Imperial College London
Department of Computing

Automated cache optimisations of stencil computations for partial differential equations

Georgios Bismpas

March 2023

Supervised by: Prof. Paul H. J. Kelly
In collaboration with: Dr. Fabio Luporini

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing
of Imperial College London
and the Diploma of Imperial College London

Declaration

I herewith certify that the material in this thesis that is not my own work has been properly acknowledged and referenced.

Georgios Bismpas

Abstract

This thesis focuses on numerical methods that solve partial differential equations. Our focal point is the finite difference method, which solves partial differential equations by approximating derivatives with explicit finite differences. These partial differential equation solvers consist of stencil computations on structured grids. Stencils for computing real-world practical applications are patterns often characterised by many memory accesses and non-trivial arithmetic expressions that lead to high computational costs compared to simple stencils used in much prior proof-of-concept work. In addition, the loop nests to express stencils on structured grids may often be complicated. This work is highly motivated by a specific domain of stencil computations where one of the challenges is non-aligned to the structured grid ("off-the-grid") operations. These operations update neighbouring grid points through scatter and gather operations via non-affine memory accesses, such as A[B[i]]. In addition to this challenge, these practical stencils often include many computation fields (need to store multiple grid copies), complex data dependencies and imperfect loop nests.

In this work, we aim to increase the performance of stencil kernel execution. We study automated cache-memory-dependent optimisations for stencil computations. This work consists of two core parts with their respective contributions.

The first part of our work tries to reduce the data movement in stencil computations of practical interest. Data movement is a dominant factor affecting the performance of high-performance computing applications. It has long been a target of optimisations due to its impact on execution time and energy consumption. This thesis tries to relieve this cost by

applying temporal blocking optimisations, also known as time-tiling, to stencil computations. Temporal blocking is a well-known technique to enhance data reuse in stencil computations. However, it is rarely used in practical applications but rather in theoretical examples to prove its efficacy. Applying temporal blocking to scientific simulations is more complex. More specifically, in this work, we focus on the application context of seismic and medical imaging. In this area, we often encounter scatter and gather operations due to signal sources and receivers at arbitrary locations in the computational domain. These operations make the application of temporal blocking challenging. We present an approach to overcome this challenge and successfully apply temporal blocking.

In the second part of our work, we extend the first part as an automated approach targeting a wide range of simulations modelled with partial differential equations. Since temporal blocking is error-prone, tedious to apply by hand and highly complex to assimilate theoretically and practically, we are motivated to automate its application and automatically generate code that benefits from it. We discuss algorithmic approaches and present a generalised compiler pipeline to automate the application of temporal blocking. These passes are written in the Devito compiler. They are used to accelerate the computation of stencil kernels in areas such as seismic and medical imaging, computational fluid dynamics and machine learning. Devito is a Python package to implement optimised stencil computation (e.g., finite differences, image processing, machine learning) from high-level symbolic problem definitions. Devito builds on SymPy and employs automated code generation and just-in-time compilation to execute optimised computational kernels on several computer platforms, including CPUs, GPUs, and clusters thereof.

We show how we automate temporal blocking code generation without user intervention and often achieve better time-to-solution. We enable domain-specific optimisation through compiler passes and offer temporal blocking gains from a high-level symbolic abstraction. These automated optimisations benefit various computational kernels for solving real-world application problems.

To the people of Livadi, Larissa, Greece

Acknowledgements

In this section, I would like to express my gratitude to those who supported me while pursuing my PhD studies at Imperial College London.

I could not have undertaken this journey without my primary supervisor, Paul H.J. Kelly, whose advice, knowledge, patience and support have been everpresent during the last years. Paul not only guided me with safety along this academic journey but helped me balance learning, balance, emotions, and attitude. Thanks for contributing to my academic and personal growth. I am privileged to have been co-supervised by Fabio Luporini. Fabio was there in all my problems and improved my skills in all aspects of programming. Fabio shaped all of my technical skills and knowledge throughout this PhD degree. Gerard Gorman, I cannot thank you enough for giving me the opportunity to work on a top interdisciplinary project and for making me appreciate the impact of science on real-world problems. I feel fortunate to have worked with you in a number of scientific areas. Paul, Fabio and Gerard offered me the chance to pursue and finish my PhD and accomplish one of my life goals. I feel deeply indebted.

To my family, whose support was never-ending through this journey. Without them, I would have never accomplished this PhD thesis.

To Navjot Kukreja, a precious friend and ex-fellow-PhD student who was my first guide and supported me when I started my studies. Your experience helped me a lot along the way. To Mathias Louboutin, Rhodri Nelson and Edward Caunt, great co-workers with whom I shared this trip in the Devito project and who helped me a lot during my PhD challenges.

To the Devito community that answered my questions and with whom I enjoyed fruitful discussions.

To Nikos P. Pitsianis and Dimitrios Floros, who shaped my skills in high-performance computing before I started my PhD at Imperial. You have been the first to supervise my work and contributed to my personal growth. You are responsible for making me love HPC and pursue a career in this area.

To Jan Huckelheim, Richard Michael Veras and J. (Ram) Ramanujam, from whom I learned a lot and expanded my knowledge. To the people I shared my office with, Riku Murai, Eduardo Carvalho, TJ Sun, Edward Stow, Nicolai Stawinoga and Matthew Taylor. The fruitful discussions and their great company were essential to my time during my PhD.

To great friends, Luca Castiglione, with whom we started our PhDs together, supported me every day and was my university buddy for the last four years. To Anastasios, Ioannis, Athanasios, Christos, Evangelos, Nikolaos, Konstantinos, Athanasios, Kostas, Andreas. Thank you for being patient with me and present whenever I needed you.

To the late John Niamas and his scholarship fund, whose legacy and memory should remain eternal and influence future scientists in my birth-place, Livadi, Larissa, Greece. To the late Pantelis Simetis, who I believe would be proud. He taught me pseudocode at the age of 16 and fired up my passion for engineering and computer science. Rest in peace, beloved teacher.

Finally, I would like to acknowledge the funding grant for my PhD through the EPSRC's Centre for Doctoral Training in High Performance Embedded and Distributed Systems (EP/L016796/1).

Contents

Listings

List of Tables

List of Figures

Chapter 1

Introduction

Increasing the performance of numerical methods for partial differential equations is challenging. Cache-related optimisations are tedious to apply by hand. This task is even more challenging when targeting real-world problems. By raising the abstraction level, we show how to apply cache optimisations for various partial differential equations automatically.

1.1. Overview

Numerous scientific fields, including computational fluid dynamics, seismic and medical imaging, computational electromagnetics, finance, epidemic spread modelling, and others employ partial differential equations to model phenomena. Numerical techniques, such as the finite difference, finite element and finite volume method, are widely employed to approximate the solutions for these phenomena.

Structured grids, also referred to as structured meshes, are employed to discretise the computational domain. The solution is approximated by applying suitable numerical operations, or kernels, to the points or cells of the grid. On clusters of multi-core processor nodes, usually, a kernel is executed in a shared-memory parallel fashion in a node. At the same time, distributed-memory parallelism is employed to decompose the solution domain to different nodes or processors. This is a widely-adopted execution model for many real-world applications and frameworks.

When applying numerical kernels, we aim to compute simulations:

1. within a specific time limit, to have answers at a specific point in time

2. within a specific margin of adequate accuracy to be confident about the reliability of the answers

To get more reliable answers, we could try some obvious solutions, such as increasing the discretisation order for which we solve partial differential equations or increasing the grid resolution. Higher discretisation order leads to longer execution times as mathematical expressions get more complex to compute. The higher the grid resolution, the higher the number of grid points in which the equation domain must be discretised. The equation domain needs to be discretised into a significant number of points to obtain a satisfactory solution approximation. This number is usually of the order of billions, as in Bauer et al. [2015].

Nevertheless, most simulations often have a strict time or resource limit to yield results. Obviously, to increase accuracy, we often compromise by spending more resources, time included.

Weather forecasts often use the above compromise to trade off accuracy for prediction range. Scientists aim for high accuracy for short-range forecasts, less accuracy for medium-range forecasts and even less accuracy for longer-range predictions. More specifically, in the case of the UK Met Office, weather forecasting is updated every 60 minutes [Brown et al., 2012] for a detailed forecast of the next 6 hours. Seismic imaging mandates high accuracy within a reasonable cost. The cost is valued at millions of US dollars for an oil and gas company to request 3D acquisition of 3D seismic data. The cost of processing data is around an order of magnitude less than acquiring it. Naturally, state-of-the-art methods focus more on data reprocessing than new acquisitions. Consequently, inefficient kernels may have a high impact on incurred costs. Medical imaging also requires fast results and high accuracy for evident reasons of accurate medical predictions. It is expensive, and results are needed as soon as possible to protect lives. The work of Guasch et al. [2019] presents an example for brain and chest imaging. Overall, efficient kernels have a direct scientific payoff in higher resolution and, therefore, more accurate forecasts in less time.

This work explores automated performance optimisations of numerical methods based on finite-difference structured grids. More specifically, we explore optimisations related to efficiently scheduling computations to benefit from reusing data stored in the cache.

The performance optimisation of numerical methods based on structured grids is a challenging task that requires knowledge of the computational domain and expertise in computer architecture and software engineering. This burden can be relieved by raising the abstraction level. We express applications through domain-specific languages while automating optimisations through high-level compilers and run-time support. With high-performance, structured, and extensible software being developed, researchers maximise their productivity.

This thesis shows that we can apply advanced performance optimisations without user intervention by using domain-specific languages to express a class of numerical methods based on structured grids. Compilers often improve the performance of computational kernels by translating the problem specification into low-level code (e.g., C) and applying sophisticated transformations. Compilers abstract away these transformations from the user side. In a world of low-quality or non-optimising compilers, the user would need to manually write, specialise, and tune each application for each architecture. This, however, would be unrealistic, as all fundamental concerns of software engineering, including maintainability, extendibility, and modularity, would be compromised.

As this thesis shall demonstrate, improving the performance of applications via domain-specific compiler optimisations delivers significant benefits:

1. Simplicity: The high-level symbolic syntax captures both the program structure and domain properties, facilitating the compiler's job.

2. Portability: The user can write once and run everywhere. Multiple platforms can be supported without requiring user changes in the code written in the domain-specific language. The compiler drives the code generation according to the target architecture.

3. Separation of concerns. Application specialists focus on modelling applications using a symbolic notation close to the mathematics of textbooks. In contrast, performance optimisation is hidden in the lower abstraction layers where computer scientists can focus on their domain and ignore the higher levels of mathematics or physics. Productivity in interdisciplinary teams is improved.

1.2. Thesis outline, state-of-the-art and contributions

1.2.1. Thesis Outline

This section briefly introduces the outline of this thesis and familiarises the reader with the class of problems that this thesis is focused on and the contributions offered.

This thesis focuses on applying temporal blocking, a cache locality optimisation, to stencil computations that form part of expensive pipelines in seismic and medical imaging. Temporal blocking is complicated to apply manually, and we aim to generalise it as an automated optimisation pass within the Devito Domain-Specific Language and compiler framework. Practical simulations in imaging problems differentiate from standard stencil kernels as they usually have additional complexities mainly stemming from the application domain. These complexities stem from sources injecting waves and receivers that collect traces in positions not aligned to the structured finite-difference grid.

1.2.2. Related work

This section positions the context of the related work within the more general scientific topics discussed in this thesis. Here we only briefly refer to the related work that has influenced our work, and we mainly categorise this work into three parts:

1. We are working within a Devito [Luporini et al., 2020], a DSL and compiler framework that aims to automate the solution of PDEs and apply automated optimisations. We also discuss other frameworks that aim to solve PDEs from high-level abstractions and help facilitate optimisation passes as in Rathgeber et al. [2017], Reguly et al. [2018], Jacobs et al. [2017], Balay et al. [2015], Ragan-Kelley et al. [2017]. These frameworks and others will be discussed in more detail in Section 2.2.2.
2. Secondly, we are working on cache-related optimisations for stencil computations that stem from solving PDEs through the FD method. Since we focus on temporal blocking, we are reviewing the state of the art in applying temporal blocking optimisations as in Wonnacott [2000], Guohua Jin et al. [2001], Wonnacott [2004], Wellein

et al. [2009], Strzodka et al. [2011], we specially discuss the wavefront method [Yount and Duran, 2016], and additionally, more sophisticated variants such as diamond temporal blocking [Bertolacci et al., 2015, Bandishti et al., 2012, Malas et al., 2015, Muranushi and Makino, 2015, Levchenko and Perepelkina, 2017, Akbudak et al., 2020] and Wang and Chandramowlishwaran [2020]. Finally, in Table 2.2, we will be summarising a number of frameworks based on their attributes. More details on stencil-related frameworks and their optimisation can be found in Sections 2.4.2, 2.4.5 and 3.1.

3. Since we are aiming to automate temporal blocking within larger frameworks, as in answering how past research tried to automate works from item 2 in frameworks such as the ones mentioned in item 1. Notable work on automating the application of temporal blocking variants within larger frameworks can be found in Holewinski et al. [2012], Yount et al. [2017], Bertolacci et al. [2015], Kuroda et al. [2017] and others. More related work and discussion can be found in Section 4.2.1.

Again, here we only aim to provide a "map" of the related work in this thesis, and we kindly recommend that the reader focuses on the corresponding sections in every Chapter.

1.2.3. Thesis Contributions

Summarising, this thesis offers original contributions in two areas:

1. the application of *temporal blocking* or *time tiling* to a class of non-trivial finite difference kernels involving sources and receivers - operators that inject and interpolate signals at points not aligned with the grid.
2. the generalisation of *temporal blocking* as an automated compiler transformation, whose aim is to improve data locality in time-stepping iterative solvers and benefit performance

Both topics span the following research directions:

- automating temporal blocking as a loop transformation for a broad scope of PDE-dominated solvers expressed through a high-level DSL

- exploring temporal blocking effectiveness as a hardware-agnostic transformation, evaluating it on a number of CPUs with different characteristics
- providing infrastructure towards facilitating the adoption of other temporal blocking schemes such as split or diamond temporal blocking

The thesis comprises five chapters, including the present introductory chapter and the conclusions. Chapter 2 establishes the foundation for the contributions in the subsequent chapters. It includes the necessary knowledge to familiarise the reader with the topic of this thesis and refers to the state-of-the-art related work from the literature. The essential theory of the finite difference method is discussed, and the transition from PDEs to stencils is presented. We refer to the frameworks that facilitate this transition to stencils. Afterwards, we present stencil-related optimisations, focusing specifically on loop blocking. These optimisations are demanding and challenging to apply by hand. In the last decade, notable work has been done to automate these optimisations for many reasons, which will be discussed later. The frameworks that work towards automated optimisations are presented and discussed.

Chapters 3 and 4, which respectively treat the aforementioned topics 1 and 2, represent the core contributions of this thesis, as detailed next:

Chapter 3 In this chapter, we present a scheme to enable temporal blocking to finite-difference stencil kernels with the presence of sparse non-aligned to the structured grid operators [Bisbas et al., 2021]. These kernels are common when modelling wave-propagation kernels that require the presence of scattering (sources that emit waves) and gathering operations (receivers that interpolate values). These operators are non-trivial, and their inherent structure makes applying temporal blocking schemes such as wavefront temporal blocking challenging. This work was co-authored by Fabio Luporini (Imperial College London, Devito Codes Ltd.); Mathias Louboutin (Georgia Tech, Atlanta, USA); Rhodri Nelson (Imperial College London); Gerard J. Gorman (Imperial College London) and supervised by Paul H.J. Kelly (Imperial College London). This thesis's author entirely performed the

algorithm's design and implementation to enable temporal blocking for sparse operations non-aligned to the structured grid.

Chapter 4 We extend and complement the work [Bisbas et al., 2021] presented in Chapter 3 to fully automating temporal blocking as a compiler pass in the Devito DSL Luporini et al. [2020]. Users can benefit from the enhanced cache locality of temporal blocking without writing this technique's manually tedious and error-prone code structures. We present experimental results showing performance gains for various PDE operators on several CPUs.

This thesis advances state-of-the-art automated cache optimisations by introducing a generalised way to automate temporal blocking code generation. We exploit compiler technology to automate new performance optimisations for structured grid computations. We emphasise stencil kernels deriving from the finite difference method and automatically produced through Devito.

1.3. Dissemination

The work implemented for this thesis is released under open-source licenses. The underlying theory has been exposed to the scientific community through the following publications and presentations:

Publications From this thesis derives one main publication; a further publication is planned for the most recent achievements in automated temporal blocking.

- G. Bisbas, F. Luporini, M. Louboutin, R. Nelson, G. J. Gorman, and P.H.J. Kelly. *Temporal blocking of finite-difference stencil operators with sparse "off-the-grid" sources (IPDPS), 2021 (Bisbas et al. [2021])*. Available online. This conference paper, presented in Chapter 3, describes the work on scheduling temporal blocking for non-aligned to the structured grid operators.

The author of this thesis has also been involved in other works which do not directly contribute to the thesis.

- Mathias Louboutin, Fabio Luporini, Philipp Witte, Rhodri Nelson, George Bisbas, Jan Thorbecke, Felix J. Herrmann, and Gerard Gorman. 2020. Scaling through abstractions high-performance vectorial wave simulations for seismic inversion with Devito. (2020). Available online
- Rhodri Nelson, Fabio Luporini, Mathias Louboutin, George Bisbas, Gerard Gorman (2020). TheMatrix: An automated cross-platform benchmarking suite. Submitted to The Journal of Open Source Software, Available online

Presentations A number of formal and informal presentations were given at various conferences, workshops, and meetings. The most relevant, in chronological order, are:

- G. Bisbas, F. Luporini, M. Louboutin, R. Nelson, G. J. Gorman, and P.H.J. Kelly. *Temporal blocking of finite-difference stencil operators with sparse "off-the-grid" sources (IPDPS), 2021 (Bisbas et al. [2021])*. Oral presentation, IPDPS 2021 (IPDPS21). [Video]
- *Accelerating real-world stencil computations using temporal blocking: handling sparse sources and receivers.* Poster presentation, Supercomputing Conference 2019 (SC19). [Poster]
- G. Bisbas, F. Luporini, M. Louboutin, R. Nelson, G. Gorman, P.H.J. Kelly *Temporal blocking for wave propagation with sparse off-the-grid sources* Presented at Rice Oil and Gas HPC (OGHPC 2021) conference. [Slides] [Video]
- G. Bisbas, F. Luporini, M. Louboutin, R. Nelson, G. Gorman, P.H.J. Kelly *Temporal blocking of finite-difference stencil operators with sparse "off-the-grid" sources* Presented at the 21st Workshop on Compilers for Parallel Computing (CPC21, Porto) conference.
- *Temporal blocking of finite-difference stencil operators with sparse non-grid-aligned sources and receivers in Devito* Presented at Domain-Specific Languages in High-Performance Computing 2020. [Slides] [Video]
- *Automated Temporal Blocking in the Devito Compiler* Presented at Stencil Computation for Scientific Applications MinisymposIum, held with SIAM CSE23

In preparation The following work is under preparation, and we aim to submit it in the near future. It is mostly the work described in Chapter 4.

- G. Bisbas, F. Luporini, G. J. Gorman, and P.H.J. Kelly. *Automated temporal blocking code generation from high-level abstractions, 202x*.

Software The research described in this thesis has contributed towards improving the Devito framework.

- Devito, GitHub repository [Luporini et al., 2021]

Acknowledgments

The research presented in this thesis was funded by the following Engineering and Physical Sciences Research Council (EPSRC) grants:

EP/R029423/1 PRISM: Platform for Research In Simulation Methods

EP/V001493/1 Gen X: ExCALIBUR working group on Exascale continuum mechanics through code generation.

EP/L016796/1 HiPEDS Center for Doctoral Training

EP/W007789/1 Efficient Cross-Domain DSL Development for Exascale

Chapter 2

Background and Related Work

This chapter establishes the necessary scientific background for the main chapters of this thesis. In Section 2.1, we present the mathematical theory establishing the finite difference (FD) method. We discuss the theory of how the FD method lowers the solution of partial differential equations to stencil kernels. Some of the most commonly used examples for educational purposes are presented to the reader.

In Section 2.2 we use the Devito framework to model these partial differential equation problems and automatically generate high-performing code. We primarily present FD software abstractions used for solving partial differential equations, however we also briefly present a few FE, FV and other frameworks. These abstractions may have a lower-level API compared to Devito. By lower-level APIs, we describe those APIs that are closer to the final generated C code. The level of an API, may be relative to the framework, as an abstraction in some framework may be either an IR description or DSL anstraction. For this reason we distinguish them in categories based on the semantics they describe.

The next step for the efficient solution of partial differential equations is accelerating their execution. Several techniques facilitate this, one of the most prominent ones being cache optimisations. The following Section 2.3 is focused on a subset of these optimisations, namely cache blocking, and more specifically in temporal blocking, which will be discussed in Section 2.3.3. We focus on conventional and advanced loop blocking techniques. We briefly discuss other non-cache-related loop optimisations. We observe that manually handling these optimisations is challenging;

thus, automating them is essential. For this purpose, several frameworks have been developed.

Most of the frameworks presented in subsections 2.2.2 and 2.2.3 have machinery to efficiently lower the high-level DSL specification to machine executable code. In addition, some tools primarily focus on applying and automating optimisations at different levels of abstraction. In Section 2.4, we review these frameworks and the state-of-the-art literature in performance optimisation, focusing on frameworks that aim to automate compiler optimisations. We present automatically generated optimised code from Devito and discuss domain-specific optimisations that influenced this work and tools supporting a great variety of optimisations.

Finally, in Section 2.5, we briefly refer to the terminology adopted throughout the thesis.

2.1. The Finite Difference Method

The finite difference method (FDM) [LeVeque, 2007, Smith, 1985] is used in various domains to approximate the solution of partial differential equations (henceforth, PDEs). It helps solve various linear, non-linear, time-dependent and independent problems. It is considered one of the oldest and simplest methods to solve differential equations dating back to the 18th century in the works of L. Euler and C. Runge. Scientists started to use finite difference methods more heavily in the early 1950s. At the same time, their wider adoption was prompted by the development of computer systems that offered a pathway to solve simulation problems more efficiently. Nowadays, FDM is considered one of the most common approaches to numerical solutions of PDEs, along with finite element methods [Zienkiewicz et al., 1977]. Both methods apply to boundary value problems. FDM is mostly used for problems with regular geometries and simpler boundary conditions. In these cases, FDM is the recommended approach and can provide efficient solutions. FEM can handle better irregular geometries, i.e. complex shapes more naturally [Collatz, 2012].

Over the last decades, the accuracy, stability and convergence of the FDM method have been firmly established. In this Chapter, we review the essential theoretical background for understanding the contributions in Chapters 3 and 4. The content and the notation used in this section are

inspired by LeVeque [2007], Langtangen and Linge [2017], Smith [1985], and Peirce [2018]. For a more comprehensive review of the subject, the reader is invited to refer to Levy and Lessman [1992] and Thomas [2013].

2.1.1. Finite Difference Approximations

Finite difference methods approximate solutions to differential equations (henceforth DE). This solution is a discrete approximation to the DE and should satisfy a given relationship between any expression of derivatives defined over some given region of space and time, with boundary conditions along the edges of this domain and initial conditions along the domain [LeVeque, 2007]. Usually, this is a complex problem, and finding an analytic formula for the solution is a challenging task that may even be impossible. By replacing the derivatives in the differential equations with finite difference approximations, we yield a large but finite algebraic system of equations. This system can then be solved numerically.

We approximate a function's derivatives by using only values of the function itself at discrete points. We will now present how to derive simple one-sided and centered approximations.

Let $f(x)$ be a one-variable function defined over $x \in \mathrm{X}$ assumed to be smooth. Being smooth means that $f(x)$ has continuous derivatives, and each derivative is a well-defined bounded function over a domain $\mathrm{X}' \subseteq \mathrm{X}$ that contains a particular point of interest $\bar{x} \in \mathrm{X}'$. Let us approximate $f'(\bar{x})$ by finite differences using only values of f at a finite number of points near x.

We start with a one-sided approximation to f' since f is evaluated only at values of $x \geq \bar{x}$ for some small value of h.

$$\Delta_+ f(\bar{x}) \equiv \frac{f(\bar{x}+h) - f(\bar{x})}{h} \tag{2.1}$$

This is motivated by the standard definition of the derivative as the limiting value of this expression as $h \to 0$. Note that $\Delta_+ f(\bar{x})$ is the slope of the line interpolating f at the points $\bar{x}$ and $\bar{x}+h$ (see Figure 2.1).

The expression 2.1 is a one-sided approximation to f' since f is evaluated only at values of $x \geq \bar{x}$. Another one-sided approximation would be

$$\Delta_- f(\bar{x}) \equiv \frac{f(\bar{x}) - f(\bar{x}-h)}{h} \tag{2.2}$$

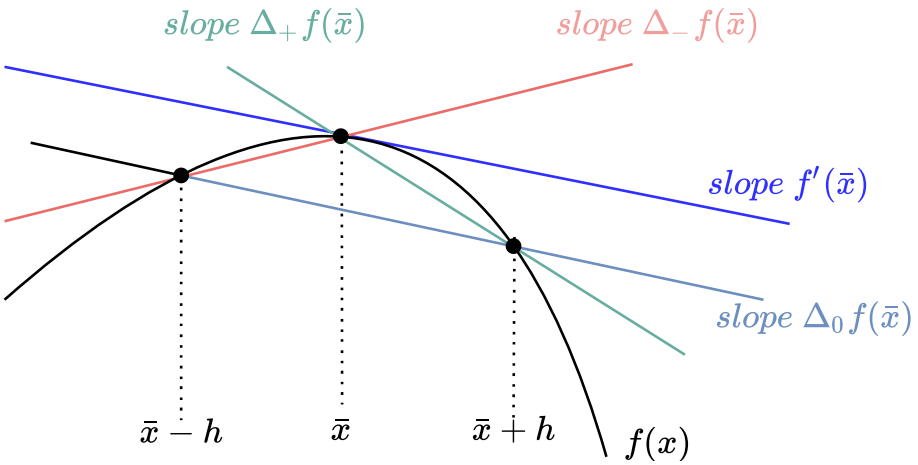


Figure 2.1.: Various approximations to $f'(\bar{x})$ interpreted as the slope of secant lines. Adapted and edited from LeVeque [2007]

Both 2.1 and 2.2 are first order accurate approximations to $f'(\bar{x})$. First-order accurate means that the size of the error is roughly proportional to h itself.

Other than one-sided approximations, we can use the centered approximation:

$$\Delta_0 f(\bar{x}) \equiv \frac{f(\bar{x}+h) - f(\bar{x}-h)}{2h} = \frac{1}{2}\left(\Delta_+ f(\bar{x}) + \Delta_- f(\bar{x})\right) \tag{2.3}$$

The centered approximation is the slope of the line interpolating f at $\bar{x} - h$ and $\bar{x} + h$ and is simply the average of the two one-sided approximations 2.1, 2.2. From Figure 2.1, it should be clear that we would expect $\Delta_0 f(\bar{x})$ to give a better approximation than either of the one-sided approximations. This gives a second-order accurate approximation as the error is proportional to h^2. Since $h \to 0$, h^2 is much smaller than the error h in each first-order approximation.

By approximating using more points, we can increase the order of accuracy. We briefly refer to a third-order accurate approximation:

$$\Delta_3 u(\bar{x}) \equiv \frac{1}{6h}\left[2u(\bar{x}+h) + 3u(\bar{x}) - 6u(\bar{x}-h) + u(\bar{x}-2h)\right] \tag{2.4}$$

The error in the third-order accurate approximation is proportional to h^3 when $h \to 0$ is small. Assuming C is a constant and p is the order of

accuracy if we approximate with even more points, the error follows:

$$E(h) \approx Ch^p,$$

then

$$\log|E(h)| \approx \log|C| + p\log h.$$

So on a log-log scale, the error behaves linearly with a slope equal to p.

An example of deriving the size of errors in finite difference approximations, as well as the log-log scale is shown in Appendix A.1.

2.1.2. Derivation from Taylor's polynomial

This subsection will use Taylor's theorem to derive finite difference approximations and look into their truncation errors. We start by trying to derive Equation 2.1 by using Taylor series. The derivative of a function f at the point x is defined as the limit of an approximation of the derivative $f'(x)$. The smaller the h, the better the approximation of the derivative of f is:

$$f'(x) = \lim_{h\to 0} \frac{f(x+h) - f(x)}{h} \tag{2.5}$$

We will now use the Taylor series to derive an appropriate formula for a finite difference approximation to $f\prime(\bar{x})$ based on some given set of points. We start by expanding each function value of f in a Taylor series about the point $\bar{x}$. This expansion is considered a standard approach to analysing the error in a finite difference approximation. Taylor's theorem with remainder gives the following Taylor series expansion:

$$f(x+h) = f(x) + hf'(x) + h^2\frac{f''(\xi)}{2!} \tag{2.6}$$

where $\xi \in (x, x+h)$. Rearranging terms we get:

$$\frac{f(x+h) - f(x)}{h} - f'(x) = h\frac{f''(\xi)}{2} \tag{2.7}$$

Equation 2.7 shows that the error is proportional to h to the power of 1, so $\dfrac{f(x+h) - f(x)}{h}$ is said to be a "first-order" approximation (Similar to 2.1, 2.2).

If $h > 0$ is a finite (as opposed to infinitesimal) positive number, then:

$$\frac{f(x+h)-f(x)}{h} \tag{2.8}$$

is called the first-order or $O(h)$ *forward difference* of $f'(x)$ as in Equation 2.1. By combining different Taylor series expansions, we can obtain approximations of $f\prime(x)$ of various orders. For instance, subtracting the two expansions:

$$\begin{aligned} f(x+h) &= f(x)+hf'(x)+h^2\tfrac{f''(x)}{2!}+h^3\tfrac{f'''(\xi_1)}{3!}, \quad \xi_1 \in (x, x+h) \\ f(x-h) &= f(x)-hf'(x)+h^2\tfrac{f''(x)}{2!}-h^3\tfrac{f'''(\xi_2)}{3!}, \quad \xi_2 \in (x-h, x) \end{aligned}$$

gives

$$f(x+h)-f(x-h) = 2hf'(x)+h^3\frac{\left(f'''\left(\xi_1\right)+f'''\left(\xi_2\right)\right)}{6}$$

so that

$$\frac{f(x+h)-f(x-h)}{2h}-f'(x) = h^2\frac{\left(f'''\left(\xi_1\right)+f'''\left(\xi_2\right)\right)}{12}$$

Hence $\dfrac{f(x+h)-f(x-h)}{2h}$ is an approximation of $f'(x)$ whose error is proportional to h^2. It is called the second-order or $O(h^2)$ *centered difference* approximation of $f'(x)$ as in Equation 2.3

If we use expansions with more terms, higher-order approximations can be derived, e.g. consider

$$\begin{aligned} f(x+h) &= f(x)+hf'(x)+h^2\frac{f''(x)}{2!}+h^3\frac{f'''(x)}{3!}+h^4\frac{f^{(4)}(x)}{4!}+h^5\frac{f^{(5)}\left(\xi_1\right)}{5!} \\ f(x-h) &= f(x)-hf'(x)+h^2\frac{f''(x)}{2!}-h^3\frac{f'''(x)}{3!}+h^4\frac{f^{(4)}(x)}{4!}-h^5\frac{f^{(5)}\left(\xi_2\right)}{5!} \\ f(x+2h) &= f(x)+2hf'(x)+4h^2\frac{f''(x)}{2!}+8h^3\frac{f'''(x)}{3!}+16h^4\frac{f^{(4)}(x)}{4!}+32h^5\frac{f^{(5)}\left(\xi_3\right)}{5!} \\ f(x-2h) &= f(x)-2hf'(x)+4h^2\frac{f''(x)}{2!}-8h^3\frac{f'''(x)}{3!}+16h^4\frac{f^{(4)}(x)}{4!}-32h^5\frac{f^{(5)}\left(\xi_4\right)}{5!} \end{aligned}$$

By taking $8\times(f(x+h)-f(x-h))-(f(x+2h)-f(x-2h))$ we cancel out the h^2 and h^3 terms. Then by rearranging, we get a fourth-order $O(h^4)$ centered difference approximation of $f'(x)$. Approximations of higher derivatives $f''(x), f'''(x), f^{(4)}(x)$ etc. can be obtained in a similar manner.

For example, adding

$$f(x+h) = f(x) + hf'(x) + h^2\frac{f''(x)}{2!} + h^3\frac{f'''(x)}{3!} + h^4\frac{f^{(4)}(\xi_1)}{4!}\dots$$
$$f(x-h) = f(x) - hf'(x) + h^2\frac{f''(x)}{2!} - h^3\frac{f'''(x)}{3!} + h^4\frac{f^{(4)}(\xi_2)}{4!}\dots$$

and by rearranging terms, we get:

$$\frac{f(x+h) - 2f(x) + f(x-h)}{h^2} - f''(x) = h^2\frac{(f^{(4)}(\xi_1) + f^{(4)}(\xi_2))}{24}$$

Hence, $\frac{f(x+h) - 2f(x) + f(x-h)}{h^2}$ is a second-order centered difference approximation of the second derivative $f''(x)$. Here are some commonly used second- and fourth-order finite difference formulas for approximating first and second derivatives of $f'(x)$:

$O(h^2)$ centered difference approximations (which are approximations often used for discretisations in the physical space):
$f'(x): \quad \{f(x+h) - f(x-h)\}/(2h)$
$f''(x): \quad \{f(x+h) - 2f(x) + f(x-h)\}/h^2$

$O(h^2)$ forward difference approximations (which are approximations often used for time discretisation):
$f'(x): \quad \{-3f(x) + 4f(x+h) - f(x+2h)\}/(2h)$
$f''(x): \quad \{2f(x) - 5f(x+h) + 4f(x+2h) - f(x+3h)\}/h^3$

$O(h^2)$ backward difference approximations :
$f'(x): \quad \{3f(x) - 4f(x-h) + f(x-2h)\}/(2h)$
$f''(x): \quad \{2f(x) - 5f(x-h) + 4f(x-2h) - f(x-3h)\}/h^3$

$O(h^4)$ centered difference approximations :
$f'(x): \quad \{-f(x+2h) + 8f(x+h) - 8f(x-h) + f(x-2h)\}/(12h)$
$f''(x): \quad \{-f(x+2h) + 16f(x+h) - 30f(x) + 16f(x-h) - f(x-2h)\}/(12h^2)$

It is clear from the above equations that higher derivatives and hence smaller approximation errors need more terms for the approximation. Consequently, they have higher computational costs.

There are often problems in science and engineering where an exact formula for $f(x)$ is unknown. We may only have a set of data points

$(x_1,\ y_1), (x_2,\ y_2),\ \ldots, (x_n,\ y_n)$ available to describe the functional dependence $y = f(x)$.

Suppose we need to estimate the rate of change of y with respect to x in such a situation. We can use finite difference formulas to compute approximations of $f'(x)$. Using a forward difference at the left endpoint $x = x_1$, a backward difference at the right endpoint $x = x_n$, and centered difference formulas for the interior points are appropriate.

2.1.3. Finite Difference stencil derivation

In this section, we use the Taylor expansions from Section 2.1.2 to show how we compute finite-difference PDE approximations. We use the definition of the derivative and Taylor series to derive finite difference approximations to a function's first and second derivatives. We then use these finite difference quotients to approximate the heat, the Laplace and the wave equation derivatives. We show how the PDE solution leads to stencil computation. Stencil kernels are computational update patterns that are functions of the nearest neighbouring point values. In a more general definition, a stencil defines the iterative computation of an element in an n-dimensional spatial grid at time t as a function of neighbouring grid elements (space dependencies) at time $t-1, \ldots, t-k$ (time dependencies).

The following sections of this Chapter will present several stencils. A typical stencil update in the context of a scientific simulation has a three-dimensional spatial iteration space and a one-dimensional temporal iteration space. Figure 2.2 illustrates a 1D stencil and its flow dependencies. Each point is updated using values from the previous (one or more) timestep-(s) and its right and left neighbours. Arrows illustrate the flow dependencies. The points that are at the edge of the computational domain need to read from positions that are outside of the computational domain. We extend the domain by the stencil radius size with read-only points to satisfy this dependency. These points are not being updated as they do not need to be. Figure 2.2 shows this setup.

In the literature, the term "stencil" refers to cases where the access rule is an affine function. This is the case of computational methods based on structured grids, such as the finite difference method. Given an element $\boldsymbol{i}$ in the structured mesh, a stencil is a vector-valued function

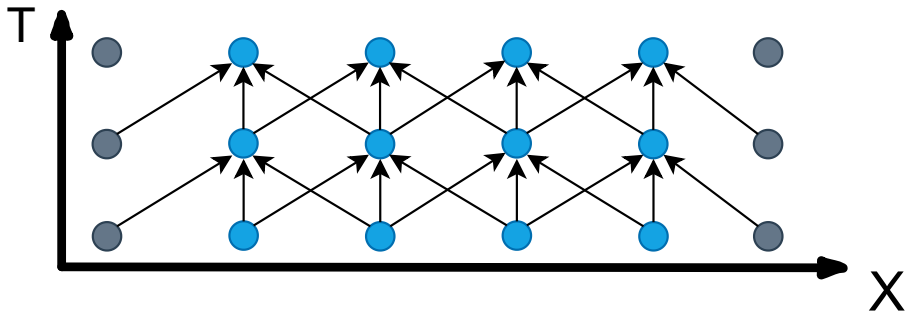


Figure 2.2.: A 1D-3pt Jacobi stencil update. Arrows show the data flow dependencies. Grey points indicate the read-only points that extend the domain.

$f(\boldsymbol{i}) = [f_1(\boldsymbol{i}), f_2(\boldsymbol{i}), ..., f_n(\boldsymbol{i})]$ which fetches the n elements that need to be accessed when updating $\boldsymbol{i}$. Wider stencils in 3D and their respective data dependencies are illustrated in Figure 2.3. A function f_j is affine and usually takes the form $f_j(\boldsymbol{i}) = \boldsymbol{i} * h + o$, with $h, o \in \mathbb{N}$. Usually, the accessed elements are within the neighbouring area of an element $\boldsymbol{i}$.

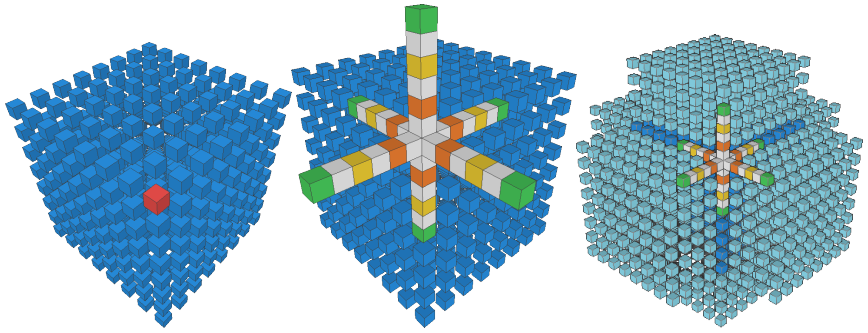


Figure 2.3.: A 6th-order, 3D-19pt stencil update. A point (red) at the edge of a block (blue) depends on a three-point deep halo of neighbouring points which extends outside the block.

The following Sections will detail how we derive finite-difference stencils starting from the PDEs. The scientific background presented in this section has been influenced by lecture notes from Peirce [2018].

2.1.3.1. The Heat equation

We start by considering the following initial-boundary value problem for the heat equation

$$\frac{\partial u}{\partial t} = \alpha^2 \frac{\partial^2 u}{\partial x^2} \quad 0 < x < 1, \quad t > 0 \tag{2.9}$$

$$\text{Boundary conditions :} \quad u(0,t) = 0 \quad u(1,t) = 0 \tag{2.10}$$

$$\text{Initial conditions :} \quad u(x,0) = f(x) \tag{2.11}$$

We replace the derivatives in the heat equation with difference quotients. We regard the relationships between u at (x,t) and its neighbours at a distance Δx apart and at a time Δt later. Similarly to the difference quotient approximations introduced in Section 2.1.2, we use the Taylor series to derive the forward difference in time and the central differences in space.

The **Forward Difference in Time** is:

$$u(x,t+\Delta t) = u(x,t) + \Delta t \frac{\partial u}{\partial t}(x,t) + \frac{\Delta t^2}{2!}\frac{\partial^2 u}{\partial t^2}(x,t) + \cdots$$

Moreover, by re-arrangement and division by Δt, we get:

$$\frac{u(x,t+\Delta t) - u(x,t)}{\Delta t} = \frac{\partial u}{\partial t}(u,t) + O(\Delta t) \tag{2.12}$$

The **Central Differences in Space** (right and left respectively) are:

$$u(x+\Delta x,t) = u(x,t) + \Delta x \frac{\partial u}{\partial x}(x,t) + \frac{\Delta x^2}{2!}\frac{\partial^2 u}{\partial x^2}(u,t) + \frac{\Delta x^3}{3!}\frac{\partial^3 u}{\partial x^3}(x,t) + \frac{\Delta x^4}{4!}\frac{\partial^4 u}{\partial x^2}(x,t) + \cdots$$

$$u(x-\Delta x,t) = u(x,t) - \Delta x \frac{\partial u}{\partial x}(x,t) + \frac{\Delta x^2}{2!}\frac{\partial^2 u}{\partial x^2}(x,t) - \frac{\Delta x^3}{3!}\frac{\partial^3 u}{\partial x^3}(x,t) + \frac{\Delta x^4}{4!}\frac{\partial^4 u}{\partial x^4}(x,t) + \cdots$$

Furthermore, by adding and re-arrangement, we get:

$$\frac{u(x+\Delta x,t)-2u(x,t)+u(x-\Delta x,t)}{\Delta x^2}=\frac{\partial^2 u}{\partial x^2}(x,t)+O\left(\Delta x^2\right) \tag{2.13}$$

By substituting 2.12 and 2.13 into 2.9 we obtain:

$$\frac{u(x,t+\Delta t)-u(x,t)}{\Delta t}=\alpha^2\left(\frac{u(x+\Delta x,t)-2u(x,t)+u(x-\Delta x,t)}{\Delta x^2}\right)+O\left(\Delta t,\Delta x^2\right)$$

and by rearranging, we get:

$$u(x,t+\Delta t)=u(x,t)+\alpha^2\left(\frac{\Delta t}{\Delta x^2}\right)\left(u(x+\Delta x,t)-2u(x,t)+u(x-\Delta x,t)\right) \tag{2.14}$$

We subdivide the spatial interval $[0,1]$ into $N+1$ equally spaced sample points $x_n = n\Delta x$. The time interval $[0,T]$ is subdivided into $M+1$ equal time levels $t_k = k\Delta t$. At each of these space-time sample points, we introduce approximations.

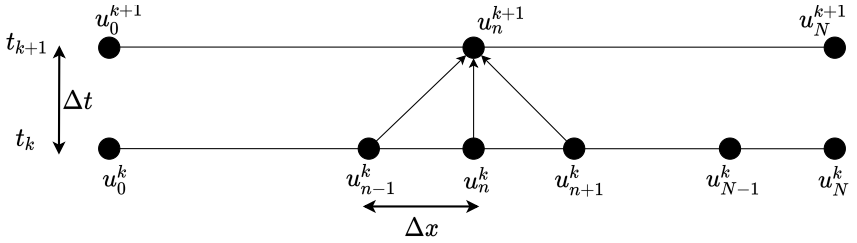


Figure 2.4.: The 1D heat stencil. Figure adapted and edited from Peirce [2018]

Assuming $u_n^k \simeq u(x_n, t_k)$, Eq. 2.14 translates to:

$$u_n^{k+1}=u_n^k+\alpha^2\left(\frac{\Delta t}{\Delta x^2}\right)\left(u_{n+1}^k-2u_n^k+u_{n-1}^k\right)$$

and Figure 2.4 shows schematically the update pattern. How is this update pattern expressed in pseudocode? The following listing shows some pseudocode of how this iterative update translates to code for execution in a computer by a low-level language.

Listing 2.1: C-like pseudocode for the 1D heat stencil.

```
1 for(k = 1; k < nt; k++){
2   for(n = 1; n < N; n++){
```

```
3      u[k+1, n] = u[k, n]
4                  + pow(a, 2) * (dt / pow(dx, 2)) *
5                  (u[k, n+1] - 2u[k, n+1] + u[k, n-1]);
6    }
7  }
```

Implementing Derivative Boundary Conditions: Let's assume that the boundary conditions in Equation 2.10 are changed to

$$\text{Boundary conditions}: \quad u(0,t) = 0, \quad \frac{\partial u}{\partial x}(1,t) = 0.$$

Consider a central difference approximation to $\frac{\partial u}{\partial x}(1,t)$, where $x_N = N\Delta x = 1$,

$$\frac{u\left(x_N + \Delta x, t\right) - u\left(x_N - \Delta x, t\right)}{\Delta x} = 0.$$

Rearranging, we obtain:

$$u\left(x_N + \Delta x, t\right) = u\left(x_N - \Delta x, t\right) \tag{2.15}$$

We notice that since $x_N = 1$, $x_N + \Delta x$ refers to a position that is outside the computational domain. To accommodate this, we introduce an extra column u_{N+1}. This extra column holds the same values of u_{N-1}. We can now proceed in applying the same difference approximation for the heat equation to the column x_N:

$$u_N^{k+1} = u_N^k + \alpha^2 \left(\frac{\Delta t}{\Delta x^2}\right)\left(u_{N+1}^k - 2u_N^k + u_{N-1}^k\right) \tag{2.16}$$

while $u_{N+1}^k = u_{N-1}^k$ (see Equation 2.15) since column u_{N-1}^k is copied to column u_{N+1}^k. Another way to implement this boundary condition without introducing the additional column is by eliminating u_{N+1} from Equations 2.15 and 2.16:

$$u_N^{k+1} = u_N^k + 2\alpha^2 \left(\frac{\Delta t}{\Delta x^2}\right)\left(u_{N-1}^k - u_N^k\right) \tag{2.17}$$

If Equation 2.17 is implemented at x_N there is no need for an extra column u_{N+1} or to implement the difference equation given in Eq. 2.16 as the derivative boundary condition is handled automatically.

2.1.3.2. The Wave equation

We consider the following initial boundary value problem for the Wave Equation:

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2} \quad 0 < x < L \tag{2.18}$$

$$\text{Boundary Conditions :} \quad u(0,t) = 0; \quad u(L,t) = 0 \tag{2.19}$$

$$\text{Initial Conditions :} \quad u(x,0) = f(x) \tag{2.20}$$

$$\frac{\partial u}{\partial t}(x,0) = g(x) \tag{2.21}$$

We consider a finite difference structured grid where $x_n = n\Delta t,\ t_k = k\Delta t$ and let the corresponding grid point values be denoted by $u_n^k \simeq u(x_n, t_k)$.

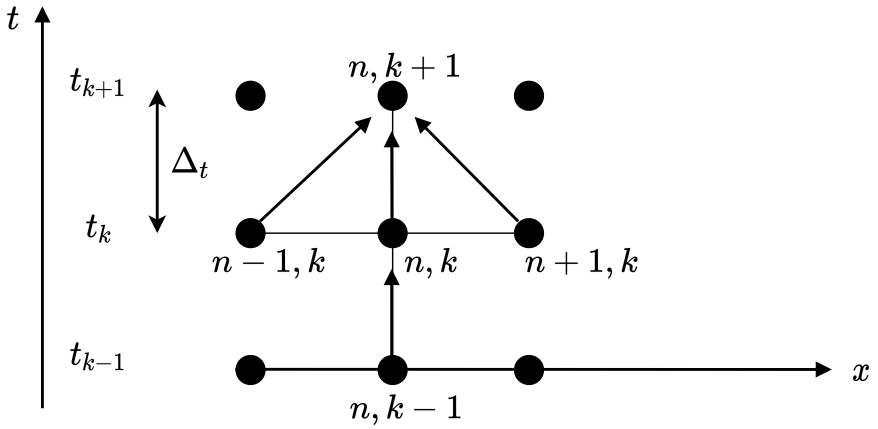


Figure 2.5.: The 1D wave equation stencil. Adapted and edited from Peirce [2018]

By approximating derivatives using central finite differences both in space and time, we obtain

$$\frac{u_n^{k+1} - 2u_n^k + u_n^{k-1}}{\Delta t^2} = c^2 \left(\frac{u_{n+1}^k - 2u_n^k + u_{n-1}^k}{\Delta x^2} \right) + O\left(\Delta x^2, \Delta t^2\right) \tag{2.22}$$

Therefore, the update pattern yields:

$$u_n^{k+1} = 2u_n^k - u_n^{k-1} + \left(\frac{c\Delta t}{\Delta x} \right)^2 \left(u_{n+1}^k - 2u_n^k + u_{n-1}^k \right) \tag{2.23}$$

We notice that time dependencies extend back in time for more than one grid point. We observe that the discrete Equation 2.24 involves three distinct levels of time in which known data is transferred from steps $k-1$ and k to step $k+1$. To update for the most recent term in time, we have to rearrange and compute:

$$\underbrace{u_n^{k+1}}_{\text{time level } k+1} = \underbrace{r^2 u_{n+1}^k + 2\left(1-r^2\right) u_n^k + r^2 u_{n-1}^k}_{\text{time level } k} - \underbrace{u_n^{k-1}}_{\text{time level } k-1} \tag{2.24}$$

where $r = (c\Delta t/\Delta x)$ and is known as the Courant Number.

The following listing shows some pseudocode of how this iterative update translates to stencil code for execution in a computer by a low-level language.

Listing 2.2: C-like pseudocode for the 1D wave equation stencil.

```
r = c * dt * dx
for(k = 1; k < nt; k++){
  for(n = 1; n < N; n++){
    u[k+1, n] =   pow(r, 2)        * u[k, n+1]
                + 2*(1- pow(r,2)) * u[k, n]
                + pow(r, 2)        * u[k, n-1]
                                   - u[k-1, n-1] ;
  }
}
```

Initial Conditions - Starting the Solution In this problem, we have a scheme with three levels of time. This scheme makes it challenging to apply the initial conditions from Equation 2.20. If we imagine a row of false mesh points at time $t = -\Delta t = t_{-1}$, then the initial velocity condition 2.21 can be approximated using central differences as:

$$\frac{u_n^1 - u_n^{-1}}{2\Delta t} = g\left(x_n\right) \tag{2.25}$$

therefore

$$u_n^{-1} = u_n^1 - 2\Delta t g(x_n) \tag{2.26}$$

Now we assume that the Discrete Wave Equation 2.24 also holds at $t = 0$ so that

$$u_n^1 = r^2 u_{n+1}^0 + 2\left(1 - r^2\right) u_n^0 + r^2 u_{n-1}^0 - u_n^{-1} \tag{2.27}$$

Substituting 2.26 into 2.27 and re-arranging we obtain:

$$u_n^1 = \frac{1}{2}\left(r^2 u_{n+1}^0 + 2\left(1 - r^2\right) u_n^0 + r^2 u_{n-1}^0\right) + \Delta t g(x_n) \tag{2.28}$$

Since $u_n^0 = f\left(x_n\right)$ and $g\left(x_n\right)$ are known, we are now capable of specifying the first two rows of grid points. This is sufficient to start the recursion 2.24 for all subsequent steps.

