

NAVJOT KUKREJA

HIGH-PERFORMANCE BACKPROPAGATION FOR
STRUCTURED-GRID SOLVERS

HIGH-PERFORMANCE BACKPROPAGATION FOR
STRUCTURED-GRID SOLVERS

NAVJOT KUKREJA

**Imperial College
London**

Submitted for the degree of Doctor of Philosophy and the Diploma of Imperial
College London

Department of Earth Science and Engineering
Imperial College London

January 2021 – version 1.1

Navjot Kukreja: *High-Performance Backpropagation for structured-grid solvers*, Submitted for the degree of Doctor of Philosophy and the Diploma of Imperial College London
Department of Earth Science and Engineering
Imperial College London, © January 2021

ABSTRACT

This doctoral project is about the solution of inverse problems on hyperbolic PDEs. It includes work on automation of code generation, trading off memory consumption with extra computation, as well as new insights on the use of lossy compression in this setting.

Domain-specific languages (DSLs) have emerged as a way to express programs at a high level of abstraction while enabling effective utilisation of modern hardware through specialised compilers. Partial Differential Equations (PDEs) are a standard language for expressing mathematical models in science and engineering, and Devito is a DSL designed to express programs that solve PDEs using the Finite Difference (FD) method on structured meshes.

Full Waveform Inversion (FWI) aims to estimate physical parameters of the earth by minimising the misfit between an observed signal and one simulated through an FD solution of the Wave PDE. This is solved using gradient-based methods, but the computation of derivatives introduces a high memory requirement. This is because the intermediate states that can be discarded for the forward problem are critical to calculating the derivatives, and storing these intermediate states requires large amounts of memory. A technique used here is checkpointing - where a subset of the states from the forward problem is stored, while the rest are discarded to save memory. The discarded states are recovered later by rerunning the forward computation from the last checkpointed state. A checkpointing library called pyRevolve was developed as part of this project. This library enables the use of checkpointing strategies with minimal changes in application code.

We present a strategy to use a combination of compression and recomputation to extend the memory-recomputation tradeoff into a three-way memory-recomputation-compression tradeoff. We first compare lossy and lossless compression for this purpose. Next, we propose a performance model that pyRevolve can use to inform its use of compression for a given problem. The thesis also includes an analysis of the effects of errors introduced during lossy checkpoint compression on the rest of the FWI problem.

The last piece of the puzzle is the automatic generation of fast, parallel derivative code corresponding to the objective function specified by the application programmer. We focus on the specific case of ex-

PLICIT FD solvers, which have a data-access pattern that is easy to run in parallel (shared or distributed memory). However, the derivatives generated by reverse mode AD do not preserve this parallelism. We show how derivatives of a finite-difference solver can be generated while preserving the parallelism of the original FD code.

DECLARATION

I hereby certify that the work presented in this dissertation is the result of my own investigations conducted either wholly by myself or partially, along with collaborators during the PhD project. Text and results obtained from other sources are referenced and properly acknowledged.

London, January 2021

Navjot Kukreja

COPYRIGHT NOTICE

The copyright of this thesis rests with the author. This work is made available under a Creative Commons Attribution Non-Commercial No Derivatives licence. Researchers are free to copy, distribute or transmit the thesis on the condition that they attribute it, that they do not use it for commercial purposes and that they do not alter, transform or build upon it. For any reuse or redistribution, researchers must make clear to others the licence terms of this work.

London, January 2021

Navjot Kukreja

This thesis is dedicated to the memory of my grandfather,
Hakim Singh Kukreja.

ACKNOWLEDGMENTS

I'll start by thanking my parents *Bani* and *Guchi Kukreja* who are responsible for bringing me to this point in life in too many ways to be able to identify in the small space I have here. They promoted a questioning nature from an early age - few parents would spend a significant portion of their income on a computer for a 10-year old (when they had no use for it), knowing full well that he might just end up using it for playing games all day ¹. Next up is my grandfather, *Hakim Singh* who has shown the entire family how a single life can affect so many. He instilled a questioning attitude not only in me but everyone he met. My uncle *Inderjit* who explained the concept of nuclear fusion and fission to a 10-year old ², and even stayed up nights to work on my middle-school science project. My sister *Nikita* who, even though she is so much younger than me, has taught me so much in life. My middle-school teacher *Monica ma'am* who noticed me writing C code on paper and wrote the note to my parents that got me the aforementioned computer. Also *Archana ma'am* who showed me at an early age that real education lies far beyond exams and marks, and kept pushing me to strive for more. Fast forward many years, *Aditya Jain* went far beyond what was required of him to put me on the path to this doctorate. *Valter de Senna*, who started off as my masters thesis advisor but ended up almost making me a part of his family, continues to be a mentor. *Vivek Krishnan* who, in his own weird way, was the biggest inspiration for this degree.

The biggest thanks of all go to *Gerard Gorman* who gave me the opportunity to do this in the first place. He has been the most supportive and inspiring supervisor/manager that I have had the pleasure of working with in my life. *Paul Kelly* has been the unwavering pillar of knowledge for this dissertation - everything seems so simple when he explains it. *Michael Lange*, whose ideas started the research that is this dissertation - even though I may not have seen it at the time. *Jan Hückelheim* has been the single largest contributor to this research³, both in ideas, the execution, and even my education during

¹ I believe I lived up to the expectation by only spending about half the time playing *Age of Empires*

² who went on to explain it to his entire class, forever to be labeled as a nerd after that

³ After myself, of course.

this doctorate and for that I am extremely grateful. *Mathias Louboutin* has been a (mostly) distant but ever-present contributor to this work and none of this work could have happened (starting from the conception of Devito) without him. *Fabio Luporini* took Devito from a summer project to this sophisticated machine that could enable all the follow-on research that we conducted through the last few years. *George Bisbas* taught me how to be a better student. *Philip Witte*, a saint and a scholar whose benevolence knows no bounds.

My time at Argonne was very significant in giving this research the direction that it ended up taking. I want to thank *Paul Hovland* for making that happen and for supporting me through my time there. *Oana Marin* helped me grow as a researcher even though she probably doesn't know it.

I would also like to thank my thesis examiners *Matthew Piggot* and *Guillaume Pallez* who put in a significant effort in helping me improve this thesis.

Outside of the research world, I would like to thank *Cristina* who supported this work like her own - she anthropomorphised this thesis as *Daisy*. Finally, I would like to thank *Fareed Ayaaz* and *Abu Mohammed* for introducing me to *Amir Khusrao* whose poetry has been a welcome distraction from this work.

CONTENTS

1	INTRODUCTION	29
1.1	Scientific Computing and Inverse Problems	29
1.2	Backpropagation	31
1.3	Motivation	32
1.4	Contributions	33
1.5	Thesis Outline	37
2	BACKGROUND	38
2.1	Seismic Imaging	38
2.1.1	Problem definition	38
2.2	Adjoint-state method	43
2.2.1	Gradient test	45
2.2.2	Automatic differentiation	45
2.3	Conclusion	46
3	DEVITO - A DOMAIN-SPECIFIC LANGUAGE FOR FINITE DIFFERENCE	47
3.1	Introduction	47
3.2	Background on compilers	48
3.3	Using Devito	50
3.4	Performance	51
3.5	Sparse-point interpolation	51
3.6	Correctness of the generated code	53
3.7	Implementing the Inversion kernels with Devito	53
3.8	Devito at scale	56
3.9	Conclusions	57
4	CHECKPOINTING	58
4.1	Introduction	58
4.2	Reducing the memory footprint	60
4.3	Optimal schedules	61
4.3.1	The <i>Revolve</i> algorithm	62
4.3.2	Extensions to <i>Revolve</i>	64
4.4	The <i>Revolve</i> software	65
4.5	Abstractions for Checkpointing	68
4.5.1	API, <i>pyRevolve</i> side	68
4.5.2	API, Application side	73
4.6	Verification	74
4.7	Implementation concerns	75

4.7.1	The extra checkpoint in the beginning	75
4.7.2	The extra forward step	76
4.7.3	Multi-level timestepping schemes	76
4.7.4	Zero-cost checkpointing	78
4.7.5	Operator-call overhead	84
4.7.6	Asynchronous checkpointing	85
4.7.7	“Optimal” number of checkpoints	85
4.8	Checkpointing beyond seismic	86
4.9	Future Work	87
4.10	Conclusions	88
5	COMPRESSION	89
5.1	Introduction	89
5.1.1	Adjoint-based optimisation	89
5.1.2	Techniques used to alleviate the high memory pressure	89
5.1.3	Summary of contributions	92
5.2	Recap of FWI	94
5.3	Compression algorithms	95
5.3.1	Lossless Compression	97
5.3.2	Lossy Compression	97
5.4	Other approximate methods	98
5.5	Checkpointing	98
5.6	Performance model including compression	100
5.7	Extending <i>pyRevolve</i> to support compression	102
5.8	Experimental setup	105
5.9	Results and Discussion	107
5.9.1	Evolution of compressibility	107
5.9.2	Lossless compression	109
5.9.3	Direct Lossy compression	109
5.9.4	Performance model validation	110
5.9.5	Forward propagation	110
5.9.6	Gradient computation	114
5.9.7	Stacking	116
5.9.8	Convergence	116
5.9.9	Subsampling	123
5.10	Conclusions and Future Work	125
6	AUTOMATIC DIFFERENTIATION OF FINITE-DIFFERENCE SOLVERS	128
6.1	Background	128
6.2	Related Work	131

6.3	Method and Implementation	132
6.3.1	Automatic Differentiation	132
6.3.2	1D Example	133
6.3.3	Multidimensional Stencils	136
6.3.4	Restrictions	140
6.3.5	Associativity of Floating-Point Summation	141
6.3.6	Verification	141
6.4	Test Cases	141
6.4.1	Wave Equation	142
6.4.2	Burgers Equation	145
6.5	Experimental Results	146
6.5.1	Broadwell	147
6.5.2	Knights Landing (KNL)	149
6.6	Implementation in Devito	152
6.7	Conclusion and Future Work	153
7	CONCLUSION	154
	BIBLIOGRAPHY	158

LIST OF FIGURES

- Figure 1 Visual depiction of Forward and Inverse problems. A forward problem uses a known/estimated set of parameters and a mathematical model to predict a set of observations. An inverse problem, on the other hand, uses the forward problem, as well as gradient information for the forward mathematical model to go from observations to a set of predicted parameters. 30
- Figure 2 Data flow graph of a gradient evaluation. $F(k)$ is the $(k + 1)$ th forward step. $R(k)$ is the $(k + 1)$ th reverse step. The arrows represent a data-flow dependence, i.e. F_1 uses information that F_0 computes, and so on. 31
- Figure 3 A seismic survey 39
- Figure 4 Data flow graph of the forward problem, i.e. given an $\mathbf{m}$, solving for $\mathbf{d}_{\text{sim}}(\mathbf{m})$. F_k represents the $(k + 1)$ th Forward timestep 42

- Figure 5 Roofline plot showing the performance of Devito-generated code on Broadwell. The kernel used was based on the isoacoustic wave equation, on a cubical domain of 512 points in each dimension. The optimisation level in Devito was set to *advanced*. The horizontal axis represents *Operational Intensity*, or the number of computations carried out per byte of data read. The four dotted vertical lines represent the four versions of the computational kernel tested for spatial discretisation orders of 4, 8, 12, and 16. The blue diamonds on these lines represent the actual measured performance. We have also included the absolute runtime as well as the percentage of peak. The solid black lines represent the maximum achievable performance (the peak) on that specific hardware - with the region under the angled line forming the *memory-bound* region and the region under the horizontal solid line forming the *compute-bound* region. 52
- Figure 6 Devito code required for a forward operator. This code completely implements Equation (12). This version of the code does not save the intermediate states. We shall refer to this as the *non-taped Forward Operator*. 54
- Figure 7 Devito code required for a forward operator that saves all intermediate states. The only difference from the code in Figure 6 is the `save` parameter to the `TimeFunction` constructor. We shall refer to this as the *taped Forward Operator*. 55
- Figure 8 Devito code required for an operator that calculates the adjoint and gradient in a single pass 57

- Figure 9 A visual representation of the data access pattern for the gradient computation where all intermediate results are kept in memory until they are required by the reverse pass. The horizontal axis is wallclock time and the vertical axis is simulation time/timesteps. The blue line represents forward computation, followed by a reverse computation shown in red. Each dot represents a saved checkpoint. A vertical line at any point shows the number of checkpoints stored in memory at that point in time. Memory consumption is at a peak at the end of the forward computation. Image source: [115] 59
- Figure 10 Checkpoint-recompute strategy where only a subset of intermediate results are stored. The other timesteps can be recomputed when required by rerunning the forward computation from the last available checkpoint. The sections where the red line is horizontal are the parts of the computation where the reverse computation waits for the forward computation to be repeated in order to recompute some data that was previously discarded. The straight-dotted red line shows the hypothetical path of the reverse computation had there been no recomputation required. 60
- Figure 11 Summary of extensions to *Revolve* that are relevant to the most common applications in Devito 65
- Figure 12 *Revolve* algorithm with calls to the application interface 66
- Figure 13 Packages overview. *Revolve* has been described in Section 4.3. The packages *pyRevolve* and *cRevolve* and how they are used to create a high-level abstraction of checkpointing are explained in Section 4.5. 69
- Figure 14 Crevolve classes. 71
- Figure 15 *pyRevolve* classes. The abstract classes *Checkpoint* and *Operator* are implemented by the client application 72

Figure 16	Devito code to utilize checkpointing based on <i>pyRevolve</i> 73	
Figure 17	Devito operators, and the implementation of a Checkpoint class. 74	
Figure 18	The non-simplified data flow graph of the forward computation showing that the data dependence is actually not just on the last time-step, but on the last two. 76	
Figure 19	Time-to-solution when running the same problem with <i>pyRevolve</i> with different number of checkpoints - with time spent on different sub-parts of the process. The problem was a single forward-reverse pass on an FWI test case on the Overthrust velocity model[4]. The blues represent storage overheads - <i>Revolve</i> assumes these to be zero. 79	
Figure 20	Same experiment as Figure 19, but now with a more efficient memcpy implementation and reduced data movement (described in Section 4.7.3) 80	
Figure 21	The existing (and most common) approach to checkpointing in Devito/ <i>pyRevolve</i> . Checkpoints are stored and retrieved with deep copies. 81	
Figure 22	The suggested approach where the Operators operate directly on checkpointing memory 82	
Figure 23	Comparison of execution times for the two strategies. The existing strategy is called “explicit” here and the suggested one is called “implicit”. For each experiment, the number of checkpoints was the same as the number of timesteps, to bring out the worst case since the data movement is worst when the number of checkpoints is equal to the number of timesteps. 83	
Figure 24	Example Devito code to create a symbol representing externally allocated memory 83	
Figure 25	Timeline of events for conventional adjoint, <i>Revolve</i> checkpointing, and asynchronous multi-stage checkpointing. 85	

