10) instead of a linear slope.

At this point, we can define $\mathcal{I}_{min}$, which is the minimum OI required for an algorithm to become compute-bound on a particular architecture, as the x-axis coordinate of the ridge point in Figure 2.4 - 2.6 and 2.7 - 2.9. Note that the ridge point x-axis position changes between the two different architectures. This difference in compute-bound limit shows that a different spatial order discretization should be used on the two architecture to optimize hardware usage. As reported by [100] the $\mathcal{I}_{min}$ as derived from achievable peak rates is $9.3\ FLOPs/byte$ for the Intel Xeon and $10.89\ FLOPs/byte$ for the Intel Xeon Phi. This entails that while the acoustic anisotropic wave-equation and VTI are memory bound for discretizations up to 24^{th} order, the TTI equation reaches the compute bound region with even a 6^{th} order discretization.

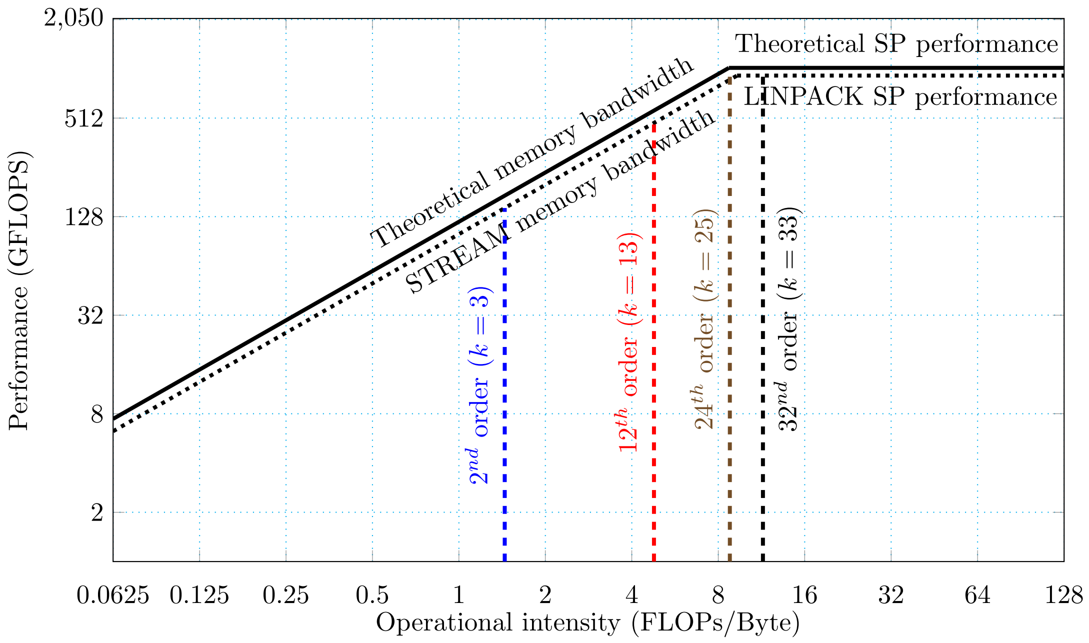


Figure 2.5: Increase in algorithmic OI with increasing stencil sizes on a dual-socket Intel Xeon E5-2697 v2 [100] for a 3D VTI kernel. Similarly to the acoustic model, the 24^{th} order stencil is coincident with the ridge point.

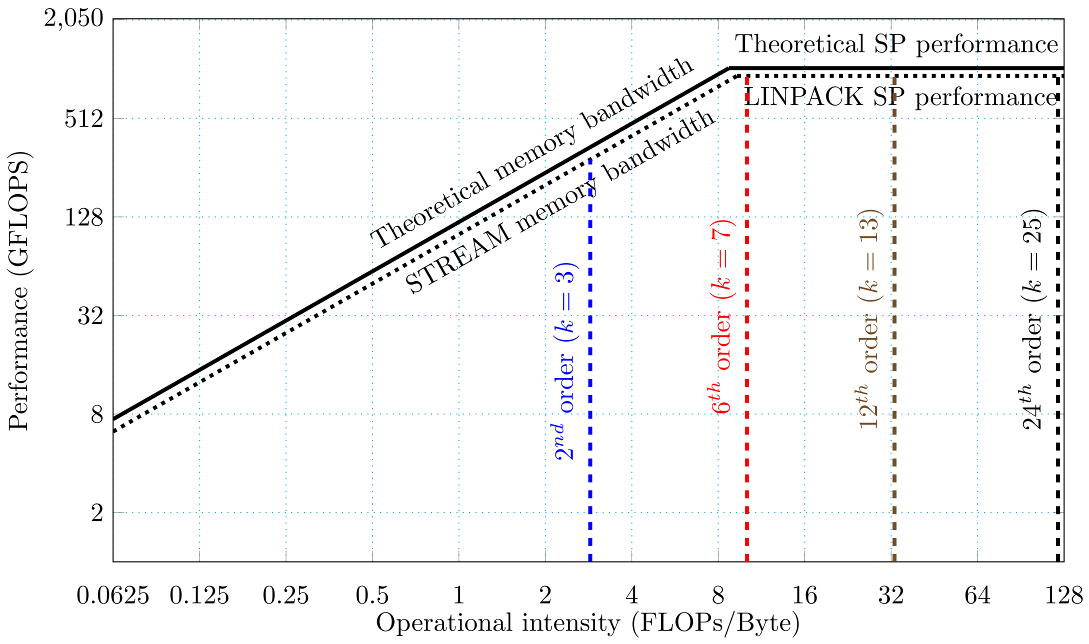


Figure 2.6: Increase in algorithmic OI with increasing stencil sizes on a dual-socket Intel Xeon E5-2697 v2 [100] for a 3D TTI kernel. The 6^{th} order stencil is already compute-bound.

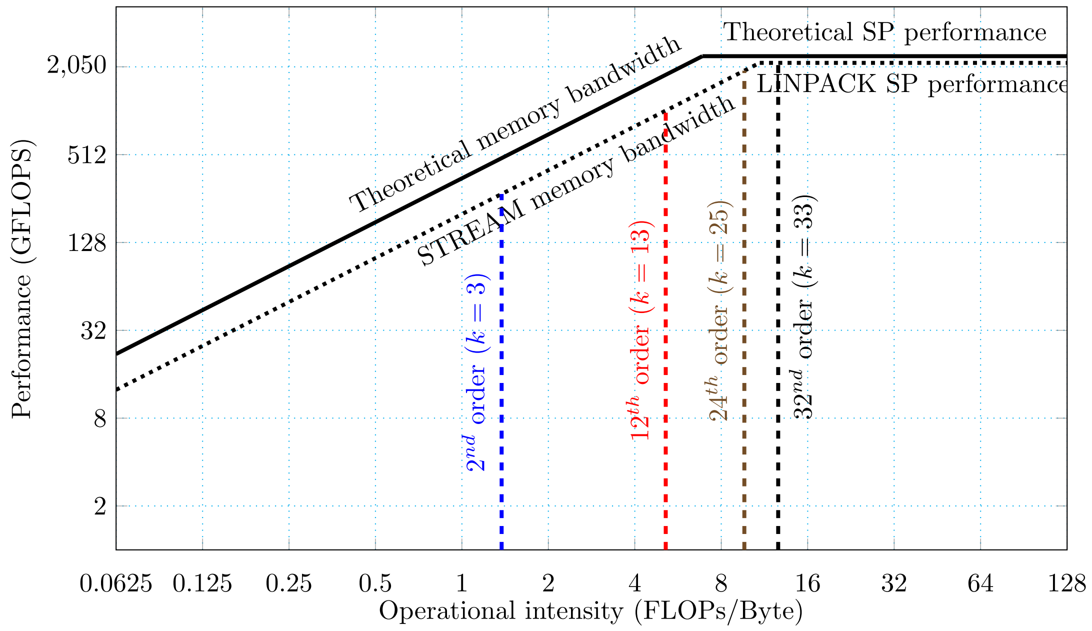


Figure 2.7: Increase in algorithmic OI with increasing stencil sizes on a Intel Xeon Phi 7120A co-processor [100] for a 3D acoustic kernel. Unlike the Xeon E5-2697, the 30^{th} order stencil is the smallest one to be compute-bound (vs 24^{th} order).

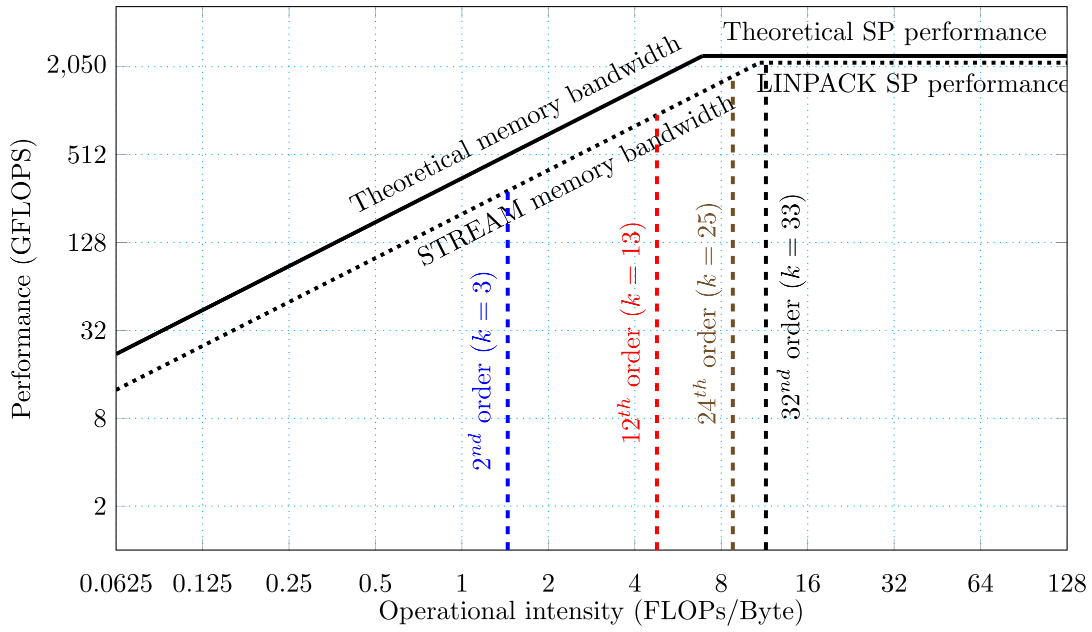


Figure 2.8: Increase in algorithmic OI with increasing stencil sizes on a Intel Xeon Phi 7120A co-processor [100] for a 3D VTI kernel. 32^{nd} is the minimum compute-bound stencil. It is not equivalent to the acoustic on this architecture.

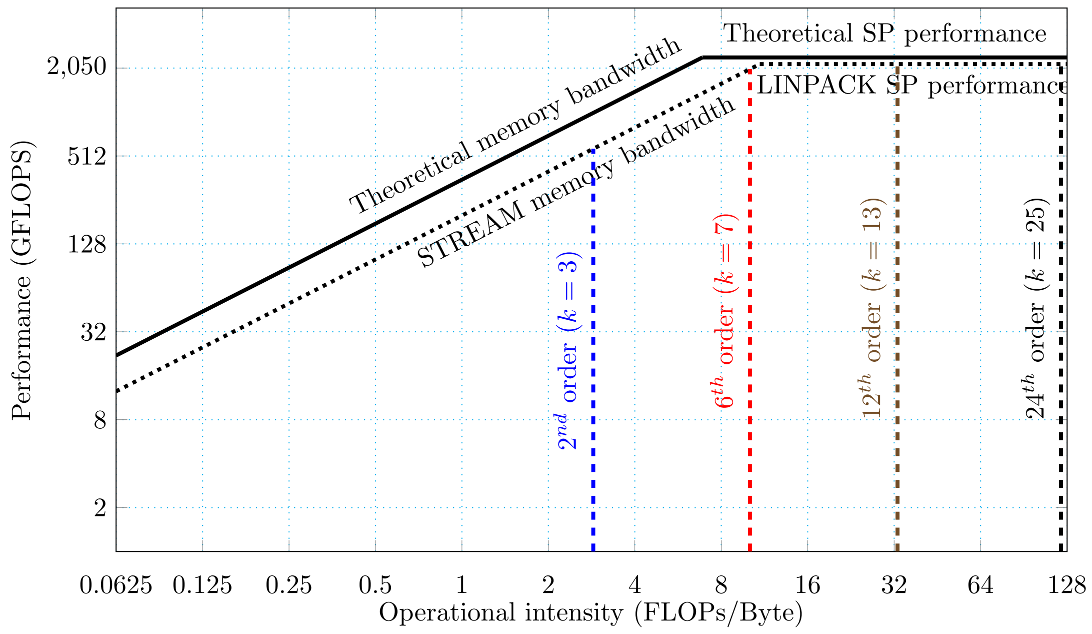


Figure 2.9: Increase in algorithmic OI with increasing stencil sizes on a Intel Xeon Phi 7120A co-processor [100] for a 3D TTI kernel. The 6^{th} order stencil is already compute-bound similarly to the Xeon E5-2697.

From the analytical expression derived we can now generalize the derivation of minimum OI values by plotting the simplified expressions for $\mathcal{I}_{alg}(k)$ against known hardware OI limitations, as shown in Figure 2.10. We obtain a theoretical prediction about the minimum spatial order required for each algorithm to provide enough arithmetic load to allow implementations to become compute-bound. Most importantly, Figure 2.10 shows that the TTI wave-equation has a significantly steeper slope of $\mathcal{I}(k)$, which indicates that it will saturate a given hardware for a much smaller spatial discretization than the acoustic wave or the VTI algorithm.

Moreover, assuming a spatial discretization order of $k-1$, we can predict that on the Intel Xeon CPU we require a minimum order of 24 for the acoustic wave solver, 26 for VTI and 6 for TTI. On the Nvidia GPU, with a slightly lower hardware $\mathcal{I}$, we require a minimum order of 22 for the acoustic wave solver, 24 for VTI and 6 for TTI, while even larger stencils are required for the Intel Xeon Phi accelerator: a minimum order of 28 for the acoustic wave solver, 30 for VTI and 6 for TTI. This derivation demonstrates that

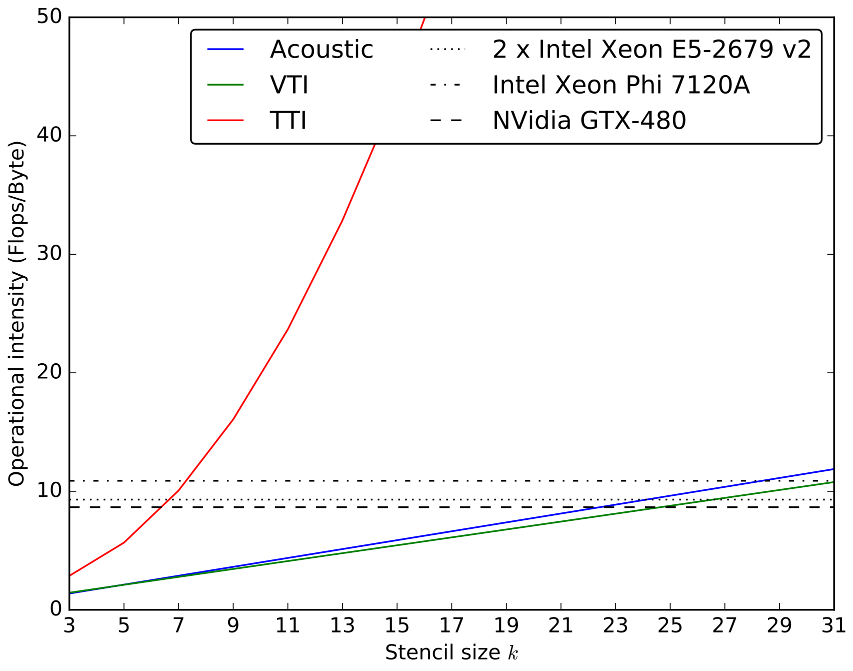


Figure 2.10: Increase in OI with stencil size k and machine-specific minimum OI values for all three hardware architectures considered in this paper.

overall very large stencils are required for the acoustic anisotropic solver and VTI to fully utilize modern HPC hardware, and that even TTI requires at least order 6 to be able to computationally saturate HPC architectures with a very high arithmetic throughput, like the Intel Xeon Phi.

2.5 Example: MADAGASCAR modelling kernel

We demonstrate our proposed performance model and its flexibility by applying it on a broadly used and benchmarked modelling kernel contained in Madagascar [12]. We are illustrating the ease to extend our method to a different wave-equation and by extension to any PDE solver. The code implements the 3D anisotropic elastic wave-equation and is described in [105]. We are performing our analysis based on the space order, hardware and runtime described in [105]. The governing equation considered is:

$$\begin{aligned}
&\rho\frac{d^2u_i}{dt^2} = \frac{d\sigma_{ij}}{dx_j} + F_i,\\
&\sigma_{ij} = c_{ijkl}\epsilon_{kl},\\
&\epsilon_{kl} = \frac{1}{2}[\frac{u_l}{dx_k} + \frac{u_k}{dx_l}],\\
&u_i(.,0) = 0,\\
&\frac{du_i(x,t)}{dt}|_{t=0} = 0.
\end{aligned} \tag{2.3}$$

where ρ is the density, u_i is the i^{th} component of the three dimensional wavefield displacement ($i = 1, 2, 3$ for x, y, z), F is the source term,ϵ is the strain tensor, σ is the stress tensor and c is the stiffness tensor. The equation is discretized with an 8^{th} order star stencil for the first order derivatives and a second order scheme in time and solves for all three components of u. Equation 2.3 uses Einstein notations meaning repeated indices represent summation:

$$\begin{aligned}
\frac{d\sigma_{ij}}{dx_j} &= \sum_{j=1}^{3}\frac{\sigma_{ij}}{dx_j},\\
c_{ijkl}\epsilon_{kl} &= \sum_{k=1}^{3}\left(\sum_{l=1}^{3}c_{ijkl}\epsilon_{kl}\right).
\end{aligned} \tag{2.4}$$

From this equation and knowing the finite-difference scheme used we can already compute the minimum required bandwidth and operational intensity. We need to solve this equation for all three components of the wave u at once as we have coupled equations in ϵ and u. For a global estimate of the overall memory traffic, we need to account for loading and storing $2 \times 3N$ values of the displacement vector and loading N values of ρ. In case the stiffness tensor is constant in space the contribution of c_{ijkl} is 64 independently of N, which yields an overall data volume of $\mathcal{B}_{global} = 4N(6 + 1) + 64 \simeq 28N\ Bytes$. In the realistic physical configuration of a spatially varying stiffness tensor, we would estimate loading $64N$ values of c_{ijkl}, leaving us with a data volume of $B_{global} = 4N(6 + 1 + 64) =$

$284N\ Bytes$. Finally we consider symmetries in the stiffness tensor are taken into account reducing the number of stiffness values to load to $21N$ and leading to a data volume of $B_{global} = (6 + 1 + 21) \times 4N = 112N\ Bytes$.

The number of valuable FLOPs performed to update one grid point can be estimated by:

- 9 first derivatives ($\partial_k u_l$, for all $k, l = 1, 2, 3$) : $9 \times (8 \text{ mult} + 7 \text{ add}) = 135\ FLOPs$
- 9 sums for ϵ_{kl} (9×9 adds) and 9×8 mult for σ_{ij} = 153 $FLOPs$
- 9 first derivatives $\partial_j \sigma_{ij}$ and 9 sums = 144 $FLOPs$
- 3 times 3 sums to update u_i = 9 $FLOPs$.

The summation of all four contributions gives a total of 441 operations and by dividing by the memory traffic we obtain the operational intensity $\mathcal{I}_{stiff}$ for variable stiffness and $\mathcal{I}_{const}$ for constant stiffness:

$$\begin{aligned} \mathcal{I}_{stiff} &= \frac{441N}{112N} = 3.93, \\ \mathcal{I}_{const} &= \frac{441N}{28N} = 15.75. \end{aligned} \tag{2.5}$$

Using the OI values derived above we can now quantify the results presented by [105] by interpreting their runtime results with respect to our performance measure. The achieved GFLOPS have been obtained on the basis of 1000 time steps with 8^{th} order spatial finite-differences and 2^{nd} order temporal finite-differences. We interpret Figure 11a) of [105] to give a run time of approximately 53 seconds and a domain size of $N = 225^3$. We obtain with this parameter the following achieved performances:

$$\begin{aligned} F &= \frac{N^3 F_{kernel} N_t}{W}, \\ &= \frac{225^3 \times 441 \times 1000}{53}, \\ &= 94.8\text{GFLOPS}, \end{aligned} \tag{2.6}$$

where N_t is the number of time steps, and W is the run time.

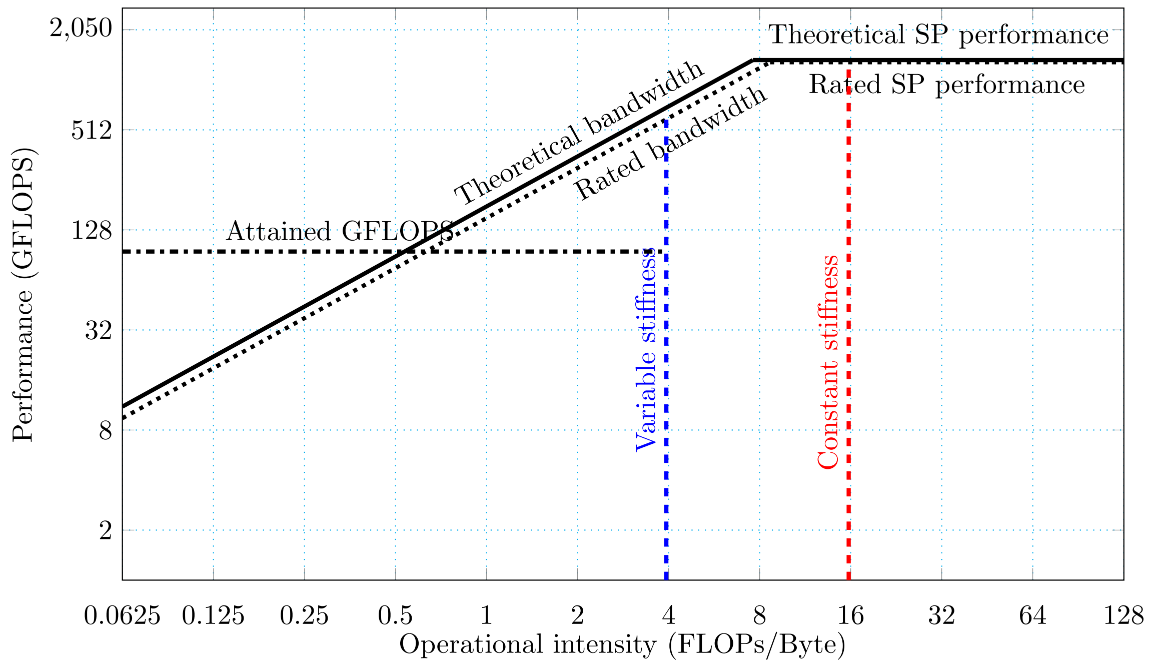


Figure 2.11: Roofline model for the 3D elastic anisotropic kernel from [105] on a 480-core NVIDIA GTX480 GPU (with hardware specification from [106]).

Figure 2.11 shows the calculated performance in relation to our predicted algorithmic bounds $\mathcal{I}_{stiff}$ and $\mathcal{I}_{const}$. The use of a constant stiffness tensor puts the OI of the considered equation in the compute-bound region for the benchmarked GPU architecture (NVIDIA GTX480). Assuming a spatially varying stiffness tensor, we can calculate an achieved hardware utilization of 40.5% based on the reported results, assuming an achievable peak memory bandwidth of $150.7\ GByte/s$, as reported by [106] and a maximum achievable performance of $150.7\ GByte/s \times 1.5528\ FLOPs/Byte = 234\ GFLOPS$. Assuming 80% [100] of peak performance is achievable, the roofline model suggests that there is still potential to double the performance of the code through software optimization. It is not possible to draw such a conclusion from traditional performance measures such as timings or scaling plots. This highlights the importance of a reliable performance model that can provide an absolute measure of performance in terms of the algorithm and the computer architecture.

2.6 Cost-benefit analysis

So far we discussed the design of finite-difference algorithms purely from a performance point of view without regard to the numerical accuracy and cost-to-solution. Now we discuss the impact of the discretization order on the achieved accuracy of the solution and how that, in turn, affects the wall clock time required for computation. To do so, we look at the numerical requirements of a time-stepping algorithm for the wave-equation. More specifically we concentrate on two properties, namely dispersion and stability, in the acoustic case. This analysis is extendable to more advanced wave-equations such as VTI and TTI with additional numerical analysis. The dispersion criteria and stability condition for the acoustic wave-equation is given by [107, 108]:

$$\begin{aligned} &\frac{v_{max} dt}{h} \leq \sqrt{\frac{a_1}{a_2}} \text{ CFL condtion, stability} \\ &h \leq \frac{v_{min}}{p f_{max}} \text{ dispersion criterion,} \end{aligned} \tag{2.7}$$

where:

a_1 is the sum of the absolute values of the weights of the finite-difference scheme for the second time derivative of the wavefield; $\frac{\partial^2 u}{\partial t^2}$

a_2 is the sum of the absolute values of the weights of the finite-difference approximation of $\nabla^2 u$;

v_{max} is the maximum velocity;

f_{max} is the maximum frequency of the source term that defines the minimum wavelength for a given minimum velocity $\lambda_{min} = \frac{v_{min}}{f_{max}}$;

p is the number of grid points per wavelength. The number of grid points per wavelength impacts the amount of dispersion (different wavelengths propagating at different velocities) generated by the finite-difference scheme. The lower the number, the higher the dispersion will be for a fixed discretization order.

These two conditions define the computational setup for a given source and physical model size. Knowing that a_2 increases with the spatial discretization order, Equation 2.7 shows that higher discretization orders require a smaller time-step hence increasing the total number of time steps for a fixed final time and grid size. However, higher order discretizations also allow to use less grid points per wavelength (smaller p). A smaller number of grid points per wavelengths leads to a smaller overall computational domain as a fixed physical distance is represented by a coarser mesh and as the grid spacing has been increased, the critical time-step (maximum stable value) is also increased. Overall, high order discretizations have better computational parameters for a predetermined physical problem. From these two considerations, we can derive an absolute cost-to-solution estimation for a given model as a function of the discretization order for a fixed maximum frequency and physical model size. The following results are not experimental runtimes but estimations of the minimum achievable runtime assuming a perfect performance implementation. We use the following setup:

- We fix the physical model size as 500 grid point in all three directions for a second order discretization (minimum grid size).
- The number of grid points per wavelength is $p = 6$ for a second order spatial discretization and $p = 2$ for a 24th order discretization and varies linearly for intermediate orders.
- The number of time steps is 1000 for the second order spatial discretization and computed according to the grid size/time step for other spatial orders.

The hypothetical numerical setup (with $a_1 = 4$, second order time discretization) is summarized in Table 2.2. We combine the estimation of a full experimental run with the estimated optimal performance and obtain an estimation of the optimal time-to-solution for a fixed physical problem. The estimated runtime is the ratio of the total number of GFLOPs (multiply $\mathcal{F}_{kernel}$ by the number of grid points and time steps) to the maximum achievable performance for this OI. Table 2.3 shows the estimated runtime assuming peak performance on two systems: a dual-socket Intel Xeon E5-2697 v2 and an Intel Xeon Phi 7120A co-processor.

Table 2.2: Cost-to-solution computational setup summary.

Order	a_2	p	h	dt	N	n_t
2nd order	12	6	1	0.5774	1.25e+08	1000
6th order	18.13	5	1.2	0.5637	7.24e+07	1024
12th order	21.22	4	1.5	0.6513	3.70e+07	887
18th order	22.68	3	2	0.8399	1.56e+07	688
24th order	23.57	2	3	1.2359	4.63e+06	468

Table 2.3: Cost-to-solution estimation for several spatial discretizations on fixed physical problem.

Order	$\mathcal{I}_{alg}(k)$	GFLOPs	GFLOPS Xeon	GFLOPS Phi	Runtime Xeon	Runtime Phi
2nd	1.375	2.75e+03	137.5	275	20s	10s
6th	2.875	3.414e+03	287.5	575	12s	6s
12th	5.125	2.691e+03	512.5	1025	6s	3s
18th	7.375	1.266e+03	737.5	1475	2s	1s
24th	9.625	3.337e+02	962.5	1925	1s	1s

We see that by taking advantage of the roofline results in combination with the stability conditions, we obtain an estimate of the optimal cost-to-solution of an algorithm. It can be seen that higher order stencils lead to better hardware usage by lowering the wall-time-to-solution. These results, however, rely on mathematical results based on homogeneous velocity. In the case of an heterogenous model, high order discretizations may result in inaccurate, even though stable and non dispersive, solutions to the wave-equation. The choice of the discretization order should then be decided with more than just the performance in mind.

2.7 Conclusions

Implementing an optimizing solver is generally a long and expensive process. Therefore, it is imperative to have a reliable estimate of the achievable peak performance, FLOPS, of an algorithm at both the design and optimized implementation stages of development.

The roofline model provides a readily understandable graphical tool, even for a non-specialist, to quickly assess and evaluate the computational effectiveness of a particular implementation of an algorithm. We have shown how the roofline model can be applied to finite-difference discretizations of the wave-equation commonly used in the geophysics community. Although the model is quite simple, it provides a reliable estimate of the peak performance achievable by a given finite-difference discretization regardless of the implementation. Not only does this aid the algorithm designer to decide between different discretization options but also gives solver developers an absolute measure of the optimality of a given implementation. The roofline model has also proved extremely useful in guiding further optimization strategies, since it highlights the limitations of a particular version of the code, and gives an indication of whether memory bandwidth optimizations, such as loop blocking techniques, or FLOPs optimizations, such as SIMD vectorization, are likely to improve results.

However, one should always be mindful of the fact that it does not provide a complete measure of performance and should be complemented with other metrics, such as time to solution or strong scaling metrics, to establish a full understanding of the achieved performance of a particular algorithmic choice and implementation.

2.A Appendix

2.A.1 Wave-equations

In the following equations u is the pressure field in the case of acoustic anisotropic while p, r are the split wavefields for the anisotropic case. We denote by $u(., 0)$ and respectively p, r the value of u for all grid points at time $t = 0$. The physical parameters are m the square slowness, ϵ, δ the Thomsen parameters and θ, ϕ the tilt and azimuth. The main problem with the TTI case is the presence of transient functions (cos, sin) known to be extremely expensive to compute (typically about an order of magnitude more expensive than an add or multiply). Here we will assume these functions are precomputed and come from a look-up table, thus only involving memory traffic In the acoustic anisotropic case the governing equations are:

$$\begin{aligned}
&m\frac{d^2u(x,t)}{dt^2} - \nabla^2 u(x,t) = q,\\
&u(.,0) = 0,\\
&\frac{du(x,t)}{dt}|_{t=0} = 0.
\end{aligned} \tag{2.8}$$

In the anisotropic case we consider the equations describe in [104]. More advanced formulation have been developed however this equation allow an explicit formulation on the operational intensity and simple stencil expression. It is the formulation we are also using in our code base. In the VTI case the governing equations are:

$$\begin{aligned}
&m\frac{d^2p(x,t)}{dt^2} - (1+2\epsilon)D_{xx}p(x,t) - \sqrt{(1+2\delta)}D_{zz}r(x,t) = q,\\
&m\frac{d^2r(x,t)}{dt^2} - \sqrt{(1+2\delta)}D_{xx}p(x,t) - D_{zz}r(x,t) = q,\\
&p(.,0) = 0,\\
&\frac{dp(x,t)}{dt}|_{t=0} = 0,\\
&r(.,0) = 0,\\
&\frac{dr(x,t)}{dt}|_{t=0} = 0.
\end{aligned} \tag{2.9}$$

For TTI the governing equations are:

$$
\begin{aligned}
&m\frac{d^2p(x,t)}{dt^2} - (1+2\epsilon)(G_{\bar{x}\bar{x}} + G_{\bar{y}\bar{y}})p(x,t) - \sqrt{(1+2\delta)}G_{\bar{z}\bar{z}}r(x,t) = q,\\
&m\frac{d^2r(x,t)}{dt^2} - \sqrt{(1+2\delta)}(G_{\bar{x}\bar{x}} + G_{\bar{y}\bar{y}})p(x,t) - G_{\bar{z}\bar{z}}r(x,t) = q,\\
&p(.,0) = 0,\\
&\frac{dp(x,t)}{dt}|_{t=0} = 0,\\
&r(.,0) = 0,\\
&\frac{dr(x,t)}{dt}|_{t=0} = 0,
\end{aligned}
\tag{2.10}
$$

where the rotated differential operators are defined as

$$
\begin{aligned}
G_{\bar{x}\bar{x}} =& cos(\phi)^2cos(\theta)^2\frac{d^2}{dx^2} + sin(\phi)^2cos(\theta)^2\frac{d^2}{dy^2} +\\
& sin(\theta)^2\frac{d^2}{dz^2} + sin(2\phi)cos(\theta)^2\frac{d^2}{dxdy} - sin(\phi)sin(2\theta)\frac{d^2}{dydz} - cos(\phi)sin(2\theta)\frac{d^2}{dxdz}\\
G_{\bar{y}\bar{y}} =& sin(\phi)^2\frac{d^2}{dx^2} + cos(\phi)^2\frac{d^2}{dy^2} - sin(2\phi)^2\frac{d^2}{dxdy}\\
G_{\bar{z}\bar{z}} =& cos(\phi)^2sin(\theta)^2\frac{d^2}{dx^2} + sin(\phi)^2sin(\theta)^2\frac{d^2}{dy^2} +\\
& cos(\theta)^2\frac{d^2}{dz^2} + sin(2\phi)sin(\theta)^2\frac{d^2}{dxdy} + sin(\phi)sin(2\theta)\frac{d^2}{dydz} + cos(\phi)sin(2\theta)\frac{d^2}{dxdz}.
\end{aligned}
\tag{2.11}
$$

CHAPTER 3
DEVITO API

3.1 introduction

Large-scale inversion problems in exploration seismology constitute some of the most computationally demanding problems in industrial and academic research. Developing computationally efficient solutions for applications such as seismic inversion requires expertise ranging from theoretical and numerical methods for partial differential equation (PDE) constrained optimization to low-level performance optimization of PDE solvers. Progress in this area is often limited by the complexity and cost of developing bespoke wave propagators (and their discrete adjoints) for each new inversion algorithm or formulation of wave physics. Traditional software engineering approaches often lead developers to make critical choices regarding the numerical discretization before manual performance optimization for a specific target architecture and making it ready for production. This workflow of bringing new algorithms into production, or even to a stage that they can be evaluated on realistic datasets can take many person-months or even person-years. Furthermore, it leads to mathematical software that is not easily ported, maintained or extended. In contrast, the use of high-level abstractions and symbolic reasoning provided by domain-specific languages (DSL) can significantly reduce the time it takes to implement and verify individual operators for use in inversion frameworks, as has already been shown for the finite element method [31, 32, 109].

