Imperial College London

MEng Individual Project

Department of Computing

Imperial College of Science, Technology and Medicine

Temporal Tiling for Distributed Parallel Solution of Partial Differential Equations

Author:
Nicholas Duer

Supervisors:
Paul Kelly, George Bisbas

June 20, 2023

Abstract

The computation of the finite-difference approximation to partial differential equations (PDEs) has numerous applications, ranging from medical imaging to the study of fluid dynamics. To optimise the performance of finite-difference algorithms, loop tiling techniques have been widely employed to exploit locality of reference and improve cache performance. *Wavefront tiling* is a popular tiling scheme that extends loop tiles with a time dimension, to ensure finite-difference code benefits from temporal locality as well as spatial locality. The downside of wavefront tiling is its inability to be applied to the distributed memory computation of finite-difference algorithms.

In this study, we propose the use of the *overlapped tiling* scheme to harness the benefits of both distributed computation and temporal locality from wavefront tiling. We implement the overlapped tiling scheme in finite-difference code for the Laplace and wave PDEs. Then, we evaluate the performance of our overlapped tiling implementation on Intel's Xeon v2 and Xeon v3 CPU platforms. We report speed-ups of 55% and 28% in execution times for Laplace stencils of space order 2 and 4, respectively, when compared to a non-overlapped tiling implementation.

Further contributions are made to improve the understanding of overlapped tiling benefits. We provide an analysis of the impact of overlapped tile parameters on execution times and tune our tile sizes to find the optimal dimensions. We conclude our study by presenting the derivation of mathematical models for the impact of overlapped tiling on redundant computation and MPI communication times.

Acknowledgements

I would like to thank Professor Paul Kelly for being such a great supervisor and investing so much time into the project.

I am incredibly grateful to George Bisbas for his dedication and ever-present guidance throughout my project. He ensured the countless hours spent in the William Penny Laboratory were always enjoyable and I wish him the best of luck for the future.

I am indebted to my parents for their never-ending optimism. Their support was invaluable in ensuring I always saw the positives in any situation. Thanks for everything!

I am very grateful to Andrew for his 90+4' last-minute comprehensive review of my thesis.

I extend my utmost gratitude to Marco Silva and Fulham Football Club for a historic season.

Contents

1 Introduction **4**
- 1.1 Problem Statement . . . 4
- 1.2 Motivation . . . 4
- 1.3 Contributions . . . 6
- 1.4 Ethical Considerations . . . 7

2 Preliminaries **8**
- 2.1 Partial Differential Equations . . . 8
- 2.2 Finite Difference Stencils . . . 9
  - 2.2.1 Finite Difference Approximation . . . 9
  - 2.2.2 Stencil Computations . . . 10
- 2.3 Automating Stencil Code . . . 12
  - 2.3.1 Automating with the Devito Compiler . . . 13
- 2.4 Parallel Computation with Devito . . . 14
  - 2.4.1 OpenMP Threads with Devito . . . 14
  - 2.4.2 MPI Processes with Devito . . . 15
  - 2.4.3 Hybrid MPI-OpenMP Parallelism with Devito . . . 16
- 2.5 Loop Tiling . . . 17
  - 2.5.1 Loop Blocking . . . 17
  - 2.5.2 Temporal Blocking . . . 19
- 2.6 Concluding Remarks . . . 20

3 Related Work **21**
- 3.1 Solving Partial Differential Equations . . . 21
  - 3.1.1 Alternative Methods . . . 21
  - 3.1.2 Finite Element Method Frameworks . . . 22
  - 3.1.3 Finite Difference Method Frameworks . . . 23
  - 3.1.4 Polyhedral Compilers . . . 25
- 3.2 Loop Tiling . . . 26
  - 3.2.1 Overlapped Tiling . . . 28
- 3.3 Concluding Remarks . . . 29

4 Implementation **30**
- 4.1 Wavefront Tiling . . . 31
- 4.2 Overlapped Tiling . . . 33
  - 4.2.1 Increasing the Halo Region . . . 33

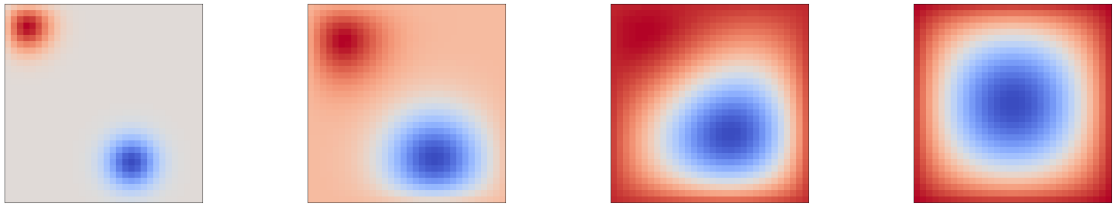
4.2.2 Modifying the Loop Bounds 34
4.3 Overlapped Tiling with Wavefront Tiling 37
4.4 Concluding Remarks 39

5 Evaluation **40**
5.1 Experimental Setup 40
5.1.1 CPU Platforms 40
5.1.2 Stencils 41
5.1.3 Metrics 43
5.2 Performance Evaluation 43
5.2.1 Preliminary Results 43
5.2.2 Overlapped Tiling Performance 45
5.3 Tuning Tile Parameters 47
5.3.1 Tuning Tile Height 48
5.3.2 Modelling Redundant Calculations 50
5.3.3 Tuning Wavefront Tile Widths 52
5.4 MPI Communication Times 53
5.5 Scalability 55
5.6 Concluding Remarks 56

6 Conclusions **57**
6.1 Future Work 58

Chapter 1

Introduction



(a) $t = 64$ (b) $t = 256$ (c) $t = 1028$ (d) $t = 4096$

Figure 1.1: Simulation of heat transfer with a hot and cold pole using the heat equation PDE. State displayed at different unit time intervals, t. Generated using Devito (see Section 2.3.1).

1.1 Problem Statement

Temporal tiling is an optimisation that improves cache reuse in the approximation of solutions to partial differential equations. Wavefront tiling is a commonly used temporal tiling scheme that suffers from its inability to be applied to distributed memory execution. In this study, we aim to harness the benefits of both temporal tiling and distributed memory parallelism.

1.2 Motivation

Partial differential equations (PDEs) are ubiquitous in a wide range of physical problems and have endless applications. These range from the imaging and diagnosis of brain tumours [18, 47] to the analysis of fluid flow [10, 34, 36]. As a result, PDEs are a subject of intensive study in modern mathematics [42, 49].

Finding an exact solution to a PDE is very challenging and often impossible. Instead, solutions are approximated using numerical methods. This concept has given rise

to an extremely wide range of contemporary research and approaches [14, 17, 38]. One such approach is the *finite-difference* method which involves discretising the problem by creating a large grid. Each point on this grid is then visited and a mathematical operator is applied. To obtain an acceptable degree of accuracy, a very large grid is required and the points of the grid often number in the billions [12]. Thus, finite-difference algorithms are computationally intensive and must be optimised.

A powerful optimisation applied to finite-difference code is *loop tiling* [32, 52, 53]. Instead of iterating over the grid in standard fashion, the grid is partitioned into smaller tiles each of which are iterated over in turn. Loop tiling greatly improves cache reuse by taking advantage of locality of reference.

PDEs are used to model problems with both spatial and temporal components. It is common practice to use loop tiles with a temporal dimension as well as a spatial dimension. With a temporal dimension, the finite-difference code will benefit from temporal locality. *Wavefront tiling* [13, 54] is a popular method of temporal tiling but has a major limitation: it cannot be applied to the distributed computation of finite-difference code. By first applying a scheme known as *overlapped tiling* [23, 57], code intended for distributed memory computation can then be tiled with wavefront tiles (see Figure 1.2). The benefits of temporal locality and distributed computation then work in tandem. Combining these benefits forms the focus of this study.

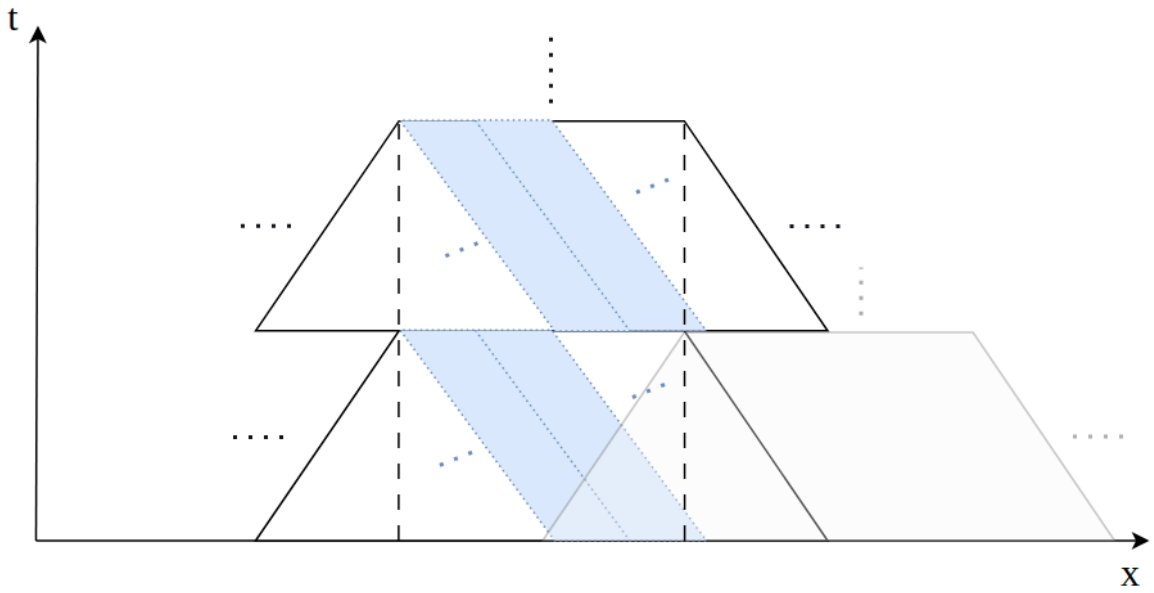


Figure 1.2: Overlapped tiling with wavefront tiling in one spatial dimension, x. In the y-axis, we see the temporal dimension, t. The trapezium shapes are our overlapped tiles. They are tiled with blue wavefront tiles. Each 'stack' (extending in the t-dimension) of overlapped tiles is assigned to one process. Here, the grey (right) overlapped tile is owned by a different process than the process owning both black (left) overlapped tiles.

Code implementing the finite-difference method can become very complex for PDEs modelling real-world applications. This makes it difficult to write by hand and even harder to optimise. For this reason, many different code-generation frameworks have been produced [22, 30, 43]. These frameworks take as input a high-level description of the PDE and return optimised code. In this paper, we work with the Devito framework [31].

1.3 Contributions

With this paper, we aim to improve the understanding of overlapped tiling and outline its benefits when applied to distributed computation for solving PDEs. To this end, we have made the following contributions:

- **Wavefront Tiling Implementations**

  We implement the wavefront tiling scheme on finite-difference code generated by Devito. Implementations are provided for both the Laplace and wave PDEs.

- **Overlapped Tiling Implementations**

  We implement the overlapped tiling scheme on finite-difference code generated by Devito. The code produced is written for distributed computation. Implementations are provided for both the Laplace and wave PDEs.

- **Temporal Tiling Performance Evaluation**

  We evaluate the performance of overlapped tiling across two different architectures. Wavefront tiling is also evaluated to provide insights into the efficacy of temporal locality within overlapped tiling.

- **Temporal Tiling Performance Modelling**

  We provide mathematical models which quantify the positive impacts of overlapped tiling on communications times and the negative impacts on redundant computation.

- **Temporal Tiling Parameter Tuning**

  We investigate the impact of overlapped tiling parameters on execution times. Tile parameters are varied to find the optimum performance.

In Chapter 2 we provide the reader with background on partial differential equations, finite-difference computation and loop tiling. Chapter 3 outlines landmark research in the field. In Chapter 4 we discuss our tiling implementations and highlight key design decisions. Then, in Chapter 5 we evaluate our tiling implementations and model various aspects of overlapped tiling. Positive results are reported, including a 55% speed-up in execution time on the Xeon v3 CPU when using an overlapped tiling implementation compared to standard Devito code.

1.4 Ethical Considerations

A key ethical concern stems from environmental considerations. One of Devito's main use cases is seismic imaging, where pulses are produced which reflect off various subterranean materials (see Figure 1.3). By detecting the properties of the reflected pulses, models of the Earth's rock may be produced to uncover hydrocarbon-based energy sources. A PDE is used to model this physical problem. Indeed, in the Devito repository, a folder detailing example seismic use cases may be found [1]. Therefore, the Devito project potentially reduces the cost of oil and gas exploration. There are environmental concerns associated with fossil fuel energy sources with many economies switching their efforts towards low-carbon energy sources such as renewables. Thus, there is an ethical concern with the use of this research in the energy sector.

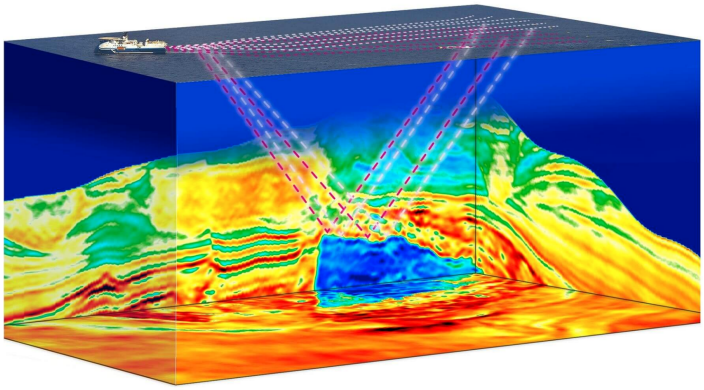


Figure 1.3: Seismic imaging used in hydrocarbon exploration. Image from CGG webpage [2]

While these considerations are of concern, they do not directly relate to the impact and findings of this study. Throughout this study, we discuss mathematical and theoretical concepts only. We do not expand on any specific physical applications. The research presented in this project is abstract and we avoid any major ethical concerns.

There are no legal or licensing concerns with this project. Devito is an open-source available on GitHub [3]. All external libraries used during this project do not involve copyright implications. Ensuring our project is open source is a key ethical consideration. It guarantees that our results are reproducible and that our research is transparent. This is vital in maintaining public trust in the scientific community.

The project does not involve the collection of personal data. There are no human participants. All evaluation is carried out using quantitative software bench-marking. As a result, there are no ethical concerns associated with our data processing.

Chapter 2

Preliminaries

In this Chapter, we provide the reader with the required background knowledge on overlapped tiling. We begin in Section 2.1 with a definition of a partial differential equation (PDE). In Section 2.2 we outline the finite-difference method for solving a PDE. Then, in Sections 2.3 and 2.4 we discuss the Devito code-generation framework. We conclude the Chapter with Section 2.5 where we introduce the loop tiling optimisation.

2.1 Partial Differential Equations

A partial differential equation is an equation involving:

- Two or more independent variables $x_1, x_2, ..., x_n$.
- An *unknown* function $u = u(x_1, x_2, ..., x_n)$ of these independent variables.
- Partial derivatives of the function u with respect to the independent x_i. Note these are often denoted using subscript notation, e.g. $u_x = \frac{\partial u}{\partial x}, u_{xy} = \frac{\partial^2 u}{\partial y \partial x}$

An example of a PDE is shown below:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = -y$$

We wish to find the unknown function $u(x, y)$ explicitly in terms of our independent variables. u should satisfy the PDE given. Generally, there are many possible solutions to a PDE.

PDEs are solved subject to certain constraints on u. These are referred to as **boundary conditions**. Possible boundary conditions for the example above are:

$$y = 0, u(x, 0) = 1; \quad x = 0, u(0, y) = 0$$

Finding solutions to PDEs has many important physical applications. These include seismic imaging [50], medical imaging [18, 47] and modelling of fluid flow [10, 34, 36]. Finding an explicit solution to a PDE is generally difficult and often theoretically impossible. Numerical methods to approximate solutions are often used (see Section 2.2.2).

2.2 Finite Difference Stencils

As mentioned at the end of Section 2.1, we are often forced to approximate our solutions (the function u) to PDEs. For this, we can use numerical methods. We focus on a commonly used method that relies on the finite difference approximation (see 2.2.1).

The finite difference approximation is used to approximate the partial derivatives in terms of the function u. We find a relationship between partial derivatives and values of the function u itself. These approximations to the partial derivatives are substituted for the partial derivatives which appear in the PDE. This creates an approximate recurrence relation for our function u.

For example, we may relate the value of our function u at time $t = n$ to the value of u at time $t = n - 1$. Starting with initial boundary conditions, (see Section 2.1) we use the recurrence relation to approximate values of the unknown function u for a large grid.

2.2.1 Finite Difference Approximation

We aim to approximate the partial derivatives of a multi-variable function for different values of our variables x_i. It is enough to approximate the derivative, u_x, of a single-valued function, $u(x)$. Then, the partial derivatives of a multi-variable function can be approximated in the same way by fixing all but one variable. The following material is adapted from LeVeque [28]. In the following calculations, $u'(x)$ denotes our derivative u_x and $\bar{x}$ denotes our choice of x at which we approximate $u(x)$. $D_* u(\bar{x})$ will be used to denote our different approximations.