- Figure 26 An illustration of the approach presented in this chapter. Checkpoints are compressed using lossy compression to combine lossy compression and the checkpoint-recompute strategies. [91](#)
- Figure 27 The speedups predicted by the performance model for varying memory. The baseline (1.0) is the performance of a *Revolve*-only implementation under the same conditions. The different curves represent kernels with differing compute times (represented here as multiples of T , where T is the time taken to compress and decompress a single timestep). The original isoacoustic compute kernel took about half the time to compute as compared to compressing/decompressing a single timestep ($0.56T$). The next two are hypothetical kernels chosen to be slightly slower and slightly slower than the original kernel. The first vertical line at 55GB marks the spot where the compressed wavefield can completely fit in memory and *Revolve* is unnecessary if using compression. The second vertical line at 2.2 TB marks the spot where the entire uncompressed wavefield can fit in memory and neither *Revolve* nor compression is necessary. The region to the right is where these optimisations are not necessary or relevant. The middle region has been the subject of past studies using compression in adjoint problems. The region to the left is the focus of this chapter. [103](#)
- Figure 28 The speedups predicted by the performance model for varying compute cost. The baseline (1.0) is the performance of a *Revolve*-only implementation under the same conditions. The benefits of compression drop rapidly if the computational cost of the kernel that generated the data is much lower than the cost of compressing the data. For increasing computational costs, the benefits are bounded. [104](#)

- Figure 29 The speedups predicted by the performance model for varying number of timesteps. It can be seen that compression becomes more beneficial as the number of timesteps is increased. 104
- Figure 30 Evolution of compressibility through the simulation. We compressed every time step of the reference problem using an absolute error tolerance (*atol*) setting of 10^{-4} . The Compression factor achieved is plotted here as a function of the timestep number. Higher is more compression. The dashed horizontal line represents *no compression*. We can see that the first few timesteps are compressible to 1000x - since they are mostly zeros. The achievable compression factor drops as the wave propagates through the domain and seems to stabilize to 20x towards the end. We pick the last time step as the reference field for further experiments. 108
- Figure 31 Reference wavefield for compression and decompression. This field was formed after a Ricker wavelet source was placed at the surface of the model and the wave propagated for 2500 timesteps. This is a vertical (x-z) cross-section of a 3D field, taken at the y source location. 108
- Figure 32 Direct lossy compression: Errors introduced during compression and decompression using the fixed-tolerance mode. It is interesting to note that the errors are more or less evenly distributed across the domain with only slight variations corresponding to the wave amplitude (from Figure 31). A small block-like structure characteristic of ZFP can be seen. *atol* was set to 10^{-3} - the highlighted point in Figure 33 109

- Figure 33 Direct lossy compression: Effect of tolerance settings of ZFP in fixed-tolerance mode on Compression ratio. We define compression factor as the ratio between the size of the uncompressed data and the compressed data. The dashed line represents no compression. The highlighted point is $atol = 10^{-3}$ and corresponds to the setting used for Figure 32 110
- Figure 34 Direct lossy compression: We compress the wave field at the last time step of the reference solution using different $atol$ settings and report the Peak Signal to noise ratio (PSNR) achieved. Higher PSNR is lower error. The PSNR is very high for low absolute error tolerance ($atol$) and drops predictably as $atol$ is increased. See Figures 35 and 36 for more metrics on this comparison. 111
- Figure 35 Direct lossy compression: L_∞ norm of error versus $atol$. This plot verifies that ZFP respects the tolerance we set. 111
- Figure 36 Direct lossy compression: L_1 norm of error versus $atol$. From the difference in magnitude between the L_∞ plot and this one, we can see how the error is spread across the domain. 112
- Figure 37 Predicted vs measured runtimes for the OT2 kernel, without compression. 112
- Figure 38 Predicted vs measured runtimes for the OT2 kernel, with compression enabled. The compression setting used was ZFP with $atol = 10^{-6}$ 113
- Figure 39 Predicted vs measured runtimes for the OT4 kernel, with compression enabled. The compression setting used was ZFP with $atol = 10^{-6}$ 113

- Figure 40 Forward propagation: We run the simulation for 500 timesteps. We then compress the state of the wavefield at this point using $atol = 10^{-4}$. We then continue the simulation from the lossy checkpoint and compare with the reference version. Here we report the L_∞ and L_2 norms of the error between the wavefields of these two versions as a function of the number of timesteps evolved from this lossy checkpoint. L_∞ initially grows sharply, but plateaus after about 150 timesteps. However, the growing L_2 norm tells us that the average error is going up possibly because the error is spreading around the domain without increasing in magnitude. 115
- Figure 41 Gradient computation: In this experiment, we carry out the full forward-reverse computation to get a gradient for a single shot, while compressing the checkpoints at different $atol$ settings. This plot shows the PSNR of true vs lossy gradient as a function of $atol$ on the lossy checkpoints. We can see that the PSNR remains unchanged until about $atol = 10^{-6}$ and is very high even at very high values of $atol$. 116
- Figure 42 Gradient computation: L_∞ and L_2 norms of gradient error versus $atol$. It can be seen that the error stays almost constant and very low up to a threshold value of 10^{-4} 117

- Figure 43 Gradient computation: Angle between the lossy gradient vector and the reference gradient vector (in radians) vs $atol$ and compression factor. If the lossy gradient vector was pointing in a significantly different direction as compared to the reference gradient, we could expect to see that on this plot. However, the angles seem too small to make a difference. The number of timesteps does not seem to affect the result by much. The results are also resilient to increasing $atol$ up to 10^{-2} . Compression factors of over 100x do not seem to significantly distort the results either. 118
- Figure 44 Gradient error: L_∞ and L_2 norms of the gradient error as a function of the achieved compression factor (CF). It can be seen that errors are negligible in the range of CF up to 16. Compare this to subsampling in Figure 51. Note that we achieved much higher CF values as part of the experiment but cut the axis in this figure to make it comparable to Figure 51 119
- Figure 45 Gradient linearisation: Comparison of gradient linearisation errors for two compression settings vs reference solution. H is a perturbation in the velocity model, and we measure the error induced as a function of H . The two curves in each of the plots follow each other so closely that they are indistinguishable. This confirms that the lossy gradients satisfy the Taylor-linearisation properties just as well as the reference gradient. Plots for smaller values of $atol$ look the same so we omit them here for brevity. 120
- Figure 46 Gradient error: In this plot we measure the effect of varying number of checkpoints on the error in the gradient. We report PSNR of lossy vs reference gradient as a function of number of checkpoints, for four different compression settings. 121

- Figure 47 Shot stacking: The gradient is first computed for each shot and then added up for all the shots. In this experiment, we measure the propagation of errors through this step. This plot shows that while errors do have the potential to accumulate through the step - as can be seen from the curve for $\text{atol} = 10^{-1}$, for compression settings that are useful otherwise, the errors do not accumulate significantly. 121
- Figure 48 Reference solutions for the complete FWI problem. Above is the true solution and below is the solution after running *reference* FWI for 30 iterations 122
- Figure 49 Convergence: As the last experiment, we run complete FWI to convergence (up to max 30 iterations). Here we show the convergence profiles of two lossy settings vs the reference problem. The reference curves are so closely followed by the lossy curve that the reference curves are hidden behind. 124
- Figure 50 Final images after running FWI for two lossy settings. The two are visually indistinguishable from each other and from the reference solution in Figure 48. 125
- Figure 51 Subsampling: We set up an experiment with subsampling as a baseline for comparison. Subsampling is when the gradient computation is carried at a lower timestepping than the simulation itself. This requires less data to be carried over from the forward to the reverse computation at the cost of solution accuracy so is comparable to lossy checkpoint compression. This plot shows L_∞ and L_2 norms of gradient error versus the compression factor CF for this experiment. Compare this to the errors in Figure 44 that apply for lossy checkpoint compression. 126

- Figure 52 Stencils with gather access occur in convolutions and many other applications and are easy to parallelise. Automatic differentiation or backpropagation transforms this operation into a scatter update, which is hard to parallelise without race conditions. With adjoint stencils, this scatter update can be rearranged back into a gather/stencil and backpropagation can be performed by using only easily parallelisable stencils with gather access. 130
- Figure 53 Adjoint stencil transformation step by step, for a one-dimensional three-point stencil. This illustration does not show the read accesses to the primal array u that may still be needed in the derivative code. Red arrows in the primal and adjoint show a group of updates that is performed in the same loop iteration. Those same updates are then highlighted in the split, shifted, and core loop. After regeneration, the red updates are distributed among several iterations, and one iteration in the core loop performs a set of updates as shown by green arrows. 134
- Figure 54 Iteration spaces of adjoint stencil loop nests for a five-point stencil in two dimensions (that is, reading from $[i-1, j]$, $[i+1, j]$, $[i, j]$, $[i, j-1]$, $[i, j+1]$). Each of the large squares represents one point in the iteration space. Areas with uniform colour are part of the same loop nest. The core loop is represented by the dark area in the centre of the image. Within each loop nest, a stencil gather operation is implemented. The read indices are a subset of the original five-point star stencil. The subset that is actually implemented in that particular loop nest is shown by filled red boxes; the expressions that are not valid in this area are shown with empty red boxes. 139

- Figure 55 Python script that uses the PerforAD package to generate C code, which implements a three-dimensional wave equation solver. The input and output arrays are represented in PerforAD using SymPy function objects. All scalars, including loop counters and bounds, are represented by SymPy symbol objects. The SymPy expression object `expr`, a list of loop counters, and a dictionary with bounds associated with each counter are passed to the `makeLoopNest()` function, which is part of PerforAD. It returns a `LoopNest` object, which encapsulates all necessary data to represent a stencil computation, and provides functions to generate the primal and adjoint C code. 143
- Figure 56 One time step for the primal and adjoint wave equation solver implemented in C, as generated by PerforAD. The output array is written to at the central index i, j, k , while the input arrays are read in the neighbourhood around that centre. The loop is parallelised. In contrast, the conventional adjoint implements a scatter operation to a neighbourhood around the centre and is manually parallelised by using atomic updates. 144
- Figure 57 Python script to generate the Burgers equation solver. An upwinding scheme is implemented by using `min` and `max` functions, which are piecewise differentiable and correctly handled by SymPy and PerforAD. 145
- Figure 58 Primal and adjoint stencil solvers for the Burgers equation in one dimension. The upwinding scheme results in the use of ternary operators in the adjoint code. 146
- Figure 59 Speedups for the wave equation solver on a Broadwell processor, using up to 12 threads. The Tapenade-generated code with manual parallelisation does not scale at all. The primal and PerforAD-generated adjoint benefit from using all 12 cores. 147

- Figure 60 Speedups for the Burgers equation solver. The PerforAD-generated adjoint has near-perfect scalability. 148
- Figure 61 Absolute runtimes for wave equation primal and adjoint stencils and conventional adjoints in serial, as well as best observed primal and adjoint stencil run time in parallel. The best-observed performance of adjoint stencils was with 12 threads and is faster than the conventional adjoint by a factor of $3.4\times$. 148
- Figure 62 Absolute runtimes for the Burgers equation primal and adjoint stencils and conventional adjoints in serial, as well as best observed primal and adjoint stencil runtime in parallel. Despite being slower in serial, the adjoint stencil outperforms conventional adjoints by a factor of $5.7\times$. 149
- Figure 63 Speedups for the wave equation solver on KNL, using up to 64 threads. The primal loop scales well only up to 16 threads. The adjoint stencil loop generated by PerforAD continues to scale up to 32 threads. The manually parallelised adjoint solver scales only up to 2 threads. 150
- Figure 64 Speedups for the Burgers equation solver, showing near-perfect scalability up to 64 threads for the primal and adjoint stencil solver on a KNL processor. The scatter adjoints with atomics do not scale at all. 150
- Figure 65 Absolute runtimes for wave equation solvers. The fastest primal stencil used 128 threads, while the fastest adjoint stencil used 256 threads. Since the conventional adjoint code does not scale well, adjoint stencils lead to a much-reduced runtime in parallel, over $19\times$ faster than the best runtime of the conventional adjoint code. 151

Figure 66 Absolute runtimes for the Burgers equation solvers. The adjoint stencil loop is faster than the conventional adjoint code generated by Tapenade. The latter is slowed by the access to the intermediate value stack that is used to store the results of the min and max functions. This, combined with the differences in scalability, results in a $125\times$ run time difference between Tapenade and PerforAD. 151

LIST OF TABLES

Table 1	Estimated computational requirements of a Full-Waveform Inversion problem based on the SEAM model [30]. Note that real-world FWI problems are likely to be larger. ¹ A gradient computation involves a forward simulation followed by a reverse/adjoint computation. For simplicity we assume the same size of computation during the forward/adjoint pass. 59
Table 2	Some results from experimenting with all possible compressors and settings in blosc. We selected the best compression ratio seen for each compressor. "Setting" here is the choice between speed and compression, where 0 is fastest and 9 is highest compression. Blosc also provided a setting called <i>Shuffle mode</i> . We noticed our results were always better with this setting enabled than disabled. 109

LIST OF CODE REFERENCES

1. Devito: <https://github.com/devitocodes/devito> This is the main project that provides the context for the whole thesis.
2. PyRevolve: <https://github.com/devitocodes/pyrevolve/> This is the library corresponding to Chapter 4.
3. DAKS: <https://github.com/devitocodes/daks> This is the framework used to run experiments from Chapter 5 on the Azure cloud infrastructure.
4. PerforAD: <https://github.com/jhueckelheim/PerforAD> This is the library corresponding to Chapter 6.
5. Revolve: <https://www.mathematik.hu-berlin.de/de/forschung/forschungsgebiete/mathematische-optimierung/andrea-walther/walther-software> Referenced in Chapter 4. This is the original Revolve algorithm upon which pyRevolve (above) is based.
6. LibAdjoint: <https://bitbucket.org/dolphin-adjoint/libadjoint> Referenced in Chapter 4. This is another checkpointing library from which we use the C wrapper of the above C++ code.
7. In place operators: <https://github.com/devitocodes/navjotk/tree/allocator3> Prototype implementation of a Devito feature discussed in Section 4.7.4.1.
8. PyZFP: <https://github.com/navjotk/pyzfp> Python wrapper for ZFP developed for the experiments in Chapter 5.
9. PySZ: <https://github.com/navjotk/pysz> Python wrapper for SZ developed for the experiments in Chapter 5.