State-of-the-art seismic imaging is primarily based upon explicit finite difference schemes due to their relative simplicity and ease of implementation [110, 111, 105]. When considering how to design a DSL for explicit finite difference schemes, it is useful to recognize the algorithm as being primarily a sub-class of stencil algorithms or polyhedral computation [112, 100, 113]. However, stencil compilers lack two significant features required to develop a DSL for finite differences: symbolic computational support required to express finite difference discretizations at a high level, enabling these expressions to be composed and manipulated algorithmically; support for algorithms that are not stencil-like, such as source and receiver terms that are both sparse and unaligned with the finite difference grid. Therefore, the design aims behind the Devito DSL can be summarized as:

- create a high-level mathematical abstraction for programming finite differences to enable composability and algorithmic optimization,
- insofar as possible use existing compiler technologies to optimize the affine loop nests of the computation, which account for most of the computational cost,
- develop specific extensions for other parts of the computation that are non-affine (e.g., source and receiver terms).

The first of these aims is primarily accomplished by embedding the DSL in *Python* and leveraging the symbolic mathematics package Sympy [45]. From this starting point, an abstract syntax tree is generated and standard compiler algorithms can be employed to either generate optimized and parallel C code or to write code for a stencil DSL - which itself will be passed to the next compiler in the chain. The fact that this can be all performed just-in-time (JIT) means that a combination of static and dynamic analysis can be used to generate optimized code. However, in some circumstances, one might also choose to compile offline.

The use of symbolic manipulation, code generation and just-in-time compilation allows the definition of individual wave propagators for inversion in only a few lines of *Python* code, while aspects such as varying the problem discretization become as simple as changing a single parameter in the problem specification, for example changing the order of the spatial discretization [43]. This article explains *what* can be accomplished with Devito, showing how to express real-life wave propagators as well as their integration within larger workflows typical of seismic exploration, such as the popular Full-Waveform Inverison (FWI) and Reverse-Time Migration (RTM) methods. The Devito compiler, and in particular *how* the user-provided SymPy equations are translated into high-performance C, are also briefly summarized, although for a complete description the interested reader should refer to [42].

The remainder of this paper is structured as follows: first, we provide a brief history of optimizing compilers, DSL and existing wave equation seismic frameworks. Next, we highlight the core features of Devito and describe the implementation of the featured wave equation operators in Section 3.3. We outline the seismic inversion theory in Section 3.4. Code verification and analysis of accuracy in Section 3.5 is followed by a discussion of the propagators computational performance in Section 3.6. We conclude by presenting a set of realistic examples such as seismic inversion and computational fluid dynamics and a discussion of future work.

3.2 Background

Improving the runtime performance of a critical piece of code on a particular computing platform is a non-trivial task that has received significant attention throughout the history of computing. The desire to automate the performance optimization process itself across a range of target architectures is not new either, although it is often met with skepticism. Even the very first compiler, A0 [114], was received with resistance, as best summarized in the following quote: *"Dr. Hopper believes,..., that the result of a compiling technique should be a routine just as efficient as a hand tailored routine. Some others do not completely agree with this. They feel the machine-made routine can approach hand tailored coding, but they believe there are "tricks of the trade" that apply to various special cases that a computer cannot be expected to utilize."* [115]. Given the challenges of porting optimized codes to a wide range of rapidly evolving computer architectures, it seems natural to raise again the layer of abstraction and use compiler techniques to replace much of the manual labor.

Community acceptance of these new "automatic coding systems" began when concerns about the performance of the generated code were addressed by the first "optimizing compiler", FORTRAN, released in 1957 – which not only translated code from one language to another but also ensured that the final code performed at least as good as a hand-written low-level code [116]. Since then, as program and hardware complexity rose, the same problem has been solved over and over again, each time by the introduction of higher levels of abstractions. The first high-level languages and compilers were targeted at solving a large variety of problems and hence were restricted in the kind of optimizations they could leverage. As these generic languages became common-place and the need for further improvement in performance was felt, restricted languages focusing on smaller problem domains were developed that could leverage more "tricks of the trade" to optimize performance. This led to the proliferation of DSLs for broad mathematical domains or sub-domains, such as APL [117], Mathematica, Matlab®or R.

In addition to these relatively general mathematical languages, more specialized frameworks targeting the automated solution of PDEs have long been of interest [118, 119, 120, 121]. More recent examples not only aim to encapsulate the high-level notation of PDEs, but are often centered around a particular numerical method. Two prominent contemporary projects based on the finite element method (FEM), FEniCS [31] and Firedrake [32], both implement a common DSL, UFL [122], that allows the expression of variational problems in weak form. Multiple DSLs to express stencil-like algorithms have also emerged over time, including [112, 123, 124, 125, 126, 127, 128, 129, 130, 113]. Other stencil DSLs have been developed with the objective of solving PDEs using finite differences [49], [46] and [131]. However, in all cases their use in seismic imaging problems (or even more broadly in science and engineering) has been limited by a number of factors other than technology inertia. Firstly, they only raise the abstraction to the level of polyhedral-like (affine) loops. As they do not generally use a symbolic mathematics engine to write the mathematical expressions at a high-level, developers must still write potentially complex numerical kernels in the target low-level programming language. For complex formulations this process can be time consuming and error prone, as hand-tuned codes for wave propagators can reach thousands of lines of code. Secondly, most DSLs rarely offer enough flexibility for extension beyond their original scope (e.g. sparse operators for source terms and interpolation) making it difficult to work the DSL into a more complex science/engineering workflow. Finally, since finite difference wave propagators only form part of the over-arching PDE constrained (wave equation) optimization problem, composability with external packages, such as the *SciPy* optimization toolbox, is a key requirement that is often ignored by self-contained standalone DSLs. The use of a fully embedded *Python* DSL, on the other hand, allows users to leverage a variety of higher-level optimization techniques through a rich variety of software packages provided by the scientific *Python* ecosystem.

Moreover, several computational frameworks for seismic imaging exist, although they provide varying degrees of abstraction and are typically limited to a single representation of the wave equation. IWAVE [132, 133, 134, 111], although not a DSL, provides a high-level of abstraction and a mathematical framework to abstract the algebra related to the wave-equation and its solution. IWAVE provides a rigorous mathematical abstraction for linear operations and vector representations including Hilbert space abstraction for norms and distances. However, its C++ implementation limits the extensibility of the framework to new wave-equations. Other software frameworks, such as Madagascar [12], offer a broad range of applications. Madagascar is based on a set of subroutines for each individual problem and offers modelling and imaging operators for multiple wave-equations. However, the lack of high-level abstraction restricts its flexibility and interfacing with high level external software (i.e *Python* , *Java*). The subroutines are also mostly written in C/Fortran and limit the architecture portability.

3.3 Symbolic definition of finite difference stencils with Devito

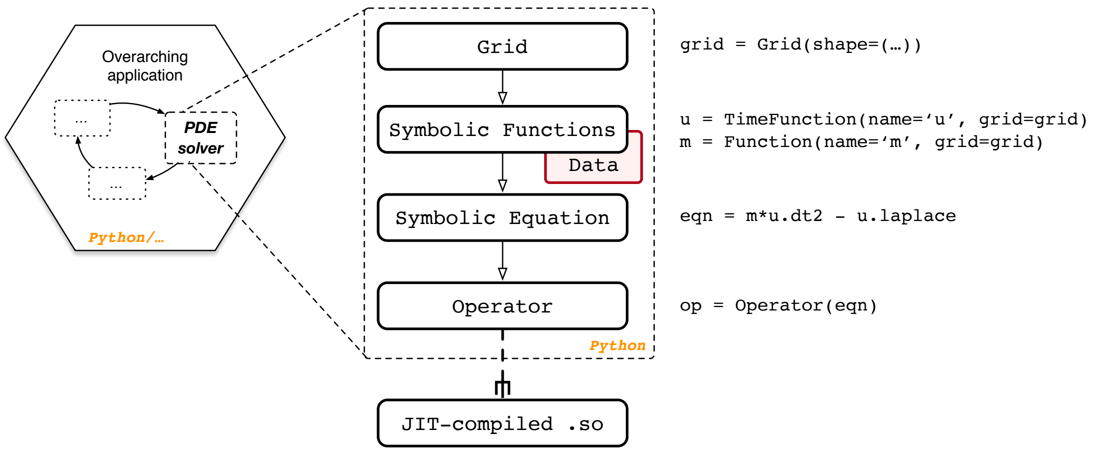
In general, the majority of the computational workload in wave-equation based seismic inversion algorithms arises from computing solutions to discrete wave equations and their adjoints. There are a wide range of mathematical models used in seismic imaging that approximate the physics to varying degrees of fidelity. To improve development and innovation time, including code verification, we describe the use of the symbolic finite difference framework Devito to create a set of explicit matrix-free operators of arbitrary spatial discretization order. These operators are derived for example from the acoustic wave equation

$$m(x)\frac{\partial^2 u(t,x)}{\partial t^2} - \Delta u(t,x) = q(t,x), \tag{3.1}$$

where $m(x) = \frac{1}{c(x)^2}$ is the squared slowness with $c(x)$ the spatially dependent speed of sound, symbol $\Delta u(t,x)$ denotes the Laplacian of the wavefield $u(t,x)$ and $q(t,x)$ is a source usually located at a single location x_s in space ($q(t,x) = f(t)\delta(x_s)$). This formulation will be used as running example throughout the section.

3.3.1 Code generation - an overview

Devito aims to combine performance benefits of dedicated stencil frameworks [130, 129, 112, 113] with the expressiveness of symbolic PDE-solving DSLs [31, 32] through automated code generation and optimization from high-level symbolic expressions of the mathematics. Thus, the primary design objectives of the Devito DSL are to allow users to define explicit finite difference operators for (time-dependent) PDEs in a concise symbolic manner and provide an API that is flexible enough to fully support realistic scientific use cases. To this end, Devito offers a set of symbolic classes that are fully compatible with the general-purpose symbolic algebra package *SymPy* that enables users to derive discretized stencil expressions in symbolic form. As we show in Figure 3.1, the primary symbols in such expressions are associated with user data that carry domain-specific meta-data infor-



```
grid = Grid(shape=(…))

u = TimeFunction(name='u', grid=grid)
m = Function(name='m', grid=grid)

eqn = m*u.dt2 - u.laplace

op = Operator(eqn)
```

Figure 3.1: Overview of the Devito architecture and the associated example workflow. Devito's top-level API allows users to generate symbolic stencil expressions using data-carrying function objects that can be used for symbolic expressions via *SymPy* . From this high-level definition, an operator then generates, compiles and executes high-performance C code.

mation to be used by the compiler engine (e.g. dimensions, data type, grid). The discretized expressions form an abstract operator definition that Devito uses to generate low-level C code (C99) and OpenMP at runtime. The encapsulating `Operator` object can be used to execute the generated code from within the *Python* interpreter making Devito natively compatible with the wide range of tools available in the scientific *Python* software stack. We manage memory using our own allocators (e.g. to enforce alignment and NUMA optimizations) and therefore we also take control over freeing memory. We wrap everything with the NumPy array API to ensure interoperability with other modules that use NumPy.

A Devito `Operator` takes as input a collection of symbolic expressions and progressively lowers the symbolic representation to semantically equivalent C code. The code generation process consists of a sequence of compiler passes during which multiple automated performance-optimization techniques are employed. These can be broadly categorized into two classes and are performed by distinct sub-packages:

- **Devito Symbolic Engine (DSE):** Symbolic optimization techniques, such as Common Sub-expression Elimination (CSE), factorization and loop-invariant code motion are utilized to reduce the number of floating point operations (flops) performed within the computational kernel [135]. These optimization techniques are inspired by *SymPy* but are custom implemented in Devito and do not rely on *SymPy* implementation of CSE for example.

- **Devito Loop Engine (DLE):** Well-known loop optimization techniques, such as explicit vectorization, thread-level parallelization and loop blocking with auto-tuned block sizes are employed to increase the cache utilization and thus memory bandwidth utilization of the kernels.

A complete description of the compilation pipeline is provided in [42].

3.3.2 Discrete function symbols

The primary user-facing API of Devito allows the definition of complex stencil operators from a concise mathematical notation. For this purpose, Devito relies strongly on *SymPy* (Devito 3.1.0 depends upon SymPy 1.1 and all dependency versions are specified in Devito's requirements file). Devito provides two symbolic object types that mimic *SymPy* symbols, enabling the construction of stencil expressions in symbolic form:

- **Function:** The primary class of symbols provided by Devito behaves like `sympy.Function` objects, allowing symbolic differentiation via finite difference discretization and general symbolic manipulation through *SymPy* utilities. Symbolic function objects encapsulate state variables (parameters and solution of the PDE) in the operator definition and associated user data (function value) with the represented symbol. The metadata, such as grid information and numerical type, which provide domain-specific information to the Devito compiler are also carried by the `sympy.Function` object.

- **Dimension:** Each `sympy.Function` object defines an iteration space for stencil operations through a set of `Dimension` objects that are used to define and generate the corresponding loop structure from the symbolic expressions.

In addition to `sympy.Function` and `Dimension` symbols, Devito supplies the construct `Grid`, which encapsulates the definition of the computational domain and defines the discrete shape (number of grid points, grid spacing) of the function data. The number of spatial dimensions is hereby derived from the shape of the `Grid` object and inherited by all `Function` objects, allowing the same symbolic operator definitions to be used for two and three-dimensional problem definitions. As an example, a two-dimensional discrete representation of the square slowness of an acoustic wave $\vec{m}[x, y]$ inside a 5 by 6 grid points domain can be created as a symbolic function object as illustrated in Figure 3.2.

```
>>> grid = Grid(shape=(5, 6))
>>> m = Function(name='m',
     grid=grid)

>>> m
m(x, y)

>>> m.data.shape
(5, 6)
```

Figure 3.2: Defining a Devito `Function` on a `Grid`.

It is important to note here that $\vec{m}[x, y]$ is constant in time, while the discrete wavefield $\vec{u}[t, x, y]$ is time-dependent. Since time is often used as the stepping dimension for buffered stencil operators, Devito provides an additional function type `TimeFunction`, which automatically adds a special `TimeDimension` object to the list of dimensions. `TimeFunction` objects derive from `Function` with an extra time dimension and inherit all the symbolic properties. The creation of a `TimeFunction` requires the same parameters as a `Function`, with an extra optional `time_order` property that defines the discretization order for the time dimension and an integer `save` parameter that defines the size of the time axis when the full time history of the field is stored in memory. In the case of a buffered time dimension `save` is equal to `None` and the size of the buffered dimension is automatically inferred from the `time_order` value. As an example, we can create an equivalent symbolic representation of the wavefield as `u =`

`TimeFunction(name='u', grid=grid)`, which is denoted symbolically as `u(t, x, y)`.

Spatial discretization

The symbolic nature of the function objects allows the automatic derivation of discretized finite difference expressions for derivatives. Devito `Function` objects provide a set of shorthand notations that allow users to express, for example, $\frac{d\vec{u}[t,x,y,z]}{dx}$ as `u.dx` and $\frac{d^2\vec{u}[t,x,y,z]}{dx^2}$ as `u.dx2`. Moreover, the discrete Laplacian, defined in three dimensions as $\Delta\vec{u}[t,x,y,z] = \frac{d^2\vec{u}[t,x,y,z]}{dx^2} + \frac{d^2\vec{u}[t,x,y,z]}{dy^2} + \frac{d^2\vec{u}[t,x,y,z]}{dz^2}$ can be expressed in shorthand simply as `u.laplace`. The shorthand expression `u.laplace` is agnostic to the number of spatial dimensions and may be used for two or three-dimensional problems.

The discretization of the spatial derivatives can be defined for any order. In the most general case, we can write the spatial discretization in the x direction of order k (and equivalently in the y and z direction) as:

$$\frac{\partial^2\vec{u}[t,x,y,z]}{\partial x^2} = \frac{1}{h_x^2}\sum_{j=0}^{\frac{k}{2}}\left[\alpha_j(\vec{u}[t,x+jh_x,y,z] + \vec{u}[t,x-jh_x,y,z])\right], \tag{3.2}$$

where h_x is the discrete grid spacing for the dimension x, the constants α_j are the coefficients of the finite difference scheme and the spatial discretization error is of order $O(h_x^k)$.

Temporal discretization

We consider here a second-order time discretization for the acoustic wave equation, as higher order time discretization requires us to rewrite the PDE [136]. The discrete second-order time derivative with this scheme can be derived from the Taylor expansion of the discrete wavefield $\vec{u}(t, x, y, z)$ as:

$$\frac{d^2\vec{u}[t, x, y, z]}{dt^2} = \frac{\vec{u}[t + \Delta t, x, y, z] - 2\vec{u}[t, x, y, z] + \vec{u}[t - \Delta t, x, y, z]}{\Delta t^2}. \tag{3.3}$$

In this expression, Δt is the size of a discrete time step. The discretization error is $O(\Delta t^2)$ (second order in time) and will be verified in Section 3.5.

Following the convention used for spatial derivatives, the above expression can be automatically generated using the shorthand expression `u.dt2`. Combining the temporal and spatial derivative notations, and ignoring the source term q, we can now define the wave propagation component of Equation 3.1 as a symbolic expression via `Eq(m * u.dt2 - u.laplace, 0)` where `Eq` is the *SymPy* representation of an equation. In the resulting expression, all spatial and temporal derivatives are expanded using the corresponding finite difference terms. To define the propagation of the wave in time, we can now rearrange the expression to derive a stencil expression for the forward stencil point in time, $\vec{u}(t + \Delta t, x, y, z)$, denoted by the shorthand expression `u.forward`. The forward stencil corresponds to the explicit Euler time-stepping that updates the next time-step `u.forward` from the two previous ones `u` and `u.backward` (Equation 3.4). We use the *SymPy* utility `solve` to automatically derive the explicit time-stepping scheme, as shown in Figure 3.3 for the second order in space discretization.

$$\vec{u}[t + \Delta t, x, y, z] = 2\vec{u}[t, x, y, z] - \vec{u}[t - \Delta t, x, y, z] + \frac{\Delta t^2}{\vec{m}[x, y, z]}\Delta\vec{u}[t, x, y, z]. \tag{3.4}$$

The iteration over time to obtain the full solution is then generated by the Devito compiler from the time dimension information. Solving the wave-equation with the above explicit Euler scheme is equivalent to a linear system $\mathbf{A}(\vec{m})\vec{u} = \vec{q}_s$ where the vector $\vec{u}$ is the discrete wavefield solution of the discrete wave-equation, $\vec{q}_s$ is the source term and $\mathbf{A}(\vec{m})$ is the matrix representation of the discrete wave-equation. From Equation 3.4 we can see

```
>>> from sympy import Eq, solve, init_printing, pprint
>>> init_printing(use_latex=True)
>>> from devito import Function, TimeFunction, Grid

>>> grid = Grid(shape=(5, 5))
>>> u = TimeFunction(name='u', grid=grid, space_order=2, time_order=2)
>>> m = Function(name='m',grid=grid)

>>> eqn = Eq(m * u.dt2 - u.laplace)
>>> stencil = solve(eqn, u.forward)[0]
>>> pprint(Eq(u.forward, stencil))
```

Produces output equivalent to:

$$u(t+s,x,y) = 2u(t,x,y) - u(t-s,x,y) \\ \left. \begin{matrix} -\frac{2s^2u(t,x,y)}{h_y^2 m(x,y)} + \frac{s^2u(t,x,y-h_y)}{h_y^2 m(x,y)} + \frac{s^2u(t,x,y+h_y)}{h_y^2 m(x,y)} \\ -\frac{2s^2u(t,x,y)}{h_x^2 m(x,y)} + \frac{s^2u(t,x-h_x,y)}{h_x^2 m(x,y)} + \frac{s^2u(t,x+h_x,y)}{h_x^2 m(x,y)} \end{matrix} \right\} \frac{\Delta t^2 \Delta u}{m(x,y)}$$

Figure 3.3: Example code defining the two-dimensional wave equation without damping using Devito symbols and symbolic processing utilities from *SymPy* . Assuming $h_x = \Delta x$, $h_y = \Delta y$ and $s = \Delta t$ the output is equivalent to Equation 3.1 without the source term $\vec{q}_s$.

that the matrix $\mathbf{A}(\vec{m})$ is a lower triangular matrix that reflects the time-marching structure of the stencil. Simulation of the wavefield is equivalent to a forward substitution (solve row by row from the top) on the lower triangular matrix $\mathbf{A}(\vec{m})$. Since we do not consider complex valued PDEs, the adjoint of $\mathbf{A}(\vec{m})$ is equivalent to its transpose denoted as $\mathbf{A}^\top(\vec{m})$ and is an upper triangular matrix. The solution $\vec{v}$ of the discrete adjoint wave-equation $\mathbf{A}(\vec{m})^\top\vec{v} = \vec{q}_a$ for an adjoint source $\vec{q}_a$ is equivalent to a backward substitution (solve from the bottom row to top row) on the upper triangular matrix $\mathbf{A}(\vec{m})^\top$ and is simulated backward in time starting from the last time-step. These matrices are never explicitly formed, but are instead matrix free operators with implicit implementation of the matrix-vector product, $\vec{u} = \mathbf{A}(\vec{m})^{-1}\vec{q}_s$ as a forward stencil. The stencil for the adjoint wave-equation in this self-adjoint case would simply be obtained with `solve(eqn, u.backward)` and the compiler will detect the backward-in-time update.

Boundary conditions

The field recorded data is measured on a wavefield that propagates in an infinite domain. However, solving the wave equation in a discrete infinite domain is not feasible with finite differences. In order to mimic an infinite domain, Absorbing Boundary Conditions (ABC) or Perfectly Matched Layers (PML) are necessary [137]. These two methods allow the approximation of the wavefield as it is in an infinite medium by damping and absorbing the waves within an extra layer at the limit of the domain to avoid unnatural reflections from the edge of the discrete domain.

The least computationally expensive method is the Absorbing Boundary Condition that adds a single damping mask in a finite layer around the physical domain. This absorbing condition can be included in the wave-equation as:

$$\vec{m}[x,y,z]\frac{d^2\vec{u}[t,x,y,z]}{dt^2} - \Delta\vec{u}[t,x,y,z] + \vec{\eta}[x,y,z]\frac{d\vec{u}[t,x,y,z]}{dt} = 0. \tag{3.5}$$

The $\vec{\eta}[x,y,z]$ parameter is equal to 0 inside the physical domain and increasing from inside to outside within the damping layer. The dampening parameter $\vec{\eta}$ can follow a linear or exponential curve depending on the frequency band and width of the dampening layer. For methods based on more accurate modelling, for example in simulation-based acquisition design [138, 139, 140, 141], a full implementation of the PML will be necessary to avoid weak reflections from the domain limits.

Sparse point interpolation

Seismic inversion relies on data fitting algorithms, hence we need to support sparse operations such as source injection and wavefield ($\vec{u}[t,x,y,z]$) measurement at arbitrary grid locations. Both operations occur at sparse domain points, which do not necessarily align with the logical cartesian grid used to compute the discrete solution $\vec{u}(t,x,y,z)$. Since such operations are not captured by the finite differences abstractions for implementing PDEs, Devito implements a secondary high-level representation of sparse objects [142] to create a set of *SymPy* expressions that perform polynomial interpolation within the containing grid cell from pre-defined coefficient matrices.

The necessary expressions to perform interpolation and injection are automatically generated through a dedicated symbol type, `SparseFunction`, which associates a set of coordinates with the symbol representing a set of non-aligned points. For examples, the syntax `p.interpolate(expr)` provided by a `SparseFunction p` will generate a symbolic expressions that interpolates a generic expression `expr` onto the sparse point locations defined by `p`, while `p.inject(field, expr)` will evaluate and add `expr` to each enclosing point in `field`. The generated *SymPy* expressions are passed to Devito `Operator` objects alongside the main stencil expression to be incorporated into the generated C kernel code. A complete setup of the acoustic wave equation with absorbing boundaries, injection of a source function and measurement of wavefields via interpolation at receiver locations can be found in Section 3.4.2.

3.4 Seismic modeling and inversion

Seismic inversion methods aim to reconstruct physical parameters or an image of the earth's subsurface from multi-experiment field measurements. For this purpose, a wave is generated at the ocean surface that propagates through to the subsurface and creates reflections at the discontinuities of the medium. The reflected and transmitted waves are then captured by a set of hydrophones that can be classified as either moving receivers (cables dragged behind a source vessel) or static receivers (ocean bottom nodes or cables). From the acquired data, physical properties of the subsurface such as wave speed or density can be reconstructed by minimizing the misfit between the recorded measurements and the numerically modelled seismic data.

3.4.1 Full-Waveform Inversion

Recovering the wave speed of the subsurface from surface seismic measurements is commonly cast into a non-linear optimization problem called full-waveform inversion (FWI). The method aims at recovering an accurate model of the discrete wave velocity, $\vec{c}$, or alternatively, the square slowness of the wave, $\vec{m} = \frac{1}{\vec{c}^2}$(not an overload), from a given set of measurements of the pressure wavefield $\vec{u}$.In [35, 6, 17, 40] is shown that this can be expressed as a PDE-constrained optimization problem. After elimination of the PDE constraint, the reduced objective function is defined as:

$$\underset{\vec{m}}{\text{minimize}} \quad \Phi_s(\vec{m}) = \frac{1}{2} \left\| \mathbf{P}_r \vec{u} - \vec{d} \right\|_2^2 \quad \text{with:} \quad \vec{u} = \mathbf{A}(\vec{m})^{-1} \mathbf{P}_s^T \vec{q}_s, \tag{3.6}$$

where $\mathbf{P}_r$ is the sampling operator at the receiver locations, $\mathbf{P}_s^T$ (^{T} is the transpose or adjoint) is the injection operator at the source locations, $\mathbf{A}(\vec{m})$ is the operator representing the discretized wave equation matrix, $\vec{u}$ is the discrete synthetic pressure wavefield, $\vec{q}_s$ is the corresponding pressure source and $\vec{d}$ is the measured data. While we consider the acoustic isotropic wave equation for simplicity here, in practice, multiple implementations of the wave equation operator $\vec{A}(\vec{m})$ are possible depending on the choice of physics. In the most advanced case, $\vec{m}$ would not only contain the square slowness but also anisotropic or orthorhombic parameters.

To solve this optimization problem with a gradient-based method, we use the adjoint-state method to evaluate the gradient [41, 40]:

$$\nabla \Phi_s(\vec{m}) = \sum_{\vec{t}=1}^{n_t} \vec{u}[\vec{t}] \vec{v}_{tt}[\vec{t}] = \mathbf{J}^T \delta \vec{d}_s, \tag{3.7}$$

where n_t is the number of computational time steps, $\delta \vec{d}_s = \left(\mathbf{P}_r \vec{u} - \vec{d} \right)$ is the data residual (difference between the measured data and the modeled data), $\mathbf{J}$ is the Jacobian operator and $\vec{v}_{tt}$ is the second-order time derivative of the adjoint wavefield that solves:

$$\mathbf{A}^T(\vec{m})\vec{v} = \mathbf{P}_r^T \delta\vec{d}_s. \tag{3.8}$$

The discretized adjoint system in Equation 3.8 represents an upper triangular matrix that is solvable by modelling wave propagation backwards in time (starting from the last time step). The adjoint state method, therefore, requires a wave equation solve for both the forward and adjoint wavefields to compute the gradient. An accurate and consistent adjoint model for the solution of the optimization problem is therefore of fundamental importance.

3.4.2 Acoustic forward modelling operator

We consider the acoustic isotropic wave-equation parameterized in terms of slowness $\vec{m}[x, y, z]$ with zero initial conditions assuming the wavefield does not have any energy before zero time. We define an additional dampening term to mimic an infinite domain (see Section 3.3.2). At the limit of the domain, the zero Dirichlet boundary condition is satisfied as the solution is considered to be fully damped at the limit of the computational domain. The PDE is defined in Equation 3.5. Figure 3.4 demonstrates the complete set up of the acoustic wave equation with absorbing boundaries, injection of a source function and sampling wavefields at receiver locations. The shape of the computational domain is hereby provided by a utility object `model`, while the damping term $\eta \frac{d\vec{u}[x,y,z,t]}{dt}$ is implemented via a utility symbol `eta` defined as a `Function` object. It is important to note that the discretization order of the spatial derivatives is passed as an external parameter `order` and carried as meta-data by the wavefield symbol `u` during construction, allowing the user to freely change the underlying stencil order.

The main (PDE) stencil expression to update the state of the wavefield is derived from the high-level wave equation expression `eqn = u.dt2 - u.laplace + damp*u.dt`

```python
def forward(model, source, receiver,
     space_order=2):
   m, eta = model.m, model.damp
   # Allocate wavefield and auxiliary fields
   u = TimeFunction(name='u', grid=model.grid,
       time_order=2,
                 space_order=space_order)

   # Derive stencil from symbolic equation
   eqn = m * u.dt2 - u.laplace + eta * u.dt
   stencil = solve(eqn, u.forward)
   update_u = Eq(u.forward, stencil)

   # Source injection and receiver interpolation
   src = source.inject(field=u.forward,
       expr=src * dt**2 / m)
   rec = receiver.interpolate(expr=u)

   op = Operator([update_u] + src + rec,
       subs=model.spacing_map)
```

Figure 3.4: Example definition of a forward operator.

using *SymPy* utilities as demonstrated before in Figure 3.3. Additional expressions for the injection of the wave source via the `SparseFunction` object `src` are then generated for the forward wavefield, where the source time signature is discretized onto the computational grid via the symbolic expression `src * dt**2 / m`. The weight $\frac{dt^2}{m}$ is derived from rearranging the discretized wave equation with a source as a right-hand-side similarly to the Laplacian in Equation 3.4. A similar expression to interpolate the current state of the wavefield at the receiver locations (measurement points) is generated through the `receiver` symbol. The combined list of stencils, a sum in Python that adds the different expressions that update the wavefield at the next time step, inject the source and interpolate at the receivers, is then passed to the `Operator` constructor alongside a definition of the spatial and temporal spacing $h_x, h_y, h_z, \Delta t$ provided by the `model` utility. Devito then transforms this list of stencil expressions into loops (inferred from the symbolic Functions), replaces all necessary constants by their values if requested, prints the generated C code and compiles it. The operator is finally callable in Python with `op.apply()`.

A more detailed explanation of the seismic setup and parameters such as the source and receiver terms in Figure 3.4 is covered in [143].

3.4.3 Discrete adjoint wave-equation and FWI gradient

To create the adjoint that pairs with the above forward modeling propagator we can make use of the fact that the isotropic acoustic wave equation is self-adjoint. This entails that for the implementation of the forward wave equation `eqn`, shown in Figure 3.5, only the sign of the damping term needs to be inverted, as the dampening time-derivative has to be defined in the direction of propagation ($\frac{\partial}{\partial n(t)}$). For the PDE stencil, however, we now rearrange the stencil expression to update the backward wavefield from the two next time steps as $\vec{v}[t-\Delta t, x, y, z] = f(\vec{v}[t, x, y, z], \vec{v}[t+\Delta t, x, y, z])$. Moreover, the role of the sparse point symbols has changed (Equation 3.8), so that we now inject time-dependent data at the receiver locations (`adj_src`), while sampling the wavefield at the original source location (`adj_rec`).

Based on the definition of the adjoint operator, we can now define a similar operator to update the gradient according to Equation 3.7. As shown in Figure 3.6, we can replace the expression to sample the wavefield at the original source location with an accumulative update of the gradient field `grad` via the symbolic expression `Eq(grad, grad - u * v.dt2)`.