One possible approximation to $u'(\bar{x})$, using values of $u(\bar{x})$ is:

$$u'(\bar{x}) \approx D_+ u(\bar{x}) = \frac{u(\bar{x}+h) - u(\bar{x})}{h} \tag{2.1}$$

where h is taken to be small. This is known as the **forward difference** approximation. This approximation is inspired by the formal definition of the derivative:

$$u'(\bar{x}) = \lim_{h \to 0} \frac{u(\bar{x}+h) - u(\bar{x})}{h} \tag{2.2}$$

Similarly, we also have the **backward difference** approximation:

$$u'(\bar{x}) \approx D_- u(\bar{x}) = \frac{u(\bar{x}) - u(\bar{x}-h)}{h} \tag{2.3}$$

where h is taken to be small. These are both examples of **first-order approximations**, where our error is proportional to h. We can take the average of these approximations in Equations 2.1 and 2.3 to find a **second-order approximation**:

$$u'(\bar{x}) \approx D_0 u(\bar{x}) = \frac{u(\bar{x}+h) - u(\bar{x}-h)}{2h} \tag{2.4}$$

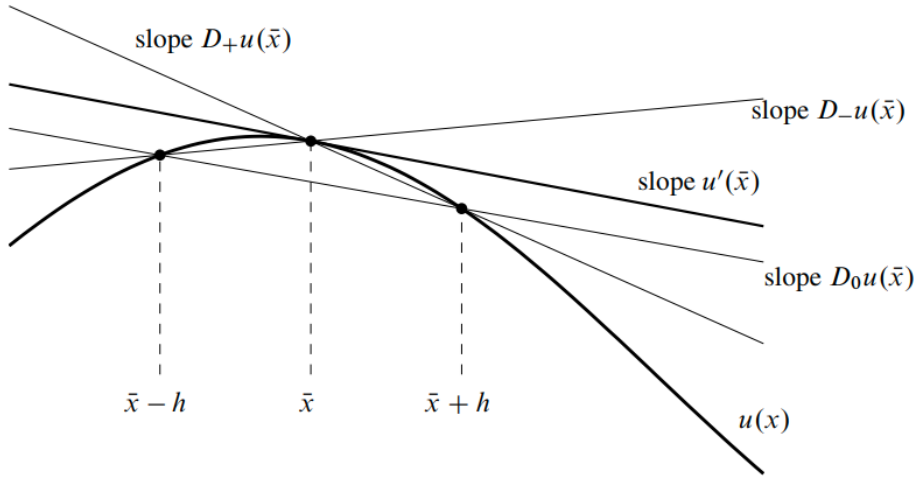


Figure 2.1: Approximations to $u'(\bar{x})$. Image from LeVeque [28]

This is known as a **centred approximation**. Its error is proportional to h^2. In Equations 2.1, 2.3, 2.4, we have used two values of $u(\bar{x})$ in our approximation. By using more values of $u(\bar{x})$, we can find higher-order approximations to the first derivative.

Approximating Higher Order Derivatives

Above we have presented approximations to the first derivative, $u'(x)$. We can obtain approximations of higher derivatives, $u''(x), u'''(x), ...$ in a similar way. Below we find an approximation for $u''(x)$. The derivation uses **Taylor's expansion**:

$$u(x+h) = u(x) + hu'(x) + h^2\frac{u''(x)}{2!} + h^3\frac{u'''(x)}{3!} + O(h^4) \tag{2.5}$$

Replacing h with $-h$ in 2.5 we get:

$$u(x-h) = u(x) - hu'(x) + h^2\frac{u''(x)}{2!} - h^3\frac{u'''(x)}{3!} + O(h^4) \tag{2.6}$$

Adding 2.5 and 2.6 and rearranging terms, we get:

$$u''(x) \approx \frac{u(x+h) - 2u(x) + u(x-h)}{h^2} \tag{2.7}$$

which is a second-order centred difference approximation of the second derivative $u''(x)$ (accurate up to $O(h^4)$ terms).

2.2.2 Stencil Computations

Stencil computations are functions that update a given point in a grid. The functions use the neighbours of our point as inputs. Below is an example of a stencil computation:

$$u(i,j) = \frac{1}{4}\left(u(i-1,j) + u(i+1,j) + u(i,j-1) + u(i,j+1)\right) \tag{2.8}$$

In the above Equation 2.8, (i, j) is our point being updated.

We express this in C code as:

```
1 u[i][j] = 0.25 * (u[i - 1][j] + u[i + 1][j] + u[i][j - 1] + u[i][j + 1])
```

The above stencil computation updates a point with the average of its direct neighbours. The pattern used in a stencil is called the **kernel**. This kernel of this particular stencil can be seen in Figure 2.2. Note that while this stencil is normalised (using the $\frac{1}{4}$ factor), this is not the case in general.

0	$\frac{1}{4}$	0
$\frac{1}{4}$	0	$\frac{1}{4}$
0	$\frac{1}{4}$	0

Figure 2.2: Example stencil kernel

Using the material in Section 2.2.1, we can create finite-difference stencils to approximate solutions to PDEs.

The Heat Equation

As an example, we will derive a stencil for the one-dimensional **heat equation**. The heat equation is used to model the distribution of heat in a body and is a widely studied mathematical equation [15]. The following derivation is based on lecture notes by Peirce [41]. The spatial component of u is 1-dimensional (just x, which denotes temperature):

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

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

We divide our spatial interval $[0, 1]$ into $N + 1$ sample points, $x_n = n\Delta x$, where $n \in \{0, ..., N\}$. We have to select an upper bound T on our time interval over which we wish to approximate values of u. Then the time interval $[0, T]$ is divided into $M + 1$ time levels $t_k = k\Delta t$, where $k \in \{0, ..., M\}$. Δx and Δt are the equidistant gaps between our spatial and temporal sample points respectively and are given by:

$$\Delta x = \frac{1}{N} \tag{2.11}$$

$$\Delta t = \frac{T}{M} \tag{2.12}$$

At each of these time-space sample points in our grid, we introduce approximations to u: $u_n^k \approx u(x_n, t_k)$. These grid points will approximate the values of the unknown u.

Referring to Section 2.2.1, we use finite differences to approximate both u_t and u_{xx}. The first-order **Forward Difference in Time** (Equation 2.1) is:

$$\frac{\partial u}{\partial t}(x,t) \approx \frac{u(x,t+\Delta t) - u(x,t)}{\Delta t} \tag{2.13}$$

The second-order **Central Difference in Space** (Equation 2.7) is:

$$\frac{\partial^2 u}{\partial x^2}(x,t) \approx \frac{u(x+\Delta x,t) - 2u(x,t) + u(x-\Delta x,t)}{\Delta x^2} \tag{2.14}$$

By substituting 2.13 and 2.14 into 2.9, we get:

$$\frac{u(x,t+\Delta t) - u(x,t)}{\Delta t} \approx \alpha^2 (\frac{u(x+\Delta x,t) - 2u(x,t) + u(x-\Delta x,t)}{\Delta x^2})$$

then rearranging gives:

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

This provides us with a recurrence relation for our grid points:

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

which may be expressed in pseudo-code as:

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

2.3 Automating Stencil Code

In Section 2.2.2 we demonstrate how to derive stencil code which can be used to approximate solutions to PDEs. Unfortunately, producing this code by hand is very difficult due to the following reasons:

- The entire process requires a wide range of expertise. Physicists, mathematicians, etc. focus on modelling the physical problem to produce the relevant PDEs. Then, software engineers produce and optimise stencil code. This requires a large number of specialists.
- In practice the stencil kernels are very large and complex. This is due to the following considerations:

  - We use derivatives of higher order
  - We use finite difference approximations which have higher orders of accuracy
  - Spatial coordinates are generally 3-dimensional (as opposed to 1-dimensional in the heat equation introduced in Section 2.2.2)

- Optimising stencil code often requires careful consideration of loop bounds (see Section 2.5.1). Reasoning correctly about this can be very difficult.

2.3.1 Automating with the Devito Compiler

The Devito project [31] automates the solutions of PDEs by generating the required stencil code. The project defines a language that allows users to specify PDEs in Python. Then, the Devito compiler lowers this mathematical expression through various intermediate representations. The final output is C/C++ code implementing the finite difference stencil described above.

As an example, we will demonstrate how Devito can generate stencil code for the heat equation. Below we see the Python specification of the heat equation (as defined in Section 2.2.2, taking $\alpha = 1$):

```
1  from devito import Grid, TimeFunction, Eq, Operator, solve
2  grid = Grid(shape=(10, 10))
3  u = TimeFunction(name='f', grid=grid)
4
5  # Some variable declarations
6  nx, nt = 10, 10
7  dx = 2. / (nx - 1)
8  sigma = .2
9  dt = sigma * dx
10
11 grid = Grid(shape=(nx, ), extent=(1.,))
12 u = TimeFunction(name='u', grid=grid, space_order=2)
13
14 eq = Eq(u.dt, u.dx2, grid=grid)
15 stencil = solve(eq, u.forward)
16
17 # Boundary conditions
18 t = grid.stepping_dim
19 bc_left = Eq(u[t + 1, 0], 0.)
20 bc_right = Eq(u[t + 1, nx-1], 0.)
21
22 op = Operator([Eq(u.forward, stencil), bc_left, bc_right])
23 op.apply(time_M=nt, dt=dt)
```

On line 14 we can see the definition of the PDE associated with the heat equation. *u.dt* and *u.dx2* denote u_t and u_{xx} respectively. This gives the desired PDE: $u_t = u_{xx}$.

The Devito compiler then generates C code which implements the stencil code. Below is part of the code generated by Devito (simplified and adapted slightly):

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

The C code generated here is similar to the pseudo-code given in Section 2.2.2. On lines 8 and 9 we see our stencil update.

2.4 Parallel Computation with Devito

The Devito compiler provides support for parallel execution of stencil code. Devito can generate stencil code implementing any of the following parallel models:

- **Threads**. Execution units share the same memory space. Devito implements this model using the OpenMP library [4]. Also known as **shared-memory** parallelism.
- **Processes**. Execution units are independent and run in separate memory spaces. Processes must communicate by sending messages. Devito implements this model using the Message Passing Interface (MPI) library [5]. Also known as **distributed-memory** parallelism.
- **Hybrid**. The program uses a combination of processes and threads. Multiple processes are created, each with its own memory space. Within these separate processes, we run multiple threads.

When using a high-performance computer with many nodes, we prefer to use a hybrid approach. Each node has separate memory, so it is impossible to use threading across multiple nodes. However, we still want the benefit from multi-threading while also making use of distributed computation. We create processes that exist within one node. Threading is then applied within the processes.

2.4.1 OpenMP Threads with Devito

When executing stencil computations, we can often process many points on a grid in parallel. Dependencies tend to only span across timesteps. Hence, we can calculate points within the same timestep concurrently. The OpenMP library allows us to

specify regions of code to which we would like to apply threading.

Below we see part of the code generated by Devito for the example used in Section 2.3.1. Lines 3 and 5 make use of the OpenMP library. They ensure all commands in the for-loop are executed in parallel. As this is applied to the inner for-loop, we only apply threading to points within the same timestep.

```
1  for (int t = time_m; t <= time_M; t += 1)
2  {
3    #pragma omp parallel num_threads(nthreads)
4    {
5      #pragma omp for collapse(1) schedule(static,1)
6      for (int x = x_m; x <= x_M; x += 1)
7      {
8        u[t + 1][x + 2] = dt*(r0*u[t][x + 2] + r1*u[t][x + 1] +
9          r1*(-2.0F*u[t][x + 2]) + r1*u[t][x + 3]);
10     }
11   }
12 }
```

2.4.2 MPI Processes with Devito

When applying distributed parallelism to stencil computation, we divide our grid into smaller grids. Each process is assigned one smaller sub-grid and then executes stencil computation in parallel. Note MPI assigns a number to each separate process, known as its **rank**. For this reason, we often refer to each parallel process as a rank. We use MPI to pass messages between the ranks.

Communication between ranks is required due to the dependencies in kernel computation. When calculating at the boundary of a grid, one rank may need a point that has been allocated to a different rank. For example, we can consider the following stencil:

```
1  u[t+1][x] = u[t][x-1] + u[t][x] + u[t][x+1]
```

This is a 2-dimensional stencil, with one spatial dimension, x. Suppose our grid has x width of 10 and we wish to use 2 ranks. The grid will be divided up and the first rank will own points from $x \in \{0, ..., 4\}$ and the second from $\{5, ..., 9\}$. Then, for the first rank to compute points at $x = 4$, it will need the point at $x = 5$ from the previous timestep. Likewise, the second rank will need the point at $x = 4$. We must pass these values between the ranks at every single timestep.

This region in each rank is known as the **halo region**. In the example above, Devito will allocate a halo region of size 1 on either side of the owned points. Each rank actually then owns a grid of width 7, but the stencil is still only applied to 5 points. The rank does not iterate over the halo region but will only use it for dependencies.

Before every timestep, a rank will receive MPI communication and update its halo regions with the new values.

Below is part of the code generated by Devito for the example used in Section 2.3, with MPI calls. On line 3, we see the call to a halo update function, which is defined using the MPI library.

```
1 for (int t = time_m; t <= time_M; t += 1)
2   {
3   haloupdate(t);
4   for (int x = x_m; x <= x_M; x += 1)
5   {
6     u[t + 1][x + 2] = dt*(r0*u[t][x + 2] + r1*u[t][x + 1] +
7       r1*(-2.0F*u[t][x + 2]) + r1*u[t][x + 3]);
8   }
9 }
```

Execution time is now split between communication time and computation time. Before performing stencil computation at each timestep, we must first wait for the MPI calls in the halo update to return. This is the **communication time**.

Devito provides many levels of MPI implementation. Above we see the simplest implementation. We can generate an optimised version that attempts to overlap communication and computation time. The optimised code will compute the inner points of the grid while performing asynchronous MPI calls. These inner points do not depend on the halo region. Once the MPI calls return, we can update the points at the boundaries. The code generated is more complex.

2.4.3 Hybrid MPI-OpenMP Parallelism with Devito

As mentioned earlier in Section 2.4, Devito can generate code implementing a hybrid model of parallelism. Below we see part of the code generated by Devito for the example used in Section 2.3.1. We note the use of both OpenMP and MPI libraries.

```
1  for (int t = time_m; t <= time_M; t += 1)
2  {
3    haloupdate(t);
4    #pragma omp parallel num_threads(nthreads)
5    {
6      #pragma omp for collapse(1) schedule(static,1)
7      for (int x = x_m; x <= x_M; x += 1)
8      {
9        u[t + 1][x + 2] = dt*(r0*u[t][x + 2] + r1*u[t][x + 1] +
10         r1*(-2.0F*u[t][x + 2]) + r1*u[t][x + 3]);
11     }
12   }
13 }
```

2.5 Loop Tiling

This Section introduces the loop tiling optimisation for stencil code. In Section 2.5.1 we discuss loop blocking and outline spatial tiling. Then, in Section 2.5.2 we focus on advanced tiling techniques which extend to the temporal dimension.

2.5.1 Loop Blocking

The execution of stencil code is generally limited by memory bandwidth. Each bit of data fetched is used in only a few floating-point operations, so CPU performance is not usually the limiting factor. Therefore, to optimise stencil computation, we look to improve memory reuse (in particular cache reuse).

To improve cache reuse, we consider **locality of reference**. A program will tend to access similar memory addresses in a given time period. We try to re-order floating point operations so our computation exhibits a stronger locality of reference. This means we can keep data blocks in cache for longer to reduce memory bandwidth. This is achieved through loop transformations.

A commonly used transformation is **loop blocking** (or **loop tiling**). The idea is to tile our for-loop iteration space with smaller blocks. Then, instead of iterating over the entire for-loop as usual, we iterate over the smaller tiles in turn. The block size is chosen to allow it to exist entirely in cache.

As an example, we present loop tiling in a loop with the following kernel update:

```
1 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]);
```

This is spatially 2-dimensional (n, m) and also has a temporal dimension, k. The kernel is the same as shown in Figure 2.2. Then an implementation of loop tiling is:

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

In the implementation, we tile our $N \times N$ iteration space with blocks of size $b \times b$ (where $b < N$). To illustrate the usefulness of this transformation, consider the scenario where our cache has size $16 = 4 \times 4$ and $N = 10$. We set $b = 4$. Note that the point p_0 at $(k, 2, 2)$ is used to calculate both $p_1 = (k + 1, 2, 3)$ and $p_2 = (k + 1, 3, 2)$. With a blocking implementation, we can keep all of our bottom left $b \times b$ square in cache (including the point p_0). Then p_1 and p_2 are both updated when visiting this tile, so p_0 can be retrieved from cache twice. In the standard implementation, this is not possible, as we will update $(k+1, 2, 10)$ before coming back to visit $(k+1, 3, 2)$.

A further usefulness of loop blocking is to extract parallelism. In each time step, all the tiles can be executed concurrently, as there are no inter-dependencies between the tiles. These tiles are particularly well-suited for applying OpenMP threading (see Section 2.4.1).

Three Dimensional Tiling

In the loop blocking example above, we demonstrate loop tiling with 2 spatial dimensions. When tiling in 3 spatial dimensions, there are two tiling strategies:

- **3D Tiling**. Extend the loop tiling strategy to 3 dimensions in the same way. We introduce blocking in the extra dimension.
- **2.5D Tiling**. Do not tile in the third dimension. We only apply loop tiling in the first two dimensions. We iterate over the last dimension as normal. Note iteration over the last dimension takes place as the most inner for-loop.

A comparison between the strategies may be seen in Figure 2.3. Devito generates code using 2.5D tiling. The larger block size allows for a greater degree of concurrency with OpenMP threads.

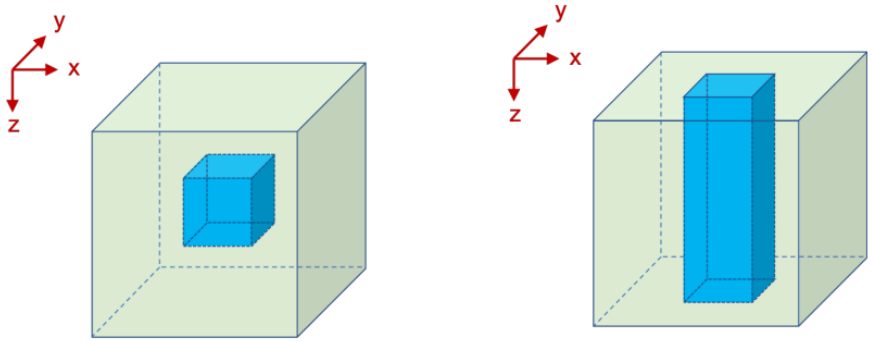


Figure 2.3: Comparison between 2.5D tiling and 3D tiling. On the left, we see 3D tiling, on the right 2.5D. Figure from Sai et al. [46].