INTRODUCTION

Computational scientists solve tomorrow's problems with yesterday's computers; computer scientists seem to do it the other way around.

— Anonymous [69]

1.1 SCIENTIFIC COMPUTING AND INVERSE PROBLEMS

Scientific computing, also known as computational science and engineering, can be defined as the solution of problems seen in the sciences and engineering, on a computer. It can be argued that scientific computing was one of the earliest drivers of computer science and technology, and continues to play that role.

The application areas are fairly diverse, ranging from finance, to biology and, of course, physics, but the problems being solved usually represent some physical system. Although some definitions of the term *scientific computing* can include problems that can be solved exactly, often the system of interest is so complex that we can only deal with approximations (models). The definition of interest in this thesis only includes the latter class of problems, for which solving the exact system is either practically or really impossible. For these problems where one can only hope to solve an approximate model, a slightly better model that consumes slightly more computational resources can always be constructed. Hence solving these problems is a trade-off between model accuracy/fidelity and computational resources. For this reason, any code implementing a solver for this class of scientific computing problems must maximise its use of the available computational resources. Even as the computational resources available to us increase with time, the same problem can be solved to a higher accuracy – using all available resources, at least in theory.

Common examples of such problems include gravitational wave physics, weather and climate models, and seismic imaging. Although the work here is intended to be general to most problems in this class of scientific computing workloads, we shall use seismic imaging as the representative example for the class of scientific computing.

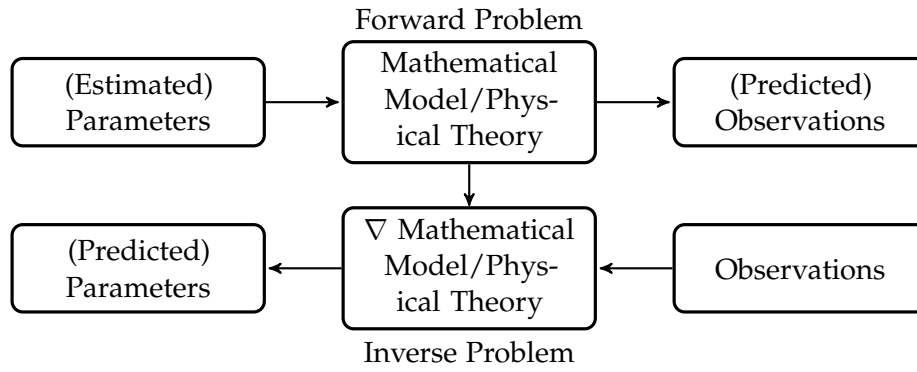


Figure 1: Visual depiction of Forward and Inverse problems. A forward problem uses a known/estimated set of parameters and a mathematical model to predict a set of observations. An inverse problem, on the other hand, uses the forward problem, as well as gradient information for the forward mathematical model to go from observations to a set of predicted parameters.

Most scientific computing workloads can be broadly classified into *Forward problems* and *Inverse problems* (see Figure 1). In a physical system a set of parameters define the behaviour of the system, along with a set of governing equations. Solving these equations with the parameters in place leads to a set of predicted observations. This is a forward problem. An example of this in wave propagation physics would be asking the question: *Given a source signal that was sent out, and given the medium's (spatially varying) wave propagation speed, calculate the signal that would be received at a specific point in the domain.* However, it is often the case that the observations are simpler to measure directly than the physical parameters of the system, so this system may be solved in reverse - to answer the question of *What set of parameters would lead to these observations?*. These are called *inverse problems*. In wave propagation physics, this would be akin to asking the question: *Given the source signal that was sent out, and given the signal that was received at specific points in the domain, what is the (spatially varying) wave propagation speed of the earth?*

Such a problem would be solved by setting up an *adjoint-based optimisation problem*. This problem would have an objective function to be minimised (or maximised), finding the set of parameters that achieve this minimisation. It would be solved using a gradient-based method where the adjoint helps in calculating the gradient (see Section 2.2). In the next section, we discuss how that gradient is calculated.

1.2 BACKPROPAGATION

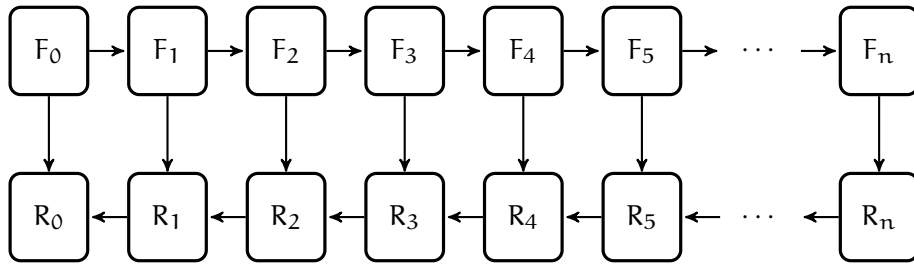


Figure 2: Data flow graph of a gradient evaluation. $F(k)$ is the $(k+1)$ th forward step. $R(k)$ is the $(k+1)$ th reverse step. The arrows represent a data-flow dependence, i.e. F_1 uses information that F_0 computes, and so on.

Backpropagation is a term used to describe a pattern of computation where an original computation (the *forward computation*) is followed by a similar computation where the direction of data flow is reversed - hence *back* propagation (see Figure 2). This is done to calculate derivatives of the forward problem.

This term is most commonly used in the field of machine learning. For the machine learning community, the history of this term can be traced back to Rumelhart, Hinton, and Williams [100]. However the term is practically synonymous with *reverse-mode automatic differentiation*, which is used in the automatic differentiation community, and the *adjoint-based method*, which is used in the numerical computing community. As a concept, its history can be traced back to Linnainmaa [76]. The field of machine learning has gone from being a niche to household terminology in the last decade. During this growth in leaps and bounds, the field has discovered a vast amount of new knowledge. However, in this rapid growth, the field has also reinvented many concepts that already existed elsewhere. Backpropagation is one example of this, while Checkpointing (Chapter 4) is another.

In this thesis, we use the term *backpropagation* to highlight the fact that backpropagation, reverse-mode automatic differentiation and the adjoint-based method are the same thing. This also raises hopes for an easier crossover of ideas between these three subfields more easily in the future. When solving an inverse problem (or training a machine learning algorithm), it is essential to calculate gradients - *fast!* This is the focus of this thesis.

1.3 MOTIVATION

As discussed in Section 1.1, it is essential for code written for scientific computing workloads to maximally utilise the computational resources available. This is true to such an extent that the terms scientific computing and high-performance computing are often used interchangeably.

As Moore’s law comes to an end, processor clock speeds have flattened out. The increase in computational power since then has come from increasing parallelism and increasingly complex computer architectures designed to expose this parallelism.

This increasing complexity of computer architectures means that code written to maximally utilise the resources available also needs to get increasingly complex. It is no longer possible to just (recompile and) rerun the same code on a new processor to have it run faster *for free*¹.

The increasing complexity of computer code means that performance-sensitive programming can be seen as a trade-off between programmer productivity and code performance. Code that is quick to write is usually slow to run, and code that is quick to run usually requires significant programming effort.

Domain-specific languages promise the best of both these worlds – code that is quick to write, and also quick to run. The work in this thesis was done as part of one such domain-specific language - *Devito*. Working inside a domain-specific language has its advantages and disadvantages - while program analysis and high-level abstractions can make information available that would not otherwise be available, the need for the implementations to make as few assumptions as possible about the problem presents some unique challenges.

Devito’s performance in the *forward problem*, while absolutely essential to solving an inverse problem efficiently, is not of concern in this thesis. In this work, we focus on evaluating gradients efficiently - to solve the inverse problem as quickly as possible.

Taking another look at Figure 2, it can be seen that each step of the gradient evaluation requires data from the corresponding forward step. This implies that the intermediate results from the complete forward computation need to be stored. This can increase the memory requirement of the gradient computation to tens of terabytes quite

¹ Despite reassurances to the contrary from chip manufacturers

easily. This high memory requirement is often the primary bottleneck of the gradient computation.

The first element required to calculate gradients efficiently is a formulation of the gradient itself. While it is possible to ask the user of Devito to write down the mathematical expression(s) for the gradient, this would require the user to hand-derive gradients corresponding to the objective function and the forward model equations. This would seriously restrict the flexibility of the user, and even the usability of Devito. Therefore the first requirement for evaluating gradients efficiently is to derive the formulation automatically. This is what we call *automatic differentiation* in this thesis. Not only does automatic differentiation improve user productivity, it can also lead to faster code performance.

As mentioned in the previous section, gradient computations have an extremely high memory footprint – which is the primary bottleneck. In this thesis, we discuss a class of techniques that reduce the memory footprint of these computations by a combination of storing certain intermediates in memory while recomputing others. We call this *Checkpointing*.

Checkpointing aims to reduce the memory footprint while maintaining numerical fidelity with respect to the original computation, but it might be possible to reduce the memory requirement further while reducing the precision of the final result. We explore this by combining lossy compression with checkpointing into a combined strategy that we call *Lossy checkpoint compression*.

1.4 CONTRIBUTIONS

- *pyRevolve*– A software library that can easily integrate checkpointing and compression into any application (here we use Devito as an example)
- We propose the combination of lossy compression and checkpointing
- Implementation and experiments of lossy checkpoint compression, concluding that the problem can tolerate large amounts of lossy compression with little impact on the solution.
- Automatic differentiation of pure stencil loops - the scatter-gather transform

- Implementation and experiments of the above, concluding that this rearrangement technique scales much better than other methods.

In addition, this project has led to the following publications, which form the basis of this thesis:

- [1] This paper was conceived and designed in discussions between Jan Hückelheim and myself. I carried out the seismic experiments in this paper. This paper is the basis for Chapter 6.
- [1] Jan Hückelheim, Navjot Kukreja, Sri Hari Krishna Narayanan, Fabio Luporini, Gerard Gorman, and Paul Hovland. “Automatic differentiation for adjoint stencil loops.” In: *Proceedings of the 48th International Conference on Parallel Processing*. 2019, pp. 1–10.
- [2] The design of the experiments that would verify parts of Devito was done jointly between Jan Hückelheim and me. The experiments were coded up and run mostly by me. Interestingly, we discovered bugs in Devito’s implementation of tiling during the course of this work - Fabio worked to fix those bugs.
- [2] Jan Hückelheim, Ziqing Luo, Fabio Luporini, Navjot Kukreja, Michael Lange, Gerard Gorman, Stephen Siegel, Matthew Dwyer, and Paul Hovland. “Towards self-verification in finite difference code generation.” In: *Proceedings of the First International Workshop on Software Correctness for HPC Applications*. ACM. 2017, pp. 42–49.
- [3] The code and text of this paper were written mostly by me, with many inputs from Jan Hückelheim. This paper forms the basis for Chapter 4.
- [3] Navjot Kukreja, Jan Hückelheim, Michael Lange, Mathias Louboutin, Andrea Walther, Simon W Funke, and Gerard Gorman. “High-level python abstractions for optimal checkpointing in inversion problems.” In: *arXiv preprint arXiv:1802.02474* (2018).
- [4] This paper was conceived and designed between Jan Hückelheim and myself. The experiments were coded up and run by me. I also wrote most of the text. Chapter 5 is partially based on this.

- [4] Navjot Kukreja, Jan Hückelheim, Mathias Louboutin, Paul Hovland, and Gerard Gorman. “Combining checkpointing and data compression to accelerate adjoint-based optimization problems.” In: *European Conference on Parallel Processing*. Springer. 2019, pp. 87–100.
- [5] This paper was conceived and designed between Jan Hückelheim and myself. The experiments were coded up and run by me. I also wrote most of the text. Chapter 5 is partially based on this.
- [5] Navjot Kukreja, Jan Hueckelheim, Mathias Louboutin, John Washbourne, Paul HJ Kelly, and Gerard J Gorman. “Lossy Checkpoint Compression in Full Waveform Inversion.” In: *arXiv preprint arXiv:2009.12623* (2020).
- [6] I wrote most of the code and the text for this paper. This was the first publication that announced Devito to the world. Chapter 3 is partially based on this paper, with updates from more updated publications.
- [6] Navjot Kukreja, Mathias Louboutin, Felipe Vieira, Fabio Luporini, Michael Lange, and Gerard Gorman. “Devito: Automated fast finite difference computation.” In: *2016 Sixth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*. IEEE. 2016, pp. 11–19.
- [7] The conception of this paper was mine as was most of the text. This paper was about an application of checkpointing to neural networks.
- [7] Navjot Kukreja, Alena Shilova, Olivier Beaumont, Jan Hückelheim, Nicola Ferrier, Paul Hovland, and Gerard Gorman. “Training on the Edge: The why and the how.” In: *arXiv preprint arXiv:1903.03051* (2019).
- [8] This paper was the first time we tried using Devito for something other than seismic. The code this was based on was largely written by me. I also contributed to parts of the text.
- [8] Michael Lange, Navjot Kukreja, Mathias Louboutin, Fabio Luporini, Felipe Vieira, Vincenzo Pandolfo, Paulius Velesko, Paulius Kazakas, and Gerard Gorman. “Devito: towards a generic finite difference DSL using symbolic Python.” In: *2016 6th Workshop on Python for High-Performance and Scientific Computing (PyHPC)*. IEEE. 2016, pp. 67–75.

- [9] The API that this paper talks about was mostly written by me, with design inputs from Michael Lange. The text of the paper was mostly written by Michael Lange, with some inputs from me.
- [9] Michael Lange, Navjot Kukreja, Fabio Luporini, Mathias Louboutin, Charles Yount, Jan Hückelheim, and Gerard J Gorman. “Optimised finite difference computation from symbolic equations.” In: *arXiv preprint arXiv:1707.03776* (2017).
- [10] In this paper, I helped formulate the calculation of Operational Intensity in terms of the definition of the minimum data and compute that a stencil needs. This definition became a standard for plotting roofline plots after that. I also wrote the background section of this paper, that I include in Chapter 3.
- [10] Mathias Louboutin, Michael Lange, Felix J Herrmann, Navjot Kukreja, and Gerard Gorman. “Performance prediction of finite-difference solvers for different computer architectures.” In: *Computers & Geosciences* 105 (2017), pp. 148–157.
- [11] This paper talks about Devito’s API. I made code contributions to Devito’s API through the course of this work. I also contributed to part of the text. Chapter 3 includes parts of this paper.
- [11] Mathias Louboutin, Michael Lange, Fabio Luporini, Navjot Kukreja, Philipp A Witte, Felix J Herrmann, Paulius Velesko, and Gerard J Gorman. “Devito: an embedded domain-specific language for finite differences and geophysical exploration.” In: *arXiv preprint arXiv:1808.01995* (2018).
- [12] This is one of the background papers that form the basis for Chapter 2. I contributed to this paper through discussions where we formulated the paper, and also through parts of the text and revisions.
- [12] Mathias Louboutin, Philipp Witte, Michael Lange, Navjot Kukreja, Fabio Luporini, Gerard Gorman, and Felix J Herrmann. “Full-waveform inversion, Part 1: Forward modeling.” In: *The Leading Edge* 36.12 (2017), pp. 1033–1036.
- [13] This paper talks about the internal workings of Devito. My contributions to Devito’s code were my primary contribution to this paper. I also helped improve the experiments through various discussions and wrote code for the improved versions. Chapter 3 includes parts of this paper.