To compute the gradient, the forward wavefield at each time step must be available which leads to significant memory requirements. Many methods exist to tackle this memory issue, but all come with their advantages and disadvantages. For instance, we implemented optimal checkpointing with the library Revolve [27] in Devito to drastically reduce the memory cost by only saving a partial time history and recomputing the forward wavefield when needed [29]. The memory reduction comes at an extra computational cost as optimal checkpointing requires $log(n_t) + 2$ extra PDE solves. Another method is boundary wavefield reconstruction [144, 145, 146] that saves the wavefield only at the boundary of the model, but still requires us to recompute the forward wavefield during the back-propagation. This boundary method has a reduced memory cost but necessitates the computation of the forward wavefield twice (one extra PDE solve), once to get the data

```
def adjoint(model, adj_src, adj_rec,
     space_order=2):
    m, eta = model.m, model.damp
    # Allocate wavefield and auxiliary fields
    v = TimeFunction(name='v', grid=model.grid,
                     time_order=2,
                          space_order=space_order)

    # Derive stencil from symbolic equation
    eqn = m * v.dt2 - v.laplace - eta * v.dt
    stencil = solve(eqn, v.backward)
    update_v = Eq(u.backward, stencil)

    # Receiver injection and adj-source
         interpolation
    src_a = adj_src.inject(field=v.backward,
                           expr=rec * dt**2 / m)
    rec_a = adj_rec.interpolate(expr=v)

    op = Operator([update_v] + src_a + rec_a,
                  subs=model.spacing_map)
```

Figure 3.5: Example definition of an adjoint operator.

```
def gradient(model, u, adj_src, space_order=2):
    m, eta = model.m, model.damp
    # Allocate wavefield and auxiliary fields
    v = TimeFunction(name='v', grid=model.grid,
              time_order=2,
                   space_order=space_order)
    grad = Function(name='grad', grid=model.grid)

    # Derive stencil from symbolic equation
    eqn = m * v.dt2 - v.laplace - eta * v.dt
    stencil = solve(eqn, v.backward)
    update_v = Eq(u.backward, stencil)

    # Receiver injection and gradient update
    src_a = adj_src.inject(field=v.backward,
                    expr=rec * dt**2 / m)
    update_grad = Eq(grad, grad - u * v.dt2)

    op = Operator([update_v] + src_a +
         update_grad,
                   subs=model.spacing)
```

Figure 3.6: Example definition of a gradient operator.

than a second time from the boundary values to compute the gradient.

3.4.4 FWI using Devito operators

At this point, we have a forward propagator to model synthetic data in Figure 3.4, the adjoint propagator for Equation 3.8 and the FWI gradient of Equation 3.7 in Figure 3.6. With these three operators, we show the implementation of the FWI objective and gradient with Devito in Figure 3.8. With the forward and adjoint/gradient operator defined for a given source, we only need to add a loop over all the source experiments and the reduction operation on the gradients (sum the gradient for each source experiment together). In practice, this loop over sources is where the main task-based or MPI based parallelization happens. The wave-equation propagator does use some parallelization with multithreading or domain decomposition but that parallelism requires communication. The parallelism over source experiment is task-based and does not require any communication between the separate tasks as the gradient for each source can be computed independently and reduced to obtain the full gradient. With the complete gradient summed over the source experiments,

```python
def fwi_gradient(model, op_fwd, op_grad):
    """
    Function to compute a single FWI gradient
    """
    u = TimeFunction(name='u', grid=model.grid,
                     space_order=order)
    grad = Function(name='grad', grid=model.grid)

    for i in nshots:
        # Update source location for each shot
        src.coordinates.data[0. :] =
             source_loc[i]

        # Run forward modelling operator
        op_fwd(u=u, src=src, rec=smooth_d)

        # Compute gradient from data residual and
        # update objective function
        residual = smooth_d.data[:] -
             true_d.data[:]
        objective +=
             .5*np.linalg.norm(residual)**2
        op_grad(rec=residual, u=u, m=model.m,
             grad=grad)

    return objective, grad.data
```

Figure 3.7: Definition of FWI gradient update.

```python
model = Model(...)
dt, nt = <timestepping parameters>

# Define source and receiver geometry
src = RickerSource(...)
rec = Receiver(...)

# Create forward and gradient operators
op_fwd = forward(model, src, rec, order)
op_grad = gradient(model, rec, order)

# Run FWI with gradient descent
for i in range(0, fwi_iterations):
    # Compute functional value and gradient
    # for the current model estimate
    phi, direction = fwi_gradient(model.m)

    # Artificial Step length for gradient descent
    alpha = .005 / np.max(direction)

    # Update the model estimate and inforce
    # minimum/maximum values
    m_updated = model.m.data - alpha*direction
    model.m.data[:] = box_constraint(m_updated)
```

Figure 3.8: FWI algorithm with linesearch.

we update the model with a simple fixed step length gradient update [69].

This FWI function in Figure 3.7 can then be included in any black-box optimization toolbox such as *SciPy* `optimize` to solve the inversion problem Equation 3.6. While black-box optimization methods aim to minimize the objective, there are no guarantees they find a global minimum because the objective is highly non-linear in m and other more sophisticated methods are required [147, 20, 148, 78].

3.5 Verification

Given the operators defined in Section 3.3 we now verify the correctness of the code generated by the Devito compiler. We first verify that the discretized wave equation satisfies the convergence properties defined by the order of discretization, and secondly we verify the correctness of the discrete adjoint and computed gradient.

3.5.1 Numerical accuracy

The numerical accuracy of the forward modeling operator (Figure 3.4) and the runtime achieved for a given spatial discretization order and grid size are compared to the analytical solution of the wave equation in a constant media. We define two measures of the accuracy that compare the numerical wavefield in a constant velocity media to the analytical solution:

- **Accuracy versus size**, where we compare the obtained numerical accuracy as a function of the spatial sampling size (grid spacing).
- **Accuracy versus time**, where we compare the obtained numerical accuracy as a function of runtime for a given physical model (fixed shape in physical units, variable grid spacing).

The measure of accuracy of a numerical solution relies on a hypothesis that we satisfy for these two tests:

- The domain is large enough and the propagation time small enough to ignore boundary related effects, i.e. the wavefield never reaches the limits of the domain.
- The source is located on the grid and is a discrete approximation of the Dirac to avoid spatial interpolation errors. This hypothesis guarantees the existence of the analytical and numerical solution for any spatial discretization [149].

Convergence in time We analyze the numerical solution against the analytical solution and verify that the error between these two decreases at a second order rate as a function of the time step size Δt. The velocity model is a $400\text{m} \times 400\text{m}$ domain with a source at the center. We compare the numerical solution to the analytical solution on Figure 3.9.

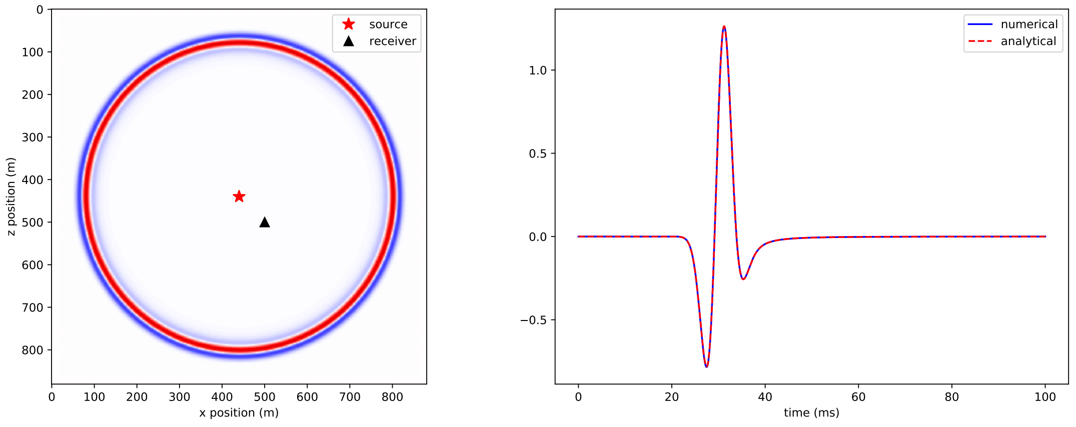


Figure 3.9: Numerical wavefield for a constant velocity $dt = .1\text{ms}$, $h = 1\text{m}$ and comparison with the analytical solution.

The analytical solution is defined as [150]:

$$u_s(r,t) = \frac{1}{2\pi}\int_{-\infty}^{\infty}\{-i\pi H_0^{(2)}(kr)\,q(\omega)e^{i\omega t}d\omega\} \tag{3.9}$$

$$r = \sqrt{(x - x_{src})^2 + (y - y_{src})^2}, \tag{3.10}$$

where $H_0^{(2)}$ is the Hankel function of second kind and $q(\omega)$ is the spectrum of the source function. As we can see on Figure 3.10 the error decreases near quadratically with the size of the time step with a time convergence rate of slope of 1.94 in logarithmic scale that matches the theoretical expectation from a second order temporal discretization.

Spatial discretization analysis The spatial discretization analysis follows the same method as the temporal discretixzation analysis. We model a wavefield for a fixed temporal setup with a small enough time-step to ensure negligeable time discretization error ($dt = .00625ms$). We vary the grid spacing (dx) and spatial discretization order and the and compute the error between the numerical and analytical solution. The convergence rates should follow the theoretical rates defined in Equation 3.2. In details, for a k^{th} order discretization in space, the error between the numerical and analytical solution should decrease as $O(dx^k)$.

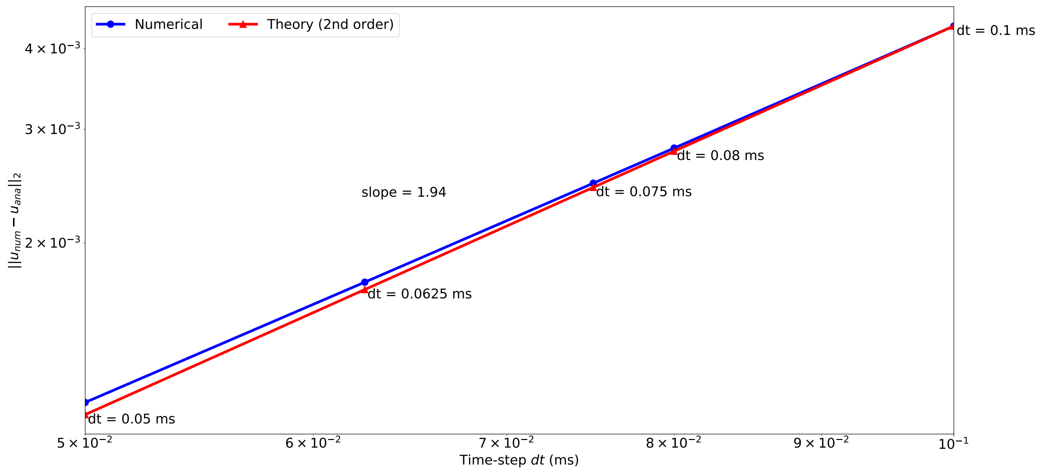


Figure 3.10: Time discretization convergence analysis for a fixed grid, fixed propagation time (150ms) and varying time step values. The result is plotted in a logarithmic scale and the numerical convergence rate (1.94 slope) shows that the numerical solution is accurate.

The best way to look at the convergence results is to plot the error in logarithmic scale and verify that the error decrease linearly with slope k. We show the convergence results on Figure 3.11. The numerical convergence rates follow the theoretical ones for every tested order $k = 2, 4, 6, 8$ with the exception of the 10^{th} order for small grid size. This is mainly due to reaching the limits of the numerical accuracy and a value of the error on par with the temporal discretization error. This behavior for high order and small grids is however in accordance with the literature as in in [151].

The numerical slopes obtained and displayed on Figure 3.11 demonstrate that the spatial finite difference follows the theoretical errors and converges to the analytical solution at the expected rate. These two convergence results (time and space) verify the accuracy and correctness of the symbolic discretization with Devito. With this validated simulated wavefield, we can now verify the implementation of the operators for inversion.

3.5.2 Propagators verification for inversion

We concentrate now on two tests, namely the adjoint test (or dot test) and the gradient test. The adjoint state gradient of the objective function defined in Equation 3.7 relies on the solutions of the forward and adjoint wave equations, therefore, the first mandatory property

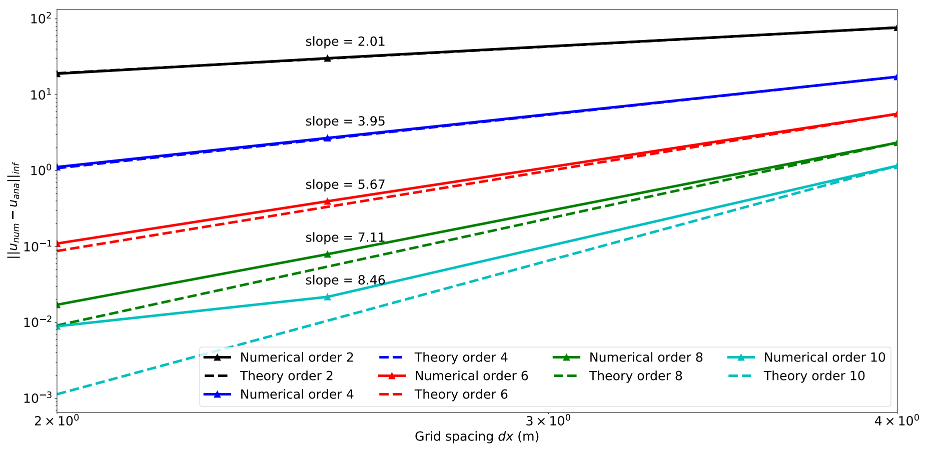


Figure 3.11: Comparison of the numerical convergence rate of the spatial finite difference scheme with the theoretical convergence rate from the Taylor theory. The theoretical rates are the dotted line with the corresponding colors. The result is plotted in a logarithmic scale to highlight the convergence orders as linear slopes and the numerical convergence rates show that numerical solution is accurate.

to verify is the exact derivation of the discrete adjoint wave equation. The mathematical test we use is the standard adjoint property or dot-test:

$$\begin{aligned}&\text{for any random } \vec{x} \in \text{span}(\vec{P}_s\vec{A}(\vec{m})^{-T}\vec{P}_r^{-T}), \vec{y} \in \text{span}(\vec{P}_r\vec{A}(\vec{m})^{-1}\vec{P}_s^{-T})\\ &\frac{< \vec{P}_r\vec{A}(\vec{m})^{-1}\vec{P}_s^{-T}\vec{x}, \vec{y} > - < \vec{x}, \vec{P}_s\vec{A}(\vec{m})^{-T}\vec{P}_r^{-T}\vec{y} >}{< \vec{P}_r\vec{A}(\vec{m})^{-1}\vec{P}_s^{-T}\vec{x}, \vec{y} >} = 0.0. \qquad (3.11)\end{aligned}$$

The adjoint test is also individually performed on the source/receiver injection/interpolation operators in the Devito tests suite. The results, summarized in Tables 3.1 and 3.2 with $\mathbf{F} = \vec{P}_r\vec{A}(\vec{m})^{-1}\vec{P}_s^{-T}$, verify the correct implementation of the adjoint operator for any order in both 2D and 3D. We observe that the discrete adjoint is accurate up to numerical precision for any order in 2D and 3D with an error of order $1e-16$. In combination with the previous numerical analysis of the forward modeling propagator that guarantees that we solve the wave equation, this result verifies that the adjoint propagator is the exact numerical adjoint of the forward propagator and that it implements the adjoint wave equation.

Table 3.1: Adjoint test for different discretization orders in 2D, computed on a two layer model in double precision.

Order	$< \mathbf{F}\vec{x}, \vec{y} >$	$< \vec{x}, \mathbf{F}^T\vec{y} >$	relative error
2nd order	7.9858e+05	7.9858e+05	0.0000e+00
4th order	7.3044e+05	7.3044e+05	0.0000e+00
6th order	7.2190e+05	7.2190e+05	4.8379e-16
8th order	7.1960e+05	7.1960e+05	4.8534e-16
10th order	7.1860e+05	7.1860e+05	3.2401e-16
12th order	7.1804e+05	7.1804e+05	6.4852e-16

Table 3.2: Adjoint test for different discretization orders in 3D, computed on a two layer model in double precision.

Order	$< \mathbf{F}\vec{x}, \vec{y} >$	$< \vec{x}, \mathbf{F}^T\vec{y} >$	relative error
2nd order	5.3840e+04	5.3840e+04	1.3514e-16
4th order	4.4725e+04	4.4725e+04	3.2536e-16
6th order	4.3097e+04	4.3097e+04	3.3766e-16
8th order	4.2529e+04	4.2529e+04	3.4216e-16
10th order	4.2254e+04	4.2254e+04	0.0000e+00
12th order	4.2094e+04	4.2094e+04	1.7285e-16

With the forward and adjoint propagators tested, we finally verify that the Devito operator that implements the gradient of the FWI objective function (Equation 3.7, Figure3.6) is accurate with respect to the Taylor expansion of the FWI objective function. For a given velocity model and associated squared slowness $\vec{m}$, the Taylor expansion of the FWI objective function from Equation 3.6 for a model perturbation $\vec{dm}$ and a perturbation scale h is:

$$\begin{aligned}
&\Phi_s(\vec{m} + h\vec{dm}) = \Phi_s(\vec{m}) + \mathcal{O}(h) \\
&\Phi_s(\vec{m} + h\vec{dm}) = \Phi_s(\vec{m}) + h\langle \nabla\Phi_s(\vec{m}), \vec{dm}\rangle + \mathcal{O}(h^2). \qquad (3.12)
\end{aligned}$$

These two equations constitute the gradient test where we define a small model perturbation

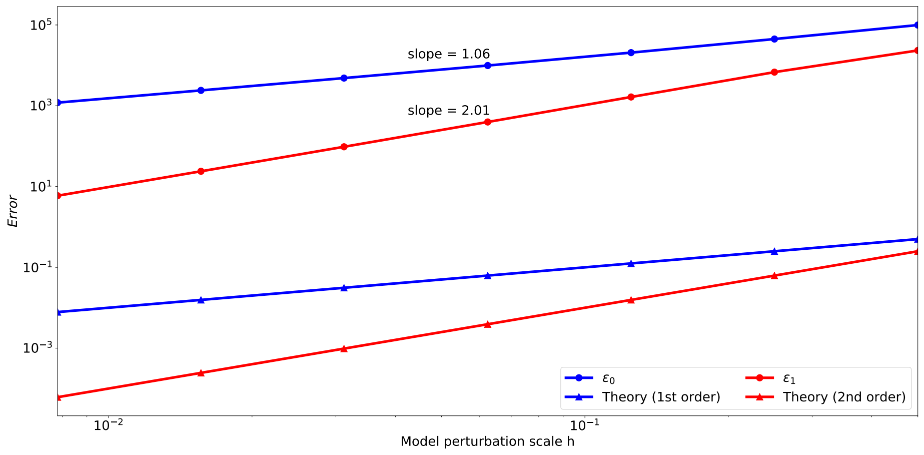


Figure 3.12: Gradient test for the acoustic propagator. The first order (blue) and second order (red) errors are displayed in logarithmic scales to highlight the slopes. The numerical convergence order (1.06 and 2.01) show that we have a correct implementation of the FWI operators.

$\vec{dm}$ and vary the value of h between 10^{-6} and 10^0 and compute the error terms:

$$\begin{aligned}\epsilon_0 =&\Phi_s(\vec{m} + h\vec{dm}) - \Phi_s(\vec{m})\\ \epsilon_1 =&\Phi_s(\vec{m} + h\vec{dm}) - \Phi_s(\vec{m}) - h\langle\nabla\Phi_s(\vec{m}), \vec{dm}\rangle.\end{aligned} \tag{3.13}$$

We plot the evolution of the error terms as a function of the perturbation scale h knowing ϵ_0 should be first order (linear with slope 1 in a logarithmic scale) and ϵ_1 should be second order (linear with slope 2 in a logarithmic scale). We executed the gradient test defined in Equation 3.12 in double precision with a 8^{th} order spatial discretization. The test can be run for higher orders in the same manner but since it has already been demonstrated that the adjoint is accurate for all orders, the same results would be obtained.

In Figure 3.12, the matching slope of the error term with the theoretical h and h^2 slopes from the Taylor expansion verifies the accuracy of the inversion operators. With all the individual parts necessary for seismic inversion, we now validate our implementation on a simple but realistic example.

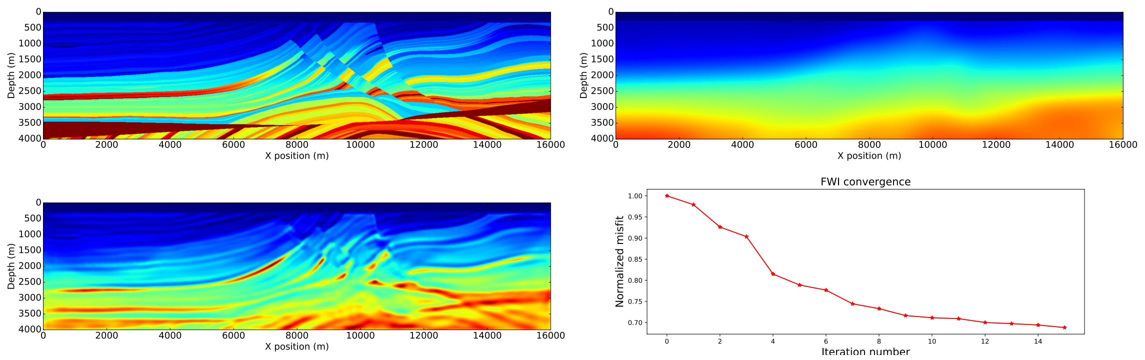


Figure 3.13: FWI on the acoustic Marmousi-ii model. The top-left plot is the true velocity model, the top-right is the initial velocity model, the bottom-left plot is the inverted velocity at the last iteration of the iterative inversion and the bottom-right plot is the convergence.

3.5.3 Validation: Full-Waveform Inversion

We show a simple example of FWI Equation 3.7 on the Marmousi-ii model [152]. This result obtained with the Julia interface to Devito JUDI [78, 153] that provides high-level abstraction for optimization and linear algebra. The model size is $4\text{km} \times 16\text{km}$ discretized with a 10m grid in both directions. We use a 10Hz Ricker wavelet with 4s recording. The receivers are placed at the ocean bottom (210m depth) every 10m. We invert for the velocity with all the sources, spaced by 50m at 10m depth for a total of 300 sources. The inversion algorithm used is minConf_PQN [154], an l-BFGS algorithm with bounds constraints (minimum and maximum velocity values constraints). While conventional optimization would run the algorithm to convergence, this strategy is computationally not feasible for FWI. As each iteration requires two PDE solves per source q_s (see adjoint state in Section 3.4), we can only afford a $\mathcal{O}(10)$ iterations in practice ($\mathcal{O}(10^4)$ PDE solves in total). In this example, we fix the number of function evaluations to 20, which, with the line search, corresponds to 15 iteration. The result is shown in Figure 3.13 and we can see that we obtain a good reconstruction of the true model. More advanced algorithms and constraints will be necessary for more complex problem such as less accurate initial model, noisy data or field recorded data [78, 148]; however the wave propagator would not be impacted, making this example a good proof of concept for Devito.

This result highlights two main contributions of Devito. First, we provide PDE simulation tools that allow easy and efficient implementation of inversion operator for seismic problem and potentially any PDE constrained optimization problem. As described in Section 3.3 and 3.4, we can implement all the required propagators and the FWI gradient in a few lines in a concise and mathematical manner. Second, as we obtained this results with JUDI [153], a seismic inversion framework that provides a high-level linear abstraction layer on top of Devito for seismic inversion, this example illustrates that Devito is fully compatible with external languages and optimizations toolboxes and allows users to use our symbolic DSL for finite difference within their own inversion framework.

3.5.4 Computational Fluid Dynamics

Finally we describe three classical computational fluid dynamics examples to highlight the flexibility of Devito for another application domain. Additional CFD examples can be found in the Devito code repository in the form of a set of Jupyter notebooks. The three examples we describe here are the convection equation, the Burger equation and the Poisson equation. These examples are adapted from [155] and the example repository contains both the original Python implementation with Numpy and the implementation with Devito for comparison.

Convection

The convection governing equation for a field u and a speed c in two dimensions is:

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} + c\frac{\partial u}{\partial y} = 0. \tag{3.14}$$

The same way we previously described it for the wave equation, u is then defined as a `TimeFunction`. In this simple case, the speed is a constant and does not need a symbolic representation, but a more general definition of this equation is possible with the creation of c as a Devito `Constant` that can accept any runtime value. We then discretized the

```
u = TimeFunction(name='u', grid=grid)

# Derive stencil from symbolic equation
eq = Eq(u.dt + c*u.dxl + c*u.dyl)
stencil = solve(eq, u.forward)

# Apply boundary conditions
u.data[:, 0, :] = 1.
u.data[:, -1, :] = 1.
u.data[:, :, 0] = 1.
u.data[:, :, -1] = 1.

# Create an Operator that updates the forward stencil
# point in the interior subdomain only.
op = Operator(Eq(u.forward, stencil,
     subdomain=grid.interior))
```

Figure 3.14: Convection equation in Devito. In this example, the initial Dirichlet boundary conditions are set to 1 using the API indexing feature, which allows to assign values to the `TensorFunction` data.

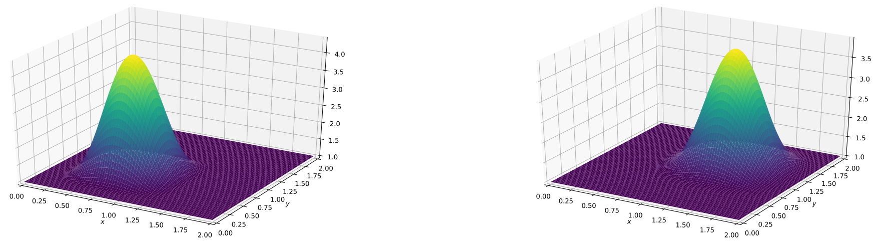


Figure 3.15: Initial (left) and final (right) time of the simulation of the convection equation.

PDE using forward differences in time and backward differences in space:

$$u_{i,j}^{n+1} = u_{i,j}^{n} - c\frac{\Delta t}{\Delta x}(u_{i,j}^{n} - u_{i-1,j}^{n}) - c\frac{\Delta t}{\Delta y}(u_{i,j}^{n} - u_{i,j-1}^{n}), \tag{3.15}$$

which is implemented in Devito as in Figure 3.14.

The solution of the convection equation is displayed on Figure 3.15 that shows the evolution of the field u and the solution is consistent with the expected result produced by [155].

Burgers' equation

In this second example, we show the solution of Burgers' equation. This example demonstrates that Devito supports coupled system of equations and non linear equations easily. The Burgers' equation in two dimensions is defined as the following coupled PDE system:

$$\begin{cases} \frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = \nu \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right), \\ \frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} = \nu \left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right), \end{cases} \tag{3.16}$$

where u, v are the two components of the solution and ν is the diffusion coefficient of the medium. The system of coupled equations is implemented in Devito in a few lines as shown in Figure 3.16.

We show the initial state and the solution at the last time step of the Burgers' equation in Figure 3.17. Once again, the solution corresponds to the reference solution of [155].

Poisson

We finally show the implementation of a solver for the Poisson equation in Devito. While the Poisson equation is not time dependent, the solution is obtained with an iterative solver and simplest one can easily be implemented with finite differences. The Poisson equation for a field p and a right hand side b is defined as:

$$\frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} = b, \tag{3.17}$$

and its solution can be computed iteratively with:

```
# Define our velocity fields and initialise with hat
     function
u = TimeFunction(name='u', grid=grid, space_order=2)
v = TimeFunction(name='v', grid=grid, space_order=2)
init_hat(field=u.data[0], dx=dx, dy=dy, value=2.)
init_hat(field=v.data[0], dx=dx, dy=dy, value=2.)

# Write down the equations with explicit backward
     differences
a = Constant(name='a')
u_dx = first_derivative(u, dim=x, side=left, order=1)
u_dy = first_derivative(u, dim=y, side=left, order=1)
v_dx = first_derivative(v, dim=x, side=left, order=1)
v_dy = first_derivative(v, dim=y, side=left, order=1)
eq_u = Eq(u.dt + u*u_dx + v*u_dy, a*u.laplace,
     subdomain=grid.interior)
eq_v = Eq(v.dt + u*v_dx + v*v_dy, a*v.laplace,
     subdomain=grid.interior)

# Let SymPy rearrange our stencils to form the update
     expressions
stencil_u = solve(eq_u, u.forward)
stencil_v = solve(eq_v, v.forward)
update_u = Eq(u.forward, stencil_u)
update_v = Eq(v.forward, stencil_v)

# Create Dirichlet BC expressions using the low-level
     API
bc_u = [Eq(u[t+1, 0, y], 1.)] # left
bc_u += [Eq(u[t+1, nx-1, y], 1.)] # right
bc_u += [Eq(u[t+1, x, ny-1], 1.)] # top
bc_u += [Eq(u[t+1, x, 0], 1.)] # bottom
bc_v = [Eq(v[t+1, 0, y], 1.)] # left
bc_v += [Eq(v[t+1, nx-1, y], 1.)] # right
bc_v += [Eq(v[t+1, x, ny-1], 1.)] # top
bc_v += [Eq(v[t+1, x, 0], 1.)] # bottom

# Create the operator
op = Operator([update_u, update_v] + bc_u + bc_v)
```

Figure 3.16: Burgers' equations in Devito. In this example, we use explicitly the FD function `first_derivative`. This function provides more flexibility and allows to take an upwind derivative, rather than a standard centered derivative (`dx`), to avoid odd-even coupling, which leads to chessboard artifacts in the solution.

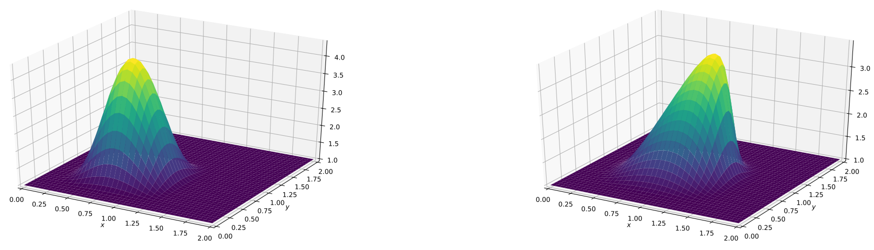


Figure 3.17: Initial (left) and final (right) time of the simulation of the Burgers' equations.

$$p_{i,j}^{n+1} = \frac{(p_{i+1,j}^{n} + p_{i-1,j}^{n})\Delta y^2 + (p_{i,j+1}^{n} + p_{i,j-1}^{n})\Delta x^2 - b_{i,j}^{n}\Delta x^2 \Delta y^2}{2(\Delta x^2 + \Delta y^2)}, \tag{3.18}$$

where the expression in Equation 3.18 is computed until either the number of iterations is reached (our example case) or more realistically when $||p_{i,j}^{n+1} - p_{i,j}^{n}|| < \epsilon$. We show two different implementations of a Poisson solver in Figure 3.18, 3.19. While these two implementations produce the same result, the second one takes advantage of Devito's `BufferedDimension` that allows to iterate automatically alternating between p^n and p^{n+1} as the two different time buffers in the `TimeFunction`.

The solution of the Poisson equation is displayed on Figure 3.20 with its right-hand-side b.

These examples demonstrate the flexibility of Devito and show that a broad range of PDE can easily be implemented with Devito including non linear equation, coupled PDE system and steady state problems.

3.6 Performance

In this section we demonstrate the performance of Devito from the numerical and the inversion point of view, as well as the absolute performance from the hardware point of view. This section only provides a brief overview of Devito's performance and a more detailed

```
p = Function(name='p', grid=grid, space_order=2)
pd = Function(name='pd', grid=grid, space_order=2)
p.data[:] = 0.
pd.data[:] = 0.

# Initialise the source term 'b'
b = Function(name='b', grid=grid)
b.data[:] = 0.
b.data[int(nx / 4), int(ny / 4)] = 100
b.data[int(3 * nx / 4), int(3 * ny / 4)] = -100

# Create Laplace equation base on 'pd'
eq = Eq(pd.laplace, b, subdomain=grid.interior)
# Let SymPy solve for the central stencil point
stencil = solve(eq, pd)
# Now we let our stencil populate our second buffer 'p'
eq_stencil = Eq(p, stencil)

# Create boundary condition expressions
x, y = grid.dimensions
t = grid.stepping_dim
bc = [Eq(p[x, 0], 0.)]
bc += [Eq(p[x, ny-1], 0.)]
bc += [Eq(p[0, y], 0.)]
bc += [Eq(p[nx-1, y], 0.)]

# Now we can build the operator that we need
op = Operator([eq_stencil] + bc)

# Run the outer loop explicitly in Python
for i in range(nt):
    # Determine buffer order
    if i % 2 == 0:
        _p = p
        _pd = pd
    else:
        _p = pd
        _pd = p

    # Apply operator
    op(p=_p, pd=_pd)
```

Figure 3.18: Poisson equation in Devito with field swap in Python.

```
# Now with Devito we will turn 'p' into 'TimeFunction'
# object to make all the buffer switching implicit
p = TimeFunction(name='p', grid=grid, space_order=2)

# Initialise the source term 'b'
b = Function(name='b', grid=grid)
b.data[:] = 0.
b.data[int(nx / 4), int(ny / 4)] = 100
b.data[int(3 * nx / 4), int(3 * ny / 4)] = -100

# Create Laplace equation base on 'p'
eq = Eq(p.laplace, b)
# Let SymPy solve for the central stencil point
stencil = solve(eq, p)
# Let our stencil populate the buffer 'p.forward'
eq_stencil = Eq(p.forward, stencil)

# Create boundary condition expressions
# Note that we now add an explicit "t + 1"
# for the time dimension.
bc = [Eq(p[t + 1, x, 0], 0.)]
bc += [Eq(p[t + 1, x, ny-1], 0.)]
bc += [Eq(p[t + 1, 0, y], 0.)]
bc += [Eq(p[t + 1, nx-1, y], 0.)]

# We can even switch performance logging back on,
# since we only require a single kernel invocation.
configuration['log-level'] = 'INFO'

# Create and execute the operator for nt iterations
op = Operator([eq_stencil] + bc)
op(time=nt)
```

Figure 3.19: Poisson equation in Devito with buffered dimension for automatic swap at each iteration.