2.5.2 Temporal Blocking

In Section 2.5.1 we introduced loop blocking. We note our tiles did not span more than one time step. We only tile spatially. This is called **spatial blocking**. To further improve cache reuse, we can extend our tiles with a time dimension. This is known as **temporal blocking**.

Temporal blocking is often not as straightforward as spatial blocking. This is due to dependencies that span across time steps. In the above example in Section 2.5.1, we could not just use a tiling of dimension $b \times b \times b$. To calculate values at the bottom face of a block, we would have to fetch values calculated in the block 'beneath'. Each tile should produce all the data it needs. This allows concurrent execution. This problem is addressed by wavefront tiling.

Wavefront Tiling

Wavefront tiling is suitable when we have diagonal time dependencies. For example, the following kernel:

```
1 u[t+1][x] = u[t][x-2] + u[t][x-1] + u[t][x] + u[t][x+1] + u[t][x+2]
```

We can use 'wave' shaped tiles to preserve dependencies, as in Figure 2.4.

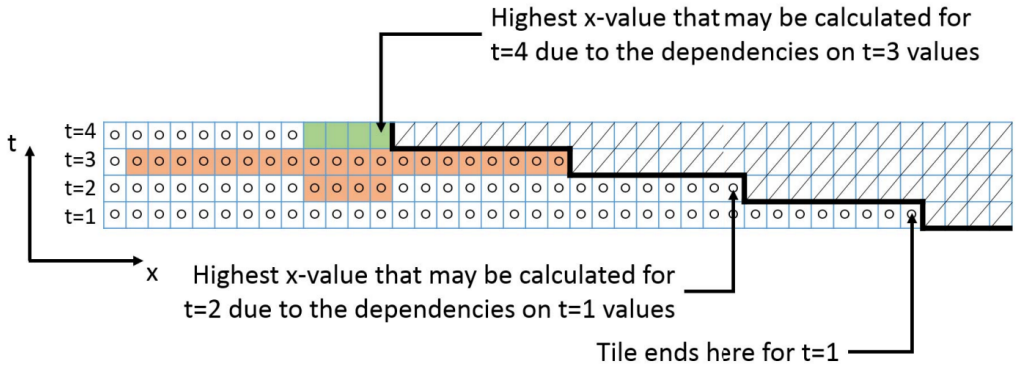


Figure 2.4: Wavefront tiling for a kernel with diagonal time dependencies. Each update at point (t, x) depends on $(t-1, j)$ and $(t-2, x)$, where j runs from $x-8$ to $x+8$. The green points are our points being updated. Note we update four points at once (see SIMD in Section 3.2). The orange points are our dependencies used in the update. Figure from Yount and Duran [55].

Note that we can combine wavefront tiling with the loop blocking introduced in Section 2.5.1. Within our larger wavefront tiles, we have smaller rectangular-shaped blocks which exist only in a single timestep. Using smaller blocks within a larger tile allows us to take advantage of different levels of cache.

Overlapped Tiling

A major challenge arises when attempting to use wavefront temporal blocking in conjunction with MPI processes (see Section 2.4.2). The code generated by Devito must perform a halo update before every timestep. This forces a sequential approach: calculate all values at t, halo update for $t + 1$, calculate all values at $t + 1$, halo update for $t + 2$, and so on. By performing a halo update every timestep, it is impossible to extend tiles with a time dimension. When using overlapped tiles with a height of T, we perform one halo update every T timesteps, instead of every timestep.

Overlapped tiling is implemented by increasing the number of grid points owned by each rank. Instead of perfectly partitioning the grid across all ranks, there is some overlap between ranks. This allows each rank to compute more timesteps without any communication from other ranks. As with wavefront tiling, the kernel dependencies produce a slanted shape. With each increasing timestep, the number of points we can calculate decreases. This occurs at both edges of the grid space, giving a trapezium shape (see Figure 2.5). More detail is provided in Section 4.2.1.

Once overlapped tiling has been applied, wavefront tiling can be applied within the larger overlapped tiles. This allows temporal blocking to be used with MPI. As a result, we benefit from the speed-up from both parallel computation and temporal locality. Note a drawback is the need for redundant computation. The computation in the overlapping regions is performed twice by different ranks.

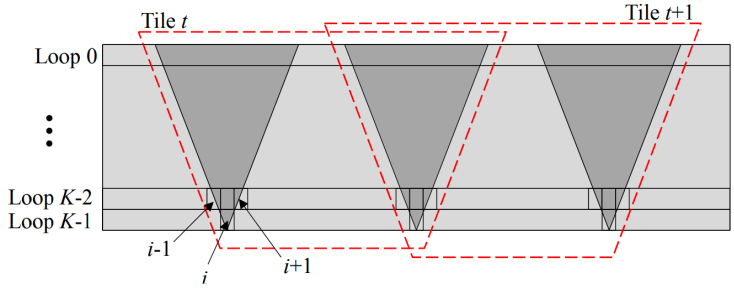


Figure 2.5: Overlapped tiling, where our trapezium tiles are outlined with red dotted borders. Note that in this image, time increases in the negative y-direction. Figure from Zhou et al. [58]

2.6 Concluding Remarks

In this Chapter, we equipped the reader with the background knowledge required in the ensuing Chapters. PDEs were introduced to the reader in Section 2.1 and in Section 2.2 we explained the key concepts behind the finite-difference method. In Sections 2.3 and 2.4 we focused on Devito before introducing the loop tiling optimisation in Section 2.5.

Chapter 3

Related Work

In this Chapter, we outline key work related to our investigations. In Section 3.1 we discuss different methods and code generation frameworks proposed to approximate solutions to PDEs. Then, in Section 3.2 we present positive results from various loop tiling schemes. In particular, we focus on overlapped tiling. This provides motivation for our implementation of overlapped tiling in Devito.

3.1 Solving Partial Differential Equations

In Section 3.1.1 we present alternative methods used to approximate solutions to PDEs. In Section 3.1.2 we focus on the popular finite element method and discuss frameworks implementing the method. In Section 3.1.3 we return to the finite difference method and outline implementing frameworks. Then we discuss polyhedral compilers, a related body of work, in Section 3.1.4.

3.1.1 Alternative Methods

In Section 2.2 we outlined the finite difference method for approximating solutions to PDEs. This is the method implemented by the Devito framework and is the focus of this paper. There are other commonly used methods:

- **Finite element method** [37]. We divide the domain of our problem into a finite set of smaller pieces. This set of small pieces is called a **mesh** (see Figure 3.1). Each of these has an associated PDE which is simpler to solve as extra assumptions may be made across the smaller element. We solve all of these simpler equations to then solve the problem across the whole domain. For a fine enough grid, our numerical approximation converges to the true solution.
- **Spectral element method** [40]. A variation on the finite element method. In each smaller subdomain, we approximate the solution using higher-order polynomials.
- **Finite volume method** [19]. Similar to the finite element method, we start by creating a mesh of smaller subdomains. Each subdomain is called a **control**

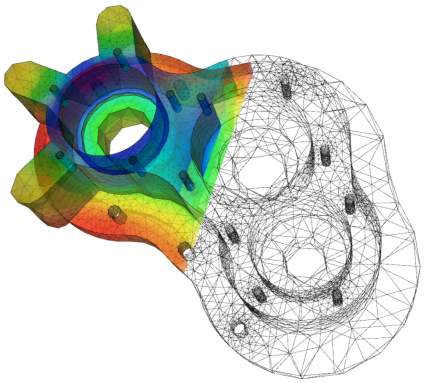


Figure 3.1: Example of a finite element mesh. Note our mesh can be unstructured, which allows application to more complex shapes. This is in contrast to the finite difference mesh, which requires a structured grid. Figure from Mourad [35]

volume. We then integrate our PDE over these control volumes to find new equations. This transforms the partial equations into linear equations, which are easier to solve.

3.1.2 Finite Element Method Frameworks

In this Section, we focus on code-generation frameworks implementing the finite element method (see Section 3.1.1).

FEniCS [30] is a framework that allows users to define PDE problems in a Python or C++ program. Users also define the mesh used in the finite element method (see Figure 3.1). FEniCS then produces optimised finite element code.

Listing 3.1: FEniCS specification of a PDE. On line 2 we see the definition of the finite element mesh. Code adapted from FEniCS tutorial [29]

```
1  # Create mesh and define function space
2  mesh = UnitSquareMesh(8, 8)
3  V = FunctionSpace(mesh, 'P', 1)
4
5  # Define boundary condition
6  u_D = Expression('1 + x[0]*x[0] + 2*x[1]*x[1]', degree=2)
7
8  bc = DirichletBC(V, u_D, boundary)
9
10 # Define variational problem
11 u = TrialFunction(V)
12 v = TestFunction(V)
13 f = Constant(-6.0)
14 a = dot(grad(u), grad(v))*dx
15 L = f*v*dx
16
17 # Compute solution
```

```
18 u = Function(V)
19 solve(a == L, u, bc)
```

FEniCS requires the user to specify their PDE problem using Unified Form Language (UFL) [9]. This is a high-level language that lets users formulate PDEs with numerical approximations in mind. UFL is an example of a **domain specific language** (DSL). A DSL is a language specialised to a particular application. In this case, the application is expressing PDEs.

Firedrake [43] is another framework that produces optimised finite element code. Like the FEniCS project, Firedrake allows the user to define a PDE problem using UFL.

PyFR [51] is a framework implementing the **flux reconstruction** method. This is a slight variation on the spectral element method (see Section 3.1.1). Unlike Devito and Firedrake, PyFR requires users to define stencil kernels rather than PDEs. The level of abstraction is lower.

3.1.3 Finite Difference Method Frameworks

The Devito framework is the main focus of this paper. Aside from Devito, there are other notable finite difference frameworks.

The Oxford Parallel library for structured mesh solvers (OPS) [45] is a DSL with support for C. The API offered by OPS is of a lower level than Devito. Instead of starting from the PDE specification, OPS allows the user to express a stencil specification in C. The code translation provided by OPS enables automatic parallelisation.

Listing 3.2: Stencil code specified using OPS. We see the declaration of stencils on lines 2-4. On lines 12-14, we see memory management. Code adapted from OPS sample program [44]

```
1  //declare stencils
2  int s2D_00[]       = {0,0};
3  ops_stencil S2D_00 = ops_decl_stencil( 2, 1, s2D_00, "00");
4  int s2D_00_P10_M10_0P1_0M1[]     = {0,0, 1,0, -1,0, 0,1, 0,-1};
5
6  // declare datasets
7  int d_p[2] = {1,1}; //max halo depth in the positive direction
8  int d_m[2] = {-1,-1}; //max halo depths in the negative direction
9  int base[2] = {0,0};
10 int uniform_size[2] =
       {(logical_size_x-1)/ngrid_x+1,(logical_size_y-1)/ngrid_y+1};
11 double* temp = NULL;
12 ops_dat *coordx = (ops_dat *)malloc(ngrid_x*ngrid_y*sizeof(ops_dat*));
13 ops_dat *coordy = (ops_dat *)malloc(ngrid_x*ngrid_y*sizeof(ops_dat*));
14 ops_dat *u = (ops_dat *)malloc(ngrid_x*ngrid_y*sizeof(ops_dat*));
```

OpenSBLI [24] is a Python framework that can generate finite difference code in C for a set of differential equations. This generated code is then specialised for specific hardware backends using the aforementioned OPS library. Supported backends include MPI, CUDA and OpenCL. When specifying the problem, the user must write their PDE in **Einstein notation** [6], which is a shorthand convention used to sum over the partial derivatives of a function.

ExaStencils [27] is another framework that generates stencil code from higher-level user input. A key advantage offered by ExaStencils is multi-layered DSLs. ExaStencils offers four DSLs for the user to formulate their problem. With rising layer number, the DSL becomes more abstract. This creates a wide range of target expertise. Layer 1 is a Latex-like formulation of the problem. Layer 4 allows fine-tuning of data structures used in stencil generation. ExaStencils is written in Scala.

STELLA [22] is a DSL for formulating stencil codes. Similar to OPS, STELLA provides a DSL which allows a user to express stencils rather than complete PDE problems. STELLA abstracts away from stencil composition by allowing the user to specify loop bounds and execution order. The focus of STELLA was on weather applications. GridTools [8] was a later attempt to improve the framework provided by STELLA. GridTools was reported to outperform STELLA in most applications. Moreover, GridTools was published as an open-source software. STELLA could not be made open source due to a dependence on the COSMO model, which was not open source.

Listing 3.3: Example stencil composition in DSL provided by STELLA. We see the definition of loop bounds (for example on line 5). Code from Gysi et al. [22]

```
1  define loops(
2      define sweep<cKIncrement>(
3       define stages(
4           StencilStage<Lap,
5           IJRange<cIndented,1,1, 1 ,1>,
6           KRange<FullDomain,0,0> >(),
7           StencilStage<Flx,
8           IJRange<cIndented,1,0,0,0>,
9           KRange<FullDomain,0,0> >(),
10          StencilStage<Fly,
11          IRange<cIndented,0,0,1,0>,
12          KRange<FullDomain,0,0> >(),
13          StencilStage<Res,
14          IJRange<cComplete,0,0,0,0>,
15          KRange<FullDomain,0,0> >()
16         )
17     )
18 )
```

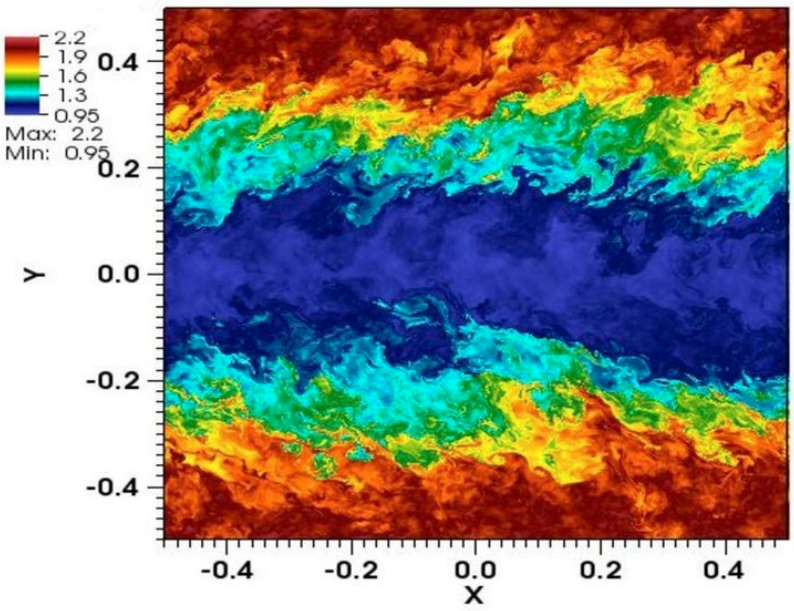


Figure 3.2: Numerical simulation of Kelvin-Helmholtz instability generated using Simflowny. Image from Simflowny wiki [33]

Simflowny [39] stands out from other finite difference frameworks by providing a graphical user interface (see Figure 3.2). Given a PDE, Simflowny produces efficient, parallelisable stencil code. One restriction on Simflowny is the system must be first order in time (e.g. cannot have any u_{tt} terms).

Earlier work in stencil code abstraction comes from Guo et al. (2008) [21]. The focus was on simplifying the implementation of overlapped tiling (see Section 2.5.2). A **hierarchically tiled array** (HTA) data type was created to help manage overlapping zones. The authors reported a 78% reduction in the number of communication statements when using the HTA abstraction.

3.1.4 Polyhedral Compilers

Polyhedral compilers are a related body of work aimed at code optimisation. The compilers take as input source code (e.g. nested loops) and output high-performing code. The source code is analysed by the compiler, which then selects a number of optimisations to apply. DSL compilers (sections 3.1.2, 3.1.3) can make use of these polyhedral compilers to optimise low-level code.

PLUTO [16] is a polyhedral framework aimed at optimising loop nests. Loop tiling is applied to optimise code and enable parallelism. The parallelism is implemented using the OpenMP API. PLUTO takes as input C code and outputs OpenMP C code. Many other miscellaneous optimisations are also applied. An example is vectorisation, where we transform loops to faster vector operations.

TIRAMISU [11] is another polyhedral framework designed for the optimisation of stencil code. TIRAMISU has an advantage over PLUTO by supporting GPU code generation, as well as CPU code generation. TIRAMISU offers transformations that map code to different memory hierarchies, processors and distributed machines. A C++ API is provided which allows the user to define the optimisation pipeline.

3.2 Loop Tiling

In Section 2.5.1 we presented loop tiling as a stencil optimisation. Multiple research efforts have focused on loop blocking as a means of improving memory bandwidth. Below we present some key publications. In Section 3.2.1 we focus specifically on work related to overlapped tiling. The large quantity of existing research suggests an overlapped tiling implementation in Devito is worthwhile.

Yount and Duran [54] presented the effectiveness of wavefront tiling (see Section 2.5.2) when applied to high-performance stencil computations. The authors ran experiments on an Intel Xeon Phi 7250 processor. The processor was equipped with high-bandwidth memory which could be fully exploited using temporal tiling. A 2.4x speed-up was recorded when using wavefront tiling. The wavefront tiling from the paper was used in the YASK framework [56]. The publication also presented other common stencil optimisations, including:

- **SIMD**: There is overlap in data use in neighbouring calculations (e.g. calculating $(t+1, x)$ and $(t+1, x+1)$ could both require (t, x)). SIMD involves calculating multiple values at the $t+1$ layer at once by taking advantage of this data overlap.
- **Vector Folding**: Instead of storing a $16 \times 1 \times 1$ block in cache, we could, for example, store a $4 \times 2 \times 2$. This increases data reuse when using SIMD as we overlap in many dimensions. The downside is a more complex memory layout.