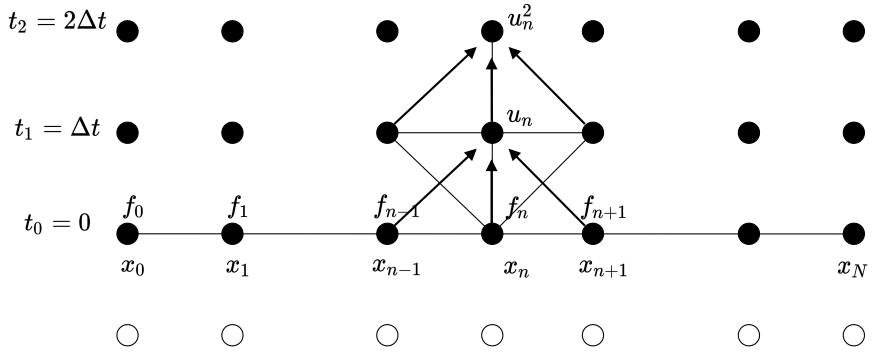


Figure 2.6.: Initial conditions and the 1D wave equation stencil. The white nodes are used to derive Eq. 2.28 but are not used in the computation. Adapted and edited from Peirce [2018].

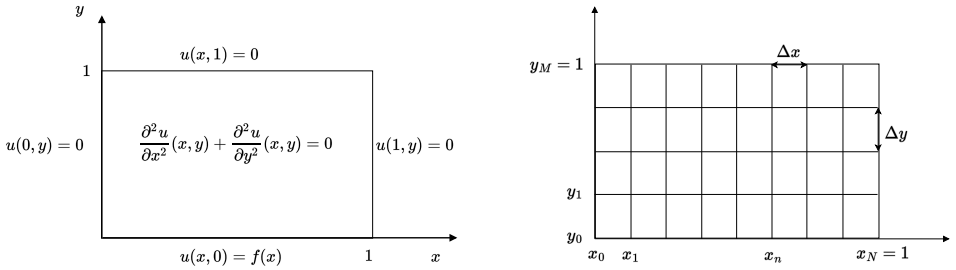
2.1.3.3. The Laplace equation

Let us consider the following boundary value problem:

$$\frac{\partial^2 u}{\partial x^2}(x,y) + \frac{\partial^2 u}{\partial y^2}(x,y) = 0 \quad 0 < x, y < 1 \tag{2.29}$$

$$\text{BC} : u(0,y) = 0; \quad u(1,y) = 0; \quad u(x,0) = f(x); \quad u(x,1) = 0. \tag{2.30}$$

Compared to the heat and the wave equation, this problem has an additional dimension in space. Figure 2.7 helps to visualise the computational domain of the problem. As before, we replace the second derivatives in Eq. 2.29 with central difference quotients that are second-order accurate:



(a) Equations on the computational domain **(b)** The computational grid

Figure 2.7.: Equations on the domain and the computational grid

$$\frac{u(x+\Delta x,y)-2u(x,y)+u(x-\Delta x,y)}{\Delta x^2}=\frac{\partial^2 u}{\partial x^2}(x,y)+O\left(\Delta x^2\right) \tag{2.31}$$

$$\frac{u(x,y+\Delta y)-2u(x,y)+u(x,y-\Delta y)}{\Delta y^2}=\frac{\partial^2 u}{\partial y^2}(x,y)+O\left(\Delta y^2\right) \tag{2.32}$$

We partition the interval $0 \leq x \leq 1$ into $(N+1)$ equally spaced grid points $x_n = n\Delta x$ and the interval $0 \leq y \leq 1$ into $(M+1)$ equally spaced grid points $y_m = m\Delta y$. Replacing the derivatives in Eq. 2.29 by the difference quotients in Eq. 2.31 and Eq. 2.32, and representing the mesh values at (x_n, y_m) by $u_{nm} \simeq u\left(x_n, y_m\right)$ we obtain:

$$\frac{u_{n+1,m}-2u_{n,m}+u_{n-1,m}}{\Delta x^2}+\frac{u_{n,m+1}-2u_{n,m}+u_{n,m-1}}{\Delta y^2}= \left(u_{xx}+u_{yy}\right)_{(x_n,x_m)}+O\left(\Delta x^2,\Delta y^2\right)$$

If we choose $\Delta x = \Delta y$, then we obtain

$$u_{n+1,m}+u_{n-1,m}+u_{n,m+1}+u_{n,m-1}-4u_{n,m}=0 \tag{2.33}$$

where $1 \leq n, m \leq (N-1), (M-1)$

Figure 2.8 shows the finite difference stencil that relates $u_{n,m}$ to its four nearest neighbours. This is a system of $(N-1) \times (M-1)$ unknowns for the values of $u_{n,m}$ in the interior of the computational domain. The boundary values stemming from Equation 2.30 are already specified!

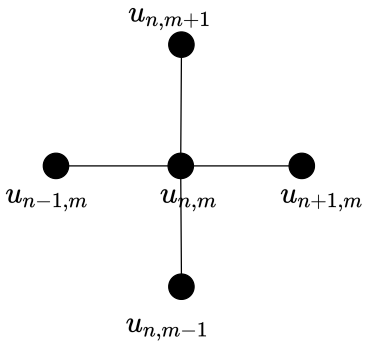


Figure 2.8.: The Jacobi stencil. Adapted and edited from Peirce [2018]

Solving the System of Equations by Jacobi Iteration We will now solve the system of Equations 2.33 by looping through each of the mesh points and updating $u_{n,m}$ according to Equation 2.33 under the assumption that the nearest neighbours already have values close to the exact solution. We will keep updating the point values until the difference between successive iterations converges below a certain tolerance threshold.

To implement this iterative update pattern, we rewrite the discrete Laplace Equation 2.33 in the form:

$$u_{n,m}^{k+1} = \frac{u_{n+1,m}^{k} + u_{n-1,m}^{k} + u_{n,m+1}^{k} + u_{n,m-1}^{k}}{4} \tag{2.34}$$

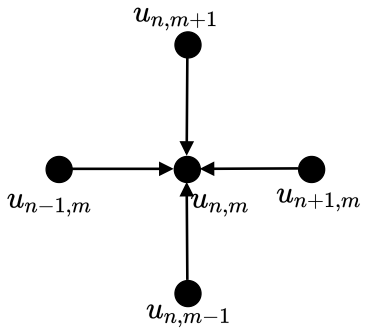


Figure 2.9.: The update pattern of a Jacobi stencil. Adapted and edited from Peirce [2018]

The following listing shows some pseudocode of how this iterative update translates to code for execution in a computer by a low-level language.

Listing 2.3: C-like pseudocode for the 2D Laplace equation stencil.

```
1 for(k = 1; k < nt; k++){
2   for(n = 1; n < N; n++){
3     for(m = 1; m < M; m++){
4       u[k+1, n, m] = 0.25 * (u[k, n+1, m] + u[k, n, m+1] + u[k, n
            -1, m] + u[k, n, m-1] );
5     }
6   }
7 }
```

Thus, $u_{n,m}$ is the average value of its (four) nearest neighbours. We introduce a new superscript index k to represent the grid point values at the $k-$th iteration. Each iteration can be viewed as taking successive neighbour averages until the change value is below a certain threshold. At this point, the value of $u_{n,m}$ equals the average of the values at its mesh neighbours.

This section showed how we derive stencils from partial differential equations through Taylor polynomials. The solution of PDEs can be described through stencils operating on n-dimensional structured grids. Acceleration of stencil computations is the end goal of the work in this thesis. The following Section 2.2 will show how the process of deriving stencils can be automated through several frameworks.

2.2. Automating PDE solutions through abstractions

The previous sections showed how by starting from a problem modelled through a system of Partial Differential Equations (Section 2.1) we can transition to stencil expressions (Section 2.1.3) that are used to update our computational domain iteratively. In scientific simulations, this process involves coordination and cooperation from interdisciplinary scientists. Interdisciplinary teams must work together so that mathematicians, physicists, geophysicists, and chemists can focus on the PDEs and the problem modelling. In contrast, computer, software and compiler engineers can focus on improving the execution of stencil computations on a plethora of hardware. This practice enhances productivity as each scientist can work on their domain where their expertise can be more helpful. However, it is often impossible to form teams with all the skills required to efficiently run the above pipeline from top to bottom.

The need to automatically lower the PDE math specification to stencil ex-

pressions contributed to developing Domain-Specific Languages for PDEs. As stated in Van Deursen et al. [2000]: *A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power is focused on and usually restricted to a particular problem domain.* According to this definition, the critical characteristic of DSLs is their focused expressive power. The slow and tedious development cycle of high-performing, robust, and portable code for finite-difference solvers has stimulated academia and the industry to develop approaches based on automated code generation. Domain-specific languages were a pathway for bridging the communication gap between interdisciplinary scientists. They also helped increase productivity by helping scientists focus on their domain rather than other non-domain tasks that reduced their available time.

Several efforts in the past tried to separate the concerns of modelling PDEs over stencil computation. For this objective, several frameworks have been implemented. These frameworks provide the infrastructure to model PDEs, optimise, and compute stencil expressions through several steps. This subsection refers to the software frameworks that focus on automating the Finite Difference Method. We start with Devito and review other notable projects.

2.2.1. Automating finite differences with Devito

The Devito project [Luporini et al., 2020] is a representative example of success in the context of automating the solution of PDEs through explicit finite differences. Devito was initially influenced by Firedrake [Rathgeber et al., 2017]. We will refer to Firedrake in Section 2.2.3. In Devito, a domain-specific language helps formulate a problem's mathematics at a high level of abstraction. Subsequently, the Devito compiler infrastructure manipulates the mathematical, symbolic representation. It transforms it into a finite-difference stencil representation with the help of SymPy [Meurer et al., 2017]. This transformation takes place through a compilation process that involves several steps. Compilation passes operate in the lowered stencil representation to reduce the arithmetic cost of the mathematical expressions. We often use the term flop-reduction for this process. This stencil representation is then lowered to a new representation of loop-iterative

stencil computation. Subsequently, the compiler performs loop-related optimisations to this new intermediate representation (IR). Devito is written in Python to simplify the analysis of the top-level domain-specific language and synthesis of the IR towards the automatic generation of C code. Afterwards, the compiler translates the representation into C code. The C code is then just-in-time compiled and executed (see Figure 2.10). Throughout this thesis, we focus on Devito. The algorithms and the techniques presented here have been developed and integrated within this framework, though all the ideas potentially apply in other contexts and frameworks.

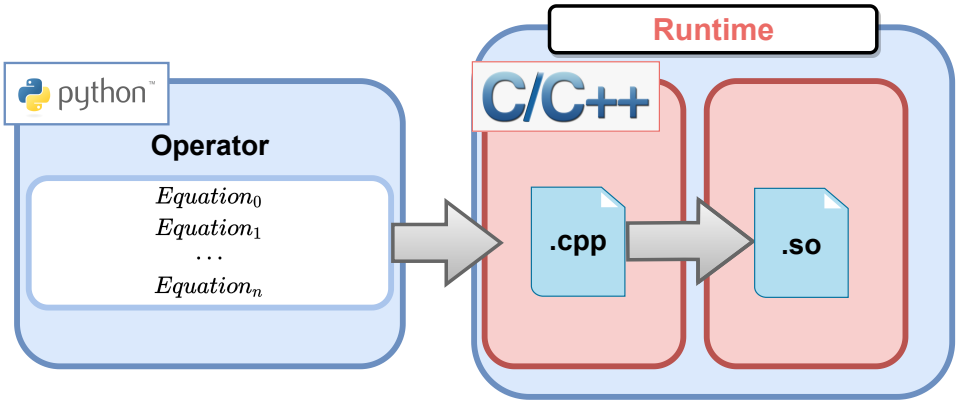


Figure 2.10.: Devito automatically generates C/C++ code from a high-level abstraction. Refer to Figure 4.1 for a detailed figure of the Devito compiler architecture.

Problem Specification Devito uses a symbolic mathematical language based on SymPy [Meurer et al., 2017] to allow concise expression of finite-difference stencil operations. Although the Devito DSL is used primarily to build PDE solvers, its rich DSL also describes other operations, such as tensor linear algebra operations, classic linear operators (e.g., convolutions), tensor contractions, boundary conditions, interpolations, etc.

Here we review some of the most prominent Devito API objects. In order to construct a simple finite-difference solver, some of the objects needed are `Grid`, `Function/TimeFunction`, `Eq` and `Operator`. More objects are available to support modelling in Devito DSL (see devitoproject.org). In all of the following examples, keywords of the language are shown in orange. For more details, the reader is referred to the Devito API Reference.

Grid The `Grid` object represents a discrete cartesian computational grid on which we can define parameters and equations. It is defined as a `Grid(shape)` object, where `shape` is the number of grid points in each spatial dimension. Over a `Grid` we can discretise a `Function` or/and a `TimeFunction`. A `Grid` encapsulates the topology and geometry information of the computational domain that a Function can be discretised on. It defines and provides the physical coordinate information of the logical cartesian grid underlying the discretised Functions.

Listing 2.4: Defining a two-dimensional Grid in Devito with 10 points per dimension.

```
>>> from devito import Grid
>>> grid = Grid(shape=(10, 10))
>>> grid
Grid[extent=(1.0, 1.0), shape=(10, 10), dimensions=(x, y)]
```

Function/TimeFunction A `Function` represents a discrete function in symbolic equations. This object carries multi-dimensional data and provides operations to create FD approximations. A `Function` is defined on an existing `Grid`. A Function encapsulates space-varying data; for data that also varies in time, we use `TimeFunction`. A `TimeFunction` represents a discrete function that is both spatially varying and time-dependent. Similarly to a `Function`, a `TimeFunction` is defined on an existing `Grid`.

Listing 2.5 shows how we can define a `Function` and a `TimeFunction` on a `Grid`. We can see how `TimeFunction` differs in its dimensions compared to a `Function` having `t` as an additional TimeDimension (object subclassing a Dimension) representing a time dimension.

Listing 2.5: Defining a Function and a TimeFunction over a Grid in Devito.

```
>>> from devito import Grid, TimeFunction
>>> grid = Grid(shape=(10, 10))
>>> u = Function(name='u', grid=grid)
>>> u
u(x, y)
>>> f = TimeFunction(name='f', grid=grid)
f(t, x, y)
```

Eq An `Eq` object represents a mathematical equation in symbolic notation. We can define an equal relation between two objects, the left-hand and right-hand sides. The Devito compiler digests this relation in the form of an assignment. In the following example, we write an equation `Eq` to update the values of `Function` by one.

Listing 2.6: An example of defining a mathematical equation (Eq) in Devito. All values of $f(x, y)$ are incremented by 1.

```
>>> from devito import Grid, Function, Eq
>>> grid = Grid(shape=(4, 4))
>>> f = Function(name='f', grid=grid)
>>> Eq(f, f + 1)
Eq(f(x, y), f(x, y) + 1)
```

Operator An `Operator` performs three fundamental tasks: compilation of high-level code to low-level code, just-in-time (JIT) compilation of the low-level code, and execution. An `Operator` accepts one or more symbolic equations as arguments. In the following example, Listing 2.7, we see how to construct and execute an `Operator` for the equation shown in Listing 2.6. We print the data held by f before and after executing the unary increment `Operator`.

Listing 2.7: Example of an Operator in Devito, where all values of $f(x, y)$ are incremented by 1. The data in Function f is printed before and after the generation and execution of the Operator.

```
>>> from devito import Grid, Function, Eq, Operator
>>> grid = Grid(shape=(2, 2))
>>> f = Function(name='f', grid=grid)
>>> eq = Eq(f, f + 1)
>>> f.data
Data([[0., 0.],
      [0., 0.],], dtype=float32)
>>> op = Operator([eq])
>>> op.apply()
>>> f.data
Data([[1., 1.],
      [1., 1.],], dtype=float32)
```

The `Operator` is working in the background to produce the automatically generated code. In Listing 2.8, we see a snippet of the code auto-

Listing 2.8: Part of the C code generated directly from Devito for the unary increment. The user has specified only the PDEs through the DSL. The code in this Listing is automatically generated.

```
1 for(int x = x_m; x <= x_M; x += 1)
2 {
3   for(int y = y_m; y <= y_M; y += 1)
4   {
5     f[x + 1][y + 1] = f[x + 1][y + 1] + 1;
6   }
7 }
```

matically generated from Devito for the unary increment `Operator`. The code in Listing 2.8 can be further optimised with parallelism and other optimisations.

So far, this subsection has illustrated how the user can work with the Devito API to model a simple problem. In the following examples, we will now look into how we can express the three problems presented in subsections 2.1.3.1, 2.1.3.2, 2.1.3.3 using the Devito DSL. Instead of deriving the stencil by hand, equation by equation, as shown in Section 2.1 we will only define the PDE in Devito DSL and let the framework do the rest of the work for us.

2.2.1.1. The Heat equation with Devito

Listing 2.9 shows the modelling of the heat equations 2.1.3.1 using the Devito DSL. We implement the heat equation 2.9 along with the boundary conditions 2.10, 2.11. Listing 2.9 introduces some additional parts of the Devito API. The `space_order` argument defines the discretisation order of the space derivatives. The `solve` function helps to algebraically solve an equation given the left and right-hand sides. The `stepping_dim` property returns the iterator used for time-buffering. The `apply` call tells the compiler to just-in-time compile the C-code and executes the computational kernel.

The `Operator` in Listing 2.9 automatically generates the stencil code for the heat equation in C language. A snippet of it is shown in Listing 2.10.

Listing 2.10 shows the time-iterative stencil update along with the applied boundary conditions.

Listing 2.9: Devito specification of the Heat equation as defined in Section 2.1.3.1.

```
from devito import Grid, TimeFunction, Eq, Operator, solve

# Some variable declarations
nx, nt = 10, 10
dx = 2. / (nx - 1)
sigma = .2
dt = sigma * dx

grid = Grid(shape=(nx, ), extent=(1.,))
u = TimeFunction(name='u', grid=grid, space_order=2)

eq = Eq(u.dt, u.dx2, grid=grid)
stencil = solve(eq, u.forward)

# Boundary conditions
t = grid.stepping_dim
bc_left = Eq(u[t + 1, 0], 0.)
bc_right = Eq(u[t + 1, nx-1], 0.)

op = Operator([Eq(u.forward, stencil), bc_left, bc_right])
op.apply(time_M=nt, dt=dt)
```

A note on time-buffering We notice that the time loop is successively interchanging the values of $t0$ and $t1$. In time-iterative stencils, keeping a copy of the grid in memory for every timestep is often useless. Time buffering is a standard implementation technique to reduce memory requirements. This is a generally adopted practice, keeping in memory only the necessary time slices needed to compute the next one. Devito automatically applies the time-buffering to time-iterative stencils.

In Listing 2.10, we need two copies of the grid to compute the problem. Assuming that $t0$ indicates the first slice to read from and $t1$ the first slice to write to, we keep iterating accesses over these grids in a fashion similar to write: t0, read: t1, write: t1, read: t0, write: t0, read: t1. This technique is so standard in stencil optimisations that over the last years, it is not even considered an optimisation. Figure 2.11 shows a schematic representation of the time buffering technique.

2.2.1.2. The Wave equation with Devito

Listing 2.11 shows the modelling of the Wave equation as described in Section 2.1.3.2. We implement the wave equation 2.18 along with the

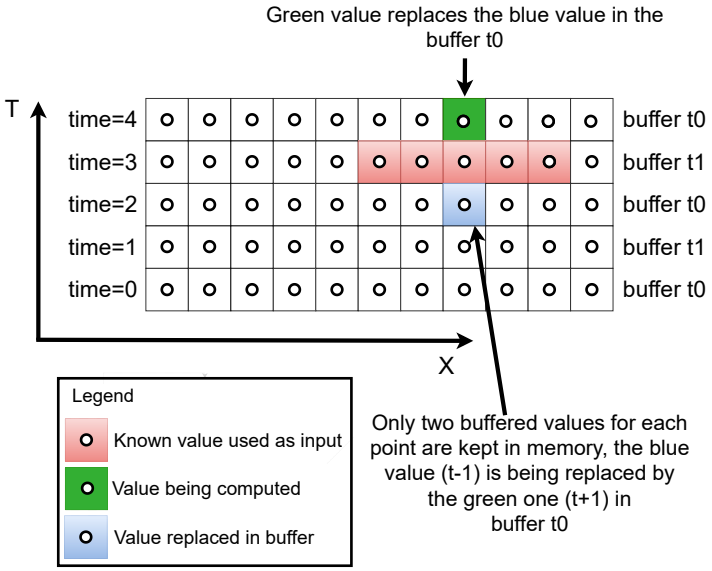


Figure 2.11.: The time buffering technique. Having a dependency of one point in time, only two copies of the grid are needed to compute the stencil updates.

Listing 2.10: Part of the C code generated directly from Devito for the heat equation. The stencil and the boundary condition updates are illustrated.

```
1  float r0 = 1.0F/dt;
2  float r1 = 1.0F/(h_x*h_x);
3
4  for (int time = time_m, t0 = (time)%(2), t1 = (time + 1)%(2); time
       <= time_M; time += 1, t0 = (time)%(2), t1 = (time + 1)%(2))
5    {
6      for (int x = x_m; x <= x_M; x += 1)
7      {
8        u[t1][x + 2] = dt*(r0*u[t0][x + 2] + r1*u[t0][x + 1] + r1
             *(-2.0F*u[t0][x + 2]) + r1*u[t0][x + 3]);
9      }
10     u[t1][2] = 0.0F;
11     u[t1][11] = 0.0F;
12   }
```

boundary conditions shown in Equations 2.19, 2.20 and 2.21.

The `Operator` in Listing 2.11 automatically generates the stencil code for the wave in C language. A snippet of it is shown in Listing 2.12. Listing 2.12 shows the time-iterative stencil update along with the applied boundary conditions.

Listing 2.11: Devito specification of the Wave equation as defined in Section 2.1.3.2.

```
# Some variable declarations
nx, nt = 10, 10
dx = 2. / (nx - 1)
sigma = .2
dt = sigma * dx

grid = Grid(shape=(nx, ), extent=(1.,))
u = TimeFunction(name='u', grid=grid, space_order=2)

eq = Eq(u.dt2, u.dx2, grid=grid)
stencil = solve(eq, u.forward)

# Boundary conditions
t = grid.stepping_dim
bc_left = Eq(u[t + 1, 0], 0.)
bc_right = Eq(u[t + 1, nx-1], 0.)

op = Operator([Eq(u.forward, stencil), bc_left, bc_right])
op.apply(time_M=nt, dt=dt)
```

Listing 2.12: Part of the C code generated directly from Devito for the wave equation. The stencil and the boundary condition updates are illustrated.

```
float r0 = 1.0F/(dt*dt);
float r1 = 1.0F/(h_x*h_x);

for (int time = time_m, t0 = (time)%(3), t1 = (time + 2)%(3), t2 =
     (time + 1)%(3); time <= time_M; time += 1, t0 = (time)%(3),
    t1 = (time + 2)%(3), t2 = (time + 1)%(3))
{
  for (int x = x_m; x <= x_M; x += 1)
  {
    float r2 = -2.0F*u[t0][x + 2];
    u[t2][x + 2] = (dt*dt)*(-r0*r2 - r0*u[t1][x + 2] + r1*r2 + r1*
        u[t0][x + 1] + r1*u[t0][x + 3]);
  }
  u[t2][2] = 0.0F;
  u[t2][11] = 0.0F;
}
```

2.2.1.3. The Laplace equation with Devito

Listing 2.13 shows the modelling of the Laplace equation (see 2.29) as shown in Section 2.1.3.3 and its implementation along with the boundary conditions in Equation 2.30. We use two Functions for pseudo-time-stepping for the one and only update we perform. Another difference

Listing 2.13: Devito specification of the Laplace equation as defined in Section 2.1.3.3.

```
from devito import Grid, Function, Eq, Operator, solve
# Some variable declarations
nx, ny = 10, 10
dx, dy = 2./(nx-1) , 2./(ny-1)

grid = Grid(shape=(nx, ny), extent=(1., 1.))
x, y = grid.dimensions

# Create two explicit buffers for pseudo-time-stepping
p = Function(name='p', grid=grid, space_order=2)
pn = Function(name='pn', grid=grid, space_order=2)

# Create the Laplace equation based on `pn`
eqn = Eq(pn.laplace, subdomain=grid.interior)
# Let SymPy solve for the central stencil point
stencil = solve(eqn, pn)
# Now, we let our stencil populate our second buffer `p`
eq_stencil = Eq(p, stencil)

# Boundary conditions
bc_left = Eq(p[0, y], 0.)
bc_right = Eq(p[nx, y], 0.)

bc_top = Eq(p[x, 0], 0.)
bc_bottom = Eq(p[x, ny], x)

op = Operator([eq_stencil, bc_left, bc_right, bc_top, bc_bottom])
op.apply()
```

compared to the example in Section 2.1.3.3 is that we use x in place of $f(x)$ for the bottom boundary condition. The `Operator` in Listing 2.13 generates the stencil code shown in Listing 2.14. Listing 2.14 shows the stencil update along with the applied boundary conditions.

Abstracting away the mathematics from the C code is valuable for automating our work. This section illustrated some typical stencil examples using the Devito DSL. We only looked at the first and last block of Figure 2.12. The middle block, which contains all the compiler machinery and the compiler architecture, will be presented later in this thesis in Chapter 4. We will see how the arithmetic operations are optimised and more details on how the Devito compiler automatically generates optimised stencil code starting from the high-level abstraction.

Listing 2.14: Example of the C code generated directly from Devito for the Laplace Equation. The user has specified only the PDEs through the DSL and automatically gets the code shown in this Figure.

```
1  float r0 = 1.0F/(h_x*h_x + h_y*h_y);
2  for (int x = x_m; x <= x_M; x += 1)
3  {
4    for (int y = y_m; y <= y_M; y += 1)
5    {
6      p[x + 2][y + 2] = 5.0e-1F*r0*((h_x*h_x)*(pn[x + 2][y + 1] + pn
           [x + 2][y + 3]) + (h_y*h_y)*(pn[x + 1][y + 2] + pn[x + 3][
           y + 2]));
7    }
8  }
9
10 for (int y = y_m; y <= y_M; y += 1)
11 {
12   p[2][y + 2] = 0.0F;
13   p[12][y + 2] = 0.0F;
14 }
15
16 for (int x = x_m; x <= x_M; x += 1)
17 {
18   p[x + 2][2] = 0.0F;
19   p[x + 2][12] = x;
20 }
```

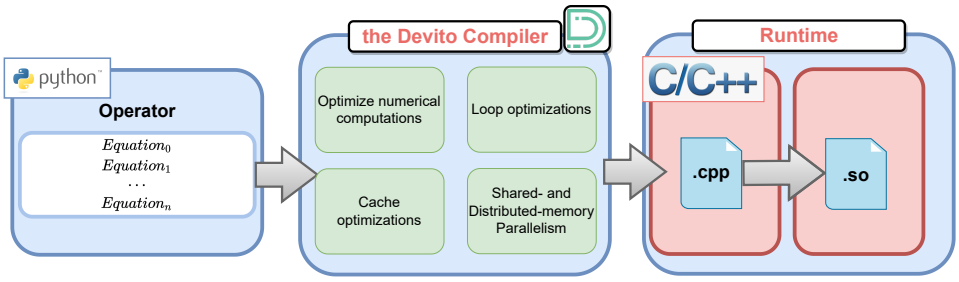


Figure 2.12.: Devito gets user input in its symbolic API, the compiler is doing all the hard work, the user again enjoys high-performing automatically generated C/C++ code from a high-level abstraction. For more details on the Devito compiler architecture, the reader should refer to Figure 4.1.

2.2.2. Finite Difference frameworks

Apart from Devito there are other notable frameworks to facilitate the abstraction of the FD method. Most of these frameworks have a lower-level API compared to Devito. As lower-level APIs, we describe APIs that use notation that resembles closer to the final generated C code. This subsection refers to these frameworks by focusing on their modelling abstraction. Later in this Chapter, we dive into their supported automated

optimisations.

The Oxford Parallel library for Structured mesh solvers (OPS) [Reguly et al., 2018] is a domain-specific language offering a C/C++/Fortran domain-specific API. The OPS API is not starting from the PDEs but rather from a lower-level language stencil specification using C language. OPS, similarly to other DSLs, separates high-level code from low-level implementation. This separation of concerns helps domain scientists write high-level code and automatically benefit from low-level optimisations.

OpenSBLI [Jacobs et al., 2017] is another high-level Python-based DSL to model differential equations. Domain scientists can express differential equations in Einstein notation. OpenSBLI may solve any set of equations written in Einstein notation. OpenSBLI mainly focuses on the compressible Navier-Stokes equations with application to shock-boundary layer interactions (SBLI) but is not limited to this. OpenSBLI translates the problem specification to C code with the help of the OPS library [Reguly et al., 2018].

Exastencils [Lengauer et al., 2014] is a code generation framework that processes input in its multi-layered domain-specific language (DSL). This DSL, named ExaSlang, helps to compose highly optimised and massively parallel geometric multigrid solvers on (block-)structured grids. ExaSlang offers a Latex-like syntax to express the continuous formulation of the problem and to discretise using Finite Differences (FD), Finite Volume (FV) or Finite Elements (FE), depending on the domain.

Exastencils can support several backends based on MPI, OpenMP, CUDA and combinations thereof, targetting CPUs, GPUs and hybrid architectures.

STELLA [Gysi et al., 2015] is a domain-specific language targetting weather applications. STELLA is a production-grade language that describes PDE operators with a concise syntax similar to the discretised mathematical description. The DSL increases readability by hiding the complexity of loops and hardware-dependent optimisations. STELLA helped port the COSMO model to NVIDIA GPUs, providing performance portability while retaining a single source code.

The GridTools library [Afanasyev et al., 2021] started as an effort to generalise Stella. Gridtools consists of a set of libraries and utilities to develop performance-portable weather, climate (and other stencil-like) applications [Thaler et al., 2019]. The core component of GridTools is the

stencil composition module that implements a C++-embedded DSL for stencils and stencil-like patterns. The user code is written in a higher-level form which is then lowered and optimised for a given architecture at compile time, achieving performance portability. The GridTools framework is used with great success to accelerate the dynamic core of the COSMO model. It demonstrates the library's production quality and feature completeness for models on latitude-longitude grids. GridTools offers an API to support the definition of boundary conditions.

Simflowny [Palenzuela et al., 2021] is an open platform that automatically generates efficient parallel code for scientific dynamical models for different simulation frameworks. Simflowny differentiates from the other frameworks by offering a graphical user interface combined with a semantic DSL based on XML to automatically generate code for several PDEs. Simflowny offers automatic code generation. It offers PDE abstraction for Navier-Stokes and Einstein equations, can discretise them using the finite-difference method and automatically generates parallel code. However, it is not considered a compiler-based framework such as the previously mentioned ones.

2.2.3. Other frameworks for automated code-generation

Partial differential equations may also be solved using the finite element method (FEM) [Zienkiewicz et al., 1977], the finite volume method (FVM) [Eymard et al., 2000], the spectral element method (SEM) [Patera, 1984], flux reconstruction methods [Huynh, 2007] or others.

Two of the most prominent frameworks that automate the finite element method are FEniCS [Logg et al., 2012] and Firedrake [Rathgeber et al., 2017]. FEniCS enables users to express scientific models at a high-level abstraction and automatically translate them to high-performing finite element code. FEniCs specifies weak variational forms via the Unified Form Language (UFL) [Alnæs et al., 2014]. UFL is an embedded domain-specific language for defining variational forms for finite element discretisation. More precisely, it defines a flexible interface for choosing finite element spaces and defining expressions for weak forms in a notation close to mathematical notation [Alnæs et al., 2014, Alnæs et al., 2016]. FEniCS offers high-level Python and C++ interfaces and runs on platforms ranging from laptops to

high-performance clusters. Similarly to FEniCS, Firedrake [Rathgeber et al., 2017] automates the solution of partial differential equations through the finite element method (FEM). Firedrake uses UFL from the FEniCS project to express PDEs and uses sophisticated code generation to provide mathematicians, scientists, and engineers with enhanced productivity towards sophisticated and high-performing simulations. The assembled operators are using the *P*ortable, Extensible Toolkit for Scientific Computation library (PETSc library) [Balay et al., 2015]. PETSc is entirely implemented in C; however, it offers petsc4py [Firedrake, 2016] to interfere with Python frameworks (e.g. Firedrake).

PETSc is a notable open-source project, a parallel linear and non-linear solver framework with a collection of data structures and routines for scalable solutions for scientific applications modelled by partial differential equations. Many of its functionalities are built on top of existing libraries (e.g., BLAS). PETSc libraries aim to accelerate large-scale application projects.

Firedrake uses PyOP2 [Rathgeber et al., 2016] to apply computational kernels over the discretised equation domain. PyOP2 is a DSL embedded in Python and relies on just-in-time (JIT) compilation and execution. Among others, it supports the parallel application of kernels over unstructured meshes, an essential requirement for the finite element method. It also provides global data structures, such as sparse matrices, essential for solving linear systems.

OP2 [Mudalige et al., 2012] is a high-level embedded domain-specific language for writing unstructured mesh algorithms with automatic parallelisation on multi-core and many-core architectures. OP2 uses source-to-source translation and compilation to transform code written in the OP2 API to target different backend platforms. The main target is the performance portability of CFD applications for CPUs, GPUs, and clusters thereof.

Other tools help the solving PDEs in various ways, e.g. by providing their own DSL, like FreeFEM [Hecht, 2012] or for example by helping to model physics for 2D or 3D problems while taking advantage of an extensive list of finite elements usable in the continuous and discontinuous Galerkin method framework. MFEM [Anderson et al., 2021] is a FEM toolbox providing building blocks to facilitate the development of FEM

algorithms. It has a lower-level API written in C++ to help model the problem. Another one is HighPerMeshes [Alhaddad et al., 2020], a domain-specific programming and target-platform-aware compiler infrastructure for algorithms on unstructured grids and Liszt [DeVito et al., 2011], a high-level language interface to construct mesh-based solvers.

An example of a framework to solve PDEs using the Flux Reconstruction method is PyFR [Witherden et al., 2014]. PyFR is an open-source framework written in Python, offers a template-like DSL and can successfully target clusters of CPUs and GPUs.

2.3. Cache blocking optimisations in numerical kernels

In the previous Section 2.2.2, we presented frameworks that automate several computational methods. Quite a few of these computational methods heavily depend upon cache-related optimisations. This section provides the necessary background on cache-related optimisations. This section will pave the path towards Section 2.4, where we will present frameworks that automate cache-blocking optimisations.

Over the last few decades, researchers have put considerable effort into cache-friendly algorithms to improve stencil performance. In most stencil applications of interest, kernels have low operational intensity and are memory bandwidth bound. The reason is that they have few floating-point operations per byte of data accessed (aka low arithmetic intensity (see Section 2.5)). This section presents the related work on cache-blocking optimisations. Blocking data accesses to organise better how data fits in the cache is a well-known algorithmic approach to enhance performance. Blocking optimisations reschedule the order of computations towards increased cached memory reuse. The throughput is increased as memory bandwidth boundedness issues are alleviated. The overarching aim is to fill the cache with a data set subset and then work to the highest degree possible with this data set loaded in the cache memory. The more our program works with the data loaded in the cache, the less often the need to fetch data from the main memory arises. As a result, the memory bandwidth pressure is reduced.

To appreciate the importance of cache optimisations, one must be familiar with the terms “locality of reference”, “spatial”, and “temporal” locality.

Locality of reference The term “Locality of reference”, also known as the principle of locality, describes a computer program’s tendency to perform similar memory accesses for a particular time period. Locality is a commonly expected behaviour in computer systems. When a system exhibits a good locality of reference, cache optimisations are strong candidates for performance optimisation. There are two main types of reference locality, spatial and temporal locality.

Temporal Locality Temporal locality refers to current data fetched and may be needed again soon. A reasonable thing to do is store that data fetched in the cache memory to bypass searching in the main memory for the same data again. When the CPU accesses the current main memory location for fetching required data or instructions, this also gets stored in the cache. It is stored in the cache because we work under the hypothesis that the same data or instruction may be needed again shortly. This kind of locality is known as temporal locality.

Spatial Locality Spatial locality refers to accessing data elements within relatively close storage locations. Data that is near the current fetched memory location may be soon needed in the future. When referring to spatial locality, we are talking about nearly located memory locations, while in the temporal locality, we are talking about the actual memory location being fetched.

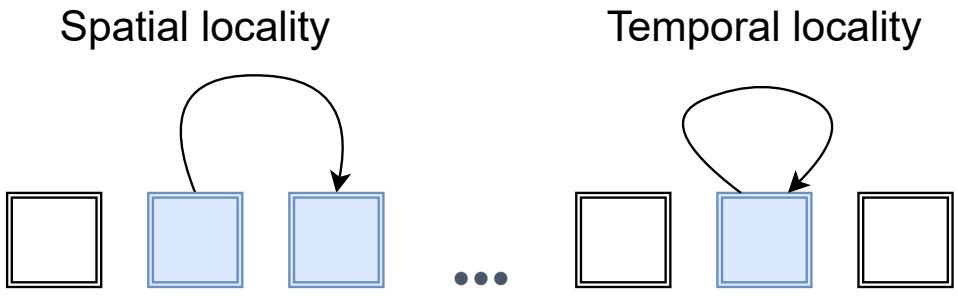


Figure 2.13.: In spatial locality, items close in memory tend to be referenced together. In temporal locality, a referenced item may be referenced again shortly.

The following subsection presents the necessary background and related work for cache blocking to enhance spatial and temporal locality and other optimisations.

Note: It is important to note that throughout the following Sections, we refer to loop blocking, standard loop blocking or spatial blocking as the blocking that happens only along dimensions that represent and describe space in stencil computations. For example, x, y, z in a 3D cartesian system. Every reference to temporal loop blocking includes schemes where the time loop of time-iterative simulations is blocked. In the context of time-iterative stencils, loop blocking offers enhanced spatial and temporal reuse over the "flat" iteration. As the stencil update sweeps over the iteration space, the read-write stencil accesses overlap. Consequently, we exploit both spatial and temporal reuse. Both spatial and temporal blocking enhance temporal locality.

2.3.1. Loop blocking

Loop Blocking, also known as *loop tiling*, is one of the most studied and influential transformations in the category of cache optimisations [Unat et al., 2017]. The fundamental idea is to partition the iteration space of a loop nest into blocks of a predefined shape.

Loop tiling is primarily used to enhance data locality. In addition, it is often combined with parallelisation. This iteration space partitioning required additional loops in the code structure, as shown in Listing 2.15. In Listing 2.15, square-shaped tiles of size $b \times b$ are employed to increase data reuse along the dimensions iterated by N and M.

Loop tiling was presented in the early studies of Wolfe [1989], Wolf and Lam [1991], Ramanujam and Sadayappan [1991], Ramanujam and Sadayappan [1992]. As loop tiling often requires considerable effort to write additional loops and control their bounds, more recent works have tackled the automation challenge. In addition, several works have presented methods to improve tiling algorithms, such as Bondhugula et al. [2008b], Acharya and Bondhugula [2015], Klöckner [2015]. These tools, often utilising the polyhedral model, will be discussed in more detail in Section 2.4. Tile shape can significantly impact the tiling performance, as it depends on the architecture of the underlying cache. Krishnamoorthy

Listing 2.15: Illustration of classical rectangular loop tiling in a classic Jacobi-like update kernel similar to 2.3. The matrices are square-shaped of size $N \times N$. Data reuse can be achieved if b is chosen small enough to fit some cache level.

```
1  // Original implementation
2
3  for k = 1 to nt, 1
4    for n = 1 to N, 1
5      for m = 1 to M, 1
6        u[k+1][n][m] = 0.25 (u[k][n+1][m] + u[k][n][m+1] + u[k][n
             -1][m] + u[k][n][m-1]);
7
8
9  // Loop blocking implementation
10
11 for k = 1 to nt, 1
12   for n0 = 1 to N, b
13       for m0 = 1 to M, b
14           for n = n0 to min(n0+b-1, N)
15               for m = m0 to min(m0+b-1, M)
16                   u[k+1][n][m] = 0.25 (u[k][n+1][m] + u[k][n][m+1]
                         + u[k][n-1][m] + u[k][n][m-1]);
```

et al. [2007] and Grosser et al. [2014b] have investigated, among others, the impact of tile shape.

Tiling is very popular, and it is an essential performance optimisation in the context of stencils. However, the effort needed to implement it is non-trivial. The iteration space of visiting the finite-difference grids is decomposed into tiles, and computations benefit from cache blocking as illustrated in Wolfe [1989], Wolf and Lam [1991], Ramanujam and Sadayappan [1992], Xue [1997], Frigo and Strumpen [2005, 2006], Datta et al. [2008], Tang et al. [2011], Frigo et al. [2012]. In a more general context regarding time-iterative simulations, the non-time iteration space blocks can be executed in a parallel fashion within each time iteration. Figure 2.14 shows a symbolic example of this partitioning along with block execution order.

The same idea applies to unstructured grids, though this requires more sophisticated algorithms to create efficient schedules [Luporini et al., 2019]. Loop blocking optimisations have been extended to GPUs. Related work includes several schemes such as automated split tiling with trapezoids [Grosser et al., 2013a], hybrid hexagonal tiling [Grosser et al., 2014a] and automated HPC GPU code [Schäfer and Fey, 2011, Holewinski et al., 2012,

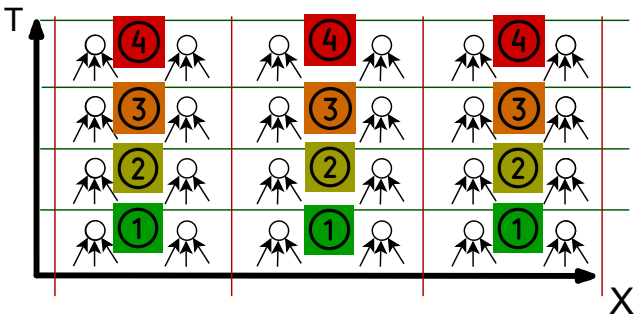


Figure 2.14.: In standard, time-iterative space loop blocking partitions are executed in parallel for every timestep.

Listing 2.16: Illustration of loop interchange in a simple mathematical computation.

```
1  // Original implementation
2
3  for m = 1 to M, 1
4    for n = 1 to N, 1
5      a[n][m] = b[n][m] + c[n][m];
6
7  // Loop interchange implementation facilitating row-by-row
       accesses for better vectorisation of the innermost loop
8
9  for n = 1 to N, 1
10   for m = 1 to M, 1
11     a[n][m] = b[n][m] + c[n][m];
```

Rawat et al., 2018].

2.3.2. Loop reordering transformations

Loop reordering transformations mainly focus on techniques for improving data locality in loop nests. Characteristic examples of loop reordering transformations are:

Interchange This transformation [Allen and Kennedy, 1984] consists of exchanging the position of two loops in a nest. Possible objectives include exposing a vectorisable dimension or increasing the temporal and spatial locality in a loop. A simple example is shown in Listing 2.16.

Skewing Loop skewing, aims to improve data reuse and expose parallelism in wavefront computations. In these computations, one or more

arrays are updated at every loop iteration, and the updates propagate in a wave fashion over the subsequent iterations (e.g., `A[i][j] = f(A[i-1][j], A[i][j-1])`). This wave-like propagation occurs due to diagonal dependencies. Skewing leads to non-rectangular tiles, as shown in Figure 2.15. We expand on loop skewing and wavefront temporal blocking in Chapter 3.

Figure 2.15 shows the order of computing mesh points in space-time partitions. Parallelism can also be applied among points within a group of execution and with the same time index. Parallelism opportunities are not shown in Figure 2.15.

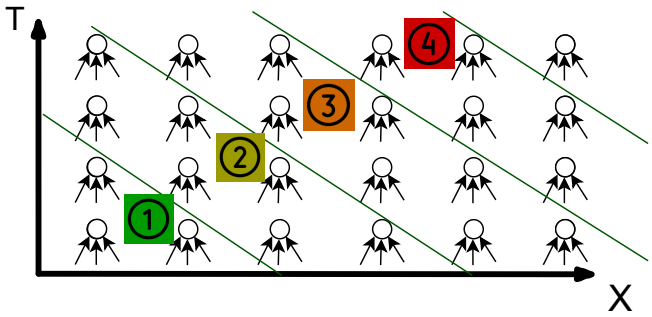


Figure 2.15.: Traditional loop skewing. Parallelism can also be applied among points within a group of execution and with the same time index.

2.3.3. Temporal loop blocking

This subsection aims to introduce temporal loop blocking to the reader. Earlier in this section, we discussed temporal locality (see Section 2.3). Later, Section 3.1.4.1 will present its historical evolution and focus on discussing work related to its application in stencils.

The central idea behind temporal blocking is to further enhance temporal locality compared to classical loop blocking by reducing cache misses. Some simulations often have another dimension, time, and the extension of loop blocking in the time dimension seems natural. For this purpose, several time-blocking methods have been developed to take advantage of cache reuse in an additional dimension. Temporal blocking often consists of loop reordering transformation when applied to already loop blocked code.

This section briefly introduces temporal blocking as a loop transformation. In Section 3.1.4.1, we will focus on prior attempts to apply temporal blocking to stencils and in Section 4.2 we will discuss prior attempts at automating this procedure.

In iterative simulations, data dependencies span different iterations. Examples in Section 2.1.3 show iterative stencils where data dependencies span different time steps. Rectangular tiles are not enough. Tiling in the time dimension is more complex than in space dimensions, mostly due to these dependencies. Iteration space partitioning cannot happen similarly to spatial blocking, as data dependencies will be violated. These dependencies complicate the application of temporal blocking, so the block shapes in temporal blocking are often irregular and multidimensional, only sometimes straightforward to visualise and comprehend. In order to preserve the validity of execution for temporal blocking, several methods of iteration space partitioning have been devised and will be discussed in more detail in Section 2.3.3.1.

We utilise computed values in a block to update grid point values at the next timestep where possible. While one or more timesteps for a given block's values are stored in the cache, we start computing the next timestep for this block, not depending on the requirement to compute all the blocks of a given grid for previous timesteps. Often, spatial and temporal reuse are combined into hybrid models to harness the advantages of both methods [Zohouri et al., 2018]. Plenty of research has been conducted in designing and evaluating temporal blocking schemes. Some of these schemes are presented in Section 2.3.3.1. There is a notable amount of work spanning cache-optimising approaches without using loop blocking in time dimension schemes such as Wolfe [1986] to several temporal blocking schemes in Wonnacott [2000], Guohua Jin et al. [2001], Wonnacott [2004], Wellein et al. [2009], Strzodka et al. [2011] and wavefront [Yount and Duran, 2016], and then, to more sophisticated such as diamond Bertolacci et al. [2015], Bandishti et al. [2012], Malas et al. [2015], Muranushi and Makino [2015], Levchenko and Perepelkina [2017], Akbudak et al. [2020] and Wang and Chandramowlishwaran [2020]. Temporal blocking has historically been applied to several stencil codes; however, it may not be a good fit for real-world problem loop structures. Examples of applying temporal blocking to real-world applications are shown in Chapter 3. Regarding performance

gains, while narrow (low space discretisation order) stencil kernels exhibit good temporal locality, temporal blocking profits decrease with wider stencils. In problems with high discretisation order, the temporal locality is limited. While narrow stencil kernels exhibit good temporal locality, temporal blocking gains decrease when space-order increases. Higher space order problems limit temporal locality as wide update dependencies in space require more grid points to be known in order to update one value in the time dimension.

Reuse level for medium space discretisation order stencils		
	Spatial locality	Temporal locality
Flat	Low	Low
Loop blocking	High	Medium
Temporal Loop blocking	High	High

Table 2.1.: Overview of a "conceptual" degree of data reuse level for spatial and temporal locality

2.3.3.1. Temporal loop blocking schemes

This section presents a short overview of some temporal tiling schemes. Several schemes have been implemented concerning temporal blocking. Most of these schemes are named after the shape of tiles that they end up computing. These schemes vary in the degree of parallelism and synchronisation, their efficiency on lower or higher space discretisation order and their ratio of computation and communication overlap. Some of these schemes are briefly described here.

Overlapped tiling Different tiles share subsets of iterations. A shared iteration is "owned" by only one tile and is executed redundantly by the set of overlapping tiles. These tiles store intermediate values in "ghost" regions of memory. This approach removes the requirement for a partial execution ordering at the price of redundant computation.

Wavefront temporal blocking Wavefront temporal blocking (WTB) extends standard loop blocking with loop skewing and interchange to exploit

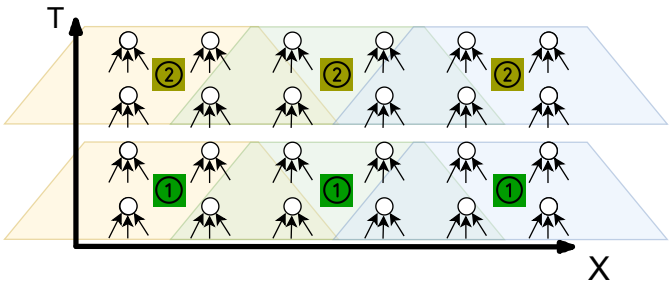


Figure 2.16.: Overlapped temporal blocking. Temporal locality is enhanced but with the trade-off of redundant computation.

locality in the time dimension. The grid points are updated in a diagonal space-time plane, aiming to increase the number of cache hits compared to standard loop blocking.

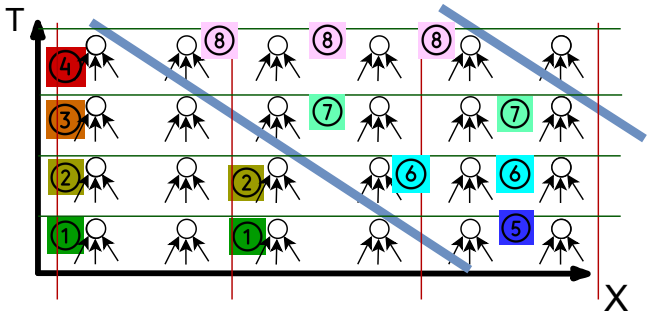


Figure 2.17.: Wavefront temporal blocking. Traditional loop skewing (Figure 2.15) is enhanced with space and time blocking for improved spatial and temporal locality.