- [13] Fabio Luporini, Michael Lange, Mathias Louboutin, Navjot Kukreja, Jan Hückelheim, Charles Yount, Philipp Witte, Paul HJ Kelly, Gerard J Gorman, and Felix J Herrmann. “Architecture and performance of Devito, a system for automated stencil computation.” In: *arXiv preprint arXiv:1807.03032* (2018).
- [14] This paper was the first use of pyRevolve. I helped coding up the integration of pyRevolve with the Julia framework.
- [14] Philipp A Witte, Mathias Louboutin, Navjot Kukreja, Fabio Luporini, Michael Lange, Gerard J Gorman, and Felix J Herrmann. “A large-scale framework for symbolic implementations of seismic inversion algorithms in Julia.” In: *Geophysics* 84.3 (2019), F57–F71.
- [15] This paper is part of the series that forms the theoretical background. I contributed to this paper through discussions where we formulated the paper, and also through parts of the text and revisions. Chapter 2 includes parts of this paper.
- [15] Philipp Witte, Mathias Louboutin, Keegan Lensink, Michael Lange, Navjot Kukreja, Fabio Luporini, Gerard Gorman, and Felix J Herrmann. “Full-waveform inversion, part 3: optimization.” In: *The Leading Edge* 37.2 (2018), pp. 142–145.

1.5 THESIS OUTLINE

We start the thesis by discussing some essential pieces of background theory in Chapter 2, followed by background on and description of Devito in Chapter 3. This establishes a context to our work, which is then presented in the following chapters.

In Chapter 4, we present our work on checkpointing and in Chapter 5, our work on combining checkpointing with lossy compression. In Chapter 6, we present our work on automatic differentiation, with some special focus towards the end on its implementation in Devito.

Final conclusions of the thesis are presented in Chapter 7.

BACKGROUND

A vulgar Mechanick can practice what he has been taught or seen done, but if he is in an error he knows not how to find it out and correct it, and if you put him out of his road, he is at a stand; Whereas he that is able to reason nimbly and judiciously about figure, force and motion, is never at rest till he gets over every rub.

— Isaac Newton [90]

In this chapter, we establish some theoretical background that we build upon in the subsequent chapters. This chapter borrows concepts and ideas from Louboutin et al. [82], Louboutin et al. [83] and Witte et al. [119].

2.1 SEISMIC IMAGING

Seismic imaging uses many different computational methods. However, for the purposes of this work, we are only interested in Reverse-Time Migration (RTM) and Full-Waveform Inversion(FWI). Mathematically, FWI can be seen as multiple iterations of RTM, although RTM is usually solved on a much finer grid. Therefore, this thesis shall focus on FWI with the implication that all discussions apply equally to RTM unless otherwise specified.

2.1.1 Problem definition

Full-waveform inversion is a computationally demanding problem that has been of interest to both, academia and industry for a while now [113]. In it, we try to computationally recreate an offshore seismic survey typically carried out as shown in Figure 3. In the survey, a ship is sent out at sea with a source of seismic waves attached (e.g. an airgun) and an array of ≈ 10000 receivers in tow. A single experiment consists of a signal sent out by the source and its reflections as detected at each of the receivers. A full survey consists of ≈ 40000 such experiments, where the location of the ship changes from one experiment to another. We refer to these different experiments as data

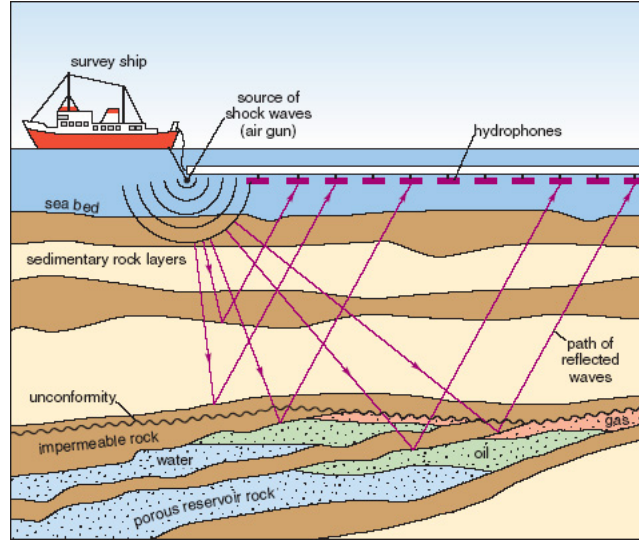


Figure 3: An offshore seismic survey (Source: The Open University ¹). The ship carries a seismic source that produces a wave that travels through the water and into the earth's subsurface. Reflections are created at every sharp boundary. The ship carries thousands of microphones in tow. These microphones record the reflected waves. The set of signals received at all the microphones forms the seismic dataset that is the input to FWI.

points in a seismic dataset where one data point consists of one source signal that was emitted and n_{rec} receiver signals - one for each of the receivers. All signals are t seconds in length, sampled at s samples per second. Given a dataset like this, the objective of FWI is to estimate the values of the physical parameters of the earth's subsurface that best explain the data. This is done by using a wave equation to simulate the propagation of the known source signal through the medium (earth), whose parameters ($\mathbf{m}$) have been estimated and this estimate, $\mathbf{m}$, is revised at every iteration. The solution to the wave equation can be used to calculate the simulated signal $\mathbf{d}_{sim}(\mathbf{m})$, for a given $\mathbf{m}$. We also have $\mathbf{d}_{obs}$, which was the signal observed when the physical experiment was conducted. We can define a *cost function* $\Phi_s(\mathbf{m})$, that calculates the error between the two signals:

$$\Phi_s(\mathbf{m}) = \frac{1}{2} \|\mathbf{d}_{sim}(\mathbf{m}) - \mathbf{d}_{obs}\|_2^2, \quad (1)$$

where, $\mathbf{m}$ represents the model (of the earth's subsurface) parameters being solved for, $\mathbf{d}_{sim}(\mathbf{m})$ represents the simulated receiver data and $\mathbf{d}_{obs}$ the observed receiver data. Both the $\mathbf{d}$ s are represented as

¹ Reproduced from <https://www.open.edu/openlearn/nature-environment/environmental-studies/earths-physical-resources-petroleum/content-section-3.2.1> under CC-BY-SA.

matrices with one dimension n_{rec} and the other dimension being $n_{\text{t}} = t \times s$. Mathematically, FWI is a PDE-constrained optimisation problem where this objective is minimised - we choose the simplest definition of $\Phi_s(\mathbf{m})$ above, but in practice the objective function will have many additional regularising terms. These regularising terms can help turn the problem from ill-posed to well-posed. Common regularisations include penalties that limit the velocity values within physical limits, and Total Variation minimisation (TV) - which codes our theoretical knowledge that the solution in $\mathbf{m}$ is likely to consist of large constant sections with sharp boundaries between them. FWI can therefore be represented as [113]:

$$\underset{\mathbf{m}}{\text{minimize}} \Phi_s(\mathbf{m}), \quad (2)$$

where $\Phi_s(\mathbf{m})$ is defined in Equation (1). The simulated data $\mathbf{d}_{\text{sim}}(\mathbf{m})$ is calculated using a wave equation. Whilst many variations of the wave equation (modelling different physical parameters) are of interest for geophysicists, we focus here on the simplest equation because this discussion is unaffected by the choice of the wave equation. We also present it in a single-dimension formulation for conciseness since, again, the dimensionality does not affect this discussion. The acoustic wave equation can be written in the time domain as:

$$\left\{ \begin{array}{l} \mathbf{m} \frac{d^2 u(x, t)}{dt^2} - \nabla^2 u(x, t) = q \\ u(., 0) = 0 \\ \frac{du(x, t)}{dt} \big|_{t=0} = 0 \end{array} \right. , \quad (3)$$

where $\mathbf{m}$ represents the model parameters being solved for (in this case - squared slowness), $u(x, t)$ represents the wavefield, $\frac{d^2}{dt^2}$ is the second partial derivative in time, ∇^2 is the laplacian, and q is the wave source which is a known quantity provided as part of the data. Although this equation is presented here in one dimension for conciseness, the analysis does not change for up to 3 dimensions, which is the most interesting for the computational optimisations discussed later. This equation is solved using a finite-difference discretisation scheme to calculate the state of the wavefield u at every timestep, given $\mathbf{m}$ and q . While Equation (3) represents the theoretical equation we are solving, we often add a dampening term in practice, to

implement absorbing boundaries. This modifies the equation as follows:

$$\begin{cases} \mathbf{m} \frac{d^2 u(x, t)}{dt^2} - \nabla^2 u(x, t) + \eta \frac{du(x, t)}{dt} = q \\ u(., 0) = 0 \\ \frac{du(x, t)}{dt} \big|_{t=0} = 0 \end{cases}, \quad (4)$$

where η is the dampening coefficient that we set to zero throughout the physical domain, and increasing rapidly in the region of the absorbing boundary.

The central finite-difference expression for a first derivative is given by:

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x-h)}{2h}, \quad (5)$$

which extends to the second derivative as follows:

$$f''(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - 2 * f(x) + f(x-h)}{h^2}. \quad (6)$$

We can use Equation (6) to write $\frac{d^2 u(x, t)}{dt^2}$ as:

$$\frac{d^2 u(x, t)}{dt^2} = \frac{u(x, t + \Delta t) - 2 * u(x, t) + u(x, t - \Delta t)}{\Delta t^2}. \quad (7)$$

With this, Equation (3) can be fully discretised. Let us only explore the time discretisation for now, keeping in mind that a similar treatment is done for the space discretisation to expand $\nabla^2 u(x, t)$, which we shall skip in the following discussion for brevity. Let us rewrite Equation (3) using Equation (7):

$$\mathbf{m} \frac{u(x, t + \Delta t) - 2 * u(x, t) + u(x, t - \Delta t)}{\Delta t^2} - \nabla^2 u(x, t) = q \quad (8)$$

Rearranging to express $u(x, t + \Delta t)$ in terms of $u(x, t)$ and $u(x, t - \Delta t)$, we get:

$$u(x, t + \Delta t) = 2 * u(x, t) - u(x, t - \Delta t) + \frac{\Delta t^2}{\mathbf{m}} (\nabla^2 u(x, t) + q). \quad (9)$$

Equation (9) gives us our timestepping scheme that defines timestep $i + 1$ in terms of timesteps i and $i - 1$, allowing us to progress the simulation further in time. This leads to a dataflow pattern shown in Figure 4. Note that the graph assumes a two-level time discretisation

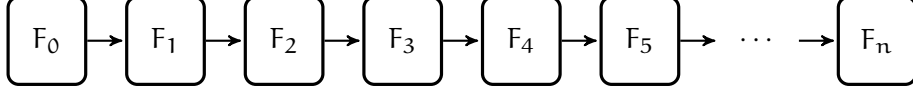


Figure 4: Data flow graph of the forward problem, i.e. given an $\mathbf{m}$, solving for $\mathbf{d}_{\text{sim}}(\mathbf{m})$. F_k represents the $(k+1)$ th Forward timestep

for simplicity. It can be seen from the graph that the data dependencies in the forward problem are quite straightforward. The total length of the simulation is chosen so that the wave has enough time to reach the entire domain but not too much time that reflections start becoming significant.

We can now define an operator $A(\mathbf{m})$ based on Equation (3):

$$A(\mathbf{m}) = \mathbf{m} \frac{d^2}{dt^2} - \nabla. \quad (10)$$

The discretised wavefield is now given as $A(\mathbf{m})^{-1}$, still ignoring the contribution of the source. An injection operator $\mathbf{P}_s^T$ can now be introduced that represents the injection of the discretised source term q_s into the wavefield. The final wavefield, including the source term is hence:

$$\mathbf{u} = A(\mathbf{m})^{-1} \mathbf{P}_s^T q_s. \quad (11)$$

Similar to the injection operator, another projection operator $\mathbf{P}_r$ can be introduced, that interpolates the signal received at the (known) receiver locations which are located at points not necessarily aligned with the finite-difference grid. The simulated receiver data then becomes:

$$\mathbf{d}_{\text{sim}}(\mathbf{m}) = \mathbf{P}_r \mathbf{u} = \mathbf{P}_r A(\mathbf{m})^{-1} \mathbf{P}_s^T q_s. \quad (12)$$

What Equation (12) describes is also often called the “forward problem” in this context.

Using Equation (12), we can rewrite Equation (1) to capture the full problem as:

$$\underset{\mathbf{m}}{\text{minimize}} \frac{1}{2} \|\mathbf{P}_r A(\mathbf{m})^{-1} \mathbf{P}_s^T q_s - \mathbf{d}_{\text{obs}}\|_2^2. \quad (13)$$

2.2 ADJOINT-STATE METHOD

Equation (13) makes it clear how the “misfit error”, i.e. the objective function value is computed. However, to use a gradient-based method, we also need to calculate the gradient of this function, with respect to $\mathbf{m}$. For this, we use the adjoint-state method. To proceed, we first require the adjoint of the PDE being solved in the forward problem (Equation (3)). If the forward equation is represented as an operator A , the adjoint (represented as A^*) is an operator that satisfies the property:

$$\langle Ax, y \rangle = \langle x, A^* y \rangle. \quad (14)$$

This property is used as part of the dot-product test to verify the correctness of an adjoint. For the special case of the acoustic equation (Equation (3)), the adjoint PDE is the same as the forward PDE. The adjoint of Equation (4) is only slightly different:

$$m \frac{d^2 v(x, t)}{dt^2} - \nabla^2 v(x, t) - \eta \frac{dv(x, t)}{dt} = q_r, \quad (15)$$

where q_r is the adjoint source. Typically $q_r = \mathbf{d}_{\text{sim}}(\mathbf{m}) - \mathbf{d}_{\text{obs}}$, the misfit between the simulated and the observed signal is injected as a source into the adjoint solution.