Further work on wavefront tiling comes from Bisbas [13]. The author's contributions include a modification to the Devito compiler to generate wavefront tiled code. Positive results for stencils of space order 4 were reported (see Figure 3.3). The paper also discusses auto-tuning of temporal tiling parameters. When performing tiling, various parameters must be chosen. These include tile size in both spatial and temporal dimensions. The entire parameter space is very large. Auto-tuning to find the optimal parameters is therefore desirable.

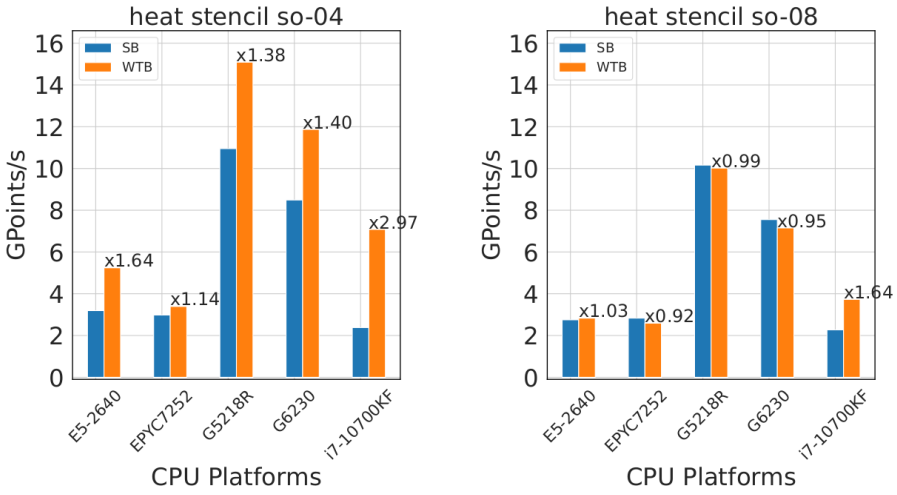


Figure 3.3: Devito wavefront tiling results when applied to the heat equation PDE. The author presents results across multiple architectures. The left graph is a stencil of space order 4 and the right of space order 8. Note in general we do not see any speed-up for space order 8. For information on the GPoints/s metric, see Section 5.1.3. SB denotes 'Spatial Blocking' and WTB denotes 'Wavefront Temporal Blocking'. Results from Bisbas [13].

Grosser et al. [20] proposed a hexagonal tiling layout (see Figure 3.4). The method combined hexagonal tiling across a temporal dimension with classical tiling in the spatial dimensions. The tiling scheme provided multi-level parallelism and greater data reuse, without any redundant computation. The authors reported an improvement in performance over contemporary stencil compilers.

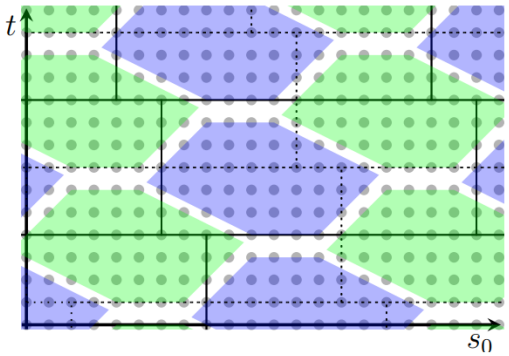


Figure 3.4: Hexagonal tiling proposed by Grosser et al [20]. Hexagonal tiles are applied across the time and one chosen spatial dimension, s_0.

Early exploration into loop tiling comes from Wolf and Lam (1991) [52]. Loop tiling was applied to various numerical algorithms. These included matrix multiplication, LU decomposition and QR factorisation. The authors presented several positive results and benchmarks indicating the effectiveness of tiling (see Figure 3.5). Among the earliest positive results on loop tiling are reported by McKellar and Coffman

(1969) [32] and Karp et al. (1967) [25]. The ideas were applied to matrix operations. *Loop Tiling for Parallelism* [53] is a detailed textbook authored by Xue (2000).

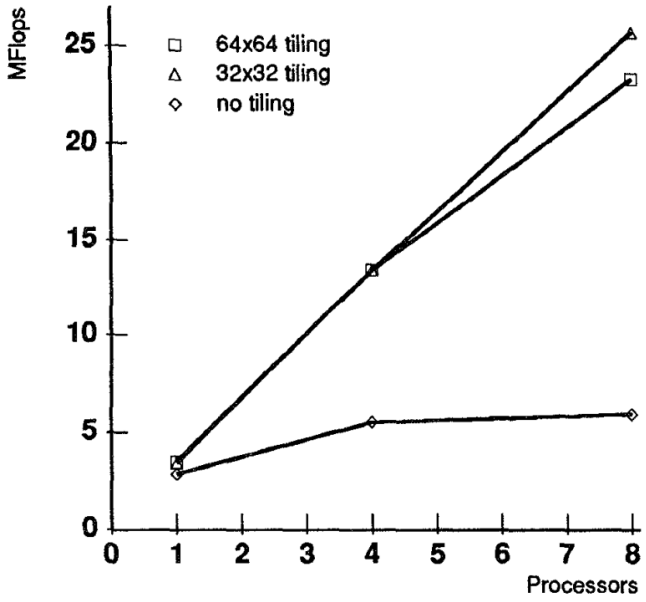


Figure 3.5: Performance of LU decomposition algorithm. We note a linear speed-up when increasing the number of processors when using tiling. Graph from Wolf and Lam [52]. For information on the MFlops/s metric see Section 5.1.3

.

3.2.1 Overlapped Tiling

We now focus our attention on landmark publications on the subject of overlapped tiling.

Holewinski et al. [23] presented code generation for stencil computation on GPU architectures. The scheme performed temporal tiling using overlapped tiles. There was a significant improvement in performance when using overlapped tiles (see Figure 3.6). The authors also investigated the impact of time tile sizes. For smaller tiles, global memory access was the limiting factor. For larger tiles, the limiting factor was computational overhead from large overlap. Positive results were also recorded by Takei et al. [48]. Overlapped tiling was implemented on an FPGA board using the OpenCL compiler. FPGA processing speed was observed to be 5-20 times faster with overlapped tiling.

In *Flextended Tiles*, Zhao and Cohen [57] implemented overlapped tiling for use in image processing stencils. The authors used a tighter trapezoid shape for tiling to reduce overlapped zones. This would optimise code by reducing the redundant computation.

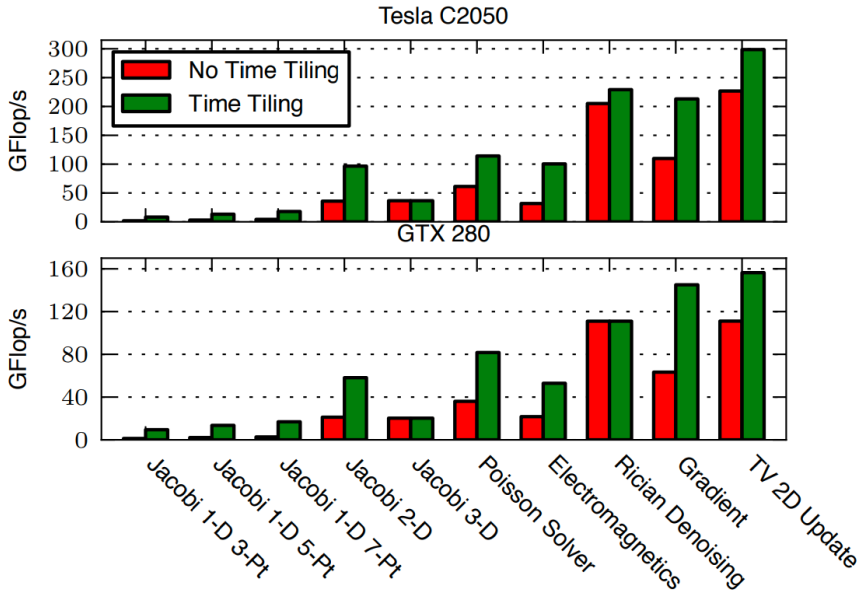


Figure 3.6: Comparison of stencil performance on two different architectures. Multiple different stencil computations are used in the comparison. Results from Holewinski ei al. [23]. For information on the GFlops/s metric, see Section 5.1.3.

Krishnamoorthy et al. [26] used overlapped tiling to create higher degrees of parallelisation in stencil code. The authors present the theory behind overlapped tiling and its benefits in facilitating **concurrent starts**. This is the idea that we can start the execution of multiple tiles concurrently. Code generation for overlapped tiling is also discussed.

Zhou et al. [58] introduced **hierarchical** overlapped tiling to reduce the amount of redundant computation. Instead of assigning an overlapped tile to each core, the space was first split up across processors. Each processor would be assigned an overlapped tile. Then within each processor tile, overlapped tiling was again applied. These smaller overlapped tiles were then assigned to cores within each processor. The authors noted an average 16% speed-up when using hierarchical overlapped tiling compared to regular overlapped tiling. The downside was increased compilation time.

3.3 Concluding Remarks

The analysis provided in this Chapter provides strong motivation for our exploration of overlapped tiling within Devito. The extremely wide range of code generation frameworks outlined in Section 3.1 demonstrates the high demand for optimised stencil code for PDEs. The positive results on overlapped tiling from Section 3.2.1 motivate us to explore this tiling scheme for distributed computation within Devito.

Chapter 4

Implementation

In this Chapter, we outline the implementation of overlapped tiling in 3D-stencil code. In Section 2.5.2, we introduce wavefront tiling and discuss the need for overlapped tiling. Tiling MPI code with overlapped tiles allows us to then tile our overlapped tiles with wavefront tiles. We begin by modifying the stencil code generated by Devito to implement wavefront tiling in just one dimension (Section 4.1). In Section 4.2 we demonstrate modified Devito MPI code which implements 1D-overlapped tiling without wavefront tiling. Then in Section 4.3, we combine our implementations to tile 1D-overlapped tiles with 1D-wavefront tiles. We conclude by presenting the final 3D implementation.

All code snippets throughout this Chapter are slightly modified from the source code to improve clarity. For example, we may remove lines of OpenMP code related to SIMD and loop collapsing (see Section 3.2). We also omit all closing '}' brackets of our for-loop and OpenMP blocks for brevity.

The initial generated Devito code is presented below:

```
1  for (int time = time_m; time <= time_M; time += 1)
2  {
3    #pragma omp parallel num_threads(nthreads)
4    {
5      for (int x_blk = x_m; x_blk <= x_M; x_blk += x_blk_size)
6      {
7        for (int y_blk = y_m; y_blk <= y_M; y_blk += y_blk_size)
8        {
9          for (int x = x_blk; x <= MIN(x_M, x_blk + x_blk_size - 1); x += 1)
10         {
11           for (int y = y_blk; y <= MIN(y_M, y_blk + y_blk_size - 1); y += 1)
12           {
13             for (int z = z_m; z <= z_M; z += 1)
14             {
15             // Stencil update at grid point (time, x, y, z)
```

We define the variables:

- **x_blk**, **y_blk**. The bottom (x, y)-coordinate of each loop tile.
- **x_blk_size**, **y_blk_size**. Constant parameters which define the x and y-dimensions of the loop tile.
- **time_m**, **time_M**, **x_m**, **x_M**, **y_m**, **y_M**, **z_m**, **z_M**. The dimensions of our grid.

On line 3, we see the OpenMP directive. We can calculate all points within the same timestep concurrently as there are no spatial dependencies. Note on lines 5 and 7, Devito performs loop tiling in the x-dimension and y-dimension respectively. The implementation is identical to the example given in Section 2.5.1. Recall we refer to this tiling as **spatial tiling**. Devito performs 2.5D tiling and does not tile in the z-dimension.

We must ensure every point we process is within the bounds of our problem. This explains the bound **MIN(x_M, x_blk + x_blk_size - 1)**, to prevent the x-value exceeding **x_M**.

On line 15 we see our stencil update. We implemented overlapped tiling for Laplace and wave stencils (see Section 5.1.2). However, outside of line 15, the choice of stencil does not impact the implementation. As such, we will not focus on any particular stencil in this Chapter.

4.1 Wavefront Tiling

We begin by demonstrating a wavefront implementation for a stencil with just one spatial dimension, x. For now, we will not consider the smaller spatial tiles. Our 1D implementation can be seen below:

```
1  for (int t_wf_blk = time_m; t_wf_blk <= time_M; t_wf_blk += wf_blk_height)
2  {
3    for (int x_wf_blk = x_m; x_wf_blk <= x_M; x_wf_blk += x_wf_blk_size)
4    {
5      for (int time = t_wf_blk, int wf_offset = 0; time <= MIN(time_M,
           t_wf_blk + wf_blk_height - 1); time += 1, wf_offset += angle)
6      {
7        #pragma omp parallel num_threads(nthreads)
8        {
9          for (int x = MAX(x_m, x_wf_blk - wf_offset); x <= MIN(x_M, x_wf_blk
               + x_wf_blk_size - wf_offset - 1); x += 1)
10         {
11         // Stencil update at (time, x)
```

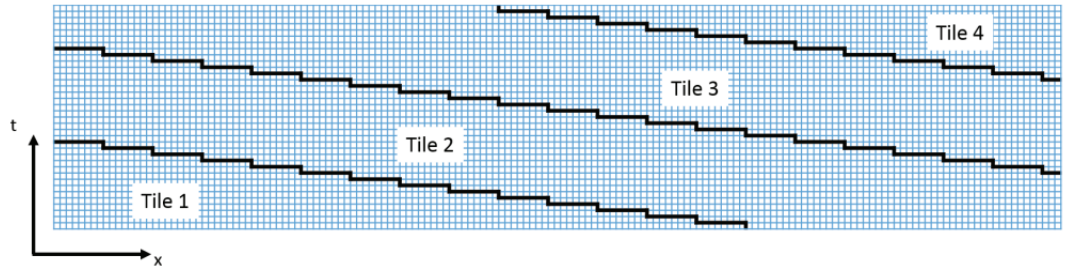


Figure 4.1: Wavefront tiling in one spatial dimension. Figure from Yount and Duran [55].

We define the variables:

- **t_wf_blk**. The time value at the base of each wavefront tile.
- **wf_blk_height**. Constant parameter defining the height of each wavefront tile. Read in as an environment variable.
- **x_wf_blk**. The starting x-coordinate of each wavefront tile.
- **x_wf_blk_size**. Constant parameter defining the x-width of each wavefront tile. Read in as an environment variable.
- **wf_offset**. The offset from the initial base of the wave tile at the current timestep. Figure 4.1 demonstrates wavefront tiling. We note the slanted shape. When moving up each timestep, we must increase this offset to obtain the slanted slope.
- **angle**. Constant parameter defining the angle of our wavefronts. We increase the wave offset by **angle** every timestep. For our purposes, we take **angle** to be $s/2$, where s denotes the space order of our stencil. The space order s stencil will access $s/2$ points on either side of the point being processed.

On line 9, we see the use of our wavefront offset. The starting x-coordinate is decreased by this wavefront offset. The upper bound is also decreased by the same offset.

We then add spatial tiling to our 1D wavefront implementation. Lines 9-11 now become:

```
1 for (int x_blk = MAX(x_wf_blk - wf_offset, x_m); x_blk <= x_wf_blk +
      x_wf_blk_size - wf_offset - 1; x_blk += x_blk_size)
2 {
3   for (int x = x_blk; x <= MIN(MIN(x_M, x_wf_blk + x_wf_blk_size -
        wf_offset - 1), x_blk + x_blk_size - 1); x += 1)
4   {
5   // Stencil update at (time, x)
```

On line 3, we see three different upper bounds for our x-value:

1. The boundary of the grid, **x_M**.
2. The boundary of our wave tile, **x_wf_blk + x_wf_blk_size - wf_offset -1**
3. The boundary of our spatial tile, **x_blk + x_blk_size - 1**

On line 1 we can see the lower bound of the spatial tile already takes into account the lower boundaries of both our grid and the wave tile.

4.2 Overlapped Tiling

We now discuss the implementation of overlapped tiling. Note in this Section, we will not tile our overlapped tiles with wavefront tiles. This will be covered in Section 4.3. Here, we will outline pure overlapped tiling. The goal is to pass MPI messages between processes once every T timesteps, instead of once every timestep. T denotes the overlapped tile height. We will ignore spatial tiling and OpenMP directives in this Section. First, we increase the size of the halo region of each process such that our tiles overlap. Once this is achieved, we modify the loop bounds to obtain our trapezium shape.

4.2.1 Increasing the Halo Region

As in Section 4.1, we start by considering one spatial dimension. The aim is to allocate enough grid points to avoid the need for communication until we reach timestep T. The process must be able to calculate all points (t, x), where $x \in \{x_m, ..., x_M\}$, $t \in \{0, ...T-1\}$. Note this will only cover the bottom overlapped tile running from $t \in \{0, ...T-1\}$. However, all subsequent tiles are the same.

For simplicity, we will first cover the stencil:

```
1 u[t+1][x] = u[t][x-1] + u[t][x] + u[t][x+1]
```

Suppose at timestep t, we are able to calculate all points (t, x), $x \in \{x_b, ...x_B\}$ (note x_b, x_B is an arbitrary range lying within our grid boundaries, x_m, x_M). Then moving from t to $t+1$, we find we are unable to process the point at $(t+1, x_b)$. This point relies on $(t, x_b - 1)$, which we have not calculated. This is illustrated in Figure 4.2. Without receiving this value from another process, we cannot proceed. The same applies to $(t+1, x_B)$. Hence, we can only process the points $(t+1, x)$ for $x \in \{x_b+1, ...x_B-1\}$.