In this thesis, wavefront temporal blocking is an essential background for the contributions presented in Chapter 3.1.1. We provide more details on the update patterns in wavefront temporal blocking to get more familiar with the space-time updates of temporal blocking. Space-time wavefronts traverse our domain in wavefront temporal blocking, diagonally updating grid point values. As illustrated earlier in Figure 2.17, these diagonals describe the order of grid point updates. For naming convention, as used in Yount and Duran [2016], we will use the term “block” for spatial-only grouping and “tile” when multiple temporal updates are allowed. Figure 2.18 shows grid point updates in wavefront temporal blocking.

The green point is updated using the values from the red points.

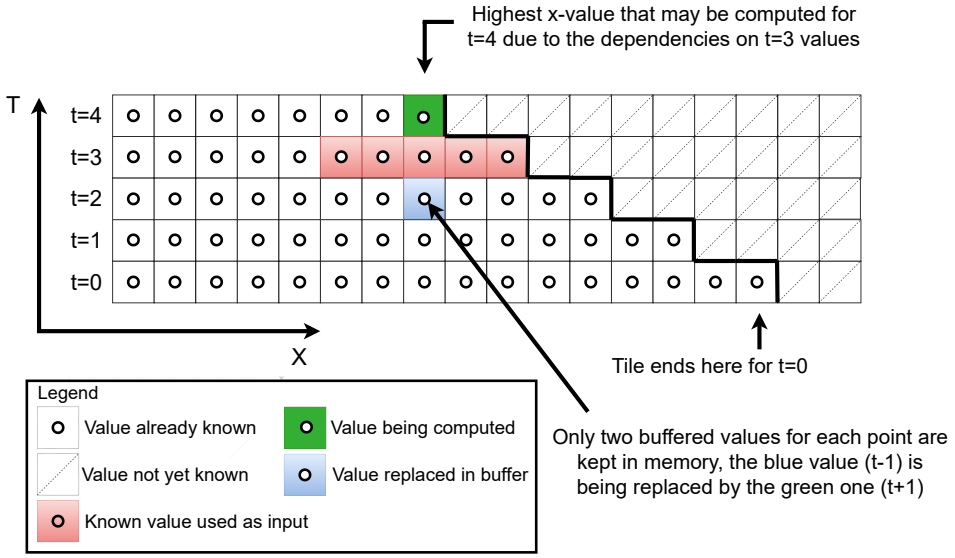


Figure 2.18.: Illustration of stencil kernel update in wavefront temporal blocking. The green point is updated using the red values. This stencil kernel has a space order of 4, thus allowing a margin of 2 points on the right not to violate data dependencies. Only two timesteps are kept in memory, as the stencil depends on values of the previous timestep only, so the green value substitutes the blue one. Figure partially adapted from Yount and Duran [2016].

This stencil has a radius of size two. Thus a margin of two points is required to preserve data dependencies. Only two timesteps are stored in memory, as the stencil depends on values of the previous timestep only, so the green value substitutes the yellow one in the buffer. Keeping two timesteps in memory is a standard technique to minimize the memory requirements (see Paragraph 2.2.1.1).

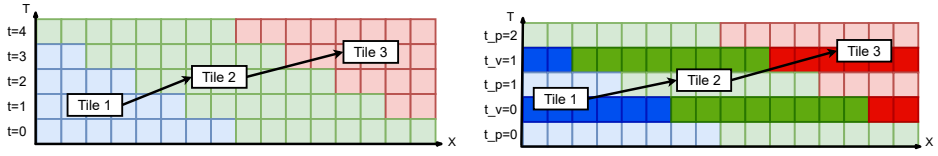
The stencil radius affects the wavefront angle (i.e. the ratio of spatial indices needed to update one point in the next time step). The angle gets steeper with a higher stencil radius.

WTB can also be applied to staggered grids as illustrated in Figure 2.19b. In a typical example, in this case, we have more than one grids that are updated with stencils that read data from another grid. Assuming we have a grid p and a grid v, we may need to compute both p and v as a function of p and v at the previous timestep. A simple pseudocode example for such a case is shown in 1. A real-world example that ends up in similar stencils is presented later in Section 3.3.3.

ALGORITHM 1: A typical time-stepping loop nest structure where multiple fields are being updated, reading from one another and their selves

```
1 for t = 1 to nt do
2   for x = 1 to nx do
3     for y = 1 to ny do
4       for z = 1 to nz do
5         p[t, x, y, z] = p[t-1, x, y, z] + v[t-1, x, y, z] + Σ_{r=1}^{r=so/2} w_r ( v[t-1, x - r, y, z] + v[t-1,
          x + r, y, z] + v[t-1, x, y - r, z] + v[t-1, x, y + r, z] + v[t-1, x, y, z - r] + v[t-1, x, y, z +
          r] );
6   for x = 1 to nx do
7     for y = 1 to ny do
8       for z = 1 to nz do
9         v[t, x, y, z] = v[t-1, x, y, z] + p[t-1, x, y, z] Σ_{r=1}^{r=so/2} w_r ( p[t-1, x - r, y, z] + p[t-1, x +
          r, y, z] + p[t-1, x, y - r, z] + p[t-1, x, y + r, z] + p[t-1, x, y, z - r] + p[t-1, x, y, z + r] );
```

In this case, two or more grids may be updated, often having inter-dependencies [Yount and Duran, 2016]. It is then necessary to shift the wavefront angle (allow more margin to preserve dependencies) by an amount, depending on the stencil radius of data dependencies in each loop, as shown in Figure 2.19b compared to Figure 2.19a. Figure 2.19b shows an example where we have two fields, namely t_p and t_v, who read from one another and their self using a stencil of the same radius. As a result, an update in t_p needs two times the stencil radius to be updated.



(a) The figure shows multiple wavefront tiles evaluated sequentially, partially adapted from Yount and Duran [2016].

(b) The figure shows multiple wavefront tiles evaluated sequentially in stencil codes with multiple fields.

Figure 2.19.: Wavefront updates for single- and multi-field stencil updates.

Split (Diamond/Hexagonal) tiling We assemble tiles to satisfy all data dependencies and execute them in a partial order. Many split tiling schemes have been studied for structured stencil codes. These schemes are usually named after the tile shape they end up producing (such as **hexagonal** or **diamond** tiling). They mainly differ in the achieved

trade-off between parallelism and data locality. We can visualise a tile's geometrical shape by plotting the fused iteration space and grouping iterations according to the tile they belong to. Diamond tiling is mainly regarded as an advanced temporal blocking technique. It is considered a split tiling technique. The tile slope depends on the stencil's semi-bandwidth. Diamond tiling has several advantages [Malas et al., 2015, Bertolacci et al., 2015]. First, it allows maximum data reuse. Secondly, it allows better concurrency in updating tiles and increased parallelism.

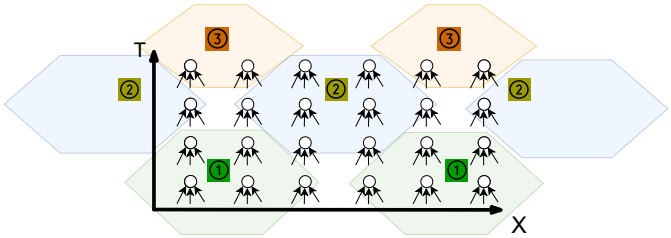


Figure 2.20.: Hexagonal/Diamond loop tiling. Grid points are grouped in space-time hexagons.

Cache-oblivious tiling In cache-oblivious tiling, we aim to benefit from cache memory without explicitly knowing the size of the cache. Cache-oblivious tiling aims to perform well on different-sized caches and with different memory hierarchies. Cache-oblivious tiling may also be substituted with block-size autotuning by paying the price of the additional experiments needed to find the optimal block shape combination.

Hybrid tiling Hybrid tiling does not take a particular shape or form of tiling. It can take several forms by combining different tiling variants. Usually, it combines several forms of space and time tiling or different shapes along space dimensions. For example, it may use diamond tiling along x and rectangular tiling along y, or alternatively, hierarchical loop tiling and time skewing.

Stencil-shape aware tiling schemes Often blocking optimisation for stencils does not distinguish between stencils with different shapes. There are techniques aiming to optimise performance concerning the stencil's shape. Stencil tesselation [Yuan et al., 2019] is new cache-related

optimisation that handles box and star stencils differently.

Some of these strategies have been reviewed in Grosser et al. [2013b]. We already noted in Chapter 1 that this thesis aims to deliver techniques to facilitate the automated application of wavefront temporal blocking. Nevertheless, the theory of this approach aims to include every other temporal blocking scheme (i.e. the advanced ones) like split, overlapped and diamond.

2.3.4. Other loop optimisations

Apart from tiling-related optimisations, there are also other loop optimisations that aim to enhance code performance. Most of these optimisations are supported by modern general compilers and compiler frameworks (like Devito). Such loop optimisations include:

Fission/Distribution Sometimes called loop distribution, loop fission breaks a loop into a sequence of multiple loops. The new loops have the same iteration space as the original but include only a subset of statements. The aim is to break down a possibly large loop body into smaller "chunks" to achieve better data locality.

Fusion A sequence of loops can be fused or "merged" to improve data locality and reduce the loop overhead. In the most straightforward instance, all loops in a sequence have the same iteration space. Given S_1 a statement in a first loop and S_2 a statement in a subsequence loop, S_2 does not modify any data read by S_1. Applying loop fusion is straightforward in such a case since no data dependencies are violated. In general, loops can have different bounds, and the data dependencies in (a set of) statements may be non-trivial. In these cases, loop fusion becomes more challenging or simply not feasible. This challenge has been formally tackled by Darte [2000]. Loop fusion is only sometimes beneficial and can lead to slower execution times or increased memory usage. It is essential to carefully analyse the program and evaluate the potential benefits of loop fusion before applying it. Additionally, loop fusion can make code more complex and harder to read and maintain, so it should be used judiciously.

Loop invariant code motion Often loops compute expressions whose result is the same for every loop iteration (i.e., a loop-invariant quantity). To improve efficiency, we can move this computation from the inside to the outside of the loop. Instead of computing the expression for every loop iteration, we only compute it once.

(Auto-) Parallelization Often loops are restructured in order to run efficiently on multiprocessor systems. This may be accomplished either automatically, through compilers with automatic parallelisation or manually by inserting parallel directives (e.g. OpenMP pragmas).

(Auto-) vectorisation Similar to (Auto-)Parallelization, but instead, loop iterations are scheduled to run efficiently on a SIMD system. They are often used in tandem with (Auto-) Parallelisation. Vectorisation is a well-known paradigm that generalises scalar computation to vector computation. That is, arrays of contiguous elements. A single instruction, multiple data (SIMD) computation employs vectorisation to carry out a sequence of instructions. SIMD architectures, ubiquitous nowadays, emit vector code in two circumstances. The first case is when program sections are explicitly vectorised (e.g., through high-level libraries, intrinsic instructions, or assembly code);. The second case is when a compiler transforms scalar code into vector code. The latter is called auto-vectorisation since SIMD instructions are generated without user intervention. Auto-vectorisation should be preferred over explicit vectorisation for portability reasons when possible and if demonstrated to be effective. Autovectorisation is typically applied to inner loops, although more advanced compilers (e.g., Intel's) also support block vectorisation [Larsen and Amarasinghe, 2000].

Unrolling Loop unrolling (or unwinding) increases program efficiency by reducing or removing loop control and test instructions. This technique aims to decrease the number of times the loop condition is evaluated and the number of jumps, which may degrade performance by impairing the instruction pipeline. Completely unrolling a loop eliminates all overhead but requires that the number of iterations is known at compile time. We should ensure that multiple

re-calculation of indexed variables has a manageable overhead than advancing pointers within the original loop.

2.4. Compilers and Libraries for Automated Optimisations

In this section, we review state-of-the-art compilers and libraries for loop optimisation. Some of the frameworks already mentioned in Section 2.2 also support automated optimisations through built-in compiler infrastructure. However, most frameworks today for automating loop optimisations do not have a high-level symbolic mathematical language. Most of these compilers are based on the polyhedral model [Bastoul, 2004, Verdoolaege, 2010] to express an Intermediate Representation (IR). Other compilers, frameworks or source-to-source translators are based on analysing the source code, applying rewrite rules, and producing another IR. An IR is a graph-like data structure (e.g., the static single assignment form, or SSA, in the LLVM compiler) used internally by a compiler to represent source code. This review discusses tools that fall into the categories mentioned here and have influenced our work. This section provides the necessary context for the contributions presented in Chapters 3 and 4.

The polyhedral model The polyhedral model provides an abstraction for performing high-level transformations such as loop-nest optimisation and parallelisation on affine loop nests [Griebl et al., 1998]. In the context of the polyhedral model, loop nests are represented as *polyhedra*. Every loop iteration within nested loops as lattice points inside polyhedra. In polyhedral representations, loop bounds and array indices must be affine expressions in the enclosing loop indices, and polyhedral transformations are a combination of several one-dimensional affine transformations. An affine transformation can be explained as transforming a convex polyhedron into another convex polyhedron. The compilers based on the polyhedral model (henceforth, polyhedral compilers) target parallelism and data locality by composing *affine scheduling functions*. Once the polyhedron is available, different scheduling functions can be compared and applied. A schedule defines the order in which the statement instances of a

loop nest are executed; a scheduling function can change the original order. This work focuses on structured stencil codes with sparse, non-aligned to the structured grid points operators. These operators render the loop nests non-affine, precluding polyhedral compilers' adoption. There have been efforts to extend the polyhedral model to non-affine loops [Venkat et al., 2014a]. However, the applicability to real-world applications has yet to be established. In Chapter 3, we discuss why we cannot use a polyhedral approach for temporal blocking with sparse off-the-grid operators.

Intermediate representations (IRs) Compilers often use IRs to optimise code. IRs should be accurate and well-defined and include all the necessary information to represent the semantics of the source code. Some of these IR-based compiler frameworks have automated or/and pre-defined optimisation processes. In contrast, others may offer some form of control over the optimisation process. IR-based transformations are often asserted through data dependence analysis. The impact of the transformations on performance is often evaluated using cost models. The cost model is expected to answer questions like: Should we optimise loops using fusion and then CSE or the other way round?

Furthermore, this chapter further categorises frameworks based on the characteristics of the DSL and their intermediate representations. More specifically, we often use the following terms as features of each framework.

The features that are included are:

Symbolic math DSL Support for a high-level domain-specific language that can support problem setups in a notation close to textbook mathematics.

Stencil-level DSL Support for a domain-specific language where users can set up their problem using stencils and loops, but not mathematics or physics.

Scheduling language Support for a user-defined schedule to set the order and the nature of the already available optimisations within a larger optimisation context.

Graph/Tree-based IR Framework actively uses an IR capable of being transformed or rewritten. The IR is not a text-like representation but rather a graph-like/tree data structure.

Polyhedral Framework is based upon the Polyhedral model, supporting polyhedral transformations.

2.4.1. Automated cache blocking, the example of Devito

Section 2.3 introduced the reader to space loop blocking, one of the most prominent optimisations regarding stencil computations. More specifically, Listing 2.15 illustrated a code snippet of a standard non-blocked implementation to a loop-blocked, cache-optimised implementation. This optimised implementation requires significant manual effort. Optimising by hand should be reduced, which is where automating optimisations come to the rescue. This subsection will show an example of an automatically generated code output. We aim to showcase the ease of optimising code from a user perspective. The answer to how this is done will be presented in Chapter 4.

We will use a time iterative Laplacian 3D stencil example. Listing 2.17 shows the DSL code to model such an example and shows how to generate two different Operators. A non-optimised one named `op0` and an optimised one named `op1`.

In Listing 2.17, we observe that the only change between `op0` and `op1` is one keyword. The underlying compiler takes care of the optimisation level (given as an argument when constructing an operator) and produces optimised code. The stencil code snippet from the unoptimised version is shown in Listing 2.18. Now let us see the optimised version of this code. In Listing 2.19, we notice the additional generated loops that facilitate cache blocking, as previously seen in Section 2.3 and more specifically in Listing 2.15. The generated code also has other optimisations, like loop invariant code motion (also known as lifting dimension-invariant expressions) or common sub-expression elimination. Applying all these optimisations by hand is tedious and error-prone. We must carefully maintain the iteration space's properties and correctness and avoid falling into any arithmetic mistakes. Frameworks like Devito ease our life when it comes to these optimisations. The following subsections review frameworks that work

Listing 2.17: Generation of two kernels for a Laplacian time-iterative model. Unoptimised (op0) and optimised (op1). Optimised code generation is attained with a single argument in the Operator construction.

```
# Some variable declarations
nt, nx, ny, nz = 50, 10, 10, 10
dx = 2. / (nx - 1)
dy = 2. / (ny - 1)
dz = 2. / (nz - 1)
sigma = .25
nu = .5
dt = sigma * dx * dy * dz / nu


# Grid initialisation
grid = Grid(shape=(nx, ny, nz))
u = TimeFunction(name='u', grid=grid, space_order=2)

# Initialise u
init_value = 6.5
u.data[:, :, :] = init_value

a = Constant(name='a')
eq = Eq(u.dt, a*u.laplace + 0.1, subdomain=grid.interior)
stencil = solve(eq, u.forward)
eq0 = Eq(u.forward, stencil)

op0 = Operator(eq0, opt=('noop'))
op1 = Operator(eq0, opt=('advanced'))
```

towards this direction and have influenced both Devito and this thesis.

2.4.2. DSL frameworks with builtin compiler

In Section 2.2, we discussed several frameworks that offer stencil modelling from a higher or lower-level abstraction. Some of these frameworks do not only abstract away the symbolic nature of modelling. They also abstract away stencil-related optimisations.

Devito offers a built-in compiler that automatically applies a plethora of optimisations to stencil computations. Apart from cache-blocking optimisations, as seen in Section 2.4.1, Devito can support the generation of highly optimised parallel code supporting SIMD vectorisation, CPU and GPU parallelism via OpenMP and OpenACC, and multi-node parallelism via MPI. Additionally, it offers cache blocking, aggressive symbolic transformations for FLOP reduction and distributed arrays over multi-node (MPI) domain

Listing 2.18: A non-optimised version of Devito generated code for a Jacobi-like Laplacian stencil.

```
1  for (int time = time_m, t0 = (time)%(2), t1 = (time + 1)%(2); time
       <= time_M; time += 1)
2  {
3    /* Begin section0 */
4    START_TIMER(section0)
5    for (int x = x_m; x <= x_M; x += 1)
6    {
7      for (int y = y_m; y <= y_M; y += 1)
8      {
9        for (int z = z_m; z <= z_M; z += 1)
10       {
11         u[t1][x + 2][y + 2][z + 2] = dt*(a*(u[t0][x + 1][y + 2][z
               + 2]/pow(h_x, 2) - 2.0F*u[t0][x + 2][y + 2][z + 2]/pow
               (h_x, 2) + u[t0][x + 3][y + 2][z + 2]/pow(h_x, 2) + u[
               t0][x + 2][y + 1][z + 2]/pow(h_y, 2) - 2.0F*u[t0][x +
               2][y + 2][z + 2]/pow(h_y, 2) + u[t0][x + 2][y + 3][z +
                2]/pow(h_y, 2) + u[t0][x + 2][y + 2][z + 1]/pow(h_z,
               2) - 2.0F*u[t0][x + 2][y + 2][z + 2]/pow(h_z, 2) + u[
               t0][x + 2][y + 2][z + 3]/pow(h_z, 2)) + 1.0e-1F + u[t0
               ][x + 2][y + 2][z + 2]/dt);
12       }
13     }
14   }
15   STOP_TIMER(section0,timers)
16   /* End section0 */
17 }
```

decompositions, inspection and customisation of the generated code, autotuning framework to ease performance tuning and smooth integration with popular Python packages such as NumPy, SymPy, Dask, and SciPy and machine learning frameworks such as TensorFlow and PyTorch.

OPS [Reguly et al., 2018] uses a source-to-source translator to automatically parallelise and apply loop-blocking optimisations to structured grid stencils targetting multi- and many-core architectures. Additionally, as benefited in OpenSBLI [Jacobs et al., 2017], the OPS library then optimises the code. It targets specific hardware backends, such as MPI/OpenMP execution on CPUs and CUDA/OpenCL execution on GPUs.

Exastencils [Lengauer et al., 2014] code generation framework consists of a source-to-source compiler that, among others, includes vectorisation, common subexpression elimination, polyhedral loop transformations and loop unrolling. Exastencils can support several backends based on MPI, OpenMP, CUDA and combinations thereof, targeting CPUs, GPUs and hybrid architectures.

Listing 2.19: An optimised version of Devito generated code for a Jacobi-like Laplacian stencil.

```
1  float r0 = 1.0F/dt;
2  float r1 = 1.0F/(h_x*h_x);
3  float r2 = 1.0F/(h_y*h_y);
4  float r3 = 1.0F/(h_z*h_z);
5
6  for (int time = time_m, t0 = (time)%(2), t1 = (time + 1)%(2); time
       <= time_M; time += 1)
7  {
8    /* Begin section0 */
9    START_TIMER(section0)
10   for (int x0_blk0 = x_m; x0_blk0 <= x_M; x0_blk0 += x0_blk0_size)
11   {
12     for (int y0_blk0 = y_m; y0_blk0 <= y_M; y0_blk0 +=
           y0_blk0_size)
13     {
14       for (int x = x0_blk0; x <= MIN(x0_blk0 + x0_blk0_size - 1,
             x_M); x += 1)
15       {
16         for (int y = y0_blk0; y <= MIN(y0_blk0 + y0_blk0_size - 1,
               y_M); y += 1)
17         {
18           #pragma omp simd aligned(u:32)
19           for (int z = z_m; z <= z_M; z += 1)
20           {
21             float r4 = -2.0F*u[t0][x + 2][y + 2][z + 2];
22             u[t1][x + 2][y + 2][z + 2] = dt*(r0*u[t0][x + 2][y +
                   2][z + 2] + a*(r1*r4 + r1*u[t0][x + 1][y + 2][z +
                   2] + r1*u[t0][x + 3][y + 2][z + 2] + r2*r4 + r2*u[
                   t0][x + 2][y + 1][z + 2] + r2*u[t0][x + 2][y + 3][
                   z + 2] + r3*r4 + r3*u[t0][x + 2][y + 2][z + 1] +
                   r3*u[t0][x + 2][y + 2][z + 3]) + 1.0e-1F);
23           }
24         }
25       }
26     }
27   }
28   STOP_TIMER(section0,timers)
29   /* End section0 */
30 }
```

STELLA [Gysi et al., 2015] and GridTools library [Afanasyev et al., 2021] with regards to automated optimisations, provide halo exchanges, data management and bindings to C and Fortran. Besides the CPU backend, they offer a CUDA GPU backend with improved performance versus the current official version.

2.4.3. Polyhedral compilers

These compilers enhance productivity by exploring a specific optimisation space and searching for an optimal combination of all or a subset of the available optimisation passes. This Section discusses several compilers that fall within the polyhedral compilation area.

PENCIL [Baghdadi et al., 2015] aims to attain parallelism and performance portability on accelerators. A strictly defined subset of GNU C99 is enriched with additional language constructs to produce highly optimised code targeting accelerators. PENCIL can be regarded and used as a portable implementation language for libraries and a target language for DSL compilers. PENCIL incorporates sequential semantics, strictly complies with C, and has a rich set of attributes and pragmas that enable static analysis. PENCIL can apply parallelism and advanced loop transformations to non-static affine code and access patterns based on the polyhedral model.

PLUTO [Bondhugula et al., 2008b,a] serves as an automatic parallelisation tool for affine loop nests targetting coarse-grained parallelism and data locality through a source-to-source transformation. The core transformation framework mainly searches for affine transformations for efficient tiling, OpenMP parallelism and vectorisation. Although the schedule is fully automatic (C to OpenMP C), users can still tune tile shapes, unroll factors, and outer loop fusion structure before execution. PLUTO's code generation uses **Cloog-ISL** [Bastoul, 2004].

Polly [Grosser et al., 2012a] aims to seamlessly integrate the polyhedral model into production-grade compilers, specifically, into LLVM. For this purpose, Polly translates code to LLVM's internal, low-level IR. Polly introduces infrastructure for polyhedral optimisations on LLVM's IR. By identifying the section of codes that can be expressed as polyhedra, Polly translates them to a Z-polyhedral representation. This representation is further optimised, and then an optimised IR is generated. Afterwards, parallelism is applied by generating SIMD and OpenMP code. Polly is also bridged with PLuTo as an external optimiser and parallelizer.

Loo.py [Klöckner, 2014] is embedded in Python and mainly targets array-type computations such as dense linear algebra, convolutions, n-body interactions, finite difference and finite element computations. Loo.py

has a library of transformations that operate on a predefined data model for array-type computations. Loo.py aims to provide user-friendly and extensible IRs and retain a high-level interface. Loop tiling, vectorisation, storage management, unrolling, instruction-level parallelism, and data-layout transformations are among the transformations offered. Loo.py aims to provide a smooth way to transit from the prototype to high-performance implementation.

Tensor Comprehensions [Vasilache et al., 2018] is based on generalised Einstein notation for computing on multi-dimensional arrays. TC offers a DSL to express mathematics for deep learning and a polyhedral Just-In-Time compiler to transform mathematical descriptions of a deep-learning DAG into a CUDA kernel. TC library has been designed to offer performance portability and machine-learning framework agnostic. It only requires a tensor library with memory allocation, offloading and synchronisation capabilities. TC's syntax helps to simplify machine learning framework implementations and can be efficiently translated to high-performance computation kernels. TC tries to use other frameworks when possible, avoiding reimplementing existing science. TC synthesises high-performance machine learning kernels using Halide[Ragan-Kelley et al., 2017], ISL[Bastoul, 2004], NVRTC [1] or LLVM [Lattner and Adve, 2004].

Other compiler frameworks are based on the polyhedral model or offer users a scheduling language. This scheduling language provides the users with the necessary infrastructure to explore the optimisation space and fine-tune their applications.

A representative example of this class of frameworks is **TIRAMISU** [Baghdadi et al., 2019]. TIRAMISU is a polyhedral compiler with a scheduling language featuring novel commands for controlling data communication, synchronisation, and mapping to different memory hierarchies, multicore X86 CPUs, Nvidia GPUs, Xilinx FPGAs (Vivado HLS) and distributed machines (using MPI). Tiramisu's target applications are dense and sparse deep learning and data-parallel algorithms. More specifically, optimising recursive neural networks, image processing and scientific computing. Its polyhedral nature helps perform complex loop transformations to programs with cyclic data flow graphs. It also helps to represent non-

[1] https://docs.nvidia.com/cuda/nvrtc/index.html

rectangular loops, affine transformations, and data layout transformations and guarantees optimisations' correctness through dependence analysis. A simple C++ API is provided for expressing algorithms and scheduling their optimisation pipeline. Tiramisu's IR is divided into four layers to ease the scheduling language implementation. These four layers manage algorithm abstraction, computation, data, and communication management. Finally, an abstract syntax tree leads to automated code generation. This four-layer system allows portable and simply composable architecture-specific lowering transformations.

AlphaZ [Yuki et al., 2013] differs from many existing tools in that it tries to offer the user complete control of the transformations applied to the code. Still a polyhedral framework that aims to act as an automatic parallelizer. Users can experiment with new program optimisations that similar tools may need to automate, as AlphaZ aims to offer a smooth prototyping experience.

2.4.4. Other compiler frameworks

This subsection refers to other compiler frameworks not based on the polyhedral framework. We discuss their structure and the features they offer.

Such an example is **The Open Earth Compiler** [Gysi et al., 2021]. The Open Earth Compiler follows the approach of domain-specific multi-Level IR rewriting to target GPU-accelerated climate simulation. The core of the Open Earth compiler consists of a set of MLIR dialects [Lattner et al., 2020]. MLIR dialects are collections of domain-specific operations, transformations, and conversions between them. A stencil dialect helps define the stencil at a higher level and covers a range of applications except climate modelling (such as image processing and seismic imaging). Stencils are lowered through a series of IRs featuring affine operations, structured control flow, and arithmetic operations starting from the stencil dialect. These are available in MLIR and enhanced with loop- and value-level transformations such as unrolling or common subexpression elimination. The Open Earth compiler IRs help to represent a structured loop abstraction. This structure is used to design a generic GPU kernel dialect and implement loop-to-kernel conversion using simple patterns. These patterns are based

on the parallelism information preserved from the stencil level. Open Earth transforms a high-level climate code into a fast target-specific binary as an end-to-end compiler.

Lift [Steuwer et al., 2015, 2017] is a framework built on a level close to the generated code compared to other high-level DSLs. Lift arose as a promising performance portability framework based upon a small set of reusable parallel primitives upon which DSLs can be built. Lift aims to bridge the gap between the growing ecosystem of DSLs in order to reduce the effort every DSL is making to support standard optimisations such as parallelism. Lift encodes optimisations as a system of extensible rewrite rules. These rules can then be used to explore the optimisation space. The primary target computational kernels are linear algebra ones and recently has been extended to stencil computations [Hagedorn et al., 2018]. Lift is based upon semantic-preserving rewrite rules. Applications employ a set of functional primitives, and optimisations are encoded as rewrite rules. These rules form a space for exploring optimisations and act as a foundation for alleviating programmers working on higher abstraction levels tediously rewriting and tuning their implementations. The rewrite passes optimisation space is automatically explored [Steuwer et al., 2016] for each target architecture. To assemble an application, the programmers can benefit from the high-level functional primitives (map, reduce, scan). This functional primitive technique allows the expression of multidimensional stencil kernels by composing and nesting intuitive primitives. Furthermore, the rewrite rules help formalise algorithmic and device-specific optimisations for performance portability. There is a clear separation between the computation and the implementation of it. High-level expressed computations are rewritten to lower-level expressions, which are still functional and encode parallelism and cache-related aspects for optimisation.

YASK [Yount, 2015, Yount and Duran, 2016, Yount et al., 2017] is yet another framework to generate high-performance stencil code efficiently. YASK has heavily influenced the work in this thesis. It can support modelling boundary conditions and staggered-grid stencils and has APIs for C++ and Python. The generated code offers various optimisations, including vector-folding to increase data reuse via non-traditional data layout and shared- and distributed-memory parallelism supporting computa-

tion/communication overlap. It also supports temporal tiling in multiple dimensions to increase cache locality further compared to space blocking, which we will discuss in Section 4.2.1. Both traditional loop blocking and temporal blocking can be autotuned.

Apache TVM [Chen et al., 2018] is an open-source machine learning compiler framework for CPUs, GPUs, and machine learning accelerators. It aims to enable machine learning engineers to optimise and run computations efficiently on any hardware backend by providing end-to-end compilation. TVM offers a compiler stack to bridge the gap between productivity and efficiency, focused on deep learning frameworks and their hardware backends. It is another project that targets teams of interdisciplinary scientists. TVM compiles deep learning models into minimum deployable modules and infrastructure to automatically generate and optimise models on more backends with better performance.

Delite [Sujeeth et al., 2014] compiler provides components to ease the process of DSL development on top of it. These components, such as parallel patterns, optimisations, and code generators, can be reused in the DSL ecosystem. Delite is embedded in Scala but uses metaprogramming to construct an IR layer of user programs and compile to multiple targets (including C++, CUDA, and OpenCL). Delite offers automatic parallelisation and heterogeneous platform execution for machine learning, data querying, graph analysis, and scientific computing.

PATUS [Christen et al., 2011] is a code generation tool for stencil computations. It provides a software infrastructure targetting automatic code generation for architecture-specific (CPUs, GPUs) stencil kernels. Stencil specification at a higher level incorporates domain-specific knowledge to optimise the computation further. In addition, it serves as an experimentation toolbox for parallelism and optimisation strategies. Patus DSL is small and predefined strategies can be employed to schedule the kernel optimisations and parallelism. Custom user schedules also make it possible to experiment with other algorithms or better map to the hardware.

Pochoir [Tang et al., 2011] is a notable work in this area and consists of three main components: an embedded domain-specific language in native C++ for stencil specification, a C++ template run-time library, and a domain-specific compiler to optimise the performance at compile time. In Pochoir, a compiler translates the high-level specification into cache-

oblivious algorithms for multi-core CPUs using the Cilk language.

Other frameworks that contributed to the scientific community are briefly referred to here. **Unified Representation Universal Kernel (URUK)** [Girbal et al., 2006] which is based upon the polyhedral compiler. The **Bricks** framework [Zhao et al., 2018a,b] is primarily working with data layout transformations for performance portability across multiple architectures. It generates code with optimised blocking and efficient halo exchanges for domain decomposition. **POET** [Yi, 2012] a script-driven transformation engine for source-to-source transformations aiming to expose parameterized transformations via scripts. **SBLOCK** [Brandvik and Pullan, 2010], a Python framework for structured grid stencil computations. **Mint** [Unat et al., 2012] a framework based on pragma directives targeting GPUs. It has been used to accelerate 3D earthquake simulations. **PPCG** [Verdoolaege et al., 2013], a polyhedral compiler, source-to-source translator targetting GPU code generation for affine loops. **NOVA** [Collins et al., 2014], a functional language and compiler for multi-core CPUs and GPUs offering a polymorphic, statically-typed functional language with a suite of higher-order functions that are used to express parallelism. NOVA compiler is stand-alone and can be easily used as a target for higher-level or domain-specific languages or embedded in other applications. It generates code for a variety of target platforms and is performance-portable. **Composable High-Level Loop (CHiLL)** [Chen et al., 2007] is another high-level transformation and parallelisation framework using the polyhedral model. CHiLL supports a wide variety of loop transformations for loop nests. These transformations can be pipelined and applied in sequence. **SDSLc** [Henretty et al., 2013] is a compiler for the Stencil-DSL (SDSL), embedded in C, C++ and MATLAB. The SDSL compiler targets performance portability for CPUs, GPUs and FPGAs.

Other structured stencil code-generation frameworks worth mentioning and aiming to increase the performance of the generated code were presented in Zhang and Mueller [2012], Datta et al. [2008], Henretty et al. [2013], Zhao et al. [2018a], Hawick and Playne [2013].

Note: Although it is unclear whether some of these works are being employed in production codes, they paved the pathway to generalising several approaches towards high-performance code generation. Some of these frameworks have been discontinued, but some are still maintained as part of other larger projects.

2.4.5. Domain-Specific Abstractions and Optimisations

General-purpose or polyhedral compilers aim to abstract away optimisations from the users. This abstraction often targets broad classes of problems trying to apply general-purpose optimisations to the maximum possible subset of these problems. The trade-off of this approach is that we often miss optimisation opportunities that apply to only a narrow class of problems. Specific mathematic properties of the problem may often not be accounted towards these specific optimisations. This section briefly reviews some domain-specific optimisation systems that target narrow classes of problems and inspire our work. We review some high-level compiler frameworks that are not targetting finite differences; however, they have a notable compiler infrastructure that has influenced our work.

Halide [Ragan-Kelley et al., 2013, 2017] introduced Halide, a high-level language for expressing image processing and tensor kernels. Halide is embedded in C++ and is portable on many CPU and GPU architectures. Halide is proven to be successful in practical applications. It is currently employed to develop production code by several companies such as Google and Adobe.

Image processing kernels are often a sequence of interconnected stages. Most people have often used a gaussian blur, greyscaling, brightening, and contrast to edit an image. Each of these operations is a numerical computation kernel over an image. This image can be conceived as a tensor or array. There are often practical applications where image processing pipelines can be very complex and consist of several stages of the order of tens.

Halide is based on separating the underlying algorithm from its schedule. It allows programmers to express many optimisation combinations given a space of schedules. The Halide compiler uses the schedule to express many possible algorithm organisations. Halide supports a range of optimisations. Some are cache locality optimisations, loop reordering transformations, shared- and distributed memory parallelism, and SIMD vectorisation. The optimisation space is explored and auto-tuned at runtime. The optimisation strategy selected does not affect the correctness of the program. The resulting performance has been demonstrated to match or outperform state-of-the-art manually written and optimised codes.

PolyMage [Mullapudi et al., 2015] is a domain-specific language and optimising code generator. Similarly to Halide, Polymage seeks to optimise image processing pipelines automatically. The user expresses an image processing pipeline in a high-level Python-embedded language. Polymage gets this pipeline as input and generates optimised and parallelised C++ code to target multicore CPUs. In contrast to Halide, PolyMage's optimiser is primarily polyhedral-based for parallelism and data locality optimisations.

Forma [Ravishankar et al., 2015] is a DSL and compiler framework to generate C code for multicore CPUs and NVIDIA GPUs automatically. Forma targets image processing pipelines and is an excellent example of abstracting problem modelling from low-level optimisations. Forma DSL offers a rich stencil-level DSL and automatically handles, among others, memory management, parallelisation, vectorisation, fusion and tiling.

The **Tensor Contraction Engine (TCE)**, now part of **NWChem** [Apra et al., 2020], is a domain-specific language for quantum chemistry. Chemists express computations in a high-level language, and TCE optimises their computation. TCE uses algebraic transformations to reduce the cost of computational kernels [Hartono et al., 2006, 2009] and tunes the kernel by exploring the most suitable trade-off between redundant computation and data movement [Lam et al., 2011]. TCE employs several optimisations, such as loop fusion and optimal evaluation for dynamically allocated intermediate arrays. Recomputation can be traded for data locality and storage requirements. Other optimisations improve the communication cost for multi-processor CPUs and minimise the time needed to fetch data by reducing data movement. Finally, the Fortran code is generated. Many of these optimisations exploit the mathematical structure inherent in tensor contractions.

ARTEMIS, [Rawat et al., 2018], is another code generation framework targetting stencil computations on GPUs that abstracts away code generation concepts from the users. ARTEMIS offers a stencil-level DSL and supports optimisation scheduling and bottleneck analysis to tune the performance of applications. By stencil-level DSL, we refer to languages that specify computations in a stencil-like language.

Other DSL and code generation frameworks that will be further discussed later in this thesis (see Section 4.2) is **AN5D**, [Matsumura et al., 2020], and **Formura DSL**, [Muranushi et al., 2016, Tanaka et al., 2018].

The **Spiral project** [Püschel et al., 2005, Franchetti et al., 2018] is a pioneering project on automated code generation targetting digital signal processing transforms, including the discrete Fourier transform and other trigonometric, filter, and discrete wavelet transforms. Starting from a high-level specification of a mathematical problem, Spiral generates high-performing, performance-portable, tuned code through a set of rewrite rules (see also 2.4.4). The mathematical formulae are turned into an IR, enabling explicit parallelism and vectorisation. Afterwards, the IR is lowered to low-level code for compilation and execution. Spiral is different from other projects in employing a feedback-driven optimiser. This feedback enables further optimisations that map better to the underlying microarchitectural details.

Computations in engineering, finance, and scientific computing are often reduced to **linear algebra** operations. The cost of these computations is often high due to the number of linear algebra operations, the input size, or both. Academics or vendors have developed a considerable number of math libraries to facilitate the need to reduce this cost. Significant efforts have been made to improve the performance of these libraries to ensure high performance for the scientific applications that use these libraries. Some of the libraries that are worth to be mentioned in this thesis are presented here.

Intel's Math Kernel Library [Wang et al., 2014] is one of the fastest and most famous libraries targetting Intel-based systems. Main Intel MKL interfaces include vector, matrix-vector, and matrix-matrix operations (*BLAS* [Blackford et al., 2011]), operations on sparse vectors and matrices (Sparse BLAS [Duff et al., 2002]), systems of linear equations, least-square problems, eigenvalue and singular value problems [Anderson, 1987] and others. **Nvidia's cuBLAS** library is highly optimised for performance on NVIDIA GPUs. It leverages tensor cores to accelerate low and mixed-precision matrix multiplication [NVIDIA, 2022].

Other libraries include **LIBXSMM** [Heinecke et al., 2016] and **BLIS** [Van Zee and van de Geijn, 2015]. The field of linear algebra libraries is of interest to us because cache-level abstractions are interesting for our work presented in this thesis in Chapter 4.

2.4.6. Frameworks Overview

Finally, we present an overview of the majority of frameworks mentioned in this Chapter. Table 2.2 illustrates the features that can shortly characterize these frameworks.

2.5. On the Terminology Adopted

In this section, we expand on the terminology used throughout this thesis. The terminology used is standard and close to the one used in reference textbooks such as Hennessy and Patterson [2017]. We review the relevant keywords and provide a brief reference for each. We consider this review to guide the reader when discussing performance evaluation metrics. We encourage the reader to refer to this section, primarily when performance evaluation is discussed throughout this thesis. Some of these terms are commonly used in similar works in the extended area of computer architecture, domain-specific languages and HPC [Rathgeber, 2014, Luporini, 2016].

Compilers

General-purpose compiler We generically refer to any open-source or commercial compilers capable of translating low-level source code (e.g., Fortran, C, C++) into lower-level machine code (e.g. assembly/object/machine code). Examples are the GNU (*gcc, g++*), Intel (*icc*), LLVM (*clang*), Cray (*cce*) and NVIDIA (*nvc, nvc++*) compilers. By "general-purpose," we aim to differentiate these compilers from all other higher-level compilers, such as those based on the polyhedral model or those used for translating domain-specific languages (see Section 2.4).

Source-to-source compiler By source-to-source compiler or source-to-source translator, we mostly refer to compilers whose input and output have a similar level of abstraction as opposed to traditional compilers that start from a higher and end to a lower level of abstraction.

Table 2.2.: Overview of notable stencil-related DSL/compiler frameworks and their features in literature

		Features				
		Symb. math level DSL	*Stencil level DSL*	*Scheduling Language*	*Graph/AST-based (IR)*	*Polyhedral*
DSL and compiler frameworks	Devito	✓	✓	✓	✓	
	OPS/OP2		✓		✓	
	Exastencils	✓	✓		✓	
	GridTools		✓		✓	
	Stella		✓			
	Simflowny	✓				
	PENCIL				✓	✓
	PLUTO					✓
	Polly			✓	✓	✓
	Loo.py		✓	✓	✓	✓
	TIRAMISU		✓	✓	✓	✓
	Halide		✓	✓	✓	
	Forma		✓	✓		
	Polymage		✓	✓		✓
	Tensor CE	✓			✓	
	Lift		✓		✓	
	ChiLL			✓		✓
	Bricks		✓			
	ARTEMIS		✓	✓		
	Formura	✓	✓			
	AN5D		✓			✓
	OpenSBLI	✓			✓	
	Apache TVM	✓		✓		

Performance Metrics and Cost Models

Data locality Data locality indicates how close the distance between where data is and where this data needs to be processed. The shorter the distance, the better the data locality is [Unat et al., 2017].

(High) Memory pressure This is often used to emphasise the fact that one or more levels of the memory hierarchy (e.g., RAM, caches, registers) are stressed by a relatively large number of load/store instructions. High memory pressure is often responsible for performance degradation.

Operational intensity This parameter defines the ratio of total operations to total data movement (bytes) between the DRAM and the cache hierarchy for a given code section. The operational intensity helps to "*p*redict the DRAM bandwidth needed by a kernel on a particular computer". A computational kernel is primarily compute-bound with high operational intensity and memory-bound with low operational intensity.

Memory- and compute-boundedness A code section can be compute-bound or memory-bound on a specific platform. In the former case, the performance achieved is limited by the operation throughput of the CPU; in the latter case, the memory bandwidth or the memory latency is the limiting factor. Cache-level optimisations such as space blocking and temporal blocking increase data locality, thus tackling memory-boundedness by maximising the cache hit ratio, thus reducing latency and memory pressure. Many domain-specific optimisations, such as flop reduction techniques, target compute-boundedness instead; for example, several Devito optimisations manipulate mathematical formulae to reduce the operation count of the resulting kernels.

Arithmetic intensity Sometimes, arithmetic intensity is used instead of operational intensity. There are mainly two differences. First, only the fraction of arithmetic operations emitted is considered instead of all operations. Secondly, we work under the hypothesis that the total data movement occurs between the CPU and the last cache level. We explicitly distinguish it in our metrics whenever we do not work under this hypothesis (see item 2.5 - The roofline model).

The roofline model The operational intensity is helpful to derive *roofline plots* [Williams et al., 2009]. A roofline plot is particularly helpful for studying a program's computational behaviour since it provides insight into the performance bottlenecks and, therefore, the most valuable optimisations. A roofline model mainly comprises two lines, a line with a slope that depicts memory-bandwidth limitations and a flat line depicting compute-bound limitations. For cache-related limitations, multiple sloped lines can be added to depict these limitations for every cache level. More information on the roofline model can be retrieved from [Williams et al., 2009] and [Ilic et al., 2014].

There are two main roofline models we can use for cache-related benchmarking.

- The Classical roofline, introduced by [Williams et al., 2009]. Here, the arithmetic intensity is determined by measuring traffic from DRAM.
- The Cache-Aware roofline, introduced by [Ilic et al., 2014]. Here, we measure memory traffic from any given level in the memory hierarchy (L1, L2, L3).

This thesis often uses roofline models to indicate performance improvement from cache-related optimisations.

Non-uniform memory architecture and locality issues Non-Uniform Memory Architecture separates RAM and memory management units associated with CPU sockets. Often a processor living in a socket needs to access memory from another socket. There is additional latency overhead compared to accessing local memory in this case. In order to increase data locality, it is good to avoid remote memory access. Proper data placement increases the overall bandwidth and improves the latency to memory [Lameter, 2013]. These are called NUMA issues, and for more details on how we resolve them when running benchmarks, see Paragraph 3.4.

Miscellanea

Real-world applications Throughout the thesis, we often use the term "real world" or "practical applications" to emphasise that we evaluate our models based on equations, numerical methods, and datasets used in scientific applications in academia or industry. This practice provides compelling evidence about our contributions' effectiveness and applicability to the optimisations presented. Often, people evaluate against simplistic benchmarks that do not comprise the challenges existing in real-world problems.

2.6. Summary

In this Chapter, we reviewed the essential scientific background for this thesis. Section 2.1 presented the mathematical theory for the finite-difference method and how this translates from PDEs to stencil kernels. We present simple examples to help the reader progress through a smooth transition.

In Section 2.2, we introduce frameworks that automate the finite difference method. We showcase Devito to model these partial differential equation problems and automatically generate high-performing code. We discuss FD software abstractions used for solving partial differential equations via lowering to stencils. Additionally, we briefly mention other frameworks for solving PDEs with other methods.

Section 2.3.1 shows that stencils can be highly optimised with several optimisations. We primarily focus on cache-related optimisations like loop blocking 2.3.1. We also discuss temporal blocking, an essential part of this thesis. These optimisations are challenging to apply by hand. The need to automate optimisations is obvious.

Automating optimisations has been a research target for several years. In Section 2.4, we review these frameworks and the state-of-the-art in performance optimisation, focusing on frameworks that aim to automate compiler optimisations. Specific focus is given in Devito.

Finally, Section 2.5 briefly refers to the terminology adopted throughout the thesis.

Chapter 3

Wavefront temporal blocking of finite-difference stencil operators with sparse "off-the-grid" sources

As discussed in Chapter 2 and more particularly in Section 2.3.1, cache optimisations help reduce the required memory bandwidth of stencil computations by reusing data from the cache. Similarly, temporal blocking, another cache optimisation presented in Section 2.3.3, enhances this reuse in the time dimension. However, even though the efforts of the frameworks presented in Section 2.4 are notable, there are still real-world problem cases where cache optimisations are hard to apply. This chapter introduces an approach to applying wavefront temporal blocking to a class of stencil kernels with loop structures more complex than the stencils commonly studied in the literature. This complexity makes the application of cache optimisations far from straightforward. This chapter will focus on scientific applications with stencils on structured meshes that consist of sources and receivers, resulting in non-typical loop structures with additional complexities. These loop structures stem from practical applications and consist of sparsely located operators not aligned with the computational grid. From now on, we refer to these positions as "off-the-grid" positions [Yan et al., 2015, Poon and Peyré, 2019]. More details on the "off-the-grid" posi-

tions will be presented in Section 3.1.1. We explain why these loops pose challenges to temporal blocking and how our approach overcomes these challenges. To overcome these challenges, we introduce a methodology to inspect complex data dependencies arising from "off-the-grid" operators and reorder the computation, leading to performance gains in stencil codes where temporal blocking has not been applicable. We use the wavefront temporal blocking scheme to showcase the successful application of our methodology approach.

We implement this novel methodology in the Devito domain-specific compiler toolchain. We evaluate the potential improvement and the limits to applicability using standard benchmarks in industrial seismic imaging such as the isotropic acoustic, anisotropic acoustic, and isotropic elastic wave propagators. We compare this wavefront temporal blocking model over highly-optimised, spatially-blocked vectorised code. Performance evaluation after auto-tuning shows that our contributions unlock substantial speed-up ranging from 1.1x to 1.6x for space discretisation orders of 4 and 8.

3.1. Motivation and Related Work

Often the stencil kernels benchmarked in the literature about cache-blocking optimisations for FD codes do not resemble scientific simulations. These stencils often neither simulate real-world phenomena nor resemble part of practical studies. These benchmarks, as in Wonnacott [2000], Guohua Jin et al. [2001], Wonnacott [2004], Wellein et al. [2009], Strzodka et al. [2011], Schäfer and Fey [2011], Holewinski et al. [2012], Bandishti et al. [2012], Grosser et al. [2013a, 2014a], Bertolacci et al. [2015], Muranushi and Makino [2015], Levchenko and Perepelkina [2017], Rawat et al. [2018], Zohouri et al. [2018], Wang and Chandramowlishwaran [2020] are more "theoretic rather than practical," meaning that they are closer to textbook examples rather than simulations of industrial interest. Characteristic examples are PDEs similar to the Heat and the Laplace equations (see subsections 2.1.3.1 and 2.1.3.3). To model scientific simulations, textbook examples need to be enriched with more equations to assist in modelling, such as initial and boundary conditions or external operators affecting the computation. Our interest only focuses on the prementioned specific class

of sparse "off-the-grid" operators.

Temporal blocking has proven beneficial for stencil kernels, improving cached data reuse. However, published benchmarks rarely widen their focus beyond the stencil kernel, resulting in a disconnect between reported results and real-world applications, which often need more equations to assist modelling, such as damping equations, initial and boundary conditions or external operators that may be affecting the computation.