Let us start by differentiating Equation (1) w.r.t. the model parameter $\mathbf{m}$:

$$\frac{d\Phi}{d\mathbf{m}} = \frac{d}{d\mathbf{m}} \left(\frac{1}{2} \|\mathbf{d}_{\text{sim}}(\mathbf{m}) - \mathbf{d}_{\text{obs}}\|_2^2 \right) \quad (16)$$

$$= \frac{d}{d\mathbf{m}} (\mathbf{d}_{\text{sim}}(\mathbf{m}) - \mathbf{d}_{\text{obs}})^T \cdot (\mathbf{d}_{\text{sim}}(\mathbf{m}) - \mathbf{d}_{\text{obs}}) \quad (17)$$

$$= \frac{d}{d\mathbf{m}} (\mathbf{P}_r A^{-1}(\mathbf{m}) \mathbf{P}_s^T q_s)^T \cdot \delta d. \quad (18)$$

We know that:

$$\frac{dA^{-1}(\mathbf{m})}{d\mathbf{m}} = -A^{-1}(\mathbf{m}) \frac{dA(\mathbf{m})}{d\mathbf{m}} A^{-1}(\mathbf{m}). \quad (19)$$

Hence,

$$\frac{d\Phi}{d\mathbf{m}} = (\mathbf{P}_r A^{-1}(\mathbf{m}) \text{diag}(\frac{dA(\mathbf{m})}{d\mathbf{m}} A^{-1}(\mathbf{m}) \mathbf{P}_s^T q_s))^T \cdot \delta d. \quad (20)$$

Differentiating Equation (10) w.r.t. $\mathbf{m}$:

$$\frac{dA}{d\mathbf{m}} = \frac{d^2}{dt^2}. \quad (21)$$

Using Equations (11) and (21) in (20):

$$\frac{d\Phi}{d\mathbf{m}} = (\mathbf{P}_r \mathbf{A}^{-1}(\mathbf{m}) \text{diag}(\frac{d^2 \mathbf{u}}{dt^2}))^T \cdot \delta \mathbf{d} \quad (22)$$

$$= -\text{diag}(\mathbf{u}_{tt}[\mathbf{t}]) \cdot \mathbf{A}^{-T}(\mathbf{m}) \mathbf{P}_r^T \cdot \delta \mathbf{d}. \quad (23)$$

Since we know that,

$$\mathbf{J}^T = -\text{diag}(\mathbf{u}_{tt}[\mathbf{t}]) \cdot \mathbf{A}^{-T}(\mathbf{m}) \mathbf{P}_r^T, \quad (24)$$

we can conclude that:

$$\frac{d\Phi}{d\mathbf{m}} = \mathbf{J}^T \delta \mathbf{d}, \quad (25)$$

where $\mathbf{J}$ is the Jacobian of $\mathbf{A}$. We also know that:

$$\mathbf{A}^{-T}(\mathbf{m}) \mathbf{P}_r^T \cdot \delta \mathbf{d} = \mathbf{v}[\mathbf{t}], \quad (26)$$

where $\mathbf{v}$ is the adjoint wavefield, and hence:

$$\frac{d\Phi}{d\mathbf{m}} = -\text{diag}(\mathbf{u}_{tt}[\mathbf{t}]) \cdot \mathbf{v}[\mathbf{t}]. \quad (27)$$

Since the second time derivative ($\frac{dA(\mathbf{m})}{d\mathbf{m}}$) is linear w.r.t. $\mathbf{u}$, it is interchangeable and we can also write:

$$\frac{d\Phi}{d\mathbf{m}} = -\text{diag}(\mathbf{u}[\mathbf{t}]) \cdot \mathbf{v}_{tt}[\mathbf{t}]. \quad (28)$$

Consolidating, the gradient of the objective function $\Phi_s(\mathbf{m})$ with respect to the model parameter $\mathbf{m}$ is given by :

$$\nabla \Phi_s(\mathbf{m}) = \sum_{t=1}^{n_t} \mathbf{u}_{tt}[\mathbf{t}] \mathbf{v}[\mathbf{t}] = \sum_{t=1}^{n_t} \mathbf{u}[\mathbf{t}] \mathbf{v}_{tt}[\mathbf{t}] = \mathbf{J}^T \delta \mathbf{d}, \quad (29)$$

where $\mathbf{u}[\mathbf{t}]$ is the forward wavefield and $\mathbf{v}[\mathbf{t}]$ is the adjoint field, $\mathbf{u}_{tt}[\mathbf{t}]$ the second time derivative of $\mathbf{u}$, and similarly $\mathbf{v}_{tt}[\mathbf{t}]$ the second time derivative of $\mathbf{v}$. While we have given the derivation of the above relation by the chain rule, Plessix [95] provides a derivation of the same relationship using lagrangian multipliers (Equation 36). The adjoint solution is constrained by the forward solution. Specifically, the misfit

between the simulated and observed version of the signals received at the receivers during the forward propagation are injected into the adjoint field, which is propagated backward in time.

To complete the gradient calculation according to Equation (29), the adjoint field is correlated with the forward field. Figure 2 shows the data flow of this computation. The correlation clearly requires the complete progression of the forward wavefield in time. For anything but the most trivial problems, the space required for storage becomes prohibitively expensive. In Chapter 4, we discuss checkpointing as one strategy to address this memory footprint. In Chapter 5, we discuss many other alternatives.

2.2.1 Gradient test

When we implement Equation (29) in code, we would require a test to verify the correctness of the implementation. Here we describe a test that we turn to repeatedly for this purpose. The test uses the Taylor expansion of the gradient to test whether the calculated gradient follows the expected convergence for small perturbations. The test involves calling the objective function repeatedly in a small local region and verifying that the result varies as expected. The test can be written mathematically as:

$$\begin{aligned}\epsilon_0 &= \Phi_s(\mathbf{m}_0 + h\mathbf{d}\mathbf{m}) - \Phi_s(\mathbf{m}_t) \\ \epsilon_1 &= \Phi_s(\mathbf{m}_0 + h\mathbf{d}\mathbf{m}) - \Phi_s(\mathbf{m}_0) - h\langle \nabla \Phi_s(\mathbf{m}_0), \mathbf{d}\mathbf{m} \rangle,\end{aligned}\quad (30)$$

where Φ_s is defined in Equation (13). Here, ϵ_0 is the first-order error that can be expected to grow linearly with h and ϵ_1 is the second-order error that can be expected to grow with h^2 . We use this test implemented in code to verify our implementation of the gradient function everywhere.

2.2.2 Automatic differentiation

The adjoint-state method, as it applies to PDE-constrained optimisation problems, can be seen as a special case of a more general method. Reverse-mode automatic differentiation is a general technique that can be used to calculate the gradient of an arbitrary objective function defined in code. It can be shown that the two methods arrive at the same expression for the gradient of the FWI objective func-

tion given in Equation (13) [99]. The automatically derived gradient, although numerically equivalent, is much harder to optimise for computational performance and hence we have only used hand-written code for the gradient so far. Chapter 6 describes the possibility of automating this.

2.3 CONCLUSION

In this chapter, we briefly discussed some concepts that are going to be useful in the rest of this thesis. In the next chapter, we discuss Devito and how it helps us implement the equations we discussed here.

DEVITO - A DOMAIN-SPECIFIC LANGUAGE FOR FINITE DIFFERENCE

*That language is an instrument of human reason, and not merely a medium
for the expression of thought, is a truth generally admitted.*

— Boole [17]

In this chapter, we introduce Devito, a domain-specific language and compiler for finite-difference computations that formed the context for all the investigations we do in this thesis. This chapter is based on Kukreja et al. [67] and Louboutin et al. [80]. Section 3.2 is reproduced exactly from Louboutin et al. [80], while the rest of the text in this chapter is new. The author was the first contributor to Devito, but over the course of this work, the team grew to many more people. The author does not take credit for all of Devito, but for some parts of it. This chapter presents Devito to provide context for the rest of the work.

3.1 INTRODUCTION

Devito is a domain-specific language (DSL) for finite-difference simulations that accepts the problem definition in a symbolic form and generates highly performant C code to solve the problem. Devito is embedded in python, inheriting all its constructs and adding a few of its own. Devito is closely associated with this thesis as the Devito project forms the context of this work and was used for all the experiments presented in this thesis.

Before Devito, this problem would be solved by writing a huge monolithic code, most likely in C/C++. This code would have performance optimisations and mathematical details spread all around. Anyone trying to modify the code would need to understand the mathematics as well as the hardware-specific details of code optimisation. This code would need to be rewritten for every new hardware platform.

The DSL approach has several advantages over the current approach of software development for this problem. Firstly, a DSL that is close to the mathematics of the problem introduces a clear separation

of concerns between the physicist/mathematician and the computer scientist since the former only needs enough programming knowledge to be able to write the problem in Devito’s mathematical syntax. The latter does not need to know the details of the problem being solved. This also reduces redundant effort as the performance optimisations do not need to be reapplied every time the underlying equation changes. Performance portability is also easier to achieve with code written using Devito.

The Devito project began with the automation of the generation of code for finite-difference simulations used in seismic imaging [67]. In the initial versions, the code generation was less like a compiler and largely template-based, where compilable C code would be assembled automatically using pre-written code fragments combined just-in-time. Even at that time, the approach was general enough to work beyond just the wave equation, for example the diffusion equation [70]. This was followed by performance optimisation of the generated code without changes to the DSL [71]. Since, with Devito, it was possible to change the finite-difference discretisation with a single parameter, it was easy to study the impact of the choice of mathematical discretisation on computational performance [79].

3.2 BACKGROUND ON COMPILERS

Improving the performance of a critical piece of code on a particular computing platform is a non-trivial task that has received significant attention throughout the history of computer science. The desire to automate the performance optimisation process itself across a range of target architectures is not new either, although it is often met with scepticism. Even the very first compiler, Ao by Hopper [47], was received with resistance, as best summarised in the following quote: *“Dr. Hopper believes,..., that the result of a compiling technique should be a routine just as efficient as a hand-tailored routine. Some others do not completely agree with this. They feel the machine-made routine can approach hand tailored coding, but they believe there are “tricks of the trade” that apply to various special cases that a computer cannot be expected to utilise.”* [58]. This is not very far from the current state of programming in the HPC community, more than 60 years later, as manual performance optimisation for different architectures of legacy codes is still widely applied in scientific computing.

Community acceptance of these new “automatic coding systems” began when concerns about the performance of the generated code were addressed by the first “optimising compiler”, FORTRAN, released in 1957 – which not only translated code from one language to another but also ensured that the final code performed at least as good as a hand-written low-level code [9]. Since then, as program and hardware complexity rose, the same problem has been solved over and over again, each time by the introduction of higher levels of abstractions. The first high-level languages and compilers were targeted at solving a large variety of problems and hence were restricted in the kind of optimisations they could leverage. As these generic languages became common-place and the need for further improvement in performance was felt, restricted languages focusing on smaller problem domains were developed that could leverage more “tricks of the trade” to optimise performance. This led to the proliferation of domain-specific languages for broad mathematical domains or sub-domains, such as APL [54], Mathematica, Matlab or R.

In addition to these relatively general mathematical languages, more specialised frameworks targeting the automated solution of PDEs have long been of interest [18, 22, 109, 112]. More recent examples not only aim to encapsulate the high-level notation of PDEs but are often centred around a particular numerical method. Two prominent contemporary projects based on the finite element method (FEM), FEniCS [77] and Firedrake [97], both implement a common DSL, UFL [2], that allows the expression of variational problems in the weak form. Multiple DSLs to express finite-difference problems in a symbolic form have emerged over time, including Hawick and Playne [41], Arbona et al. [6] and Jacobs, Jammy, and Sandham [55]. However, their use in seismic imaging problems is limited by at least two factors: Firstly, most modern FD DSLs rarely offer enough flexibility to handle additional features beyond the original scope of the DSL, such as sparse-point interpolation to model source injection or sampling at the receiver locations. Since FD operators only form part of the over-arching optimisation problem, composability with external packages is a key requirement that is often ignored by self-contained standalone DSLs. The use of a fully embedded Python DSL, on the other hand, allows users to leverage a variety of higher-level optimisation techniques through a rich variety of software packages provided by the scientific Python ecosystem (e.g SciPy optimisation toolbox).

At the execution level, a finite-difference code belongs to the class of “stencil” codes, i.e. an iterative kernel that updates the elements of an array based on a fixed pattern. Since the class of stencil codes is more general than that of FD codes, there are many more tools available to automate the generation of stencil codes including SDSL [43], PATUS [21], Mint [110], Impala [61], HiPACC [87], Stella [92], Pochoir [107], Pluto [15], YASK [120]. These tools require an explicit definition of the stencil computation, leaving the user to derive the discretised form of the governing equation. For complex formulations this process can be time-consuming and error-prone.

3.3 USING DEVITO

The Devito API is based around two key concepts that allow users to express finite difference problems in a concise symbolic notation:

- **Functions:** Devito’s Function objects behave like `sympy.Function` objects and provide a set of shorthand notations for generating derivative expressions, while also managing user data. The rationale for this duality is that many stencil optimisation algorithms rely on data layout changes, mandating that Devito needs to be in control of data allocation and access.
- **Operators:** An `Operator` creates, compiles and executes a single executable kernel from a set of SymPy expressions. The code generation and optimisation process involves various stages and accepts a mixture of high-level and low-level expressions to allow the injection of customised code.

Using the above classes, Equation (3) can now be solved using the following code:

```
from devito import Grid, TimeFunction
grid = Grid()
m = Function(name='m', grid=grid, space_order=2)
u = TimeFunction(name='u', grid=grid, time_order=2, space_order=2)
# Derive stencil from symbolic equation
eqn = m * u.dt2 - u.laplace
stencil = solve(eqn, u.forward)[0]
update_u = [Eq(u.forward, stencil)]
op = Operator(update_u)
```

This operator can be applied by calling `op.apply()`, which, by default, would use data from the objects used to create it. Since Devito initialises all its data objects with zeros, the initial values from Equation (3) are implicit in the above code. The data can be replaced by passing the replacement data as arguments to `apply`. It can also be applied to only a part of the domain (in space or time) by passing start and end values for any dimension. We shall extend this code in the following sections.

3.4 PERFORMANCE

Since finite-difference kernels are essentially stencils, the means to achieve high performance are well studied. Devito already implements common stencil optimisations like loop blocking, OpenMP based parallelisation, vectorisation etc. It also performs symbolic optimisations like flop reduction using common subexpression elimination (CSE). Figure 5 shows the performance of Devito-generated code on Intel Broadwell CPU. These optimisations are implemented inside Devito as modules called the Devito Symbolic Engine (DSE) and the Devito Loop Engine (DLE). The DSE is responsible for symbolic optimisations like flop count reduction using CSE. The DLE is responsible for loop-level optimisations like blocking, parallelism etc.