We see the bounds of the computable points shrink by one for each timestep. Therefore, to calculate (T, x), $x \in \{x_m, ..., x_M\}$, we must allocate an extra $T-1$ points either side of our grid.

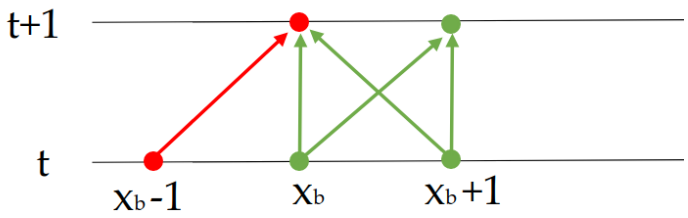


Figure 4.2: Data dependencies when moving from timestep t to timestep $t+1$. Points in green are points we can calculate. Points in red cannot be calculated from the points in green. Arrows indicate the dependencies in our stencil calculations. An arrow from point a to b indicates we require the value at a to compute b.

Generalising, for a stencil accessing $\frac{d}{2}$ points on either side of the point being processed, we must allocate an extra $\frac{d}{2}(T-1)$ halo points on each side. For all PDEs discussed in this paper, a stencil of space order s will access $\frac{s}{2}$ on either side of the point being processed. Hence we allocate an extra $\frac{s}{2}(T-1)$ halo points.

This is achieved by modifying Devito's **halo_setup** function. Below we see part of the modification made in Python:

```
1 time_tile_size = os.getenv('TIME_TILE_SIZE')
2 halo_size += (space_order // 2) * (time_tile_size - 1)
```

On line 1 we read the time tile size from an environment variable.

4.2.2 Modifying the Loop Bounds

Below we see an implementation of overlapped tiling in one spatial dimension:

```
1 for (int t_ol_blk = time_m; t_ol_blk <= time_M; t_ol_blk += ol_blk_height)
2 {
3   haloupdate(t_ol_blk);
4   for (int time = t_ol_blk, int ol_offset = (ol_blk_height - 1) * angle;
        time <= MIN(time_M, t_ol_blk + ol_blk_height - 1); time += 1,
        ol_offset -= angle)
5   {
6     for (int x = x_m - ol_offset; x <= x_M + ol_offset; x += 1)
7     {
8     // Stencil update at (time, x)
```

We define the variables:

- **t_ol_blk**. The time value at the bottom of each overlapped tile.
- **ol_blk_height**. The height of each overlapped tile.

- **ol_offset**. The extra space on either side of x_m and x_M visited by the stencil at the current timestep. Note when allocating extra halo space, x_m and x_M are unaffected. Hence, at the bottom of the tile, we must use an offset equal to the number of extra points allocated. We then decrease this offset by **angle** each timestep (see Section 4.2.1).

On line 3 we see our halo update. Note this is now performed once every T timesteps, as required.

Processes on the Grid Boundaries

We now consider an important edge case. When executing stencil code with MPI processes, we first divide the grid between each process. Consider the case for just two processes. Denote $x_{\frac{M}{2}} = \frac{x_M}{2}$ (we assume x_M is even). For simplicity, assume $x_m = 0$. The first process will be assigned the grid $x \in \{0, ...x_{\frac{M}{2}} - 1\}$, the second $x \in \{x_{\frac{M}{2}}, ..., x_M\}$. Then, we note if we proceed with the code above, the first process will visit negative values of x when applying overlapped tiling to the left. Similarly, the second process will visit values of x which will exceed x_M. This is incorrect, as these points are outside our problem grid and cannot be accessed.

We must prevent the first process from applying the overlapped tile slanting to its left and the second process from applying the overlapped tile slanting to its right. This is a difficult problem to solve. Devito generates code with the philosophy that the same code should be executed on every process. We cannot follow this philosophy, as a process on the edge of the grid cannot apply standard overlapped tiling. There are two possible approaches:

- **Provide each process with information about the global grid boundaries**. When Devito executes stencil code in each process, the process only knows the bounds of the grid it has been allocated. For the example above, both processes will see the grid bounds $x \in \{0, ...x_{\frac{M}{2}} - 1\}$. Neither process has any knowledge of the global x_M bound. This is done to ensure the code on each process is the same.

  By providing extra information about the boundaries of the process, we can prevent the process from accessing points out of bounds. We must provide:

  - The global x_M bound.
  - The relative bounds of the process. For example, the second process will be given the bounds $\{x_{\frac{M}{2}}, ..., x_M\}$

- **Use the neighbourhood construct to check for neighbouring processes**. When we pass messages using MPI, we need to send messages to the correct process. If we briefly consider four processes on our x-grid, each process must know its neighbouring processes. For example, the second process must know

to transfer its rightmost halo region to the third process and its leftmost halo region to the first process.

Devito defines a C struct called **neighborhood** to hold information about neighbouring processes. The definition of the struct in one dimension can be seen below:

```
1    struct neighborhood
2    {
3      int l;
4      int c;
5      int r;
6    } ;
```

By accessing the elements **l** and **r**, we obtain the MPI ranks of the processes to the left and right of our process respectively. If this process does not exist, the value is set to **MPI_PROC_NULL**. Then the process is the leftmost process if and only if **l** is set to **MPI_PROC_NULL**. This process will not apply the overlapped tiling offset to its left.

Note the field **c** in the struct is redundant for one spatial dimension. However, it is needed for two spatial dimensions. For example, **lc** is the process to our left in the x-dimension, but with the same y-bounds as us.

The second approach is preferred. Implementing the first approach would require large modifications of the Devito compiler. The second approach uses values already provided by Devito.

Using this approach, we may define functions to check if a process is the 'leftmost' or 'rightmost' process. Below is the updated code covering this edge case:

```
1  for (int t_ol_blk = time_m; t_ol_blk <= time_M; t_ol_blk += ol_blk_height)
2  {
3    haloupdate(t_ol_blk);
4    for (int time = t_ol_blk, int ol_offset = (ol_blk_height - 1) * angle;
         time <= MIN(time_M, t_ol_blk + ol_blk_height - 1); time += 1,
         ol_offset -= angle)
5    {
6      int x_ol_offset_lower = ol_offset;
7      int x_ol_offset_upper = ol_offset;
8
9      if (is_left)
10       ox_l_offset_lower = 0;
11
12     if (is_right)
13       x_ol_offset_right = 0;
14
```

```
15      for (int x = x_m - ol_offset_lower; x <= x_M + ol_offset_upper; x += 1)
16      {
17      // Stencil update at (time, x)
```

We define the following variables:

- **is_left**, **is_right**. Indicates if our process is the leftmost or rightmost process. These Boolean flags are set before any stencil calculations.
- **x_ol_offset_lower**, **x_ol_offset_upper**. Offsets are used for overlapped tiling at the lower and upper bounds on x respectively. We note if the process is the leftmost process, the lower bound is set to zero. Similarly for the upper bound for the rightmost process.

4.3 Overlapped Tiling with Wavefront Tiling

We now combine the work from Sections 4.1 and 4.2 to implement overlapped tiling with wavefront tiling. The goal is to tile our overlapped tiles with wavefront tiles. Note the height of the overlapped tile (**ol_blk_height**) and the height of the wavefront tile (**wf_blk_height**) are equal. The variable **wf_blk_height** will not appear in this Section, we will use **ol_blk_height**.

We first provide an implementation for overlapped tiling with wavefront tiling in just one spatial dimension, without spatial tiling:

```
1  int ol_offset_tot = (ol_blk_height - 1) * angle;
2  for (int t_ol_blk = time_m; t_ol_blk <= time_M; t_ol_blk += ol_blk_height)
3    {
4      haloupdate(t_ol_blk);
5        for (int x_wf_blk = x_m - ol_offset_tot; x_wf_blk <= x_M +
             ol_offset_tot; x_wf_blk += x_wf_blk_size)
6        {
7          for (int time = t_ol_blk, int wf_offset = 0, int ol_offset =
               ol_offset_tot; time <= MIN(time_M, t_ol_blk + ol_blk_height -
               1); time += 1, wf_offset += angle, ol_offset -= angle)
8          {
9          int x_ol_offset_lower = ol_offset;
10         int x_ol_offset_upper = ol_offset;
11
12         if (is_left)
13             x_ol_offset_lower = 0;
14
15         if (is_right)
16             x_ol_offset_right = 0;
17
18         #pragma omp parallel num_threads(nthreads)
19         {
```

```
20          for (int x = MAX(x_wf_blk - wf_offset, x_m - x_ol_offset_lower); x
                <= MIN(x_M + x_ol_offset_upper, x_wf_blk + x_wf_blk_size -
                wf_offset - 1); x += 1)
21          {
22            // Stencil update at (time, x)
```

We define the following variable:

- **ol_offset_tot**. Constant parameter defining the overlapped offset at the base of each overlapped tile. We note it is equal to the number of extra points added to our halo region.

On line 22 we see the bounds of both an overlapped tile and a wavefront tile.

We then add spatial tiling to our 1D implementation. Lines 22-24 now become:

```
1 for (int x_blk = MAX(x_wf_blk - wf_offset, x_m - x_ol_offset_lower); x_blk
    <= x_wf_blk + x_wf_blk_size - wf_offset - 1; x_blk += x_blk_size)
2 {
3   for (int x = x_blk; x <= MIN(MIN(x_M + x_ol_offset_upper, x_wf_blk +
      x_wf_blk_size - wf_offset - 1), x_blk + x_blk_size - 1); x += 1)
4   {
5     // Stencil update at (time, x)
```

On line 3, we see three different upper bounds for our x-value:

1. The boundary of the overlapped tile, **x_M + x_ol_offset_upper**.
2. The boundary of our wave tile, **x_wf_blk + x_wf_blk_size - wf_offset -1**
3. The boundary of our spatial tile, **x_blk + x_blk_size - 1**

Finally, we present the complete 3D implementation of overlapped tiling with wavefront tiling:

```
1  int ol_offset_tot = (ol_blk_height - 1) * angle;
2  for (int t_ol_blk = time_m; t_ol_blk <= time_M; t_ol_blk += ol_blk_height)
3    {
4      haloupdate(t_ol_blk);
5      for (int x_wf_blk = x_m - ol_offset_tot; x_wf_blk <= x_M +
         ol_offset_tot; x_wf_blk += x_wf_blk_size)
6      {
7        for (int y_wf_blk = y_m - ol_offset_tot; y_wf_blk <= y_M +
           ol_offset_tot; y_wf_blk += y_wf_blk_size)
8        {
9          for (int time = t_ol_blk, int wf_offset = 0, int ol_offset =
             ol_offset_tot; time <= MIN(time_M, t_ol_blk + ol_blk_height -
             1); time += 1, wf_offset += angle, ol_offset -= angle)
10         {
11         int x_ol_offset_lower = ol_offset;
12         int x_ol_offset_upper = ol_offset;
```

```
13          int y_ol_offset_lower = ol_offset;
14          int y_ol_offset_upper = ol_offset;
15
16          if (is_left)
17              x_ol_offset_lower = 0;
18
19          if (is_right)
20              x_ol_offset_right = 0;
21
22          if (is_bottom)
23              y_ol_offset_lower = 0;
24
25          if (is_top)
26              y_ol_offset_upper = 0;
27
28          #pragma omp parallel num_threads(nthreads)
29          {
30              for (int x_blk = MAX(x_wf_blk - wf_offset, x_m -
                    x_ol_offset_lower); x_blk <= x_wf_blk + x_wf_blk_size -
                    wf_offset - 1; x_blk += x_blk_size)
31              {
32              for (int y_blk = MAX(y_wf_blk - wf_offset, y_m -
                    y_ol_offset_lower); y_blk <= y_wf_blk + y_wf_blk_size -
                    wf_offset - 1; y_blk += y_blk_size)
33              {
34                  for (int x = x_blk; x <= MIN(MIN(x_M + x_ol_offset_upper,
                        x_wf_blk + x_wf_blk_size - wf_offset - 1), x_blk +
                        x_blk_size - 1); x += 1)
35                  {
36                  for (int y = y_blk; y <= MIN(MIN(y_M + y_ol_offset_upper,
                        y_wf_blk + y_wf_blk_size - wf_offset - 1), y_blk +
                        y_blk_size - 1); y += 1)
37                  {
38                      for (int z = z_m; z <= z_M; z += 1)
39                      {
40                      // Stencil update at (time, x, y, z)
```

Recall we implement 2.5D tiling, so do not tile in the z-direction. We refer to the processes at the boundaries of the y-axis as the 'bottommost' and 'topmost'.

4.4 Concluding Remarks

In this Chapter, we presented our implementations of wavefront and overlapped tiling. In Sections 4.1 and 4.2 we outlined our wavefront and overlapped tiling implementations respectively and explained key design decisions. Then, in Section 4.3 we combined our implementations to tile our overlapped tiles with wavefront tiles.

Chapter 5

Evaluation

In this Chapter we report performance improvements when using overlapped tiling as presented in Chapter 4. By implementing overlapped time tiling, we obtained a notable speed-up in stencil execution time when compared to the standard MPI code generated by Devito. We can then conclude we are benefiting from temporal locality within distributed computation, as desired. These results can be found in Section 5.2. Then, in Section 5.3, we discuss the impact of tuning our overlapped tile dimensions. The Chapter concludes with Section 5.4, where we investigate the impact of overlapped tiling on MPI communication times. In this Chapter, we also present mathematical models for the impact of overlapped tiling on redundant computation and MPI communication times. These are found in Section 5.3.2 and Section 5.4 respectively. We conclude with an investigation into the scalability of overlapped tiling in Section 5.5.

5.1 Experimental Setup

We benchmark the overlapped implementation against the standard MPI code generated by Devito. Both stencil computations were executed across two different CPU platforms (see Section 5.1.1). After execution, we calculated the L2-norm [7] of the produced grid points. To verify the correctness of the overlapped implementation, we ensured the norm produced by the overlapped implementation matches that of Devito's standard implementation. In Section 5.1.2, we introduce the two different stencils used in the experiments. All experiments were repeated three times and an average result was taken.

5.1.1 CPU Platforms

Two different CPU architectures were used to run our experiments. The first CPU is the Intel(R) Xeon(R) CPU E5-2660 v2 model. The second is the Intel(R) Xeon(R) CPU E5-2640 v3 model. Both CPUs have three levels of cache: L1d/i, L2 and L3. Our C stencils are compiled with the GCC compiler in both architectures. We used GCC versions 9.4.0 and 8.5.0 on the Xeon v3 and Xeon v2 architectures respectively.

CPU Properties		
	Xeon v2	Xeon v3
CPUs	40	32
Threads per core	2	2
Cores per socket	10	8
Sockets	2	2
CPU GHz	3	3.4
L1 cache	32KiB	32KiB
L2 cache	256KiB	256KiB
L3 cache	25MiB	20MiB

Table 5.1: Comparison of CPU properties used in our experiments.

The CPUs have different L3 cache sizes. This is important, as temporal blocking is a cache-related optimisation. A summary of the CPU properties is displayed in Table 5.1. On both architectures, we ran our MPI experiments with 2 ranks (excluding the results presented in Section 5.5).

Thread Pinning

As outlined in Section 2.4.1, Devito uses the OpenMP library to implement threading. Both the standard Devito code and our overlapped tiling implementation take advantage of this parallelism.

These threads exist within the same process and share resources, for example, cache and memory bandwidth. The execution of stencil code is therefore highly dependent on the OpenMP threads working efficiently in parallel. To ensure this, we pin OpenMP threads to specific cores. This prevents interference between threads running on different cores. Pinning is also important to ensure results are reproducible. Without pinning, the execution time will be dependent on the thread scheduling of a particular run.

Thread pinning with OpenMP can be achieved using OpenMP environment variables. These are *OMP_PROC_BIND* and *OMP_PLACES*. When using a hybrid parallel model with MPI, we must also provide pinning instructions through MPI. Experiments were run with the MPI directives *–bind-to* and *–map-by*.

5.1.2 Stencils

We use the 4-dimensional Laplace and wave equations in our experiments. Both these PDEs are commonly used to model various physical scenarios. From the equations, we produce stencils of varying space orders (see Section 2.2.2). In our experiments, we consider space orders of 2, 4 and 8. All experiments were run on a finite difference grid of dimension $512 \times 512 \times 512$ with 256 timesteps.

Laplace Equation

The Laplace equation is a second-order PDE commonly used to model heat transfer and diffusion. The Laplace equation is given by:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = 0 \tag{5.1}$$

This 3-dimensional spatial equation inspires a family of 4-dimensional Laplacian PDEs of the form:

$$\frac{\partial u}{\partial t} = \alpha(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}) + c \tag{5.2}$$

For our experiments, we took $\alpha = 0.5$ and $c = 0.1$. We defined $t = 0, u(0, x, y, z) = 1$ as the initial boundary condition. Note taking $\alpha = 1$ and $c = 0$ gives the heat equation described in Section 2.2.2.

Equation (5.2) gives rise to a space order 2 stencil of the form:

```
u[t+1][x][y][z] = b * u[t][x][y][z] + d * (u[t][x-1][y][z] +
    u[t][x+1][y][z] + u[t][x][y-1][z] + u[t][x][y+1][z] + u[t][x][y][z-1] +
    u[t][x][y][z + 1]) + c
```

Above, b and d are constants depending on α and c is as defined in Equation 5.2.

Wave Equation

Wave equations are a family of PDE used to model the motion of waves, such as sound waves or light waves. The wave equation is second order in time. One possible form of the wave equation is given by:

$$\frac{\partial^2 u}{\partial t^2} = w(x, y, z)\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}\right) + v(x, y, z)\frac{\partial u}{\partial t} \tag{5.3}$$

In the above equation, the function $v(x, y, z)$ is a **velocity profile**, describing observed physical properties of the wave. v is dependent only on spatial coordinates. $w(x, y, z)$ is another function also dependent only on spatial coordinates. The above form is known as the **damped wave equation**.