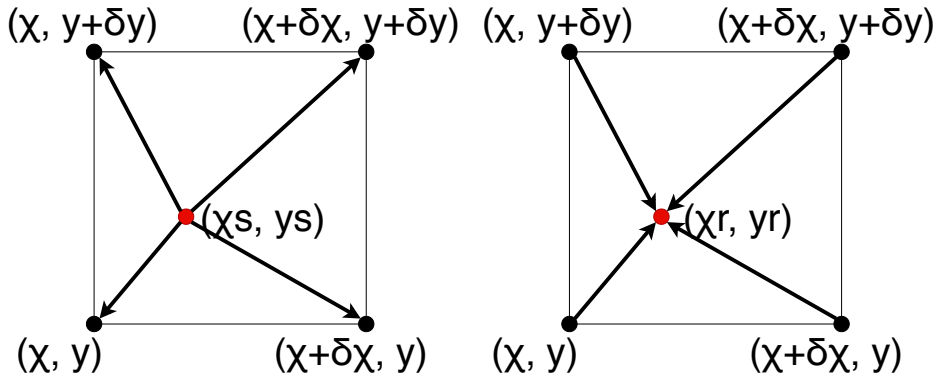
These additional factors affect its complexity and, therefore, its computational cost. This complexity may impede the application of optimisations, as this Chapter will describe in Section 3.1.1. Our aim in this thesis is to apply temporal blocking to building blocks of real-world scientific simulations. More specifically, our approach is evaluated using the wavefront temporal blocking scheme.

The work presented in this chapter is motivated by applying temporal blocking to practical problems in which source injections result in wavefields that must be measured at receivers by interpolation from the gridded wavefield. This setup is prevalent in seismic and medical imaging. Especially in seismic imaging, solvers for isotropic acoustic [Levander, 1988], anisotropic acoustic [Zhang et al., 2011, Duveneck and Bakker, 2011] and elastic [Virieux, 1986] wave propagation kernels using explicit finite differences are commonplace. These wave propagation kernels are responsible for a significant computational part of forward and backward modelling in seismic applications. Characteristic examples of such applications include full-waveform inversion (FWI) [Virieux and Operto, 2009] and reverse time migration (RTM) [Baysal et al., 1983, Chang and McMechan, 1990]. The following subsections describe an example of a scientific application with sources and receivers, how we iterate over them, the additional data dependencies induced, and a description of wavefront temporal blocking, one of the many available temporal blocking schemes.

3.1.1. Sparse "off-the-grid" operators

As discussed earlier, this work is highly influenced by real-world problems, mainly in the area of wave propagators. Sources that inject waves and receivers that collect traces are required to model seismic and medical imaging problems. It is a common pattern to have sparse "off-the-grid"

operators together with structured grid operations. These operators are sparsely placed within the computational domain and non-aligned to the structured FD grid. Usually, we need to model the scattering or gathering of data at positions non-aligned to the structured grid. These operations interpolate values and are treated differently than a classical PDE-stencil update. We call the scattering of data a "source operation" and the gathering of data a "receiver operation". These sources and receivers are commonly present in real-world applications. Figure 3.1 shows an example of bilinear interpolation, where 4 points are affected in 2D space. Subfigure 3.1a resembles a source injection of a wave, while subfigure 3.1b resembles a receiver collecting traces by interpolation.



(a) An off-the-grid source injects values to neighbouring grid points.

(b) An off-the-grid receiver interpolates values from neighbouring grid points.

Figure 3.1.: A source injection and a receiver measurement interpolation at off-the-grid positions in a 2-D FD-grid. We assume linear interpolation.

The bilinear interpolation is the weighted average of the values at the four corners of the rectangle (χ, y), $(\chi, y+\delta y)$, $(\chi+\delta\chi, y+\delta y)$, $(\chi+\delta\chi, y)$. The weights are specified regarding the distance from $(\chi r, yr)$. Consequently, assuming $w1$, $w2$, $w3$, and $w4$ are respectively the weights based on these distances, the value at $(\chi r, yr)$ is computed from an equation of the following form:

$$f(\chi r, yr) = w1 * f(\chi, y) + w2 * f(\chi, y+\delta y) + w3 * (\chi+\delta\chi, y+\delta y) + w4 * (\chi+\delta\chi, y) \tag{3.1}$$

Subfigure 3.1a shows a "source" injecting at position $(\chi s, ys)$ and values being scattered to the neighbouring grid-aligned positions. On the other hand, Figure 3.1b shows a gather operation where values from the neighbouring points are accumulated through bilinear interpolation to $(\chi r, yr)$. Similarly, in three-dimensional problems, we use tri-linear interpolation.

An example of sparse "off-the-grid" operators in a real-world application This paragraph will briefly showcase an example of sparse "off-the-grid" operators in the context of seismic imaging. Before focusing on the sparse "off-the-grid" operators, we quickly refer to seismic surveys and their importance.

Seismic surveys provide a way to capture detailed images of geological/subsurface structures. Using such images, we can interpret subsurface structure to determine the geological history and setting: essential considerations when prospecting for valuable minerals such as oil and gas. In seismic imaging, an airgun or explosives are used to produce pulses which reflect off and refract through layers in the subsurface, with the returned signal recorded using sensors called geophones (a process analogous to an ultrasound scan or sonar survey). Accurate imaging yields improved geological understanding, reducing the chances of unsuccessful exploratory drilling. Unsuccessful exploratory drilling may cost hundreds of thousands of dollars without any benefit. In land settings, sources and geophones are laid out in a grid arrangement or towed behind the vessel as a streamer at sea. These pulses are then picked up by the geophones attached to the lines towed by a ship. The information collected can subsequently be used for three-dimensional models of the subsurface. Geologists and geophysicists analyse these models to identify structures that imply the target resource's presence. As a result, high-quality imaging is critical for target identification and de-risking the drilling process.

Figure 3.2 shows a 2D slice of a 3D seismic imaging survey. Figure 3.3 shows the discretised computational domain of a 2D slice in a 3D seismic imaging survey. We can see the equally spaced points where we compute the PDE stencil updates and the sparse "off-the-grid" operators (sources and receivers) that are not aligned to the computational grid.

The following paragraph explains how we iterate over sparse operators and apply their effect on structured grid problems.

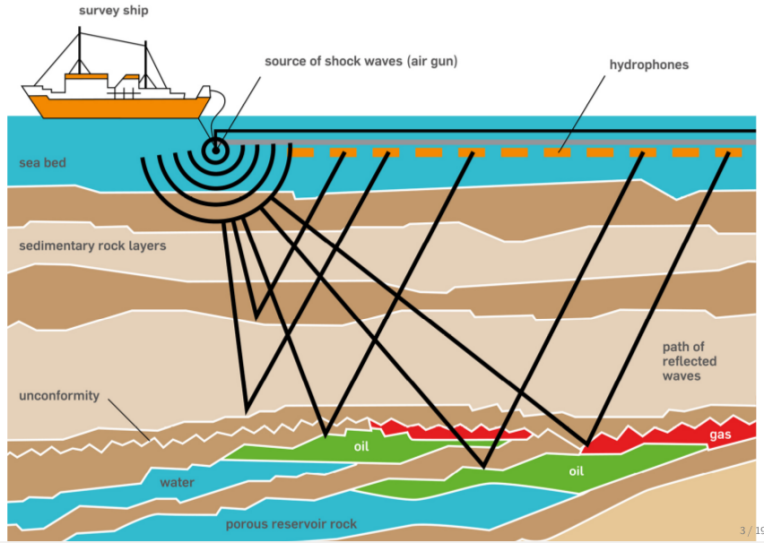


Figure 3.2.: A seismic imaging survey. A vessel's sources inject pulses, and geophones' receivers take measurements. Adapted from open.edu

.

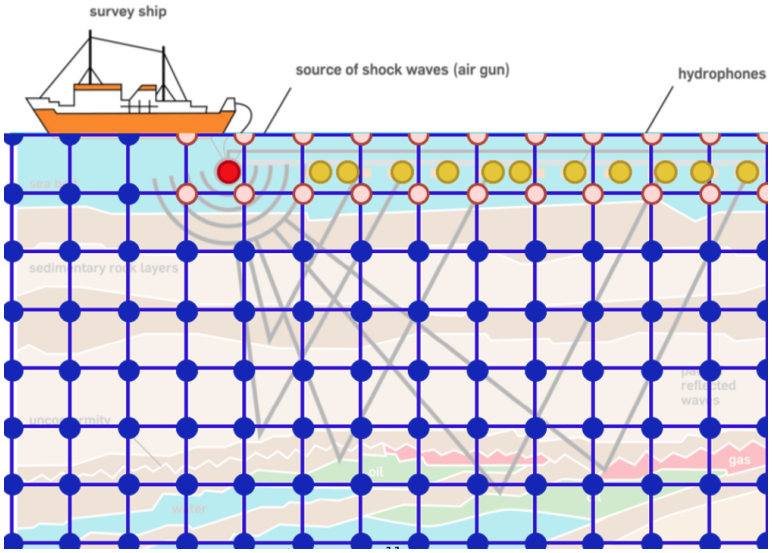


Figure 3.3.: Structured grid setup in a slice of a 3D seismic imaging survey. The geophones (red) and the receivers (yellow) are not aligned to the structured grid. Original background image: open.edu

Iterating over sparse operators Sources and receivers are both sets of sparsely-distributed off-the-grid points. We iterate over these sparsely-located sets through indirect accesses (A[B[i]]). Their effect is applied to the grid points just after iterating the three dimensions of the grid for each

timestep to update the points with the stencil kernel. Listing 2 illustrates a loop nest structure with source injection through indirection (see line 8). nt is the number of time steps; nx, ny, and nz are the number of grid points along the x, y, and z dimensions respectively, $A(t, x, y, z)$ is the stencil kernel update, and finally, so is the space discretisation order. The `src` is of size $nt \times len(sources)$ holding the wavefield for each timestep for every source, where *sources* is the structure holding the information for modelling source injection. Variable `np` accounts for the number of points affected by a source, and f is the function defining the type of interpolation (e.g., bilinear, trilinear). The `sources` set shown in Listing 2 provides the sparse off-the-grid coordinates for the injection. We iterate this set of coordinates that determine the affected neighbouring points. The wave amplitude is scattered to these affected points.

ALGORITHM 2: A typical time-stepping loop nest structure for a stencil update with source injection. This stencil has one temporal and three spatial dimensions.

```
1 for t = 1 to nt do
2   for x = 1 to nx do
3     for y = 1 to ny do
4       for z = 1 to nz do
5         A(t, x, y, z) ≡ u[t, x, y, z] = u[t-1, x, y, z] + Σ_{r=1}^{r=so/2} w_r ( u[t-1, x - r, y, z] + u[t-1, x
          + r, y, z] + u[t-1, x, y - r, z] + u[t-1, x, y + r, z] + u[t-1, x, y, z - r] + u[t-1, x, y, z + r]
          );
6   foreach s in sources do // For every source
7     for i = 1 to np do // Get the points affected
8       xs, ys, zs = map(s, i) // through indirection
9       u[t, xs, ys, zs] + = f(src(t, s)) // add their impact on the field
```

Note: Computing the source injection after the PDE stencil computation is an implementation option. We could very well apply source injection before the PDE stencil update with only minor changes in the "time" accesses of the code.

More complex data dependencies In Figures 3.1a, 3.1b and Listing 2 it can be observed that in addition to data dependencies stemming from sparse operators (see Figures 2.2, 2.3), practical applications such as seismic wave modelling introduce additional dependencies owing to the interpolation of data not directly associated with grid points into the model (e.g., scatter/gather operations for interpolation). These dependencies stem

from positions that are not aligned with the grid points. These positions are sparsely-distributed "off-the-grid" positions.

Applying classic time-tiling schemes to codes with "off-the-grid" operators without further adjustments would violate data dependencies. This violation will be explained in more detail in Section 3.1.2. When interacting with the space-time update patterns of temporal blocking schemes, these dependencies violate the correctness of our models as data dependencies are not honoured. To the best of our knowledge, there is no straightforward approach in the literature to solve this problem.

3.1.2. Problem overview: a running example

In this subsection, we aim to explain in simple terms why temporal blocking and sparse operators pose a challenging issue. Space blocking [Wolfe, 1989, Wolf and Lam, 1991] can be applied with no issues in computational patterns similar to Listing 2. The effect of sparse operators is applied after all the grid points of the domain have been updated for a specific time step. Consequently, separating the FD grid into blocks and computing grid point updates and "off-the-grid" operators within the same time step does not violate any data dependencies. As illustrated in Figure 3.4, we can first execute pairs of green-red or yellow-blue blocks and then apply the effect of sparse operators. Red diamonds indicate the "off-the-grid" coordinates of sparse operators. The red arrows that start from diamonds show the grid points affected by these sparse operators.

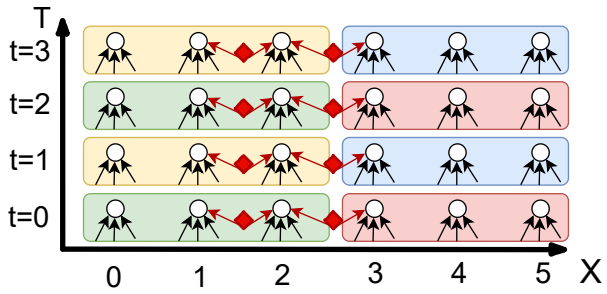


Figure 3.4.: Rectangular space blocking. All grid points can be updated in parallel at a specific time step. Sparse operators fit within the context of space blocking as their effect is imposed after all points have been updated for a specific time step. The existence of "off-the-grid" operators does not violate data dependencies in space blocking.

Applying temporal blocking to "pure stencil loops", i.e. problems with

only a PDE computation update and without the presence of sparse operators, has long been exercised. It requires manual and tedious work. However, it is valid, feasible and highly profitable for performance. Algorithm 3 shows the 2.5D blocking strategy followed by Devito. Pseudocode shows source injection alongside loop blocking, which is valid.

ALGORITHM 3: Algorithm shows the 2.5D blocking strategy that is followed from Devito.

```
1  for t to nt do
2      OpenMP parallelism - collapse(2)
3      for xblk to nx do
4          for yblk to ny do
5              for x to xblk do
6                  for y t0 yblk do
7                      # SIMD vectorisation
8                      for z = 1 to nz do
9                          A(t, x, y, z);
10                     for s in sources do
11                         for i = 1 to np do
12                             xs, ys, zs = map(s, i);
13                             u[t, xs, ys, zs] += f(src(t, s))
```

On the other hand, applying temporal blocking to stencils with sparse operators is challenging, as we illustrate with a 1-D example in Figure 3.5. In contrast to space blocking, temporal blocking is not an easy fit with "off-the-grid" operators.

Additional data dependencies are introduced when a sparse operator is encountered at an off-the-grid position. These data dependencies affect points that may often belong to different blocks in any spatial or temporal loop blocking variant. While this is not a problem for spatial blocking, as described earlier in this Section and shown in Figure 3.4, things are particularly problematic when it comes to applying temporal blocking.

The space-diagonals seen in Figure 3.5 cannot guarantee that the effect of sparse operators is applied at the space and time required for the affected grid points. Consequently, as we proceed in time, we often miss the effect of injections or interpolations, thus generating erroneous results that propagate within the computation.

In other words, the violation occurs because updates in space may pause for a particular time step, and computation will proceed in time rather than space. Consequently, a sparse operator update may be computed, and points that have not yet been updated through the stencil kernel updates

may be affected. Similarly, a point may be erroneously updated due to a forward move in time but may miss an injection from a neighbouring off-the-grid operator due to space-time block constraints. Similar violations emerge in other variants of temporal blocking, such as diamond temporal blocking [Bertolacci et al., 2015], [Malas et al., 2015]. All temporal blocking methods include space-time diagonals to honour the stencil update data dependencies; thus, there is no clear strategy for executing sparse operators.

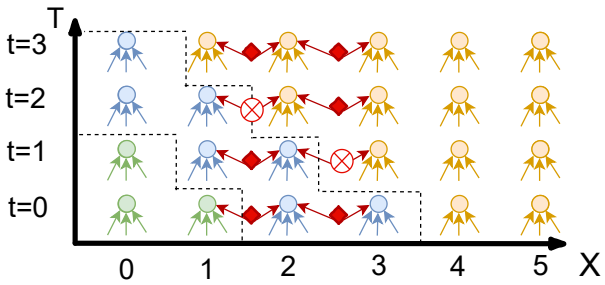


Figure 3.5.: Skewed/Wavefront temporal blocking. Grid points are updated in waves. During a wavefront update, we compute grid point values for multiple timesteps. Applying sparse operators in the space boundary may lead to erroneous updates since source injection may precede the stencil update for a particular timestep. We have a data dependency violation.

Because of the sparsity and the indirect nature of the accesses, the loops generated in Listing 2 by modelling source injection consisting of non-affine accesses (due to indirections, e.g. $A[B[i]]$) as illustrated in Listing 2.

Applying polyhedral transformations to stencil kernels with non-affine accesses is not straightforward. As a consequence, this is a problem for polyhedral tools as well. There have been works trying to tackle this challenge, as in Venkat et al. [2014b] where the authors present a framework to represent and transform computations with non-affine index arrays in loop bounds and subscripts via a new interface between compile-time and run-time abstractions. However, this model has yet to be inherited to simulation-level stencils to the best of our knowledge. There have also been other works such as [Strout et al., 2018, Rodríguez and Pouchet, 2018], however, primarily used in sparse linear algebra.

Polyhedral tools such as PLUTO [Bondhugula et al., 2008b, Bondhugula, 2013], Polly [Grosser et al., 2012b], Loopy [Klöckner, 2014], and CLooG [Bastoul, 2004] have indeed proved themselves to do noteworthy work in the stencil context only. They manage to deal with transformations

of the stencil update. However, they cannot deal with transforming the non-affine nature of the source injection loop nests. Additional machinery is needed to represent the sparse loops and the indirections and apply the transformations through inspector/executor schemes. The sparse off-the-grid nature of the source operator, combined with the non-affine nature of our loop structure, is blocking the application of time-tiling.

3.1.3. Contributions

We aim to overcome the prementioned limitations through our contributions in this chapter. The methodology presented in Section 3.2 is our approach in this direction. In summary, the contributions of our work in this chapter are:

- We propose a scheme that precomputes the off-the-grid sparse operators' effect, allowing to reorder the computations for FD wave propagators, thus enabling the application of temporal blocking to stencil codes consisting of sparse operators such as source injection and measurement interpolation. Our scheme is cost-efficient, adding a negligible overhead compared to the measured gains.

- We implement the scheme directly on top of the Devito DSL, harnessing the power of automated code generation, thus providing a pathway to express any similar operator in a form that exploits the benefits of time tiling with only minimal coding effort. In future work, we aim to deliver these optimisations as a fully automated workflow. This will be presented in Chapter 4.

- We evaluate our scheme using 3D stencils encountered in wave propagation applications (isotropic acoustic, isotropic elastic, and anisotropic acoustic (TTI)). Each stencil has varying memory and computation requirements.

- We achieve performance gains ranging from 15% to 60% for space orders 4 and 8 for isotropic acoustic and elastic and anisotropic acoustic, as well as 5% to 10% gains for elastic and TTI cases at space order 12.

3.1.4. Related work

This section will mainly discuss works on applying temporal blocking on stencil codes. We present related work on the history of temporal blocking, its variants, and benchmarking of temporal blocking codes. We primarily discuss works that have manually applied temporal blocking in standalone codes and benchmarks. This part will consist of works that apply temporal blocking without trying to automate it as a compiler pass. In Section 4.2, we will focus more on the efforts to automate temporal blocking as a transformation.

3.1.4.1. Temporal blocking for stencils

Temporal blocking as optimisation has long been applied to stencil kernels. Ideas around loop skewing to derive the wavefront method started more than 50 years ago Karp et al. [1967] and further consolidated later [Wolfe, 1986]. At that time, the wavefront method did not involve the tiling of iteration spaces until this was introduced a few years later from Wolfe [1989]. Two years later [Wolf and Lam, 1991], the idea of temporal blocking initially started as an extension of traditional loop skewing, from non-time-iterative to time-iterative loops. Tiling shapes were explored more, and the first adoption within a compiler happened.

Later on, Wonnacott [2004, 2000] was one of the first to generalise time skewing for multiprocessor architectures and multilevel caches. Guohua Jin et al. [2001] extended time skewing to a prismatic time skewing variant and showed substantial improvement for large-scale scientific applications.

Wellein et al. [2009] present a more efficient wavefront temporal blocking where wavefronts follow a pipelined fashion of execution. Strzodka et al. [2011] presented an algorithm to reduce multi-dimensional problems to one-dimensional wavefront traversals and to break memory-bandwidth limits for a few benchmarks. Yount [2015], Yount and Duran [2016] whose work on YASK has highly influenced this thesis, extended wavefront temporal blocking to be more effective for high-bandwidth memory caches, as in Intel Knights Landing. The optimisations presented include data layout transformations and enhanced vectorisation. YASK, formerly integrated with Devito, can automatically convert stencil code to an automatically temporally blocked implementation (discussed in Chapter 4). It has heavily

influenced this thesis through its work on wavefront temporal blocking [Yount and Duran, 2016]. However, YASK's temporal blocking does not support sparse "off-the-grid" operators. YASK was eventually removed from Devito due to several software engineering incompatibilities and challenges that undermined software maintenance. However, YASK's ideas and optimisations can be in the future roadmap of Devito without trying to reinvent the wheel.

Spatial and temporal reuse are often fused into hybrid models (equidistant locality) to harness the advantages of both methods Zohouri et al. [2018]. Apart from wavefront methods, there have been other temporal blocking variants, mostly named after the shape of the wave patterns.

One of the main reasons driving research for better tile shapes and sizes is the non-concurrent start-up of wavefront setups. In Bondhugula et al. [2008b], authors initially introduced diamond time-tiling for 1D stencils. Then Bandishti et al. [2012] discussed the load balance issue and presented variants to maximise parallelism and increase concurrency in tile computations. Malas et al. [2015] was the first to combine multicore wavefront temporal blocking and diamond tiling (MWD-TB). The new stencil update schemes showed significant reductions in memory pressure compared to existing approaches. Later on, Akbudak et al. [2020] applied MWD-TB to real-world seismic applications. Muranushi and Makino [2015] explored the state-of-the-art temporal blocking methods in temporal blocking and introduced a tiling with "parallelotopes in Tilted Cube Hierarchy". This scheme showed improved throughput over its predecessors. Later, Levchenko and Perepelkina [2017] introduced another diamond variant called DiamondTetris, aiming for more efficient use of SIMD parallelism and increasing the parallel execution levels. Yuan et al. [2019] introduced a new algorithm for "tesselating" star and box stencils aiming again to enhance data reuse. In other related works, [Grosser et al., 2014a,c] diamond-shaped tiles influenced the adoption of other shapes, such as 2D hexagons and 3d truncated octahedra.

Related work in Section 4.2 discusses frameworks that aim to automate temporal blocking as a loop transformation.

The contribution presented in this chapter aims to enable such schedules to be used in applications with off-the-grid operators. The rest of this chapter is organised as follows: Section 3.2 presents our methodology,

proposed algorithm, and approach to solve our problem. Section 3.3 introduces the mathematics and physics to model the wave-propagation kernels to be evaluated. Section 3.4 presents an experimental evaluation of the applicability and impact of the approach. Finally, in Section 3.5, we discuss and summarise our work and briefly refer to our plans for future work.

3.2. Methodology and implementation

This section describes our approach that enables temporal blocking for wave propagators with sparse operators. We describe the individual steps and present the details of our implementation. The whole precomputation workflow benefits from the power of the Devito DSL [Luporini et al., 2020, Louboutin et al., 2019] to automatically generate code and the data structures our scheme requires in its DSL. Afterwards, we manually transform the generated loops to implement wavefront temporal blocking (WTB) Wonnacott [2004], Ramanujam and Sadayappan [1991], Lamport [1974], a representative temporal blocking schedule.

The scheme presented in this section aims to precompute the source injection effect on the neighbouring affected grid points. It is modelled using the Devito API. The basic idea behind this scheme is to save the effect of sparse operators in data structures containing the necessary information to perform an equivalent computation using operations aligned to the grid points.

3.2.1. Source injection precomputation

A sparse operator performing source injection consists of the following parameters:

- The number of sources (a scalar number)
- The coordinates of each source in the 2D/3D space (a 2D/3D vector)
- The wavelet time series of each source (a $1xnt$ vector, where nt is the number of timesteps for the injected wavelet)

These parameters and interpolation type are enough to precompute their effect on the neighbouring grid points.

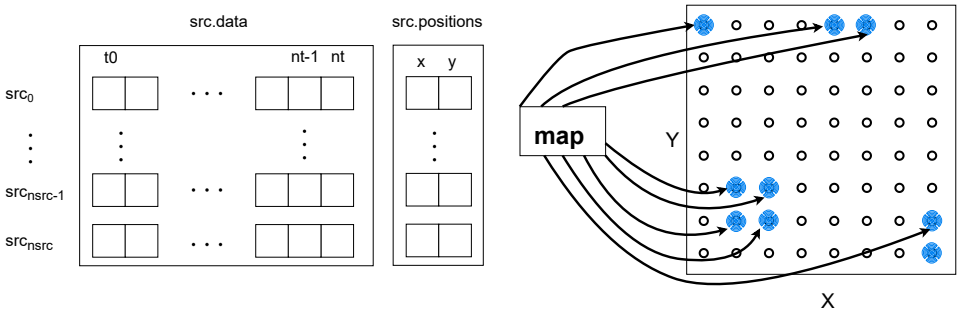


Figure 3.6.: Sources have, among others, data and coordinates. Through a mapping function, we inject the data into the grid points that are near the corresponding physical coordinates.

We are working under the assumption that the sources' coordinates are constant across our models' time domain though this may only sometimes happen. However, Devito's API could support the moving sources' case, and our algorithm is independent of it.

In order to inject the data of each source, we are using a mapping to match the grid points closest to the physical coordinates. Figure 3.6 shows the data *src.data* and the positions *src.positions*. The positions are mapped to the nearest grid points in the discretised domain to perform the source injection. This mapping process uses indirections of the form *A[B[i]]* as described in more detail in Section 3.2.1.1.

3.2.1.1. Iterate over the sources' coordinates and store the indices of affected points

Initially, we iterate over each source and inject it into a grid initialised with zero values for one timestep, assuming the wavefield amplitude is not zero at the first timestep. If the wavefield is zero at the first timestep, we may inject for more timesteps. The source wavefields used in the experiments have non-zero values at the first timesteps. The pseudocode for this "short" injection is presented in Algorithm 4. We use Devito to model and automatically generate code for this step. This scheme is independent of the injection and interpolation type (e.g., non-linear injection) because Devito automatically handles the mathematical difference of the interpolation type. The next step is to store the grid coordinates where values are non-zero after the injection.

ALGORITHM 4: Source injection to an empty grid. No PDE stencil update is happening.

```
1 for t = 1 to 2 do
2     foreach s in sources do
3         for i = 1 to np do
4             xs, ys, zs = map(s, i);
5             u[t, xs, ys, zs] + = f(src(t, s))
```

3.2.1.2. Generate sparse binary mask and unique IDs

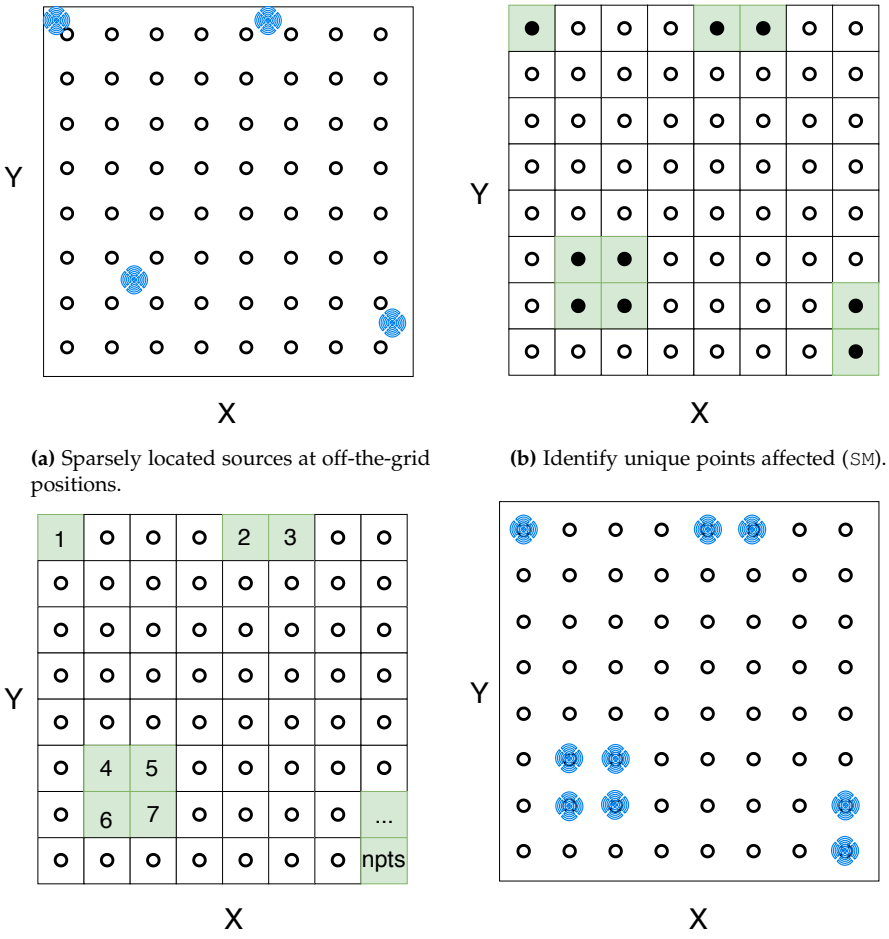
Using the non-zero indices, we populate two arrays. The first array (Fig. 3.7b) is a binary integer mask of our grid with ones at indices where u is non-zero. We call this array `SM`. Ones are shown as filled bullet circles with a green background (see Fig. 3.7b). The second one (see Fig. 3.7c) has the same shape and is populated with unique ascending values for each unique point affected, and we name it `SID`. It is common to encounter points affected by more than one source. Figures show an x-y plane (z-slice) of the 3D grid.

3.2.1.3. Decompose wavefields

Knowing the unique positions affected and their coordinates, we now use Devito's source injection mechanism to decompose the off-the-grid positioned wavefields to grid-aligned point wavefields. Using the `SID` structure, we perform an indirection and decomposition of the sources' wavefields to per-affected-point wavefields. This results in more source wavefields than the original ones. The pseudocode for that workflow is presented in Listing 5. `src_dcmp` holds now the data necessary for source injection, replacing `src` in our source injection computations. Instead of having sources at off-the-grid positions, as in Figure 3.7a, we now have new, decomposed sources aligned to the grid points, as in Figure 3.7d.

ALGORITHM 5: Decomposing the source injection wavefields.

```
1 for t = 1 to nt do
2     foreach s in sources do
3         for i = 1 to np do
4             xs, ys, zs = map(s, i);
5             src_dcmp[t, SID[xs, ys, zs]] + = f(src(t, s));
```



(a) Sparsely located sources at off-the-grid positions.

(b) Identify unique points affected (`SM`).

(c) Assign a unique ID to each of the `npts` affected points (`SID`).

(d) `npts` sources are now aligned with grid point coordinates.

Figure 3.7.: Illustration of the four steps through which source impact is aligned to the computational grid. The figures show an x-y plane slice of the 3D grid.

3.2.1.4. Fuse iteration spaces

Using the structure `src_dcmp`, which contains the necessary data to perform injection aligned to the grid points, we can fuse the source injection loop inside the kernel update iteration space. There is no loop over `sources` as sparse data can be expressed in 3D coordinates. We fuse the source injection loop at the same loop level as the stencil update `z` loop. The source mask `SM` acts as a binary mask and is used to add (if 1) or not (if 0) the source impact while `SID` is used to access the impact values indirectly as we iterate over the grid dimensions. The resulting loop

structure is illustrated in the following pseudocode in Listing 6 and offers SIMD vectorisation opportunities over the `z2` loop.

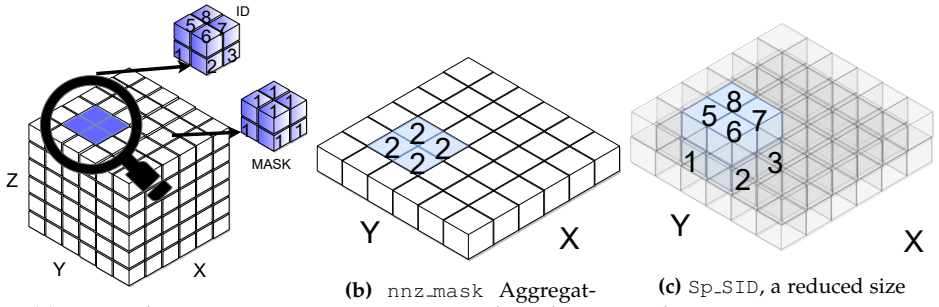
ALGORITHM 6: Stencil kernel update with fused source injection.

```
1 for t = 1 to nt do
2   for x = 1 to nx do
3     for y = 1 to ny do
4       for z = 1 to nz do
5         A(t, x, y, z, s);
6       for z2 = 1 to nz do
7         u[t, x, y, z2] + = SM[x, y, z2] * src_dcmp[t, SID[x, y, z2]];
```

3.2.1.5. Reducing the iteration space size

The 3D structures, introduced in 3.2.1.2 that are used to iterate through sources (`SM` and `SID`) in the `z2` loop are, in the general case, massively sparse. Multiplications by zero dominate the computations, accounting for a significant percentage of all operations. We should compute just the necessary iterations in the `z` dimension to alleviate this redundancy. Our approach for this is the following: We aggregate non-zero occurrences along the z-axis of `SM`, recording them in a structure named `nnz_mask`. Then, we reduce the size of `SID`, cutting off z-slices where all elements are zero. We name the new structure for the convention as `Sp_SID`. These structures reduce the iteration space size of `z2` to perform only the necessary computations. Listing 7 illustrates the pseudocode for the new iteration structure. The opportunity to reduce the iteration space applies to most problems in seismic wave propagators that we evaluated in our approach. The reason is that in the general case, sparse operations are located along a slice of the 3D domain, (near surface of the 3D domain, see Figure 3.2). Nevertheless, even for cases where this may not be true, we show that the benefits of this iteration space reduction are not limited (see Section 3.4.5).

The decomposition of the sparse sources' effect to aligned effect leads to more sources that naturally have less amplitude per source. Assuming the decomposition of a source as in Figure 3.1a, we see in Figure 3.9 a simple example of the amplitude of a 2D, 4 point decomposed non-aligned source along with the new and aligned sources.



(a) `SID` and `SM` are very sparse in the general case.

(b) `nnz_mask` Aggregating non-zero values along z-axis.

(c) `Sp_SID`, a reduced size `SID`.

Figure 3.8.: We aggregate non-zero occurrences along the z-axis, keeping count of them. The size of `SID` is reduced by cutting off z-slices where all elements are zero.

ALGORITHM 7: Stencil kernel computation with source injection. `nnz_mask[x][y]` helps to iterate over a reduced size iteration space for source injection.

```
1 for t = 1 to nt do
2   for x = 1 to nx do
3     for y = 1 to ny do
4       for z = 1 to nz do
5         A(t, x, y, z, s);
6       for z2 = 1 to nnz_mask[x][y] do
7         I(t, x, y, z) ≡{ zind = Sp_SID[x, y, z2];
8         u[t, x, y, z2] += src_dcmp[t, SID[x, y, zind]]; }
```

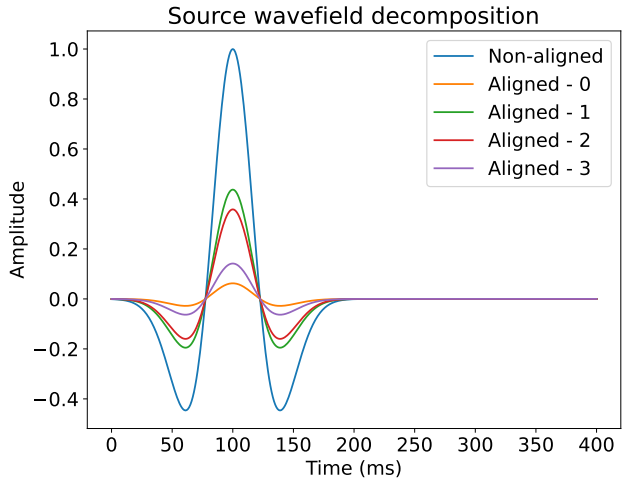


Figure 3.9.: The original non-aligned source wavelet and the aligned on the grid-point wavelets.

3.2.2. Applying wavefront temporal blocking

As a result of the work presented in the previous subsections, source injection is now precomputed. The data dependencies are now aligned to the grid points. Figure 3.10 shows the new data dependencies. The indirections are now hidden in the red diamonds.

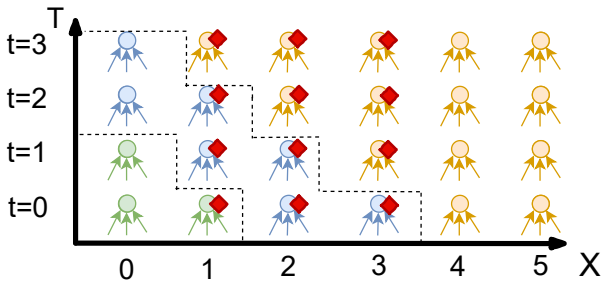


Figure 3.10.: Sparse operations are now aligned with the grid points. Data dependencies do not span anymore in different space-time tiles.

Since the problem of having data dependencies interfering with space-time updates is not present anymore, we can now apply wavefront temporal blocking over our stencil with source injection. Listing 7 illustrates code with manually applied temporal blocking. The automated application of this transformation is the main contribution of Chapter 4.

Applying temporal blocking is now feasible. We split the time-space iteration space into tiles, as shown in Figure 2.19a. Each tile is then partitioned into space blocks. By applying the transformations required from wavefront temporal blocking to Listing 6, we now have the loop structure in Listing 8. This structure is a time-tiled wavefront scheme computing a stencil kernel update with source injection.

The following Section 3.3 provides details about the evaluated kernels, their data dependencies, and their inherent loop structure.

3.3. Structure of wave-propagation kernels

To illustrate our technique, we selected three representative kernels implementing explicit FD methods for wave propagation. The chosen kernels significantly differ in the operational intensity and working set size Louboutin et al. [2017]. The kernels are implemented and validated in the

ALGORITHM 8: Algorithm shows the wavefront temporal loop blocking structure after applying our proposed scheme.

```
for t_tile in time_tiles do
  for xtile in xtiles do
    for ytile in ytiles do
      for t in t_tile do
        OpenMP parallelism - collapse(2)
        for xblk in xtile do
          for yblk in ytile do
            for x in xblk do
              for y in yblk do
                # SIMD vectorisation
                for z = 1 to nz do
                  A(t, x − time, y − time, z);
                for z2 = 1 to nnz_mask[x][y] do
                  I(t, x − time, y − time, z2);
```

Devito framework. The Devito compiler generates a C implementation for each kernel given a symbolic specification expressed with the Devito DSL.

3.3.1. Isotropic acoustic

The first equation we consider is the most straightforward and generally known wave equation in an anisotropic acoustic medium. This equation is a single scalar PDE with a Jacobi-like stencil. The acoustic wave equation for the square slowness m, defined as $m = \frac{1}{c^2}$, where c is the speed of sound in the given physical media, and a source q is given by:

$$\begin{cases} m(x)\frac{\partial^2 u(t,x)}{\partial t^2} - \Delta u(t,x) = \delta(x_s)q(t) \\ u(0,.) = \frac{\partial u(t,x)}{\partial t}(0,.) = 0 \\ d(t,.) = u(t,x_r). \end{cases} \tag{3.2}$$

where $u(t,x)$ is the pressure wavefield, x_s is the point source position, $q(t)$ is the source time signature, $d(t,.)$ is the measured data at positions x_r and $m(x)$ is the squared slowness. This equation can be written in only a few lines with the Devito symbolic API in Listing 3.1. We only show the equation definition part and where we omit the Grid, Function, TimeFunction constructions as shown earlier in Section 2.2.1:

The discretised acoustic wave equation is generally memory-bound due to the low operational intensity of the standard Laplacian [Louboutin et al., 2017, Williams et al., 2009].

Listing 3.1: Symbolic definition of the wave-equation

```
1 from devito import ... solve, Eq, Operator ...
2 ...
3 eq = m * u.dt2 - u.laplace
4 update = Eq(u.forward, solve(eq, u.forward))
5 ...
```

3.3.2. Anisotropic acoustic (TTI)

The second wave equation we consider is the most commonly used in industrial applications for subsurface imaging (RTM, FWI) [Zhang et al., 2011, Louboutin et al., 2018, Duveneck and Bakker, 2011, Alkhalifah, 2000, Bube et al., 2016] as it captures the effects of layered geological strata. This is a pseudo-acoustic anisotropic equation consisting of a coupled system of two scalar PDEs. Unlike the most simple acoustic isotropic equation, this formulation considers direction-dependent propagation speeds that translate into the discretised equation into a rotated anisotropic Laplacian. Such a kernel increases the operation count drastically [Louboutin et al., 2017]. For example, the first dimension x component of the Laplacian is defined as:

$$\begin{aligned} G_{\bar{x}\bar{x}} &= D_{\bar{x}}^T D_{\bar{x}} \\ D_{\bar{x}} &= \cos(\theta)\cos(\phi)\frac{\partial}{\partial x} + \cos(\theta)\sin(\phi)\frac{\partial}{\partial y} - \sin(\theta)\frac{\partial}{\partial z}. \end{aligned} \tag{3.3}$$

where θ is the (spatially dependent) tilt angle (rotation around z), and ϕ is the (spatially dependent) azimuth angle (rotation around y). A more detailed description of the physics and discretisation can be found in Zhang et al. [2011], Louboutin et al. [2018].

For a brief description, the resulting stencils in the TTI case are computed by updating two groups of fields having more complex data dependencies. After optimisation, the first group of stencil computes fields of temporaries depending upon single timestep values of TimeFunction fields while the second group of stencil computations uses those temporaries to more efficiently compute the TTI equation stencil updates.

Unlike what we saw in Figure 2.19b, the first group update does not need additional skewing as the updates do not depend on neighbouring

points of the previous group of fields. Thus, to give a more accurate representation of the dependencies of the TTI wave propagation stencils, we modify Figure 2.19b to Figure 3.11.

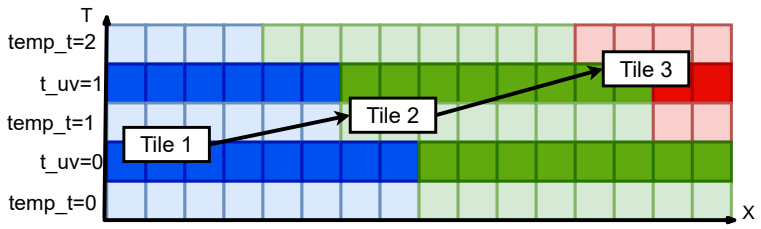


Figure 3.11.: The figure shows multiple wavefront tiles evaluated sequentially to describe TTI-like stencil updates. The additional data dependencies do not require additional skewing for the second set of loops.

An example of the stencil complexity of TTI wave propagation is shown in the Appendix Section A.1. More stencils need to be computed, and there is additional data movement along with computational intensity.

3.3.3. Isotropic elastic

Finally, we consider the isotropic elastic equation, which encapsulates complete elastic physics, supporting compressional and shear waves. Unlike the two previous acoustic approximations, this equation has two significant properties. First, this is a first-order system in time, which allows us to extend our work to a smaller range of local data dependency over time. Second-order in time systems require three buffers to store the necessary dependencies, while first-order systems only require one. Consequently, we demonstrate that the benefits of time-blocking and our implementation of it are not limited to a single pattern along the time dimension. Second, this equation is a coupled vectorial and a tensorial PDE system, which drastically increases the data movement. While the isotropic acoustic has one or two state parameters, the isotropic elastic has nine on the wavefield and contains non-scalar expressions of the source and receiver expressions that involve multiple wavefields.

The isotropic elastic wave-equation, parametrised by the Lamé parame-

ters λ, μ and the density ρ, is defined as Virieux [1986]:

$$\begin{aligned} \frac{1}{\rho}\frac{\partial v}{\partial t} &= \nabla.\tau \\ \frac{\partial \tau}{\partial t} &= \lambda \text{tr}(\nabla v) I + \mu(\nabla v + (\nabla v)^T) \end{aligned} \tag{3.4}$$

where v is a vector-valued function with one component per cartesian direction, and the stress τ is a symmetric second-order tensor-valued function.

The resulting stencil has cross-loop data dependencies, as we have two sets of loops that compute stencils depending on values computed at the other loop.

An example of the stencil complexity of the elastic wave propagation is shown in the Appendix Section A.2. The working set in the elastic wave propagation is higher than Acoustic and TTI and uses nine fields.

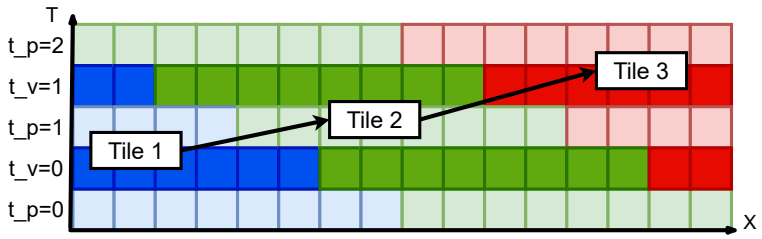


Figure 3.12.: The figure shows multiple wavefront tiles evaluated sequentially to describe elastic-like stencil updates. The additional data dependencies require additional skewing for the second set of loops.

An example of the stencil complexity of the Elastic wave propagation is shown in the Appendix Section A.2.

In the following section, we consider these three wave equations for varying spatial discretisation orders to verify and analyse our temporal blocking method.

3.4. Experimental Evaluation

We outline in Section 3.4.1 the experimental setup followed for performance evaluation. Section 3.4.2 discusses more details about the setup of the test cases for the physics discussed in Section 3.3. We aim to demonstrate the performance improvement achieved by our approach, illustrate its potential

for impact on key applications, and probe its applicability limits.

Evaluating the performance of stencil computations is a task that should be conducted carefully, as there are quite a few factors that may affect the performance of stencils. Therefore, we must detail the settings and practices followed for the conducted experiments. In the following paragraphs, we refer to a few crucial points when conducting benchmarking.

GPoints/s versus GFlops/s Often in the literature, we encounter works that benchmark their optimisations using the metric of GFlops/s. Although this is correct when comparing the performance of a cache-optimised versus a non-cache implementation, it makes it more challenging to compare across different implementations of stencils that describe the same mathematical equations. Therefore GFlops/s is not a safe metric to compare stencil implementations and draw conclusions about the time required to compute a stencil kernel. A characteristic example of such a misconception occurs when a mathematically non-optimised stencil is compared against a mathematically optimised stencil. Examples of non-optimised versus optimised mathematical kernels were shown in Listings 2.18 and 2.19 respectively and discussed with more detail in Section 2.4.1. In this case, a mathematically non-optimised stencil kernel may score higher GFlops/s than the mathematically optimised ones. However, the time-to-completion, the GPoints/s metric, may not be better than the mathematically optimised one. Such an example is shown in the roofline model in Figure 3.13.

Figure 3.13 shows a roofline with two benchmarked stencil kernels. The green circle is a kernel where all math and loop optimisations have been applied, while the red square is a kernel with only loop blocking applied without any other optimisations. We use a cache-aware roofline model on the left-hand side of the diagram, while on the right-hand side, we use the traditional roofline mode. The unoptimised kernel performs at around 82.46GFlops/s, requiring 10.14 seconds to complete. On the other hand, the optimised kernel stands lower in the roofline model with only 65.31 GFlops/s! However, it is more efficient as it completes execution in 9.03 secs. To summarise, we may have 26% more GFlops/s but 12% more time to the solution, which translates to higher resource costs.

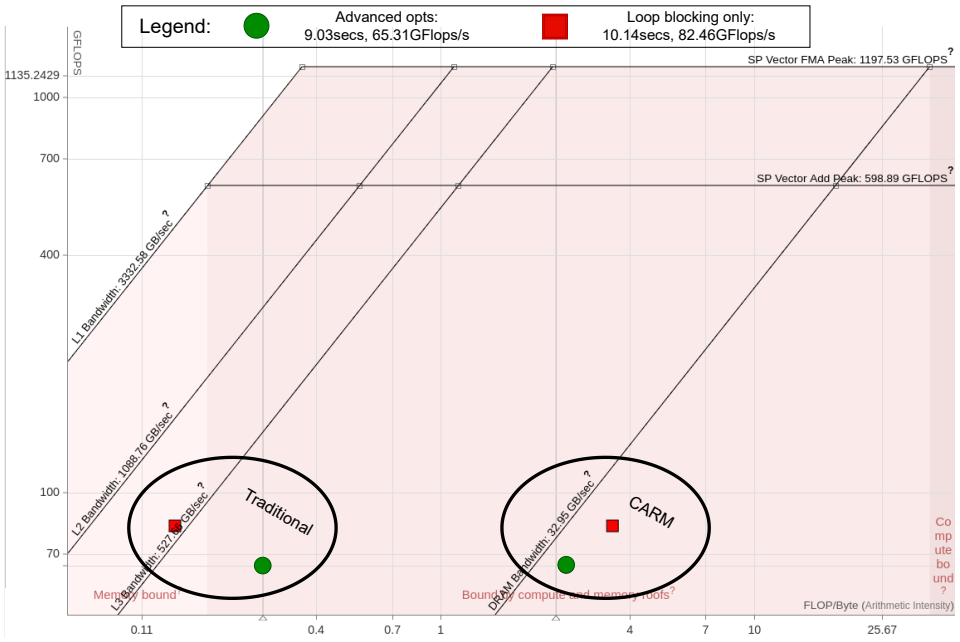


Figure 3.13.: Traditional (left-hand ellipsis) and cache-aware (right-hand ellipsis) roofline model. The unoptimised kernel stands higher in the roofline space. However, it takes a worse time-to-solution complete.

The impact of Devito optimisations in the kernel operational intensity
The table presented in this section shows the Devito compiler optimisations' impact on the kernels' operational intensity. This reduction is not work done in the frame of this thesis but is necessary to illustrate that temporal blocking is applied on highly-optimised code thanks to the work of Luporini et al. [2020], Louboutin et al. [2017].

Assuming we apply a polynomial regression model on the flops required before and after applying optimisations, we can notice that the flops for some discretisation order x for the Acoustic model are reduced from $y = 6x + 26$ to $y = 3.5x + 17$, the flops for some discretisation order x for the Elastic model are reduced from $y = 54x + 81$ to $y = 26.44x + 54.51$, and finally, the flops for some discretisation order x for the TTI model are (approximately) reduced from $y = 18x^2 + 92x + 63$ to $y = -0.28x^2 + 18x + 34$. It is evident that there is a big difference in flops required before and after applying the mathematical optimisations, thus yielding the need for much less data movement.