While the optimisations provided by “core” Devito provide good performance on CPUs, hardware like Intel® Xeon Phi™ (Knights Landing) can benefit from platform-specific code. For this, Devito supports the concept of backends and the first such backend made was to support YASK [120]. Using this backend, Devito generates code targeted at the YASK compiler instead of the regular C compiler. As expected, code compiled through Yask performs at close to machine peak.

3.5 SPARSE-POINT INTERPOLATION

Another area that required some work was the interpolation of sparse points on to grid points. This occurs commonly for seismic sources and receivers from the physical seismic experiment where these points are not aligned with the grid. Earlier, Devito implemented automatic linear interpolation to handle these. However, more sophisticated interpolators are often required for real-life problems. As a general solution, Devito’s generic `PrecomputedSparseFunction` class accepts pre-

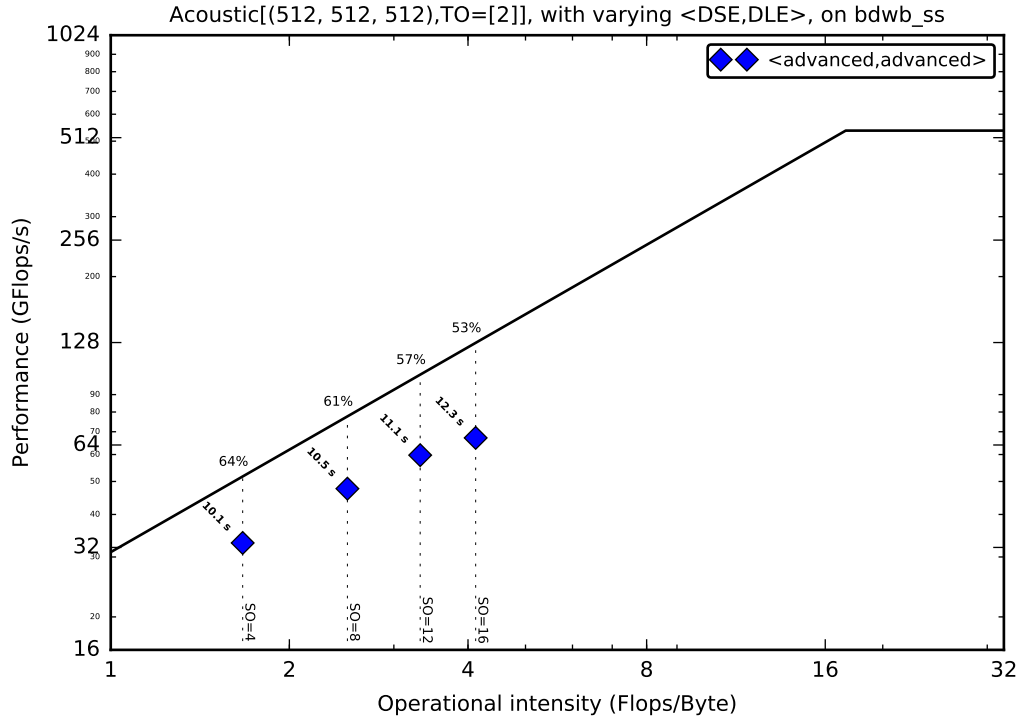


Figure 5: Roofline plot showing the performance of Devito-generated code on Broadwell. The kernel used was based on the isoacoustic wave equation, on a cubical domain of 512 points in each dimension. The optimisation level in Devito was set to *advanced*. The horizontal axis represents *Operational Intensity*, or the number of computations carried out per byte of data read. The four dotted vertical lines represent the four versions of the computational kernel tested for spatial discretisation orders of 4, 8, 12, and 16. The blue diamonds on these lines represent the actual measured performance. We have also included the absolute runtime as well as the percentage of peak. The solid black lines represent the maximum achievable performance (the peak) on that specific hardware - with the region under the angled line forming the *memory-bound* region and the region under the horizontal solid line forming the *compute-bound* region.

computed interpolation coefficients corresponding to every physical point. This allows any interpolator like a sinc-based [45] interpolator to be used easily by reusing implementations from existing libraries.

3.6 CORRECTNESS OF THE GENERATED CODE

Although automated testing, including unit-tests, adjoint (dot-product) tests and gradient tests (using a linearised approximation of the known objective function), was a part of Devito since the first version, passing such tests is only a necessary condition for a correct implementation - not a sufficient one. Formal code verification methods can provide necessary *and* sufficient conditions that ensure correct implementations but are too cumbersome to implement for non-trivial software and hence are rarely used. The fact that Devito generates code automatically made it easier to adapt the generated code to the requirements of CIVL, a popular verification tool [51].

3.7 IMPLEMENTING THE INVERSION KERNELS WITH DEVITO

Before we delve into the implementation of inversion kernels with Devito, let's do a quick recap of the relevant equations we will be implementing. We start by reproducing Equation (12) here for convenience.

$$\mathbf{d}_{\text{sim}}(\mathbf{m}) = \mathbf{P}_r \mathbf{u} = \mathbf{P}_r \mathbf{A}(\mathbf{m})^{-1} \mathbf{P}_s^T \mathbf{q}_s. \quad (12)$$

Here, we are solving for the simulated receiver signal ($\mathbf{d}_{\text{sim}}(\mathbf{m})$) and $\mathbf{q}_s$ is the source term, $\mathbf{P}_s^T$ the source injection operator, $\mathbf{P}_r$ the receiver interpolation operator, and $\mathbf{A}(\mathbf{m})^{-1}$ is the operator representation of the wave equation, which in the simple case is the acoustic wave equation given in Equation (3) and also reproduced below:

$$\left\{ \begin{array}{l} \mathbf{m} \frac{d^2 \mathbf{u}(\mathbf{x}, t)}{dt^2} - \nabla^2 \mathbf{u}(\mathbf{x}, t) = \mathbf{q} \\ \mathbf{u}(\cdot, 0) = 0 \\ \frac{d\mathbf{u}(\mathbf{x}, t)}{dt} \big|_{t=0} = 0 \end{array} \right. . \quad (3)$$

The forward wavefield can then be obtained as a time-marching procedure, solving Equation (12) forward in time. Figure 6 shows the complete code required to implement this in Devito, when we do not need to save the intermediate states. Figure 7 shows a version of the

```

def forward(model, m, eta, src, rec, order=2):
    # Create the wavefield function
    u = TimeFunction(name='u', shape=model.shape, time_order=2,
                     space_order=order, save=None)

    # This is Equation 3
    eqn = m * u.dt2 - u.laplace + eta * u.dt

    stencil = solve(eqn, u.forward)[0]

    # This defines the time-marching scheme for computing the
    # wavefield
    update_u = [Eq(u.forward, stencil)]

    # The parameter src represents  $q_s$  from Equation 12

    # The next line defines  $P_s^T$ , the source injection operator.
    # The expression is from Equation (9).
    src_term = src.inject(field=u, expr=src * dt**2 / m)

    # The next line defines  $P_R$ , the receiver interpolation
    # operator
    rec_term = rec.interpolate(expr=u)

    # Create operator with source and receiver terms
    return Operator(update_u + src_term + rec_term,
                   subs={s: dt, h: model.spacing})

```

Figure 6: Devito code required for a forward operator. This code completely implements Equation (12). This version of the code does not save the intermediate states. We shall refer to this as the *non-taped Forward Operator*.

```

def forward(model, m, eta, src, rec, order=2):
    # Create the wavefield function
    u = TimeFunction(name='u', shape=model.shape, time_order=2,
                     space_order=order, save=nt)

    # This is Equation 3
    eqn = m * u.dt2 - u.laplace + eta * u.dt

    stencil = solve(eqn, u.forward)[0]

    # This defines the time-marching scheme for computing the
    # wavefield
    update_u = [Eq(u.forward, stencil)]

    # The parameter src represents qs from Equation 12

    # The next line defines PsT, the source injection operator.
    # The expression is from Equation (9).
    src_term = src.inject(field=u, expr=src * dt**2 / m)

    # The next line defines PR, the receiver interpolation
    # operator
    rec_term = rec.interpolate(expr=u)

    # Create operator with source and receiver terms
    return Operator(update_u + src_term + rec_term,
                    subs={s: dt, h: model.spacing})

```

Figure 7: Devito code required for a forward operator that saves all intermediate states. The only difference from the code in Figure 6 is the `save=` parameter to the `TimeFunction` constructor. We shall refer to this as the *taped Forward Operator*.

code that saves the intermediate states. Knowing $\mathbf{d}_{\text{sim}}(\mathbf{m})$, we can now calculate the value of the objective function of the optimisation problem, which, for our purposes, is simply the L_2 norm of the difference between $\mathbf{d}_{\text{sim}}(\mathbf{m})$ and the given $\mathbf{d}_{\text{obs}}$.

The next requirement for solving the optimisation problem is to calculate the gradient. This is done using Equation (29), which we reproduce here for exposition:

$$\nabla \Phi_s(\mathbf{m}) = \sum_{t=1}^{n_t} \mathbf{u}_{tt}[\mathbf{t}] \mathbf{v}[\mathbf{t}] = \mathbf{J}^T \delta \mathbf{d} \quad (29)$$

where, $\mathbf{u}_{tt}[\mathbf{t}]$ is the second time derivative of the forward wavefield, which is only a short further computation from the result we had in Equation (12). In this equation, $\mathbf{v}[\mathbf{t}]$ represents the adjoint wavefield, which is obtained by solving the adjoint Equation (15).

The procedure to compute a gradient for a single shot of FWI can then be summarised as:

1. Compute the synthetic data $\mathbf{d}_{\text{sim}}(\mathbf{m})$ with a forward solve with Equation (12).
2. Compute the adjoint wavefield from the data residual with the adjoint Equation (15).
3. Compute the gradient as the correlation of the forward and adjoint wavefield with equation (29).

For the first step, we have already presented the code in Figure 7. The `operator` thus created can be used to model a forward where the `src_term` contains the source to be injected into the field and `rec_term` will be extracting the receiver information from the simulation.

Clearly, the third step requires the intermediate data from both the previous steps. This means we must use the code in Figure 7 instead of Figure 6. Storing both the forward and adjoint wavefields in memory would be naive since two complete wavefields need to be stored. The first obvious optimisation is to merge steps 2 and 3 as a single pass where step 3, the gradient calculation, can use the output from step 2 (the adjoint wavefield) as it is calculated, hence saving the need for storing the adjoint wavefield in memory. The `operator` for the combined steps 2 and 3 can be created in Devito using the code given in Figure 8. The second time-derivative of u is given by `u.dt2`. Having a time derivative implies having an index of the time dimension in the expression. Devito automatically detects dimensions and their indices in expressions, and surrounds those expressions with loops in those dimensions. Hence the space/time loops are always implicit in Devito code.

3.8 DEVITO AT SCALE

While most experiments for this work only required a single gradient evaluation for a single shot – which is a much smaller problem than FWI, we report results on some runs on FWI in Chapter 5. This required distributed implementations of FWI running at scale. One implementation [116], uses Julia’s built-in task-farming capabilities to do this transparently. We first built an implementation that used traditional HPC schedulers to run FWI experiments across many nodes. In later work, we incorporated DASK to handle the parallelism across

```

def gradient(model, m, eta, src, rec, order=2):
    # Create the adjoint wavefield function
    v = TimeData(name='v', shape=model.shape, time_order=2,
                  space_order=order)

    # This implements Equation (29)
    gradient_update = Eq(grad, grad - u.dt2 * v)

    # This is Equation (15)
    eqn = m * v.dt2 - v.laplace - eta * v.dt
    stencil = solve(eqn, v.backward)[0]
    eqn = Eq(v.backward, stencil)

    # The objects that were receivers in the forward pass
    # will act as sources here, so we reuse the name and inject
    # them.
    # The expression is from Equation (9).
    receivers = rec.inject(field=v, expr=rec * dt * dt / m)

    return Operator([eqn] + [gradient_update] + receivers, subs={
        s:dt, h: model.get_spacing()}, time_axis=Backward)

```

Figure 8: Devito code required for an operator that calculates the adjoint and gradient in a single pass

shots. This is an embarrassingly parallel dimension since, for each shot, there is a long-running computation (that also requires a large amount of memory), followed by a resulting gradient that has to be reduced across the shot dimension. This is a *map-reduce* problem that DASK is well-suited to handle.

To scale this on the cloud, we built Docker containers – one corresponding to the *login* node, where the optimisation loop runs, one for DASK’s scheduler, and a varying number of DASK workers (often 15-20). This was orchestrated using Kubernetes. Using shell scripts we wrote for the purpose, we could launch this *cluster* at the start of an experiment and remove it at the end.

3.9 CONCLUSIONS

Through this chapter, we have introduced Devito and how it enables developers to implement finite-difference operators that are necessary for FWI, among many other problems. We went into detail showing code examples of the implementation of the specific kernels that we will be using further in the next chapters.

CHECKPOINTING

There are 2 hard problems in computer science: cache invalidation, naming things, and off-by-1 errors.

—Leon Bambrick

This chapter introduces the checkpoint-restart method that is commonly used to reduce the memory footprint of a derivative calculation. The text in this chapter before Section 4.7 is a reproduction from Kukreja et al. [64], while the rest of the text is new.

4.1 INTRODUCTION

In the last chapter, we developed the basic kernels required for FWI - the forward and the gradient (reverse) kernels. We also developed an appreciation of the fact that the gradient kernel requires the forward kernel to save all the intermediate states. This requirement introduces a significant memory footprint in the solver program. Figure 9 shows a visual depiction of this data access pattern where the forward pass generates a large amount of data, which is then used in a reverse pass - in the reverse order.

This memory footprint is the focus of this chapter (and Chapter 5). It is important to note that despite the similar terminology, the checkpointing we refer to in this chapter is not done for resilience or failure recovery. The objective of this checkpointing is to reduce the memory footprint of a large computation.

Table 1 estimates the computational requirements of an FWI problem on the SEAM Model [30]. Although the grid-spacing and time-step interval depends on various problem-specific factors, we can do a back-of-the-envelope calculation to appreciate the scale of FWI. To estimate the number of operations per grid point, we use a variant of Equation (3) called Tilted-Transverse Isotropy (TTI) [121], which is commonly used today in commercial FWI. Such a problem would require almost 90 days of continuous execution at 1 PFLOP/s. The memory requirements for this problem are also prohibitively high. As can be seen in Table 1, the gradient computation step is responsible for this problem's high memory requirement.