For our experiments, we defined v, w as:

$$v(x, y, z) = \begin{cases} 1.5 & \text{if } z \leq 50 \\ 2.5 & \text{otherwise} \end{cases} \quad w(x, y, z) = \frac{1}{v(x, y, z)^2}$$

We used the initial boundary condition $u(0, \frac{x_M}{2}, \frac{y_M}{2}, z_M - 20) = 0.1$, where $x_M \times y_M \times z_M$ were the spatial dimensions of the grid allocated to the MPI process. The stencil produced by Equation 5.3 is complex and not reproduced here for brevity.

5.1.3 Metrics

When evaluating the performance of stencil execution, there are three metrics we can consider:

- **Flops/s**. Measures the number of floating point operations executed per second.
- **Points/s**. Measures the number of grid points processed per second by stencil computation.
- **Elapsed Time**. Measures the total execution time of the stencil computation.

We will focus on Elapsed Time in this chapter. When executing our MPI code, we have two contributions to Elapsed Time, computation time and communication time (see Section 2.4.2). By focusing on Elapsed Time, we can easily break down our execution times into the two components. This allows us to investigate the impact of overlapped tiling on both communication and computation.

Despite this, we will also briefly present results using Points/s. Flops/s and Points/s are more commonly seen in the literature than Elapsed Time. We prefer Points/s to Flops/s as it is a more robust metric when considering mathematically optimised kernels. The same PDE equation can produce many valid stencils. By performing various mathematical optimisations, we may produce a stencil that requires fewer floating point operations. The optimised stencil may have a faster execution time while having lower (worse) Flops/s. By using Points/s instead, we can compare across PDE equations without worrying about any applied optimisations.

There is a strong correspondence between Elapsed Time and Points/s. A lower Elapsed Time will correspond to higher Points/s. Note we use GPoints/s to measure the billions of grid points processed per second.

5.2 Performance Evaluation

In this Section, we evaluate the performance of our overlapped tiling implementation presented in Chapter 4. We first demonstrate preliminary results which provide motivation for overlapped tiling. Then, the performance of overlapped tiling is evaluated against the standard MPI code produced by Devito.

5.2.1 Preliminary Results

Both the experiments discussed in this Section were executed with the Laplace stencil.

In Section 2.4.2 we discussed the use of MPI processes within Devito. We predicted using MPI to implement distributed computing on a high-performance computer will

speed up stencil execution on the Xeon v3 CPU. In Figure 5.1, we can observe the benefits of distributed computing. We compare Devito code implementing MPI processes against Devito code executing on a single process. A speed-up is seen for all space orders. For space order 4, the MPI code is 41% faster. Note the speed-up is less for higher space orders. This is expected, as our halo regions are larger. Hence, each MPI call must move more data, so communication times increase. All code in this experiment was generated by the Devito compiler.

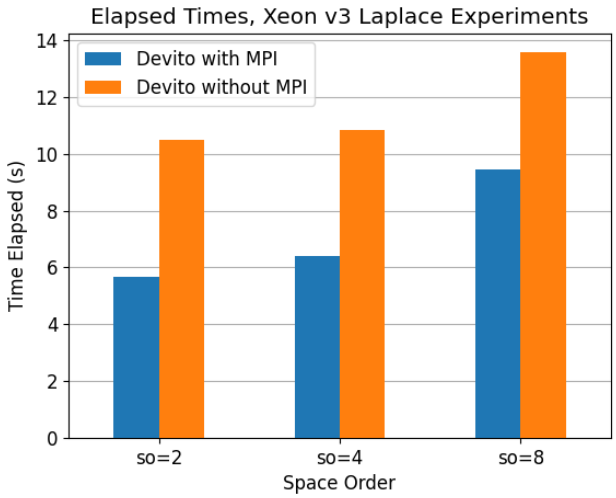


Figure 5.1: Improvement in Devito code performance when executed on multiple MPI processes vs. a single process.

In Section 2.5.2 we discussed wavefront tiling. We suggested the use of wavefront tiles in Devito stencil code allows us to take advantage of temporal blocking. This should speed up stencil execution. In Figure 5.2 we observe the benefits of wavefront tiling. Here we present results from both Xeon v2 and Xeon v3 architectures.

A speed-up is seen for all space orders. However, the improvement is less notable for higher space orders. When executing on Xeon v3 for space order 2, the wavefront tiling code is 63% faster. For space order 8, it is only 3% faster. This is due to the larger stencil size. The larger stencil requires more memory accesses to process. Hence, it is more difficult to take advantage of cache. The wavefront code was produced by manually modifying code generated by Devito (see Section 4.1). Note our results mirror those reported by Bisbas in Figure 3.3.

Figures 5.1 and 5.2 demonstrate the speed-up from distributed parallelism and wavefront tiling respectively. We would like to be to take advantage of both improvements simultaneously. This motivates the implementation of overlapped tiling. Recall from Section 2.5.2 that larger overlapped tiles can be tiled with smaller wavefront tiles. From Figure 5.2, we expect the speed-up provided by overlapped tiling to be most notable for space order 2. We do not expect a significant improvement for space order 8.

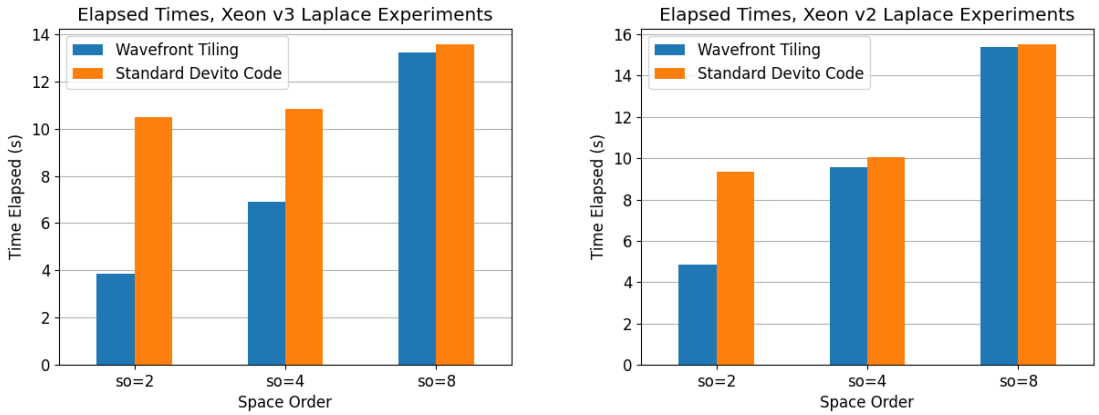


Figure 5.2: Improvement in Devito code performance when executed with wavefront tiling. There is no distributed parallelism involved. All code is run on a single process in this experiment. On the left, we see results from Xeon v3 and on the right from Xeon v2. Note the lack of major improvement for space order 8.

5.2.2 Overlapped Tiling Performance

We now present the key result from this Chapter. In Figure 5.3 we compare Laplace stencil execution times for overlapped MPI code against standard MPI code on the Xeon v3 CPU. We note a speed-up across all space orders. As predicted in Section 5.2.1, the speed-up is most notable for space order 2. For space order 2, our overlapped code is 55% faster than the MPI code generated by Devito. For space orders 4 and 8, our overlapped implementation runs 28% and 7% faster respectively. The overlapped code was produced by manually modifying the MPI code generated by Devito (see Section 4.3).

As outlined in Section 5.1.3, we focus on Elapsed Time as our key metric. A comparison of GPoints/s for the experiment can be seen in Figure 5.4. We note a strong correspondence between the two metrics. For example, using overlapped tiling for space order 8 saw an 8% increase in GPoints/s. This matches the 7% speed-up (allowing for rounding errors when collecting results).

In Figure 5.5 we present further positive results from overlapped tiling. This time, we show results from the Xeon v2 architecture. A speed-up is noted for the execution of both Laplace and wave stencils. The wave stencils take longer to execute than the Laplace stencils. This is due to the increased complexity of the wave stencil, which requires more memory accesses. A similar pattern across both architectures and stencils can be seen when considering space order. Less speed-up is recorded the higher the space order.

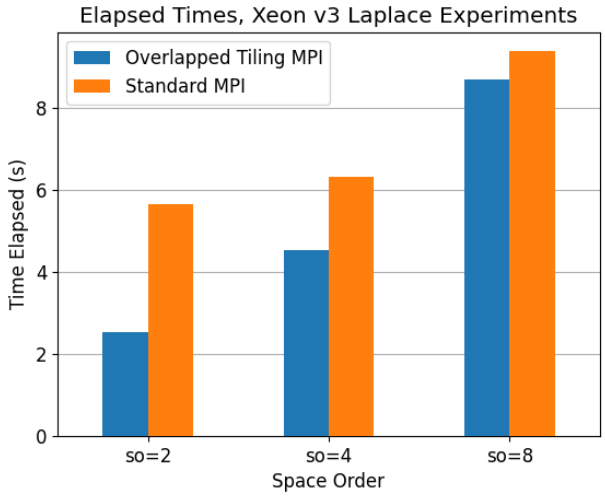


Figure 5.3: Improvement in Devito MPI code performance when executed with overlapped tiling on Xeon v3 architecture. The experiment uses the Laplace stencil. Overlapped code produced by modifying code generated by Devito.

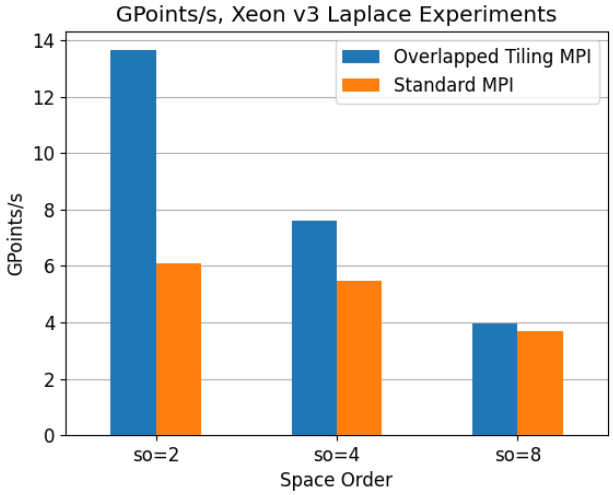


Figure 5.4: Improvement in Devito MPI code performance when executed with overlapped tiling. Results from the same experiment are displayed in Figure 5.3. Here we compare GPoints/s rather than Elapsed Time.

An exception to this trend may be seen in the Xeon v2 Laplace experiments. When executing the space order 8 Laplace stencil, we record a speed-up of 24%. This is greater than the 21% speed-up recorded when executing the space order 4 Laplace stencil. The result can be partially explained by consulting Figure 5.2. On the right graph, we display results from the Laplace wavefront experiment on the Xeon v2 architecture (no MPI involved). Unlike the Xeon v3 results shown on the left, wavefront tiling does not produce a significant speed-up for space order 4. This is due to the smaller cache size of Xeon v3. The smaller cache prevents the architecture from taking advantage of the tiling cache optimisation. Therefore, the speed-up from

space orders 4 and 8 is not due to the use of wavefront tiling within overlapped tiling. The speed-up can be explained by considering the impact of overlapped tiling on MPI communication times (see Section 5.4).

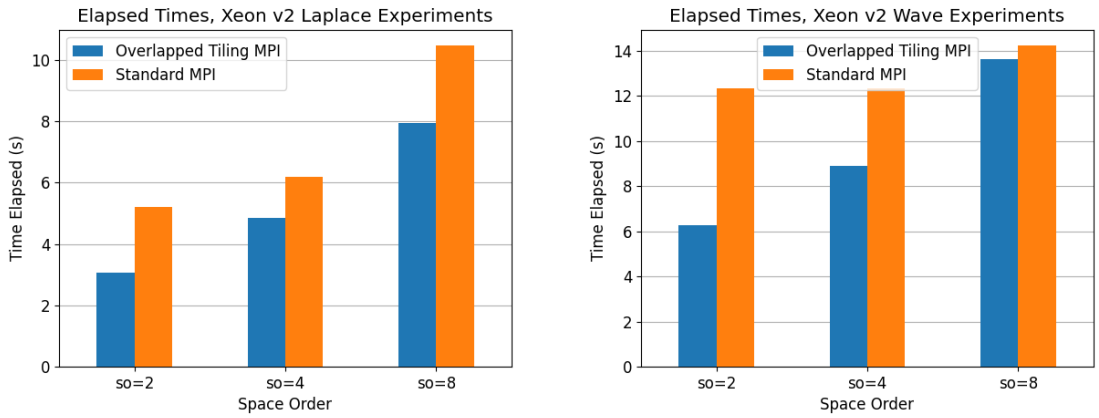


Figure 5.5: Improvement in Devito MPI code performance when executed with overlapped tiling on Xeon v2 architecture. On the left, we see results from the Laplace experiment. On the right, we see the results from the wave experiment.

5.3 Tuning Tile Parameters

When performing overlapped time tiling, there are tile parameters to consider:

- **Time Tile Height**. This is the temporal dimension of our time tile. This dictates how far our tile extends in the time dimension. This will also be the height of the wavefront tiles within the overlapped tile.
- **Wavefront Tile Widths**. These are the spatial dimensions of our wavefront tiles within the overlapped tile. Although our wavefront tiles are slanted in time, the width of the tile is constant for each timestep. Devito performs 2.5D tiling (see Section 2.5.1) and does not tile in the third z-dimension. Hence, we only consider the x and y dimensions of our wave tile. Note the width of the overlapped tile cannot be varied. The width of the tile is determined by the height of our time tile (see Section 4.2.1).

The time tile parameters chosen have a major impact on the execution times of the stencil. We begin by exploring the impact of time tile height before investigating the impact of wavefront base widths. Varying these dimensions to find the optimal execution time is known as **tuning**.

Recall that Devito also performs its own tiling. Devito applies spatial blocking with smaller tiles. We maintain this optimisation by tiling our wavefront tiles with these smaller spatial tiles (see Section 4.1). All experiments in this Chapter are executed with *DEVITO_AUTOTUNING* set to *aggressive*. Devito searches the entire parameter

space for the optimal x and y width of the tiles. This ensures tuning of our overlapped tiles is done in collaboration with Devito's auto-tuning. For each different combination of overlapped tile dimensions, Devito selects the optimal block size.

5.3.1 Tuning Tile Height

In Figure 5.6 we compare overlapped tiling execution times across different time tile heights. Note we also varied the wavefront base widths and selected the fastest experiment for each height. The results were obtained by executing the Laplace stencil on the Xeon v3 architecture. In this Section, we will not compare between architectures or stencils, as the relationship between tile height and Elapsed Time is the same across all experiments. This can be seen when observing the corresponding Xeon v2 Laplace and wave experiments in Figure 5.7.

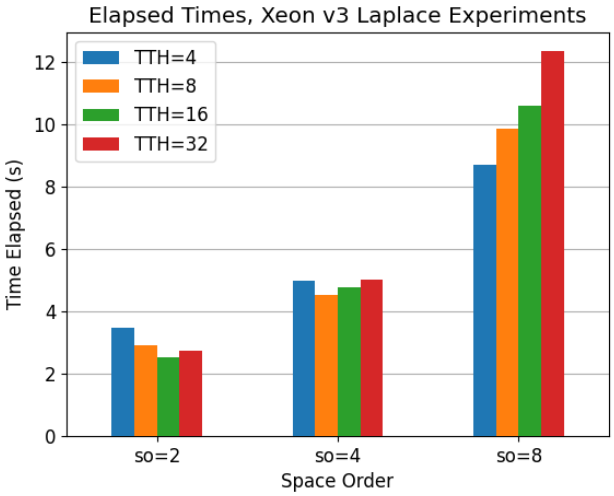


Figure 5.6: Comparison of overlapped tiling execution times across various time tile heights. We present results for three different space orders. TTH denotes time tile height.

We investigate time tile heights $T \in \{4, 8, 16, 32\}$ across all three space orders. For space orders 2 and 4, we see increasing the tile height improves performance up to a certain height. Beyond this, increasing the height will start to decrease performance. For space order 8, the smaller the tile height, the faster the execution.

Note this does not correspond to the results obtained from the same experiment on a single process. In Figure 5.8 we see a comparison of execution times for wavefront tiles of varying heights. For space orders 2 and 4, performance improves with time tile height. The larger wave tiles allow the stencil to take greater advantage of temporal blocking. For space order 8, there is not much variation. This is to be expected. We recall from Figure 5.2 that wavefront tiling does not improve execution times for a space order 8 Laplace stencil.

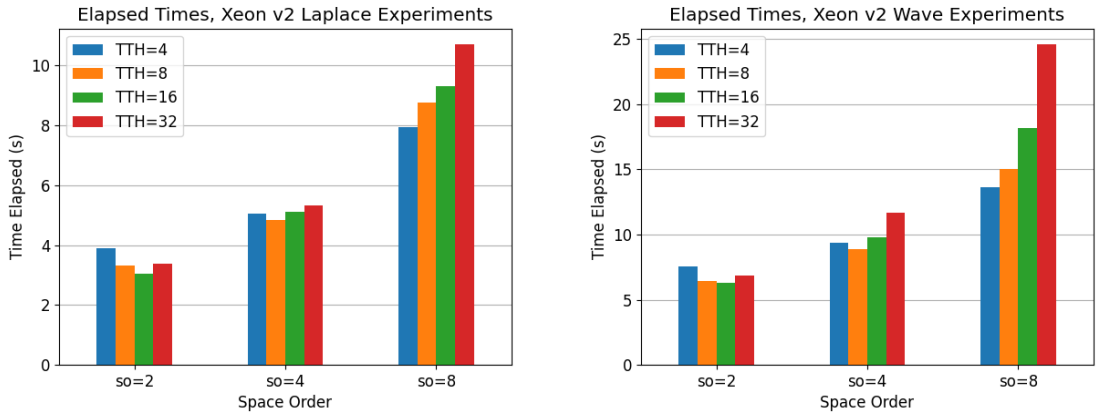


Figure 5.7: Comparison of overlapped tiling execution times across various time tile heights. TTH denotes time tile height. On the left, we see results from the Laplace experiment. On the right, we see the results from the wave experiment. We note these results have a similar distribution to the results displayed in Figure 5.6.