Floating point operations per stencil				
	SDO [a]	Unoptimised	Optimised	OI[b]
Acoustic	2	38	24	1.16
	4	50	31	1.51
	8	74	45	2.02
	12	98	59	2.44
	16	122	73	2.79
Elastic	2	189	103	1.04
	4	297	164	1.65
	8	513	268	2.59
	12	729	372	3.50
	16	945	476	4.30
TTI	2	NA	NA	NA
	4	743	102	1.70
	8	1999	164	3.44
	12	3831	223	5.33
	16	6239	282	7.64

Table 3.1.: Floating point operations per wave propagator stencil and space order

[a] Space discretisation order.
[b] Operational Intensity

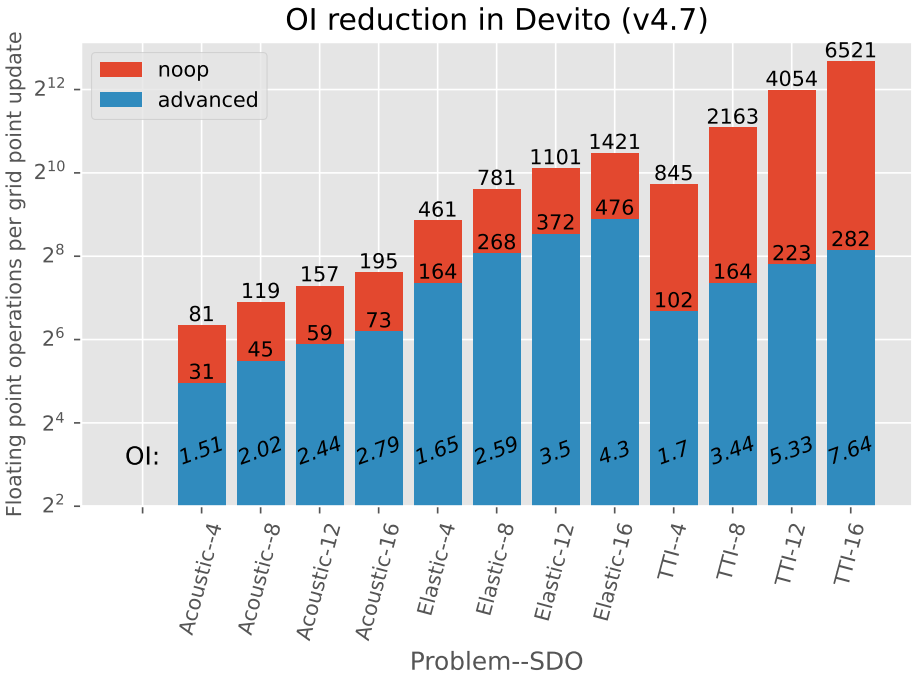


Figure 3.14.: Flop reduction in popular operators using the Devito optimisations

On the importance of optimisations working smoothly in tandem Stencil computations are subject to several optimisations. This thesis does not aim to explain these optimisations but only refers to their potential impact on a finite-difference kernel. These optimisations include, among others, mathematical optimisations to reduce the number of operations, cache-related optimisations to exploit locality, and shared- and distributed-memory parallelism. This paragraph showcases a simple example of why it is required to smoothly integrate all these optimisations and make the best out of each one. We often need to find a schedule for incrementally applying optimisations effectively in computational problems. Finding this schedule is essential for complex non-trivial computational codes and requires extra software engineering work to find the proper schedule. In the area of finite-difference stencils, heuristic schedules work well enough; however, this is a topic with lots of research (e.g. Milepost GCC in Fursin et al. [2011]) as we discussed in Section 2.4 with regards to the frameworks supporting a scheduling language. Applying temporal blocking to a stencil with optimised data movement can yield more significant gains than applying it to a non-optimised one. Temporal blocking performance gains are not benefiting the same memory-bound and compute-bound problems.

Figure 3.15 shows the throughput of an anisotropic acoustic wave-propagation kernel after incrementally applying optimisations. Cross-iteration redundancies elimination, common sub-expression elimination, parallelism and others progressively reduce the time needed to execute the kernel. Note: Time reduction for some optimisation passes is not indicative for other kernels and target platforms.

The order in which optimisations are applied is important, and their impact may differ depending on when applied to the kernel. An optimisation may unlock some space for other optimisations, or there are often cases where one optimisation can be applied twice. Figure 3.16 provides a brief example of this and uses a CARM to show the impact of temporal blocking when applied to a kernel with arithmetically reduced operations. For reference, we use a Laplacian stencil kernel with a space discretisation order of 4 on an i7-10700KF CPU (see characteristics of the platform in table 4.1). We reduce execution time to 20 seconds instead of 33 seconds by applying temporal blocking to a mathematically optimised kernel. A mathematically non-optimised kernel requires even more data movement and is even more

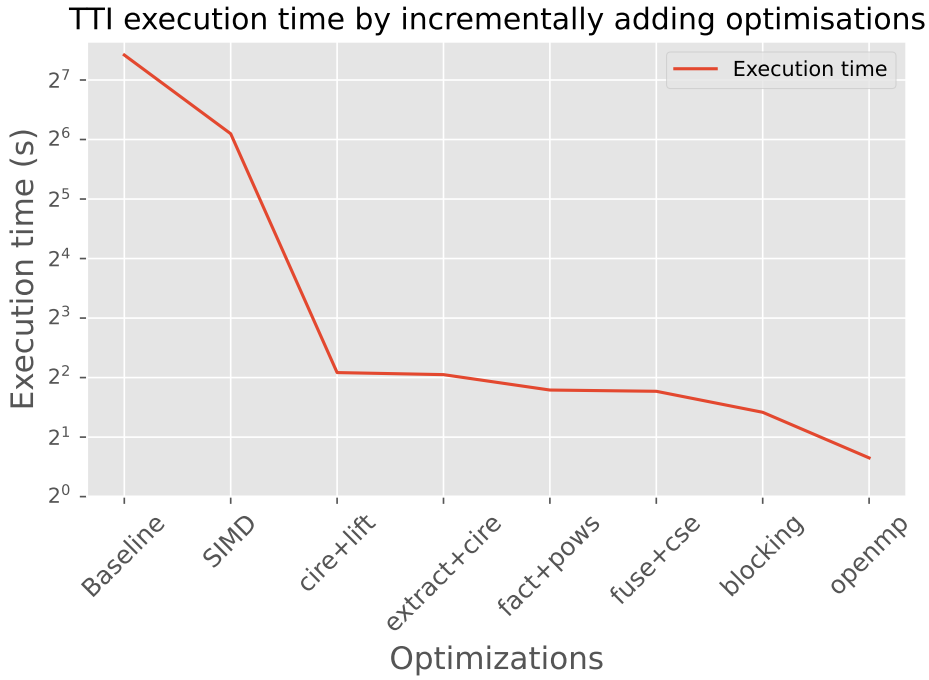


Figure 3.15.: TTI kernel execution time by incrementally adding optimisations to the baseline implementation. Optimisations are added as we move to the right-hand side. The figure shows the incrementally improved execution time in seconds. Evaluated on an i7-10700KF CPU (see characteristics of the platform in table 4.1). Note: Time reduction for some optimisation passes is not indicative for other kernels and target platforms.

memory-bandwidth bound than an optimised one. Temporal blocking has almost the same effect as space blocking, as seen with the square and triangle dots on the left-hand side. As cache-blocking optimisations help optimise for cache hits, we are more bound from the L3 cache roof on the left than on the right. This improvement is primarily due to having more space to get higher on the roofline model instead of being bounded by the L3 bandwidth.

Thread pinning The performance of stencil computations is heavily dependent on the cooperation of threads working in parallel to access memory locations and compute. These threads share cache resources and often compete when accessing data. Therefore the correct placement of threads within the execution of a problem is essential. We control thread pinning through environment variables. Erratic thread pinning affects the

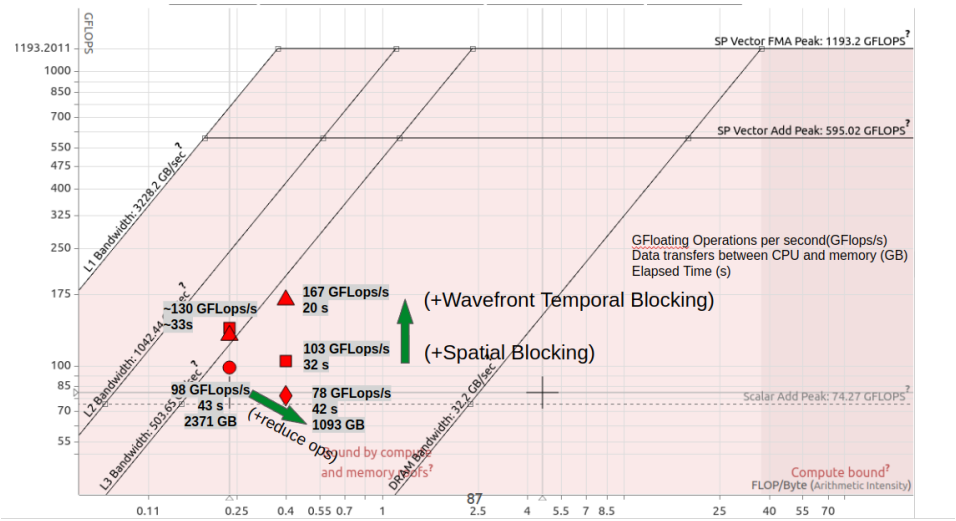


Figure 3.16.: Temporal blocking can be even more effective when combined with other optimisations. This roofline shows that temporal blocking can provide even higher performance with already reduced data movement. Evaluated on an i7-10700KF CPU (see characteristics of the platform in table 4.1).

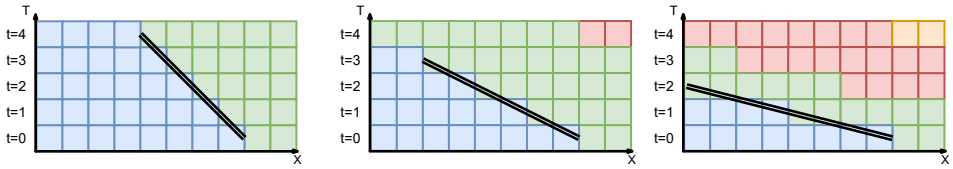
performance and the reproducibility of computational kernels. We pin thread to the physical threads of a core through the environment variables 'OMP_PROC_BIND=close, OMP_PLACES=threads' for the GCC compiler.

NUMA issues NUMA issues (see Paragraph 2.5) may emerge when running on CPUs with more than one NUMA region, and memory accesses may not be uniform. The distance a processor needs to access memory in a different NUMA region is bigger, leading to delays in accessing data. Thus, placing the data close to the processing units is essential. In our experiments, we use 'numactl –cpubind=0 –membind=0' to allocate the memory in the same partition as the cores running the simulation.

Why not compare against an MPI implementation One of the main limitations of this thesis is that the temporal blocking model presented is for shared-memory parallelism only. Temporal blocking performs better when cores access data stored in proximity to them. We only evaluate our model in one NUMA node. Extending execution to more than one NUMA region (see Paragraph 3.4) may only increase performance in a few rare cases. However, this is not a preferred practice when distributed-

memory parallelism (through MPI or others) implementation is available to our codes. Performance is expected to fall behind compared to MPI decomposition as we are only limited to single NUMA-node execution. Providing support for MPI and temporal blocking in tandem is a work in progress, and more details are provided in Section 5.2. Thus, we compare our contributions against highly efficient shared-memory parallel and vectorised implementations running in a single NUMA node. Domain decomposition techniques are only sometimes used in production. There are legacy codes that do not use domain decomposition. There are real-world problems that indeed fit in a single NUMA node. To conclude, our contribution here is helpful for legacy codes even though there is support for temporal blocking with MPI. This remains as future work.

Temporal blocking and high-order stencils Temporal blocking is an optimisation that has reduced performance gains for high-order stencils. By default, when the space order gets larger, the behaviour of temporal blocking converges to standard space blocking. Asymptotically if space order converges to $+\infty$, temporal blocking decomposes to standard loop blocking. As the stencil radius gets more extensive, the amount of data that can be reused in time is limited. Figure 3.17 depicts how the amount of data that can be computed by starting from the number of known points is reduced as space order increases. For example, the known cells of Figure 3.17a are enough to compute a larger area than the one in Figure 3.17b and a much larger one compared to 3.17c.



(a) Space discretisation order of 2 requires a skewing factor of 1

(b) Space discretisation order of 4 requires a skewing factor of 2

(c) Space discretisation order of 8 requires a skewing factor of 4

Figure 3.17.: High-order stencils limit the number of grid point updates that can be performed within wave for a number of time steps. Thus resulting in reduced temporal reuse. Spatio-Temporal waves are shown with different colours.

Most real-world applications at production give satisfactory results using a space discretisation order of 8. This thesis is another work that validates

the hypothesis that temporal blocking performance gains for high space orders are limited. However, the nature of the applications we work for can benefit even from lower gains. Gains for seismic imaging can be translated to thousands of dollars. A more helpful end goal is medical imaging applications, where dropping execution times could have a more significant impact on saving lives.

3.4.1. Compiler and system setup

To evaluate the optimised kernels, we used Virtual Machines in Microsoft Azure cloud computing services [1]. We evaluated two architectures: Intel® Xeon® Processor E5 v4 Family (formerly called Broadwell) and Intel® Xeon® Scalable Processors (formerly called Skylake 8171M). Access was granted on VMs (courtesy of Microsoft Azure), namely Standard_E16s_v3 and Standard_E32s_v3, running Ubuntu 18.04.4. The first system, E16s_v3, has an 8-core Intel Broadwell E5-2673 v4 single-socket CPU with AVX2 support. Each Intel Broadwell CPU has three cache levels: L1 (32KB) and L2 (256KB) caches private to each core and a 50MB L3 cache shared per socket. The second system has a 16-core IntelSkylake Platinum 8171M single-socket CPU with AVX512 support. Each Intel Skylake CPU has three cache levels: L1 (32KB) and L2 (1MB) caches private to each core and a 35.75MB L3 cache shared per socket.

The compilers used were GCC 7.5.0* and ICC 2021.1. We used OpenMP shared-memory parallelism with dynamic scheduling and SIMD vectorisation. Thread pinning was enabled using the environment variables OMP_PROC_BIND (for GCC) and KMP_AFFINITY (for ICC) running experiments on a single socket, single NUMA region, physical threads only. Experiments were built with Devito v.4.2.3. The experimentation frame-

[1] https://azure.microsoft.com/en-gb/

work and instructions on reproducibility are available in Section 3.4.6.

CPU characterstics		
Azure codename	E16s_v3	E32s_v3
CPU model	E5-2673 v4	Platinum 8171M
Architecture	Broadwell	Skylake
CPU(s)	16	32
Thread(s) per core:	2	2
Core(s) per socket:	8	16
Socket(s):	1	1
NUMA node(s):	1	1
L1d cache:	32KiB	32KiB
L1i cache:	32KiB	32KiB
L2 cache:	256KiB	1 MiB
L3 cache:	50MiB	35.75MiB
ISA	AVX2	AVX512F

3.4.2. Test case setup

We evaluate the performance of operators relevant to seismic imaging. We model three different wave propagation kernels: *isotropic acoustic*, *anisotropic acoustic (TTI)* and *isotropic elastic*. The isotropic acoustic and TTI wave equations are discretised with second order in time while isotropic elastic with first order in time, and we study varying space orders of 4, 8, and 12. Experiments were executed with 32-bit single precision. No 64-bit double precision experiments were executed. We use zero initial conditions and damping fields with absorbing boundary layers (ABCs) for all test cases. Waves are injected into the subsurface model using a time-dependent, spatially localised seismic source wavelet. We benchmark velocity models of 512^3 grid points, with a grid spacing of 10 meters for isotropic and elastic and 20 meters for TTI. Wave propagation is modelled in single precision for 512ms, resulting in 228 timesteps for isotropic acoustic, 436 for isotropic elastic, and 587 for anisotropic acoustic. The time-stepping interval is selected regarding the Courant-Friedrichs-Lewy

(CFL) condition Courant et al. [1967], ensuring the explicit time-stepping scheme's stability is determined by the subsurface model's highest velocity and the grid spacing.

3.4.3. Autotuning temporally blocked code

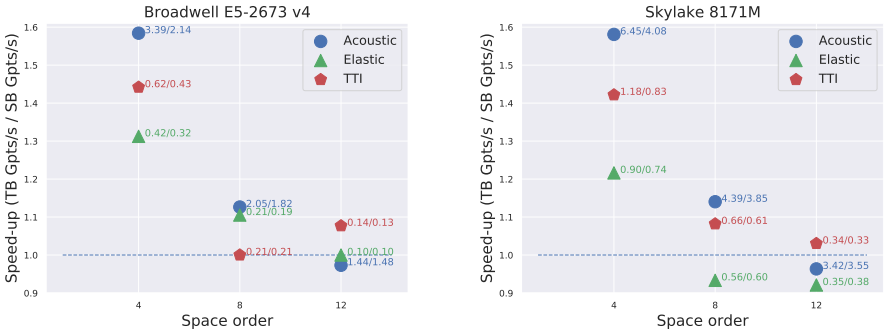
It should be noted that the parameter space for temporal blocking schemes is extensive. We report results obtained by following the guidance and experience from the state-of-the-art codes and literature Wittmann et al. [2010]. Simulation codes are hard to generalise in terms of performance as multiple configurations may be used from case to case. An operator's performance depends upon many factors, such as grid shape, discretisation space order, tile and block shapes, number of other fields, number of timesteps, platforms, and others. To tune our C code for the underlying hardware, we swept over the whole parameter space regarding our implementation's tile and block shapes. We compare the global performance maxima of both spatial and temporal blocking implementations. We executed our experiments using the best-performing tile and block sizes, ensuring a fair comparison versus Devito 's flop-optimised, aggressively tuned spatially-blocked and vectorised code. Algorithm 3 shows the algorithm corresponding to the blocking strategy used by Devito, where 2.5D blocking is automatically applied alongside flop reduction and mathematical optimisations. The best-performing tile sizes found after auto-tuning for temporal blocking are reported in Table 3.2.

	$tile_x, tile_y, block_x, block_y$	
Problem	Broadwell	Skylake
Acoustic O(2,4)	32, 32, 8, 8	64, 64, 8, 8
Acoustic O(2,8)	64, 64, 8, 8	64, 64, 8, 8
Acoustic O(2,12)	256, 256, 8, 8	128, 128, 8, 8
Elastic O(1,4)	32, 32, 8, 8	32, 32, 8, 8
Elastic O(1,8)	32, 32, 8, 8	64, 56, 8, 12
Elastic O(1,12)	256, 256, 8, 8	256, 256, 8, 8
TTI O(2,4)	40, 32, 4, 4	48, 48, 8, 8
TTI O(2,8)	32, 32, 8, 8	64, 64, 8, 8
TTI O(2,12)	256, 256, 8, 8	256, 256, 8, 8

Table 3.2.: Optimal tile-block shapes after tuning wavefront temporal blocking

3.4.4. Results discussion

Figure 3.18 illustrates the throughput (GPoints/s) speedup achieved for each of the three wave propagation kernels for space order discretisations of 4, 8, and 12 on two different machines (see Subfigures 3.18a and 3.18b). The evaluation shows speedup for space order four discretisation on both platforms. The acoustic wave propagation kernel benefits the most with around 1.6x, and TTI follows with around 1.44x. Elastic wave propagation is accelerated by 1.3x on Broadwell and 1.22x on Skylake. Concerning space order 8, a commonly used practice in industrial applications, we observe speedups of 1.13x or more for acoustic on both platforms, elastic on Broadwell and acoustic, and TTI on Skylake. No significant performance gains are observed for space order 12, excluding gains of around 5% on Broadwell with isotropic elastic and TTI.



(a) Throughput speedup of kernels for Broadwell.

(b) Throughput speedup of kernels for Skylake.

Figure 3.18.: Throughput speedup of temporal blocking kernels versus highly-optimised vectorised spatially-blocked code in Intel Xeon architectures, Broadwell and Skylake.

Figure 3.19 shows the isotropic acoustic kernels' roofline performance for the Broadwell microarchitecture. The roofline is a cache-aware roofline model representing cumulative (L1+L2+LLC+DRAM) traffic-based Arithmetic Intensity for application kernels [2]. We showcase improvement for the acoustic model breaking the ceiling of the L3 cache.

[2]https://software.intel.com/content/www/us/en/develop/articles/integrated-roofline-model-with-intel-advisor.html

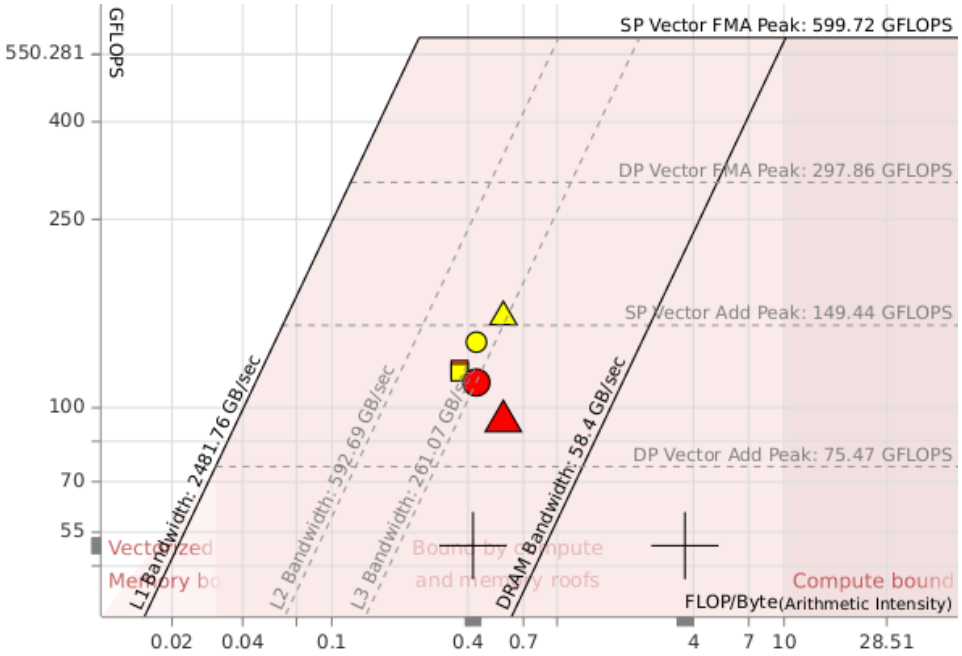


Figure 3.19.: Cache-aware roofline model on Broadwell for isotropic acoustic model space order 4 (triangles), 8 (circles), and 12 (squares). Red markers show the performance of spatially blocked vectorised kernels, while yellow ones show our temporal blocking scheme's performance.

3.4.5. Corner cases

Although our test cases use a single source, exploring how our model performs in the presence of more off-the-grid operators is interesting. Each source is decomposed into its surrounding grid points, so the overhead increases due to the number of initial sources and the number of grid points affected. We evaluate the overhead induced for two cases:

1. an increasing number of sparsely located sources: in this case, we have an increasing number of sources located at an x-y plane slice of the 3D grid, a scenario which can be of practical interest

2. an increasing number of sources densely and uniformly located all over the 3D grid.

Figure 3.20 shows that the increasing number of sources does not Faffect performance gains for the isotropic acoustic wave propagation. The higher the density of the located sources, the less the scheme benefits from dealing with the structure sparsity. Still, we observe gains of around 1.4x

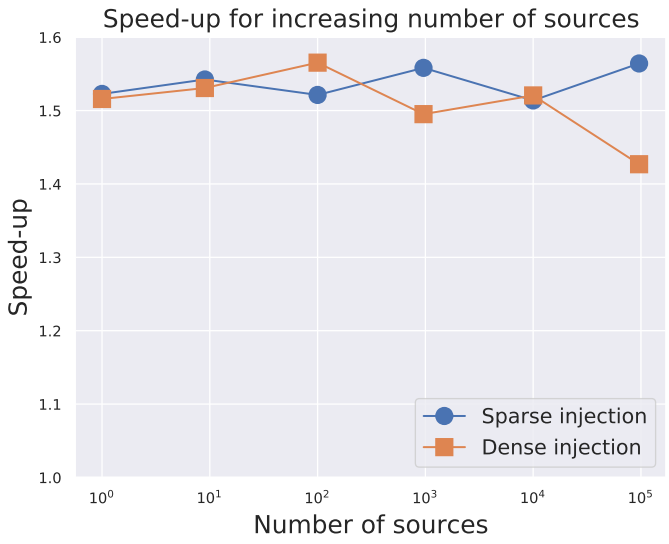


Figure 3.20.: Throughput speedup for an isotropic acoustic operator for space discretisational order 4 (four) over an increasing number of sparsely and densely located sources.

compared to 1.55x previously. We evaluate starting from 1 source up to 10^5 sources uniformly distributed at a z-slice within the 512^3-point grid. The higher density analogy of sources compared to grid points on the z-slice is $10^5/512^2 \approx 38\%$.

3.4.6. Code availability

The implementation of the methods described in this chapter is available in a Devito fork repository under the MIT open-source license available at DOI: 10.5281/zenodo.7472534. See the README.md for instructions on how to see the code used and reproduce the results in the chapter.

3.5. Conclusions and Future work

3.5.1. Conclusions

This chapter introduced a mechanism to enable temporal blocking in stencil computations involving sparse off-the-grid operators as encountered, for example, with sources and receivers in seismic inversion problems. These operators complicate the application of temporal blocking by scattering and gathering operations that interpolate values from and points in

structured grids. We overcame this limitation by proposing an approach to avoid interpolations and perform operations at every point. Owing to this contribution, we applied wavefront temporal blocking to wave propagators commonly used in practical applications. These kernels range from isotropic acoustic to more advanced, such as isotropic elastic and anisotropic acoustic (TTI). We evaluated the effect of temporal blocking on these kernels and compared them against highly-optimised, loop-blocked, vectorised implementations. This chapter's evaluation results are mainly motivated by the seismic imaging domain; however, the target applications are not limited to this scope. Experimental evaluation showed that the optimised kernels on Broadwell and Skylake microarchitectures showed compelling evidence of substantial acceleration of at least 1.5x for low and at least 1.1x for medium space order wave-propagation kernels.

3.5.2. Future work

Achieving performance improvement with high-space order kernels requires further research. Methods such as stencil retiming [Stock et al., 2014] have shown promise in alleviating this performance bottleneck, and a possible combination with temporal blocking may be promising. Another promising solution can be data layout transformations [Yount, 2015]. Near-term plans include evaluating our scheme on more diverse architectures (e.g., ARM) and accelerators (e.g., GPUs), mainly to explore whether different cache-memory hierarchies can help squeeze more performance out of this implementation. This work's next step and the natural consequence is the complete automation and integration in the Devito DSL and compiler framework [Luporini et al., 2020]. This work on automating the application of temporal blocking is discussed in the next Chapter 4. We aim to deliver automated, scalable optimisations on generated code beyond our kernels' current roofline performance limit.

Chapter 4

Automated temporal blocking through compiler technology

Temporal blocking is a performance optimisation targetting stencil kernels that aims to reuse data from the cache for multiple time steps. It can be expressed in various schemes, each scheme having intrinsic properties, advantages and disadvantages. Today a range of scientific applications, including computational fluid dynamics, seismic and medical imaging, image processing, and neural networks, could harness the benefits of temporal blocking, but they are not. There are several reasons:

1. Writing HPC code is challenging. Manually writing temporal blocking code is especially hard, tedious and error-prone. There needs to be more compiler technology to automate temporal blocking. The software engineering effort required to combine all the pieces to achieve automated temporal blocking from high-level symbolic abstractions is considerable.
2. Real-world applications often include complexities that pose additional challenges to applying temporal blocking compared to simpler stencil benchmarks.
3. Existing state-of-the-art technology to generate temporal blocking, such as the polyhedral model, cannot handle these complexities.

Our work aims to automatically generate temporal blocking code for a wide range of PDEs solved through the FD method and stemming

from practical applications. We aim to generate temporal blocking code automatically, starting from a sequence of symbolic equations as input from the user. We propose a compiler approach that, through compiler passes, helps to automatically generate wavefront temporal blocking code from a high level of abstraction. We present a compiler workflow to use temporal blocking in tandem with other stencil optimisations. Performance evaluation of the generated code shows benefits over highly-optimised vectorised spatially-blocked code. Our work is open-source and available online.

4.1. Introduction

This chapter focuses on generalising the concept of wavefront temporal blocking automation as a compiler pass. We present a strategy to integrate temporal blocking in the Devito compiler pipeline. As shown in Section 3.4, temporal blocking performs better when applied in tandem with other optimisations. Thus, we aim to accomplish the task of integrating it in a production-ready optimising compiler pipeline. We present a strategy for this task and show results that test and support the hypothesis of achieving improved performance alongside other optimisations. We designed a compiler pass within the Devito DSL and compiler framework based on Devito's IR. This compiler pass transforms the content of the IR. It helps to apply wavefront temporal blocking to commonly encountered stencil kernels and a variety of non-trivial industry-level simulation codes targetting CPUs. Consequently, we offer a complete compiler pipeline to automatically generate high-performing temporal blocking code from a high-level abstraction in the Devito DSL. Regarding GPUs, supporting kernels with sparse operators is currently a work in progress (see Section 4.7).

This chapter makes the following contributions:

- We present a generalised compiler pass that can operate on the IR of a compiler. Furthermore, it can effectively generate code optimised with wavefront temporal blocking (see Section 3.2.2).
- We implement this compiler pass in the Devito DSL and compiler framework, offering automated generation of temporal blocking code

from a high-level symbolic abstraction for a wide range of PDE-modelled simulations. To the best of our knowledge, this is the first approach to offering an end-to-end solution starting from symbolic maths to automatically generated HPC code with temporal blocking.

- We evaluate our scheme using low to high discretisation order 3D stencils encountered in computational fluid dynamics (heat diffusion) and wave propagation applications (isotropic acoustic and anisotropic acoustic (TTI)). The evaluated stencils have different memory and computation requirements, covering a wide range of cases.

- We explore the performance achieved on different applications with different characteristics and CPU architectures. We achieve performance gains ranging from 1.10x to 3.3x for a space discretisation order of 4 and up to 1.90x for a space discretisation order of 8 depending on the kernel and the hardware platform against highly optimised spatially blocked and vectorised code (2.5D blocking). We validate our hypothesis about the effective interaction of our method in tandem with the already existing optimisations.

- All contributions presented are open source and available online (hyperlinks available in Section 4.7.1).

4.2. Related work

This section discusses attempts to automate temporal blocking code generation for stencil computations. Most of the frameworks presented in Section 2.3.3 apply temporal blocking but not in an automated way. Here, we discuss approaches to integrating temporal blocking within larger frameworks and automating code generation for various simulation problems. We discuss and detail frameworks where their code generation either starts from a higher abstraction or may be source-to-source translators. Some of these frameworks have already been presented earlier in Section 2.4.2; however, the focus was mainly on their general capability to generate optimised code and less on their skillset concerning temporal blocking.

4.2.1. DSL and compiler frameworks supporting temporal blocking

Integrating temporal blocking within larger DSL and compiler frameworks is a challenging task. It requires not only significant software engineering effort but also interdisciplinary teamwork. This section discusses efforts from the literature that target the automation of temporal blocking code generation.

Holewinski et al. [2012] introduced automatic code generation for overlapped time-tiled code. The authors use a stencil-level DSL to model a complete stencil program targetting massively threaded GPU architectures.

SDSL [Henretty et al., 2013] is another stencil DSL to specify stencils and automatically generate code with locality and vectorisation. It supports several loop and data layout transformations to achieve locality and parallelism. SDSL compiler can support automatic code generation for hybrid space-time tiling. Later on, SDSLc [Rawat et al., 2015] extended the work of Henretty et al. [2013] to target GPUs and FPGAs.

YASK employs wavefront temporal blocking, mainly targeting Intel Knights Landing hardware. This hardware benefits from high-bandwidth memory, facilitating additional performance for problems that expose cache locality. YASK can automatically combine several blocking strategies, SIMD vectorisation, and advanced data layout transformations, namely vector folding [Yount, 2015]. YASK [Yount, 2015, Yount and Duran, 2016, Yount et al., 2017], formerly integrated with Devito as discussed in Section 3.1.4.1, can convert stencil code to an automatically temporally blocked implementation.

Bertolacci et al. [2015] tried to tackle the need for compiler technology for automating and tuning diamond temporal blocking. Instead of providing a fully automatic solution, they presented a library for parametrised time-tiling execution. Thanks to Chapel Parallel Iterators, the authors provide a solution to express the execution schedule and the complicated code-generation challenges automatically.

Bondhugula et al. [2017], introduced automated tiling transformations through the polyhedral model using a source-to-source translation through Pluto [Bondhugula et al., 2008b]. This work provided the conditions for concurrent tile start-up and evaluated diamond tiling over state-of-the-art

approaches. The methodology was based on the polyhedral model, and the target was computations with affine accesses only. As such, non-affine accesses of sparse "off-the-grid" operators cannot be supported in Pluto as they cannot be supported in other polyhedral frameworks.

Kuroda et al. [2017] proposed an automated directive-based compiler approach for temporal blocking. They extended the polyhedral compilation in the Polly/LLVM framework [Grosser et al., 2012a], with a loop transformation mechanism to introduce temporal blocking and automatically execute OpenMP parallel codes. This work presented a compiling toolchain to facilitate code transformations for temporal blocking and autotuning these codes.

STENCILGEN, introduced from Rawat et al. [2018], is another DSL to specify stencils from a higher-level abstraction. STENCILGEN's abstraction is at the stencil level. STENCILGEN targets code generation for GPUs, and among other optimisations such as GPU streaming, it supports overlapped temporal blocking. Rawat et al. [Rawat et al., 2018] additionally explore the relation of the "machine balance" parameter (peak machine DP-FP performance to peak global memory bandwidth) for achieving close-to-peak performance with stencils and discussing good practices to achieve efficiency in GPUs for stencil computations through a DSL.

Tanaka et al. [2018] introduced Formura DSL [Muranushi et al., 2016] to automate the generation of finite-difference stencils from a higher-level mathematic abstraction. Formura falls in the category of DSLs that support problem description from a higher-level symbolic mathematic abstraction (see Table 2.2). The authors automate temporal blocking code generation and evaluate their approach to many-core systems. In addition, Formura also supports MPI domain decomposition. The problems targeted are large-scale CFD simulations on systems based on the PEZY-SC2 many-core processor.

Matsumura et al. [2020] proposed AN5D to automatically transform and optimise stencil patterns in a given C source code. The source-to-source transformer automatically generates CUDA code from a C source code. This work targets GPUs. Spatial and temporal blocking optimisations are offered in tandem with other low-level optimisations. It can help generate code, offering novel optimisation strategies to reduce shared memory and register pressure compared to existing implementations. The AN5D

framework is publicly available.

4.3. Background

The aim of this section is twofold. First, it helps the reader better understand the Devito compiler and get more familiar with the lowering process from symbolic math to HPC code. Second, we provide the necessary introduction for the loop blocking machinery in Devito. This introduction will be beneficial for presenting our temporal blocking contributions in Section 4.4.

4.3.1. A deeper look in the Devito Compiler

Section 2.2.1 briefly introduced parts of the Devito compiler. This section includes a more detailed look at the Devito compiler and its IR layers. An overview of the Devito Compiler and its IR layers are shown in Figure 4.1.

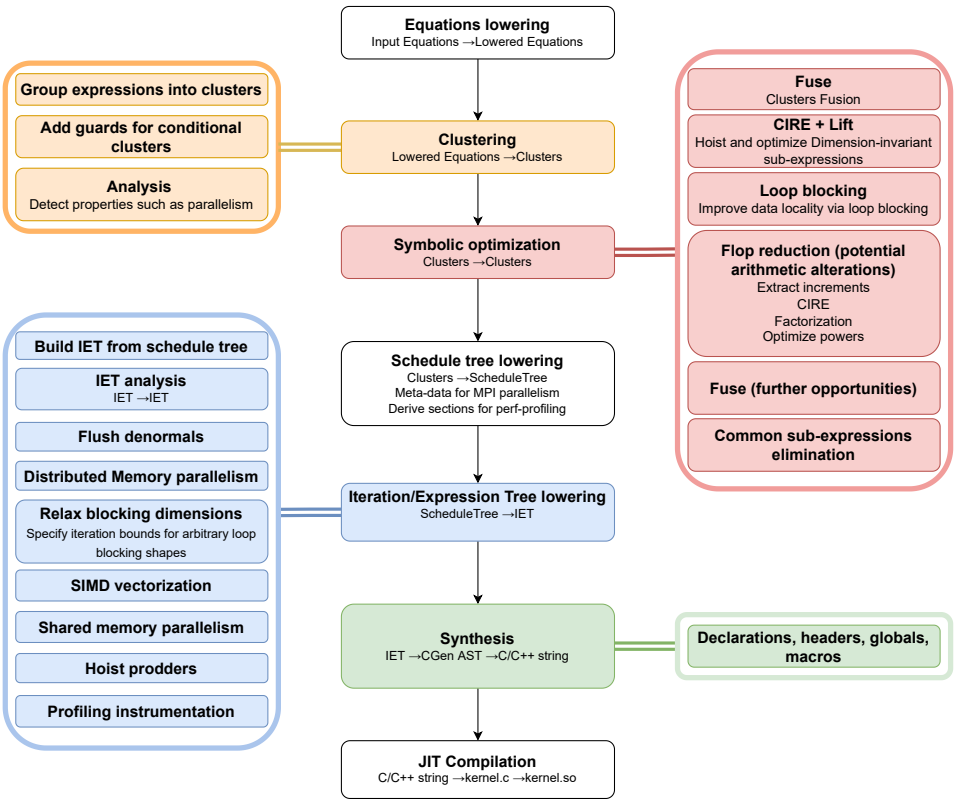


Figure 4.1.: An overview of the Devito DSL and compiler framework architecture.

More specifically, we will see how the compiler automates space blocking. Understanding this methodology is a prerequisite to discussing how we synthesise temporal blocking later.

4.3.1.1. Elements of the Intermediate Representation

In Section 2.2.1, we presented how we can use the rich API of the Devito DSL to express PDEs and introduce the essential API elements. In addition, this rich API can express scatter/gather operations, conditional execution of expressions, subdomains and others. In this section, we will examine the Devito IR's main elements. We look into how we use these IR elements to apply transformations through compilation passes. We start by presenting some early stages of the Devito compiler by briefly showing how the input mathematical expressions (see Section 2.2.1) are expressed when lowered to stencil expressions. To detail some of the IR parts, we will use Listing 4.1, which models a simple Laplacian-like kernel (see 2.1.3.1), as a running example to illustrate the lowering process within the compiler.

Listing 4.1: An example of a 3D Laplacian kernel using Devito

```
1 from devito import Grid, Eq, TimeFunction, Operator, Constant,
      solve
2 # Define variables for the problem setup
3 nx, ny, nz = 100, 100, 100
4 nu = .5
5 dx, dy, dz = 2./(nx - 1), dy = 2./(ny - 1), dz = 2./(nz - 1)
6 sigma = .25
7 dt = sigma * dx * dz * dy / nu
8 # Define grid and field
9 grid = Grid(shape=(nx, ny, nz))
10 u = TimeFunction(name='u', grid=grid, space_order=2)
11 # Initialise field data
12 u.data[:, :, :] = 0.2
13 # Create an equation with second-order derivatives
14 a = Constant(name='a')
15 eq = Eq(u.dt, a*u.laplace, subdomain=grid.interior)
16 stencil = solve(eq, u.forward)
17 eq0 = Eq(u.forward, stencil)
18 op = Operator(eq0)
19 op.apply(time_M=10, dt=dt, a=nu)
```

Derivatives and discretised expressions The input Equations from the symbolic DSL specification are given as arguments to the Operator in Listing 4.1. At the early stages of the compiler (see orange coloured "Clustering" area in Figure 4.1), the equations from Listing 4.1 are lowered to a Derivative representation as shown in Listing 4.2. This lowering happens with the help of SymPy Python package [Meurer et al., 2017].

Listing 4.2: The equations that were passed as arguments to the Operator are lowered to Derivative expressions

```
1 (Pdb) expressions
2 (Eq(u(t + dt, x, y, z), dt*(a*(Derivative(u(t, x, y, z), (x, 2)) +
      Derivative(u(t, x, y, z), (y, 2)) + Derivative(u(t, x, y, z),
      (z, 2))) + u(t, x, y, z)/dt)),)
```

Derivatives in Listing 4.2 are further lowered to their stencil form expression. After indexification and alignment to the computational domain, the stencils are lowered to a stencil representation as illustrated in Listing 4.3. In Listing 4.3, we see the first formulation of the discretised mathematical expression as a stencil.

Listing 4.3: Stencil-like expressions

```
1 (Pdb) expressions
2 [Eq(u[t + 1, x + 2, y + 2, z + 2], dt*(a*(u[t, x + 1, y + 2, z +
     2]/h_x**2 - 2.0*u[t, x + 2, y + 2, z + 2]/h_x**2 + u[t, x + 3,
      y + 2, z + 2]/h_x**2 + u[t, x + 2, y + 1, z + 2]/h_y**2 -
     2.0*u[t, x + 2, y + 2, z + 2]/h_y**2 + u[t, x + 2, y + 3, z +
     2]/h_y**2 + u[t, x + 2, y + 2, z + 1]/h_z**2 - 2.0*u[t, x + 2,
      y + 2, z + 2]/h_z**2 + u[t, x + 2, y + 2, z + 3]/h_z**2) + u[
     t, x + 2, y + 2, z + 2]/dt))]
```

It is noteworthy to mention that before they are lowered to a stencil representation, derivatives undergo preliminary optimisation for performance improvement. The linearity of finite differences is exploited to collect `Derivative` objects of the same type. For example, an expression similar to $u.dx.dx + u.dy.dx$ would typically lead to two temporaries, one for $u.dx$ and one for $u.dy$. Devito factorises the outer $.dx$ derivative, leading to $(u.dx + u.dy).dx$. A minimal example of this is shown in Listing 4.4 where we have the unoptimised initial derivative "expressions" and the optimised "processed" expressions.

Listing 4.4: Collecting derivatives to optimise expressions

```
1 (Pdb) expressions # initial -- unoptimised expressions
2 (Eq(u(t + dt, x, y),
3 dt*(Derivative(Derivative(u(t, x, y), x), x) + Derivative(
      Derivative(u(t, x, y), y), x) + u(t, x, y)/dt)),)
4 (Pdb) processed  # processed -- optimised expressions
5 [Eq(u(t + dt, x, y), u(t, x, y) + Derivative(dt*Derivative(u(t, x,
       y), x) + dt*Derivative(u(t, x, y), y), x))]
```

Cluster (of expressions) The expressions in Listing 4.3 are afterwards analysed for their data dependencies using built-in machinery based on a theoretic background presented in Lamport [1974]. First, they are grouped according to several properties and are part of a `Cluster` of expressions. Next, they are grouped according to the iteration space they belong. This part may not be more detailed in this thesis. This section will guide us through the `Cluster` as part of the Devito compiler.

The `Cluster` (of expressions) is one of the most critical parts of the Devito compiler. It primarily consists of a group, an ordered sequence of expressions. Listing 4.5 shows a Cluster holding a single expression. Later in this section, we will see clusters holding more than one expression and dive more into grouping expressions according to common properties.

Listing 4.5: A Devito Cluster. Here we only see a group of a single stencil expression.

```
1 (Pdb) clusters
2 (Cluster([Eq(u[t1, x + 2, y + 2, z + 2], dt*(a*(u[t0, x + 1, y +
      2, z + 2]/h_x**2 - 2.0*u[t0, x + 2, y + 2, z + 2]/h_x**2 + u[
      t0, x + 3, y + 2, z + 2]/h_x**2 + u[t0, x + 2, y + 1, z + 2]/
      h_y**2 - 2.0*u[t0, x + 2, y + 2, z + 2]/h_y**2 + u[t0, x + 2,
      y + 3, z + 2]/h_y**2 + u[t0, x + 2, y + 2, z + 1]/h_z**2 -
      2.0*u[t0, x + 2, y + 2, z + 2]/h_z**2 + u[t0, x + 2, y + 2, z
      + 3]/h_z**2) + u[t0, x + 2, y + 2, z + 2]/dt))]),)
```

We saw in the previous paragraph that the partial differential equations are initially discretised to finite-difference expressions. These expressions are then grouped based on their iteration space and data dependencies. The expressions in this sequence *share the same `IterationSpace`, same control flow and no dimension-carried "true" anti-dependencies among them [Luporini et al., 2020].* A `Cluster` , apart from the expressions, owns essential

metadata, such as the `IterationSpace`, the directions of the `Cluster` dimensions and the properties. Figure 4.2 shows a brief class diagram with the relationship between the `Cluster` and `IterationSpace` classes.

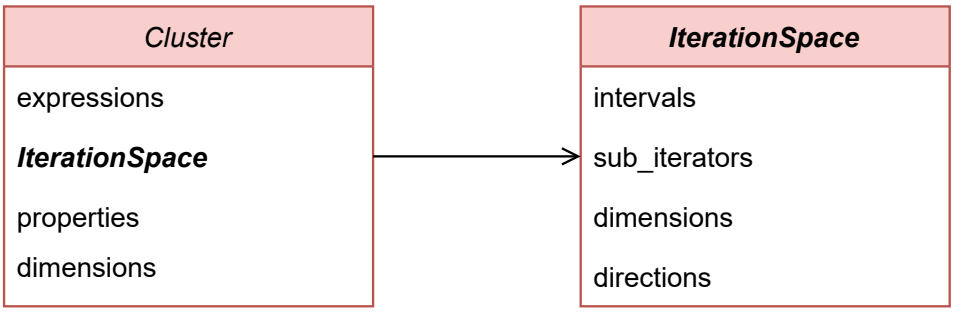


Figure 4.2.: A class Diagram to explain the relationship of a `Cluster` to an IterationSpace. Most of the optimisation passes rebuild an IterationSpace and then rebuild a `Cluster` since the IterationSpace is a property of the `Cluster` class.

The `IterationSpace` describes the iteration space in which the stencil belongs (i.e. the dimensions that are used as indices in the stencil expression). The properties are relevant computational properties (e.g. parallelism) produced when doing data dependency analysis. This analysis helps us build knowledge of properties such as parallelism and affinity. Directions indicate the direction in which this dimension is iterating (++ for forward or - - for backward). Listing 4.6 shows the information of a cluster of expressions. The info is accessed as class members.

Listing 4.6: A Cluster holds important information for the enclosing expressions. Some of them are ispace, dimensions and properties

```
1 (Pdb) clusters[0].dimensions
2 {y, t, z, x, time}
3 (Pdb) clusters[0].ispace
4 IterationSpace[time[0,0]++, x[0,0]++, y[0,0]++, z[0,0]++]
5 (Pdb) clusters[0].properties
6 <frozendict {time: {affine, sequential}, x: {affine, parallel}, y:
      {affine, parallel}, z: {affine, parallel}}>
7 (Pdb) clusters[0].directions
8 <frozendict {time: ++, x: ++, y: ++, z: ++}>
```

The dimensions show the accesses used in the expressions. We can also see the `IterationSpace`, the properties corresponding to each dimension of the `IterationSpace`, and the directions of every dimension. Later

on in the compilation, all this info will be used to construct loops and guide the correct placement of expressions within these loops. Listing 4.7 shows the most critical components of an `IterationSpace`. These are the "dimensions", the "relations", the "intervals," and the "subiterators".

Listing 4.7: The `IterationSpace` of a `Cluster` is holding important metadata for the expressions. The `IterationSpace` holds, among others, the dimensions, the relations, the intervals and the subiterators

```
1 (Pdb) clusters[0].ispace.dimensions
2 (time, x, y, z, t0, t, t1)
3 (Pdb) clusters[0].ispace.relations
4 {(t, x, y, z), (), (time, t), (time, x, y, z)}
5 (Pdb) clusters[0].ispace.intervals
6 IntervalGroup[time[0,0], x[0,0], y[0,0], z[0,0]]
7 (Pdb) clusters[0].ispace.sub_iterators
8 <frozendict {time: (t0, t1), x: (), y: (), z: ()}>
```

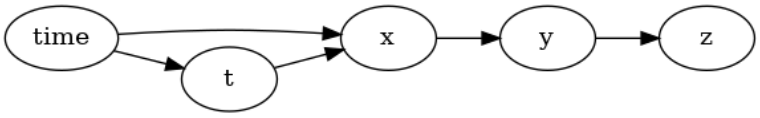
To define a valid ordering of a `Cluster`'s dimensions within an iteration space, metadata such as the *relations* define a partial ordering of the dimensions involved in a `Cluster`. The relations are extracted after parsing the order in which dimensions appear in the mathematical expressions. The relations are used as vertices, and the dimensions are used as edges to build an acyclic graph that explicitly defines the order in which dimensions should appear as loops when constructing the loop structure. Let us look at a simple example: Assuming the expression $Eq(u[x+1, y+1, z+1],\ 0.0)$, the initial relations are formed with a set of dimensions of the following ordering: (x, y, z).

By using the expression deriving from the running example (see Listing 4.1) used in this Chapter, assuming the discretised stencil expression is: $Eq(u[t+1, x+2, y+2, z+2], (-(-2.0 * u[t, x+2, y+2, z+2]/dt * *2 + u[t-1, x+2, y+2, z+2]/dt * *2)/vp[x+2, y+2, z+2] * *2...)..)$, The relations extracted from the dimension ordering of the above expression are: $(t, x, y, z), (x, y, z)$

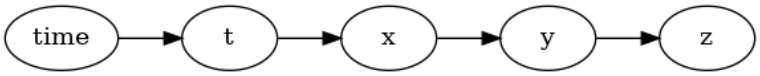
The *intervals* of the `IterationSpace` describe the dimensions used and any possible offsets that are added to the start and end bounds of a dimension. Finally, the *subiterators* show other additional variables that may be used to access indices. Subiterators are very niche in their use. A typical example of subiterators are $t0$ and $t1$ used along the time dimension

to implement the time buffering technique, (see Paragraph 2.2.1.1 for more details on time-buffering).

The cluster in Listing 4.7 has the following set of relations: (t, x, y, z), $()$, $(time, t)$, $(time, x, y, z)$. The DAG that is formed from these relations before the topological sorting is shown in Figure 4.3a. The result of the topological sorting is shown in Figure 4.3b.