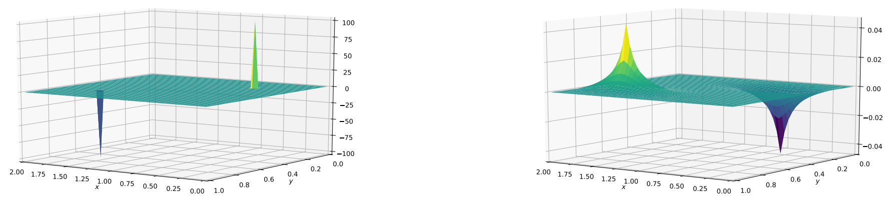


Figure 3.20: Right hand side (left) and solution (right) of the Poisson equations

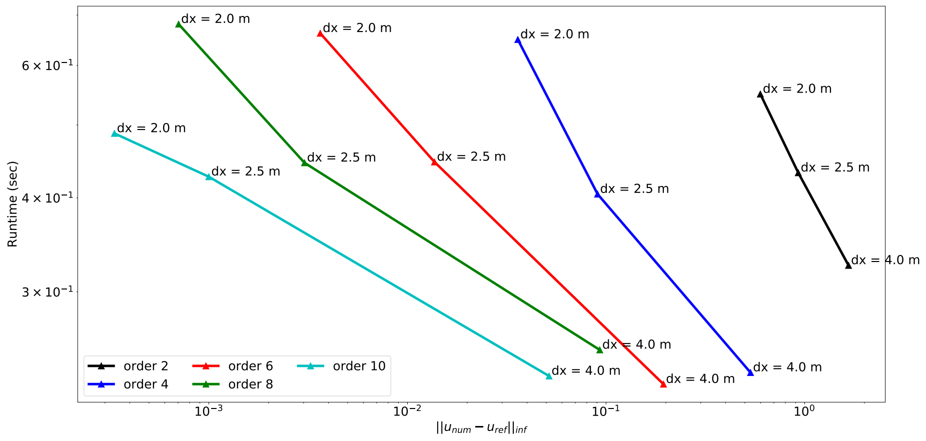


Figure 3.21: Different spatial discretization orders accuracy against runtime for a fixed physical setup (model size in m and propagation time).

description of the compiler and its performance is covered in [42].

3.6.1 Error-cost analysis

Devito's automatic code generation lets users define the spatial and temporal order of FD stencils symbolically and without having to reimplement long stencils by hand. This allows users to experiment with trade-offs between discretization errors and runtime, as higher order FD stencils provide more accurate solutions that come at increased runtime. For our error-cost analysis, we compare absolute error in L_2-norm between the numerical and the reference solution to the time-to-solution (the numerical and reference solution are defined in the previous Section 3.5). Figure 3.21 shows the runtime and numerical error obtained for a fixed physical setup. We use the same parameter as in Sections 3.5.1 with a domain of $400\mathrm{m} \times 400\mathrm{m}$ and we simulate the wave propagation for $150\mathrm{ms}$.

The results in Figure 3.21 illustrate that higher order discretizations produce a more accurate solution on a coarser grid with a smaller runtime. This result is very useful for inverse problems, as a coarser grid requires less memory and fewer time steps. A grid size two times bigger implies a reduction of memory usage by a factor of 2^4 for 3D modeling. Devito then allows users to design FD simulators for inversion in an optimal way, where the discretization order and grid size can be chosen according to the desired numerical accuracy and availability of computational resources. The order of the FD stencils also affects the best possible hardware usage that can theoretically be achieved and whether an algorithm is compute or memory bound, a trade-off that is described by the roofline model.

3.6.2 Roofline analysis

We present performance results of our solver using the roofline model, as previously discussed in [73, 75, 34, 74, 43]. Given a finite difference scheme, this method provides an estimate of the best achievable performance on the underlying architecture, as well as an absolute measure of the hardware usage. We also show a more classical metric, namely *time to solution*, in addition to the roofline plots, as both are essential for a clear picture of the achieved performance. The experiments were run on an Intel Skylake 8180 architecture (28 physical cores, 38.5 MB shared L3 cache, with cores operating at 2.5 Ghz). The observed Stream TRIAD [102] was 105 GB/s. The maximum single-precision FLOP performance was calculated as $\#\text{cores} \cdot \#\text{avx units} \cdot \#\text{data items per vector register} \cdot 2(\text{fused multiply-add}) \cdot \text{core frequency} = 4480\ \mathrm{GFLOPs/s}$. A (more realistic) performance peak of $3285\ \mathrm{GFLOPs/s}$ was determined by running the LINPACK benchmark [156]. These values are used to construct the roofline plots. In the following performance results, the operational intensity (OI) is computed by the Devito profiler from the symbolic expression after the compiler optimization. While the theoretical OI could be use, we chose to recompute it from the final optimized symbolic stencil for a more accurate performance measure. A more detailed overview of Devito's performance model is described in [42].

We show three different roofline plots, one plot for each domain size attempted, in Figure 3.22, 3.23 and 3.24. Different space orders are represented as different data points. The time-to-solution in seconds is annotated next to each data point. The experiments were run with all performance optimizations enabled. Because auto-tuning is used at runtime to determine the optimal loop-blocking structure, timing only commences after autotuning has finished. The reported operational intensity benefits from the use of expression transforma-

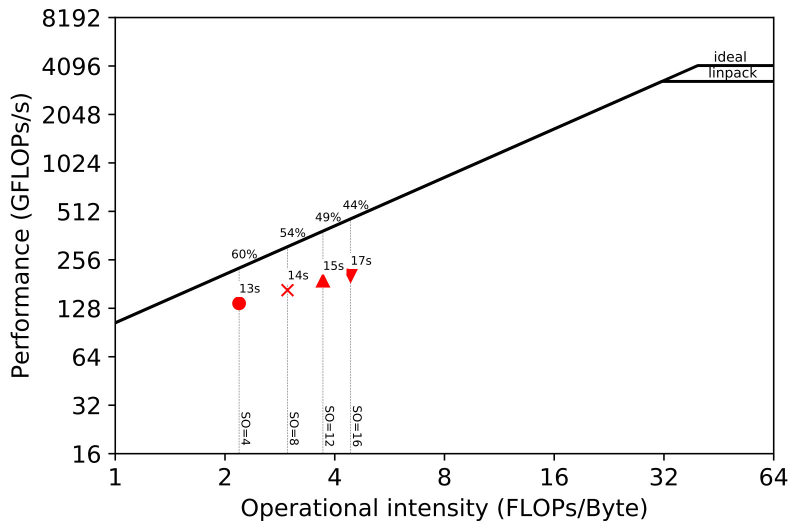


Figure 3.22: Roofline plots for a $512 \times 512 \times 512$ model on a Skylake 8180 architecture. The run times correspond to 1000ms of modeling for four different spatial discretization orders (4, 8, 12, 16).

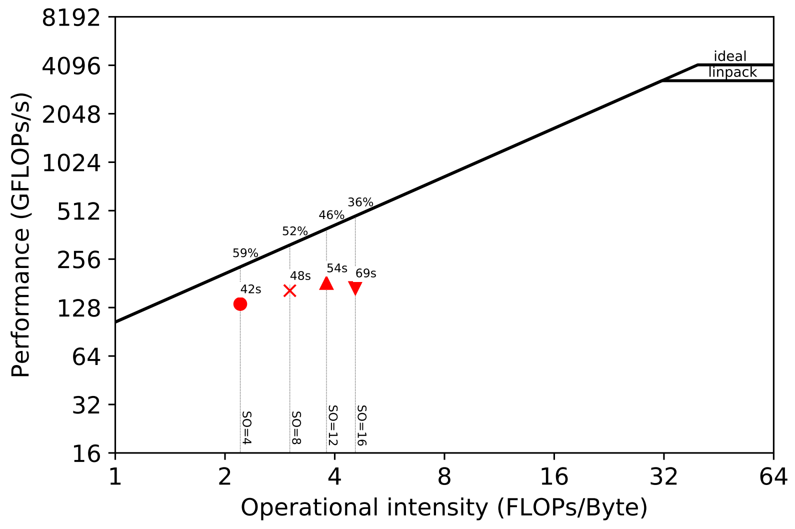


Figure 3.23: Roofline plots for a $768 \times 768 \times 768$ model on a Skylake 8180 architecture. The run times correspond to 1000ms of modeling for four different spatial discretization orders (4, 8, 12, 16).

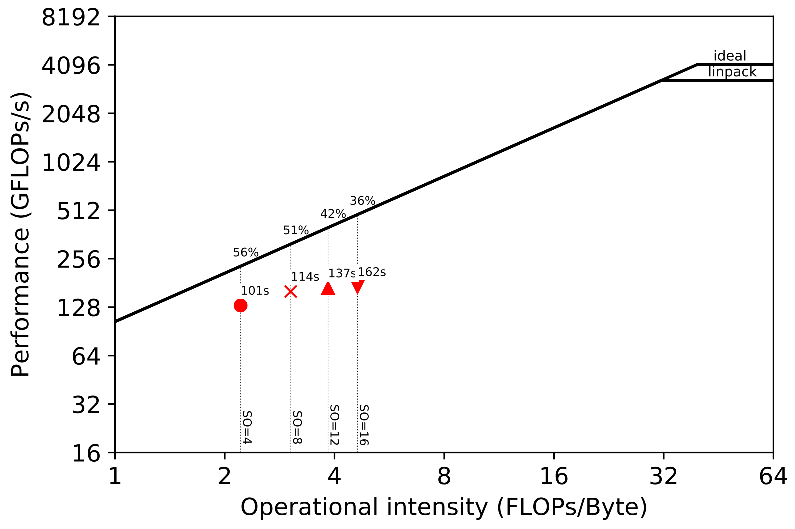


Figure 3.24: Roofline plots for a $1024 \times 1024 \times 1024$ model on a Skylake 8180 architecture. The run times correspond to $1000\mathrm{ms}$ of modeling for four different spatial discretization orders (4, 8, 12, 16).

tions as described in Section 3.3; particularly relevant for this problem is the factorization of FD weights.

We observe that the time to solution increases nearly linearly with the size of the domain. For example, for a 16th order discretization, we have a $17.1\mathrm{sec}$ runtime for a $512 \times 512 \times 512$ domain and $162.6\mathrm{sec}$ runtime for a $1024 \times 1024 \times 1024$ domain (8 times bigger domain and about 9 times slower). This is not surprising: the computation lies in the memory-bound regime and the working sets never fit in the L3 cache. We also note a drop in performance with a 16th order discretization (relative to both the other space orders and the attainable peak), especially when using larger domains (Figure 3.23 and 3.24). Our hypothesis, supported by profiling with Intel VTune [157], is that this is due to inefficient memory usage, in particular misaligned data accesses. Our plan to improve the performance in this regime consists of resorting to a specialized stencil optimizer such as YASK (see Section 3.7). These results show that we have a portable framework that achieves good performance on different architectures. There is small room for improvements, as the machine peak is still relatively distant, but 50-60% of the attainable peak is usually considered very good. Finally, we remark that testing on new architectures will only require extensions to the Devito compiler, if any, while the application code remains unchanged.

3.7 Future Work

A key motivation for developing an embedded DSL such as Devito is to enable quicker development, simpler maintenance, and better portability and performance of solvers. The other benefit of this approach is that HPC developer effort can be focused on developing the compiler technology that is reapplied to a wide range of problems. This software reuse is fundamental to keep the pace of technological evolution. For example, one of the current projects in Devito regards the integration of YASK [113], a lower-level stencil optimizer conceived by Intel for Intel architectures. Adding specialized backends such as YASK – meaning that Devito can generate and compile YASK code, rather than pure C/C++ – is the key for long-term performance portability, one of the goals that we are pursuing. Another motivation is to enable large scale computation and as many PDE is possible. In practice, this means that a staggered grid setup with half-node discretization and domain decomposition will be required. These two main requirements to extend the DSL to a broader community and applications is in full development and will be available in future releases.

3.8 conclusions

We have introduced a DSL for time-domain simulation for inversion and its application to a seismic inverse problem based on the finite difference method. Using the Devito DSL a highly optimized and parallel finite difference solver can be implemented within just a few lines of Python code. Although the current application focuses on features required for seismic imaging applications, Devito can already be used in problems based on other equations; a series of CFD examples are included in the code repository.

The code traditionally used to solve such problems is highly complex. The primary reason for this is that the complexity introduced by the mathematics is interleaved with the complexity introduced by performance engineering of the code to make it useful for practical use. By introducing a separation of concerns, these aspects are decoupled and both simplified. Devito successfully achieves this decoupling while delivering good computational performance and maintaining generality, both of which shall continue to be improved in future versions.

3.9 Code Availability

The code source code, examples and test script are available on github at `https://github.com/opesci/devito` and contains a README for installation. A more detailed overview of the project, list of publication and documentation of the software generated with Sphinx is available at `http://www.devitoproject.org/`. To install Devito:

```
git clone -b v3.1.0 https://github.com/opesci/devito
cd devito
conda env create -f environment.yml
source activate devito
pip install -e .
```

CHAPTER 4
DEVITO COMPILER

4.1 Introduction

Developing software for high-performance computing requires a considerable interdisciplinary effort, as it often involves domain knowledge from numerous fields such as physics, numerical analysis, software engineering and low-level performance optimization. The result is typically a monolithic application where hardware-specific optimizations, numerical methods, and physical approximations are interwoven and dispersed throughout a large number of loops, functions, files and modules. This frequently leads to slow innovation, high maintenance costs, and code that is hard to debug and port onto new computer architectures. A powerful approach to alleviate this problem is to introduce a separation of concerns and to raise the level of abstraction by using domain-specific languages (DSLs). DSLs can be used to express numerical methods using a syntax that closely mirrors how they are expressed mathematically, while a stack of compilers and libraries is responsible for automatically creating the optimized low-level implementation in a general purpose programming language such as C++. While the focus of this paper is on finite-difference (FD) based codes, the DSL approach has already had remarkable success in other numerical methods such as the finite-element (FE) and finite-volume (FV) method, as documented in Section 4.2.

This work describes the architecture of *Devito*, a system for automated stencil computations from a high-level mathematical syntax. Devito was developed with an emphasis on FD methods on structured grids. For this reason, Devito's underlying DSL has many features to simplify the specification of FD methods, as discussed in Section 4.3. The original motivation was to solve large-scale partial differential equations (PDEs) in the context of seismic inverse problems, where FD methods are commonly used for solving wave equations as part of complex workflows (e.g., data inversion using adjoint-state methods and backpropogation). Devito is equally useful as a framework for other stencil computations in general; for example, computations where all array indices are affine functions of loop variables. The Devito compiler is also capable of generating arbitrarily nested, possibly irregular, loops. This key feature is needed to support many complex algorithms that are used in engineering and scientific practice, including applications from image processing, cellular automata, and machine-learning.

One of the design goals of Devito was to enable high-productivity, so it is fully written in *Python* , with easy access to solvers, optimizers, input and output, and the wide range of other libraries in the *Python* ecosystem. At the same time, Devito transforms high-level symbolic input into optimized C++ code, resulting in a performance that is competitive with hand-optimized implementations. While the examples presented in this paper focus on using Devito from a *Python* application, exploiting the full potential of on-the-fly code generation and just-in-time (JIT) compilation, a practical advantage of generating C++ as an intermediate step is that it can be also used to generate libraries for legacy software, thus enabling incremental code modernization.

Compared to other DSL frameworks that are used in practice, Devito uses compiler technology, including several layers of intermediate representations, to perform optimizations in multiple passes. This allows Devito to perform more complex optimizations, and to better optimize the code for individual target platforms. The fact that these optimizations are performed programmatically facilitates performance portability across different computer architectures [158]. This is important, as industrial codes are often used on a variety of platforms, including clusters with multi-core CPUs, GPUs, and many-core chips spread across several compute nodes as well as various cloud platforms. Devito also performs high-level transformations for floating-point operation (FLOP) reduction based on symbolic manipulation, as well as loop-level optimizations as implemented in Devito's own optimizer, or using a third-party stencil compiler such as YASK [159]. The Devito compiler is presented in detail in Sections 4.4, 4.5 and 4.6.

After the presentation of the Devito compiler, we show test cases in Section 4.7 that are inspired by real-world seismic-imaging problems. The paper finishes with directions for future work and conclusions in Sections 4.8 and 4.9.

4.2 Related work

The objective of maximizing productivity and performance through frameworks based upon DSLs has long been pursued. In addition to well-known systems such as Mathematica® and Matlab®, which span broad mathematical areas, there are a number of tools specialized in numerical methods for PDEs, some dating back to the 1970s [118, 119, 120, 121].

4.2.1 DSL-based frameworks for partial differential equations

One noteworthy contemporary framework centered on DSLs is FEniCS [31], which allows the specification of weak variational forms, via UFL [33], and finite-element methods, through a high-level syntax. Firedrake [32] implements the same languages as FEniCS, although it differs from it in a number of features and architectural choices. Devito is heavily influenced by these two successful projects, in particular by their philosophy and design. Since solving a PDE is often a small step of a larger workflow, the choice of *Python* to implement these software provides access to a wide ecosystem of scientific packages. Firedrake also follows the principle of graceful degradation, by providing a very simple lower-level API to escape the abstraction when non-standard calculations (i.e., unrelated to the finite-element formulation) are required. Likewise, Devito allows injecting arbitrary expressions into the finite-difference specification; this feature has been used in real-life cases, for example for interpolation in seismic imaging operators. On the other hand, a major difference is that Devito lacks a formal specification language such us UFL in FEniCS/Firedrake. This is partly because there is no systematic foundation underpinning FD, as opposed to FE which relies upon the theory of Hilbert spaces [160]. Yet another distinction is that, for performance reasons, Devito takes control of the time-stepping loop. Other examples of embedded DSLs are provided by the OpenFOAM project, with a language for FV [161], and by PyFR, which targets flux reconstruction methods [162].

4.2.2 High-level approaches to finite differences

Due to its simplicity, the FD method has been the subject of multiple research projects, chiefly targeting the design of effective software abstraction and/or the generation of high performance code [49, 46, 47, 48]. Devito distinguishes itself from previous work in a number of ways including: support for the principle of graceful degradation for when the DSL does not cover a feature required by an application; incorporation of a symbolic mathematics engine; using actual compiler technology rather than template-based code generation; adoption of a native *Python* interface that naturally allows composition into complex workflows such as optimisation and machine-learning frameworks.

At a lower level of abstraction there are a number of tools targeting "stencil" computation (FD codes belong to this class), whose major objective is the generation of efficient code. Some of them provide a DSL [159, 163, 164, 165], whereas others are compilers or user-driven code generation systems, often based upon a polyhedral model, such as [130, 166]. From the Devito standpoint, the aim is to harness these tools – for example by integrating them, to maximize performance portability. As a proof of concept, we shall discuss the integration of one such tool, namely YASK [159], with Devito.

4.2.3 Devito and seismic imaging

Devito is a general purpose system, not restricted to specific PDEs, so it can be used for any form of the wave equation. Thus, unlike software specialized in seismic exploration, like IWAVE [11] and Madagascar [12], it suffers neither from the restriction to a small set of wave equations and discretizations, nor from the lack of portability and composability typical of a pure C/Fortran environment.

4.2.4 Performance optimizations

The Devito compiler can introduce three types of performance optimizations: FLOPs reduction, data locality, and parallelism. Typical FLOPs reduction transformations are common sub-expressions elimination, factorization, and code motion. A thorough review is provided in [167]. To different extent, Devito applies all of these techniques (see Section 4.5.1). Particularly relevant for stencil computation is the search for redundancies across consecutive loop iterations [168, 169, 170]. This is at the core of the strategy described in Section 4.6, which essentially extends these ideas with optimizations for data locality. Typical loop transformations for parallelism and data locality [171] are also automatically introduced by the Devito compiler (e.g., loop blocking, vectorization); more details will be provided in Sections 4.5.2 and 4.5.3.

4.3 Specification of a finite-difference method with Devito

The Devito DSL allows concise expression of FD and general stencil operations using a mathematical notation. It uses *SymPy* [172] for the specification and manipulation of stencil expressions. In this section, we describe the use of Devito's DSL to build PDE solvers. Although the examples used here are for FD, the DSL can describe a large class of operations, such as convolutions or basic linear algebra operations (e.g., chained tensor multiplications).

4.3.1 Symbolic types

The key steps to implement a numerical kernel with Devito are shown in Figure 4.1. We describe this workflow, as well as fundamental features of the Devito API, using the acoustic wave equation, also known as d'Alembertian or Box operator. Its continuous form is given by:

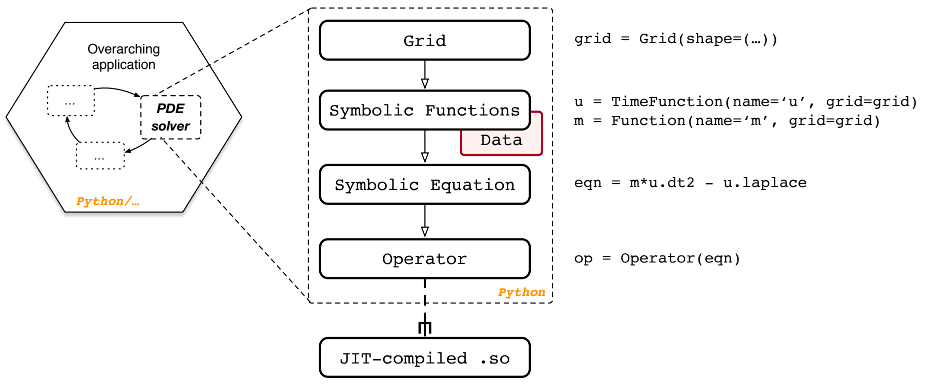


Figure 4.1: The typical usage of Devito within a larger application.

$$\begin{aligned} m(x,y,z)\frac{d^2u(x,y,z,t)}{dt^2} - \nabla^2 u(x,y,z,t) &= q_s, \\ u(x,y,z,0) &= 0, \\ \frac{du(x,y,z,t)}{dt}\Big|_{t=0} &= 0, \end{aligned} \tag{4.1}$$

where the variables of this expression are defined as follows:

- $m(x,y,z) = \frac{1}{c(x,y,z)^2}$, is the parametrization of the subsurface with $c(x,y,z)$ being the speed of sound as a function of the three space coordinates (x,y,z);
- $u(x,y,z,t)$, is the spatially varying acoustic wavefield, with the additional dimension of time t;
- q_s is the source term, which is a point source in this case.

The first step towards solving this equation is the definition of a discrete computational grid, on which the model parameters, wavefields and source are defined. The computational grid is defined as a `Grid(shape)` object, where `shape` is the number of grid points in each spatial dimension. Optional arguments for instantiating a `Grid` are `extent`, which defines the extent in physical units, and `origin`, the origin of the coordinate system, with respect to which all other coordinates are defined.

The next step is the symbolic definition of the squared slowness, wavefield and source. For this, we introduce some fundamental types.

- `Function` represents a discrete spatially varying function, such as the velocity. A `Function` is instantiated for a defined `name` and a given `Grid`.
- `TimeFunction` represents a discrete function that is both spatially varying and time dependent, such as wavefields. Again, a `TimeFunction` object is defined on an existing `Grid` and is identified by its `name`.
- `SparseFunction` and `SparseTimeFunction` represent sparse functions, that is functions that are only defined over a subset of the grid, such as a seismic point source. The corresponding object is defined on a `Grid`, identified by a `name`, and also requires the `coordinates` defining the location of the sparse points.

Apart from the grid information, these objects carry their respective FD discretization information in space and time. They also have a `data` field that contains values of the respective function at the defined grid points. By default, `data` is initialized with zeros and therefore automatically satisfies the initial conditions from equation 4.1. The initialization of the fields to solve the wave equation over a one-dimensional grid is displayed in Listing 1.

Listing 1 Setup `Functions` to express and solve the acoustic wave equation.

```
>>> from devito import Grid, TimeFunction, Function, SparseTimeFunction
>>> g = Grid(shape=(nx,), origin=(ox,), extent=(sx,))
>>> u = TimeFunction(name="u", grid=g, space_order=2, time_order=2)  # Wavefield
>>> m = Function(name="m",  grid=g)  # Physical parameter
>>> q = SparseTimeFunction(name="q", grid=g, coordinates=coordinates)  # Source
```

4.3.2 Discretization

With symbolic objects that represent the discrete velocity model, wavefields and source function, we can now define the full discretized wave equation. As mentioned earlier, one of the main features of Devito is the possibility to formulate stencil computations as concise mathematical expressions. To do so, we provide shortcuts to classic FD stencils, as well as the functions to define arbitrary stencils. The shortcuts are accessed as object properties and are supported by `TimeFunction` and `Function` objects. For example, we can take spatial and temporal derivatives of the wavefield `u` via the shorthand expressions `u.dx` and `u.dt` (Listing 2).

Listing 2 Example of spatial and temporal FD stencil creation.

```
>>> u.dx
-u(t, x - h_x)/(2*h_x) + u(t, x + h_x)/(2*h_x)
>>> u.dt
-u(t - dt, x)/(2*dt) + u(t + dt, x)/(2*dt)
>>> u.dt2
-2*u(t, x)/dt**2 + u(t - dt, x)/dt**2 + u(t + dt, x)/dt**2
```

Furthermore, Devito provides shortcuts for common differential operations such as the Laplacian via `u.laplace`. The full discrete wave equation can then be implemented in a single line of *Python* (Listing 3).

Listing 3 Expressing the wave equation.

```
>>> wave_equation = m * u.dt2 - u.laplace
>>> wave_equation
(-2*u(t, x)/dt**2 + u(t - dt, x)/dt**2 + u(t + dt, x)/dt**2)*m(x) + 2*u(t, x)/h_x**2 - u(t, x -
    h_x)/h_x**2 - u(t, x + h_x)/h_x**2
```

To solve the time-dependent wave equation with an explicit time-stepping scheme, the symbolic expression representing our PDE has to be rearranged such that it yields an update rule for the wavefield u at the next time step: $u(t + dt) = f(u(t), u(t - dt)))$. Devito allows to rearrange the PDE expression automatically using the `solve` function, as shown in Listing 4.

Listing 4 Time-stepping scheme for the acoustic wave equation. `region=INTERIOR` ensures that the Dirichlet boundary conditions at the edges of the Grid are satisfied.

```
>>> from devito import Eq, INTERIOR, solve
>>> stencil = Eq(u.forward, solve(wave_equation, u.forward), region=INTERIOR)
>>> stencil
Eq(u(t + dt, x), -2*dt**2*u(t, x)/(h_x**2*m(x)) + dt**2*u(t, x - h_x)/(h_x**2*m(x)) + dt**2*u(t,
     x + h_x)/(h_x**2*m(x)) + 2*u(t, x) - u(t - dt, x))
```

Note that the stencil expression in Listing 4 does not yet contain the point source q. This could be included as a regular Function which has zeros all over the grid except for a few points; this, however, would obviously be wasteful. Instead, SparseFunctions allow to perform operations, such as injecting a source or sampling the wavefield, at a subset of grid points determined by coordinates. In the case in which coordinates do not coincide with grid points, bilinear (for 2D) or trilinear (for 3D) interpolation are employed. To inject a point source into the stencil expression, we use the inject function of the SparseTimeFunction object that represents our seismic source (Listing 5).[1]

Listing 5 Expressing the injection of a source into a field.

```
>>> injection = q.inject(field=u.forward, expr=dt**2 * q / m)
>>> injection
[Eq(u[t + 1, INT(floor((-o_x + q_coords[p_q, 0])/h_x))], dt**2*(1 - FLOAT(-h_x*INT(floor((-o_x +
      q_coords[p_q, 0])/h_x)) - o_x + q_coords[p_q, 0])/h_x)*q[time, p_q]/m[INT(floor((-o_x +
     q_coords[p_q, 0])/h_x))] + u[t + 1, INT(floor((-o_x + q_coords[p_q, 0])/h_x))]),
 Eq(u[t + 1, INT(floor((-o_x + q_coords[p_q, 0])/h_x)) + 1], dt**2*FLOAT(-h_x*INT(floor((-o_x +
      q_coords[p_q, 0])/h_x)) - o_x + q_coords[p_q, 0])*q[time, p_q]/(h_x*m[INT(floor((-o_x +
      q_coords[p_q, 0])/h_x)) + 1]) + u[t + 1, INT(floor((-o_x + q_coords[p_q, 0])/h_x)) + 1])]
```

The inject function takes the field being updated as an input argument (in this case u.forward), while expr=dt**2 * q / m is the expression being injected. The result of the inject function is a list of symbolic expressions, similar to the stencil expression we defined earlier. As we shall see, these expressions are eventually joined together and used to create an Operator object – the solver of our PDE.

4.3.3 Boundary conditions

Simple boundary conditions (BCs), such as Dirichlet BCs, can be imposed on individual equations through special keywords (see Listing 4). For more exotic schemes, instead, the BCs need to be explicitly written (e.g., Higdon BCs [173]), just like any of the symbolic expressions defined in the Listings above. For reasons of space, this aspect is not elaborated further; the interested reader may refer to [174].

[1]More complicated interpolation schemes can be defined by precomputing the grid points corresponding to each sparse point, and their respective coefficients. The result can then be used to create a Precomputed-SparseFunction, which behaves like a SparseFunction at the symbolic level.

4.3.4 Control flow

By default, the extent of a `TimeFunction` in the time dimension is limited by its time order. Hence, the shape of u in Listing 1 is $(time\_order + 1, nx) = (3, nx)$. The iterative method will then access u via modulo iteration, that is $u[t\%3, ...]$. In many scenarios, however, the entire time history, or at least periodic time slices, should be saved (e.g., for inversion algorithms). Listing 6 expands our running example with an equation that saves the content of u every 4 iterations, up to a maximum of $save = 100$ time slices.

Listing 6 Implementation of time sub-sampling.

```
>>> from devito import ConditionalDimension
>>> ts = ConditionalDimension('ts', parent=g.time_dim, factor=4)
>>> us = TimeFunction(name='us', grid=g, save=100, time_dim=ts)
>>> save = Eq(us, u)
```

In general, all equations that access `Functions` (or `TimeFunctions`) employing one or more `ConditionalDimensions` will be conditionally executed. The condition may be a number indicating how many iterations should pass between two executions of the same equation, or even an arbitrarily complex expression.

4.3.5 Domain, halo, and padding regions

A `Function` internally distinguishes between three regions of points.

Domain Represents the *computational domain* of the `Function` and is inferred from the input `Grid`. This includes any elements added to the *physical domain* purely for computational purposes, e.g. absorbing boundary layers.

Halo The grid points surrounding the domain region, i.e. "ghost" points that are accessed by the stencil when iterating in proximity of the domain boundary.

Padding The grid points surrounding the halo region, which are allocated for performance optimizations, such as data alignment. Normally this region should be of no interest to a user of Devito, except for precise measurement of memory allocated for each `Function`.

4.4 The Devito compiler

In Devito, an `Operator` carries out three fundamental tasks: generation of low-level code, JIT compilation, and execution. The `Operator` input consists of one or more symbolic equations. In the generated code, these equations are scheduled within loop nests of suitable depth and extent. The `Operator` also accepts substitution rules (to replace symbols with constant values) and optimization levels for the Devito Symbolic Engine (DSE) and the Devito Loop Engine (DLE). By default, all DSE and DLE optimizations that are known to unconditionally improve performance are automatically applied. The same `Operator` may be reused with different input data; JIT-compilation occurs only once, triggered by the first execution. Overall, this lowering process – from high-level equations to dynamically compiled and executable code – consists of multiple compiler passes, summarized in Figure 4.2 and discussed in the following sections (a minimal background in data dependence analysis is recommended; the unfamiliar reader may refer to a classic textbook such as [175]).

4.4.1 Equations lowering

In this pass, three main tasks are carried out: *indexification*, *substitution*, and *domain-alignment*.

- As explained in Section 4.3, the input equations typically involve ore or more indexed `Functions`. The *indexification* consists of converting such objects into actual arrays. An array always keeps a reference to its originating `Function`. For instance, all accesses to u such as $u[t, x+1]$ and $u[t+1, x-2]$ would store a pointer to the same, user-defined `Function` $u(t, x)$. This metadata is exploited throughout the various compilation passes.

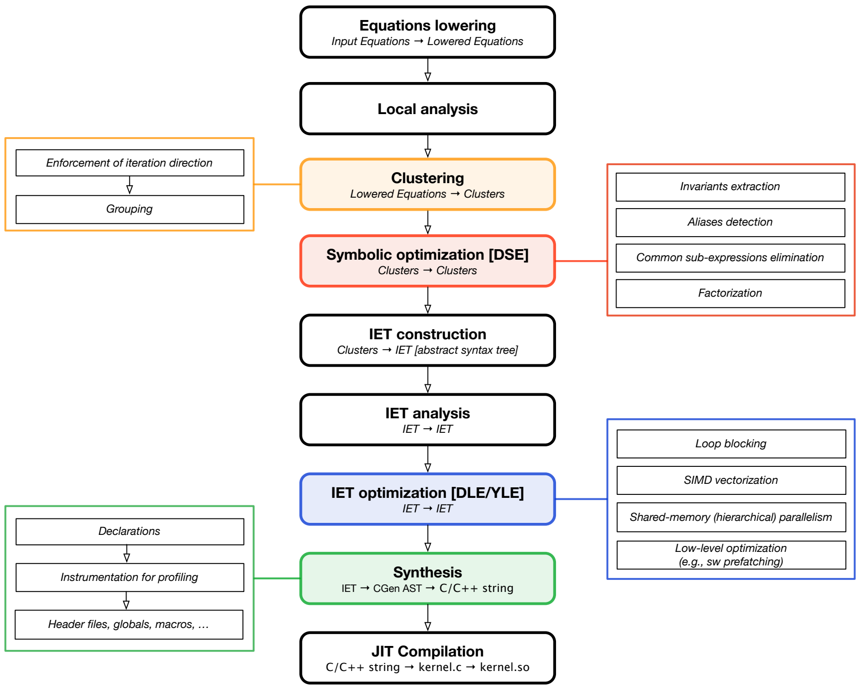

Equations lowering
Input Equations → Lowered Equations
Local analysis
Clustering
Lowered Equations → Clusters
Enforcement of iteration direction
Grouping
Symbolic optimization [DSE]
Clusters → Clusters
Invariants extraction
Aliases detection
Common sub-expressions elimination
Factorization
IET construction
Clusters → IET [abstract syntax tree]
IET analysis
IET → IET
IET optimization [DLE/YLE]
IET → IET
Loop blocking
SIMD vectorization
Shared-memory (hierarchical) parallelism
Low-level optimization (e.g., sw prefatching)
Synthesis
IET → CGen AST → C/C++ string
Declarations
Instrumentation for profiling
Header files, globals, macros, ...
JIT Compilation
C/C++ string → kernel.c → kernel.so

Figure 4.2: Compiler passes to lower symbolic equations into shared objects through an `Operator`.