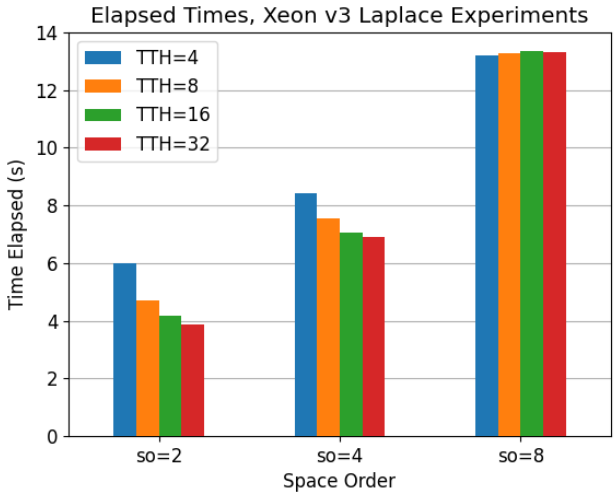


Figure 5.8: Comparison of wavefront tiling execution times across various time tile heights. We present results for three different space orders. TTH denotes time tile height.

The differences seen between Figure 5.6 and 5.8 can be explained by considering the negative effects of overlapped tiling. In Section 2.5.2 we explain the need to process extra grid points when performing overlapped tiling. Hence, we apply temporal blocking at the cost of redundant calculation. In Figure 5.6, we see the impact of these extra points. In some cases, improving cache reuse by increasing time tile height is outweighed by the negative impact of extra calculations.

5.3.2 Modelling Redundant Calculations

We now model the impact of time tile height on redundant calculations. In the following calculations, T will denote our time tile height, s will denote our space order and (t_M, x_M, y_M, z_M) denote the dimensions of the grid designated to our MPI process. In our experiments, we take $(t_M, x_M, y_M, z_M) = (256, 256, 512, 512)$. Note $x_M = 256$, despite our experiment grid having x-dimension 512. This is due to the division of the grid space which occurs when using MPI processes. We chose to use two ranks and Devito splits the grid in half in the x-dimension.

In Section 4.2.1 we demonstrated the number of extra points an overlapped tile must be allocated in each dimension is given by:

$$s(T-1) \tag{5.4}$$

For every increasing timestep, the number of points we are able to process decreases by s due to stencil dependencies. Hence, for each timestep $t \in \{0, 1, ..., T-1\}$, the number of extra points we must process is given by:

$$s(T-1) - st = \tag{5.5}$$
$$s((T-1) - t) \tag{5.6}$$

In 2.5D tiling, we tile spatially in the x and y dimensions. We then find the total number of points visited by the process in the overlapped tile in each timestep $t \in \{0, 1, ..., T-1\}$ is given by:

$$[x_M + s((T-1) - t)][y_M + s((T-1) - t)]z_m \tag{5.7}$$

We can expand Equation 5.7 to find a quadratic in t:

$$at^2 + bt + c \tag{5.8}$$

Here a, b and c are given as:

$$a = z_M s^2, \quad b = z_M s[-x_M - y_M - 2s(T-1)] \tag{5.9}$$
$$c = z_M[x_M y_M + s(T-1)(x_M + y_M) + s^2(T-1)^2] \tag{5.10}$$

By summing over $t \in \{0, 1, ..., T-1\}$, we find the total number of visited points in our overlapped tile is given by:

$$\sum_{t=0}^{T-1} at^2 + bt + c = \tag{5.11}$$
$$a\sum_{t=0}^{T-1} t^2 + b\sum_{t=0}^{T-1} t + \sum_{t=0}^{T-1} c = \tag{5.12}$$
$$\frac{a}{6}T(T-1)(2T-1) + \frac{b}{2}T(T-1) + cT \tag{5.13}$$

Then, to find the total number of points visited by our process, we multiply Equation 5.11 by the number of overlapped tiles used in the grid. This is $\frac{t_M}{T}$. Hence the total number of visited points is:

$$t_M \left[\frac{a}{6}(T-1)(2T-1) + \frac{b}{2}(T-1) + c \right] \tag{5.14}$$

The standard number of visited points by an MPI process is $t_M x_M y_M z_M$. Using Equation 5.14, we can calculate the ratio of the number of points visited by the MPI process when using overlapped tiling compared to standard MPI code. This is given by:

$$A(T-1)(2T-1) + B(T-1) + C \tag{5.15}$$

Here A, B and C are given as:

$$A = \frac{s^2}{6x_M y_M}, \quad B = \frac{s[-x_M - y_M - 2s(T-1)]}{2x_M y_M} \tag{5.16}$$

$$C = \frac{x_M y_M + s(T-1)(x_M + y_M) + s^2(T-1)^2}{x_M y_M} \tag{5.17}$$

Note the ratio given in Equation 5.15 does not depend on t_M or z_M. In Figure 5.9 we graph the % increase in the number of points visited by the process when performing overlapped tiling. This increases very quickly with T for a given s. For example, at $(s, T) = (8, 4)$, we observe an increase of 7%. At $(s, T) = (8, 32)$, the increase is 89%. This explains the notable increase in execution times for larger tile heights for space order 8 observed in Figure 5.6.

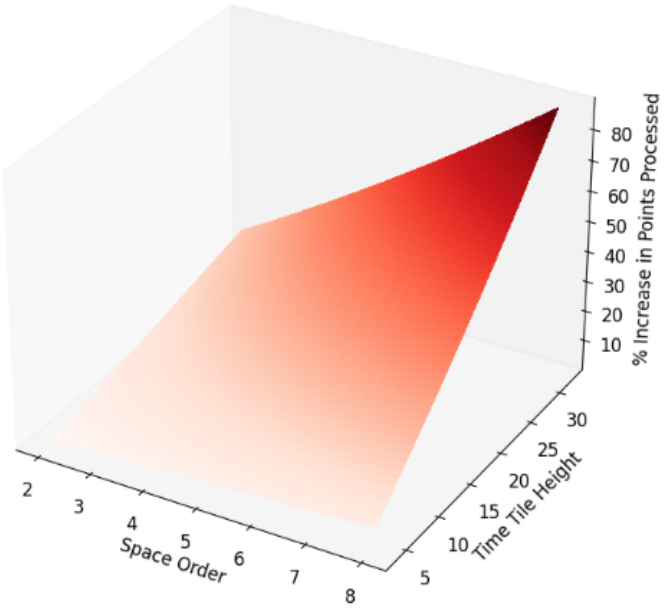


Figure 5.9: % Increase in the number of points processed when using overlapped tiling compared to standard Devito MPI code.

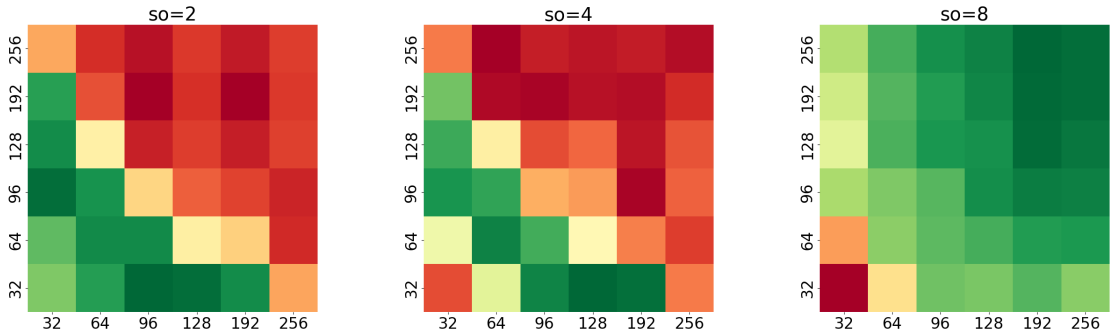


Figure 5.10: Impact of wavefront tile widths on Laplace stencil execution times on Xeon v2 architecture. Wavefront height is fixed to 16. Greener results correspond to faster execution times.

5.3.3 Tuning Wavefront Tile Widths

We now examine the impact of wavefront tile widths on overlapped tiling execution times. In both the x-dimension and the y-dimension, we consider wavefront widths with values from $\{32, 64, 96, 128, 192, 256\}$. Time tile height is fixed across each experiment. In all heatmaps, the tile's x-dimension is given in the x-axis and the y-dimension in the y-axis.

In Figure 5.10 we present visual results from the Laplace stencil executed on the Xeon v2 architecture. The results are summarised in Table 5.2. We note smaller wavefront tiles are preferred for the lower space orders. This is expected, as the smaller space orders have a narrower stencil. The results have a greater spread for space order 2 than for space orders 4 and 8. This is also expected, as space order 2 exhibits a greater benefit from temporal locality. Hence, there is more performance to be gained through optimisation.

Space Order	Fastest Time (s)	Slowest Time (s)	Percentage Difference (%)
so=2	3.06 @ (96, 32)	5.28 @ (96,192)	44
so=4	5.11 @ (128, 32)	6.25 @ (64, 256)	18
so=8	9.32 @ (192, 256)	11.52 @ (32, 32)	21

Table 5.2: Fastest and slowest times from tile width tuning experiment displayed in Figure 5.10. The percentage difference between the two times is displayed. With each time, we give the dimensions which attain such a value.

We can compare our results in Figure 5.10 to those in Figure 5.11. The only difference is the time tile height used. We use a height of 8 instead of 16. The heat maps are very similar. This is expected, as tile width is a spatial dimension of our tiles. Hence, the time dimension of our tile should not impact this experiment.

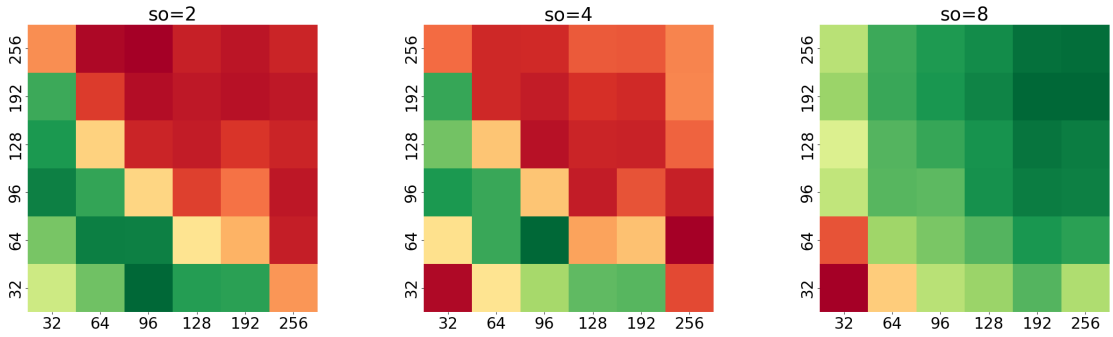


Figure 5.11: Impact of wavefront tile widths on Laplace stencil execution times on Xeon v2 architecture. Wavefront height is fixed to 8. Greener results correspond to faster execution times.

5.4 MPI Communication Times

When modifying Devito code to implement overlapped tiling, we noticed a reduction in the MPI communication times. We demonstrate below that the total amount of data communicated when using overlapped tiling is less. This accounts for the improved communication times. Note this is due to the behaviour of Devito and does not apply to overlapped tiling in general.

In the following calculations, s, T denote space order and time tile height respectively. When allocating data to MPI processes, Devito will use a halo region of size s on each side of the grid. In general, this size is too large. Most stencils of space order s will need to access only $\frac{s}{2}$ grid points on either side of the point being processed. This is the case for all stencils mentioned in this study. In most cases, a halo region of size $\frac{s}{2}$ should be sufficient. However, there are some cases where this extra space is necessary. For example, a PDE with the term u_{xx} may make full use of this extra halo region. Choosing to allocate a width of s is a design decision made by Devito, as there is no way of knowing from just the space order what PDE a user will define.

When implementing overlapped tiling, we choose to respect this decision. On either side of the grid, our overlapped tiles must be allocated an extra $\frac{s}{2}(T-1)$ points (see Section 4.2.1). Hence, the halo region has size:

$$s + \frac{s}{2}(T-1) \tag{5.18}$$

Note then for $T = 1$, we still allocate s points as done by Devito. We do not need to allocate an extra $s(T-1)$ points to respect the design decision made by Devito. The extra space allocated by Devito is to cater to points at the edge of the grid. For our overlapped tiles, these points at the edge of the grid still have the halo space of s.

The size of the halo directly impacts the size of the MPI transfer, as each process must send/receive the halo regions. The overlapped tile does this once every T timesteps.

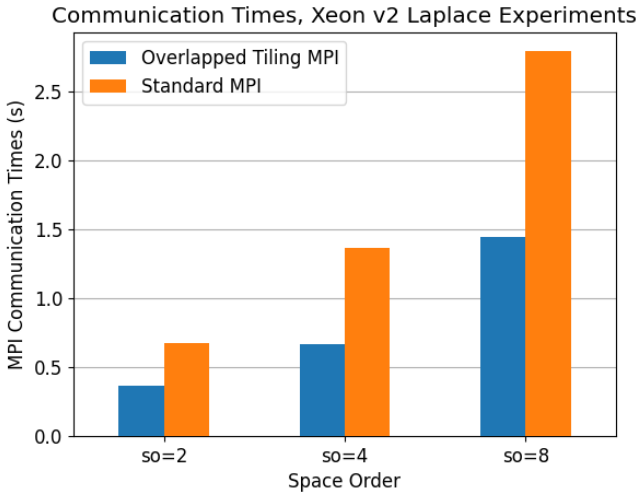


Figure 5.12: Comparison of MPI communication times between overlapped tiling and the standard Devito MPI implementation.

Hence the ratio between the amount of data transferred by the overlapped tiling implementation and the standard Devito implementation is given by:

$$\frac{s + \frac{s}{2}(T-1)}{sT} = \tag{5.19}$$

$$\frac{T+1}{2T} \approx \frac{1}{2} \tag{5.20}$$

From Equation 5.20, we therefore expect MPI communication times for overlapped tiling to be approximately half that of the standard Devito implementation. Figure 5.12 shows this relationship holds for the Xeon v2 Laplace experiments. Our ratios for space orders 2, 4 and 8 are 53%, 49% and 52%.

This explains the speed-up obtained from overlapped tiling for the Xeon v2 Laplace experiment in Figure 5.5. The difference in data transfer times for space order 8 is approximately 2 seconds. The overlapped tiling implementation executes approximately 2 seconds faster than the standard MPI code.

It is important to stress the lowered communication time does not account for all of the performance improvement from overlapped tiling. The use of wavefront tiling to implement temporal blocking is often still the main contributor to faster execution times. This can be seen in Figure 5.13. In these graphs, we consider computation times only and ignore the communication times. Any improvement in computation times may be attributed to the temporal blocking cache optimisation. On the left, we see the Xeon v2 wave experiments and on the right, the Xeon v3 Laplace experiments. We see for space orders 2 and 4 the impact of wavefront tiling. For space order 8, we do not see any benefit from wavefront tiling, as predicted in Section 5.2.1.

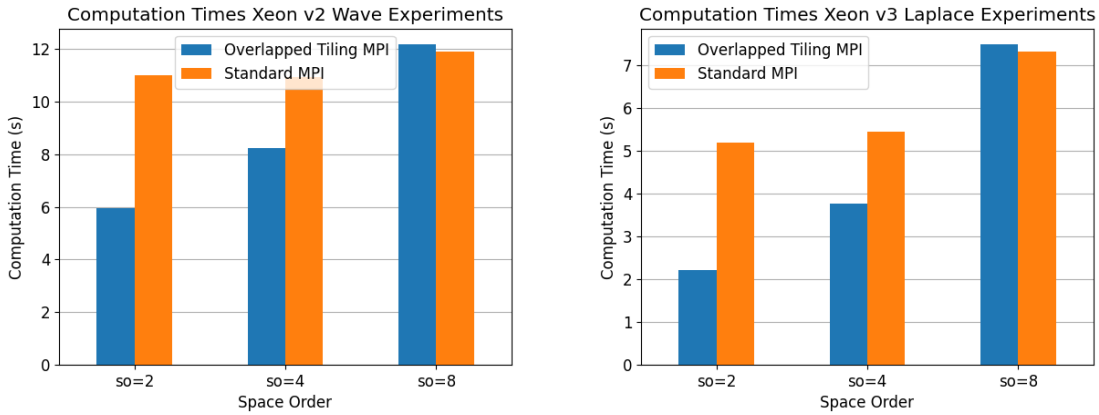


Figure 5.13: Improvement in computation times when using overlapped tiling compared to standard Devito code. On the left we see results from the Xeon v2 wave experiment. On the right, we see the Xeon v3 Laplace experiment.

5.5 Scalability

We conclude by evaluating the scalability of our overlapped tiling implementation by performing a **strong scaling** experiment. The problem size is kept constant (maintaining a grid size of $512 \times 512 \times 512$ and 256 timesteps), while the number of MPI processes used is increased. Recall we use the term *ranks* to denote MPI processes.

In our scaling experiment, we use the Laplace stencil of space order 2 and the 2nd Gen AMD EPYC 7742 CPU platform. This CPU is not referenced elsewhere in the Chapter and is not listed in Section 5.1.1. Its properties are summarised in Table 5.3.