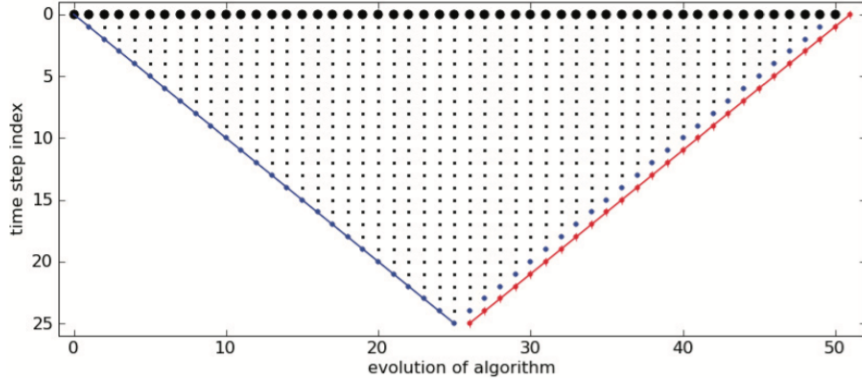


Figure 9: A visual representation of the data access pattern for the gradient computation where all intermediate results are kept in memory until they are required by the reverse pass. The horizontal axis is wall-clock time and the vertical axis is simulation time/timesteps. The blue line represents forward computation, followed by a reverse computation shown in red. Each dot represents a saved checkpoint. A vertical line at any point shows the number of checkpoints stored in memory at that point in time. Memory consumption is at a peak at the end of the forward computation. Image source: [115]

Description	Number	Peak Memory	No. of Flops
Single grid point (TTI)	-	8 bytes	6300
Complete grid	$1000 \times 1000 \times 1000$	8GB	6.3×10^{12}
Forward propagation	10000	24GB	6.3×10^{16}
Gradient Computation	2 (FW+REV) ¹	80TB	1.26×10^{17}
Shots	10000	80TB	1.26×10^{21}
Optim. Iterations	20	80TB	2.52×10^{22}

Table 1: Estimated computational requirements of a Full-Waveform Inversion problem based on the SEAM model [30]. Note that real-world FWI problems are likely to be larger. ¹A gradient computation involves a forward simulation followed by a reverse/adjoint computation. For simplicity we assume the same size of computation during the forward/adjoint pass.

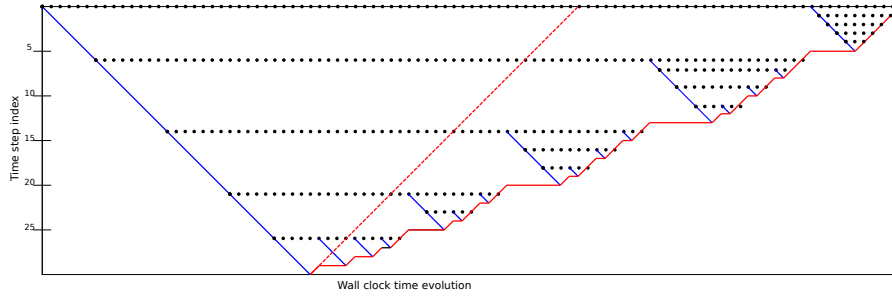


Figure 10: Checkpoint-recompute strategy where only a subset of intermediate results are stored. The other timesteps can be recomputed when required by rerunning the forward computation from the last available checkpoint. The sections where the red line is horizontal are the parts of the computation where the reverse computation waits for the forward computation to be repeated in order to recompute some data that was previously discarded. The straight-dotted red line shows the hypothetical path of the reverse computation had there been no recomputation required.

4.2 REDUCING THE MEMORY FOOTPRINT

One way of reducing this footprint is by only storing the forward wavefield at certain timesteps. The wavefield for the rest of the timesteps can be recovered when required by redoing the forward computation starting from the nearest available checkpoint. Figure 10 shows a visual depiction of this idea where some of the data from the forward computation is discarded at first. When the reverse computation reaches a point where this data is required, the forward computation is restarted from the last available state to recover this discarded data and continue the reverse computation.

Previous work on similar problems led to the *Revolve* algorithm [37] and the associated tool which provides an optimal schedule at which to store checkpoints, i.e. states from which the forward simulation can be restored. This was extended by several followup studies into other optimal algorithms. We discuss *Revolve* and extensions in Section 4.3. A study of optimal checkpointing for seismic inversion was done in [106] but this was not accompanied by a high-level abstraction that made the integration of other software easier with *Revolve*.

The *Revolve* tool and algorithm, however, only provide the schedule to be used for checkpointing. Although this eases some of the complexity of the application code using the algorithm, the glue code required to manage the forward and adjoint runs is still quite complex. This acts as a deterrent to the more widespread use of the algorithm in the community.

In this chapter, we describe how the *Revolve* algorithm can be combined with code generation to make checkpointing much more accessible. We call the software that can enable this *pyRevolve* and it is described in section 4.5. This software forms the basis of the experiments we present in Chapter 5. Although we use Devito and FWI in the examples here, the abstraction and software proposed here are quite general in nature and can be used in any problem that requires checkpointing in combination with a variety of computational methods.

In Section 4.6 we provide some experimental results based on which we judged the correctness and performance of the implementation.

4.3 OPTIMAL SCHEDULES

As we have seen, the usage of adjoint methods allows the computation of gradient information within a time that is only a very small multiple of the time needed to evaluate the underlying function itself. However, for nonlinear processes like the one we saw in the previous section, the memory requirement to compute the adjoint information is in principle proportional to the operation count of the underlying function, see, e.g., [33, Sec. 4.6]. In Chap. 12 of the same book, several checkpointing alternatives to reduce this high memory complexity are discussed. Checkpointing strategies use a small number of memory units (checkpoints) to store the system state at distinct times. Subsequently, the information that is needed for the adjoint computation but not available is recomputed using these checkpoints in an appropriate way. Several checkpointing techniques have been developed, all of which seek an acceptable compromise between memory requirement and runtime increase. Here, the obvious question is where to place these checkpoints during the forward integration to minimize the overall amount of required recomputations.

A checkpointing schedule provides the sequence of operations that need to be carried out to reproduce a computational procedure (in this case, an adjoint-state gradient computation) but with a lower memory footprint. Since the forward computation is restarted from the last available checkpoint whenever a required timestep is not available in memory, the locations of the checkpoints (i.e. which timesteps to checkpoint) are critical in determining the amount of extra computation that might be required. Devising a checkpointing schedule that, for a given problem, minimises the recomputation given a cer-

tain amount of memory, is an optimisation problem in itself. We shall call this the checkpoint scheduling problem.

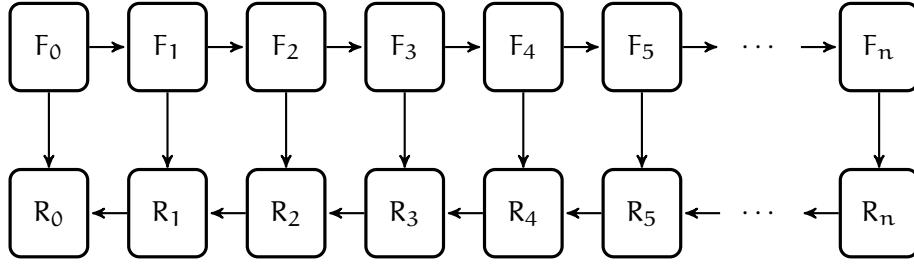


Figure 2: The data-flow graph of a gradient computation. (repeated from page 31)

To develop corresponding optimal checkpointing strategies, one has to take into account the specific setting of the application. A fixed number of time steps to perform and a constant computational cost across all the time steps to calculate is the simplest situation. It was shown in Griewank and Walther [37] that, for this case, a checkpointing scheme based on binomial coefficients yields the minimum number of time steps to be recomputed. This is the *Revolve* algorithm. It provides the optimal schedule under the following assumptions:

1. The number of timesteps is known in advance
2. One level of memory available (usually DRAM)
3. All forward steps are of uniform computational cost
4. The data to be stored in any checkpoint is the same size
5. Saving or retrieving a checkpoint takes no time
6. Stopping/restarting operators takes no time
7. Each computational step requires the output data from the immediately preceding computational step (we previously discussed this in Chapter 2, Figure 4).

4.3.1 The Revolve algorithm

While a complete description of the *Revolve* algorithm is beyond the scope of this work, we go into a brief description of the formulation here.

The checkpoint-scheduling problem aims to find the schedule of checkpoint-restart that minimises total time-to-solution ($t(N, M)$), given

the number of timesteps (N) and the number of checkpoints (M). Let us assume that the first checkpoint is placed at timestep $\tilde{N} < N$. We now have two subproblems - one to the left of $\tilde{N}$ and the one to the right of $\tilde{N}$. Since the objective function $t(N, M)$ is linear in subproblems, it can be seen intuitively that, under the assumption of uniform computational cost per step, any strategy that minimises t for the subproblems also minimises t for the overall problem. This satisfies the optimal substructures property which means dynamic programming is a good method for this problem.

We can formalise the description of the previous paragraph into the following dynamic programming relation [33, Sec. 12.3]:

$$t(N, M) = \min_{1 \leq \tilde{N} < N} \{ \tilde{N}.C + t(N - \tilde{N}, M - 1) + t(\tilde{N}, M) \} \quad (31)$$

The above formulation aims to find the location of the first checkpoint $\tilde{N}$, such that it minimises the time-to-solution $t(N, M)$. The first term $\tilde{N}.C$ is the time taken to conduct the forward computation for $\tilde{N}$ steps, where C is the time for a single forward step. The second term $t(N - \tilde{N}, M - 1)$ is the time-to-solution for the subproblem to the right of this first checkpoint, which has $N - \tilde{N}$ timesteps and $M - 1$ checkpoints. The third term $t(\tilde{N}, M)$ is the time-to-solution for the subproblem to the left of the first checkpoint, which has $\tilde{N}$ timesteps and M checkpoints.

If, for any subproblem, $M = N$, i.e. we have memory available to store every time step, we know that storing every timestep will minimise the time-to-solution. If, for any subproblem, $M = 1$ and $N > 1$, i.e. there is only checkpoint available and we have more than one timestep to reverse, there is only one strategy available. This involves using the one checkpoint available to store the initial state of this subproblem, running N forward steps (to the end), followed by the first reverse step. We then restart from the initial checkpoint, running $N - 1$ timesteps followed by the second reverse step and so on, yielding a total of $N(N - 1)/2$ forward steps. A more complete description in Griewank and Walther [33, Sec. 12.3] shows how the solution devolves to binomial coefficients for intermediate values of M .

4.3.2 Extensions to Revolve

An obvious extension of this approach would be to include flexibility with respect to the computational cost of the time steps. For example, if one uses an implicit time-stepping method based on the solution of a nonlinear system, the number of iterations needed to solve the nonlinear system may vary from time step to time step, yielding unpredictable time step costs. In this situation, it is no longer possible to derive an optimal checkpointing strategy beforehand. Some heuristics were developed to tackle this situation [102]. However, extensive testing showed that, even in the case of unknown nonuniform step costs, binomial checkpointing is quite competitive. Another very important extension is the coverage of adaptive time stepping. In this case, the number of time steps to be performed is not known beforehand. Therefore, so-called online checkpointing strategies were developed, see, e.g., [105, 115]. These studies all extend *Revolve* around the first of the assumptions given in Section 4.3.

Extensions to the *Revolve* algorithm around the second of the assumptions are very useful to FWI i. e. multistage checkpointing where more than one level of memory is used. These strategies combine storage to multiple levels of memory like RAM, local disks, network disks etc. with recomputation to find optimal combinations that minimise the total time-to-solution. The first such strategy was proposed in Stumm and Walther [104] without proof of optimality. This was followed up by Aupy et al. [8] with a proof of optimality for an offline schedule (number of timesteps is known in advance). This was further formalised in Aupy and Herrmann [7] into a periodic formulation which was independent of the number of timesteps. In followup work, the authors of the previous study provide a scheduling utility that can generate optimal schedules for memory hierarchies of arbitrary depth [44]. All these studies, however, assume synchronous disk reads and writes, i.e. that the computation stops to read or write a checkpoint, which might be wasteful in a practical implementation. Schanen et al. [101] talks about an implementation of asynchronous multistage checkpointing but with no discussion on the optimality of the provided schedule. No optimal schedules for asynchronous checkpointing exist in literature as per our understanding. The reason for this boils down to the fact that optimal scheduling of asynchronous disk access would need accurate performance models of computation that is carried out in parallel with disk access. The ideal behaviour

Scenario	Paper
Known number of timesteps, Single-level memory	Griewank and Walther [36]
Unknown number of timesteps, Single-level memory	Wang, Moin, and Iaccarino [115]
Known number of timesteps, Two-level memory Synchronous	Aupy et al. [8]
Unknown number of timesteps, Two-level memory Synchronous	Aupy and Herrmann [7]
Unknown number of timesteps, Two-level memory Asynchronous Non-optimal	Schanen et al. [101]

Figure 11: Summary of extensions to *Revolve* that are relevant to the most common applications in Devito

would be that computation running in parallel with asynchronous access would suffer no slowdown - however in practice there are some shared resources between disk access and computation, causing a noticeable slowdown in the computation which is hard to model.

The third and fourth assumptions of *Revolve*— uniform checkpoint size and computational cost, usually hold for FWI. Some extensions to *Revolve* along these assumptions are discussed in Section 4.8.

4.4 THE *revolve* SOFTWARE

The software *Revolve* implements binomial checkpointing, online checkpointing as described in [105], and hierarchical, also called multi-stage, checkpointing derived in [104]. For this purpose, it provides a data structure *Revolve* to steer the checkpointing process and the storage of all information required for the several checkpointing strategies.

To illustrate the principle structure of an adjoint computation using checkpointing, Fig. 12 illustrates the kernel of *Revolve* used for the binomial checkpointing. The two remaining checkpointing strategies are implemented in a similar fashion only taking the additional

```

r=new Revolve(steps,snaps)
do
  whatodo = r->revolve()
  switch(whatodo)
    case advance: for r->oldcapo < i <= r->capo
                  forward(u, m)
    case firstrun: eval_J(u, m)
                  init(bu, bm)
                  adjoint(bu, bm, u, m)
    case youturn: adjoint(bu, bm, u, m)
    case takeshot: store(u, ustore, r->check)
    case restore: restore(u, ustore, r->check)
while(whatodo <> terminate)

```

Figure 12: *Revolve* algorithm with calls to the application interface

extensions into account. The forward integration as well as the corresponding adjoint computation is performed within a `do-while` loop of the structure in Fig. 12, where `steps` and `snaps` denote the number N of time steps of the forward simulation and the number M of checkpoints available for the adjoint computation, respectively.