- During *substitution*, the user-provided substitution rules are applied. These may be given for any literal appearing in the input equations, such as the grid spacing symbols. Applying a substitution rule increases the chances of constant folding, but it makes the `Operator` less generic. The values of symbols for which no substitution rule is available are provided at execution time.

- The *domain-alignment* step shifts the array accesses deriving from `Functions` having non-empty halo and padding regions. Thus, the array accesses become logically aligned to the equation's natural domain. For instance, given the usual `Function` $u(t, x)$ having two points on each side of the x halo region, the array accesses $u[t, x]$ and $u[t, x + 2]$ are transformed, respectively, into $u[t, x + 2]$ and $u[t, x + 4]$. When $x = 0$, therefore, the values $u[t, 2]$ and $u[t, 4]$ are fetched, representing the first and third points in the computational domain.

4.4.2 Local analysis

The lowered equations are analyzed to collect information relevant for the `Operator` construction and execution. In this pass, an equation is inspected "in isolation", ignoring its relationship with the rest of the input. The following metadata are retrieved and/or computed:

- input and output `Functions`;
- `Dimensions`, which are topologically ordered based on how they appear in the various array index functions; and
- two notable `Spaces`: the iteration space, `ISpace`, and the data space, `DSpace`.

A `Space` is a collection of points given by the product of n compact intervals on $\mathbb{Z}$. With the notation $d[o_m, o_M]$ we indicate the compact interval $[d_m + o_m, d_M + o_M]$ over the `Dimension` d, in which d_m and d_M are parameters (specialized only at runtime), while o_m and o_M are known integers. For instance, $[x[0, 0], y[-1, 1]]$ describes a rectangular two-dimensional space over x and y, whose points are given by the Cartesian product $[x_m, x_M] \times [y_m - 1, y_M + 1]$. The `ISpace` and `DSpace` are two special types of `Space`. They usually span different sets of `Dimensions`. A `DSpace` may have `Dimensions` that do not appear in an `ISpace`, in particular those that are accessed only via integer indices. Likewise, an `ISpace` may have `Dimensions` that are not part of the `DSpace`, such as a reduction axis. Further, an `ISpace` also carries, for each `Dimension`, its iteration direction.

As an example, consider the equation *stencil* in Listing 4. Immediately we see that input $= [u, m]$, output $= [u]$, $\texttt{Dimension}s = [t, x]$. The compiler constructs the `ISpace` $[t[0, 0]^+, x[0, 0]^*]$. The first entry $t[0, 0]^+$ indicates that, along t, the equation should run between $t_m + 0$ and $t_M + 0$ (extremes included) in the *forward* direction, as indicated by the symbol $+$. This is due to the fact that there is a flow dependence in t, so only a unitary positive stepping increment (i.e., $t = t + 1$) allows a correct propagation of information across consecutive iterations. The only difference along x is that the iteration direction is now arbitrary, as indicated by $*$. The `DSpace` is $[t[0, 1], x[0, 0]]$; intuitively, the entry $t[0, 1]$ is used right before running an `Operator` to provide a default value for t_M – in particular, t_M will be set to the largest possible value that does not cause out-of-domain accesses (i.e., out-of-bounds array accesses).

4.4.3 Clustering

A `Cluster` is a sequence of equations having (i) same `ISpace`, (ii) same control flow (i.e., same `ConditionalDimensions`), and (iii) no dimension-carried "true" anti-dependences among them.

As an example, consider again the setup in Section 4.3. The equation *stencil* cannot be "clusterized" with the equations in the *injection* list as their `ISpaces` are different. On the other hand, the equations in *injection* can be grouped together in the same `Cluster` as (i) they have same `ISpace` $[t[0, 0]^*, p_q[0, 0]^*]$, (ii) same control flow, and (iii) there are no true anti-dependences among them (note that the second equation in *injection* does write to $u[t + 1, ...]$, but as explained later this is in fact a reduction, that is a "false" anti-dependence).

Iteration direction

First, each equation is assigned a new `ISpace`, based upon a *global* analysis. Any of the iteration directions that had been marked as "arbitrary" ($*$) during local analysis may now be enforced to *forward* ($+$) or *backward* ($-$). This process exploits data dependence analysis.

For instance, consider the flow dependence between *stencil* and the *injection* equations. If we want u to be up-to-date when evaluating *injection*, then we eventually need all equations to be scheduled sequentially within the t loop. For this, the `ISpaces` of the *injection* equations are specialized by enforcing the direction *forward* along the `Dimension` t. The new `ISpace` is $[t[0,0]^+, p_q[0,0]^*]$.

Algorithm 1 illustrates how the enforcement of iteration directions is achieved in general. Whenever a clash is detected (i.e., two equations with `ISpace` $[d[0,0]^+, ...]$ and $[d[0,0]^-, ...]$), the original direction determined by the local analysis pass is kept (lines 11 and 13), which will eventually lead to generating different loops.

Algorithm 1: Clustering: enforcement of iteration directions (pseudocode).

```
   Input: A sequence of equations $\mathcal{E}$.
   Output: A sequence of equations $\mathcal{E}'$ with altered ISpace.
   // Map each dimension to a set of expected iteration directions
1  mapper ← DETECT_FLOW_DIRECTIONS($\mathcal{E}$);
2  for e in $\mathcal{E}$ do
3      for dim, directions in mapper do
4          if len(directions) == 1 then
               // No ambiguity
5              forced[dim] ← directions.pop();
6          else if len(directions) == 2 then
               // No ambiguity as long as one of the two items is /Any/
7              try
8                  directions.remove(Any);
9                  forced[dim] ← directions.pop();
10             except
11                 forced[dim] ← e.directions[dim];
12         else
13             forced[dim] ← e.directions[dim];
14         end if
15     end for
16     $\mathcal{E}'$.append(e._rebuild(directions=forced))
17 end for
18 return $\mathcal{E}'$
```

Grouping

This step performs the actual clustering, checking `ISpaces` and anti-dependences, as well as handling control flow. The procedure is shown in Algorithm 2; some explanations follow.

- Robust data dependence analysis, capable of tracking flow-, anti-, and output-dependencies at the level of array accesses, is necessary. In particular, it must be able to tell

Algorithm 2: Clustering: grouping expressions into `Clusters` (pseudocode)

```
Input: A sequence of equations $\mathcal{E}$.
Output: A sequence of clusters $\mathcal{C}$.
$\mathcal{C} \leftarrow$ ClusterGroup();
for e in $\mathcal{E}$ do
    grouped $\leftarrow$ false;
    for c in reversed($\mathcal{C}$) do
        anti, flow $\leftarrow$ GET_DEPENDENCES(c, e);
        if e.ispace == c.ispace and anti.carried is empty then
            c.add(e);
            grouped $\leftarrow$ true;
            break;
        else if anti.carried is not empty then
            c.atomics.update(anti.carried.cause);
            break;
        else if flow.cause.intersection(c.atomics) then
            // cannot search across earlier clusters
            break;
    end for
    if not grouped then
        $\mathcal{C}$.append(Cluster(e));
    end if
end for
$\mathcal{C} \leftarrow$ CONTROL_FLOW($\mathcal{C}$);
return $\mathcal{C}$
```

whether two generic *array accesses* induce a dependence or not. The data dependence analysis performed is conservative; that is, a dependence is always assumed when a test is inconclusive. Dependence testing is based on the standard Lamport test [175]. In Algorithm 2, data dependence analysis is carried out by the function GET_DEPENDENCES.

- If an anti-dependence is detected along a `Dimension` i, then i is marked as *atomic* – meaning that no further clustering can occur along i. This information is also exploited by later `Operator` passes (see Section 4.4.5).
- Reductions, and in particular increments, are treated specially. They represent a special form of anti-dependence, as they do not break clustering. GET_DEPENDENCES detects reductions and removes them from the set of anti-dependencies.
- Given the sequence of equations $[E_1, E_2, E_3]$, it is possible that E_3 can be grouped

with E_1, but not with its immediate predecessor E_2 (e.g., due to a different `ISpace`). However, this can only happen when there are no flow or anti-dependences between E_2 and E_3; i.e. when the `if` commands at lines 10 and 13 are not entered, thus allowing the search to proceed with the next equation. This optimization was originally motivated by gradient operators in seismic imaging kernels.

- The routine CONTROL_FLOW, omitted for brevity, creates additional `Clusters` if one or more `ConditionalDimensions` are encountered. These are tracked in a special `Cluster` field, *guards*, as also required by later passes (see Section 4.4.5).

4.4.4 Symbolic optimization

The DSE – Devito Symbolic Engine – is a macro-pass reducing the *arithmetic strength* of `Clusters` (e.g., their operation count). It consists of a series of passes, ranging from standard common sub-expression elimination (CSE) to more advanced rewrite procedures, applied individually to each `Cluster`. The DSE output is a new ordered sequence of `Clusters`: there may be more or fewer `Clusters` than in the input, and both the overall number of equations as well as the sequence of arithmetic operations might differ. The DSE passes are discussed in Section 4.5.1. We remark that the DSE only operates on `Clusters` (i.e., on collections of equations); there is no concept of "loop" at this stage yet. However, by altering `Clusters`, the DSE has an indirect impact on the final loop-nest structure.

4.4.5 IET construction

In this pass, the intermediate representation is lowered to an Iteration/Expression Tree (IET). An IET is an abstract syntax tree in which `Iterations` and `Expressions` – two special node types – are the main actors. Equations are wrapped within `Expressions`, while `Iterations` represent loops. Loop nests embedding such `Expressions` are constructed by suitably nesting `Iterations`. Each `Cluster` is eventually placed in its own loop (`Iteration`) nest, although some (outer) loops may be shared by multiple

Clusters.

Algorithm 3: An excerpt of the cluster scheduling algorithm, turning a list (of Clusters) into a tree (IET). Here, the fact that different Clusters may eventually share some outer Iterations is highlighted.

```
Input: A sequence of Clusters $\mathcal{C}$.
Output: An Iteration/Expression Tree.
1  schedule ← list();
2  for $c$ in $\mathcal{C}$ do
3      root ← None;
4      index ← 0;
5      for $i_0$, $i_1$ in zip(c.ispace, schedule) do
6          if $i_0$ != $i_1$ or $i_0$.dimension in c.atomics then
7              break;
8          end if
9          root ← schedule[i1];
10         index ← index + 1;
11         if $i_0$.dim in c.guards then
12             break;
13         end if
14     end for
15     ⟨build as many Iterations as Dimensions in c.ispace[index:] and nest them
         inside root⟩;
16     ⟨update schedule⟩;
17     ⟨...⟩
18 end for
```

Consider again our running acoustic wave equation example. There are three Clusters in total: C_1 for *stencil*, C_2 for *save*, and C_3 for the equations in *injection*. We use Algorithm 3 – an excerpt of the actual cluster scheduling algorithm – to explain how this sequence of Clusters is turned into an IET. Initially, the *schedule* list is empty, so when C_1 is handled two nested Iterations are created (line 15), respectively for the Dimensions t and x. Subsequently, C_2's ISpace and the current *schedule* are compared (line 5). It turns out that t appears among C_2's guards, hence the for loop is exited at line 12 without inspecting the second and last iteration. Thus, $index = 1$, and the previously built Iteration over t is reused. Finally, when processing C_3, the for loop is exited at the second iteration due to line 6, since $p_q! = x$. Again, the t Iteration is reused, while a new Iteration is constructed for the Dimension p_q. Eventually, the constructed IET is as in Listing 7.

Listing 7 Graphical representation of the IET produced by the cluster scheduling algorithm for the running example.

```
for t = t_m to t_M:
 |-- for x = x_m to x_M:
 |     |-- <Eq(u[t+1,x], ...)>
 |
 |-- if t % 4 == 0
 |     |-- for x = x_m to x_M:
 |           |-- <Eq(us[t/4, x], ...)>
 |
 |-- for p_q = p_q_m to p_q_M:
       |-- <Eq(u[t+1,f(p_q)], ...)>
       |-- <Eq(u[t+1,g(p_q)], ...)>
```

4.4.6 IET analysis

The newly constructed IET is analyzed to determine `Iteration` properties such as `sequential`, `parallel`, and `vectorizable`, which are then attached to the relevant nodes in the IET. These properties are used for loop optimization, but only by a later pass (see Section 4.4.7). To determine whether an `Iteration` is `parallel` or `sequential`, a fundamental result from compiler theory is used – the i-th `Iteration` in a nest comprising n `Iterations` is parallel if for all dependences D, expressed as distance vectors $D = (d_0, ..., d_{n-1})$, either $(d_1, ..., d_{i-1}) > 0$ or $(d_1, ..., d_i) = 0$ [175].

4.4.7 IET optimization

This macro-pass transforms the IET for performance optimization. Apart from runtime performance, this pass also optimizes for rapid JIT compilation with the underlying C compiler. A number of loop optimizations are introduced, including loop blocking, minimization of remainder loops, SIMD vectorization, shared-memory (hierarchical) parallelism via OpenMP, software prefetching. These will be detailed in Section 4.5. A *backend* (see Section 4.4.9) might provide its own loop optimization engine.

4.4.8 Synthesis, dynamic compilation, and execution

Finally, the IET adds variable declarations and header files, as well as instrumentation for performance profiling, in particular, to collect execution times of specific code regions.

Declarations are injected into the IET, ensuring they appear as close as possible to the scope in which the relative variables are used, while honoring the OpenMP semantics of private and shared variables. To generate C code, a suitable tree visitor inspects the IET and incrementally builds a *CGen* tree [176], which is ultimately translated into a string and written to a file. Such files are stored in a software cache of Devito-generated `Operator`s, JIT-compiled into a shared object, and eventually loaded into the *Python* environment. The compiled code has a default entry point (a special function), which is called directly from *Python* at `Operator` application time.

4.4.9 Operator specialization through backends

In Devito, a *backend* is a mechanism to specialize data types as well as `Operator` passes, while preserving software modularity (inspired by [177]).

One of the main objectives of the backend infrastructure is promoting software composability. As explained in Section 4.2, there exist a significant number of interesting tools for stencil optimization, which we may want to integrate with Devito. For example, one of the future goals is to support GPUs, and this might be achieved by writing a new backend implementing the interface between Devito and third-party software specialized for this particular architecture.

Currently, two backends exist:

`core` the default backend, which relies on the DLE for loop optimization.

`yask` an alternative backend using the YASK stencil compiler to generate optimized C++ code for Intel® Intel®Xeon™and Intel®Xeon Phi™architectures [159]. Devito transforms the IET into a format suitable for YASK, and uses its API for data management, JIT-compilation, and execution. Loop optimization is performed by YASK through the YASK Loop Engine (YLE).

The *core* and *yask* backends share the compilation pipeline in Figure 4.2 until the loop optimization stage.

4.5 Automated performance optimizations

As discussed in Section 4.4, Devito performs symbolic optimizations to reduce the arithmetic strength of the expressions, as well as loop transformations for data locality and parallelism. The former are implemented as a series of compiler passes in the DSE, while for the latter there currently are two alternatives, namely the DLE and the YLE (depending on the chosen execution backend).

Devito abstracts away the single optimizations passes by providing users with a certain number of optimization levels, called "modes", which trigger pre-established sequences of optimizations – analogous to what general-purpose compilers do with, for example, `-O2` and `-O3`. In Sections 4.5.1, 4.5.2, and 4.5.3 we describe the individual passes provided by the DSE, DLE, and YLE respectively, while in Section 4.7.1 we explain how these are composed into modes.

4.5.1 DSE - Devito Symbolic Engine

The DSE passes attempt to reduce the arithmetic strength of the expressions through FLOP-reducing transformations [167]. They are illustrated in Listings 8-11, which derive from the running example used throughout the article. A detailed description follows.

- **Common sub-expression elimination (CSE).** Two implementations are available: one based upon *SymPy* 's `cse` routine and one built on top of more basic *SymPy* routines, such as `xreplace`. The former is more powerful, being aware of key arithmetic properties such as associativity; hence it can discover more redundancies. The latter is simpler, but avoids a few critical issues: (i) it has a much quicker turnaround time; (ii) it does not capture integer index expressions (for increased quality of the generated code); and (iii) it tries not to break factorization opportunities. A generalized common sub-expressions elimination routine retaining the features and avoiding the drawbacks of both implementations is still under development. By default, the latter implementation is used when the CSE pass is selected.

Listing 8 An example of common sub-expressions elimination.

```
>>> 9.0*dt*dt*u[t, x + 1] - 18.0*dt*dt*u[t][x + 2] + 9.0*dt*dt*u[t, x + 3]
temp0 = dt*dt
9.0*temp0*u[t, x + 1] - 18.0*temp0*u[t][x + 2] + 9.0*temp0*u[t, x + 3]
```

- **Factorization.** This pass visits each expression tree and tries to factorize FD weights. Factorization is applied without altering the expression structure (e.g., without expanding products) and without performing any heuristic search across groups of expressions. This choice is based on the observation that a more aggressive approach is only rarely helpful (never in the test cases in Section 4.7), while the increase in symbolic processing time could otherwise be significant. The implementation exploits the *SymPy* `collect` routine. However, while `collect` only searches for common factors across the immediate children of a single node, the DSE implementation recursively applies `collect` to each `Add` node (i.e., an addition) in the expression tree, until the leaves are reached.

Listing 9 An example of FD weights factorization.

```
>>> 9.0*temp0*u[t, x + 1] - 18.0*temp0*u[t][x + 2] + 9.0*temp0*u[t, x + 3]
9.0*temp0*(u[t, x + 1] + u[t, x + 3]) - 18.0*temp0*u[t][x + 2]
```

- **Extraction.** The name stems from the fact that sub-expressions matching a certain condition are pulled out of a larger expression, and their values are stored into suitable scalar or tensor temporaries. For example, a condition could be "*extract all time-varying sub-expressions whose operation count is larger than a given threshold*". A tensor temporary may be preferred over a scalar temporary if the intention is to let the *IET construction* pass (see Section 4.4.5) place the pulled sub-expressions within an outer loop nest. Obviously, this comes at the price of additional storage. This peculiar effect – trading operations for memory – will be thoroughly analyzed in Sections 4.6 and 4.7.

Listing 10 An example of time-varying sub-expressions extraction. Only sub-expressions performing at least one floating-point operation are extracted.

```
>>> 9.0*temp0*(u[t, x + 1] + u[t, x + 3]) - 18.0*temp0*u[t][x + 2]
temp1[x] = u[t, x + 1] + u[t, x + 3]
9.0*temp0*temp1[x] - 18.0*temp0*u[t][x + 2]
```

- **Detection of aliases.** The Alias-Detection Algorithm implements the most advanced DSE pass. In essence, an alias is a sub-expression that is redundantly computed at multiple iteration points. Because of its key role in the Cross-Iteration Redundancy-Elimination algorithm, the formalization of the Alias-Detection Algorithm is postponed until Section 4.6.

Listing 11 An example of alias detection.

```
>>> 9.0*temp0*u[t, x + 1] - 18.0*temp0*u[t][x + 2] + 9.0*temp0*u[t, x + 3]
temp[x] = 9.0*temp0*u[t, x]
temp[x + 1] - 18.0*temp0*u[t][x + 2] + temp[x + 3]
```

4.5.2 DLE - Devito Loop Engine

The DLE transforms the IET via classic loop optimizations for parallelism and data locality [171]. These are summarized below.

- **SIMD Vectorization.** Implemented by enforcing compiler auto-vectorization via special `pragmas` from the OpenMP 4.0 language. With this approach, the DLE aims to be performance-portable across different architectures. However, this strategy causes a significant fraction of vector loads/stores to be unaligned to cache boundaries, due to the stencil offsets. As we shall see, this is a primary cause of performance loss.

- **Loop Blocking.** Also known as “tiling”, this technique implemented by replacing `Iteration` trees in the IET. The current implementation only supports blocking over fully-parallel `Iterations`. Blocking over dimensions characterized by flow- or anti-dependences, such as the time dimension in typical explicit finite difference schemes, is instead work in progress (this would require a preliminary pass known as loop skewing; see Section 4.8 for more details). On the other hand, a feature of the present implementation is the capability of blocking across particular *sequences* of loop nests. This is exploited by the Cross-Iteration Redundancy-Elimination algorithm, as shown in Section 4.6.3. To determine an optimal block shape, an `Operator` resorts to empirical auto-tuning.

- **Parallelism.** Shared-memory parallelism is introduced by decorating `Iterations` with suitable OpenMP `pragmas`. The OpenMP `static` scheduling is used. Normally, only the outermost fully-parallel `Iteration` is annotated with the parallel `pragma`. However, heuristically nested fully-parallel `Iterations` are `collapsed` if the core count is greater than a certain threshold. This pass also ensures that all array temporaries allocated in the scope of the parallel `Iteration` are declared as `private` and that storage is allocated where appropriate (stack, heap).

Summarizing, the DLE applies a sequence of typical stencil optimizations, aiming to reach a minimum level of performance across different architectures. As we shall see, the effectiveness of this approach, based on simple transformations, deteriorates on architectures strongly conceived for hierarchical parallelism. This is one of the main reasons behind the development of the `yask` backend (see Section 4.4.9), described in the following section.

4.5.3 YLE - YASK Loop Engine

"YASK" (Yet Another Stencil Kernel) is an open-source C++ software framework for generating high-performance implementations of stencil codes for Intel® Intel®Xeon™and Intel®Xeon Phi™processors. Previous publications on YASK have discussed its overall structure [159] and its application to the Intel®Xeon Phi™x100 family (code-named Knights Corner) [178] and Intel®Xeon Phi™x200 family (code-named Knights Landing) [179, 180] many-core CPUs. Unlike Devito, it does not expose a symbolic language to the programmer or create stencils from finite-difference approximations of differential equations. Rather, the programmer provides simple declarative descriptions of the stencil equations using a C++ or Python API. Thus, Devito operates at a level of abstraction higher than that of YASK, while YASK provides performance portability across Intel architectures and is more focused on low-level optimizations. Following is a sample of some of the optimizations provided by YASK:[2]

- **Vector-folding.** In traditional SIMD vectorization, such as that provided by a vectorizing compiler, the vector elements are arranged sequentially along the unit-stride dimension of the grid, which must also be the dimension iterated over in the inner-most loop of the stencil application. Vector-folding is an alternative data-layout method whereby neighboring elements are arranged in small *multi-dimensional* tiles. Figure 4.3 illustrates three ways to pack eight double-precision floating-point values into a 512-bit SIMD register. Figure 4.3a shows a traditional 1D "in-line" layout, and 4.3b and 4.3c show alternative 2D and 3D "folded" layouts. Furthermore, these tiles may be ordered in memory in a dimension independent of the dimensions used in vectorization [178]. The combination of these two techniques can significantly increase overlap and reuse between successive stencil-application iterations, reducing the memory-bandwidth demand. For stencils that are bandwidth-bound, this can provide significant performance gains [178, 180].

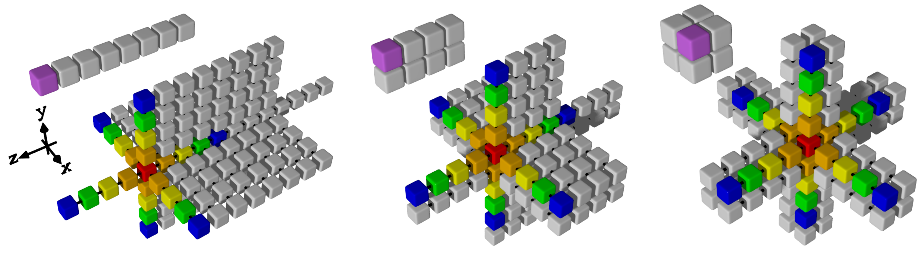
- **Software prefetching.** Many high-order or staggered-grid stencils require many streams of data to be read from memory, which can overwhelm the hardware prefetchers. YASK can be directed to automatically generate software prefetch instructions to improve the cache hit rates, especially on Xeon Phi CPUs.

- **Hierarchical parallelism.** Dividing the spatial domain into tiles to increase temporal cache locality is a common stencil optimization as discussed earlier. When

[2]Not all YASK features are currently used by Devito.

implementing this technique, sometimes called “cache-blocking”, it is typical to assign each thread to one or more small rectilinear subsets of the domain in which to apply the stencil(s). However, if these threads share caches, one thread’s data will often evict data needed later by another thread, reducing the effective capacity of the cache. YASK addresses this by employing two levels of OpenMP parallelization: the outer level of parallel loops are applied across the cache-blocks, and an inner level is applied across sub-blocks within those tiles. In the case of the Xeon Phi, the eight hyper-threads that share each L2 cache can now cooperate on filling and reusing the data in the cache, rather than evicting each other’s data.

YASK also provides other optimizations, such as temporal wave-fron tiling, as well as MPI support. These features, however, are not exploited by Devito yet. The interested reader may refer to [179, 181].



a. $1 \times 1 \times 8$ 1D fold   b. $1 \times 2 \times 4$ 2D fold   c. $2 \times 2 \times 2$ 3D fold

Figure 4.3: Various folds of 8 elements [178]. The smaller diagram in the upper-left of each sub-figure illustrates a single SIMD layout, and the larger diagram shows the input values needed for a typical 25-point stencil, as from an 8th-order finite-difference approximation of an isotropic acoustic wave. Note that the $1 \times 1 \times 8$ 1D fold corresponds to the traditional in-line vectorization.

To obtain the best of both tools, we have integrated the YASK framework into the Devito package. In essence, the Devito `yask` backend exploits the intermediate representation of an `Operator` to generate YASK kernels. This process is based upon sophisticated compiler technology. In *Devito v3.1*, roughly 70% of the Devito API is supported by the `yask` backend[3].

[3]At the time of writing, reaching feature-completeness is one the major on-going development efforts

4.6 The Cross-Iteration Redundancy-Elimination Algorithm

Aliases, or "cross-iteration redundancies" (informally introduced in Section 4.5.1), in FD operators depend on the differential operators used in the PDE(s) and the chosen discretization scheme. From a performance viewpoint, the presence of aliases is a non-issue as long as the operator is memory-bound, while it becomes relevant in kernels with a high arithmetic intensity. In Devito, the Cross-Iteration Redundancy-Elimination (CIRE) algorithm attempts to remove aliases with the goal of reducing the operation count. As shown in Section 4.7, the CIRE algorithm has considerable impact in seismic imaging kernels. The algorithm is implemented through the orchestration of multiple DSE and DLE/YLE passes, namely *extraction of candidate expressions (DSE)*, *detection of aliases (DSE)*, *loop blocking (DLE/YLE)*.

4.6.1 Extraction of candidate expressions

The criteria for extraction of candidate sub-expressions are:

- Any *maximal time-invariant* whose operation count is greater than $Thr_0 = 10$ (floating point arithmetic only). The term "maximal" means that the expression is not embedded within a larger time-invariant. The default value $Thr_0 = 10$, determined empirically, provides systematic performance improvements in a series of seismic imaging kernels. Transcendental functions are given a weight in the order of tens of operations, again determined empirically.

- Any *maximal time-varying* whose operation count is greater than $Thr_1 = 10$. Such expressions often lead to aliases, since they typically result from taking spatial and time derivatives on `TimeFunctions`. In particular, cross-derivatives are a major cause of aliases.

This pass leverages the *extraction* routine described in Section 4.5.1.

4.6.2 Detection of aliases

To define the concept of aliasing expressions, we first need to formalize the notion of *translated operands*. Here, an operand is regarded as the arithmetic product of a scalar value (or "coefficient") and one or more indexed objects. An indexed object is characterized by a label (i.e., its name), a vector of n dimensions, and a vector of n displacements (one for each dimension). We say that an operand o_1 is translated with respect to an operand o_0 if o_0 and o_1 have same coefficient, label, and dimensions, and if their displacement vectors are such that one is the translation of the other (in the classic geometric sense). For example, the operand $2 * u[x, y, z]$ is translated with respect to the operand $2 * u[x+1, y+2, z+3]$ since they have same coefficient (2), label (u), and dimensions ($[x, y, z]$), while the displacement vectors $[0, 0, 0]$ and $[1, 2, 3]$ are expressible by means of a translation.

Now consider two expressions e_0 and e_1 in fully-expanded form (i.e., a non-nested sum-of-operands). We say that e_0 is an alias of e_1 if the following conditions hold:

- the operands in e_0 (e_1) are expressible as a translation of the operands in e_1 (e_0);
- the same arithmetic operators are applied to the involved operands.

For example, consider $e = u[x] + v[x]$, having two operands $u[x]$ and $v[x]$; then:

- **u[x-1] + v[y-1]** is *not* an alias of e, due to a different dimension vector.
- **u[x] + w[x]** is *not* an alias of e, due to a different label.
- **u[x+2] + v[x]** is *not* an alias of e, since a translation cannot be determined.
- **u[x+2] + v[x+2]** is an alias of e, as the operands $u[x+2]$ and $v[x+2]$ can be expressed as a translation of $u[x]$ and $v[x]$ respectively, with $T(o_d) = o_d + \mathbf{2}$ and o_d representing the displacement vector of an operand.

The relation "e_0 is an alias of e_1" is an equivalence relation, as it is at the same time reflexive, symmetric, and transitive. Thanks to these properties, the turnaround times of the Alias-Detection Algorithm are extremely quick (less than 2 seconds running on an Intel® Intel®Xeon™E5-2620 v4 for the challenging `tti` test case with `so`=16, described in Section 4.7.2), despite the $O(n^2)$ computational complexity (with n representing the number of candidate expressions, see Section 4.6.1).

Algorithm 4 highlights the fundamental steps of the Alias-Detection Algorithm. In the worst case scenario, all pairs of candidate expressions are compared by applying the aliasing definition given above. Aggressive pruning, however, is applied to minimize the cost of the search. The algorithm uses some auxiliary functions: (i) CALCULATE_DISPLACEMENTS returns a mapper associating, to each candidate, its displacement vectors (one for each indexed object); (ii) COMPARE_OPS(e_1, e_2) evaluates to true if e_1 and e_2 perform the same operations on the same operands; (iii) IS_TRANSLATED(d_1, d_2) evaluates to true if the displacement vectors in d_2 are pairwise-translated with respect to the vectors in d_1 by the same factor. Together, (ii) and (iii) are used to establish whether two expressions alias each other (line 8).

Eventually, m sets of aliasing expressions are determined. For each of these sets $G_0, ..., G_{m-1}$, a *pivot* – a special aliasing expression – is constructed. This is the key for operation count reduction: the pivot p_i of $G_i = \{e_0, ..., e_{k-1}\}$ will be used in place of $e_0, ..., e_{k-1}$ (thus obtaining a reduction proportional to k). A simple example is illustrated in Listing 11.

Several optimizations for data locality, not shown in Algorithm 4, are also applied. The interested reader may refer to the documentation and the examples of *Devito v3.1* for more details; below, we only mention the underlying ideas.

- The pivot of G_i is *constructed*, rather than selected out of $e_0, ..., e_{k-1}$, so that it could coexist with as many other pivots as possible within the same `Cluster`. For example, consider again Listing 11: there are infinite possible pivots `temp[x + s] = 9.0*temp0*u[t, x + s]`, and the one with $s = 0$ is chosen. However, this choice is not random: the Alias-Detection Algorithm chooses pivots based on a

Algorithm 4: The Alias-Detection Algorithm (pseudocode).

```
Input: A sequence of expressions $\mathcal{E}$.
Output: A sequence of Alias objects $\mathcal{A}$.
displacements ← CALCULATE_DISPLACEMENTS($\mathcal{E}$);
$\mathcal{A}$ ← list();
unseen ← list($\mathcal{E}$);
while unseen is not empty do
    top ← unseen.pop();
    G = Alias(top);
    for e in unseen do
        if COMPARE_OPS(top, e) and IS_TRANSLATED(displacements[top], displacements[e])
          then
            G.append(e);
            unseen.remove(e);
        end if
    end for
    $\mathcal{A}$.append(G)
end while
return $\mathcal{A}$
```

global optimization strategy, which takes into account all of the m sets of aliasing expressions. The objective function consists of choosing s so that multiple pivots will have identical `ISpace`, and thus be scheduled to the same `Cluster` (and, eventually, to the same loop nest).

- Conservatively, the chosen pivots are assigned to array variables. A second optimization pass, called *index bumping and array contraction* in *Devito v3.1*, attempts to turn these arrays into scalar variables, thus reducing memory consumption. This pass is based on data dependence analysis, which essentially checks whether a given pivot is required only within its `Cluster` or by later `Clusters` as well. In the former case, the optimization is applied.

4.6.3 Loop blocking for working-set minimization

In essence, the CIRE algorithm trades operation for memory – the (array) temporaries to store the aliases. From a run-time performance viewpoint, this is convenient only in arithmetic-intensive kernels. Unsurprisingly, we observed that storing temporary arrays spanning the entire grid rarely provides benefits (e.g., only when the operation count reductions are exceptionally high). We then considered the following options.

1. Capturing redundancies arising along the innermost dimension only. Thus, only scalar temporaries would be necessary. This approach presents three main issues, however: (i) only a small percentage of all redundancies are captured; (ii) the implementation is non-trivial, due to the need for circular buffers in the generated code; (iii) SIMD vectorization is affected, since inner loop iterations are practically serialised. Some previous articles followed this path [168, 169].

2. A generalization of the previous approach: using both scalar and array temporaries, without searching for redundancies across the outermost loop(s). This mitigates issue (i), although the memory pressure is still severely affected. Issue (iii) is also unsolved. This strategy was discussed in [170].