AMD CPU Properties	
CPUs	128
Threads per core	2
Cores per socket	32
Sockets	2
CPU GHz	3.2
L1 cache	4MiB
L2 cache	32MiB
L3 cache	256MiB

Table 5.3: Comparison of CPU properties used in our experiments.

Figure 5.14 presents the results from our strong scaling experiment. We display a comparison of execution times when increasing the number of MPI ranks from 1 to 8. As can be seen, increasing the number of ranks decreases the execution time, as expected. However, the comparative performance gain decreases for the larger number of ranks. Doubling the number of ranks from 1 to 2 results in a speedup of 53%. Doubling from 4 to 8 gives a less notable speedup of 33%. This is not surpris-

ing either, as the MPI communication overhead increases as we increase the number of processes involved. These findings provide reassurance that our overlapped tiling implementation exhibits the expected scalability.

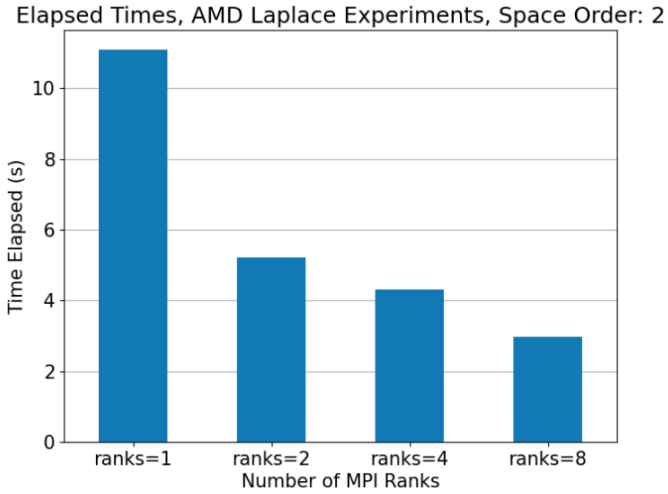


Figure 5.14: Impact of number of MPI ranks on the execution time of the Laplace space order 2 stencil. The experiment was performed on the AMD architecture.

5.6 Concluding Remarks

In this Chapter, we presented the performance improvements gained from overlapped tiling in comparison to the standard MPI code produced by Devito. In Figures 5.3 and 5.5, we presented our key results which demonstrated the speed-up in execution times. We reported a decrease in Elapsed Times of 55% and 28% for space orders 2 and 4 respectively. Then, in Section 5.3, we explored the impact of tuning our overlapped tile parameters and found the optimal tile parameters. In this Chapter, we also derived mathematical models for the positive impact of overlapped tiling on MPI communication times and the negative impact on redundant computations. These models were presented in Sections 5.3.2 and 5.4 respectively. We concluded our performance evaluation in Section 5.5, by demonstrating the scalability of our overlapped tiling implementation.

Chapter 6

Conclusions

In this study, we proposed overlapped tiling as a scheme to benefit from temporal locality in the distributed computation of solutions to partial differential equations. In Section 1.3 we outlined key contributions made to this end. We briefly revisit these contributions:

- **Wavefront Tiling Implementations**

  We outlined our wavefront implementation in Section 4.1. This was important in building up the final overlapped implementation with wavefront tiling. Our implementations also served as a useful reference throughout our evaluation in Chapter 5.

- **Overlapped Tiling Implementations**

  We outlined our overlapped implementation in Sections 4.2 and 4.3. We discussed the technical challenges faced with MPI ranks on the edge of our grid boundaries and provided an explanation for our choice of the new halo size.

- **Temporal Tiling Performance Evaluation**

  In Section 5.2 we evaluated our overlapped implementation. The key results were given in Figures 5.3 and 5.5. Here, we presented the decrease in Elapsed Time when performing distributed computation with overlapped tiling when compared to the standard Devito MPI code.

- **Temporal Tiling Performance Modelling**

  In Section 5.3.2 we derived a mathematical expression for the number of redundant points computed in our overlapped implementation. Then, in Section 5.4, we used the ratio of halo sizes to predict a 50% decrease in MPI communication times when using overlapped tiling in Devito. Both these models were used to explain empirical results presented in Chapter 5.

- **Temporal Tiling Parameter Tuning**

  In Section 5.3 we discussed the impacts of tuning the tile parameters used in overlapped tiling. The impacts of modifying overlapped tile heights and wavefront widths were both explored.

6.1 Future Work

The positive results in this study provide motivation for future work on overlapped tiling. There are many possible avenues for future work, both within the Devito framework and outside of it. Below we list some of these possibilities:

- **Asynchronous MPI calls**

  In Section 2.4.2 we outlined the MPI code generated by Devito. We described the different levels of MPI implementation. In this project, overlapped tiling was implemented for the standard MPI implementation. Devito allows the user to generate more complex MPI code, where MPI calls are asynchronous.

  Implementing overlapped tiling for this asynchronous version would be a useful optimisation. To illustrate this, we consider the Laplace results on the Xeon v3 architecture (see Chapter 5). For our space order 4 overlapped tiling implementation, the 0.58s communication time makes up 13% of the total execution time. An asynchronous implementation would provide a potential 13% speed-up.

- **Optimised halo region allocation**

  In Section 4.2.1 we discuss the increase in halo region required for overlapped tiling. We increased the halo region by a fixed size in the x and y-dimension. However, a large amount of this allocation is redundant. The overlapped tile has different widths for different timesteps and exhibits a slanted shape in the halo region. To optimise the halo region allocation, we can dynamically change the size of the halo region for different timesteps. This would decrease the time taken to allocate data to each rank and the amount of memory needed.

- **Modifications to the Devito compiler**

  We have implemented overlapped tiling on the Laplace and wave equations by modifying the code generated by Devito. By demonstrating its efficacy, we have provided motivation for a modification to the Devito compiler itself. Work on the Devito compiler would allow users to generate overlapped tiled code for a general PDE.

  Modifying the compiler to generate overlapped tiling would also pave the way for auto-tuning of tile parameters outlined in Section 5.3.3. Devito performs auto-tuning for loop tile parameters and would be able to do the same for the overlapped tile parameters.

- **Refined computation model**

  In Section 5.3.2, we model the increase in the number of points processed when using overlapped tiling when compared to standard MPI code. Extensions to the model could consider:

  – The latency of different cache levels and time taken for floating point operations. This would allow us to model the benefits of temporal locality and the exact slow-down from redundant computation.

  – The edge case of a process on the boundary of the grid (see Section 4.2). In the model we assume the process is not at the edge of our grid and overlapped tiling is performed in all directions. The model therefore provides an overestimate for these edge case processes.

    This is not relevant when considering distributed computation with many ranks, as we expect most of the ranks to apply standard overlapped tiling. Execution will take as long as the slowest rank, so we do not need to consider the edge case ranks. However, it is an important consideration when executing with a smaller number of ranks, where none of the ranks apply standard overlapped tiling.

Bibliography

[1] https://github.com/devitocodes/devito/tree/master/examples/seismic, Last accessed on 16/01/23. pages 7

[2] https://www.cgg.com/geoscience/subsurface-imaging, Last accessed on 16/01/23. pages 7

[3] https://github.com/devitocodes/devito, Last accessed on 16/01/23. pages 7

[4] https://www.openmp.org/, Last accessed on 02/06/23. pages 14

[5] https://www.open-mpi.org/, Last accessed on 02/06/23. pages 14

[6] https://mathworld.wolfram.com/EinsteinSummation.html, Last accessed on 19/06/23. pages 24

[7] https://mathworld.wolfram.com/L2-Norm.html, Last accessed on 13/06/23. pages 40

[8] Anton Afanasyev, Mauro Bianco, Lukas Mosimann, Carlos Osuna, Felix Thaler, Hannes Vogt, Oliver Fuhrer, Joost VandeVondele, and Thomas C. Schulthess. Gridtools: A framework for portable weather and climate applications. *SoftwareX*, 15:100707, 2021. pages 24

[9] Martin Sandve Alnæs, Anders Logg, Kristian B. Ølgaard, Marie E. Rognes, and Garth N. Wells. Unified form language: A domain-specific language for weak formulations of partial differential equations. *CoRR*, abs/1211.4047, 2012. pages 23

[10] A. Ammar, B. Mokdad, F. Chinesta, and R. Keunings. A new family of solvers for some classes of multidimensional partial differential equations encountered in kinetic theory modeling of complex fluids. *Journal of Non-Newtonian Fluid Mechanics*, 139(3):153–176, 2006. pages 4, 8

[11] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman 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. pages 26

[12] Peter Bauer, Alan Thorpe, and Gilbert Brunet. The quiet revolution of numerical weather prediction. *Nature*, 525(7567):47–55, September 2015. pages 5

[13] George Bisbas, Fabio Luporini, Mathias Louboutin, Rhodri Nelson, Gerard Gorman, and Paul H. J. Kelly. Temporal blocking of finite-difference stencil operators with sparse "off-the-grid" sources, 2020. pages 5, 26, 27

[14] Jan Blechschmidt and Oliver G. Ernst. Three ways to solve partial differential equations with neural networks — a review. *GAMM-Mitteilungen*, 44(2):e202100006, 2021. pages 5

[15] GEORGE W. BLUMAN and JULIAN D. COLE. The general similarity solution of the heat equation. *Journal of Mathematics and Mechanics*, 18(11):1025–1042, 1969. pages 11

[16] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. page 101–113, 2008. pages 25

[17] Zhiqiang Cai, Jingshuang Chen, Min Liu, and Xinyu Liu. Deep least-squares methods: An unsupervised learning-based numerical method for solving elliptic pdes. *Journal of Computational Physics*, 420:109707, 2020. pages 5

[18] Tony F Chan, Jianhong Shen, and Luminita Vese. Variational pde models in image processing. *Notices AMS*, 50(1):14–26, 2003. pages 4, 8

[19] Robert Eymard, Gallouet Thierry, and Raphaèle Herbin. Handbook of numerical analysis. 7:731–1018, 01 2000. pages 21

[20] Tobias Grosser, Albert Cohen, Justin Holewinski, P. Sadayappan, and Sven Verdoolaege. Hybrid hexagonal/classical tiling for gpus. page 66–75, 2014. pages 27

[21] Jia Guo, Ganesh Bikshandi, Basilio B. Fraguela, and David Padua. Writing productive stencil codes with overlapped tiling. *Concurrency and Computation: Practice and Experience*, 21(1):25–39, 2009. pages 25

[22] Tobias Gysi, Carlos Osuna, Oliver Fuhrer, Mauro Bianco, and Thomas C. Schulthess. Stella: a domain-specific tool for structured grid methods in weather and climate models. pages 1–12, 2015. pages 6, 24

[23] Justin Holewinski, Louis-Noël Pouchet, and P. Sadayappan. High-performance code generation for stencil computations on gpu architectures. page 311–320, 2012. pages 5, 28, 29

[24] Christian T. Jacobs, Satya P. Jammy, and Neil D. Sandham. Opensbli: A framework for the automated derivation and parallel execution of finite difference solvers on a range of computer architectures. *Journal of Computational Science*, 18:12–23, 2017. pages 24

[25] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. The organization of computations for uniform recurrence equations. *J. ACM*, 14(3):563–590, jul 1967. pages 28

[26] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P Sadayappan. Effective automatic parallelization of stencil computations. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, page 235–244, New York, NY, USA, 2007. Association for Computing Machinery. pages 29

[27] Christian Lengauer, Sven Apel, Matthias Bolten, Armin Größlinger, Frank Hannig, Harald Köstler, Ulrich Rüde, Jürgen Teich, Alexander Grebhahn, Stefan Kronawitter, Sebastian Kuckuk, Hannah Rittich, and Christian Schmitt. Exastencils: Advanced stencil-code engineering. pages 553–564, 2014. pages 24

[28] Randall LeVeque. *Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-Dependent Problems (Classics in Applied Mathematics Classics in Applied Mathemat)*. Society for Industrial and Applied Mathematics, USA, 2007. pages 9, 10

[29] Anders Logg, 2018. `https://github.com/hplgit/fenics-tutorial/blob/master/pub/python/vol1/ft01_poisson.py`, Last accessed on 11/01/23. pages 22

[30] Anders Logg, Garth Wells, and Kent-Andre Mardal. *Automated solution of differential equations by the finite element method. The FEniCS book*, volume 84. 04 2011. pages 6, 22

[31] Fabio Luporini, Mathias Louboutin, Michael Lange, Navjot Kukreja, Philipp Witte, Jan Hückelheim, Charles Yount, Paul H. J. Kelly, Felix J. Herrmann, and Gerard J. Gorman. Architecture and performance of devito, a system for automated stencil computation. *ACM Trans. Math. Softw.*, 46(1), apr 2020. pages 6, 13

[32] A. C. McKellar and E. G. Coffman. Organizing matrices and matrix operations for paged memory systems. *Commun. ACM*, 12(3):153–165, mar 1969. pages 5, 28

[33] B. Miñano and A. Arbona, 2019. `https://bitbucket.org/iac3/simflowny/wiki/history/Snaphots`, Last accessed on 12/01/23. pages 25

[34] Shaher Momani and Zaid Odibat. Analytical approach to linear fractional partial differential equations arising in fluid mechanics. *Physics Letters A*, 355(4):271–279, 2006. pages 4, 8

[35] J. Mourad. The finite element method (fem) – a beginner's guide, 2023. `https://www.jousefmurad.com/fem/the-finite-element-method-beginners-guide/`, Last accessed on 11/01/23. pages 22

[36] R. Naz, F.M. Mahomed, and D.P. Mason. Comparison of different approaches to conservation laws for some partial differential equations in fluid mechanics. *Applied Mathematics and Computation*, 205(1):212–230, 2008. Special Issue on Life System Modeling and Bio-Inspired Computing for LSMS 2007. pages 4, 8

[37] P. Nithiarasu O. C. Zienkiewicz, R. L. Taylor and J. Zhu. *The finite element method*, volume 3. McGraw-Hill London, 1977. pages 21

[38] Zaid Odibat and Shaher Momani. Numerical methods for nonlinear partial differential equations of fractional order. *Applied Mathematical Modelling*, 32(1):28–39, 2008. pages 5

[39] 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. pages 25

[40] Anthony T Patera. A spectral element method for fluid dynamics: Laminar flow in a channel expansion. *Journal of Computational Physics*, 54(3):468–488, 1984. pages 21

[41] A. Peirce. Solving the heat, laplace and wave equations using finite difference methods, 2018. `https://personal.math.ubc.ca/peirce/M257_316_2012_Lecture_8.pdf`, Last accessed on 03/01/23. pages 11

[42] Jean-François Pommaret. *Partial differential equations and group theory: new perspectives for applications*, volume 293. Springer Science & Business Media, 2013. pages 4

[43] Florian Rathgeber, David A. Ham, Lawrence Mitchell, Michael Lange, Fabio Luporini, Andrew T. T. Mcrae, Gheorghe-Teodor Bercea, Graham R. Markall, and Paul H. J. Kelly. Firedrake. *ACM Transactions on Mathematical Software*, 43(3):1–27, dec 2016. pages 6, 23

[44] István Z. Reguly and Gihan R. Mudalige, 2020. `https://github.com/OP-DSL/OPS/blob/develop/apps/c/poisson/poisson.cpp`, Last accessed on 12/01/23. pages 23

[45] István Z. Reguly, Gihan R. Mudalige, and Michael 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. pages 23

[46] Ryuichi Sai, John Mellor-Crummey, Xiaozhu Meng, Mauricio Araya-Polo, and Jie Meng. Accelerating high-order stencils on gpus. pages 86–108, 2020. pages 18

[47] Kristin R. Swanson, Carly Bridge, J.D. Murray, and Ellsworth C. Alvord. Virtual and real brain tumors: using mathematical modeling to quantify glioma growth and invasion. *Journal of the Neurological Sciences*, 216(1):1–10, 2003. pages 4, 8

[48] Yasuhiro Takei, Hasitha Waidyasooriya, Masanori Hariyama, and Michitaka Kameyama. Fpga-oriented design of an fdtd accelerator based on overlapped tiling. 07 2015. pages 28

[49] Mehdi Tatari and Mehdi Dehghan. A method for solving partial differential equations via radial basis functions: Application to the heat equation. *Engineering Analysis with Boundary Elements*, 34(3):206–212, 2010. pages 4

[50] Tristan van Leeuwen and Felix J. Herrmann. 3d frequency-domain seismic inversion with controlled sloppiness. *SIAM Journal on Scientific Computing*, 36(5):S192–S217, 2014. pages 8

[51] F.D. Witherden, A.M. Farrington, and P.E. 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. pages 23

[52] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. page 30–44, 1991. pages 5, 27, 28

[53] Jingling Xue. *Loop Tiling for Parallelism*. Kluwer Academic Publishers, USA, 2000. pages 5, 28

[54] Charles Yount and Alejandro Duran. Effective use of large high-bandwidth memory caches in hpc stencil computation via temporal wave-front tiling. pages 65–75, 2016. pages 5, 26

[55] Charles Yount and Alejandro Duran. Effective use of large high-bandwidth memory caches in hpc stencil computation via temporal wave-front tiling. pages 65–75, 2016. pages 19, 32

[56] Charles R. Yount, Josh Tobin, Alexander Breuer, and Alejandro Duran. Yask—yet another stencil kernel: A framework for hpc stencil code-generation and tuning. *2016 Sixth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*, pages 30–39, 2016. pages 26

[57] Jie Zhao and Albert Cohen. Flextended tiles: A flexible extension of overlapped tiles for polyhedral compilation. 16(4), dec 2019. pages 5, 28

[58] Xing Zhou, Jean-Pierre Giacalone, Robert Kuhn, María Garzarán, Yang Ni, and David Padua. Hierarchical overlapped tiling. *Proceedings - International Symposium on Code Generation and Optimization, CGO 2012*, 04 2012. pages 20, 29