(a) The Directed Acyclic Graph before topological sorting



(b) The Directed Acyclic Graph after topological sorting

Figure 4.3.: The Directed Acyclic Graph of relations shown in Listing 4.7 before and after being topologically sorted.

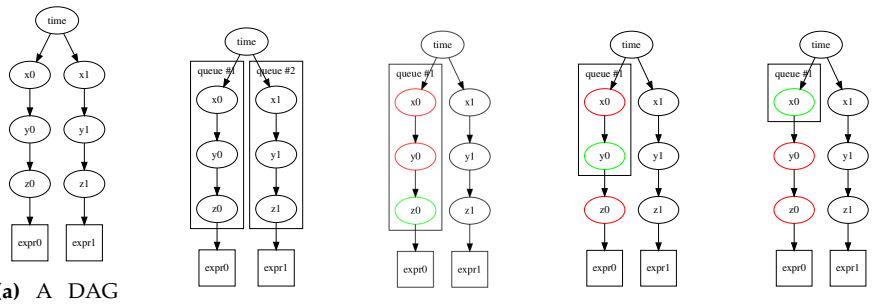
From this section, it becomes clear that a `Cluster` as a core component in the Devito intermediate representation holds essential metadata that can be used to apply arithmetic and loop optimisations. Usually, when applying an optimisation, we alter and reconstruct the `IterationSpace` dimensions, their computational "properties", and the "relations".

The optimisation passes in Devito that use Clusters as an optimisation target are often referred to as cluster-level optimisations or "cluster passes". Often in these passes, we optimise expressions according to the data from their `IterationSpace`. In the later Section 4.3.1.1, we will go through a simple example to show the differences in the information held from Clusters before and after applying an optimisation. The optimisation which will be used as an example is the Loop-Invariant code motion, a widely known optimisation from compiler textbooks [Muchnick, 1997, Appel, 2004]. This optimisation pass in Devito is called Dimension-invariant lifting. To visit a Cluster, we use a visiting pattern called "Queue," which we present in the next paragraph.

The Queue structure In this paragraph, we only briefly introduce the `Queue` structure to help the reader get more familiar with the visitor pattern used in most compiler optimisation passes that use Clusters as

subjects. Most optimisations in Devito are being applied using a `Queue` [1], a special data structure to visit and process Clusters based on a divide-and-conquer algorithm. A Queue is initialised with the dimensions of a Cluster.

Visiting Queues There are two ways to visit (iterate over) the dimensions of a Queue: the First-Divide-then-Apply strategy (FDTA) and the First-Apply-then-Divide strategy (FATD). By default, we use the First-Divide-then-Apply strategy, which we use as a running example in this paragraph. In each visit, we pop the last dimension and process the Cluster's metadata, such as the properties, the relations, etc. In the typical workflow, we visit each Cluster, and then we visit the dimensions of each `Cluster` in a "pop-visit" fashion. This "pop-visit" fashion is why this was named a `Queue`. We visit the dimensions, pop the last one, process and continue until no dimensions are left in the queue.



(a) A DAG with two mathematical expressions to compute

(b) The DAG is split in two queues susceptible to optimisations

(c) $z0$ is processed as the last element of the `Queue`

(d) $y0$ is processed as the last element of the `Queue`

(e) $x0$ is processed as the last element of the `Queue`

Figure 4.4.: Figure 4.4a shows a DAG consisting of two Clusters. Each cluster is iterated using a Queue, Figure 4.4b. These queues are used for optimisation using the First-Divide then Apply strategy. Figures 4.4c, 4.4d, 4.4e progressively show the dimension that is processed each time after being popped from the `Queue`. Figure progressively shows the visiting pattern of the first `Queue`

Assuming a problem where the computations end up in expressions computed in two different clusters, in different x, y, and z loops (let us use (i) x0, y0, z0 (first Cluster) and (ii) x1, y1, z1 (second Cluster)). A

[1] What is a `Queue` in Devito? For more information about a `Queue` class in Devito GitHub repository queue.py.

representative DAG of the structure of such a computation is shown in Figure 4.4a. This is a typical example structure encountered in elastic wave propagation (see Section 3.3.3). To express this better, this looks like two back-to-back stencils where multiple fields are being updated in a similar manner as in Algorithm 2. In this case, we have two clusters resulting in two queues of interest. The dimensions of these clusters form Queues and are considered optimisation candidates and can be seen within rectangles in Figure 4.4b. We start by iterating over the queue deriving from the first cluster. We starting from $z0$, then $y0$ and then $x0$. Then we continue to the next queue, the one deriving from the second cluster, starting from $z1$, then $y1$ and then $x1$.

Example: Loop-invariant code motion Loop-invariant code motion is a well-known optimisation often referred to as Dimension-invariant Lifting [Muchnick, 1997, Appel, 2004]. We use the same running example we used in this Section, so before applying this optimisation, we only have one `Cluster` as illustrated earlier in Listing 4.5.

We can see that there are mathematical expressions in the expression shown in Listing 4.5 that are invariant to the dimensions of the Cluster's `IterationSpace`. For example, computing $h_x * *2 (\equiv h_x^2)$ is invariant to the dimensions x, y, z. Consequently, this part of the expression can be computed in some temporary variable in a new (empty) `IterationSpace`. We now have two Clusters by lifting invariant computations, as the computations can be grouped in two different iteration spaces. After lifting the dimension-invariant expressions $h_x * *2$, $h_y * *2$, $h_z * *2$ out of their previous iteration space, we now have two clusters, as seen in Listing 4.8.

The first `Cluster` contains the lifted dimension-invariant computations, which now have an empty iteration space. The second `Cluster` keeps the same `IterationSpace` as before, using the newly created temporaries to compute the lifted dimension-invariant expressions.

4.3.2. Loop blocking in Devito

In this section, we will briefly present how Devito applies loop blocking. The theoretical background behind loop blocking optimisations was introduced in Section 2.3. Devito can automatically apply loop blocking

Listing 4.8: After dimension-invariant lifting, we now have two Clusters. The first one consists of dimension-invariant expressions and therefore has an empty `IterationSpace`. The second one has expressions accessing $time, x, y, z$. These dimensions form the second Cluster's `IterationSpace`.

```
(Pdb) len(clusters)
2
(Pdb) clusters[0]
Cluster([Eq(r0, 1/dt)
            Eq(r1, 1/(h_x*h_x))
            Eq(r2, 1/(h_y*h_y))
            Eq(r3, 1/(h_z*h_z))])
(Pdb) clusters[1]
Cluster([Eq(u[t1, x + 2, y + 2, z + 2], dt*(r0*u[t0, x + 2, y + 2,
     z + 2] + a*(r1*u[t0, x + 1, y + 2, z + 2] + r1*(-2.0*u[t0, x
    + 2, y + 2, z + 2]) + r1*u[t0, x + 3, y + 2, z + 2] + r2*u[t0,
     x + 2, y + 1, z + 2] + r2*(-2.0*u[t0, x + 2, y + 2, z + 2]) +
     r2*u[t0, x + 2, y + 3, z + 2] + r3*u[t0, x + 2, y + 2, z + 1]
     + r3*(-2.0*u[t0, x + 2, y + 2, z + 2]) + r3*u[t0, x + 2, y +
    2, z + 3]))))])
(Pdb) clusters[0].ispace
IterationSpace[]
(Pdb) clusters[1].ispace
IterationSpace[time[0,0]++, x[0,0]++, y[0,0]++, z[0,0]++]
```

optimisations as discussed in Section 2.4.1. This section details how Devito automatically applies loop blocking as a compiler pass in its optimisation pipeline. The same First-Divide then Apply visiting pattern is used to iterate over the dimensions of the clusters (see Figure 4.4). Similarly to other optimisations, the loop blocking implementation uses a Queue visiting pattern. The procedure is split into two parts, analysis and synthesis, which are detailed in the following Sections 4.3.2.1 and 4.3.2.2.

4.3.2.1. Loop blocking analysis

Loop blocking analysis is primarily based upon a cluster's iteration space properties. We primarily depend upon preliminary analysis of computational properties (e.g. parallelism) to decide whether a loop should be blocked. The Devito compiler performs data dependency analysis, computes the distance vectors of data dependences and determines these computational properties. These distances help assign properties to the clusters. Here, we refer only to those properties of interest to loop blocking analysis. Some of these properties are namely:

- SEQUENTIAL: A Dimension is sequential
- AFFINE: A Dimension used to index into tensor objects only through affine and regular accesses functions
- PARALLEL: A fully parallel Dimension

To derive the TILABLE property, that determines whether a loop should be optimised with loop blocking, a critical condition is distinguishing SEQUENTIAL and PARALLEL dimensions. If the PARALLEL condition is satisfied, we attach the TILABLE property to the dimension of this cluster. Users can use more heuristics to drive the tiling parameters that are not detailed here. Heuristics affect tiling strategies and could improve performance. By default, Devito uses heuristics empirically known to deliver performance. Some of the rules to determine whether a cluster's dimension is suitable for tiling are:

1. It satisfies the PARALLEL and AFFINE properties. These properties have been attached after preliminary data dependency analysis. Note: Conceptually, parallelism is a sufficient but not necessary condition for applying cache-blocking optimisations.
2. It is not innermost. Blocking the innermost dimensions is also possible. However, we prefer vectorising them, as it usually delivers better performance. (Heuristic)
3. It is within a SEQUENTIAL dimension. Dimensions in Clusters that are not within a SEQUENTIAL dimension (e.g. GEMM kernels) may also be blocked; however, we now focus on time-iterative computational kernels. (Empirical heuristic)

Algorithm 9 presents the pseudocode showing a part of how the loop blocking analysis is implemented as a pass in the Devito compiler. Algorithm 9 shows how we attach the TILABLE property after checking the necessary conditions for this. Note that the pseudocode and the description of the loop blocking analysis are only a simplified sketch of the complete algorithm used in Devito. We aim to refer only to the critical points necessary to understand this functionality. (For the full code see footnote [2]).

[2] The complete algorithm is included in v4.6.2/devito/passes/clusters/blocking.py.

ALGORITHM 9: Pseudocode for part of loop blocking analysis. Heuristic options are marked with 'H:' in the comments

Input: clusters, prefix
Output: processed

```
1 d = prefix[-1].dim
2 for c in clusters do
    // PARALLEL* and AFFINE are necessary conditions
3   if not PARALLEL* and AFFINE in c.properties[d] then
4     return clusters
    // H:Innermost Dimensions may be ruled out a-priori
5   if not d is c.ispace[-1].dim then
6     return clusters
    // H:TILABLE not worth it if not within a SEQUENTIAL Dimension
7   if not any ( SEQUENTIAL in in c.properties[d] for i in prefix[:-1] ) then
8     return clusters
  // All good, `d` is actually TILABLE
9 processed = attach_property(clusters, d, TILABLE) return processed
```

After completing the blocking analysis, we proceed to the loop blocking synthesis, presented in Section 4.3.2.2.

4.3.2.2. Loop blocking synthesis

To synthesise blocking, we iterate in a bottom-to-top fashion over the clusters' dimensions using a `Queue`. Remember that the clusters have been reconstructed in the loop blocking analysis phase (see Section 4.3.2.1). We only apply loop blocking to a dimension if this has the TILABLE property. The TILABLE property, if applicable, was attached during the analysis part (see Algorithm 9).

The loop blocking pass can support multiple levels of blocking. By default, it generates a single level of loop blocking per dimension. However, the user can optionally control heuristics to decide the blocking strategy to be followed. For example, Listing 4.9 shows how a user can control temporal blocking options using arguments through an Operator at the high level DSL.

Additionally, it shows how a user can specify several parameters to tune blocking code generation heuristically. For example, if a user prefers to block an innermost loop, an argument can be used to apply this heuristic (see line 6 of Listing 4.9).

Algorithm 10 shows the pseudocode with how loop blocking synthesis is implemented as a pass in Devito (see footnote 2).

Listing 4.9: Users can drive the blocking optimisation strategy by simply tweaking a few parameters.

```
# The default optimisation pipeline enables standard loop blocking
op = Operator(eq0, opt={"advanced"})
# Additionally, ask for two levels of loop blocking
op = Operator(eq0, opt={"advanced", {'blocklevels': 2}})
# Additionally, block the inner loop
op = Operator(eq0, opt={"advanced", {'blockinner': True}})
# Additionally, for two levels and inner loop blocking
op = Operator(eq0, opt={"advanced", {'blockinner': True, '
    blocklevels': 2}})
```

ALGORITHM 10: Pseudocode for part of loop blocking synthesis.

Input: clusters, prefix, levels
Output: processed

```
d = prefix[-1].dim
// Return if no TILABLE dimensions exist
if not any(TILABLE in c.properties[d] for c in clusters) then
    return clusters
block_dims = generate_block_dims(d, levels)
for c in clusters do
    if TILABLE in c.properties[d] then
        ispace = decompose(c.ispace, d, block_dims)
        new_c = c.rebuild(ispace)
        processed.append(new_c)
    else
        processed.append(c)
```

When a 'TILABLE' dimension is encountered, we generate a new iteration space and, therefore, a new `Cluster`. To construct a new `IterationSpace` we call a specific routine to decompose the old iteration space to a new one. Algorithm 11 shows the steps within the larger context of an iteration space decomposition.

More specifically, we take the following steps for loop blocking synthesis:

1. We construct n new Dimensions, with their respective bounds, if these dimensions are considered TILABLE (see Algorithm 10).
2. We generate a new set of `IterationSpace` intervals with the newly created Dimensions (see Algorithm 11).
3. We generate a new set of relations to maintain a valid topological order of the old and new dimensions (see Algorithm 11).
4. A new `IterationSpace` is constructed using the previously gener-

ALGORITHM 11: Pseudocode showing the steps for `IterationSpace` decomposition.

Input: ispace, d, block_dims
Output: ispace

```
- Step 1: Create the new Intervals
- Step 2: Create the new relations
— Step 2a: Add relation 'bbd > bd > d'
— Step 2b: Suitably replace 'd' with all 'bd's
— Step 2c: Make sure BlockDimensions at the same depth stick next to each other
// E.g., `(t, xbb, ybb, xb, yb, x, y)`, and NOT e.g.  `(t, xbb,
   xb, x, ybb, ...)`
- Step 3: Update intervals
- Step 4: Update sub_iterators
- Step 5: Update directions
return IterationSpace(intervals, sub_iterators, directions)
```

ated relations and intervals. (see Figure 4.2, Listing 4.11 and Algorithm 11)

More specifically, n new *BlockDimensions* are created, where n can be given through the user API and by default, it is one. Each block dimension has its minimum and maximum bound and a symbolic step, usually equal to one. A dimension that identifies itself as BlockDimension does also have a symbolic step. Unlike the general case, where a symbolic step is usually one (unit increment), the symbolic step of BlockDimensions is a symbol. This symbol is a variable that can be used to parametrically tune the blocked dimensions step in the compiler's later stages. When creating a new Dimension, several meta-data must be taken care of, such as the `IterationSpace` , the relations, and sub-iterators. Care is taken for the newly constructed dimensions to inherit valid properties and metadata.

Now we will show how the Algorithm 11 incrementally adds relations when space blocking is applied: The input to the ispace decomposition consists of the newly created BlockDimensions, the 'd' dimension that is about to be replaced and the current ispace. We initially append the intervals and incrementally add the new relations that emerge from the new BlockDimensions. We form the new relations in the following way:

1. block_dims form a graph path '$bbd > bd > d$'

2. Each Dimension 'd' is suitably replaced with all Blocked Dimensions 'bd''s in the already existing relations by avoiding, e.g. $x > yb$

3. BlockDimensions at same depth stick next to each other, e.g., $(t, xbb, ybb, xb, yb, x, y)$, and NOT, e.g. $(t, xbb, xb, x, ybb, ...)$. This could occur when replacements from the previous step lead to dimensions of a lower level appearing higher than ones with higher levels and is achieved by dropping relations that violate this.

In Listing 4.10, we use the running example of Chapter 4.1, where blocking happens for two dimensions (x and y) to showcase how we incrementally form the new relations that are necessary for loop blocking. We showcase only the newly added relations for every algorithm step and how they evolve. The complete code of the algorithm can be seen in footnote 2.

After we have found the new relations, we reconstrct the ispace. Listing 4.11 shows the iteration space before and after the loop blocking pass. We can see the new iteration space that has two additional dimensions, the $x0\_blk0[0,0]$ and $y0\_blk0[0,0]$.

4.3.2.3. The issue of main and remainder areas

The new dimensions progressively inherit loop bounds from their parent dimension. By parent dimension, we refer to the dimension that led to the birth of a new blocking level of a dimension. For example, x is the parent of $x0\_blk0$, and $x0\_blk0$ is the parent of $x0\_blk1$. We often use the term "root" to refer to the origin of a dimension. E.g. x is also the root of both $x0\_blk0$ and $x0\_blk1$. As an example, we can see how the loop bounds look as soon as we generate the BlockDimensions. For example, the new Dimension $x0\_blk0$ inherits the minimum and maximum bounds of x and has a new increment step, namely $x0\_blk0\_size$, and has x both as a parent and a root. The next one, $x0\_blk1$ which iterates within an increment step of $x0\_blk0$ has $x0\_blk0$ as the minimum and $x0\_blk0 + x0\_blk0\_size - 1$ as maximum bounds, x is the root $x0\_blk0$ is the parent. Its increment step is namely $x1\_blk0\_size$, and $x0\_blk0$ is the parent of $x0\_blk1$.

However, the new loop bounds created (see Listing 4.12) are not perfect divisors of the whole extent of the computation domain. As in the new Iteration spaces that are created may not cover the whole computation domain. We call this the main-remainder issue. One of the main challenges when generating blocking dimensions is defining their loop bounds. The

Listing 4.10: New BlockDimensions require updating the relations in the IterationSpace.

```
(Pdb) ispace.relations
{(t, x, y, z), (time, t), (), (time, x, y, z)}
(Pdb) block_dims
[y0_blk0, y]
(Pdb) print(d)
y
---> Form a graph path from the existing block_dims
(Pdb) relations
[(y0_blk0, y)]
---> Suitably replace `d` with all `bd`s in the already existing
    relations and avoid, e.g. `x > yb`
(Pdb) relations
[..., (t, x, y0_blk0, z), ... (time, x, y0_blk0, z), ...)]
---> BlockDimensions at the same depth stick next to each other, E
    .g., `(t, xbb, ybb, xb, yb, x, y)`, and NOT e.g. `(t, xbb, xb,
     x, ybb, ...).`
---> Not affected at this stage
(Pdb) relations
[(y0_blk0, y), (t, x, y0_blk0, z), (t, x, y, z), (time, t), (), (
    time, x, y0_blk0, z), (time, x, y, z)]
---> Next ispace decomposition/ New BlockDimension
(Pdb) block_dims
[x0_blk0, x]
(Pdb) print(d)
x
---> Form a graph path from the existing block_dims
(Pdb) relations
[(x0_blk0, x)]
---> Suitably replace `d` with all `bd`s in the already existing
    relations and avoid, e.g. `x > yb`
(Pdb) relations
[..., (time, x0_blk0, y, z), ..., (t, x0_blk0, y0_blk0, z), ..., (
    t, x0_blk0, y, z), ..., (time, x0_blk0, y0_blk0, z)]
---> BlockDimensions at the same depth stick next to each other, E
    .g., `(t, xbb, ybb, xb, yb, x, y)`, and NOT e.g. `(t, xbb, xb,
     x, ybb, ...).`
(Pdb) relations
[..., (y0_blk0, x)]
---> Final relations
(Pdb) relations
[(x0_blk0, x), (time, t), (time, x0_blk0, y, z), (time, x, y, z),
    (t, x0_blk0, y0_blk0, z), (y0_blk0, y), (t, x0_blk0, y, z), (t
    , x, y, z), (time, x0_blk0, y0_blk0, z), (), (y0_blk0, x)]
```

Listing 4.11: `IterationSpace` before and after loop blocking.

```
1 (Pdb) clusters[1].ispace
2 IterationSpace[time[0,0]++, x[0,0]++, y[0,0]++, z[0,0]++]
3 (Pdb) ---> clusters = blocking(clusters, sregistry, options)
4 (Pdb) clusters[1].ispace
5 IterationSpace[time[0,0]++, x0_blk0[0,0]++, y0_blk0[0,0]++, x
      [0,0]++, y[0,0]++, z[0,0]++]
```

Listing 4.12: The symbolic bounds of newly created BlockDimensions. We block over the x-Dimension, for two levels of blocking

```
1 (Pdb) block_dims
2 [x0_blk0, x0_blk1, x]
3 (Pdb) block_dims[0].symbolic_min
4 x_m
5 (Pdb) block_dims[0].symbolic_max
6 x_M
7 (Pdb) block_dims[0].symbolic_incr
8 x0_blk0_size
9 (Pdb) block_dims[1].symbolic_min
10 x0_blk0
11 (Pdb) block_dims[1].symbolic_max
12 x0_blk0 + x0_blk0_size - 1
13 (Pdb) block_dims[1].symbolic_incr
14 x0_blk1_size
15 (Pdb) block_dims[2].symbolic_min
16 x0_blk1
17 (Pdb) block_dims[2].symbolic_max
18 x0_blk1 + x0_blk1_size - 1
19 (Pdb) block_dims[2].symbolic_incr
20 1
```

block sizes used to iterate over these dimensions often do not perfectly divide the computational domain. Consequently, special treatment is needed to generate code that iterates the whole domain and avoids out-of-bounds accesses. Defining the correct bounds is a challenging task. The existence of subdomains, where a stencil is only executed at a specific subset of points of the whole domain, complicates this task.

We may look at Figure 4.5 to better understand this issue. Here we have a computational domain (white grid points, including blue points) and a subdomain (blue points only).

Assuming we are using loop blocking over the x dimension with an $x0\_blk0\_size$ of 8 and the computational domain size is 20, as seen in Figure

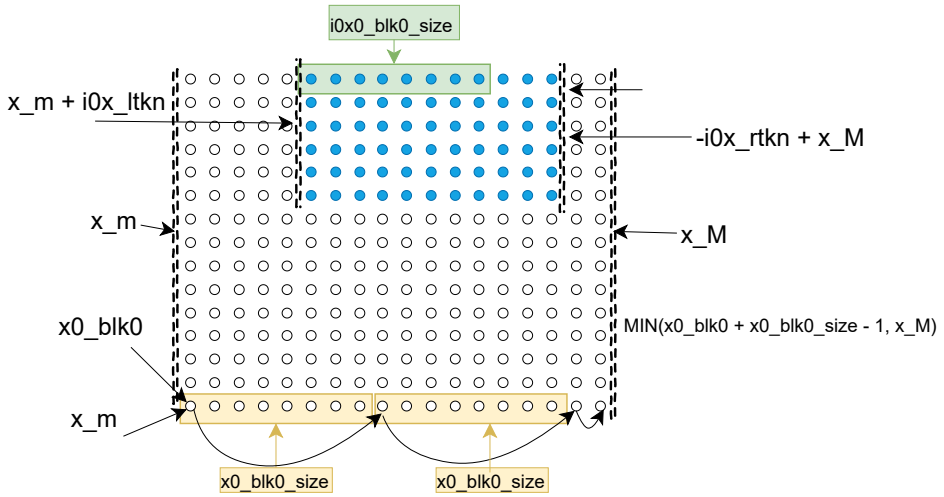


Figure 4.5.: The figure shows a computational domain with a subdomain. The points that belong to the subdomain are in blue. Yellow and green-filled blocks show that blocks of grid points do not perfectly cover the extent of the computational domain or the subdomain, respectively.

4.5. Iterating over blocks leads to iterating over $x0\_blk0 + x0\_blk0\_size - 1$ (i.e. 0, 8, 16, 24), however we cannot exceed x_M which is 20. Consequently, we must ensure that we iterate over an `IterationSpace` whose bounds are defined by some constraints. In this example, the constraints are the domain upper bound and the blocking dimension upper bound. Similar action should be taken for the start of the loop as well. Assuming $bl1max$ is the upper bound of the x blocked loop, the loop's upper bound should be:

$$\text{upper bound} = \min(bl1max, \text{domain bound}) \tag{4.1}$$

which for our running example in Figure 4.5 translates to:

$$\text{upper bound} = \min(x0\_blk0 + x0\_blk0\_size - 1, x\_M) \tag{4.2}$$

Similarly, for the subdomain (blue coloured in Figure 4.5), assuming $i0x0\_blk0\_max$ is the maximum of the block in subdomain, Equation 4.1 translates in:

$$\text{subdomain upper bound} = \min(i0x0\_blk0\_max, \text{subdomain bound}) \tag{4.3}$$

$$\text{upper bound} = \min(i0x0\_blk0 + i0x0\_blk0\_size - 1, -i0x\_rtkn + x\_M) \tag{4.4}$$

4.3.2.4. Valid but redundant code generation: hierarchical blocking

Additional constraints should be considered if more than one blocking level is generated. These constraints stem from the new blocking levels operating within the larger block shapes. Assuming $bl2\_max$ is the upper bound of the second blocked loop for the x dimension (loop iterating within the first blocked loop), the additional constraint adds to Eq. 4.1 which results in:

$$\text{upper bound} = \min(\text{domain bound}, bl1\_max, bl2\_max) \tag{4.5}$$

However, it is up to us to decide whether the inner block sizes are perfect divisors of the outer block shape. Perfect loop blocking divisors is a standard implementation technique for hierarchical space blocking [Kim et al., 2007]. Consequently, $\min(bl1\_max, bl2\_max) = bl1\_max$. However, this will lead to redundant code generation as we do not need to compute a more complex expression. Compilers often generate the expression $\min(bl1\_max, bl2\_max)$, which is still valid. We aim to reduce the redundant code generation and simplify things for better control over the loop iterations.

4.4. Temporal blocking in Devito

The previous Section 4.3 introduced in detail the loop blocking pass currently in Devito. Here, we introduce our contributions to automating temporal blocking by complementing and building upon Section 4.3.

This section describes the methodology followed to automate the generation of temporal blocking code within Devito. We add temporal blocking

as a new pass at the end of the Devito optimisation pipeline. The methodology to generate temporal blocking code was split into three independent `Cluster` passes in Devito. Figure 4.6 shows an overview of the introduced optimisation pipeline, and these new `Cluster` optimisation passes are the following:

1. A temporal blocking `Cluster` pass
2. A skewing pass
3. A bound-relaxing pass

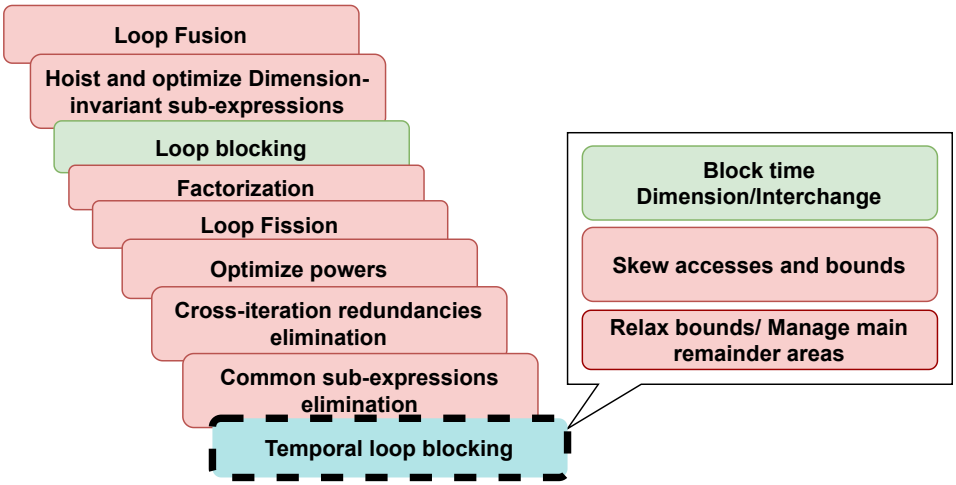


Figure 4.6.: The Devito compiler optimisation pipeline for arithmetic optimisation and loop transformations. We add temporal blocking as an additional optimisation pass. Temporal blocking consists of three sub-passes. (i) A pass for blocking a TimeDimension, (ii) a pass for skewing accesses and loop bounds, (iii) a pass for managing the bounds for satisfying main and remainder area requirements.

4.4.1. The time loop blocking pass

This `Cluster` pass is similar to the loop blocking pass presented in subsections 4.3.2.1 and 4.3.2.2. The difference is that now we should allow the compiler to generate blocking dimensions for SEQUENTIAL dimensions and, more specifically, for dimensions that refer to "Time" iterations. We care about "time" here because we can encounter other SEQUENTIAL dimensions we do not want to block. Such an example is dimensions that help to iterate over subdomains.

Listing 4.13 shows the `IterationSpace` of the `Cluster` before and after space and time blocking. The newly constructed TimeDimension gets its position at the first place of the dimensions as the handling of relations used is the same in the standard synthesis of loop blocking. To derive the relations, we follow the same strategy used in space blocking and shown in Listing 11. Listing 4.13 shows the `IterationSpace` of a `Cluster` where temporal blocking was applied and the resulting relations.

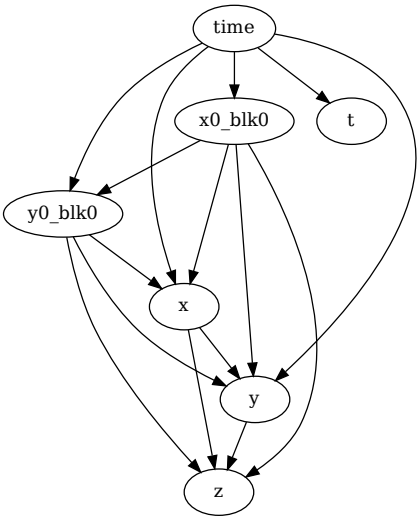
Listing 4.13: The `IterationSpace` of a cluster before and after time loop blocking.

```
1 (Pdb) clusters[1].ispace
2 IterationSpace[time[0,0]++, x0_blk0[0,0]++, y0_blk0[0,0]++, x
      [0,0]++, y[0,0]++, z[0,0]++]
3 (Pdb) ---> Synthesise temporal blocking
4 (Pdb) clusters[1].ispace
5 IterationSpace[time0_blk0[0,0]++, x0_blk0[0,0]++, y0_blk0[0,0]++,
      time[0,0]++, x[0,0]++, y[0,0]++, z[0,0]++]
6 (Pdb) clusters[1].ispace.relations
7 {(time0_blk0, time), (y0_blk0, time), (time, y), (x0_blk0, y), (
      time0_blk0, x0_blk0), (time, z), (x0_blk0, z), (time, x), (
      time0_blk0, t), (time0_blk0, y), (x0_blk0, x), (time0_blk0, z)
      , (y0_blk0, y), (t, x0_blk0, y0_blk0, z), (y0_blk0, z), (y, z)
      , (time0_blk0, x), (y0_blk0, x), (x, y), (x, z), (x0_blk0,
      y0_blk0), (time0_blk0, y0_blk0), (time0_blk0, x0_blk0, y0_blk0
      , z), (x0_blk0, time), ()}
```

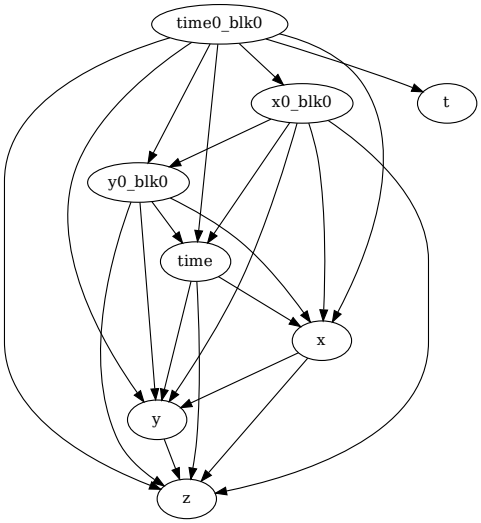
Figure 4.7 shows several snapshots of the relations that form a Directed Acyclic Graph (DAG). Subfigure 4.7a shows a DAG after a single level of loop blocking. The additional nodes (blocked dimensions) and the new relations are added. The addition of the time-blocking loop and the necessary relations that perform the interchange between space and time loops are shown in Figure 4.7b. In wavefront temporal blocking, we use two space-blocking levels as shown in Figure 4.7c. Finally, after applying transitive reduction to the directed acyclic graph, we arrive at Figure 4.7d, representing the final loop structure. This step marks the end of the blocking phase, and the skewing phase follows.

4.4.2. The skewing pass

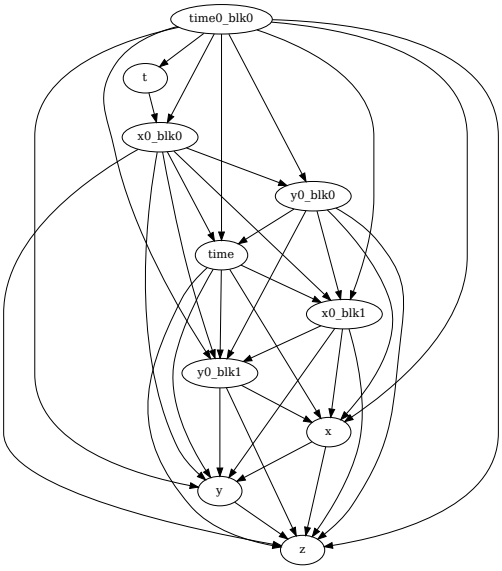
The skewing pass aims to help skew accesses and loop bounds of expressions in a cluster. Skewed accesses and bounds are a prerequisite for generating wavefront temporal blocking code.



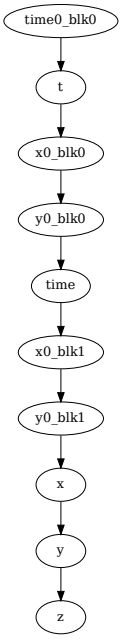
(a) The DAG after one level of loop blocking



(b) The DAG after time blocking and one-level of loop blocking



(c) The DAG after time blocking and two levels of loop blocking



(d) Flattened DAG after topological sorting (Transitive reduction)

Figure 4.7.: Figure shows the Directed Acyclic Graph (DAG) of relations after applying a level of blocking patterns. Loop interchange automatically occurs through the algorithm defining relations in Listing 11. The final loop structure emerges after simplifying the graph through several steps, such as transitive reduction. Within the Devito compiler, this procedure is called topological sorting.

Similar to the "blocking" pass, we also have an analysis and a synthesis part in skewing. Both of them will be explained in the following subsections 4.4.2.1 and 4.4.2.2.

4.4.2.1. Loop skewing analysis

We consider a loop a candidate for skewing if it satisfies the PARALLEL and AFFINE properties. The tiling analysis explained in Section 4.3.2.1 can be reused to help with that. Since the TILABLE property has already been removed for the blocked loops, we re-run the blocking analysis to re-attach the TILABLE property. Following this, we iterate over the clusters and attach the SKEWABLE property to those PARALLEL and AFFINE loops. Algorithm 12 briefly illustrates the algorithm used for the skewing pass.

ALGORITHM 12: Pseudocode for part of loop skewing analysis.

```
Input: clusters, prefix
Output: processed
1 d = prefix[-1].dim
2 for c in clusters do
      // TILABLE is a necessary condition
3     if TILABLE not in c.properties[d] then
4         return clusters
  // All good, `d` is actually SKEWABLE
5 processed = attach_property(clusters, d, SKEWABLE) return processed
```

4.4.2.2. Loop skewing synthesis

After the loop skewing analysis is finished (see Section 4.4.2.1), we proceed to loop skewing synthesis. Again, we use a Queue to implement this pass. A set of clusters is given as input to be transformed. Accesses and loop bounds are skewed. We follow different heuristic strategies to drive the skewing. The algorithm in Listing 13 shows the pseudocode for the skewing synthesis pass. We check whether any SKEWABLE property exists in the cluster's dimensions. If more than two SKEWABLE dimensions are present, we avoid applying skewing as the expected performance for skewing more than two dimensions is still unexplored territory. Then, depending on whether we have or not a blocked time loop, which would lead to interchanged loops (see Section 4.4.1), we decide at which level of the loop hierarchy we will skew the accesses. We may well support

skewed accesses without necessarily performing loop interchange (see Section 2.3.3 for loop interchange). However, skewing the accesses without loop interchange is not expected to benefit performance. The reason is that along with skewing accesses, we also skew the bounds keeping the same overall execution schedule. Thus no performance benefit is going to be exposed from this optimisation pass.

ALGORITHM 13: Pseudocode for part of loop skewing synthesis.

```
Input: clusters, prefix
Output: processed
d = prefix[-1].dim
for c in clusters do
    // TILABLE is a necessary condition
    if SKEWABLE not in c.properties[d] then
        return clusters
    if d is c.ispace[-1].dim and not self.skewinner then
        return clusters
    skew_dims = i.dim for i in c.ispace if SEQUENTIAL in c.properties[i.dim]
    if len(skew_dims) > 2: then
        return clusters
    skew_dim = skew_dims[-1]
    // Prefix is skewable and nested under a SEQUENTIAL loop
    skewlevel = 1
    intervals = []
    for i in c.ispace do
        if i.dim is d then
            // If time is blocked, skew at skewlevel + 1
            cond1 = len(skew_dims) == 2 and d._depth == skewlevel + 1
            // If time is blocked, skew at level == 0 (e.g.  subdims)
            cond3 = len(skew_dims) == 2 and d._depth == 0
            // If time is not blocked, skew at level <=1
            cond2 = len(skew_dims) == 1 and d._depth <= skewlevel
            if cond1 then
                intervals.append(Interval(d, i.lower, i.upper))
            else if cond2 or cond3 then
                intervals.append(Interval(d, skew_dim, skew_dim))
            else
                intervals.append(i)
        else if i.dim is d then
            intervals.append(i)
    intervals = IntervalGroup(intervals, relations=c.ispace.relations)
    ispace = IterationSpace(intervals, c.ispace.sub_iterators, c.ispace.directions)
    exprs = xreplace_indices(c.exprs, d: d - skew_dim)
    processed.append(c.rebuild(exprs=exprs, ispace=ispace))
return processed
```

A simple example lets us look at expressions' input and output clusters. Listing 4.14 shows the expressions and their index accesses before and after

loop skewing. In this (general) case, we skew the x and the y dimensions. We do not skew at the z dimension as vectorising over the innermost accesses has proven more profitable and is a generally accepted and applied practice in this family of stencil kernels.

Listing 4.14: Cluster expressions before and after loop skewing synthesis.

```
(Pdb) clusters[1]
Cluster([Eq(r4, -2.0*u[t0, x + 2, y + 2, z + 2])
         Eq(u[t1, x + 2, y + 2, z + 2], dt*(r0*u[t0, x + 2, y + 2,
              z + 2] + a*(r1*r4 + r1*u[t0, x + 1, y + 2, z + 2] +
             r1*u[t0, x + 3, y + 2, z + 2]....)))])

(Pdb) ---> clusters = skewing(clusters, sregistry, options)

(Pdb) clusters[1]
Cluster([Eq(r4, -2.0*u[t0, -time + x + 2, -time + y + 2, z + 2])
         Eq(u[t1, -time + x + 2, -time + y + 2, z + 2], dt*(r0*u[
             t0, -time + x + 2, -time + y + 2, z + 2] + a*(r1*r4 +
              r1*u[t0, -time + x + 1, -time + y + 2, z + 2] + r1*u
             [t0, -time + x + 3, -time + y + 2, z + 2]....)))])
```

4.4.3. Parametrizing bounds: main and remainder areas in temporal blocking

Section 4.3.2.3 introduced the issue of main and remainder areas. We showed that deriving the necessary bounds for space blocking in one or more levels of depth has a straightforward logic. However, there is more than one challenge when deriving the bounds in temporal blocking schemes. The diagonal planes (see Figure 4.8) do not only intersect with the domain bounds. They additionally intersect with the tile bounds and the loop blocking bounds.

Figure 4.8 illustrates that for the loop iterators, we have the following constraints:

1. the space 'tile' loop iterator is bounded by its self bounds (tile, red bounds), the domain bound in space and time (black arrows) and the space-time diagonal (blue line).
2. the space 'block' loop iterator is bounded by itself, and its skewed parent 'tile' bound (and consequently the domain bound) (block, red-dotted line) and the space-time diagonal (blue line)

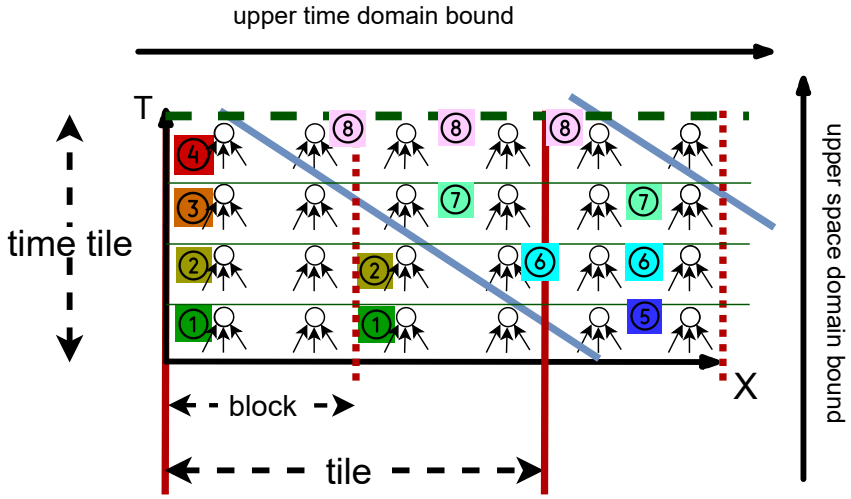


Figure 4.8.: A one-dimensional 3-point Jacobian stencil kernel, time-tiled example. The loop bounds in wavefront temporal blocking are derived by computing the intersection of several limiting factors. These factors include (in red/crimson) the 'tile' loop blocking bounds, (in red/crimson dotted) the 'block' loop blocking bounds, (in black) the domain bounds, and (in green) the time-tile bound.

3. the time iterator is bounded by the domain size (black arrows) and time-tile size/space-time diagonal (blue line).

All the complexities stem primarily from the diagonal skewed planes, which are functions of space and time. Deriving the complex loop bounds paves the way for valid schedule code generation and avoids out-of-bounds accesses. We consider the above constraints to generate the valid bounds and transform the bounds for every "Iteration" stemming from a Dimension.

For simplicity, to show the bounds of an Iteration for a specific Dimension, we use the following format:

```
Iteration name; ( minimum, maximum, step )
```

Listing 4.15 shows the mapping of the Iteration bounds before and after applying the computed iteration bounds to iterate over the grid points using the wavefront temporal blocking method. We use the same j1d-3pt example similar to Figure 4.8.

Listing 4.15: Listing shows how the Iteration loop bounds look like before and after computing the valid bounds for iterating over the grid points through the wavefront temporal blocking method. The format is "Iteration name; (minimum, maximum, step)". A j1d-3pt example is used, similar to Figure 4.8

```
1 // Iteration x0_blk0;
2 (x_m, x_M, x0_blk0_size):
3 // to
4 (x_m, time_M - time_m + x_M, x0_blk0_size)
5
6 // Iteration time[t0,t1];
7 (time0_blk0, time0_blk0 + time0_blk0_size - 1, 1):
8 // to
9 (time0_blk0, MIN(time0_blk0 + time0_blk0_size - 1, time_M), 1)
10
11 // Iteration x0_blk1;
12 (x0_blk0, x0_blk0 + x0_blk0_size - 1, x0_blk1_size):
13 // to
14 (MAX(x0_blk0, time + x_m), MIN(x0_blk0 + x0_blk0_size - 1, time +
       x_M), x0_blk1_size)
15
16 // Iteration x;
17 (x0_blk1, x0_blk1 + x0_blk1_size - 1, 1) :
18 // to
19 (x0_blk1, MIN(MIN(x0_blk0 + x0_blk0_size - 1, time + x_M), x0_blk1
        + x0_blk1_size - 1), 1)
```

4.4.4. Wavefront temporal blocking code generation

Implementing a valid strategy for deriving loop bounds helps to generate code for several combinations of skewing, time blocking and multilevel blocking (as presented in sections 4.4.1 and 4.4.2.2).

Code generation capabilities These combinations can result in several code generation schemes. Among others, we generate code for simple skewing, skewing with one-level loop interchange (Time skewing), skewing with two-level loop interchange (Wavefront temporal blocking), and multi-level loop interchange (Wavefront temporal blocking with hierarchical space blocking). In addition, after applying these optimisations, we also apply shared-memory OpenMP parallelism and SIMD vectorisation. Again, the optimisations presented in this chapter are implemented to work smoothly with already-present Devito optimisations.

Simple arguments in the Operator drive all these code-generation capabilities. Listing 4.16 briefly shows how arguments can drive code generation.

Listing 4.16: Combining compiler passes for several variants of code generation.

```
# Simple access skewing
(Pdb) op.apply('advanced', {"skewing":True})
# Time skewing
(Pdb) op.apply('advanced', {"skewing":True, "blocktime":True})
# Wavefront temporal blocking
(Pdb) op.apply('advanced', {"skewing":True, "blocklevels":1})
# Wavefront temporal blocking with hierarchical space blocking
(Pdb) op.apply('advanced', {"skewing":True, "blocklevels":2})
```

Finally, after successively applying the three compiler passes presented in Sections 4.4.1, 4.4.2 and 4.4.3, we are able to generate highly-optimised C code using wavefront temporal blocking. Listing 4.17 shows a snippet of the generated C code with applied wavefront temporal blocking.

Listing 4.17: Snippet of the generated C code with wavefront temporal blocking.

```
float r0 = 1.0F/dt;
float r1 = 1.0F/(h_x*h_x);
float r2 = 1.0F/(h_y*h_y);
float r3 = 1.0F/(h_z*h_z);

for (int time0_blk0 = time_m; time0_blk0 <= time_M; time0_blk0 += time0_blk0_size)
{
  for (int x0_blk0 = x_m; x0_blk0 <= time_M - time_m + x_M; x0_blk0 += x0_blk0_size)
  {
    for (int y0_blk0 = y_m; y0_blk0 <= time_M - time_m + y_M; y0_blk0 += y0_blk0_size)
    {
      for (int time = time0_blk0, t0 = (time)%(2), t1 = (time + 1)%(2); time <= MIN(time0_blk0 + time0_blk0_size - 1, time_M); time += 1, t0 = (time)%(2), t1 = (time + 1)%(2))
      {
        #pragma omp parallel num_threads(nthreads)
        {
          #pragma omp for collapse(2) schedule(dynamic,1)
          for (int x0_blk1 = MAX(x0_blk0, time + x_m); x0_blk1 <= MIN(x0_blk0 + x0_blk0_size - 1, time + x_M); x0_blk1 += x0_blk1_size)
          {
          for (int y0_blk1 = MAX(y0_blk0, time + y_m); y0_blk1 <= MIN(y0_blk0 + y0_blk0_size - 1, time + y_M); y0_blk1 += y0_blk1_size)
            {
              for (int x = x0_blk1; x <= MIN(MIN(x0_blk0 + x0_blk0_size - 1, time + x_M), x0_blk1 + x0_blk1_size - 1); x += 1)
              {
                for (int y = y0_blk1; y <= MIN(MIN(y0_blk0 +
```

```
                    y0_blk0_size - 1, time + y_M), y0_blk1 +
                    y0_blk1_size - 1); y += 1)
24                {
25                  #pragma omp simd aligned(u:32)
26                  for (int z = z_m; z <= z_M; z += 1)
27                  {
28                    float r4 = -2.0F*u[t0][-time + x + 2][-time +
                          y + 2][z + 2];
29                    u[t1][-time + x + 2][-time + y + 2][z + 2] =
                          dt*(a*(r1*r4 + r1*u[t0][-time + x + 1][-
                          time + y + 2][z + 2] + r1*u[t0][-time + x
                          + 3][-time + y + 2][z + 2] + r2*r4 + r2*u[
                          t0][-time + x + 2][-time + y + 1][z + 2] +
                           ... ;
30                  }
31                }
32              }
33            }
34          }
35        }
36      }
37    }
38  }
39 }
```

4.5. Performance evaluation

This section evaluates our automated temporal blocking scheme's performance on a range of stencil kernels. We use various CPU platforms to benchmark our kernels, having different core counts, cache sizes, clock rates and memory bandwidth.

The first system is a desktop range Intel CPU with a single socket 16-core Intel IceLake, namely i7-10700KF with AVX2 support. Each CPU has three cache levels: 256KiB L1d, L1i caches private to each core, and shared 2MiB L2 and a 16MiB L3 cache.

The second system is an HPC-server range Intel CPU with a dual-socket 20-core per socket Intel Cascade Lake, Gold 5218R with AVX512 support. Each CPU has three cache levels: 1.3MiB L1d, L1i caches private to each core, and shared 40MiB L2 and a 50MiB L3 cache.

The third system is an HPC-server range Intel CPU with a dual-socket 20-core per socket Intel Cascade Lake, Gold 6230 with AVX512 support. Each CPU has three cache levels: 32KiB L1d, L1i caches private to each core, and shared 1MiB L2 and a 27.5MiB L3 cache.

The fourth system is an HPC-server range AMD CPU with a dual-socket

64-core per socket EPYC7742 with AVX2 support. This system has 4 NUMA nodes per socket, so the effective physical cores we use for the benchmarking are 16. Each CPU has three cache levels: 32KiB L1d, L1i caches private to each core, and shared 512 KiB L2 and a 16MiB L3 cache.

The fifth system is an older generation HPC-server range Intel CPU with a dual socket 8-core per socket Intel SandyBridge, namely E5-2640 with AVX support. Each CPU has three cache levels: 512KiB L1d, L1i caches private to each core, and shared 4MiB L2 and a 40MiB L3 cache.

The above details are summarised in table 4.1. The compiler used was GCC 9.3.0*.

CPU characterstics					
	i7-10700KF	Gold 5218R	Gold 6230	EPYC7742	E5-2640
CPU(s)	16	80	40	256	32
Thread(s) per core:	2	2	1	2	2
Core(s) per socket:	8	20	20	64	8
Socket(s):	1	2	2	2	2
NUMA node(s):	1	2	2	8	2
Clock rate:	3.80GHz	2.10GHz	2.10GHz	2.25GHz	2.50GHz
L1d cache:	256KiB	1.3MiB	32KiB	32KiB	512KiB
L1i cache:	256KiB	1.3MiB	32KiB	32KiB	512KiB
L2 cache:	2MiB	40 MiB	1MiB	512KiB	4MiB
L3 cache:	16MiB	55 MiB	27.5MiB	16MiB	40MiB
Max Mem BW:	45GB/s	120GB/s	131GB/s	190GB/s	59GB/s

Table 4.1.: Characteristics of CPU platforms used for benchmarking

We used OpenMP shared-memory parallelism with dynamic scheduling and SIMD vectorisation. Thread pinning was enabled using the environment variables OMP_PROC_BIND (GCC) and KMP_AFFINITY (ICC). The experimentation framework and instructions on reproducibility are available in Section 3.4.6. Our work was built on top of Devito v.4.6.2 and can automatically generate code for the following combinations of compiler passes:

1. space blocking
2. time blocking (time skewing with space blocking)

3. wavefront (time skewing with hierarchical space blocking)

We compare our best-automated variant, which is wavefront temporal blocking, against the highly optimised spatially blocked and vectorised Devito kernels.