3. Using loop blocking. Redundancies are sought and captured along all available dimensions, although they are now assigned to array temporaries whose size is a function of the block shape. A first loop nest produces the array temporaries, while a subsequent loop nest consumes them, to compute the actual output values. The block shape should be chosen so that writes and reads to the temporary arrays do not cause high latency accesses to the DRAM. An illustrative example is shown in Listing 12.

The CIRE algorithm uses the third approach, based on cross-loop-nest blocking. This pass is carried out by the DLE, which can introduce blocking over sequences of loops (see Section 4.5.2).

Listing 12 The loop nest produced by the CIRE algorithm for the example in Listing 11. Note that the block loop (line 2) wraps both the producer (line 3) and consumer (line 5) loops. For ease of read, unnecessary information are omitted.

```
for t = t_m to t_M:
  for xb = x_m to x_M, xb += blocksize:
    for x = xb to xb + blocksize + 3, x += 1
      temp[x] = 9.0*temp0*u[t, x]
    for x = xb to xb + blocksize; x += 1:
      u[t+1,x,y] = ... + temp[x + 1] - 18.0*temp0*u[t][x + 2] + temp[x + 3] + ...
```

4.7 Performance evaluation

We outline in Section 4.7.1 the compiler setup, computer architectures, and measurement procedure that we used for our performance experiments. Following that, we outline the physical model and numerical setup that define the problem being solved in Section 4.7.2. This leads to performance results, presented in Sections 4.7.3 and 4.7.4.

4.7.1 Compiler and system setup

We analyse the performance of generated code using enriched roofline plots. Since the DSE transformations may alter the operation count by allocating extra memory, only by looking at GFlops/s performance and runtime jointly can a quality measure of code syntheses be derived.

For the roofline plots, Stream TRIAD was used to determine the attainable memory bandwidth of the node. Two peaks for the maximum floating-point performance are shown: the ideal peak, calculated as

$$\#[\text{cores}] \cdot \#[\text{avx units}] \cdot \#[\text{vector lanes}] \cdot \#[\text{FMA ports}] \cdot [\text{ISA base frequency}]$$

and a more realistic one, given by the LINPACK benchmark. The reported runtimes are the minimum of three runs (the variance was negligible). The model used to calculate the operational intensity assumes that the time-invariant `Functions` are reloaded at each time iteration. This is a more realistic setting than a “compulsory-traffic-only” model (i.e., an infinite cache).

We had exclusive access to two architectures: an Intel® Intel®Xeon™Platinum 8180 (formerly code-named Skylake) and an Intel®Xeon Phi™7250 (formerly code-named Knights Landing), which will be referred to as `skl8180` and `knl7250`. Thread pinning was enabled with the program `numactl`. The Intel®compiler `icc version 18.0` was used to compile the generated code. The experiments were run with *Devito v3.1* [13]. The experimentation framework with instructions for reproducibility is available at [182]. All floating point operations are performed in single precision, which is typical for seismic imaging applications.

Any arbitrary sequence of DSE and DLE/YLE transformations is applicable to an `Operator`. Devito, provides three preset optimization sequences, or “modes”, which vary in aggressiveness and affect code generation in three major ways:

- the time required by the Devito compiler to generate the code,
- the potential reduction in operation count, and
- the potential amount of additional memory that might be allocated to store (scalar, tensor) temporaries.

A more aggressive mode might obtain a better operation count reduction than a non-aggressive one, although this does not necessarily imply a better time to solution as the memory pressure might also increase. The three optimization modes – `basic`, `advanced`, and `aggressive`– apply the same sequence of DLE/YLE transformations, which includes OpenMP parallelism, SIMD vectorization, and loop blocking. However, they vary in the number, type, and order of DSE transformations. In particular,

`basic` enables common sub-expressions elimination only;

`advanced` enables `basic`, then factorization, extraction of time-invariant aliases;

`aggressive` enables `advanced`, then extraction of time-varying aliases.

Thus, `aggressive` triggers the full-fledged CIRE algorithm, while `advanced` uses only a relaxed version (based on time invariants). All runs used loop tiling with a block shape that was determined individually for each case using auto-tuning. The auto-tuning phase, however, was not included in the measured experiment runtime. Likewise, the code generation phase is not included in the reported runtime.

4.7.2 Test case setup

In the following sections, we benchmark the performance of operators modeling the propagation of acoustic waves in two different models: isotropic and Tilted Transverse Isotropy (TTI, [183]), henceforth `isotropic` and `tti`, respectively. These operators were chosen for their relevance in seismic imaging techniques [183].

Acoustic `isotropic` modeling is the most commonly used technique for seismic inverse problems, due to the simplicity of its implementation, as well as the comparatively low computational cost in terms of FLOPs. The `tti` wave equation provides a more realistic simulation of wave propagation and accounts for local directional dependency of the wave speed, but comes with increased computational cost and mathematical complexity. For our numerical tests, we use the `tti` wave equation as defined in [183]. The full specification of the equation as well as the finite difference schemes and its implementation using Devito are provided in [13, 43]. Essentially, the `tti` wave equation consists of two coupled acoustic wave equations, in which the Laplacians are constructed from spatially rotated first derivative operators. As indicated by Figure 4.4, these spatially rotated Laplacians have a significantly larger number of stencil coefficients in comparison to its isotropic equivalent which comes with an increased operational intensity.

The `tti` and `isotropic` equations are discretized with second order in time and varying space orders of 4, 8, 12 and 16. For both test cases, we use zero initial conditions, Dirichlet boundary conditions and absorbing boundaries with a 10 point mask (Section 4.3.5). The waves are excited by injecting a time-dependent, but spatially-localized seismic source wavelet into the subsurface model, using Devito's sparse point interpolation and injection as described in Section 4.3.1. We carry out performance measurements for two velocity models of 512^3 and 768^3 grid points with a grid spacing of 20 m. Wave propagation is modeled for 1000 ms, resulting in 327 time steps for `isotropic`, and 415 time steps for `tti`. The time-stepping interval is chosen according to the Courant-Friedrichs-Lewy (CFL) condition [37], which guarantees stability of the explicit time-marching scheme and is determined by the highest velocity of the subsurface model and the grid spacing.

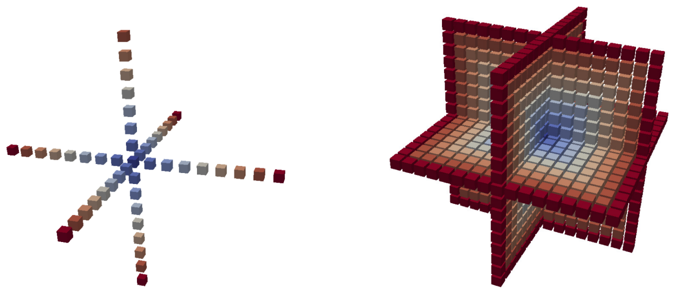


Figure 4.4: Stencils of the acoustic Laplacian for the `isotropic` (left) and `tti` (right) wave equations and `so`=16. The anisotropic Laplacian corresponds to a spatially rotated version of the isotropic Laplacian. The color indicates the distance from the central coefficient.

4.7.3 Performance: acoustic wave in isotropic model

This section illustrates the performance of `isotropic` with the `core` and `yask` backends. To relieve the exposition, we show results for the DSE in `advanced` mode only; the `aggressive` has no impact on `isotropic`, due to the memory-bound nature of the code [43].

The performance of `core` on `skl8180`, illustrated in Figure 4.5 (`yask` uses slightly smaller grids than `core` due to a flaw in the API of *Devito v3.1*, which will be fixed in *Devito v3.2*), degrades as the space order (henceforth, `so`) increases. In particular, it drops from 59% of the attainable machine peak to 36% in the case of `so`=16. This is the result of multiple issues. As `so` increases, the number of streams of unaligned virtual addresses also increases, causing more pressure on the memory system. Intel® VTune™ revealed that the lack of split registers to efficiently handle split loads was a major source of performance degradation. Another major issue for `isotropic` on `core` concerns the quality of the generated SIMD code. The in-line vectorization performed by the auto-vectorizer produces a large number of pack/unpack instructions to move data between vector registers, which introduces substantial overhead. Intel® VTune™ also confirmed that, unsurprisingly, `isotropic` is a memory-bound kernel. Indeed, switching off the DSE basically did not impact the runtime, although it did increase the operational intensity of the four test cases.

The performance of `core` on `knl7250` is not as good as that on `skl8180`. Figure 4.6 shows an analogous trend to that on `skl8180`, with the attainable machine peak systematically dropping as `so` increases. The issue is that here the distance from the peak is even larger. This simply suggests that `core` is failing at exploiting the various levels of parallelism available on `knl7250`.

The `yask` backend overcomes all major limitations to which `core` is subjected. On both `skl8180` and `knl7250`, `yask` outperforms `core`, essentially since it does not suffer from the issues presented above. Vector folding minimizes unaligned accesses; software prefetching helps especially for larger values of `so`; hierarchical OpenMP parallelism is fundamental to leverage shared caches. The speed-up on `knl7250` is remarkable, since even in the best scenario for `core` (`so`=4), `yask` is roughly $3\times$ faster, and more than $4\times$ faster when `so`=12.

4.7.4 Performance: acoustic wave in tilted transverse isotropy model

This sections illustrates the performance of `tti` with the `core` backend. `tti` cannot be run on the `yask` backend in *Devito v3.1* as some fundamental features are still missing; this is part of our future work (more details in Section 4.8).

Unlike `isotropic`, `tti` significantly benefits from different levels of DSE optimizations, which play a key role in reducing the operation count as well as the register pressure. Figure 4.14 displays the performance of `tti` for the usual range of space orders on `skl8180` and `knl7250`, for two different cubic grids.

Generally, `tti` does not reach the same level of performance as `isotropic`. This is not surprising given the complexity of the PDEs (e.g., in terms of differential operators), which translates into code with much higher arithmetic intensity. In `tti`, the mem-

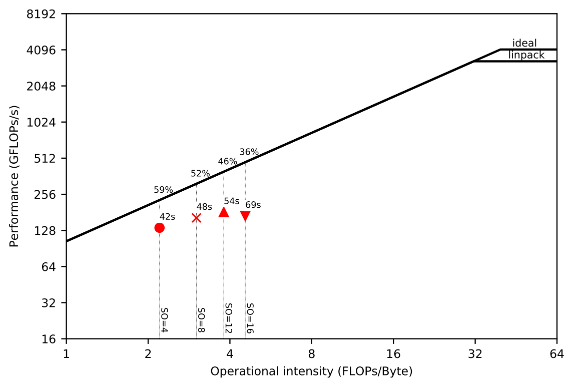


Figure 4.5: `skl8180`, `core`, 768^3 grid points.

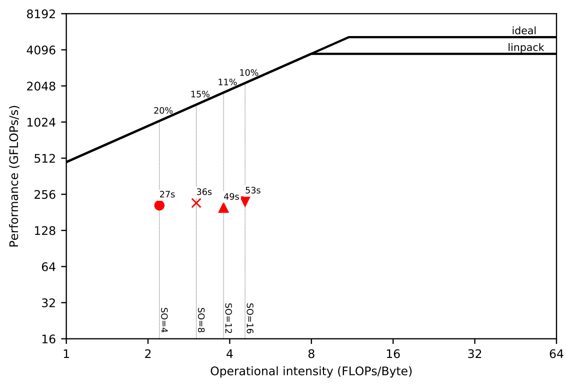


Figure 4.6: `knl7250`, `core`, 768^3 grid points.

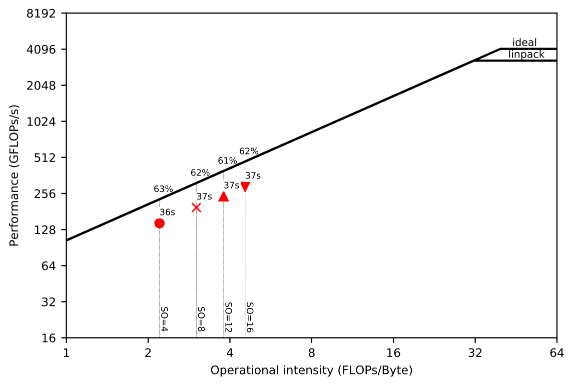


Figure 4.7: `skl8180`, `yask`, 748^3 grid points.

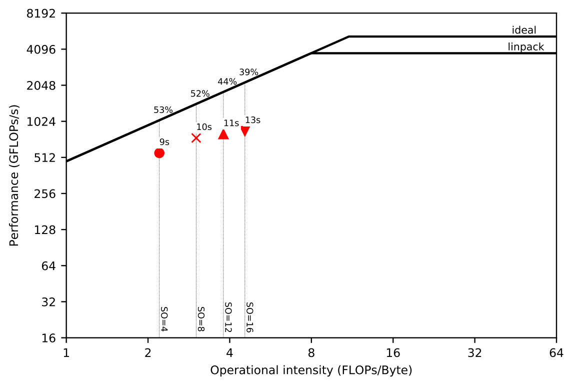


Figure 4.8: `knl7250`, `yask`, 748^3 grid points.

Figure 4.9: Performance of `isotropic` on multiple Devito backends and architectures.

ory system is stressed by a considerably larger number of loads per loop iteration than in `isotropic`. On `skl8180`, we ran some profiling with Intel® VTune™. We determined that one of the major issues is the pressure on both L1 cache (lack of split registers, unavailability of "fill buffers" to handle requests to the other levels of the hierarchy) and DRAM (bandwidth and latency). Clearly, this is only a summary from some sample kernels – the actual situation varies depending on the DSE optimizations as well as the `so` employed.

It is remarkable that on both `skl8180` and `knl7250`, and on both grids, the cutoff point beyond which `advanced` results in worse runtimes than `aggressive` is `so`=8. One issue with `aggressive` is that to avoid redundant computation, not only additional memory is required, but also more data communication may occur through caches, rather than through registers. In Figure 12, for example, we can easily deduce that `temp` is first stored, and then reloaded in the subsequent loop nest. This is an overhead that `advanced` does not pay, since temporaries are communicated through registers, for as much as possible. Beyond `so`=8, however, this overhead is overtaken by the reduction in operation count, which grows almost quadratically with `so`, as reported in Table 4.1.

Table 4.1: Operation counts for different DSE modes in `tti`

so	basic	advanced	aggressive
4	299	260	102
8	857	707	168
12	1703	1370	234
16	2837	2249	300

The performance on `knl7250` is overall disappointing. This is unfortunately caused by multiple factors – some of which already discussed in the previous sections. These results, and more in general, the need for performance portability across future (Intel®or non-Intel®) architectures, motivated the ongoing `yask` project. Here, the overarching issue is the inability to exploit the multiple levels of parallelism typical of architectures such as `knl7250`. Approximately 17% of the attainable peak is obtained when `so`=4 with `advanced` (best runtime out of the three DSE modes for the given space order). This occurs when using 512^3 points per grid, which allows the working set to completely fit in MCDRAM (our calculations estimated a size of roughly 7.5GB). With the larger grid size (Figure 4.13), the working set increases up to 25.5GB, which exceeds the MCDRAM capacity. This partly accounts for the $5\times$ slow down in runtime (from 34s to 173s) in spite of only a $3\times$ increase in number of grid points computed per time iteration.

4.8 Further work

While many simulation and inversion problems such as full-waveform inversion only require the solver to run on a single shared memory node, many other applications require

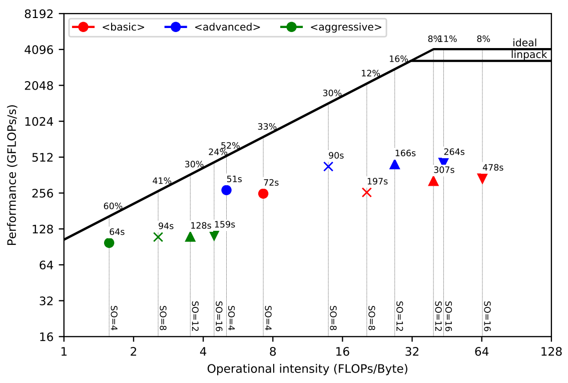


Figure 4.10: `skl8180`, 512^3 grid points.

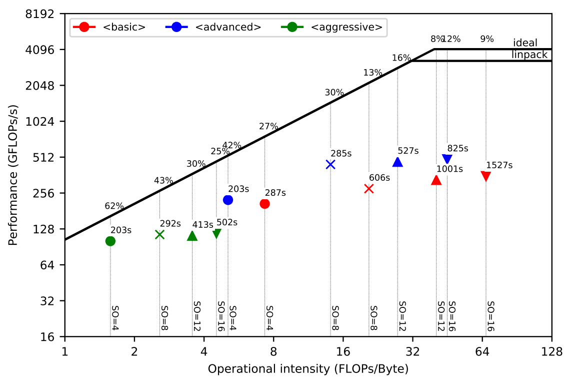


Figure 4.11: `skl8180`, 768^3 grid points.

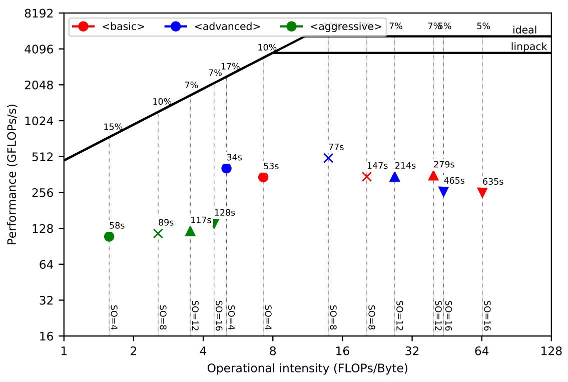


Figure 4.12: `knl7250`, 512^3 grid points.

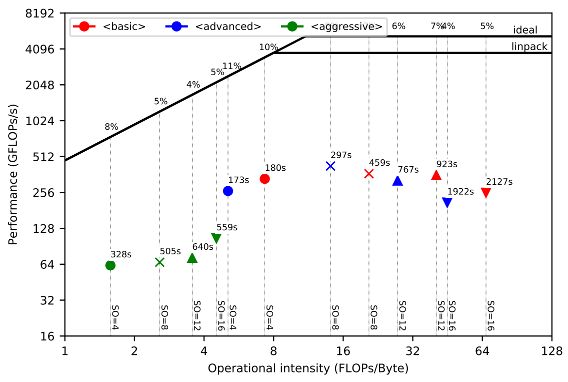


Figure 4.13: `knl7250`, 768^3 grid points.

Figure 4.14: Performance of `tti` on `core` for different architectures and grids.

support for distributed memory parallelism (typically via MPI) so that the solver can run across multiple compute nodes. The immediate plan is to leverage `yask`'s MPI support, and perhaps to include MPI support into `core` at a later stage. Another important feature is staggered grids, which are necessary for a wide range of FD discretization methods (e.g. modelling elastic wave propagation). Basic support for staggered grids is already included in *Devito v3.1*, but currently only through a low-level API – the principle of graceful degradation in action. We plan to make the use of this feature more convenient.

As discussed in Section 4.7.4, the `yask` backend is not feature-complete yet; in particular, it cannot run the `tti` equations in the presence of array temporaries. As `tti` is among the most advanced models for wave propagation used in industry, extending Devito in this direction has high priority.

There also is a range of advanced performance optimization techniques that we want to implement, such as "time tiling" (i.e., loop blocking across the time dimension), on-the-fly data compression, and mixed-precision arithmetic exploiting application knowledge. Finally, there is an on-going effort towards adding an `ops` [163] backend, which will enable code generation for GPUs and also supports distributed memory parallelism via MPI.

4.9 Conclusions

Devito is a system to automate high-performance stencil computations. While Devito provides a *Python* -based syntax to easily express FD approximations of PDEs, it is not limited to finite differences. A Devito `Operator` can implement arbitrary loop nests, and can evaluate arbitrarily long sequences of heterogeneous expressions such as those arising in FD solvers, linear algebra, or interpolation. The compiler technology builds upon years of experience from other DSL-based systems such as FEniCS and Firedrake, and wherever possible Devito uses existing software components including *SymPy* and *NumPy* , and YASK. The experiments in this article show that Devito can generate production-level code with compelling performance on state-of-the-art architectures.

CHAPTER 5
CONCLUSIONS

5.1 Summary

Seismic wave-equation based inversion is an extremely complex problem due to its size and the non-convexity of the optimization problem. In order to develop algorithms for this inverse problem, computationally efficient wave-equation propagators are necessary that can easily scale to large domain sizes (i.e billions of computational grid points). The current general philosophy is to rely on hand-coded propagators targeting specific hardware at hand. Moreover, by necessity, the implementation usually only concerns a specific wave-equation with a fixed discretization (i.e hard coded space orders) that is numerically stable for a very limited range of numerical parameters. The lack of flexibility on the numerical side with hard coded discretization, on the physical side with single wave-equation workflows, as well as the limited hardware portability, are the inherent shortcomings addressed in this thesis. Specifically, the proposed solution involve two distinct steps: the theoretical study of finite-difference solvers using the roofline model and the development, implementation and analysis of Devito, a finite-difference DSL with its own just-in-time compiler.

In the first part, I investigated the theoretical computational properties of finite-difference solvers for a family of wave-equations. This work led to two principal findings: On the one hand, I showed that the runtime is a poor measure of the computational performance of solvers. Runtime performance is based on the underlying assumption that two codes have equivalent optimal runtime, assumption that is proven inaccurate by the roofline model. I consequently adapted the roofline model for analyzing wave equation solvers with finite differences, which provides an absolute measure of the achieved performance compared to the maximum achievable one. The roofline model then allowed me to demonstrate that the computational performance of finite-difference solvers will depend on a combination of numerical parameters and hardware capabilities. This work highlights the need for flexible and portable workflows that allow to choose the best numerical parameters for the hardware at hand.

In the second part, I introduced Devito, a finite-difference DSL, where I offered a flexible alternative to legacy codes where optimized code is generated on the fly at runtime. Compared to these legacy codes, this has the advantage that highly optimized code is generated automatically. This offers the possibility to experiment with different discretizations and optimization strategies designed to make optimal use of the hardware. Additionally, Devito is designed around separation of concerns and offers easy current and future integration with other just-in-time compilers or packages (e.g. julia [153], Tensorflow, PyTorch [184]), portability to the cloud [185], or other architectures such as GPUs [186, 187].

I now summarize in detail these two main contributions.

5.1.1 Performance prediction and evaluation

The computational performance evaluation of a simulation code is a complex problem due to its numerous parameters (discretization, hardware, implementation, reference performance) and requires a rigorous definition of what makes an implementation good. The conventional one-to-one comparison of the runtime between two codes can lead to unfair or erroneous conclusions as it is trivial to implement a slow version of a code to boost the improvement achieved. In Chapter 2, I presented a performance analysis based on the roofline model, a recent performance benchmark tool, that provides an absolute hardware-level measure of performance. This work on performance prediction and evaluation is summarized as follows:

First, I demonstrated that finite-difference solvers have certain theoretical computational properties, such as the number of floating point operations per grid point, that can be estimated from the mathematical expression of finite-difference discretization formulas. This estimation provides insights into the expected maximum achievable performance for a given choice of wave-equation and discretization that can drive the design of a solver for specific hardware. This estimated optimal performance is in GFLOP/s and can then be translated into runtime for a fixed number of grid points and time steps. These performance prediction results show that a flexible design of propagator with respect to the discretization can offer better performance based on hardware specific characteristics.

Second, the roofline model also provides insights into the quality of an implementation and the amount of achievable improvement. The roofline model, on top of being a predictive and theoretical tool, also provides runtime relative performance benchmarks that inform on how well an implementation performs compared to the maximum achievable performance and consequently how much improvement is potentially achievable. This absolute performance measure can also be attached to a conventional runtime benchmark to provide a complete overview of the performance of a given implementation. Without the roofline model, a tremendous amount of work can be sunk into improving the performance of a solver that may already be near-optimal.

5.1.2 Software design with separation of concern

The design of software for computationally demanding applications, such as wave-equation based geophysical exploration and more generally PDE constrained optimization, can be extremely complex as it involves expertise in multiple scientific fields such as physics, mathematics, and computational science. Good software design calls for a separation of concerns, so that each specialist can focus on their respective area of expertise, while still providing a collaborative framework to allow rapid development. In Chapter 3 and 4 of my thesis, I introduced Devito, a finite-difference DSL that is designed around the paradigm of separation of concerns, which is necessary to facilitate efficient collaboration for multi-disciplinary research and development in fields such as exploration geophysics.

Devito is an embedded symbolic domain specific language (DSL). The user interface is symbolic and based on the open source Python package `sympy` [45] that closely follows the mathematical definition of a PDE and allows concise and readable expressions for complex mathematical representations of the physics. Devito's symbolic manipulation then automatically generates the finite-difference stencils from these high-level expressions. Additionally, non-standard operations such as source injection and receiver interpolation (to off the grid source/receiver positions) are necessary for field measurement-based data-fitting inverse problems. As Devito is not only specifically designed for finite-differences, it also provides a framework for generic computations on structured grids, operations such as source/receiver sampling are fully supported and therefore make it suitable for seismic inverse problems. Second, Devito is a code generation framework and a just-in-time compiler. Symbolic DSLs are already well developed, some of the most famous being `sympy` or `mathematica` (variational DSL such as Firedrake [32] were discussed earlier), but are usually based on computationally inefficient symbol replacement for the evaluation of expressions. Devito, on the other hand, implements the translation of symbolic expressions generated from a high-level symbolic interface to highly optimized C-code that is then compiled at runtime. In Chapter 4, I showed that the code-generation framework implements many modern compiler technologies such as vectorization, multi-threading or cache blocking. Therefore, the computational performance achieved is on par with hand-coded equivalent implementations and in some case even outperforms them. The computational performance of Devito makes it suitable not only for research and development but enables at-scale deployment of research codes to production-level validation and application.

To furthermore validate the computational performance of Devito, and thanks to the vectorial extension I present in Section 5.3, I compared the runtime of Devito with a reference open source hand-coded propagator. This propagator, described in [188] is an elastic kernel (c.f. 5.1) and has been implemented by J. W. Thorbecke who is a developer and benchmarker for Cray and Senior researcher in applied geophysics at Delft University of

Technology. The source code can be found at fdelmodc. I compared the runtime of Devito against fdelmodc for a fixed and common computational setup from one of their examples:

- 2001 by 1001 physical grid points.
- 200 grid points of dampening layer on all four sides (total of 2401x1401 computational grid points).
- 10001 time steps.
- Single point source, 2001 receivers.
- Same compiler (gcc/icc) to compile fdelmodc and run Devito.
- Intel(R) Xeon(R) CPU E3-1270 v6 @ 3.8GHz.
- Single socket, four physical cores, four physical threads, thread pinning to cores and hyperthreading off.

The runtime results are summarized in Table 5.1 and show on average that Devito performs around 75% faster with the intel compiler and 40% faster with gcc.

Table 5.1: Runtime comparison between Devito and FDELMODC [188] for a two dimensional elastic model. The first column shows the kernel execution time (call to the generated C code) of Devito and the second column shows the total runtime including code generation and compilation. Only the kernel execution time of FDELMODC is shown as the libraries are precompiled.

Compiler	Devito kernel	Devito runtime	FDELMODC kernel	Kernel speedup	Runtime speedup
GCC 9.2.0	166.07s	172.83s	237.52s	1.430	1.374
ICC 19.1.0	131.59s	136.85	237.17s	1.802	1.733

This comparison illustrate the performance achieved with Devito is at least on par with hand coded propagators. Future work will aim at providing a thorough benchmark by comparing first against a three dimensional implementations and second against state of the art stencil language.

The combination of a symbolic DSL with a just-in-time compiler makes Devito a perfect tool for seismic inverse problems. As months of manual implementations of low-level propagators are reduced to a few lines of high-level Python code, geophysicists and mathematicians can concentrate on algorithms for inverse problem, while computer scientist can still develop the compiler technology through the code-generation API. The separation of concerns applied to every level of Devito (compiler, API, seismic wrappers) also allows easy unit-testing and safeguards against large monolithic code-base. Finally, unlike similar academic projects, Devito is now quickly being adapted by the broader geoscience community and multiple projects were and are currently making use of Devito. I highlight some of these projects now.

5.2 Enabling research

As motivated in the introduction and through the content of my thesis, Devito is designed to enable and accelerate research and development. The main impact factor and validation for Devito is its current usage by the scientific community. Devito enabled projects in multiple fields for a wide range of users. Among these projects are academics projects that I have been actively involved in and collaborated with, as well as industry research and development and even running Devito at scale in production codes. While I am not at liberty to discuss industry usage of Devito and my work, I now briefly describe my contribution to two projects that were enabled by Devito and demonstrate that my work drove the development of new technologies. In the following sections, I will first discuss JUDI [153], a Julia linear algebra DSL for seismic inverse problem that interfaces and uses Devito for the solves of the wave-equation while providing a high-level linear algebra framework for implementing and solving mathematical inverse problems. Second of all, I will describe and illustrate the implementation industry-scale seismic imaging in the cloud (specifically on Microsoft Azure) using Devito [189, 185, 190].

5.2.1 JUDI: The Julia Devito Inversion framework

The Julia Devito Inversion framework (JUDI[1]) [153, 78, 30] is a parallel matrix-free linear algebra DSL for seismic modeling and inversion based on Devito [42, 13] and SegyIO [191], a performant Julia package for reading and writing large data volumes in SEG-Y format [192]. JUDI allows implementing seismic inversion algorithms as linear algebra operations, enabling rapid translations of seismic inversion algorithms to executable Julia code. The underlying wave equations are set up and solved using Devito and are interfaced from Julia using the `PyCall` package. As described in the Introduction 1 and Chapter 3, seismic inversion can be formulated in a linear algebra way that allows for a simple expression of the problem and clearer definition and implementation of inversion algorithms. The projection and modeling operators, that wrap around Devito's propagators, can be set up in JUDI in the following way:

```
ntComp = get_computational_nt(q.geometry, d_obs.geometry,
   model0)
info = Info(prod(model0.n), d_obs.nsrc, ntComp)
Pr = judiProjection(info, d_obs.geometry)
Ps = judiProjection(info, q.geometry)
Ainv = judiModeling(info, model0)
```

Seismic shot records (active-source seismic experiments) can then be modeled by running the matrix-free:

```
d_pred = Pr*Ainv*adjoint(Ps)*q
```

from the Julia command line, which is equivalent to the mathematical expression $F(\mathbf{m}; \mathbf{q}) = \mathbf{P}_r \mathbf{A}^{-1}(\mathbf{m}) \mathbf{P}_s^\top \mathbf{q}$ by virtue of the instantiation `Ainv = judiModeling(info, model0)`, which makes the wave equation solver implicitly dependent on the model defined in the

[1]A modified version of this description of JUDI was published in The Leading Edge [78] as apart of a three part tutorial on seismic modelling and inversion, the first two parts concentrating on modeling [76] and adjoint modeling and computing the FWI gradient [77] with Devito.

variable `model0`. If we started our Julia session with multiple CPU cores or nodes (`julia -p n`, with `n` being the number of workers), the wave equation solves are automatically parallelized over source locations and all shots are collected in the `d_pred` vector. We can also model a single or subset of shots by indexing the operators with the respective shot numbers. E.g. if we want to model the first two shots, we define `i=[1,2]` and then run:
`d_sub = Pr[i]*Ainv[i]*adjoint(Ps)[i]*q[i]`.
If we want to solve an adjoint wave equation with the observed data as the adjoint source and restrictions of the wavefields back to the source locations, we can simply run:
`qad = Ps*adjoint(Ainv)*adjoint(Pr)*d_obs`,
exemplifying the advantages of casting FWI in a proper computational linear algebra framework. Accordingly, a basic gradient descent (GD)example with a line search can be implemented in a few lines of Julia code. To reduce the computational cost of full GD, we will use a stochastic gradient descent (SGD) in which we only compute the gradient and objective value for a randomized subset of source locations. In JUDI, this is accomplished by choosing a random vector of integers for the sources and indexing the data vectors as described earlier. Furthermore, a box constraints is applied to the updated model, to prevent velocities (or squared slownesses) to become negative or too large. Bound constraints are applied to the updated model trough a projection operator `proj(x)`, which clips values of the slowness that lie outside the allowed range. The full algorithm for FWI with stochastic gradient descent and box constraints is implemented as follows in JUDI:

```
maxiter = 10
batchsize = 8   # number of shots for each iteration
proj(x) = reshape(median([vec(mmin) vec(x) vec(mmax)],2),
   model0.n)

for j=1:maxiter

        # FWI objective function value and gradient
        i = randperm(d_obs.nsrc)[1:batchsize]   # select
           random source locations
        fval, grad = fwi_objective(model0, q[i], d_obs[i])

        # line search and update model
        update = backtracking_linesearch(model0, q[i], d_obs
           [i], fval, grad, proj; alpha=1f0)
        model0.m += reshape(update, model0.n)

        # apply box constraints
        model0.m = proj(model0.m)
end
```

The function `backtracking_linesearch` performs an approximate line search and returns a model update that leads to a sufficient decrease of the FWI function value and is part of the JUDI optimization sub-module `JUDI.SLIM_optim`. JUDI is the perfect example of vertical integration of a software stack, that starts with low level C-code optimization, followed by Devito symbolic interface for the definition of the PDEs and finally JUDI's linear algebra DSL for the definition of an optimization algorithm to solve the inverse problem. This project was well received by both the academic and industry in the seismic community and was highlighted by the leading Edge as part of their brightspot advertisement [193].