Note: Before proceeding to results, we highly recommend that the reader has revised Section 3.4, as several paragraphs, such as NUMA issues and the reduced performance for high-order stencils, are naturally encountered.

Working set of the problem Cache optimisations aim to improve cache hits in multiple cache levels. Therefore we need to benchmark problems with a size that exceeds cache capacity. In order to ensure that in the presented results, the problem size exceeds the cache capacity, and therefore we evaluate benefits across all cache levels; we calculate and present the working set size for all the benchmarked problems.

Working set/Problem size	
Laplace	512x512x512 [1GB]
Acoustic	512x512x512 [2GB]
TTI	256x256x256 [$\approx$ 1.8GB]

Table 4.2.: Size of the working set of the problems to be evaluated

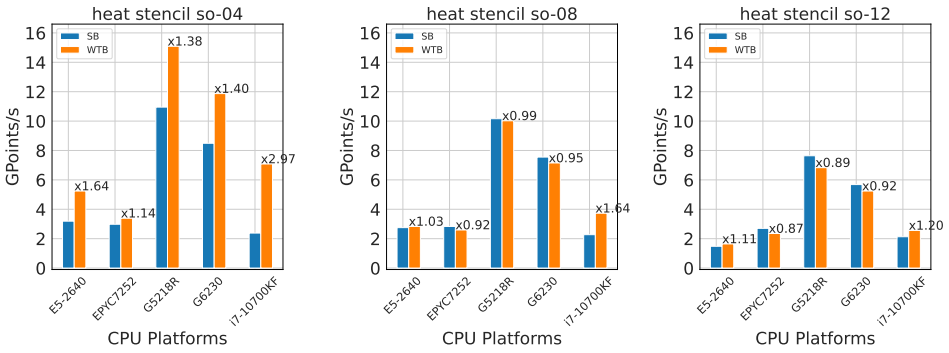
4.5.1. Evaluation of standard stencil kernels

In this subsection, we evaluate some stencils widely used in the majority of research papers that benchmark stencil kernels. Those stencils include the Heat Diffusion stencil and a standard implementation of the acoustic wave stencil, as shown in Section 2.1.1.

4.5.1.1. The heat diffusion stencil

The equations that describe Heat Diffusion were discussed in Section 2.1.3.1. We presented how the stencil kernel derives from the mathematical equations. The heat diffusion stencil lowers to a Laplacian-like Jacobi kernel, which forms a 13pt, 25pt and 37pt star stencil in 3D for space discretisation orders of 4, 8 and 12, respectively. In this subsection, we

evaluate the impact of temporal blocking on the performance of the Heat Diffusion stencil. Subfigures 4.9a, 4.9b 4.9c show the Gpts/s throughput on a variety of platforms.



(a) j3d–13pt **(b)** j3d–25pt **(c)** j3d–37pt

Figure 4.9.: Gpoints/s throughput for varying space discretisation order on various platforms for the Heat Diffusion 512x512x512 problem (Higher is better).

For space discretisation order (henceforth SDO) of 4, the temporal blocking improvement over the highly optimised spatially blocked code starts from 1.14x for the EPYC7252, around 1.4x for the G5218R and G6230, and 1.64x for E5-2640. The improvement on the i7-10700KF is highly impressive, reaching up to 3x. For SDO 8, we only see improvement on the i7-10700KF, reaching 1.64x. The rest of the platforms show no improvement. For the high SDO 12, we see speedup on the i7-10700KF and E5-2640 of 1.2x and 1.1x, respectively. Figure 4.9 shows that CPUs with high maximum memory bandwidth benefit less compared to CPUs with less memory bandwidth such as the i7-10700KF.

4.5.1.2. The wave-equation stencil

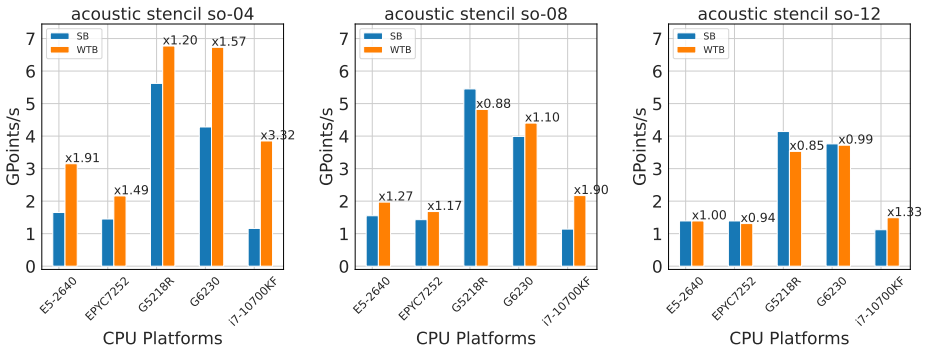
The equations that describe wave propagation were discussed in Section 2.1.3.2. We presented how the stencil kernel derives from the mathematical equations. Since the wave equation itself is similar to a more advanced version of a real-world example, we benchmark this stencil along with boundary conditions and damping fields in Section 4.5.2.1.

4.5.2. Industrial-level applications

In this subsection, we evaluate more complex stencils than compared to Section 4.5.1. Most of the physics in this subsection has already been presented in Section 3.3. These kernels contain damping fields, absorbing boundary conditions and others.

4.5.2.1. Isotropic acoustic

In Figure 4.10, we see the results of the benchmarked kernels with wave-front temporal blocking versus the highly-optimised spatially blocked kernels for the Acoustic wave equation stencil kernels.



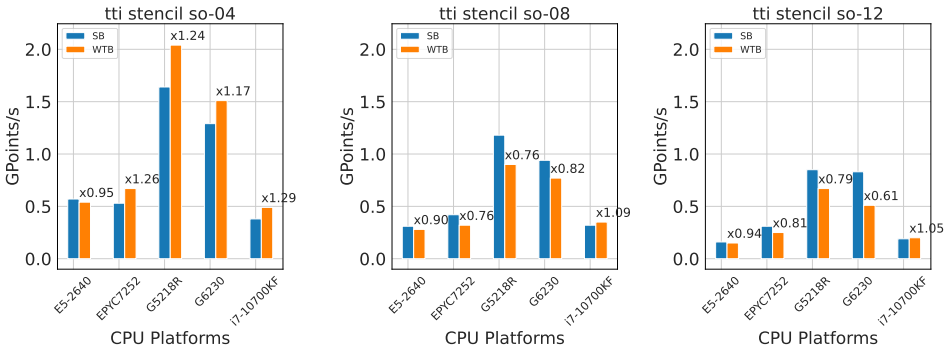
(a) acwave04 **(b)** acwave08 **(c)** acwave12

Figure 4.10.: Gpoints/s throughput for varying space discretisation order on various platforms for the Acoustic wave propagation 512x512x512 problem (Higher is better).

For SDO 4, the WTB speedup over the highly optimised spatially blocked code starts from 1.2x for the G5218R, around 1.4x for the EPYC7252 and G6230, 1.91x for E5-2640 and 3.32x for the i7-10700KF. For SDO 8, there is no improvement on G5218R. However, there is speedup for the majority of platforms. 1.10x for G6230, 1.17x for EPYC7252, 1.27x for E5-2640 and finally 1.9x for i7-10700KF. For the high SDO 12, we only see a speedup of 1.33x on the i7-10700KF. We notice that CPUs with high maximum memory bandwidth benefit less compared to CPUs with less memory bandwidth, such as the i7-10700KF.

4.5.2.2. Anisotropic acoustic (TTI)

Figure 4.11 shows the results of the benchmarked WTB kernels versus the highly optimised spatially blocked kernels for the anisotropic acoustic (TTI) wave equation stencil kernel. TTI is a very complex and demanding kernel. For more info, redirect to Section 3.3.2



(a) ttiwave04 **(b)** ttiwave08 **(c)** ttiwave12

Figure 4.11.: Gpoints/s throughput for varying space discretisation order on various platforms for the anisotropic acoustic wave propagation 256x256x256 problem (Higher is better).

For SDO 4, the WTB speedup over the highly optimised spatially blocked code starts from 1.17x, for the G6230, around 1.25x for the EPYC7252 and G5218R, and 1.29x for the i7-10700KF. For SDO 8 and SDO 12, there is a slight improvement on G5218R. However, there is speedup for the majority of platforms. 1.10x for G6230, 1.17x for EPYC7252, 1.27x for E5-2640 and finally 1.9x for i7-10700KF. For the high SDO 12, we only see a speedup of 1.33x on the i7-10700KF. Figure 4.11 shows some slight benefit towards CPUs with higher max memory bandwidth, probably because we benchmark a kernel with high arithmetic intensity, and it is mainly limited by computational resources rather than memory bandwidth. Consequently, powerful CPUs with good caches can help get some benefits.

4.5.2.3. Elastic

Automating temporal blocking code generation for the elastic-wave propagation kernel could not be implemented in this thesis. Some additional machinery is needed in order to satisfy the data dependency requirements of the elastic stencil kernel. These complex data dependencies stem from

cross-loop read-write dependencies for updating multiple fields. Automating temporal blocking for the elastic kernel needs additional work and is related to an open issue in the Devito compiler. [3] This codegen limitation does not allow our scheme to transform loops with cross-loop data dependencies as described in Section 3.3.3.

If the reader is eager to look at possible gains from applying temporal blocking to the elastic wave propagation kernels, we redirect to the performance evaluation in Section 3.18, where a semi-automated with manual edits implementation is evaluated. We consider automated temporal blocking for the elastic wave propagation as future work; more details are provided in Section 4.7.

4.6. Conclusions

This chapter presented a general compiler approach to automatically applying temporal blocking to stencil computations. Starting from a high level of symbolic abstraction, we generate code harnessing the advantages of temporal blocking automatically. By breaking temporal blocking down into independent building blocks, we managed to ease its implementation within the compiler and support multiple combinations of closely related heuristic optimisations in tandem with pre-existing optimisations. These combinations include simple skewing (accesses and bounds), skewing with one-level loop interchange (Time skewing), skewing with two-level loop interchange (Wavefront temporal blocking) and multi-level loop interchange (Wavefront temporal blocking with hierarchical space blocking). In addition, they offer straightforward integration with shared-memory parallelism (inner and outer). To the best of our knowledge, we offer an automated solution that is the first to combine high-level mathematic abstraction and temporal blocking code generation. The contributed optimisation was evaluated against state-of-the-art highly-optimised implementations from the Devito compiler and proven to offer considerable performance benefits. The experimental evaluation makes temporal blocking an attractive optimisation target for stencil computations. The compiler work presented in this thesis significantly helps automate wavefront and other temporal blocking variants.

[3] The open issue can be found here: Issue #618.

4.7. Limitations and Future work

Our work has several limitations, which we review in this section. These limitations stem mainly from the lack of time to implement additional features and the performance limitations of temporal blocking with high-order stencils. We discuss these limitations and propose future research directions to resolve them.

1. The contributed compiler infrastructure currently generates only wavefront temporal blocking (WTB) code. However, WTB is known to have its limitations in performance. Performance is often limited by the disadvantages of WTB, such as thread load imbalance at the starting and ending phases of the sequentially progressing wavefronts. Whenever the wavefront area is small, threads have little space to operate and compute points in tiles. A future research direction would be to extend the compiler infrastructure with more temporal blocking methods. Other variants of temporal blocking, such as overlapped and diamond, offer better thread load balance.
2. Temporal blocking works smoothly with all Devito-supported optimisations apart from distributed-memory parallelism (MPI code generation). Combining MPI and temporal blocking is even more challenging as additional care must be taken for a number of aspects, such as allocating more data per node to satisfy data dependencies for more point updates in time. The short answer is communicating fatter/wider halos between nodes to satisfy communication and computation for more than one timestep. Additionally, it depends upon item 1 as tiles in wavefront temporal blocking are updated sequentially, meaning there is no space for another degree of distributed-memory parallelism over shared-memory parallelism. Consequently, a distributed-memory temporal blocking method would be an extension dependent on the future work of item 1. Although distributed memory parallelism is a great enhancement for production-grade runs, it is common to encounter Full Waveform Inversion (FWI) and Reverse Time Migration (RTM) runs without using MPI distributed parallelism. Consequently, we consider that our contribution could have an impact even without MPI. More specifically, there are problems using low-frequency waves that may be small enough to be

worth running on a single socket of a relatively high-end CPU.

3. Elastic-like wave propagation kernels are not compliant with the automated temporal blocking scheme. Additional machinery is needed to support cross-loop data dependencies. For performance results on the elastic-wave propagation, we redirect the reader to the semi-automated hand-tuned kernels in Section 3.4.2.
4. Integration of signal receivers is not implemented. This limitation stems from the specific application context of seismic and medical imaging. Consequently, this thesis only evaluates the stencil kernels for the forward phase. Performance-wise, this is quite satisfactory as this is one of the bottlenecks of the imaging pipeline. In order to fully execute an imaging problem, receivers need to be fully integrated into the automatically generated wavefront temporal blocking code. This automation requires some more software engineering and is left for future work.
5. Another limitation of this work is that it is limited to CPUs only. Tiling loops for execution on GPUs differs from CPUs depending on the model employed to offload on GPUs. Supporting temporal blocking for GPUs is in the roadmap of future work.
6. Other architectures: Another platform of interest to benchmark the temporally blocked kernels would be ARM A64FX. In order to take advantage of A64FX's performance, we need to use the Fujitsu CC (fcc) compiler [Odajima et al., 2020]. Our efforts to benchmark our temporal blocking model with fcc failed as the compiler would not digest for varying size loops (main/remainder areas with min/max bounds) as seen in Section 4.4.3 that could have zero length. Supporting this platform is considered future work. Another platform that would be interesting to try is Cerebras WSE-2, where recent research has shown great potential [Jacquelin et al., 2022].

4.7.1. Code availability

An implementation of the methods described in this chapter is available in a Devito fork repository under the MIT open-source license available at DOI: 10.5281/zenodo.7472679. See the chapter4-README.md for instructions on how to see the code used and reproduce the results in the chapter.

Chapter 5

Conclusions

This final chapter revisits the contributions, achievements and limitations of this thesis. Future research directions are also discussed, and we attempt to assess the potential scientific impact of this thesis.

5.1. Summary

This thesis introduced techniques to automate cache-related optimisations for improving the performance of numerical methods for solving partial differential equations. We evaluated the contributed optimisations on standard as well as real-world problems. The work presented in this thesis is built upon three main pillars:

Motivation The performance optimisation of any computational kernel should always start with bottleneck analysis. As in this thesis, this is even more challenging in the case of already-optimised code as we attempt to squeeze the last bit of performance out of the stencil kernels. The motivation behind that attempt is accelerating stencil computations that are omnipresent in real-life applications. More specifically we focus on stencil kernels stemming from seismic and medical imaging applications. Optimisations for real-world applications are only sometimes straightforward, as several factors must be considered. These factors usually stem from several parameters that need to be considered to simulate real-world phenomena accurately (e.g. boundary conditions, external factors and others). In this thesis, we systematically attempt to solve real-world applications to the best

extent possible. The cases where this is not possible are elaborated in the following limitations and future research Section 5.2.

Automation through Devito high-level compiler The integration of all kinds of optimisations into compilers is a challenging task. Arithmetic optimisations, loop transformations, parallelism and others are highly beneficial but require significant engineering effort. Especially in codes that simulate real-world phenomena, doing this by hand is time-consuming, highly tedious and error-prone. These complexities reduce productivity for scientific individuals and teams as they often fall apart their domain expertise for developing end-to-end scientific applications. Throughout this thesis, we aim to automate wavefront temporal blocking so that this happens behind the scenes without user intervention. This automation allows domain specialists to model practical applications and harness the speedup benefits of WTB without caring for low-level optimisations.

Validation of the hypotheses The hypotheses behind performance benefits must always be validated. In this thesis, we applied optimisations to improve cache reuse. We verified that our optimisations improved cache reuse via performance profiling tools such as Intel Advisor and rooflines models. All performance numbers reported in this thesis aim to be reproducible on the platforms used for benchmarking.

In this thesis, we have investigated two main challenges in finite-difference stencil computations on structured grids.

Temporal blocking of finite-difference stencil operators with sparse "off-the-grid" sources (Chapter 3) This work introduced a mechanism to enable temporal blocking in stencil computations involving sparse off-the-grid operators as encountered, for example, with sources and receivers in seismic inversion problems. We applied wavefront temporal blocking to wave-propagators ranging from isotropic acoustic to more advanced, such as isotropic elastic and anisotropic acoustic (TTI). Experimental evaluation of the improved kernels on Broadwell and Skylake microarchitectures showed compelling evidence of substantial acceleration of at least 1.5x for low and at least 1.1x for medium space order wave-propagation kernels.

The most significant achievement of this chapter is the inspector/executor scheme for applying temporal blocking to stencil kernels used in scientific simulations. These kernels often include sparse, "off-the-grid" operators. These operators raise limitations and complicate the straightforward application of temporal blocking.

Automated temporal blocking through compiler technology (Chapter 4) This work introduced a compiler pass scheme to automate temporal blocking as a compiler pass within an already existing high-level compiler like Devito. Our contributions help to automatically generate temporal blocking code starting from a high-level mathematic symbolic input.

Temporal blocking has proven beneficial for simple and more complex (Chapter 3) stencil computations. Automated code generation for temporal blocking is a challenging task that requires careful compiler pass design. Writing temporal blocking code by hand is a very error-prone task. It requires accurate coding and thinking of cache optimisations in a higher-dimensional space with time dimension complicating this approach. Especially for stencil kernels of practical applications, this task is even more complicated as we often encounter multi-field stencils, high-arithmetic intensity, imperfectly nested loops and indirect memory accesses.

Code availability The contributions of this thesis (code implementation, dissemination, experiments) are publicly available. The code integrated into Devito is publicly available via MIT License. The hyperlinks to reproduce this work are available in the respective chapter. See subsections 3.4.6 for Chapter 3 and 4.7.1 for Chapter 4.

5.2. Limitations and Future research directions

This section will discuss in more detail the limitations of this thesis and the future research directions that aim to resolve them. Some of the limitations of the research presented in this thesis have already been partly discussed in the previous chapters (see Section 3.5 for Chapter 3 and Section 4.7 for Chapter 4) . In this subsection, we emphasise them and discuss them more thoroughly.

Temporal blocking and high-order stencils By default, temporal blocking is expected to have limited gains for high-order stencils. Our goal is to push the performance bar even at an additional 5% to 10% compared to state-of-the-art, to unlock impact that may mean thousands of dollars for seismic imaging or faster diagnoses to try and save lives for medical imaging. Overall, achieving performance improvement through temporal blocking with high-space order kernels requires further research. Other temporal blocking methods, like diamond tiling, benchmarked on real-world stencils, see Multicore Wavefront Diamond tiling [Malas et al., 2015, Qu et al., 2020] offers an enhanced alternative to simple wavefront temporal blocking. This scheme is expected to perform better as it offers a better schedule for temporal cache reuse, concurrent start-up and better load balance for the OpenMP threads.

Other methods like stencil retiming [Stock et al., 2014] have shown promise in alleviating the performance bottleneck due to the high space discretisation order, a possible combination with temporal blocking could be promising.

Another possible solution could be data layout transformations in tandem with wavefront temporal blocking as presented in Yount [2015].

We recap the open questions that arose from Chapters 3 and 4 of the thesis and provide insights into potential research directions.

Advanced temporal blocking schemes The temporal blocking scheme presented in this thesis is wavefront temporal blocking. This is only one of many temporal blocking schemes, less or more advanced, having different properties, advantages and disadvantages. Implementing wavefront temporal blocking in Devito provides the necessary compiler infrastructure to extend and scale to more temporal blocking schemes. One of the schemes proven advantageous for stencil computations is diamond temporal blocking [Malas et al., 2015, Bertolacci et al., 2015, Levchenko and Perepelkina, 2017]. We aim to extend our infrastructure and automate other schemes, such as trapezoidal and diamond temporal blocking.

Distributed-memory temporal blocking for NUMA architectures The temporal blocking optimisations, similar to other cache blocking optimisations, are particularly beneficial if memory accesses occur in the same

NUMA region. As shown in the experimental evaluation in Sections 3.4 and 4.5, all experiments were performed in a single NUMA region. It has proven to be beneficial for locality to apply distributed memory parallelism across NUMA regions and shared memory parallelism across physical cores within the same NUMA region. In order to extend temporal blocking advantages to architectures with more than one NUMA region, an MPI-aware temporal blocking scheme should be implemented. This scheme should decompose the computational domain to each NUMA region (MPI distributed-memory parallelism) and perform temporal blocking with shared memory parallelism inside the region's domain. As discussed in Section 4.7, distributed memory parallelism greatly enhances production-grade runs. Classes of Full Waveform Inversion (FWI) and Reverse Time Migration (RTM) runs without using MPI distributed parallelism. More specifically, there are problems using low-frequency waves that may be small enough to be worth running on a single socket of a relatively high-end CPU.

Faster auto-tuning Tuning temporal blocking models induces more parameters compared to typical space blocking. In space blocking, testing each tuning run for 5 or 10 timesteps is enough to establish achieved performance. Wavefront temporal blocking has an additional level of space blocking plus the time dimension being blocked. In addition, each tuning run takes longer as it is not safe to conclude the optimal shape by only evaluating for a few timesteps as we do in standard space blocking. Consequently, we need more timesteps in wavefront temporal blocking to draw safe conclusions. The optimal block shapes can be a function of the cache sizes, stencil radius, and accesses. Some works have tried to predict the cache block sizes using layer conditions (LCs) [Hammer et al., 2017] from these parameters. An auto-tuning schedule for faster prediction of the optimal parameters is considered necessary.

Complete integration of sources and receivers The implementation presented in Chapter 4 is capable of automating temporal blocking to pure stencil loops. In order to benchmark using actual data, we initially perform source injection without temporal blocking to inject the signal in its complete form in the field. After this, we evaluate the stencil kernels. Chapter

3 showed that we can apply temporal blocking in the presence of sources and receivers. However, this still needs to be fully automated within the workflow. The complete integration of automating the transformation of loops containing source injection and receiver interpolation within the compiler is a work in progress.

Temporal locality optimisations on GPUs Another research direction that emerged during our work in Chapter 3 and Chapter 4 was to extend our work on accelerators such as Nvidia and AMD GPUs. Through OpenACC and OpenMP programming models, we can easily port our automatic code generation technology to GPUs. By modifying the cache blocking and parallelisation strategies discussed earlier, we optimise our code generation for further performance improvement on GPUs. We evaluate the performance of our optimised stencil kernels on GPUs and discuss our findings. Finite-difference stencil kernels have long been executed on GPUs (e.g. see [Micikevicius, 2009, Meng and Skadron, 2009, Schäfer and Fey, 2011, Zhang and Mueller, 2012, Holewinski et al., 2012, Zhao et al., 2018b, Reguly et al., 2016, Grosser et al., 2014a, Steuwer et al., 2017, Zohouri et al., 2018, Zhao et al., 2019, Sai et al., 2020, Matsumura et al., 2020, Gysi et al., 2021]).

Bibliography

A. Acharya and U. Bondhugula. Pluto+: Near-complete modeling of affine transformations for parallelism and locality. In *ACM SIGPLAN Notices*, volume 50, pages 54–64. ACM, 2015.

A. Afanasyev, M. Bianco, L. Mosimann, C. Osuna, F. Thaler, H. Vogt, O. Fuhrer, J. VandeVondele, and T. C. Schulthess. Gridtools: A framework for portable weather and climate applications. *SoftwareX*, 15: 100707, 2021. ISSN 2352-7110. doi: https://doi.org/10.1016/j.softx.2021.100707. URL https://www.sciencedirect.com/science/article/pii/S2352711021000522.

K. Akbudak, H. Ltaief, V. Etienne, R. Abdelkhalak, T. Tonellot, and D. Keyes. Asynchronous computations for solving the acoustic wave propagation equation. *The International Journal of High Performance Computing Applications*, 34:377 – 393, 2020.

S. Alhaddad, J. Förstner, S. Groth, D. Grünewald, Y. Grynko, F. Hannig, T. Kenter, F.-J. Pfreundt, C. Plessl, M. Schotte, et al. Highpermeshes–a domain-specific language for numerical algorithms on unstructured grids. In *Euro-Par 2020: Parallel Processing Workshops*, volume 12480, page 185. Nature Publishing Group, 2020.

T. Alkhalifah. An acoustic wave equation for anisotropic media. *Geophysics*, 65:1239–1250, 2000.

J. R. Allen and K. Kennedy. Automatic loop interchange. *SIGPLAN Not.*, 19 (6):233–246, jun 1984. ISSN 0362-1340. doi: 10.1145/502949.502897. URL https://doi.org/10.1145/502949.502897.

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 Trans. Math. Softw.*, 40(2), mar 2014. ISSN 0098-3500. doi: 10.1145/2566630. URL https://doi.org/10.1145/2566630.

M. S. Alnæs, A. Logg, G. Wells, L. Mitchell, M. E. Rognes, M. Homolya, A. Bergersen, D. A. Ham, J. Ring, chrisrichardson, and K.-A. Mardal. ufl: The unified form language, Apr. 2016. URL http://dx.doi.org/10.5281/zenodo.49282.

E. Anderson. Lapack users' guide. 1987.

R. Anderson, J. Andrej, A. Barker, J. Bramwell, J.-S. Camier, J. C. V. Dobrev, Y. Dudouit, A. Fisher, T. Kolev, W. Pazner, M. Stowell, V. Tomov, I. Akkerman, J. Dahm, D. Medina, and S. Zampini. MFEM: A modular finite element methods library. *Computers & Mathematics with Applications*, 81: 42–74, 2021. doi: 10.1016/j.camwa.2020.06.009.

A. W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, USA, 2004. ISBN 0521607647.

E. Apra, E. J. Bylaska, W. A. De Jong, N. Govind, K. Kowalski, T. P. Straatsma, M. Valiev, H. J. van Dam, Y. Alexeev, J. Anchell, et al. Nwchem: Past, present, and future. *The Journal of chemical physics*, 152(18):184102, 2020.

R. Baghdadi, U. Beaugnon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, A. Betts, A. F. Donaldson, J. Ketema, J. Absar, S. van Haastregt, A. Kravets, A. Lokhmotov, R. David, and E. Hajiyev. Pencil: A platform-neutral compute intermediate language for accelerator programming. *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 138–149, 2015.

R. Baghdadi, J. Ray, M. B. Romdhane, E. Del Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, page 193–205. IEEE Press, 2019. ISBN 9781728114361.

S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, S. Zampini, and H. Zhang. PETSc Web page. http://www.mcs.anl.gov/petsc, 2015. URL http://www.mcs.anl.gov/petsc.

V. Bandishti, I. Pananilath, and U. Bondhugula. Tiling stencil computations to maximize parallelism. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2012.

C. Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 7–16. IEEE Computer Society, 2004.

P. Bauer, A. Thorpe, and G. Brunet. The quiet revolution of numerical weather prediction. *Nature*, 525(7567):47–55, 2015.

E. Baysal, D. D. Kosloff, and J. W. Sherwood. Reverse time migration. *Geophysics*, 48(11):1514–1524, 1983.

I. J. Bertolacci, C. Olschanowsky, B. Harshbarger, B. L. Chamberlain, D. G. Wonnacott, and M. M. Strout. Parameterized diamond tiling for stencil computations with chapel parallel iterators. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 197–206, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3559-1. doi: 10.1145/2751205.2751226.

G. Bisbas, F. Luporini, M. Louboutin, R. Nelson, G. J. Gorman, and P. H. Kelly. Temporal blocking of finite-difference stencil operators with sparse "off-the-grid" sources. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 497–506, 2021. doi: 10.1109/IPDPS49936.2021.00058.

S. Blackford, J. Demmel, J. J. Dongarra, I. S. Duff, S. Hammarling, G. M. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. A. Remington, and C. Whaley. Basic linear algebra subprograms (blas). In *Encyclopedia of Parallel Computing*, 2011.

U. Bondhugula. Compiling affine loop nests for distributed-memory parallel architectures. In *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2013.

U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *International Conference on Compiler Construction*, pages 132–146. Springer, 2008a.

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 Implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008b. ACM. ISBN 978-1-59593-860-2. doi: 10.1145/1375581.1375595. URL http://doi.acm.org/10.1145/1375581.1375595.

U. Bondhugula, V. Bandishti, and I. Pananilath. Diamond tiling: Tiling techniques to maximize parallelism for stencil computations. *IEEE Transactions on Parallel and Distributed Systems*, 28(5):1285–1298, 2017. doi: 10.1109/TPDS.2016.2615094.

T. Brandvik and G. Pullan. Sblock: A framework for efficient stencil-based pde solvers on multi-core platforms. In *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*, CIT '10, pages 1181–1188, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4108-2. doi: 10.1109/CIT.2010.214. URL http://dx.doi.org/10.1109/CIT.2010.214.

A. Brown, S. Milton, M. Cullen, B. Golding, J. Mitchell, and A. Shelly. Unified modeling and prediction of weather and climate: A 25-year journey. *Bulletin of the American Meteorological Society*, 93(12):1865 – 1877, 2012. doi: 10.1175/BAMS-D-12-00018.1. URL https://journals.ametsoc.org/view/journals/bams/93/12/bams-d-12-00018.1.xml.

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, pages 1110–1114. Society of Exploration Geophysicists, 2016.

W.-F. Chang and G. A. McMechan. 3d acoustic prestack reverse-time migration1. *Geophysical Prospecting*, 38(7):737–755, 1990.

C. Chen, J. Chame, and M. W. Hall. Chill : A framework for composing high-level loop transformations. 2007.

T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy. TVM: An automated End-to-End optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA, Oct. 2018. USENIX Association. ISBN 978-1-939133-08-3. URL https://www.usenix.org/conference/osdi18/presentation/chen.

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*, IPDPS '11, pages 676–687, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4385-7. doi: 10.1109/IPDPS.2011.70. URL http://dx.doi.org/10.1109/IPDPS.2011.70.

L. Collatz. *The numerical treatment of differential equations*, volume 60. Springer Science & Business Media, 2012.

A. Collins, D. Grewe, V. Grover, S. Lee, and A. Susnea. Nova: A functional language for data parallelism. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY'14, page 8–13, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450329378. doi: 10.1145/2627373.2627375. URL https://doi.org/10.1145/2627373.2627375.

R. Courant, K. Friedrichs, and H. Lewy. On the partial difference equations of mathematical physics. *IBM Journal of Research and Development*, 11(2): 215–234, 1967.

A. Darte. On the complexity of loop fusion. *Parallel Comput.*, 26(9):1175–1193, Aug. 2000. ISSN 0167-8191. doi: 10.1016/S0167-8191(00)00034-X. URL http://dx.doi.org/10.1016/S0167-8191(00)00034-X.

K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 4. IEEE Press, 2008.

Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan. Liszt: a domain specific language for building portable mesh-based pde solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 9:1–9:12, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0771-0. doi: 10.1145/2063384.2063396. URL http://doi.acm.org/10.1145/2063384.2063396.

devitoproject.org. *Devito Documentation*. Imperial College London. URL https://www.devitoproject.org/devito/index.html.

I. S. Duff, M. Heroux, and R. Pozo. An overview of the sparse basic linear algebra subprograms: The new standard from the blas technical forum. *ACM Trans. Math. Softw.*, 28:239–267, 2002.

E. Duveneck and P. M. Bakker. Stable p-wave modeling for reverse-time migration in tilted ti media. *GEOPHYSICS*, 76(2):S65–S75, 2011. doi: 10.1190/1.3533964.

R. Eymard, T. Gallouët, and R. Herbin. Finite volume methods. *Handbook of numerical analysis*, 7:713–1018, 2000.

Firedrake. petsc4py: the Python interface to petsc, Apr. 2016. URL http://dx.doi.org/10.5281/zenodo.49283.

F. Franchetti, T. M. Low, D. T. Popovici, R. M. Veras, D. G. Spampinato, J. R. Johnson, M. Püschel, J. C. Hoe, and J. M. F. Moura. Spiral: Extreme performance portability. *Proceedings of the IEEE*, 106(11):1935–1968, 2018. doi: 10.1109/JPROC.2018.2873289.

M. Frigo and V. Strumpen. Cache oblivious stencil computations. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, pages 361–366, New York, NY, USA, 2005. ACM. ISBN 1-59593-167-8. doi: 10.1145/1088149.1088197.

M. Frigo and V. Strumpen. The cache complexity of multithreaded cache oblivious algorithms. In *Proceedings of the Eighteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '06, page 271–280, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595934529. doi: 10.1145/1148109.1148157.

M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. *ACM Trans. Algorithms*, 8(1), Jan. 2012. ISSN 1549-6325. doi: 10.1145/2071379.2071383.

G. Fursin, Y. Kashnikov, A. W. Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois, et al. Milepost gcc: Machine learning enabled self-tuning compiler. *International journal of parallel programming*, 39(3):296–327, 2011.

S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming*, 34(3):261–317, 2006.

M. Griebl, C. Lengauer, and S. Wetzel. Code generation in the polytope model. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, PACT '98, page 106, USA, 1998. IEEE Computer Society. ISBN 0818685913.

T. Grosser, A. Groesslinger, and C. Lengauer. Polly — performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010, 2012a. doi: 10.1142/S0129626412500107. URL http://www.worldscientific.com/doi/abs/10.1142/S0129626412500107.

T. Grosser, A. Größlinger, and C. Lengauer. Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Process. Lett.*, 22, 2012b.

T. Grosser, A. Cohen, P. H. J. Kelly, J. Ramanujam, P. Sadayappan, and S. Verdoolaege. Split tiling for gpus: Automatic parallelization using trapezoidal tiles. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, page 24–31, New York, NY, USA, 2013a. Association for Computing Machinery. ISBN 9781450320177. doi: 10.1145/2458523.2458526.

T. Grosser, S. Verdoolaege, A. Cohen, and P. Sadayappan. The promises of hybrid hexagonal/classical tiling for gpu. Research Report RR-8339, INRIA, July 2013b. URL https://hal.inria.fr/hal-00848691.

T. Grosser, A. Cohen, J. Holewinski, P. Sadayappan, and S. Verdoolaege. Hybrid hexagonal/classical tiling for gpus. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, page 66–75, New York, NY, USA, 2014a. Association for Computing Machinery. ISBN 9781450326704. doi: 10.1145/2581122.2544160.

T. Grosser, A. Cohen, J. Holewinski, P. Sadayappan, and S. Verdoolaege. Hybrid hexagonal/classical tiling for gpus. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 66:66–66:75, New York, NY, USA, 2014b. ACM. ISBN 978-1-4503-2670-4. doi: 10.1145/2544137.2544160. URL http://doi.acm.org/10.1145/2544137.2544160.

T. Grosser, S. Verdoolaege, A. Cohen, and P. Sadayappan. The relation between diamond tiling and hexagonal tiling. *Parallel Processing Letters*, 24(03):1441002, 2014c.

L. Guasch, O. C. Agudo, M. Tang, P. Nachev, and M. Warner. Full-waveform inversion imaging of the human brain. *bioRxiv*, 2019.

Guohua Jin, J. Mellor-Crummey, and R. Fowler. Increasing temporal locality with skewing and recursive blocking. In *SC '01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, pages 57–57, 2001.

T. Gysi, C. Osuna, O. Fuhrer, M. Bianco, and T. C. Schulthess. Stella: A domain-specific tool for structured grid methods in weather and climate models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, New York, NY, USA,

2015. Association for Computing Machinery. ISBN 9781450337236. doi: 10.1145/2807591.2807627.

T. Gysi, C. Müller, O. Zinenko, S. Herhut, E. Davis, T. Wicky, O. Fuhrer, T. Hoefler, and T. Grosser. Domain-specific multi-level ir rewriting for gpu: The open earth compiler for gpu-accelerated climate simulation. *ACM Trans. Archit. Code Optim.*, 18(4), sep 2021. ISSN 1544-3566. doi: 10.1145/3469030. URL https://doi.org/10.1145/3469030.

B. Hagedorn, L. Stoltzfus, M. Steuwer, S. Gorlatch, and C. Dubach. High performance stencil code generation with lift. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 100–112. ACM, 2018.

J. Hammer, J. Eitzinger, G. Hager, and G. Wellein. Kerncraft: A tool for analytic performance modeling of loop kernels. In C. Niethammer, J. Gracia, T. Hilbrich, A. Knüpfer, M. M. Resch, and W. E. Nagel, editors, *Tools for High Performance Computing 2016*, pages 1–22, Cham, 2017. Springer International Publishing. ISBN 978-3-319-56702-0.

A. Hartono, Q. Lu, X. Gao, S. Krishnamoorthy, M. Nooijen, G. Baumgartner, D. E. Bernholdt, V. Choppella, R. M. Pitzer, J. Ramanujam, A. Rountev, and P. Sadayappan. Identifying cost-effective common subexpressions to reduce operation count in tensor contraction evaluations. In *Proceedings of the 6th International Conference on Computational Science - Volume Part I*, ICCS'06, pages 267–275, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-34379-2, 978-3-540-34379-0. doi: 10.1007/11758501_39. URL http://dx.doi.org/10.1007/11758501_39.

A. Hartono, Q. Lu, T. Henretty, S. Krishnamoorthy, H. Zhang, G. Baumgartner, D. E. Bernholdt, M. Nooijen, R. Pitzer, J. Ramanujam, and P. Sadayappan. Performance optimization of tensor contraction expressions for many-body methods in quantum chemistry†. *The Journal of Physical Chemistry A*, 113(45):12715–12723, 2009. doi: 10.1021/jp9051215. URL http://pubs.acs.org/doi/abs/10.1021/jp9051215. PMID: 19888780.

K. Hawick and D. Playne. Simulation software generation using a domain-specific language for partial differential field equations. In *Proceedings of the International Conference on Software Engineering Research and Practice*

(SERP), page 1. The Steering Committee of The World Congress in Computer Science, Computer . . . , 2013.

F. Hecht. New development in freefem++. *J. Numer. Math.*, 20(3-4):251–265, 2012. ISSN 1570-2820. URL https://freefem.org/.

A. Heinecke, G. Henry, M. Hutchinson, and H. Pabst. LIBXSMM: accelerating small matrix multiplications by runtime code generation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 84:1–84:11, Piscataway, NJ, USA, 2016. IEEE Press. ISBN 978-1-4673-8815-3. URL http://dl.acm.org/citation.cfm?id=3014904.3015017.

J. L. Hennessy and D. A. Patterson. *Computer Architecture, Sixth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 6th edition, 2017. ISBN 0128119055.

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*, ICS '13, pages 13–24, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2130-3. doi: 10.1145/2464996.2467268. URL http://doi.acm.org/10.1145/2464996.2467268.

J. Holewinski, L.-N. Pouchet, and P. Sadayappan. High-performance code generation for stencil computations on gpu architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 311–320, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1316-2. doi: 10.1145/2304576.2304619. URL http://doi.acm.org/10.1145/2304576.2304619.

H. T. Huynh. A flux reconstruction approach to high-order schemes including discontinuous galerkin methods. In *18th AIAA computational fluid dynamics conference*, page 4079, 2007.

A. Ilic, F. Pratas, and L. Sousa. Cache-aware roofline model: Upgrading the loft. *IEEE Computer Architecture Letters*, 13(1):21–24, 2014. doi: 10.1109/L-CA.2013.6.

C. T. Jacobs, S. P. Jammy, and N. Sandham. Opensbli: A framework for the automated derivation and parallel execution of finite difference solvers on a range of computer architectures. *J. Comput. Sci.*, 18:12–23, 2017.

M. Jacquelin, M. Araya-Polo, and J. Meng. Massively scalable stencil algorithm. *arXiv preprint arXiv:2204.03775*, 2022.

R. M. Karp, R. E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *J. ACM*, 14(3):563–590, jul 1967. ISSN 0004-5411. doi: 10.1145/321406.321418. URL https://doi.org/10.1145/321406.321418.

D. Kim, L. Renganarayanan, D. Rostron, S. Rajopadhye, and M. M. Strout. Multi-level tiling: M for the price of one. In *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, pages 1–12, 2007. doi: 10.1145/1362622.1362691.

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., 2014. Association for Computing Machinery. doi: 10.1145/2627373.2627387.

A. Klöckner. Loo.py: Transformation-based code generation for gpus and cpus. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY'14, pages 82:82–82:87, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2937-8. doi: 10.1145/2627373.2627387. URL http://doi.acm.org/10.1145/2627373.2627387.

A. Klöckner. Loo.py: From fortran to performance via transformation and substitution rules. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY 2015, pages 1–6, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3584-3. doi: 10.1145/2774959.2774969. URL http://doi.acm.org/10.1145/2774959.2774969.

S. Krishnamoorthy, M. Baskaran, and U. B. Ramanujam. Effective automatic parallelization of stencil computations. In *In ACM SIGPLAN PLDI 2007*, 2007.

S. Kuroda, T. Endo, and S. Matsuoka. Applying temporal blocking with a directive-based approach. In *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM-HPC'17, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450355650. doi: 10.1145/3148173.3148190. URL https://doi.org/10.1145/3148173.3148190.

C.-C. Lam, T. Rauber, G. Baumgartner, D. Cociorva, and P. Sadayappan. Memory-optimal evaluation of expression trees involving large objects. *Comput. Lang. Syst. Struct.*, 37(2):63–75, July 2011. ISSN 1477-8424. doi: 10.1016/j.cl.2010.09.003. URL http://dx.doi.org/10.1016/j.cl.2010.09.003.

C. Lameter. Numa (non-uniform memory access): An overview: Numa becomes more common because memory controllers get close to execution units on microprocessors. *Queue*, 11(7):40–51, jul 2013. ISSN 1542-7730. doi: 10.1145/2508834.2513149. URL https://doi.org/10.1145/2508834.2513149.

L. Lamport. The parallel execution of do loops. *Commun. ACM*, 17(2):83–93, Feb. 1974. ISSN 0001-0782. doi: 10.1145/360827.360844.

H. P. Langtangen and S. Linge. *Finite difference computing with PDEs: a modern software approach*. Springer Nature, 2017.

S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 145–156, New York, NY, USA, 2000. ACM. ISBN 1-58113-199-2. doi: 10.1145/349299.349320. URL http://doi.acm.org/10.1145/349299.349320.

C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, 2004. doi: 10.1109/CGO.2004.1281665.

C. Lattner, J. A. Pienaar, M. Amini, U. Bondhugula, R. Riddle, A. Cohen, T. Shpeisman, A. Davis, N. Vasilache, and O. Zinenko. Mlir: A compiler infrastructure for the end of moore's law. *ArXiv*, abs/2002.11054, 2020.

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 L. Lopes, J. Žilinskas, A. Costan, R. G. Cascella, G. Kecskemeti, E. Jeannot, M. Cannataro, L. Ricci, S. Benkner, S. Petit, V. Scarano, J. Gracia, S. Hunold, S. L. Scott, S. Lankes, C. Lengauer, J. Carretero, J. Breitbart, and M. Alexander, editors, *Euro-Par 2014: Parallel Processing Workshops*, pages 553–564, Cham, 2014. Springer International Publishing. ISBN 978-3-319-14313-2.

A. R. Levander. Fourth-order finite-difference p-sv seismograms. *Geophysics*, 53(11):1425–1436, 1988.

V. Levchenko and A. Perepelkina. The diamondtetris algorithm for maximum performance vectorized stencil computation. In V. Malyshkin, editor, *Parallel Computing Technologies*, pages 124–135, Cham, 2017. Springer International Publishing. ISBN 978-3-319-62932-2.

R. J. LeVeque. *Finite difference methods for ordinary and partial differential equations: steady-state and time-dependent problems*, volume 98. SIAM, 2007.

H. Levy and F. Lessman. *Finite difference equations*. Courier Corporation, 1992.

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. doi: 10.1007/978-3-642-23099-8.

M. Louboutin, M. Lange, F. J. Herrmann, N. Kukreja, and G. Gorman. Performance prediction of finite-difference solvers for different computer architectures. *Computers & Geosciences*, 105:148–157, 2017. ISSN 0098-3004. doi: https://doi.org/10.1016/j.cageo.2017.04.014. URL https://www.sciencedirect.com/science/article/pii/S0098300416304034.

M. Louboutin, P. Witte, and F. J. Herrmann. Effects of wrong adjoints for rtm in tti media. In *SEG Technical Program Expanded Abstracts 2018*, pages 331–335. Society of Exploration Geophysicists, 2018.

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*, 12(3):1165–1187, 2019. doi: 10.5194/gmd-12-1165-2019.

F. Luporini. *Automated optimization of numerical methods for partial differential equations*. Phd thesis, Imperial College London, 2016.

F. Luporini, M. Lange, C. T. Jacobs, G. J. Gorman, J. Ramanujam, and P. H. Kelly. Automated tiling of unstructured mesh computations with application to seismological modeling. *ACM Transactions on Mathematical Software (TOMS)*, 45(2):17, 2019.

F. Luporini, M. Louboutin, M. Lange, N. Kukreja, P. Witte, J. Hückelheim, C. Yount, P. H. J. Kelly, F. J. Herrmann, and G. J. Gorman. Architecture and performance of devito, a system for automated stencil computation. *ACM Trans. Math. Softw.*, 46(1), Apr. 2020. ISSN 0098-3500. doi: 10.1145/3374916.

F. Luporini, M. Louboutin, M. Lange, N. Kukreja, R. Nelson, G. Bisbas, V. Pandolfo, L. Cavalcante, tjb900, G. Gorman, K. Hester, O. Mojica, V. Mickus, V. Leite, EdCaunt, M. Bruno, P. Kazakas, P. Nogueira, jack lascala, C. Dinneen, J. H. Speglich, G. S. von Conta, T. Greaves, SSHz, J. F. de Souza, I. Assis, felipeaugustogudes, L. Rami, and R. Cristo. devitocodes/devito: v4.6. https://doi.org/10.5281/zenodo.5552589, October 2021.

T. Malas, G. Hager, H. Ltaief, H. Stengel, G. Wellein, and D. Keyes. Multicore-optimized wavefront diamond blocking for optimizing stencil updates. *SIAM Journal on Scientific Computing*, 37(4):C439–C464, 2015.

K. Matsumura, H. R. Zohouri, M. Wahib, T. Endo, and S. Matsuoka. An5d: Automated stencil framework for high-degree temporal blocking on gpus. *arXiv preprint arXiv:2001.01473*, 2020.

J. Meng and K. Skadron. Performance modeling and automatic ghost zone optimization for iterative stencil loops on gpus. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages

256–265, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-498-0. doi: 10.1145/1542275.1542313. URL http://doi.acm.org/10.1145/1542275.1542313.

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, v. Roučka, A. Saboo, I. Fernando, S. Kulal, R. Cimrman, and A. Scopatz. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, Jan. 2017. ISSN 2376-5992. doi: 10.7717/peerj-cs.103. URL https://doi.org/10.7717/peerj-cs.103.

P. Micikevicius. 3d finite difference computation on gpus using cuda. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, page 79–84, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605585178. doi: 10.1145/1513895.1513905. URL https://doi.org/10.1145/1513895.1513905.

S. S. Muchnick. *Advanced Compiler Design and Implementation*. 1997.

G. Mudalige, M. Giles, I. Reguly, C. Bertolli, and P. Kelly. Op2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures. In *2012 Innovative Parallel Computing (InPar)*, pages 1–12, 2012. doi: 10.1109/InPar.2012.6339594.

R. T. Mullapudi, V. Vasista, and U. Bondhugula. Polymage: Automatic optimization for image processing pipelines. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, page 429–443, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450328357. doi: 10.1145/2694344.2694364. URL https://doi.org/10.1145/2694344.2694364.

T. Muranushi and J. Makino. Optimal temporal blocking for stencil computation. *Procedia Computer Science*, 51:1303–1312, 2015.

T. Muranushi, S. Nishizawa, H. Tomita, K. Nitadori, M. Iwasawa, Y. Maruyama, H. Yashiro, Y. Nakamura, H. Hotta, J. Makino, N. Hosono,

and H. Inoue. Automatic generation of efficient codes from mathematical descriptions of stencil computation. In *Proceedings of the 5th International Workshop on Functional High-Performance Computing*, FHPC 2016, page 17–22, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450344333. doi: 10.1145/2975991.2975994. URL https://doi.org/10.1145/2975991.2975994.

NVIDIA. *Basic Linear Algebra on NVIDIA GPUs (cuBLAS)*, 2022. [Online]. Available: https://developer.nvidia.com/cublas.

T. Odajima, Y. Kodama, M. Tsuji, M. Matsuda, Y. Maruyama, and M. Sato. Preliminary performance evaluation of the fujitsu a64fx using hpc applications. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 523–530, 2020. doi: 10.1109/CLUSTER49012.2020.00075.

C. Palenzuela, B. Miñano, A. Arbona, C. Bona-Casas, C. Bona, and J. Massó. Simflowny 3: An upgraded platform for scientific modeling and simulation. *Computer Physics Communications*, 259: 107675, 2021. ISSN 0010-4655. doi: https://doi.org/10.1016/j.cpc.2020.107675. URL https://www.sciencedirect.com/science/article/pii/S0010465520303271.

A. T. Patera. A spectral element method for fluid dynamics: Laminar flow in a channel expansion. *Journal of Computational Physics*, 54(3):468–488, 1984. ISSN 0021-9991. doi: https://doi.org/10.1016/0021-9991(84)90128-1. URL https://www.sciencedirect.com/science/article/pii/0021999184901281.

A. Peirce. Solving the heat, laplace and wave equations using finite difference methods, 2018. URL https://personal.math.ubc.ca/~peirce/M257_316_2012_Lecture_8.pdf.

C. Poon and G. Peyré. Degrees of freedom for off-the-grid sparse estimation. *arXiv preprint arXiv:1911.03577*, 2019.

M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

L. Qu, R. Abdelkhalak, H. Ltaief, I. Said, and D. E. Keyes. High performance asynchronous reverse time migration for oil and gas exploration. 2020.

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*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2014-6. doi: 10.1145/2491956.2462176. URL http://doi.acm.org/10.1145/2491956.2462176.