5.2.2 Imaging in the cloud

For the major part of my Ph.D, I was lucky to have had access to a moderately size on-premise HPC cluster for running numerical experiments, without much consideration to cost. However, in recent years, our laboratory has made the choice to move the cloud. One of the main challenges in modern HPC is to modernize legacy codes for the cloud, which are usually hand-tuned or designed for on-premise clusters with a known and fixed architecture and setup. Porting these codes and algorithms to the cloud can be straightforward using a lift-and-shift strategy that essentially boils down to renting a cluster in the cloud. However, this strategy is not cost-efficient. Pricing in the cloud is typically based on a pay-as-you-go model, which charges for requested computational resources, regardless of whether or not instances and cores are actively used or sit idle. This pricing model is disadvantageous for the lift-and-shift strategy and oftentimes incurs higher costs than required by the actual computations, especially for communication-heavy but task-based algorithms that only need partial resources depending of the stage of the computation. On the other hand, serverless software design provides flexible and cost efficient usage of cloud resources including for large scale inverse problem such as seismic inversion. With Devito, we had access to a portable yet computationally efficient framework for wave-equation based seismic exploration that allowed us to quickly develop a new strategy to execute seismic inversion algorithms in the cloud. This new serverless and event-driven approach led to significant early results [190, 189] that caught the attention of both seismic inverse problems practitioners and cloud providers. This led to a proof of concept project on an industry size problem in collaboration with Microsoft Azure. The main objectives of this project were:

- Demonstrate the scalability, robustness and cost effectiveness of a serverless implementation of seismic imaging in the cloud. In this case, we imaged a synthetic three dimensional anisotropic subsurface model that mimics a realistic industry size problem (10km by 10km by 5km) with a realistic representation of the physics (TTI).

- Demonstrate the flexibility and portability of Devito. The seismic image (RTM as defined in Chapter 3) was computed with Devito and highlights the code-generation and high performance capability of Devito on an at-scale real world problem. This results shows that in addition to conventional benchmark metrics such as soft and hard scaling and the roofline model, Devito provides state of the art performance on practical applications as well.

The subsurface velocity model that was used in this study is an artificial anisotropic model I designed and built combining two broadly known and used open-source SEG/EAGE acoustic velocity models. The anisotropy parameters are derived from smoothed version of the velocity wile the tilt angles were derived from a combination of the smooth velocity models and vertical and horizontal edge detection. The final seismic image of the subsurface model is plotted in Figure 5.1 and highlights the fact that 3D seismic imaging based on a serverless approach and automatic code generation is feasible and provides good results on a realistic model.

In [189] is fully describes the serverless implementation of seismic inverse problems, including iterative algorithms for least square minimization problems (LSRTM). The 3D anisotropic imaging results were presented as part of a keynote presentation at the EAGE HPC workshop in October 2019 [185]. This work perfectly illustrates the flexibility and portability of Devito, as we were able to easily port a code only tested and developed on local hardware to the cloud, with only requiring minor adjustments. This portability included the possibility to run MPI-based code for domain decomposition in the cloud, after developing it on a desktop computer.

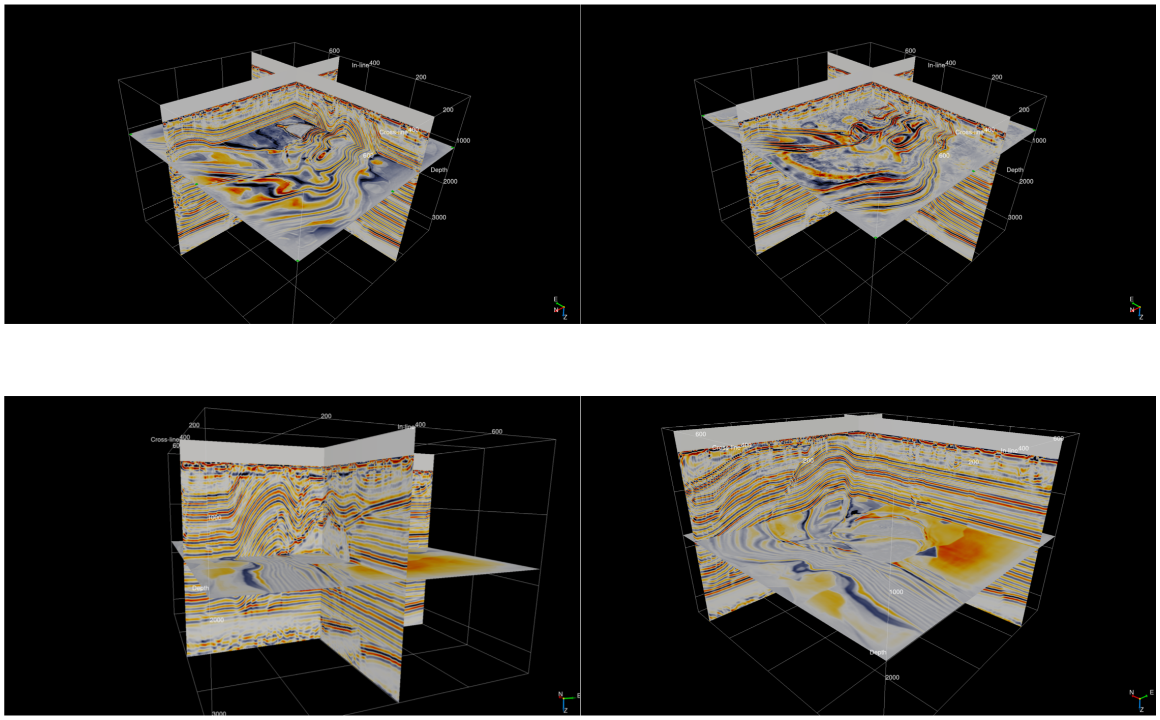


Figure 5.1: 3D TTI imaging on a custom made model.

5.3 Extended API

While the acoustic wave-equation is a good start to understand wave physics and wave propagation, it provides a very crude approximation of the true physics of the earth. In order to realistically model wave propagation in the earth, the elastic wave-equation is necessary. The elastic wave-equation 5.1 is however vectorial and requires more advanced numerical methods to be discretized and solved. The elastic isotropic wave-equation, parametrized by the Lamé parameters λ, μ and the density ρ reads:

$$\begin{aligned} &\frac{1}{\rho}\frac{dv}{dt} = \nabla.\tau \\ &\frac{d\tau}{dt} = \lambda tr(\nabla v)\mathbf{I} + \mu(\nabla v + (\nabla v)^T) \end{aligned} \tag{5.1}$$

where v is a vector valued function with one component per cartesian direction:

$$v = \begin{bmatrix} v_x(t,x,y,z) \\ v_y(t,x,y,z) \\ v_z(t,x,y,z) \end{bmatrix} \tag{5.2}$$

and the stress τ is a symmetric second-order tensor-valued function:

$$\tau = \begin{bmatrix} \tau_{xx}(t,x,y,z) & \tau_{xy}(t,x,y,z) & \tau_{xz}(t,x,y,z) \\ \tau_{xy}(t,x,y,z) & \tau_{yy}(t,x,y,z) & \tau_{yz}(t,x,y,z) \\ \tau_{xz}(t,x,y,z) & \tau_{yz}(t,x,y,z) & \tau_{zz}(t,x,y,z) \end{bmatrix}. \tag{5.3}$$

The discretization of such a set of coupled PDEs requires 5 equations in two dimensions (two equations for particle velocity and three for stress) and 9 equations in three dimensions (three particle velocities and six stress equations). However the mathematical definition only requires two for any number of dimensions. The main contribution of this work is to extend the previously scalar-only capabilities of Devito to vector and second order tensors and allow a straightforward and mathematical definition of high-dimensional PDEs such as the elastic wave equation in Equation 5.1. Once again, based on `sympy`, I recently extended the symbolic interface to vectorial and tensorial object to allow for a straightforward definition of equations such as the elastic wave-equation, as well as computational fluid dynamics equations. The extended API defines two new types, `VectorFunction` (and `VectorTimeFunction`) for vectorial objects such as the particle velocity, and `TensorFunction` (and `TensorTimeFunction`) for second order tensor objects (matrices) such as the stress. These new objects are constructed the exact same way as the previously scalar `Function` objects and automatically implement staggered grid and staggered finite-differences with the possibility of half-node averaging. This new extended API now allows users to define the elastic wave-equation in four lines as follows:

The `sympy` expressions created by these commands can be displayed via the `sympy`

```
v = VectorTimeFunction(name='v', grid=model.grid, space_order=so, time_order=1)
tau = TensorTimeFunction(name='t', grid=model.grid, space_order=so, time_order=1)

u_v = Eq(v.forward, model.damp * (v + s/rho*div(tau)))
u_t = Eq(tau.forward,  model.damp *  (tau + s * (l * diag(div(v.forward)) +
                                     mu * (grad(v.forward) + grad(v.forward).T))))
```

Particle velocity update u_v :

$$\begin{bmatrix} \mathrm{v_x}\left(t + dt, x + \frac{h_x}{2}, y\right) \\ \mathrm{v_y}\left(t + dt, x, y + \frac{h_y}{2}\right) \end{bmatrix} = \begin{bmatrix} \left(dt\left(\frac{\partial}{\partial x}\mathrm{t_{xx}}\left(t, x, y\right) + \frac{\partial}{\partial y}\mathrm{t_{xy}}\left(t, x + \frac{h_x}{2}, y + \frac{h_y}{2}\right)\right)\mathrm{irho}\left(x, y\right) + \mathrm{v_x}\left(t, x + \frac{h_x}{2}, y\right)\right)\mathrm{damp}\left(x, y\right) \\ \left(dt\left(\frac{\partial}{\partial x}\mathrm{t_{xy}}\left(t, x + \frac{h_x}{2}, y + \frac{h_y}{2}\right) + \frac{\partial}{\partial y}\mathrm{t_{yy}}\left(t, x, y\right)\right)\mathrm{irho}\left(x, y\right) + \mathrm{v_y}\left(t, x, y + \frac{h_y}{2}\right)\right)\mathrm{damp}\left(x, y\right) \end{bmatrix}$$

Figure 5.2: Update stencil for the particle velocity. The stencil for updating the stress component is left out for readability, as the equation does not fit onto a single page. However, it can be found in the Devito tutorial on elastic modeling.

pretty printer (`sympy.pprint`) as shown in Figure 5.2. This representation reflects perfectly the mathematics while still providing computational portability and efficiency through the Devito compiler.

Each component of a vectorial or tensorial object is accessible via conventional vector and matrix indices (i.e. `v[0], t[0,1],....`). I show the elastic particle velocity and stress for a well known 2D synthetic model, the elastic marmousi-ii [194, 152] model. The wavefields are shown on Figure 5.3 and its corresponding elastic shot records are displayed in Figure 5.4.

Figure 5.2 also highlights a second new feature: delayed evaluation. The equation is still a differential equation, and the finite-difference schemes are now evaluated either through an explicit user request via the `.evaluate` call, or automatically at the compiler level. This delayed evaluation of derivatives provides a cleaner and more user friendly interface to Devito, which prevents complex and lengthy finite-difference schemes being displayed after defining the symbolic expressions. Figure 5.5 illustrates this new feature showing the user-facing representation of a derivative and its evaluation.

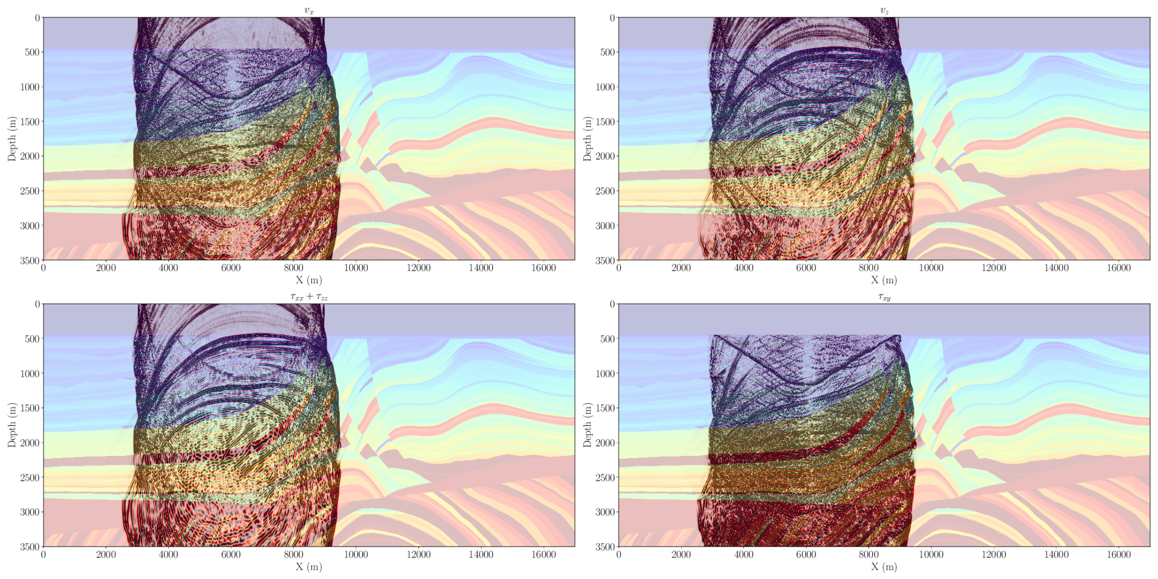


Figure 5.3: Particle velocities and stress at time $t = 3$s for a source at 10m depth and x=5\text{km} in the marmousi-ii model.

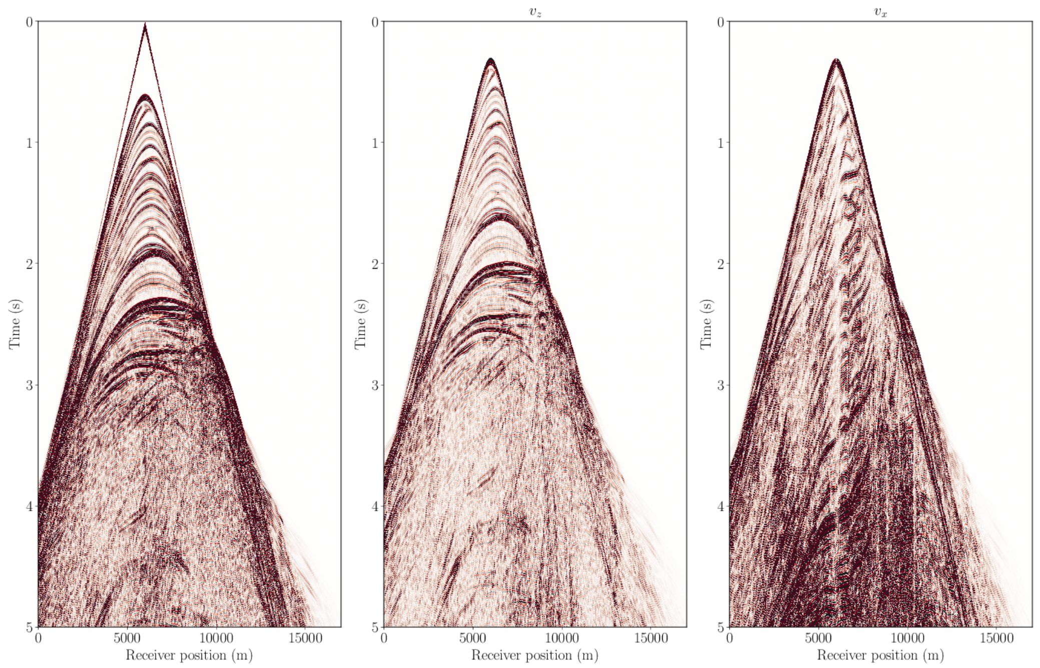


Figure 5.4: Seismic shot record for 5sec of modeling. a is the pressure (trace of stress tensor) at the surface (5m depth), b is the vertical particle velocity and c is the horizontal particle velocity at the ocean bottom (450m depth).

```
In [1]: from devito import *

In [2]: grid = Grid((10, 10 ,10))

In [3]: u = TimeFunction(name="u", grid=grid, space_order=2, time_order=2)
        v = VectorTimeFunction(name="v", grid=grid, space_order=2, time_order=2)

In [4]: u.dx
```

Out[4]:

$$\frac{\partial}{\partial x}u(t,x,y,z)$$

```
In [5]: u.dx.evaluate
```

Out[5]:

$$-\frac{u(t,x,y,z)}{h_x}+\frac{u(t,x+h_x,y,z)}{h_x}$$

```
In [6]: u.laplace
```

Out[6]:

$$\frac{\partial^2}{\partial x^2}u(t,x,y,z)+\frac{\partial^2}{\partial y^2}u(t,x,y,z)+\frac{\partial^2}{\partial z^2}u(t,x,y,z)$$

```
In [7]: u.laplace.evaluate
```

Out[7]:

$$-\frac{2.0u(t,x,y,z)}{h_z^2}+\frac{u(t,x,y,z-h_z)}{h_z^2}+\frac{u(t,x,y,z+h_z)}{h_z^2}-\frac{2.0u(t,x,y,z)}{h_y^2}+\frac{u(t,x,y-h_y,z)}{h_y^2}+\frac{u(t,x,y+h_y,z)}{h_y^2}-\frac{2.0u(t,x,y,z)}{h_x^2}+\frac{u(t,x-h_x,y,z)}{h_x^2}+\frac{u(t,x+h_x,y,z)}{h_x^2}$$

```
In [8]: div(v)
```

Out[8]:

$$\frac{\partial}{\partial x}\mathrm{v_x}\left(t,x+\frac{h_x}{2},y,z\right)+\frac{\partial}{\partial y}\mathrm{v_y}\left(t,x,y+\frac{h_y}{2},z\right)+\frac{\partial}{\partial z}\mathrm{v_z}\left(t,x,y,z+\frac{h_z}{2}\right)$$

```
In [9]: div(v).evaluate
```

Out[9]:

$$-\frac{\mathrm{v_z}\left(t,x,y,z-\frac{h_z}{2}\right)}{h_z}+\frac{\mathrm{v_z}\left(t,x,y,z+\frac{h_z}{2}\right)}{h_z}-\frac{\mathrm{v_y}\left(t,x,y-\frac{h_y}{2},z\right)}{h_y}+\frac{\mathrm{v_y}\left(t,x,y+\frac{h_y}{2},z\right)}{h_y}-\frac{\mathrm{v_x}\left(t,x-\frac{h_x}{2},y,z\right)}{h_x}+\frac{\mathrm{v_x}\left(t,x+\frac{h_x}{2},y,z\right)}{h_x}$$

Figure 5.5: Illustration of the lazy evaluation feature on a simple derivative. This examples shows that the symbolic representation of the derivatives matches the mathematical definition and that it's evaluation automatically generate the finite difference stencil associated with it.

5.4 Future work

Finally, I briefly discuss possible future work and open questions. While Devito has already grown into a very flexible framework, there are additional research directions that are still open and require future work:

- Topology (non-flat physical interfaces such as ocean bottoms). Currently, topology is not supported in any high-level and concise way that would allow users to easily formulate complex boundaries such has mountain ranges or non-flat ocean bottoms. While these issue are usually fairly irrelevant in the acoustic case, more advanced physics such as elasticity require proper definitions of the fluid-solid interfaces to avoid artifacts and accurately represent surface waves. This extension would fall under what can be defined as `SubDomain` extensions, that allow users to define objects and equations on partial parts of the `Grid` with interface continuation conditions (equations that links the fields between two `SubDomain`). Such conditions are close to trivial to implement by hand in low-level finite-difference codes, but a robust high-level and user-friendly interface is more complex. Preliminary work is currently in development at Imperial College London.

- Hardware portability. As I explained throughout this thesis, architecture portability is one of the main challenges when it comes to computational methods and software. As Devito is its own compiler, support for multiple architecture is already possible, but is currently limited to CPUs (intel, AMD), intel Xeon Phis (KNC, KNL), and ARM chips [195]. One of the main interests from the community is to extend the hardware portability to other types of accelerators, namely to GPUs and potentially FPGAs. These hardwares are currently not supported, or at the very least not for the full set of operation Devito permits, but is currently in active development. Offering support for graphic cards accelerators would greatly improve the impact of Devito and its range of potential applications. Two main ideas are currently investigated concerning GPUs. First, the `ops` backend [163] is in development that interfaces the stencil loop DSL (Oxford Parallel library for Structured meshes) in a similar way the `yask` backend (see Chapter 4) is setup. Second, we are investigating the new capabilities of openMP-5 and its GPU offloading tools that would allow to generate GPU code from OpenMP pragmas only and let the compiler translate and offload the computation. Early results on the OpenMP offloading are promising.

- Complex arithmetic. Currently, Devito only supports standard `numpy` ctypes that do not include complex valued numbers. Complex values would be a great addition to Devito as it would enable to solve problems such as Fourier based methods or electromagnetism easily. Currently, these kind of operations need to be implemented by hand by manually splitting real/imaginary parts of the problem arithmetic operations, such as multiplications. Support of complex arrays would require to implement an extension of `numpy` ctypes for complex numbers, which is already supported by native `numpy`, making this extension generally straight forward. The integration of complex numbers into Devito would then be trivial as all Devito objects are already `numpy` arrays and complex arithmetic can be integrated into the compiler.

- Machine Learning. The Devito compiler is designed to be optimal for stencil computations such as finite-differences, or in other words convolutional kernels, that are at the core of convolutional networks. It should straightforward to support machine learning applications with Devito and scale to large images or videos. Another research direction is to use Devito for PDE based machine learning where a single layer may be represented by a wave-equation such as in [196].

Appendices

APPENDIX A
APPENDIX

A.1 Distribution and copyrights

This section provides the proper distributions rights from the hournal for the papers in Chapter 2 and 3. As of Chapter 4, the paper is still in review and therefore usable in the Thesis with the authorization of the main author below.

Chapter 2 is published in Computer and Geoscience as an open source article. As stated at `https://www.elsevier.com/about/policies/copyright#personaluse` the author (me) can use the article in my Thesis for non commerical use.

Chapter 3 is published in Geoscientific Model Development `https://www.geoscientific-mo` `net` and the author (me) retains full copyright of the paper and is authorized to copy and share its content for example as part of a Thesis as stated at `https://www.geoscientific-model-` `net/about/licence_and_copyright.html`.

Chapter 4 is currently being proceessed to be published in the ACM Transactions on Mathematical Software (TOMS) as an open access paper giving authorization to reuse it as part of my thesis `https://authors.acm.org/author-services/author-rights`. The article is currently under Creative Commons on Arxiv that allows me to use it in my thesis.

REFERENCES

[1] S. J. Baines and R. H. Worden, “Geological storage of carbon dioxide,” *Geological Society, London, Special Publications*, vol. 233, no. 1, pp. 1–6, 2004.

[2] D. Lumley, “4d seismic monitoring of co 2 sequestration,” *The Leading Edge*, vol. 29, no. 2, pp. 150–155, 2010.

[3] H. Aochi, T. Ulrich, A. Ducellier, F. Dupros, and D. Michea, “Finite difference simulations of seismic wave propagation for understanding earthquake physics and predicting ground motions: Advances and challenges,” *Journal of Physics: Conference Series*, vol. 454, p. 012 010, Aug. 2013.

[4] A. W. Frederiksen and M. G. Bostock, “Modelling teleseismic waves in dipping anisotropic structures,” *Geophysical Journal International*, vol. 141, no. 2, pp. 401–412, May 2000. eprint: `http://oup.prod.sis.lan/gji/article-pdf/141/2/401/1825416/141-2-401.pdf`.

[5] M. G. Bostock and S. Rondenay, “Migration of scattered teleseismic body waves,” *Geophysical Journal International*, vol. 137, no. 3, pp. 732–746, Jun. 1999. eprint: `http://oup.prod.sis.lan/gji/article-pdf/137/3/732/1695373/137-3-732.pdf`.

[6] A. Tarantola, “Inversion of seismic reflection data in the acoustic approximation,” *GEOPHYSICS*, vol. 49, no. 8, p. 1259, 1984. eprint: `/gsw/content_public/journal/geophysics/49/8/10.1190_1.1441754/5/1259.pdf`.

[7] B. Hustedt, S. Operto, and J. Virieux, “A multi-level direct-iterative solver for seismic wave propagation modelling: Space and wavelet approaches,” *Geophysical Journal International*, vol. 155, no. 3, pp. 953–980, 2003. eprint: `/oup/backfile/content_public/journal/gji/155/3/10.1111/j.1365-246x.2003.02098.x/3/155-3-953.pdf`.

[8] C. C. Stolk, “A rapidly converging domain decomposition method for the helmholtz equation,” *Journal of Computational Physics*, vol. 241, pp. 240–252, 2013.

[9] C. C. Stolk, M. Ahmed, and S. K. Bhowmik, “A multigrid method for the helmholtz equation with optimized coarse grid corrections,” *SIAM Journal on Scientific Computing*, vol. 36, no. 6, A2819–A2841, 2014. eprint: `https://doi.org/10.1137/13092349X`.

[10] C. Da Silva and F. Herrmann, "A unified 2d/3d software environment for large-scale time-harmonic full-waveform inversion," in *SEG Technical Program Expanded Abstracts 2016*. 2016, pp. 1169–1173. eprint: `http://library.seg.org/doi/pdf/10.1190/segam2016-13869051.1`.

[11] W. W. Symes, D. Sun, and M. Enriquez, "From modelling to inversion: Designing a well-adapted simulator," *Geophysical Prospecting*, vol. 59, no. 5, pp. 814–833, 2011.

[12] S. Fomel et. al, *Madagascar: Open-source software project for multidimensional data analysis and reproducible computational experiments*, 2013.

[13] M. Louboutin, M. Lange, F. Luporini, N. Kukreja, P. A. Witte, F. J. Herrmann, P. Velesko, and G. J. Gorman, "Devito (v3.1.0): An embedded domain-specific language for finite differences and geophysical exploration," *Geoscientific Model Development*, vol. 12, no. 3, pp. 1165–1187, 2019.

[14] J. F. Claerbout, *Imaging the Earth's Interior*. Cambridge, MA, USA: Blackwell Scientific Publications, Inc., 1985, ISBN: 0-86542-304-0.

[15] R. G. Pratt, "Seismic waveform inversion in the frequency domain, part 1: Theory and verification in a physical scale model," *GEOPHYSICS*, vol. 64, no. 3, pp. 888–901, 1999. eprint: `https://doi.org/10.1190/1.1444597`.

[16] W. Mulder and R.-E. Plessix, "Exploring some issues in acoustic full waveform inversion," *Geophysical Prospecting*, vol. 56, no. 6, pp. 827–841, 2008.

[17] J. Virieux and S. Operto, "An overview of full-waveform inversion in exploration geophysics," *GEOPHYSICS*, vol. 74, no. 5, WCC1–WCC26, 2009. eprint: `http://library.seg.org/doi/pdf/10.1190/1.3238367`.

[18] G. Huang, R. Nammour, and W. Symes, "Full-waveform inversion via source-receiver extension," *GEOPHYSICS*, vol. 82, no. 3, R153–R171, 2017. eprint: `http://geophysics.geoscienceworld.org/content/82/3/R153.full.pdf`.

[19] W. W. Symes, "The search for a cycle-skipping cure: An overview," in *Institute for Pure and Applied Mathematics (IPAM): Computational Issues in Oil Field Applications*, 2017.

[20] T. van Leeuwen and F. J. Herrmann, "A penalty method for PDE-constrained optimization in inverse problems," *Inverse Problems*, vol. 32, no. 1, p. 015 007, Dec. 2015, (Inverse Problems).

[21] E. Esser, L. Guasch, T. van Leeuwen, A. Y. Aravkin, and F. J. Herrmann, "Total-variation regularization strategies in full-waveform inversion," *ArXiv e-prints*, Aug. 2016. arXiv: `1608.06159 [math.OC]`.

[22] M. Louboutin and F. J. Herrmann, "Time compressively sampled full-waveform inversion with stochastic optimization," in *SEG Technical Program Expanded Abstracts*, (SEG, New Orleans), Oct. 2015, pp. 5153–5157.

[23] ——, "Extending the search space of time-domain adjoint-state fwi with randomized implicit time shifts," in *EAGE Annual Conference Proceedings*, (EAGE, Paris), Jun. 2017.

[24] B. Peters and F. J. Herrmann, "Constraints versus penalties for edge-preserving full-waveform inversion," *The Leading Edge*, vol. 36, no. 1, pp. 94–100, 2017. eprint: `http://dx.doi.org/10.1190/tle36010094.1`.

[25] B. Peters, B. R. Smithyman, and F. J. Herrmann, "Projection methods and applications for seismic nonlinear inverse problems with multiple constraints," *GEOPHYSICS*, vol. 84, no. 2, R251–R269, 2019. eprint: `https://doi.org/10.1190/geo2018-0192.1`.

[26] N. Kukreja, J. Hückelheim, M. Louboutin, K. Hou, P. Hovland, and G. Gorman, "Combining checkpointing and data compression to accelerate adjoint-based optimization problems," Submitted to PASC19 on January 16, 2019, 2019.

[27] A. Griewank and A. Walther, "Algorithm 799: Revolve: An implementation of checkpoint for the reverse or adjoint mode of computational differentiation," *ACM Transactions on Mathematical Software*, vol. 26, no. 1, pp. 19–45, 2000, Also appeared as Technical University of Dresden, Technical Report IOKOMO-04-1997.

[28] Symes, "Reverse time migration with optimal checkpointing," *GEOPHYSICS*, vol. 72, no. 5, SM213–SM221, 2007. eprint: `http://library.seg.org/doi/pdf/10.1190/1.2742686`.

[29] N. Kukreja, J. Hückelheim, M. Lange, M. Louboutin, A. Walther, S. W. Funke, and G. Gorman, "High-level python abstractions for optimal checkpointing in inversion problems," *arXiv preprint arXiv:1802.02474*, 2018.

[30] P. A. Witte, M. Louboutin, F. Luporini, G. J. Gorman, and F. J. Herrmann, "Compressive least-squares migration with on-the-fly fourier transforms," *Geophysics*, vol. 84, no. 5, R655–R672, Aug. 2019, (Geophysics).

[31] A. Logg, K.-A. Mardal, G. N. Wells, *et al.*, *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012, ISBN: 978-3-642-23098-1.

[32] F. Rathgeber, D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. T. McRae, G. Bercea, G. R. Markall, and P. H. J. Kelly, "Firedrake: Automating the finite element method by composing abstractions," *CoRR*, vol. abs/1501.01809, 2015.

[33] M. S. Alnæs, A. Logg, K. B. Ølgaard, M. E. Rognes, and G. N. Wells, "Unified Form Language: A domain-specific language for weak formulations of partial differential equations," *ACM Transactions on Mathematical Software (TOMS)*, vol. 40, no. 2, p. 9, 2014.

[34] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 3rd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007, ISBN: 0123706068, 9780123706065.

[35] J. L. Lions, *Optimal control of systems governed by partial differential equations*, 1st. Springer-Verlag Berlin Heidelberg, 1971, ISBN: 978-3-642-65026-0.

[36] R. Courant, "Variational methods for the solution of problems of equilibrium and vibrations," *Bull. Amer. Math. Soc.*, vol. 49, no. 1, pp. 1–23, Jan. 1943.

[37] R. Courant, K. Friedrichs, and H. Lewy, "On the partial difference equations of mathematical physics," *International Business Machines (IBM) Journal of Research and Development*, vol. 11, no. 2, pp. 215–234, Mar. 1967.

[38] K. Bathe, "Finite element method," in *Wiley Encyclopedia of Computer Science and Engineering*, 2008.

[39] R. Ye, M. V. de Hoop, C. L. Petrovitch, L. J. Pyrak-Nolte, and L. C. Wilcox, "A discontinuous Galerkin method with a modified penalty flux for the propagation and scattering of acousto-elastic waves," *Geophysical Journal International*, vol. 205, no. 2, pp. 1267–1289, May 2016. arXiv: `1511.00675 [physics.comp-ph]`.

[40] E. Haber, M. Chung, and F. J. Herrmann, "An effective method for parameter estimation with PDE constraints with multiple right hand sides," *SIAM Journal on Optimization*, vol. 22, no. 3, Jul. 2012.

[41] R.-E. Plessix, "A review of the adjoint-state method for computing the gradient of a functional with geophysical applications," *Geophysical Journal International*, vol. 167, no. 2, pp. 495–503, 2006.

[42] F. Luporini, M. Lange, M. Louboutin, N. Kukreja, J. Hückelheim, C. Yount, P. Witte, P. H. J. Kelly, G. J. Gorman, and F. J. Herrmann, "Architecture and performance of devito, a system for automated stencil computation," *CoRR*, vol. abs/1807.03032, Jul. 2018. arXiv: `1807.03032`.

[43] M. Louboutin, M. Lange, F. J. Herrmann, N. Kukreja, and G. Gorman, "Performance prediction of finite-difference solvers for different computer architectures," *Computers & Geosciences*, vol. 105, pp. 148–157, Aug. 2017.

[44] M. Louboutin, P. A. Witte, and F. J. Herrmann, "Effects of wrong adjoints for rtm in tti media," in *SEG Technical Program Expanded Abstracts*, (SEG, Anaheim), Oct. 2018, pp. 331–335.

[45] A. Meurer, C. P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, T. Rathnayake, S. Vig, B. E. Granger, R. P. Muller, F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, M. J. Curry, A. R. Terrel, Š. Roučka, A. Saboo, I. Fernando, S. Kulal, R. Cimrman, and A. Scopatz, "Sympy: Symbolic computing in python," *PeerJ Computer Science*, vol. 3, e103, Jan. 2017.

[46] A. Arbona, B. Miñano, A. Rigo, C. Bona, C. Palenzuela, A. Artigues, C. Bona-Casas, and J. Massó, "Simflowny 2: An upgraded platform for scientific modeling and simulation," *arXiv preprint arXiv:1702.04715*, 2017.

[47] C. T. Jacobs, S. P. Jammy, and N. D. Sandham, "Opensbli: A framework for the automated derivation and parallel execution of finite difference solvers on a range of computer architectures," *CoRR*, vol. abs/1609.01277, 2016.

[48] C. Lengauer, S. Apel, M. Bolten, A. Größlinger, F. Hannig, H. Köstler, U. Rüde, J. Teich, A. Grebhahn, S. Kronawitter, S. Kuckuk, H. Rittich, and C. Schmitt, "Exastencils: Advanced stencil-code engineering," in *Euro-Par 2014: Parallel Processing Workshops - Euro-Par 2014 International Workshops, Porto, Portugal, August 25-26, 2014, Revised Selected Papers, Part II*, 2014, pp. 553–564.

[49] K. A. Hawick and D. P. Playne, "Simulation software generation using a domain-specific language for partial differential field equations," in *11th International Conference on Software Engineering Research and Practice (SERP'13)*, Las Vegas, USA: WorldComp, 22-25 July 2013, SER3829.

[50] L. Thomsen, "Weak elastic anisotropy," *Geophysics*, vol. 51, no. 10, pp. 1964–1966, 1986.

[51] T. Alkhalifah, "An acoustic wave equation for anisotropic media," *Geophysics*, vol. 65, no. 4, pp. 1239–1250, 2000.

[52] K. P. Bube, T. Nemeth, J. P. Stefani, R. Ergas, W. Liu, K. T. Nihei, and L. Zhang, "On the instability in second-order systems for acoustic vti and tti media," *GEOPHYSICS*, vol. 77, no. 5, T171–T186, 2012. eprint: `https://doi.org/10.1190/geo2011-0250.1`.

[53] K. P. Bube*, R. Ergas, and T. Nemeth, "Stability and energy conservation for second-order acoustic systems for vti andtti media with positive shear wavespeeds," in *SEG Technical Program Expanded Abstracts 2014*. 2014, pp. 3439–3443. eprint: https://library.seg.org/doi/pdf/10.1190/segam2014-0986.1.

[54] K. Bube, J. Washbourne, R. Ergas, and T. Nemeth, "Self-adjoint, energy-conserving second-order pseudoacoustic systems for vti and tti media for reverse time migration and full-waveform inversion," in *SEG Technical Program Expanded Abstracts 2016*. 2016, pp. 1110–1114. eprint: https://library.seg.org/doi/pdf/10.1190/segam2016-13878451.1.

[55] C. Chu, B. K. Macy, and P. D. Anno, "Approximation of pure acoustic seismic wave propagation in tti media," *GEOPHYSICS*, vol. 76, no. 5, WB97–WB107, 2011. eprint: https://doi.org/10.1190/geo2011-0092.1.

[56] E. Duveneck and P. M. Bakker, "Stable p-wave modeling for reverse-time migration in tilted ti media," *GEOPHYSICS*, vol. 76, no. 2, S65–S75, 2011. eprint: https://doi.org/10.1190/1.3533964.

[57] R. P. Fletcher, X. Du, and P. J. Fowler, "Reverse time migration in tilted transversely isotropic (tti) media," *GEOPHYSICS*, vol. 74, no. 6, WCA179–WCA187, 2009. eprint: https://doi.org/10.1190/1.3269902.

[58] P. J. Fowler, X. Du, and R. P. Fletcher, "Coupled equations for reverse time migration in transversely isotropic media," *GEOPHYSICS*, vol. 75, no. 1, S11–S22, 2010. eprint: https://doi.org/10.1190/1.3294572.

[59] N. D. Whitmore, "Iterative depth migration by backward time propagation," *1983 SEG Annual Meeting, Expanded Abstracts*, 1983.

[60] P. A. Witte, C. C. Stolk, and F. J. Herrmann, "Phase velocity error minimizing scheme for the anisotropic pure p-wave equation," in *SEG Technical Program Expanded Abstracts*, (SEG, Dallas), Oct. 2016, pp. 452–457.

[61] S. Xu and H. Zhou, "Accurate simulations of pure quasi-P-waves in complex anisotropic media," *Geophysics*, vol. 79, no. 6, pp. 341–348, 2014.

[62] L. Zhang, J. W. Rector III, and H. G. Micheal, "Finite-difference modelling of wave propagation in acoustic tilted TI media," *Geophysical Prospecting*, vol. 53, pp. 843–852, 2005.

[63] Y. Zhang, H. Zhang, and G. Zhang, "A stable TTI reverse time migration and its implementation," *Geophysics*, vol. 76, no. 3, WA3–WA11, 2011.

[64] G. Zhan, R. C. Pestana, and P. L. Stoffa, "An efficient hybrid pdeudospectral/finite-difference scheme for solving the TTI pure P-wave equation," *Journal of Geophyics and Engineering*, vol. 10, 2013.

[65] R. Hooke, D. Papin, S. Sturmy, and J. Young, *Lectures de potentia restitutiva*. London : Printed for John Martyn ..., 1678.

[66] C. B. Officer and P. M. Morse, "Introduction to the theory of sound transmission," *Physics Today*, vol. 12, no. 12, pp. 66–67, 1959. eprint: `https://doi.org/10.1063/1.3060620`.

[67] R. E. Sheriff, "Encyclopedic dictionary of exploration geophysics," 1991.

[68] R. E. Sheriff and L. P. Geldart, *Exploration seismology*. Cambridge University Press, 1995.

[69] A.-L. Cauchy, "Méthode générale pour la résolution des systèmes d'équations simultanées," *Compte Rendu des Séances de L'Académie des Sciences XXV*, vol. Série A, no. 25, pp. 536–538, Oct. 1847.

[70] R. Fletcher and C. M. Reeves, "Function minimization by conjugate gradients," *The Computer Journal*, vol. 7, no. 2, pp. 149–154, Jan. 1964. eprint: `http://oup.prod.sis.lan/comjnl/article-pdf/7/2/149/959725/070149.pdf`.

[71] M. R. Hestenes and E. Stiefel, "Methods of conjugate gradients for solving linear systems," *Journal of research of the National Bureau of Standards*, vol. 49, pp. 409–436, 1952.

[72] H. Shah, *The 2007 bp anisotropic velocity-analysis benchmark (presented at the 70th eage annual meeting, workshop)*, 2007.

[73] P. Colella, *Defining software requirements for scientific computing*, 2004.

[74] S. Williams, A. Waterman, and D. Patterson, "The roofline model offers insight on how to improve the performance of software and hardware.," *communications of the acm*, vol. 52, no. 4, 2009.

[75] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, *et al.*, "The landscape of parallel computing research: A view from berkeley," Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Tech. Rep., 2006.

[76] M. Louboutin, P. A. Witte, M. Lange, N. Kukreja, F. Luporini, G. Gorman, and F. J. Herrmann, “Full-waveform inversion - part 1: Forward modeling,” *The Leading Edge*, vol. 36, no. 12, pp. 1033–1036, Dec. 2017, (The Leading Edge).

[77] ——, “Full-waveform inversion - part 2: Adjoint modeling,” *The Leading Edge*, vol. 37, no. 1, pp. 69–72, Jan. 2018, (The Leading Edge).

[78] P. A. Witte, M. Louboutin, K. Lensink, M. Lange, N. Kukreja, F. Luporini, G. Gorman, and F. J. Herrmann, “Full-waveform inversion - part 3: Optimization,” *The Leading Edge*, vol. 37, no. 2, pp. 142–145, Jan. 2018, (The Leading Edge).

[79] S. Williams and D. Patterson, *Roofline performance model*, `http://crd.lbl.gov/assets/pubspresos/parlab08-roofline-talk.pdf`, 2008.

[80] I. Epicoco, S. Mocavero, F. Macchia, and G. Aloisio, “The roofline model for oceanic climate applications,” in *High Performance Computing & Simulation (HPCS), 2014 International Conference on*, IEEE, 2014, pp. 732–737.

[81] C. Chan, D. Unat, M. Lijewski, W. Zhang, J. Bell, and J. Shalf, “Software design space exploration for exascale combustion co-design,” in *International Supercomputing Conference*, Springer, 2013, pp. 196–212.

[82] C. Andreolli, P. Thierry, L. Borges, C. Yount, and G. Skinner, “Genetic algorithm based auto-tuning of seismic applications on multi and manycore computers,” in *EAGE Workshop on High Performance Computing for Upstream*, 2014.

[83] K. Datta and K. A. Yelick, “Auto-tuning stencil codes for cache-based multicore platforms,” PhD thesis, University of California, Berkeley, 2009.

[84] Y. Sato, R. Nagaoka, A. Musa, R. Egawa, H. Takizawa, K. Okabe, and H. Kobayashi, “Performance tuning and analysis of future vector processors based on the roofline model,” in *Proceedings of the 10th workshop on MEmory performance: DEaling with Applications, systems and architecture*, ACM, 2009, pp. 7–14.

[85] K.-H. Kim, K. Kim, and Q.-H. Park, “Performance analysis and optimization of three-dimensional FDTD on GPU using roofline model,” *Computer Physics Communications*, vol. 182, no. 6, pp. 1201–1207, 2011.

[86] Y. J. Lo, S. Williams, B. Van Straalen, T. J. Ligocki, M. J. Cordery, N. J. Wright, M. W. Hall, and L. Oliker, “Roofline model toolkit: A practical tool for architectural and program analysis,” in *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, Springer, 2014, pp. 129–148.

[87] A. Ilic, F. Pratas, and L. Sousa, "Cache-aware roofline model: Upgrading the loft," *IEEE Computer Architecture Letters*, vol. 13, no. 1, pp. 21–24, 2014.

[88] L. A. Barba and R. Yokota, "How will the fast multipole method fare in the exascale era," *SIAM News*, vol. 46, no. 6, pp. 1–3, 2013.

[89] J. Lai and A. Seznec, "Performance upper bound analysis and optimization of SGEMM on Fermi and Kepler GPUs," in *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*, IEEE, 2013, pp. 1–10.

[90] M. Wahib and N. Maruyama, "Scalable kernel fusion for memory-bound GPU applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE Press, 2014, pp. 191–202.

[91] J. Hofmann, J. Eitzinger, and D. Fey, "Execution-cache-memory performance model: Introduction and validation," *arXiv preprint arXiv:1509.03118*, 2015.

[92] D. Duplyakin, J. Brown, and R. Ricci, "Active Learning in Performance Analysis," in *Cluster Computing (CLUSTER), 2016 IEEE International Conference on*, IEEE, 2016, pp. 182–191.

[93] H. Stengel, J. Treibig, G. Hager, and G. Wellein, "Quantifying performance bottlenecks of stencil computations using the execution-cache-memory model," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ACM, 2015, pp. 207–216.

[94] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Optimization and performance modeling of stencil computations on modern microprocessors," *SIAM review*, vol. 51, no. 1, pp. 129–159, 2009.

[95] J. Hammer, G. Hager, J. Eitzinger, and G. Wellein, "Automatic loop kernel analysis and performance modeling with KernCraft," in *Proceedings of the 6th International Workshop on Performance Modeling, Benchmarking, and Simulation of High Performance Computing Systems*, ACM, 2015, p. 4.

[96] S. H. K. Narayanan, B. Norris, and P. D. Hovland, "Generating performance bounds from source code," in *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, IEEE, 2010, pp. 197–206.

[97] D. Unat, C. Chan, W. Zhang, S. Williams, J. Bachan, J. Bell, and J. Shalf, "ExaSAT: An exascale co-design tool for performance modeling," *The International Journal of High Performance Computing Applications*, vol. 29, no. 2, pp. 209–232, 2015.

[98] S. M. F. Rahman, Q. Yi, and A. Qasem, “Understanding stencil code performance on multicore architectures,” in *Proceedings of the 8th ACM International Conference on Computing Frontiers*, ACM, 2011, p. 30.

[99] J. Hofmann, D. Fey, M. Riedmann, J. Eitzinger, G. Hager, and G. Wellein, “Performance analysis of the Kahan-enhanced scalar product on current multicore processors,” in *International Conference on Parallel Processing and Applied Mathematics*, Springer, 2015, pp. 63–73.

[100] C. Andreolli, P. Thierry, L. Borges, G. Skinner, and C. Yount, “Chapter 23 - characterization and optimization methodology applied to stencil computations,” in *High Performance Parallelism Pearls*, J. Reinders and J. Jeffers, Eds., Boston: Morgan Kaufmann, 2015, pp. 377–396, ISBN: 978-0-12-802118-7.

[101] J. D. McCalpin, “Memory bandwidth and machine balance in current high performance computers,” *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.

[102] ——, “Stream: Sustainable memory bandwidth in high performance computers,” University of Virginia, Charlottesville, Virginia, Tech. Rep., 1991-2007, A continually updated technical report. http://www.cs.virginia.edu/stream/.

[103] J. Dongarra, “The linpack benchmark: An explanation,” in *Proceedings of the 1st International Conference on Supercomputing*, London, UK, UK: Springer-Verlag, 1988, pp. 456–474, ISBN: 3-540-18991-2.

[104] W. Liu, K. Bube, L. Zhang, and K. Nihei, “Stable reverse-time migration in variable-tilt TI media,” in *71st EAGE Conference and Exhibition incorporating SPE EUROPEC 2009*, 2009.

[105] R. M. Weiss and J. Shragge, “Solving 3D anisotropic elastic wave equations on parallel GPU devices.,” *Geophysics*, vol. 78, no. 2, 2013.

[106] E. Konstantinidis and Y. Cotronis, “A practical performance model for compute and memory bound GPU kernels,” in *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, Mar. 2015, pp. 651–658.

[107] W. Wu, L. R. Lines, and H. Lu, “Analysis of higher-order, finite-difference schemes in 3-D reverse-time migration,” *GEOPHYSICS*, vol. 61, no. 3, pp. 845–856, 1996. eprint: `http://dx.doi.org/10.1190/1.1444009`.

[108] L. Lines, R. Slawinski, and R. Bording, “A recipe for stability of finite-difference wave-equation computations,” *GEOPHYSICS*, vol. 64, no. 3, pp. 967–969, 1999, `http://library.seg.org/doi/pdf/10.1190/1.1444605`.

[109] P. E. Farrell, D. A. Ham, S. W. Funke, and M. E. Rognes, "Automated derivation of the adjoint of high-level transient finite element programs," *SIAM Journal on Scientific Computing*, vol. 35, no. 4, pp. C369–C393, 2013. eprint: `http://dx.doi.org/10.1137/120873558`.

[110] J. Virieux, "P-sv wave propagation in heterogeneous media: Velocity-stress finite-difference method," *GEOPHYSICS*, vol. 51, no. 4, pp. 889–901, 1986. eprint: `https://doi.org/10.1190/1.1442147`.

[111] W. W. Symes, "Acoustic staggered grid modeling in iwave," *THE RICE INVERSION PROJECT*, p. 141, 2015.

[112] T. Henretty, R. Veras, F. Franchetti, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan, "A stencil compiler for short-vector simd architectures," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS '13, Eugene, Oregon, USA: ACM, 2013, pp. 13–24, ISBN: 978-1-4503-2130-3.

[113] C. Yount, "Vector folding: Improving stencil performance via multi-dimensional simd-vector representation," in *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, Aug. 2015, pp. 865–870.

[114] G. M. Hopper, "The education of a computer," in *Proceedings of the 1952 ACM national meeting (Pittsburgh)*, ACM, 1952, pp. 243–249.

[115] J. L. Jones, "A survey of automatic coding techniques for digital computers," PhD thesis, Massachusetts Institute of Technology, 1954.

[116] J. Backus, "The history of fortran i, ii, and iii," in *History of programming languages I*, ACM, 1978, pp. 25–74.

[117] K. E. Iverson, *A Programming Language*. New York, NY, USA: John Wiley & Sons, Inc., 1962, ISBN: 0-471430-14-5.

[118] A. F. Cárdenas and W. J. Karplus, "Pdel—a language for partial differential equations," *Communications of the ACM*, vol. 13, no. 3, pp. 184–191, 1970.

[119] Y. Umetani, "Deqsol a numerical simulation language for vector/parallel processors," *Proc. IFIP TC2/WG22, 1985*, vol. 5, pp. 147–164, 1985.

[120] G. O. Cook Jr, "Alpal: A tool for the development of large-scale simulation codes," Lawrence Livermore National Lab., CA (USA), Tech. Rep., 1988.

[121] R. Van Engelen, L. Wolters, and G. Cats, "Ctadel: A generator of multi-platform high performance codes for pde-based scientific applications," in *Proceedings of the 10th international conference on Supercomputing*, ACM, 1996, pp. 86–93.

[122] M. S. Alnæs, A. Logg, K. B. Ølgaard, M. E. Rognes, and G. N. Wells, "Unified Form Language: A domain-specific language for weak formulations of partial differential equations," *ACM Transactions on Mathematical Software (TOMS)*, vol. 40, no. 2, p. 9, 2014.

[123] Y. Zhang and F. Mueller, "Auto-generation and auto-tuning of 3d stencil codes on gpu clusters," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ser. CGO '12, San Jose, California: ACM, 2012, pp. 155–164, ISBN: 978-1-4503-1206-6.

[124] M. Christen, O. Schenk, and H. Burkhart, "Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures," in *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS '11, Washington, DC, USA: IEEE Computer Society, 2011, pp. 676–687, ISBN: 978-0-7695-4385-7.

[125] D. Unat, X. Cai, and S. B. Baden, "Mint: Realizing cuda performance in 3d stencil methods with annotated c," in *Proceedings of the international conference on Supercomputing*, ACM, 2011, pp. 214–224.

[126] M. Köster, R. Leißa, S. Hack, R. Membarth, and P. Slusallek, "Platform-specific optimization and mapping of stencil codes through refinement," in *Proceedings of the 1st International Workshop on High-Performance Stencil Computations*, 2014, pp. 1–6.

[127] R. Membarth, F. Hannig, J. Teich, and H. Köstler, "Towards domain-specific computing for stencil codes in hpc," in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, IEEE, 2012, pp. 1133–1138.

[128] C. Osuna, O. Fuhrer, T. Gysi, and M. Bianco, "Stella: A domain-specific language for stencil methods on structured grids," in *Poster Presentation at the Platform for Advanced Scientific Computing (PASC) Conference, Zurich, Switzerland*, 2014.

[129] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson, "The pochoir stencil compiler," in *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, ACM, 2011, pp. 117–128.

[130] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementa-*

tion, ser. PLDI '08, Tucson, AZ, USA: ACM, 2008, pp. 101–113, ISBN: 978-1-59593-860-2.

[131] C. T. Jacobs, S. P. Jammy, and N. D. Sandham, "Opensbli: A framework for the automated derivation and parallel execution of finite difference solvers on a range of computer architectures," *CoRR*, vol. abs/1609.01277, 2016.

[132] W. W. Symes, D. Sun, and M. Enriquez, "From modelling to inversion: Designing a well-adapted simulator," *Geophysical Prospecting*, vol. 59, no. 5, pp. 814–833, 2011.

[133] W. W. Symes, "Iwave structure and basic use cases," *THE RICE INVERSION PROJECT*, p. 85, 2015.

[134] D. Sun and W. W. Symes, "Iwave implementation of adjoint state method," Tech. Rep. 10-06, Department of Computational and Applied Mathematics, Rice University, Houston, Texas, USA, Tech. Rep., 2010.

[135] F. Luporini, A. L. Varbanescu, F. Rathgeber, G.-T. Bercea, J. Ramanujam, D. A. Ham, and P. H. J. Kelly, "Cross-loop optimization of arithmetic intensity for finite element local assembly," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 4, 57:1–57:25, Jan. 2015.

[136] H. L. Seongjai Kim, "High-order schemes for acoustic waveform simulation," *Applied Numerical Mathematics*, vol. 57, pp. 402–414, 2007.

[137] R. Clayton and B. Engquist, "Absorbing boundary conditions for acoustic and elastic wave equations," *Bulletin of the Seismological Society of America*, vol. 67, no. 6, pp. 1529–1540, 1977. eprint: `http://bssa.geoscienceworld.org/content/67/6/1529.full.pdf`.

[138] Y. Liu and S. Fomel, "Seismic data interpolation beyond aliasing using regularized nonstationary autoregression," *GEOPHYSICS*, vol. 76, no. 5, pp. V69–V77, 2011. eprint: `https://doi.org/10.1190/geo2010-0231.1`.

[139] H. Wason, F. Oghenekohwo, and F. J. Herrmann, "Low-cost time-lapse seismic with distributed compressive sensing–-Part 2: Impact on repeatability," *Geophysics*, vol. 82, no. 3, P15–P30, May 2017, (Geophysics).

[140] M. Naghizadeh and M. D. Sacchi, "F-x adaptive seismic-trace interpolation," *GEOPHYSICS*, vol. 74, no. 1, pp. V9–V16, 2009. eprint: `https://doi.org/10.1190/1.3008547`.

[141] R. Kumar, H. Wason, and F. J. Herrmann, “Source separation for simultaneous towed-streamer marine acquisition –- a compressed sensing approach,” *Geophysics*, vol. 80, no. 06, WD73–WD88, Nov. 2015, (Geophysics).

[142] M. Lange, N. Kukreja, F. Luporini, M. Louboutin, C. Yount, J. Hückelheim, and G. J. Gorman, “Optimised finite difference computation from symbolic equations,” in *Proceedings of the 15th Python in Science Conference*, K. Huff, D. Lippa, D. Niederhut, and M. Pacer, Eds., 2017, pp. 89–96.

[143] M. Louboutin, P. Witte, M. Lange, N. Kukreja, F. Luporini, G. Gorman, and F. J. Herrmann, “Full-waveform inversion, part 1: Forward modeling,” *The Leading Edge*, vol. 36, no. 12, pp. 1033–1036, 2017. eprint: `https://doi.org/10.1190/tle36121033.1`.

[144] G. A. McMechan, “Migration by extrapolation of time-dependent boundary values,” *Geophysical Prospecting*, vol. 31, no. 3, pp. 413–420, 1983.

[145] R. Mittet, “Implementation of the kirchhoff integral for elastic waves in staggered-grid modeling schemes,” *GEOPHYSICS*, vol. 59, no. 12, pp. 1894–1901, 1994. eprint: `http://dx.doi.org/10.1190/1.1443576`.

[146] E. B. Raknes and W. Weibull, “Efficient 3d elastic full-waveform inversion using wavefield reconstruction methods,” *Geophysics*, vol. 81, no. 2, R45–R55, 2016. eprint: `http://geophysics.geoscienceworld.org/content/81/2/R45.full.pdf`.

[147] M. Warner and L. Guasch, “Adaptive waveform inversion: Theory,” in *SEG Technical Program Expanded Abstracts 2014*. 2014, pp. 1089–1093. eprint: `https://library.seg.org/doi/pdf/10.1190/segam2014-0371.1`.

[148] B. Peters and F. J. Herrmann, “Constraints versus penalties for edge-preserving full-waveform inversion,” *The Leading Edge*, vol. 36, no. 1, pp. 94–100, Jan. 2017, (The Leading Edge).

[149] H. Igel, *Computational Seismology: A Practical Introduction*, 1., O. U. Press, Ed. Oxford University Press, Nov. 2016, ISBN: 9780198717409.

[150] K. Watanabe, “Green’s functions for laplace and wave equations,” in *Integral Transform Techniques for Green’s Function*. Cham: Springer International Publishing, 2015, pp. 33–76, ISBN: 978-3-319-17455-6.

[151] S. Wang, A. Nissen, and G. Kreiss, “Convergence of finite difference methods for the wave equation in two space dimensions,” *Computing Research Repository*, no. 1702.01383, 2017.

[152] R. Versteeg, "The marmousi experience; velocity model determination on a synthetic complex data set," *The Leading Edge*, vol. 13, no. 9, pp. 927–936, 1994. eprint: `http://tle.geoscienceworld.org/content/13/9/927.full.pdf`.

[153] P. A. Witte, M. Louboutin, N. Kukreja, F. Luporini, M. Lange, G. J. Gorman, and F. J. Herrmann, "A large-scale framework for symbolic implementations of seismic inversion algorithms in julia," *Geophysics*, vol. 84, no. 3, F57–F71, Mar. 2019, (Geophysics).

[154] M. Schmidt, E. van den Berg, M. P. Friedlander, and K. Murphy, "Optimizing costly functions with simple constraints: A limited-memory projected quasi-newton algorithm," in *Proceedings of The Twelfth International Conference on Artificial Intelligence and Statistics (AISTATS) 2009*, D. van Dyk and M. Welling, Eds., vol. 5, Clearwater Beach, Florida, Apr. 2009, pp. 456–463.

[155] L. A. Barba and G. F. Forsyth, *Cfd python: The 12 steps to navier-stokes equations.* 2018.

[156] J. Dongarra, "The LINPACK benchmark: An explanation," in *Proceedings of the 1st International Conference on Supercomputing*, London, UK, UK: Springer-Verlag, 1988, pp. 456–474, ISBN: 3-540-18991-2.

[157] Intel Corporation, *Intel VTune Performance Analyzer*, https://software.intel.com/en-us/intel-vtune-amplifier-xe, 2016.

[158] S. J. Pennycook, J. Sewall, and V. Lee, "A metric for performance portability," *arXiv preprint arXiv:1611.07409*, 2016.

[159] C. Yount, J. Tobin, A. Breuer, and A. Duran, "Yask–yet another stencil kernel: A framework for hpc stencil code-generation and tuning," in *Proceedings of the 6th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing held as part of ACM/IEEE Supercomputing 2016 (SC16)*, ser. WOLFHPC'16, Salt Lake City, Utah, Nov. 2016.

[160] S. C. Brenner and L. R. Scott, *The Mathematical Theory of Finite Element Methods*. New York, NY: Springer New York, 2008, vol. 15, ISBN: 978-0-387-75933-3.

[161] T. O. Foundation, *Openfoam v5 user guide*.

[162] F. Witherden, A. Farrington, and P. Vincent, "Pyfr: An open source framework for solving advection–diffusion type problems on streaming architectures using the flux reconstruction approach," *Computer Physics Communications*, vol. 185, no. 11, pp. 3028–3040, 2014.

[163] I. Z. Reguly, G. R. Mudalige, M. B. Giles, D. Curran, and S. McIntosh-Smith, "The ops domain specific abstraction for multi-block structured grid computations," in *Proceedings of the Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, ser. WOLFHPC '14, New Orleans, Louisiana: IEEE Press, 2014, pp. 58–67, ISBN: 978-1-4799-7020-9.

[164] Y. Zhang and F. Mueller, "Auto-generation and auto-tuning of 3d stencil codes on gpu clusters," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ser. CGO '12, San Jose, California: ACM, 2012, pp. 155–164, ISBN: 978-1-4503-1206-6.

[165] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13, Seattle, Washington, USA: ACM, 2013, pp. 519–530, ISBN: 978-1-4503-2014-6.

[166] A. Klöckner, "Loo.py: transformation-based code generation for GPUs and CPUs," in *Proceedings of ARRAY '14: ACM SIGPLAN Workshop on Libraries, Languages, and Compilers for Array Programming*, Edinburgh, Scotland.: Association for Computing Machinery, 2014.

[167] Y. Ding and X. Shen, "Glore: Generalized loop redundancy elimination upon lernotation," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, 74:1–74:28, Oct. 2017.

[168] K. Datta, S. Williams, V. Volkov, J. Carter, L. Oliker, J. Shalf, and K. Yelick, "Autotuning the 27-point stencil for multicore," in *In In Proc. iWAPT2009: The Fourth International Workshop on Automatic Performance Tuning*, 2009.

[169] S. J. Deitz, B. L. Chamberlain, and L. Snyder, "Eliminating redundancies in sum-of-product array computations," in *Proceedings of the 15th International Conference on Supercomputing*, ser. ICS '01, Sorrento, Italy: ACM, 2001, pp. 65–77, ISBN: 1-58113-410-X.

[170] S. Kronawitter, S. Kuckuk, and C. Lengauer, "Redundancy elimination in the exastencils code generator," in *ICA3PP Workshops*, 2016.

[171] J. Jeffers and J. Reinders, *High Performance Parallelism Pearls Volume Two: Multicore and Many-core Programming Approaches*, 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2015, ISBN: 0128038195, 9780128038192.

[172] A. Meurer, C. P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, T. Rathnayake, S. Vig, B. E. Granger, R. P. Muller, F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, M. J. Curry, A. R.

Terrel, Š. Roučka, A. Saboo, I. Fernando, S. Kulal, R. Cimrman, and A. Scopatz, "Sympy: Symbolic computing in python," *PeerJ Computer Science*, vol. 3, e103, Jan. 2017.

[173] R. L. Higdon, "Numerical absorbing boundary conditions for the wave equation," *Mathematics of Computation*, vol. 49, no. 179, pp. 65–90, 1987.

[174] Mathias Louboutin, Fabio Luporini, *Boundary conditions in Devito*, in preparation (2018).

[175] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, Eds., *Compilers: principles, techniques, and tools*, Second. Boston, MA, USA: Pearson/Addison Wesley, 2007, pp. xxiv + 1009, ISBN: 0-321-48681-1.

[176] A. Klöckner, *Cgen - c/c++ source generation from an ast*, `https://github.com/inducer/cgen`, 2016.

[177] G. R. Markall, F. Rathgeber, L. Mitchell, N. Loriant, C. Bertolli, D. A. Ham, and P. H. J. Kelly, "Performance portable finite element assembly using PyOP2 and FEniCS," in *Proceedings of the International Supercomputing Conference (ISC) '13*, ser. Lecture Notes in Computer Science, In press, vol. 7905, Jun. 2013.

[178] C. Yount, "Vector folding: Improving stencil performance via multi-dimensional simd-vector representation," in *Proceedings of the IEEE 17th International Conference on High Performance Computing and Communications (HPCC)*, Aug. 2015, pp. 865–870.

[179] C. Yount and A. Duran, "Effective use of large high-bandwidth memory caches in HPC stencil computation via temporal wave-front tiling," in *Proceedings of the 7th International Workshop in Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems held as part of ACM/IEEE Supercomputing 2016 (SC16)*, ser. PMBS'16, Salt Lake City, Utah, Nov. 2016.

[180] J. Tobin, A. Breuer, A. Heinecke, C. Yount, and Y. Cui, "Accelerating seismic simulations using the intel xeon phi knights landing processor," in *Proceedings of ISC High Performance 2017 (ISC17)*, Frankfurt, Germany, to appear 2017.

[181] C. Yount, A. Duran, and J. Tobin, "Multi-level spatial and temporal tiling for efficient hpc stencil computation on many-core processors with large shared caches," *Future Generation Computer Systems*, 2017.

[182] Zenodo/devito-performance, *Devito Experimentation Framework*, Jul. 2018.

[183] Y. Zhang, H. Zhang, and G. Zhang, “A stable tti reverse time migration and its implementation,” *GEOPHYSICS*, vol. 76, no. 3, WA3–WA11, 2011. eprint: `https://doi.org/10.1190/1.3554411`.

[184] A. Siahkoohi, M. Louboutin, and F. J. Herrmann, “The importance of transfer learning in seismic modeling and imaging,” *Geophysics*, 2019, (Accepted in GEOPHYSICS).

[185] F. J. Herrmann, C. Jones, G. Gorman, J. Hückelheim, K. Lensink, P. Kelly, N. Kukreja, H. Modzelewski, M. Lange, M. Louboutin, F. Luporini, J. Selvages, and P. A. Witte, “Accelerating ideation & innovation cheaply in the cloud the power of abstraction, collaboration & reproducibility,” in *4th EAGE Workshop on High-performance Computing*, (EAGE HPC Workshop, Dubai), Oct. 2019.

[186] V. Pandolfo, “Investigating the OPS intermediate representation to target gpus in the devito DSL,” *CoRR*, vol. abs/1906.10811, 2019. arXiv: `1906.10811`.

[187] F. Luporini and G. Gorman, “Automatic code generation for gpus using devito,” 2019, Submitted to Rice Oil and Gas High Performance Computing Conference 2020 on November 27, 2019.

[188] J. W. Thorbecke and D. Draganov, “Finite-difference modeling experiments for seismic interferometry,” *GEOPHYSICS*, vol. 76, no. 6, H1–H18, 2011. eprint: `https://doi.org/10.1190/geo2010-0039.1`.

[189] P. A. Witte, M. Louboutin, H. Modzelewski, C. Jones, J. Selvage, and F. J. Herrmann, “An event-driven approach to serverless seismic imaging in the cloud,” Submitted to IEEE Transactions on Parallel and Distributed Systems on August 20, 2019, 2019.

[190] ——, “Event-driven workflows for large-scale seismic imaging in the cloud,” in *SEG Technical Program Expanded Abstracts*, (SEG, San Antonio), Sep. 2019, pp. 3984–3988.

[191] K. Lensink, *SegyIO.jl*, `https://github.com/slimgroup/SegyIO.jl`, 2017.

[192] K. M. Barry, D. A. Cavers, and C. W. Kneale, “Recommended standards for digital tape formats,” *Geophysics*, vol. 40, no. 2, pp. 344–352, Apr. 1975. eprint: `https://pubs.geoscienceworld.org/geophysics/article-pdf/40/2/344/3156872/344.pdf`.

[193] P. A. Witte, M. Louboutin, N. Kukreja, F. Luporini, M. Lange, G. J. Gorman, and F. J. Herrmann, “Geophysics bright spots: Efficient coding of large-scale seismic

inversion algorithms," *The Leading Edge*, vol. 38, no. 6, pp. 482–484, 2019, (The Leading Edge).

[194] G. S. Martin, R. Wiley, and K. J. Marfurt, "Marmousi2: An elastic upgrade for marmousi," *The Leading Edge*, vol. 25, no. 2, pp. 156–166, 2006. eprint: `https://doi.org/10.1190/1.2172306`.

[195] H. Senger, J. F. de Souza, E. S. Gomi, F. Luporini, and G. J. Gorman, "Performance of Devito on HPC-Optimised ARM Processors," *arXiv e-prints*, arXiv:1908.03653, arXiv:1908.03653, Aug. 2019. arXiv: `1908.03653 [cs.PF]`.

[196] F. J. Herrmann, A. Siahkoohi, and G. Rizzuti, "Learned imaging with constraints and uncertainty quantification," in *Neural Information Processing Systems (NeurIPS)*, Accepted on October 1, 2019, Dec. 2019.