J. Ragan-Kelley, A. Adams, D. Sharlet, C. Barnes, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand. Halide: Decoupling algorithms from schedules for high-performance image processing. *Commun. ACM*, 61 (1):106–115, dec 2017. ISSN 0001-0782. doi: 10.1145/3150211. URL https://doi.org/10.1145/3150211.

J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for nonshared memory machines. In *Supercomputing '91:Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, pages 111–120, 1991.

J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for multicomputers. *Journal of Parallel and Distributed Computing*, 16(2):108–120, 1992. ISSN 0743-7315. doi: https://doi.org/10.1016/0743-7315(92)90027-K. URL https://www.sciencedirect.com/science/article/pii/074373159290027K.

F. Rathgeber. *Productive and efficient computational science through domain-specific abstractions*. PhD thesis, Imperial College London, 2014.

F. Rathgeber, L. Mitchell, F. Luporini, G. Markall, D. A. Ham, G.-T. Bercea, M. Homolya, A. T. T. McRae, H. Dearman, C. T. Jacobs, gbts, S. W. Funke, K. Sato, and F. Russell. PyOP2: framework for performance-portable parallel computations on unstructured meshes, Apr. 2016. URL http://dx.doi.org/10.5281/zenodo.49281.

F. Rathgeber, D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. McRae, G.-T. Bercea, G. R. Markall, and P. H. Kelly. Firedrake: automating the finite element method by composing abstractions. *ACM Transactions on Mathematical Software (TOMS)*, 43(3):24, 2017.

M. Ravishankar, J. Holewinski, and V. Grover. Forma: A dsl for image processing applications to target gpus and multi-core cpus. In *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs*, GPGPU-8, page 109–120, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450334075. doi: 10.1145/2716282.2716290. URL https://doi.org/10.1145/2716282.2716290.

P. Rawat, M. Kong, T. Henretty, J. Holewinski, K. Stock, L.-N. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan. Sdslc: A multi-target domain-specific compiler for stencil computations. In *Proceedings of the 5th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, WOLFHPC '15, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450340168. doi: 10.1145/2830018.2830025. URL https://doi.org/10.1145/2830018.2830025.

P. S. Rawat, M. Vaidya, A. Sukumaran-Rajam, M. Ravishankar, V. Grover, A. Rountev, L. N. Pouchet, and P. Sadayappan. Domain-Specific Optimization and Generation of High-Performance GPU Code for Stencil Computations. *Proceedings of the IEEE*, 106(11):1902–1920, 2018. ISSN 00189219. doi: 10.1109/JPROC.2018.2862896.

I. Z. Reguly, G. R. Mudalige, C. Bertolli, M. B. Giles, A. Betts, P. H. J. Kelly, and D. Radford. Acceleration of a full-scale industrial cfd application with op2. *IEEE Transactions on Parallel and Distributed Systems*, 27(5): 1265–1278, May 2016. ISSN 1045-9219. doi: 10.1109/TPDS.2015.2453972.

I. Z. Reguly, G. R. Mudalige, and M. B. Giles. Loop tiling in large-scale stencil codes at run-time with ops. *IEEE Transactions on Parallel and Distributed Systems*, 29(4):873–886, 2018. doi: 10.1109/TPDS.2017.2778161.

G. Rodríguez and L.-N. Pouchet. Polyhedral modeling of immutable sparse matrices. In *8th International Workshop on Polyhedral Compilation Techniques. Manchester, UK*, 2018.

R. Sai, J. Mellor-Crummey, X. Meng, M. Araya-Polo, and J. Meng. Accelerating high-order stencils on gpus. In *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 86–108, 2020. doi: 10.1109/PMBS51919.2020.00014.

A. Schäfer and D. Fey. High performance stencil code algorithms for GPGPUs. *Procedia Computer Science*, 4:2027–2036, 2011. ISSN 18770509. doi: 10.1016/j.procs.2011.04.221.

G. Smith. *Numerical Solution of Partial DifferentialEquations: Finite Difference Methods*. Clarendon, 1985.

M. Steuwer, C. Fensch, S. Lindley, and C. Dubach. Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance opencl code. *SIGPLAN Not.*, 50(9):205–217, aug 2015. ISSN 0362-1340. doi: 10.1145/2858949.2784754. URL https://doi.org/10.1145/2858949.2784754.

M. Steuwer, T. Remmelg, and C. Dubach. Matrix multiplication beyond auto-tuning: Rewrite-based gpu code generation. In *2016 International Conference on Compliers, Architectures, and Sythesis of Embedded Systems (CASES)*, pages 1–10, 2016. doi: 10.1145/2968455.2968521.

M. Steuwer, T. Remmelg, and C. Dubach. Lift: A functional data-parallel ir for high-performance gpu code generation. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 74–85, 2017. doi: 10.1109/CGO.2017.7863730.

K. Stock, M. Kong, T. Grosser, L.-N. Pouchet, F. Rastello, J. Ramanujam, and P. Sadayappan. A framework for enhancing data reuse via associative reordering. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 65–76, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327848. doi: 10.1145/2594291.2594342.

M. M. Strout, M. Hall, and C. Olschanowsky. The sparse polyhedral framework: Composing compiler-generated inspector-executor code. *Proceedings of the IEEE*, 106(11):1921–1934, 2018. doi: 10.1109/JPROC.2018.2857721.

R. Strzodka, M. Shaheen, D. Pajak, and H.-P. Seidel. Cache accurate time skewing in iterative stencil computations. In *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP '11, pages 571–581, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4510-3. doi: 10.1109/ICPP.2011.47.

A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Trans. Embed. Comput. Syst.*, 13(4s), apr 2014. ISSN 1539-9087. doi: 10.1145/2584665. URL https://doi.org/10.1145/2584665.

H. Tanaka, Y. Ishihara, R. Sakamoto, T. Nakamura, Y. Kimura, K. Nitadori, M. Tsubouchi, and J. Makino. Automatic generation of high-order finite-difference code with temporal blocking for extreme-scale many-core systems. In *2018 IEEE/ACM 4th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*, pages 29–36. IEEE, 2018.

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*, SPAA '11, pages 117–128, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0743-7. doi: 10.1145/1989493.1989508. URL http://doi.acm.org/10.1145/1989493.1989508.

F. Thaler, S. Moosbrugger, C. Osuna, M. Bianco, H. Vogt, A. Afanasyev, L. Mosimann, O. Fuhrer, T. C. Schulthess, and T. Hoefler. Porting the cosmo weather model to manycore cpus. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, PASC '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367707. doi: 10.1145/3324989.3325723.

J. W. Thomas. *Numerical partial differential equations: finite difference methods*, volume 22. Springer Science & Business Media, 2013.

D. Unat, J. Zhou, Y. Cui, S. B. Baden, and X. Cai. Accelerating a 3d finite-difference earthquake simulation with a c-to-cuda translator. *Computing in Science and Engg.*, 14(3):48–59, May 2012. ISSN 1521-9615. doi: 10.1109/MCSE.2012.44. URL http://dx.doi.org/10.1109/MCSE.2012.44.

D. Unat, A. Dubey, T. Hoefler, J. Shalf, M. Abraham, M. Bianco, B. L. Chamberlain, R. Cledat, H. C. Edwards, H. Finkel, K. Fuerlinger, F. Hannig, E. Jeannot, A. Kamil, J. Keasler, P. H. J. Kelly, V. Leung, H. Ltaief, N. Maruyama, C. J. Newburn, and M. Pericás. Trends in data locality

abstractions for hpc systems. *IEEE Transactions on Parallel and Distributed Systems*, 28(10):3007–3020, 2017. doi: 10.1109/TPDS.2017.2703149.

A. Van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.

F. G. Van Zee and R. A. van de Geijn. BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Transactions on Mathematical Software*, 41(3):14:1–14:33, June 2015. URL http://doi.acm.org/10.1145/2764454.

N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*, 2018.

A. Venkat, M. Shantharam, M. Hall, and M. M. Strout. Non-affine extensions to polyhedral code generation. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 185:185–185:194, New York, NY, USA, 2014a. ACM. ISBN 978-1-4503-2670-4. doi: 10.1145/2544137.2544141. URL http://doi.acm.org/10.1145/2544137.2544141.

A. Venkat, M. Shantharam, M. Hall, and M. M. Strout. Non-affine extensions to polyhedral code generation. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, page 185–194, New York, NY, USA, 2014b. Association for Computing Machinery. ISBN 9781450326704. doi: 10.1145/2581122.2544141. URL https://doi.org/10.1145/2581122.2544141.

S. Verdoolaege. isl: An integer set library for the polyhedral model. In K. Fukuda, J. Hoeven, M. Joswig, and N. Takayama, editors, *Mathematical Software (ICMS'10)*, LNCS 6327, pages 299–302. Springer-Verlag, 2010.

S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor. Polyhedral parallel code generation for cuda. *ACM Trans. Archit. Code Optim.*, 9(4), jan 2013. ISSN 1544-3566. doi: 10.1145/2400682.2400713. URL https://doi.org/10.1145/2400682.2400713.

J. Virieux. P-sv wave propagation in heterogeneous media: Velocity-stress finite-difference method. *Geophysics*, 51(4):889–901, 1986.

J. Virieux and S. Operto. An overview of full-waveform inversion in exploration geophysics. *Geophysics*, 74(6):WCC1–WCC26, 2009.

E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang. *Intel Math Kernel Library*, pages 167–188. Springer International Publishing, Cham, 2014. ISBN 978-3-319-06486-4. doi: 10.1007/978-3-319-06486-4_7. URL https://doi.org/10.1007/978-3-319-06486-4_7.

H. Wang and A. Chandramowlishwaran. Pencil: A pipelined algorithm for distributed stencils. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16, 2020. doi: 10.1109/SC41405.2020.00089.

G. Wellein, G. Hager, T. Zeiser, M. Wittmann, and H. Fehske. Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization. *Proceedings - International Computer Software and Applications Conference*, 1:579–586, 2009. ISSN 07303157. doi: 10.1109/COMPSAC.2009.82.

S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4): 65–76, Apr. 2009. ISSN 0001-0782. doi: 10.1145/1498765.1498785. URL http://doi.acm.org/10.1145/1498765.1498785.

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*, 185(11):3028–3040, 2014. ISSN 0010-4655. doi: https://doi.org/10.1016/j.cpc.2014.07.011. URL https://www.sciencedirect.com/science/article/pii/S0010465514002549.

M. Wittmann, G. Hager, and G. Wellein. Multicore-aware parallel temporal blocking of stencil codes for shared and distributed memory. In *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–7, 2010.

M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. *ACM SIGPLAN Notices*, 26(6):30–44, 1991. ISSN 03621340. doi: 10.1145/113446.113449.

M. Wolfe. Loop skewing: The wavefront method revisited. *Int. J. Parallel Program.*, 15(4):279–293, Oct. 1986. ISSN 0885-7458. doi: 10.1007/BF01407876.

M. Wolfe. More iteration space tiling. *Proceedings of the 1989 ACM/IEEE conference on Supercomputing - Supercomputing '89*, pages 655–664, 1989. ISSN 1949-1042. doi: 10.1145/76263.76337.

D. Wonnacott. Using time skewing to eliminate idle time due to memory bandwidth and network limitations. *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*, pages 171–180, May 2000. doi: 10.1109/IPDPS.2000.845979.

D. Wonnacott. Achieving scalable locality with time skewing. *International Journal of Parallel Programming*, 30:181–221, 2004.

J. Xue. On tiling as a loop transformation. *Parallel Processing Letters*, 7(04): 409–424, 1997.

H. Yan, J. Xu, and X. Zhang. Compressed sensing radar imaging of off-grid sparse targets. In *2015 IEEE Radar Conference (RadarCon)*, pages 0690–0693, 2015.

Q. Yi. Poet: a scripting language for applying parameterized source-to-source program transformations. *Software: Practice and Experience*, 42(6): 675–706, 2012.

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*, pages 865–870. IEEE, 2015.

C. Yount and A. Duran. Effective use of large high-bandwidth memory caches in hpc stencil computation via temporal wave-front tiling. In

2016 7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), pages 65–75, 2016.

C. Yount, J. Tobin, A. Breuer, and A. Duran. YASK - Yet another stencil kernel: A framework for HPC stencil code-generation and tuning. *Proceedings of WOLFHPC 2016: 6th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing - Held in conjunction with SC 2016: The International Conference for High Performance Computing, Networking, Stor*, (1):30–39, 2017. doi: 10.1109/WOLFHPC.2016.08.

L. Yuan, S. Huang, Y. Zhang, and H. Cao. Tessellating star stencils. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP 2019, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362955. doi: 10.1145/3337821.3337835. URL https://doi.org/10.1145/3337821.3337835.

T. Yuki, G. Gupta, D. Kim, T. Pathan, and S. Rajopadhye. Alphaz: A system for design space exploration in the polyhedral model. In H. Kasahara and K. Kimura, editors, *Languages and Compilers for Parallel Computing*, pages 17–31, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-37658-0.

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*, CGO '12, pages 155–164, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1206-6. doi: 10.1145/2259016.2259037. URL http://doi.acm.org/10.1145/2259016.2259037.

Y. Zhang, H. Zhang, and G. Zhang. A stable tti reverse time migration and its implementation. *Geophysics*, 76(3):WA3–WA11, 2011.

T. Zhao, M. Hall, P. Basu, S. Williams, and H. Johansen. Simd code generation for stencils on brick decompositions. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, page 423–424, New York, NY, USA, 2018a. Association for Computing Machinery. ISBN 9781450349826. doi: 10.1145/3178487.3178537. URL https://doi.org/10.1145/3178487.3178537.

T. Zhao, S. Williams, M. Hall, and H. Johansen. Delivering performance-portable stencil computations on cpus and gpus using bricks. In *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 59–70. IEEE, 2018b.

T. Zhao, P. Basu, S. Williams, M. Hall, and H. Johansen. Exploiting reuse and vectorization in blocked stencil computations on cpus and gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362290. doi: 10.1145/3295500.3356210. URL https://doi.org/10.1145/3295500.3356210.

O. C. Zienkiewicz, R. L. Taylor, P. Nithiarasu, and J. Zhu. *The finite element method*, volume 3. McGraw-Hill London, 1977.

H. R. Zohouri, A. Podobas, and S. Matsuoka. Combined spatial and temporal blocking for high-performance stencil computation on fpgas using opencl. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '18, page 153–162, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356145. doi: 10.1145/3174243.3174248.

Appendix A

Appendix

A.1. An example for deriving the size of errors in finite difference approximations

This section uses a simple example to show some errors in finite-difference approximations and plots these errors on a log-log scale.

Let $u(x) = \sin(x)$ represent a function of one variable that is smooth and $\bar{x} = 1$; a particular point of interest, within an interval where the function is well-defined. We are trying to approximate $u'(1) = \cos(1) = 0.5403023$. Table A.1 shows the error $\Delta u(\bar{x}) - u'(\bar{x})$ for various values of h for each of the formulas 2.1, 2.2, 2.3, 2.4. We see that $\Delta_+ u$ and $\Delta_- u$ behave similarly, although one exhibits an error roughly the negative of the other. This is reasonable from Figure 2.1 and explains why $D_0 u$, the average of the two, has an error much smaller than both.

Table A.1.: Errors in various finite difference approximations

h	$\Delta_+ u(\bar{x})$	$\Delta_- u(\bar{x})$	$\Delta_0 u(\bar{x})$	$\Delta_3 u(\bar{x})$
1.0e − 01	−4.2939e − 02	4.1138e − 02	−9.0005e − 04	6.8207e − 05
5.0e − 02	−2.1257e − 02	2.0807e − 02	−2.2510e − 04	8.6491e − 06
1.0e − 02	−4.2163e − 03	4.1983e − 03	−9.0050e − 06	6.9941e − 08
5.0e − 03	−2.1059e − 03	2.1014e − 03	−2.2513e − 06	8.7540e − 09
1.0e − 03	−4.2083e − 04	4.2065e − 04	−9.0050e − 08	6.9979e − 11

We see that:

$$\Delta_+ u(\bar{x}) - u'(\bar{x}) \approx -0.42h,$$
$$\Delta_0 u(\bar{x}) - u'(\bar{x}) \approx -0.09h^2,$$
$$\Delta_3 u(\bar{x}) - u'(\bar{x}) \approx 0.007h^3,$$

confirming that these methods are first-order, second-order, and third-order accurate, respectively.

Figure A.1 shows these errors plotted against h on a log-log scale. This is a way to plot errors when we expect them to behave like some power of h since if the error $E(h)$ behaves like

The errors are:

$$E(h) \approx Ch^p,$$

then

$$\log |E(h)| \approx \log |C| + p \log h.$$

So on a log-log scale, the error behaves linearly with a slope equal to p, the order of accuracy.

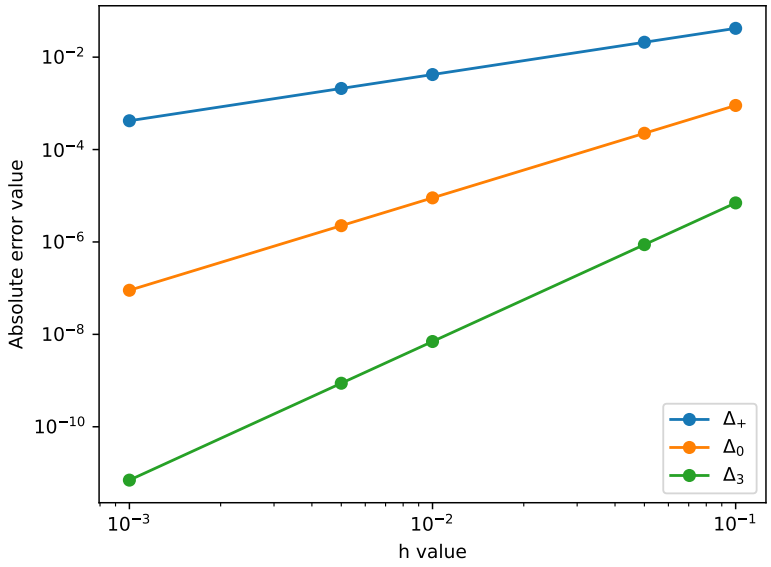


Figure A.1.: The absolute errors in $\Delta u(\bar{x}) - u'(\bar{x})$ from Table A.1 plotted against h on a log-log scale. Adapted and edited from [LeVeque, 2007]

A.2. Devito generated code for wave propagators

A.2.1. Devito generated code for TTI wave propagators

The following code shows the optimized generated C-code for the TTI wave propagation.

Listing A.1: An optimised version of Devito generated code for a TTI forward wave propagators stencil.

```
1   int ForwardTTI(struct dataobj *restrict damp_vec, struct
        dataobj *restrict delta_vec, struct dataobj *restrict
        epsilon_vec, struct dataobj *restrict phi_vec, struct
        dataobj *restrict rec_vec, struct dataobj *restrict
        rec_coords_vec, struct dataobj *restrict src_vec, struct
        dataobj *restrict src_coords_vec, struct dataobj *restrict
         theta_vec, struct dataobj *restrict u_vec, struct dataobj
         *restrict v_vec, struct dataobj *restrict vp_vec, const
        int x_M, const int x_m, const int y_M, const int y_m,
        const int z_M, const int z_m, const float dt, const float
        o_x, const float o_y, const float o_z, const int p_rec_M,
        const int p_rec_m, const int p_src_M, const int p_src_m,
        const int time_M, const int time_m, const int x0_blk0_size
        , const int y0_blk0_size, const int nthreads, const int
        nthreads_nonaffine, const int z_size, const int x_size,
        const int y_size, struct profiler * timers)
2   {
3     float **pr22_vec;
4     posix_memalign((void**)(&pr22_vec),64,nthreads*sizeof(float
          *));
5     float **pr23_vec;
6     posix_memalign((void**)(&pr23_vec),64,nthreads*sizeof(float
          *));
7     float *r16_vec;
8     posix_memalign((void**)(&r16_vec),64,(x_size + 1)*(y_size +
          1)*(z_size + 1)*sizeof(float));
9     float *r17_vec;
10    posix_memalign((void**)(&r17_vec),64,(x_size + 1)*(y_size +
          1)*(z_size + 1)*sizeof(float));
11    float *r18_vec;
12    posix_memalign((void**)(&r18_vec),64,(x_size + 1)*(y_size +
          1)*(z_size + 1)*sizeof(float));
13    float *r19_vec;
14    posix_memalign((void**)(&r19_vec),64,(x_size + 1)*(y_size +
          1)*(z_size + 1)*sizeof(float));
15    #pragma omp parallel num_threads(nthreads)
16    {
17      const int tid = omp_get_thread_num();
18      posix_memalign((void**)(&(pr22_vec[tid])),64,(x0_blk0_size
            + 1)*(y0_blk0_size + 1)*(z_size + 1)*sizeof(float));
19      posix_memalign((void**)(&(pr23_vec[tid])),64,(x0_blk0_size
            + 1)*(y0_blk0_size + 1)*(z_size + 1)*sizeof(float));
20    }
21
22    float (*restrict damp)[damp_vec->size[1]][damp_vec->size[2]]
          __attribute__ ((aligned (64))) = (float (*)[damp_vec->
        size[1]][damp_vec->size[2]]) damp_vec->data;
```

```
23      float (*restrict delta)[delta_vec->size[1]][delta_vec->size
            [2]] __attribute__ ((aligned (64))) = (float (*)[
            delta_vec->size[1]][delta_vec->size[2]]) delta_vec->data
            ;
24      float (*restrict epsilon)[epsilon_vec->size[1]][epsilon_vec
            ->size[2]] __attribute__ ((aligned (64))) = (float (*)[
            epsilon_vec->size[1]][epsilon_vec->size[2]]) epsilon_vec
            ->data;
25      float (*restrict phi)[phi_vec->size[1]][phi_vec->size[2]]
            __attribute__ ((aligned (64))) = (float (*)[phi_vec->
            size[1]][phi_vec->size[2]]) phi_vec->data;
26      float **pr22 = (float**) pr22_vec;
27      float **pr23 = (float**) pr23_vec;
28      float (*restrict r16)[y_size + 1][z_size + 1] __attribute__
            ((aligned (64))) = (float (*)[y_size + 1][z_size + 1])
            r16_vec;
29      float (*restrict r17)[y_size + 1][z_size + 1] __attribute__
            ((aligned (64))) = (float (*)[y_size + 1][z_size + 1])
            r17_vec;
30      float (*restrict r18)[y_size + 1][z_size + 1] __attribute__
            ((aligned (64))) = (float (*)[y_size + 1][z_size + 1])
            r18_vec;
31      float (*restrict r19)[y_size + 1][z_size + 1] __attribute__
            ((aligned (64))) = (float (*)[y_size + 1][z_size + 1])
            r19_vec;
32      float (*restrict rec)[rec_vec->size[1]] __attribute__ ((
            aligned (64))) = (float (*)[rec_vec->size[1]]) rec_vec->
            data;
33      float (*restrict rec_coords)[rec_coords_vec->size[1]]
            __attribute__ ((aligned (64))) = (float (*)[
            rec_coords_vec->size[1]]) rec_coords_vec->data;
34      float (*restrict src)[src_vec->size[1]] __attribute__ ((
            aligned (64))) = (float (*)[src_vec->size[1]]) src_vec->
            data;
35      float (*restrict src_coords)[src_coords_vec->size[1]]
            __attribute__ ((aligned (64))) = (float (*)[
            src_coords_vec->size[1]]) src_coords_vec->data;
36      float (*restrict theta)[theta_vec->size[1]][theta_vec->size
            [2]] __attribute__ ((aligned (64))) = (float (*)[
            theta_vec->size[1]][theta_vec->size[2]]) theta_vec->data
            ;
37      float (*restrict u)[u_vec->size[1]][u_vec->size[2]][u_vec->
            size[3]] __attribute__ ((aligned (64))) = (float (*)[
            u_vec->size[1]][u_vec->size[2]][u_vec->size[3]]) u_vec->
            data;
38      float (*restrict v)[v_vec->size[1]][v_vec->size[2]][v_vec->
            size[3]] __attribute__ ((aligned (64))) = (float (*)[
            v_vec->size[1]][v_vec->size[2]][v_vec->size[3]]) v_vec->
            data;
39      float (*restrict vp)[vp_vec->size[1]][vp_vec->size[2]]
            __attribute__ ((aligned (64))) = (float (*)[vp_vec->size
            [1]][vp_vec->size[2]]) vp_vec->data;
40
41      /* Flush denormal numbers to zero in hardware */
42      _MM_SET_DENORMALS_ZERO_MODE(_MM_DENORMALS_ZERO_ON);
43      _MM_SET_FLUSH_ZERO_MODE(_MM_FLUSH_ZERO_ON);
44
45      float r20 = 1.0F/(dt*dt);
46      float r21 = 1.0F/dt;
```

```
47
48        /* Begin section0 */
49        START_TIMER(section0)
50        #pragma omp parallel num_threads(nthreads)
51        {
52          #pragma omp for collapse(2) schedule(static,1)
53          for (int x = x_m - 1; x <= x_M; x += 1)
54          {
55            for (int y = y_m - 1; y <= y_M; y += 1)
56            {
57              #pragma omp simd aligned(delta,phi,theta:32)
58              for (int z = z_m - 1; z <= z_M; z += 1)
59              {
60                r16[x + 1][y + 1][z + 1] = sqrt(2*delta[x + 4][y +
                      4][z + 4] + 1);
61                r17[x + 1][y + 1][z + 1] = cos(theta[x + 4][y + 4][z
                       + 4]);
62                float r24 = sin(theta[x + 4][y + 4][z + 4]);
63                r18[x + 1][y + 1][z + 1] = r24*sin(phi[x + 4][y +
                      4][z + 4]);
64                r19[x + 1][y + 1][z + 1] = r24*cos(phi[x + 4][y +
                      4][z + 4]);
65              }
66            }
67          }
68        }
69        STOP_TIMER(section0,timers)
70        /* End section0 */
71
72        for (int time = time_m, t0 = (time)%(3), t1 = (time + 2)%(3)
              , t2 = (time + 1)%(3); time <= time_M; time += 1, t0 = (
              time)%(3), t1 = (time + 2)%(3), t2 = (time + 1)%(3))
73        {
74          /* Begin section1 */
75          START_TIMER(section1)
76          #pragma omp parallel num_threads(nthreads)
77          {
78            const int tid = omp_get_thread_num();
79            float (*restrict r22)[y0_blk0_size + 1][z_size + 1]
                    __attribute__ ((aligned (64))) = (float (*)[
                    y0_blk0_size + 1][z_size + 1]) pr22[tid];
80            float (*restrict r23)[y0_blk0_size + 1][z_size + 1]
                    __attribute__ ((aligned (64))) = (float (*)[
                    y0_blk0_size + 1][z_size + 1]) pr23[tid];
81
82            #pragma omp for collapse(2) schedule(dynamic,1)
83            for (int x0_blk0 = x_m; x0_blk0 <= x_M; x0_blk0 +=
                    x0_blk0_size)
84            {
85              for (int y0_blk0 = y_m; y0_blk0 <= y_M; y0_blk0 +=
                      y0_blk0_size)
86              {
87                for (int x = x0_blk0 - 1, xs = 0; x <= MIN(x0_blk0 +
                       x0_blk0_size - 1, x_M); x += 1, xs += 1)
88                {
89                  for (int y = y0_blk0 - 1, ys = 0; y <= MIN(y0_blk0
                         + y0_blk0_size - 1, y_M); y += 1, ys += 1)
90                  {
91                    #pragma omp simd aligned(u,v:32)
```

```
92                  for (int z = z_m - 1; z <= z_M; z += 1)
93                  {
94                    r22[xs][ys][z + 1] = 5.0e-2F*(-(-u[t0][x + 4][
                          y + 4][z + 4] + u[t0][x + 4][y + 4][z +
                          5])*r17[x + 1][y + 1][z + 1] - (-u[t0][x +
                           4][y + 4][z + 4] + u[t0][x + 4][y + 5][z
                          + 4])*r18[x + 1][y + 1][z + 1] - (-u[t0][x
                           + 4][y + 4][z + 4] + u[t0][x + 5][y + 4][
                          z + 4])*r19[x + 1][y + 1][z + 1]);
95                    r23[xs][ys][z + 1] = 5.0e-2F*(-(-v[t0][x + 4][
                          y + 4][z + 4] + v[t0][x + 4][y + 4][z +
                          5])*r17[x + 1][y + 1][z + 1] - (-v[t0][x +
                           4][y + 4][z + 4] + v[t0][x + 4][y + 5][z
                          + 4])*r18[x + 1][y + 1][z + 1] - (-v[t0][x
                           + 4][y + 4][z + 4] + v[t0][x + 5][y + 4][
                          z + 4])*r19[x + 1][y + 1][z + 1]);
96                  }
97                }
98              }
99              for (int x = x0_blk0, xs = 0; x <= MIN(x0_blk0 +
                    x0_blk0_size - 1, x_M); x += 1, xs += 1)
100             {
101               for (int y = y0_blk0, ys = 0; y <= MIN(y0_blk0 +
                      y0_blk0_size - 1, y_M); y += 1, ys += 1)
102               {
103                 #pragma omp simd aligned(damp,epsilon,u,v,vp:32)
104                 for (int z = z_m; z <= z_M; z += 1)
105                 {
106                   float r25 = 5.0e-2F*(-r17[x + 1][y + 1][z]*r22
                          [xs + 1][ys + 1][z] + r17[x + 1][y + 1][z
                          + 1]*r22[xs + 1][ys + 1][z + 1] - r18[x +
                          1][y][z + 1]*r22[xs + 1][ys][z + 1] + r18[
                          x + 1][y + 1][z + 1]*r22[xs + 1][ys + 1][z
                           + 1] - r19[x][y + 1][z + 1]*r22[xs][ys +
                          1][z + 1] + r19[x + 1][y + 1][z + 1]*r22[
                          xs + 1][ys + 1][z + 1]) + 2.08333329e-4F
                          *(-u[t0][x + 2][y + 4][z + 4] - u[t0][x +
                          4][y + 2][z + 4] - u[t0][x + 4][y + 4][z +
                           2] - u[t0][x + 4][y + 4][z + 6] - u[t0][x
                           + 4][y + 6][z + 4] - u[t0][x + 6][y + 4][
                          z + 4]) + 3.33333326e-3F*(u[t0][x + 3][y +
                           4][z + 4] + u[t0][x + 4][y + 3][z + 4] +
                          u[t0][x + 4][y + 4][z + 3] + u[t0][x + 4][
                          y + 4][z + 5] + u[t0][x + 4][y + 5][z + 4]
                           + u[t0][x + 5][y + 4][z + 4]) -
                          1.87499996e-2F*u[t0][x + 4][y + 4][z + 4];
107                   float r28 = 1.0F/(vp[x + 4][y + 4][z + 4]*vp[x
                           + 4][y + 4][z + 4]);
108                   float r26 = 1.0F/(r20*r28 + r21*damp[x + 1][y
                          + 1][z + 1]);
109                   float r27 = r17[x + 1][y + 1][z]*r23[xs + 1][
                          ys + 1][z] - r17[x + 1][y + 1][z + 1]*r23[
                          xs + 1][ys + 1][z + 1] + r18[x + 1][y][z +
                           1]*r23[xs + 1][ys][z + 1] - r18[x + 1][y
                          + 1][z + 1]*r23[xs + 1][ys + 1][z + 1] +
                          r19[x][y + 1][z + 1]*r23[xs][ys + 1][z +
                          1] - r19[x + 1][y + 1][z + 1]*r23[xs + 1][
                          ys + 1][z + 1];
110                   u[t2][x + 4][y + 4][z + 4] = r26*(r21*damp[x +
```

```
                                1][y + 1][z + 1]*u[t0][x + 4][y + 4][z +
                               4] + r25*(2*epsilon[x + 4][y + 4][z + 4] +
                                1) + 5.0e-2F*r27*r16[x + 1][y + 1][z + 1]
                                + r28*(-r20*(-2.0F*u[t0][x + 4][y + 4][z
                               + 4]) - r20*u[t1][x + 4][y + 4][z + 4]));
111                        v[t2][x + 4][y + 4][z + 4] = r26*(r21*damp[x +
                                1][y + 1][z + 1]*v[t0][x + 4][y + 4][z +
                               4] + r25*r16[x + 1][y + 1][z + 1] + 5.0e-2
                               F*r27 + r28*(-r20*(-2.0F*v[t0][x + 4][y +
                               4][z + 4]) - r20*v[t1][x + 4][y + 4][z +
                               4]));
112                      }
113                    }
114                  }
115                }
116              }
117            }
118            STOP_TIMER(section1,timers)
119            /* End section1 */
```

A.2.2. Devito generated code for elastic wave propagators

The following code shows the optimized generated C-code for the Elastic wave propagation stencil-only part. Sources injection and receiver interpolation are ommited from the listing.

Listing A.2: An optimised version of Devito generated code for an Elastic forward wave propagators stencil.

```
1
2      int ForwardElastic(struct dataobj *restrict b_vec, struct
           dataobj *restrict damp_vec, struct dataobj *restrict
           lam_vec, struct dataobj *restrict mu_vec, struct dataobj *
           restrict rec1_vec, struct dataobj *restrict
           rec1_coords_vec, struct dataobj *restrict rec2_vec, struct
            dataobj *restrict rec2_coords_vec, struct dataobj *
           restrict src_vec, struct dataobj *restrict src_coords_vec,
            struct dataobj *restrict tau_xx_vec, struct dataobj *
           restrict tau_xy_vec, struct dataobj *restrict tau_xz_vec,
           struct dataobj *restrict tau_yy_vec, struct dataobj *
           restrict tau_yz_vec, struct dataobj *restrict tau_zz_vec,
           struct dataobj *restrict v_x_vec, struct dataobj *restrict
            v_y_vec, struct dataobj *restrict v_z_vec, const int x_M,
            const int x_m, const int y_M, const int y_m, const int
           z_M, const int z_m, const float dt, const float o_x, const
            float o_y, const float o_z, const int p_rec1_M, const int
            p_rec1_m, const int p_rec2_M, const int p_rec2_m, const
           int p_src_M, const int p_src_m, const int time_M, const
           int time_m, const int x0_blk0_size, const int x1_blk0_size
           , const int y0_blk0_size, const int y1_blk0_size, const
           int nthreads, const int nthreads_nonaffine, struct
           profiler * timers)
3      {
4        float (*restrict b)[b_vec->size[1]][b_vec->size[2]]
```

```
    __attribute__ ((aligned (64))) = (float (*)[b_vec->size
    [1]][b_vec->size[2]]) b_vec->data;
5   float (*restrict damp)[damp_vec->size[1]][damp_vec->size[2]]
     __attribute__ ((aligned (64))) = (float (*)[damp_vec->
    size[1]][damp_vec->size[2]]) damp_vec->data;
6   float (*restrict lam)[lam_vec->size[1]][lam_vec->size[2]]
    __attribute__ ((aligned (64))) = (float (*)[lam_vec->
    size[1]][lam_vec->size[2]]) lam_vec->data;
7   float (*restrict mu)[mu_vec->size[1]][mu_vec->size[2]]
    __attribute__ ((aligned (64))) = (float (*)[mu_vec->size
    [1]][mu_vec->size[2]]) mu_vec->data;
8   float (*restrict rec1)[rec1_vec->size[1]] __attribute__ ((
    aligned (64))) = (float (*)[rec1_vec->size[1]]) rec1_vec
    ->data;
9   float (*restrict rec1_coords)[rec1_coords_vec->size[1]]
    __attribute__ ((aligned (64))) = (float (*)[
    rec1_coords_vec->size[1]]) rec1_coords_vec->data;
10  float (*restrict rec2)[rec2_vec->size[1]] __attribute__ ((
    aligned (64))) = (float (*)[rec2_vec->size[1]]) rec2_vec
    ->data;
11  float (*restrict rec2_coords)[rec2_coords_vec->size[1]]
    __attribute__ ((aligned (64))) = (float (*)[
    rec2_coords_vec->size[1]]) rec2_coords_vec->data;
12  float (*restrict src)[src_vec->size[1]] __attribute__ ((
    aligned (64))) = (float (*)[src_vec->size[1]]) src_vec->
    data;
13  float (*restrict src_coords)[src_coords_vec->size[1]]
    __attribute__ ((aligned (64))) = (float (*)[
    src_coords_vec->size[1]]) src_coords_vec->data;
14  float (*restrict tau_xx)[tau_xx_vec->size[1]][tau_xx_vec->
    size[2]][tau_xx_vec->size[3]] __attribute__ ((aligned
    (64))) = (float (*)[tau_xx_vec->size[1]][tau_xx_vec->
    size[2]][tau_xx_vec->size[3]]) tau_xx_vec->data;
15  float (*restrict tau_xy)[tau_xy_vec->size[1]][tau_xy_vec->
    size[2]][tau_xy_vec->size[3]] __attribute__ ((aligned
    (64))) = (float (*)[tau_xy_vec->size[1]][tau_xy_vec->
    size[2]][tau_xy_vec->size[3]]) tau_xy_vec->data;
16  float (*restrict tau_xz)[tau_xz_vec->size[1]][tau_xz_vec->
    size[2]][tau_xz_vec->size[3]] __attribute__ ((aligned
    (64))) = (float (*)[tau_xz_vec->size[1]][tau_xz_vec->
    size[2]][tau_xz_vec->size[3]]) tau_xz_vec->data;
17  float (*restrict tau_yy)[tau_yy_vec->size[1]][tau_yy_vec->
    size[2]][tau_yy_vec->size[3]] __attribute__ ((aligned
    (64))) = (float (*)[tau_yy_vec->size[1]][tau_yy_vec->
    size[2]][tau_yy_vec->size[3]]) tau_yy_vec->data;
18  float (*restrict tau_yz)[tau_yz_vec->size[1]][tau_yz_vec->
    size[2]][tau_yz_vec->size[3]] __attribute__ ((aligned
    (64))) = (float (*)[tau_yz_vec->size[1]][tau_yz_vec->
    size[2]][tau_yz_vec->size[3]]) tau_yz_vec->data;
19  float (*restrict tau_zz)[tau_zz_vec->size[1]][tau_zz_vec->
    size[2]][tau_zz_vec->size[3]] __attribute__ ((aligned
    (64))) = (float (*)[tau_zz_vec->size[1]][tau_zz_vec->
    size[2]][tau_zz_vec->size[3]]) tau_zz_vec->data;
20  float (*restrict v_x)[v_x_vec->size[1]][v_x_vec->size[2]][
    v_x_vec->size[3]] __attribute__ ((aligned (64))) = (
    float (*)[v_x_vec->size[1]][v_x_vec->size[2]][v_x_vec->
    size[3]]) v_x_vec->data;
21  float (*restrict v_y)[v_y_vec->size[1]][v_y_vec->size[2]][
    v_y_vec->size[3]] __attribute__ ((aligned (64))) = (
```

```
            float (*)[v_y_vec->size[1]][v_y_vec->size[2]][v_y_vec->
            size[3]]) v_y_vec->data;
22      float (*restrict v_z)[v_z_vec->size[1]][v_z_vec->size[2]][
            v_z_vec->size[3]] __attribute__ ((aligned (64))) = (
            float (*)[v_z_vec->size[1]][v_z_vec->size[2]][v_z_vec->
            size[3]]) v_z_vec->data;
23
24      /* Flush denormal numbers to zero in hardware */
25      _MM_SET_DENORMALS_ZERO_MODE(_MM_DENORMALS_ZERO_ON);
26      _MM_SET_FLUSH_ZERO_MODE(_MM_FLUSH_ZERO_ON);
27
28      float r24 = 1.0F/dt;
29
30      for (int time = time_m, t0 = (time)%(2), t1 = (time + 1)%(2)
            ; time <= time_M; time += 1, t0 = (time)%(2), t1 = (time
             + 1)%(2))
31      {
32        /* Begin section0 */
33        START_TIMER(section0)
34        #pragma omp parallel num_threads(nthreads)
35        {
36          #pragma omp for collapse(2) schedule(dynamic,1)
37          for (int x0_blk0 = x_m; x0_blk0 <= x_M; x0_blk0 +=
                x0_blk0_size)
38          {
39            for (int y0_blk0 = y_m; y0_blk0 <= y_M; y0_blk0 +=
                  y0_blk0_size)
40            {
41              for (int x = x0_blk0; x <= MIN(x0_blk0 +
                    x0_blk0_size - 1, x_M); x += 1)
42              {
43                for (int y = y0_blk0; y <= MIN(y0_blk0 +
                      y0_blk0_size - 1, y_M); y += 1)
44                {
45                  #pragma omp simd aligned(b,damp,tau_xx,tau_xy,
                        tau_xz,tau_yy,tau_yz,tau_zz,v_x,v_y,v_z:32)
46                  for (int z = z_m; z <= z_M; z += 1)
47                  {
48                    v_x[t1][x + 4][y + 4][z + 4] = dt*(r24*v_x[t0
                          ][x + 4][y + 4][z + 4] + 5.0e-1F
                          *(4.16666673e-3F*(tau_xx[t0][x + 3][y +
                          4][z + 4] - tau_xx[t0][x + 6][y + 4][z +
                          4] + tau_xy[t0][x + 4][y + 2][z + 4] -
                          tau_xy[t0][x + 4][y + 5][z + 4] + tau_xz[
                          t0][x + 4][y + 4][z + 2] - tau_xz[t0][x +
                          4][y + 4][z + 5]) + 1.12500002e-1F*(-
                          tau_xx[t0][x + 4][y + 4][z + 4] + tau_xx[
                          t0][x + 5][y + 4][z + 4] - tau_xy[t0][x +
                          4][y + 3][z + 4] + tau_xy[t0][x + 4][y +
                          4][z + 4] - tau_xz[t0][x + 4][y + 4][z +
                          3] + tau_xz[t0][x + 4][y + 4][z + 4]))*(b[
                          x + 4][y + 4][z + 4] + b[x + 5][y + 4][z +
                           4]))*damp[x + 1][y + 1][z + 1];
49                    v_y[t1][x + 4][y + 4][z + 4] = dt*(r24*v_y[t0
                          ][x + 4][y + 4][z + 4] + 5.0e-1F
                          *(4.16666673e-3F*(tau_xy[t0][x + 2][y +
                          4][z + 4] - tau_xy[t0][x + 5][y + 4][z +
                          4] + tau_yy[t0][x + 4][y + 3][z + 4] -
                          tau_yy[t0][x + 4][y + 6][z + 4] + tau_yz[
```

```
                        t0][x + 4][y + 4][z + 2] - tau_yz[t0][x +
                        4][y + 4][z + 5]) + 1.12500002e-1F*(-
                        tau_xy[t0][x + 3][y + 4][z + 4] + tau_xy[
                        t0][x + 4][y + 4][z + 4] - tau_yy[t0][x +
                        4][y + 4][z + 4] + tau_yy[t0][x + 4][y +
                        5][z + 4] - tau_yz[t0][x + 4][y + 4][z +
                        3] + tau_yz[t0][x + 4][y + 4][z + 4]))*(b[
                        x + 4][y + 4][z + 4] + b[x + 4][y + 5][z +
                         4]))*damp[x + 1][y + 1][z + 1];
50                    v_z[t1][x + 4][y + 4][z + 4] = dt*(r24*v_z[t0
                        ][x + 4][y + 4][z + 4] + 5.0e-1F
                        *(4.16666673e-3F*(tau_xz[t0][x + 2][y +
                        4][z + 4] - tau_xz[t0][x + 5][y + 4][z +
                        4] + tau_yz[t0][x + 4][y + 2][z + 4] -
                        tau_yz[t0][x + 4][y + 5][z + 4] + tau_zz[
                        t0][x + 4][y + 4][z + 3] - tau_zz[t0][x +
                        4][y + 4][z + 6]) + 1.12500002e-1F*(-
                        tau_xz[t0][x + 3][y + 4][z + 4] + tau_xz[
                        t0][x + 4][y + 4][z + 4] - tau_yz[t0][x +
                        4][y + 3][z + 4] + tau_yz[t0][x + 4][y +
                        4][z + 4] - tau_zz[t0][x + 4][y + 4][z +
                        4] + tau_zz[t0][x + 4][y + 4][z + 5]))*(b[
                        x + 4][y + 4][z + 4] + b[x + 4][y + 4][z +
                         5]))*damp[x + 1][y + 1][z + 1];
51                  }
52                }
53              }
54            }
55          }
56        }
57        #pragma omp parallel num_threads(nthreads)
58        {
59          #pragma omp for collapse(2) schedule(dynamic,1)
60          for (int x1_blk0 = x_m; x1_blk0 <= x_M; x1_blk0 +=
                x1_blk0_size)
61          {
62            for (int y1_blk0 = y_m; y1_blk0 <= y_M; y1_blk0 +=
                  y1_blk0_size)
63            {
64              for (int x = x1_blk0; x <= MIN(x1_blk0 +
                    x1_blk0_size - 1, x_M); x += 1)
65              {
66                for (int y = y1_blk0; y <= MIN(y1_blk0 +
                      y1_blk0_size - 1, y_M); y += 1)
67                {
68                  #pragma omp simd aligned(damp,lam,mu,tau_xx,
                        tau_xy,tau_xz,tau_yy,tau_yz,tau_zz,v_x,v_y,
                        v_z:32)
69                  for (int z = z_m; z <= z_M; z += 1)
70                  {
71                    float r25 = (4.16666673e-3F*(v_x[t1][x + 2][y
                        + 4][z + 4] - v_x[t1][x + 5][y + 4][z + 4]
                         + v_y[t1][x + 4][y + 2][z + 4] - v_y[t1][
                        x + 4][y + 5][z + 4] + v_z[t1][x + 4][y +
                        4][z + 2] - v_z[t1][x + 4][y + 4][z + 5])
                        + 1.12500002e-1F*(-v_x[t1][x + 3][y + 4][z
                         + 4] + v_x[t1][x + 4][y + 4][z + 4] - v_y
                        [t1][x + 4][y + 3][z + 4] + v_y[t1][x +
                        4][y + 4][z + 4] - v_z[t1][x + 4][y + 4][z
```

```
     + 3] + v_z[t1][x + 4][y + 4][z + 4]))*lam
    [x + 4][y + 4][z + 4];
72  tau_xx[t1][x + 4][y + 4][z + 4] = dt*(r24*
    tau_xx[t0][x + 4][y + 4][z + 4] + r25 +
    2*(4.16666673e-3F*(v_x[t1][x + 2][y + 4][z
     + 4] - v_x[t1][x + 5][y + 4][z + 4]) +
    1.12500002e-1F*(-v_x[t1][x + 3][y + 4][z +
     4] + v_x[t1][x + 4][y + 4][z + 4]))*mu[x
    + 4][y + 4][z + 4])*damp[x + 1][y + 1][z +
     1];
73  tau_xy[t1][x + 4][y + 4][z + 4] = dt*(r24*
    tau_xy[t0][x + 4][y + 4][z + 4] + 2.5e-1F
    *(4.16666673e-3F*(v_x[t1][x + 4][y + 3][z
    + 4] - v_x[t1][x + 4][y + 6][z + 4] + v_y[
    t1][x + 3][y + 4][z + 4] - v_y[t1][x + 6][
    y + 4][z + 4]) + 1.12500002e-1F*(-v_x[t1][
    x + 4][y + 4][z + 4] + v_x[t1][x + 4][y +
    5][z + 4] - v_y[t1][x + 4][y + 4][z + 4] +
     v_y[t1][x + 5][y + 4][z + 4]))*(mu[x +
    4][y + 4][z + 4] + mu[x + 4][y + 5][z + 4]
     + mu[x + 5][y + 4][z + 4] + mu[x + 5][y +
     5][z + 4]))*damp[x + 1][y + 1][z + 1];
74  tau_xz[t1][x + 4][y + 4][z + 4] = dt*(r24*
    tau_xz[t0][x + 4][y + 4][z + 4] + 2.5e-1F
    *(4.16666673e-3F*(v_x[t1][x + 4][y + 4][z
    + 3] - v_x[t1][x + 4][y + 4][z + 6] + v_z[
    t1][x + 3][y + 4][z + 4] - v_z[t1][x + 6][
    y + 4][z + 4]) + 1.12500002e-1F*(-v_x[t1][
    x + 4][y + 4][z + 4] + v_x[t1][x + 4][y +
    4][z + 5] - v_z[t1][x + 4][y + 4][z + 4] +
     v_z[t1][x + 5][y + 4][z + 4]))*(mu[x +
    4][y + 4][z + 4] + mu[x + 4][y + 4][z + 5]
     + mu[x + 5][y + 4][z + 4] + mu[x + 5][y +
     4][z + 5]))*damp[x + 1][y + 1][z + 1];
75  tau_yy[t1][x + 4][y + 4][z + 4] = dt*(r24*
    tau_yy[t0][x + 4][y + 4][z + 4] + r25 +
    2*(4.16666673e-3F*(v_y[t1][x + 4][y + 2][z
     + 4] - v_y[t1][x + 4][y + 5][z + 4]) +
    1.12500002e-1F*(-v_y[t1][x + 4][y + 3][z +
     4] + v_y[t1][x + 4][y + 4][z + 4]))*mu[x
    + 4][y + 4][z + 4])*damp[x + 1][y + 1][z +
     1];
76  tau_yz[t1][x + 4][y + 4][z + 4] = dt*(r24*
    tau_yz[t0][x + 4][y + 4][z + 4] + 2.5e-1F
    *(4.16666673e-3F*(v_y[t1][x + 4][y + 4][z
    + 3] - v_y[t1][x + 4][y + 4][z + 6] + v_z[
    t1][x + 4][y + 3][z + 4] - v_z[t1][x + 4][
    y + 6][z + 4]) + 1.12500002e-1F*(-v_y[t1][
    x + 4][y + 4][z + 4] + v_y[t1][x + 4][y +
    4][z + 5] - v_z[t1][x + 4][y + 4][z + 4] +
     v_z[t1][x + 4][y + 5][z + 4]))*(mu[x +
    4][y + 4][z + 4] + mu[x + 4][y + 4][z + 5]
     + mu[x + 4][y + 5][z + 4] + mu[x + 4][y +
     5][z + 5]))*damp[x + 1][y + 1][z + 1];
77  tau_zz[t1][x + 4][y + 4][z + 4] = dt*(r24*
    tau_zz[t0][x + 4][y + 4][z + 4] + r25 +
    2*(4.16666673e-3F*(v_z[t1][x + 4][y + 4][z
     + 2] - v_z[t1][x + 4][y + 4][z + 5]) +
    1.12500002e-1F*(-v_z[t1][x + 4][y + 4][z +
```

```
                                3] + v_z[t1][x + 4][y + 4][z + 4]))*mu[x
                               + 4][y + 4][z + 4])*damp[x + 1][y + 1][z +
                                1];
78                        }
79                      }
80                    }
81                  }
82                }
83              }
84              STOP_TIMER(section0,timers)
85              /* End section0 */
```