Hence, the routine *Revolve* determines the next action to be performed which must be supported by the application being differentiated. These actions are

- **advance:** Here, the user is supposed to perform a part of the forward computation based on the routine `forward(u, m)`, where u represents the state of the system and m the control. The variable `r->oldcapo` contains the current number of the state of the forward integration. That is, before starting the `for` loop, u holds the state at time $t = r->oldcapo$. The variable `r->capo` determines the targeted number of the state of the forward integration. Therefore, $(r->capo - r->oldcapo)$ time steps have to be performed to propagate the state u from the time $t = r->oldcapo$ to the time $t = r->capo$.
- **firstrun:** This action signals the start of the adjoint computation. Therefore, first the target function is evaluated. Then, the user has the possibility to initialize the adjoint variables bu and bm . Subsequently, the first adjoint step is performed.
- **youturn:** The next adjoint step has to be performed.
- **takeshot:** Here, the user is supposed to store the current state u in the checkpoint with the number `r->check`. The array of checkpoints is here denoted by `ustore` but the specific organisation of

the checkpoints is completely up to the user. During the adjoint computation `r->check` selects the checkpoint number appropriately such that all states needed for the adjoint computation are available. Once the adjoint computation has started, states that were stored in the checkpoints are also overwritten to reuse memory.

- **restore:** The content of the checkpoint with the number `r->check` has to be copied into the state `u` to recompute the forward integration starting from this state.

It is important to note that this checkpointing approach is completely independent from the method that is actually used to provide the adjoint information. As can be seen, once an adjoint computation is available the implementation can incorporate binomial checkpointing to reduce the memory requirement.

We also have to stress that *Revolve* provides a so-called serial checkpointing which means that only one forward time step or one adjoint step is performed at each stage of the adjoint computation. Nevertheless, the computation of the forward time step and/or the adjoint step may be performed heavily in parallel, i.e., may be evaluated on a large scale computer system. This is in contrast to so-called parallel checkpointing techniques where several forward time steps might be performed in parallel even in conjunction with one adjoint step. Corresponding optimal parallel checkpointing schedules were developed in [114]. However, so far no implementation to steer such a parallel checkpointing process is available. In the case of FWI, sufficient parallelism is available at other levels of the problem so we did not explore this level of parallelism for this work.

The *Revolve* software also includes an **adjust** procedure that computes, for a given number of time steps, the number of checkpoints such that the increase in spatial complexity equals approximately the increase in temporal complexity. Using the computed number as the number of checkpoints minimises cost when assuming that the user pays computational resources per node and per time, e.g. the cost is proportional to the available memory and the runtime of the computation. This scenario is most applicable in cloud computing settings.

4.5 ABSTRACTIONS FOR CHECKPOINTING

In this section, we will discuss the package *pyRevolve*, which has been developed during the course of this work to encapsulate *Revolve* checkpointing in a user-friendly, high-level python library. This library is available online, along with its source ¹. We will first provide details of its implementation in Section 4.5.1. Afterwards, we will discuss the interplay of *pyRevolve* with the C++ checkpointing implementation that was previously discussed in Section 4.3. Finally, we will discuss the usage of *pyRevolve* in an FWI application in Section 4.5.2, implemented in the Devito domain-specific language. Figure 13 shows a UML representation of the interaction between different software packages that we will describe. Although Section 4.5.2 discusses the special case of Devito, the interface of the *pyRevolve* library was designed to allow easy integration into other python codes as well.

4.5.1 API, *pyRevolve* side

The *pyRevolve* interface was designed as part of providing checkpointing to users of Devito with an accessible API. The design had the following goals:

- Making checkpointing available to users of Devito without forcing them to get involved in implementation details like loops, callbacks, data storage mechanisms. The user should choose whether to use checkpointing in one place, but not be forced to do anything beyond this.
- All knowledge of checkpointing, different strategies (online/off-line checkpointing, multistage) shall be contained within one module of the python framework, while still benefiting all operations in the code.
- The checkpointing itself should be contained in a separate library that allows others to use it easily, even if they are not interested in using Devito. This matured into *pyRevolve*.
- Since the data movement requires intricate knowledge of the data structures used and their organisation in memory, this is handled by the application code.

¹ <https://github.com/devitocodes/pyrevolve>

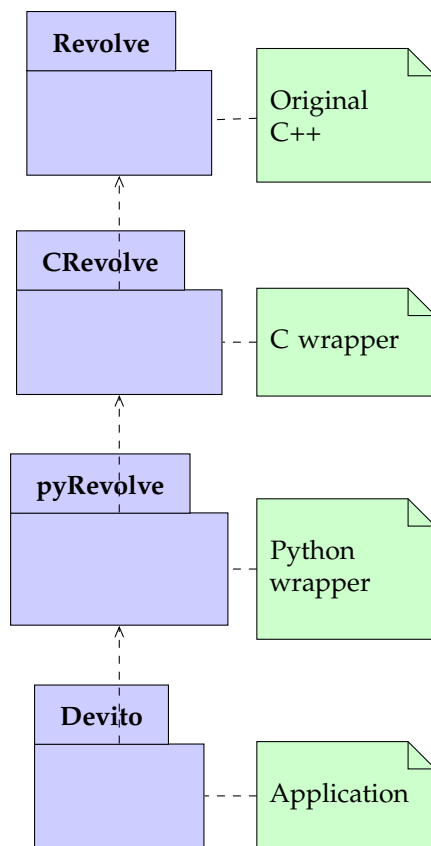


Figure 13: Packages overview. *Revolve* has been described in Section 4.3. The packages *pyRevolve* and *cRevolve* and how they are used to create a high-level abstraction of checkpointing are explained in Section 4.5.

To achieve these goals, *pyRevolve* was designed for the following overall workflow, which will be explained in more detail in the following sections. The term *application* here refers to the application using *pyRevolve* as a library (in this case Devito).

To begin with, the application creates objects with an `apply` method to perform the actual forward and reverse computations, which are both instances of a concrete implementation of the abstract base class `Operator`. The application also creates an instance of a concrete implementation of the abstract base class `Checkpoint` that can deep-copy all time-dependent working data that the operators require into a specified memory location. Next, the application instantiates *pyRevolve*'s `Revolver` object and passes the forward and reverse operators, and the checkpoint object.

When required, the application starts the `Revolver`'s forward sweep, which will complete the forward computation and store checkpoints as necessary. After the forward sweep completes, the application can finalize any computation that is based on the forward data, such as the evaluation of objective functions, or store the final result as necessary. This may be accompanied/ followed by the initialisation of the adjoint data structures. After this, the application calls the `Revolver`'s reverse sweep. This will compute the adjoint, possibly by performing partial forward sweeps and loading checkpoint data.

The *pyRevolve* package contains *cRevolve*, which is a thin C wrapper around a previously published C++ implementation². The C++ files in this package are slightly modified for compatibility with Python, but the original is available from the link in the footnote. The *cRevolve* wrapper around the C++ library is taken from `libadjoint`³. *pyRevolve* uses *cRevolve* as part of the `Scheduler` abstraction which is designed to make it easy to integrate other schedulers like *H-Revolve*. Figure 14 shows the classes that *cRevolve* exposes, along with their members.

One key design aspect is that *pyRevolve* is not responsible for performing the data copies, and therefore does not need to know about the properties or structure of the data that needs to be stored. For this purpose, *pyRevolve* provides the `Checkpoint` abstract base class that has a `size` attribute, and a `load(ptr)` and `save(ptr)` method. The user must provide a concrete implementation of such an object. The `size` attribute must contain the size of a single checkpoint in memory, this information is used by *pyRevolve* to allocate the correct amount

² <https://www.mathematik.hu-berlin.de/de/forschung/forschungsgebiete/mathematische-optimierung/andrea-walther/walther-software>

³ <https://bitbucket.org/dolphin-adjoint/libadjoint>

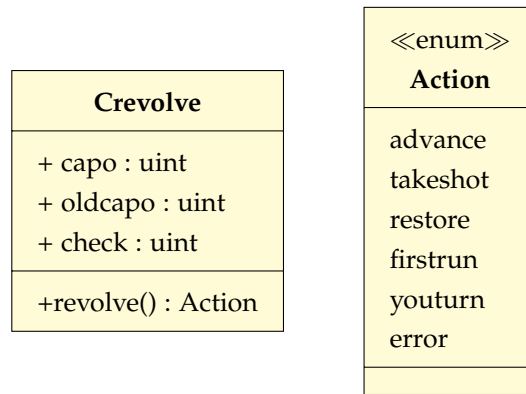


Figure 14: Crevolve classes.

of memory. The `save` method must deep-copy all working data to the memory region starting at the provided pointer, and the `load` method must restore the working data from the memory region starting at the pointer `ptr`, either by performing a deep-copy, or by pointing the computation to the existing data inside the checkpoint storage.

The *pyRevolve* library provides to the user the class `Revolver` that must be instantiated with the following arguments:

- **Checkpoint object:** This has to be an implementation of the abstract base class `Checkpoint`.
- **Forward operator:** An object that provides a function `apply()` as specified in Section 4.5.2 that performs the forward computation.
- **Reverse operator:** Similarly, an object that provides a function `apply()` that performs the reverse computation.
- **Number of checkpoints:** This is optional, and specifies the number of checkpoints that can be stored in memory. If it is not given, a default value is computed using the `adjust` method explained in Section 4.3.
- **Number of time steps:** This is also optional. If it is not given, an online checkpointing algorithm is used.

Based on either the given or computed number of checkpoints, the constructor instantiates a `Storage` object that allocates the necessary amount of memory (number of checkpoints $\times$ checkpoint size), and makes that memory accessible to the `Revolver`. Figure 15 shows the classes inside *pyRevolve* and how they interact.

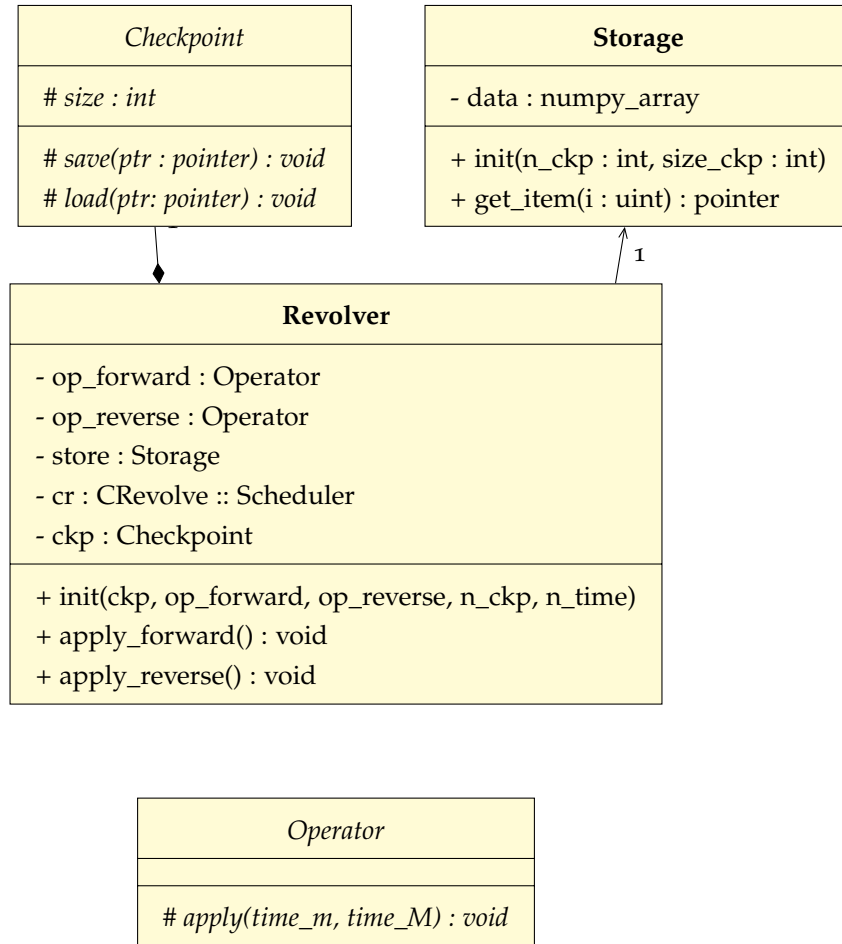


Figure 15: *pyRevolve* classes. The abstract classes *Checkpoint* and *Operator* are implemented by the client application

```

# Example time-varying field that needs checkpointing
u = TimeData(...)
# Some expression that generates the values for u
fw = Operator(...)
# Some expression that uses the values of u
rev = Operator(...)
cp = DevitoCheckpoint([u])
revolver = Revolver(cp, fw, rev, nt)
# Forward sweep that will pause to take checkpoints
revolver.apply_forward()
# Could perform some additional steps here
# Reverse sweep that uses the checkpoints
revolver.apply_reverse()

```

Figure 16: Devito code to utilize checkpointing based on *pyRevolve*

4.5.2 API, Application side

To introduce checkpointing using the *pyRevolve* library, an application must implement a particular interface. We use Devito here as an example application, however, everything discussed here is fairly general and any application may implement checkpointing using *pyRevolve* using the approach discussed in this section.

To begin with, a concrete implementation of *pyRevolve*'s abstract base class *Checkpoint*, called *DevitoCheckpoint* was created. This class has three methods:

- **save(ptr)**: Save the contents of the working memory into the location ptr.
- **restore(ptr)**: Restore a previously stored checkpoint from location ptr into working memory.
- **size**: Report the amount of memory required by a single checkpoint. This is used to decide the total amount of memory to be allocated and to calculate offsets.

Along with this, we set up two Operators, one a *ForwardOperator* to carry out the forward computation and a *GradientOperator* that computes the gradient. These operators were explained in Chapter 3 and the code included in Figures 6 and 8. These can be used to initialize a *Revolver* object as shown in Figure 16.

On initialisation of the *Revolver*, object, the *DevitoCheckpoint* object is queried for the size of one checkpoint and *pyRevolve* allocates `n_checkpoints*checkpoint.size` bytes of memory for the storage of checkpoints. Calling `revolver.apply_forward()` carries out a forward

DevitoCheckpoint
+ size : int
+ save(ptr : pointer) : void + load(ptr: pointer) : void

ForwardOperator	ReverseOperator
- data	- data
+ apply(time_m, time_M) : void	+ apply(time_m, time_M) : void

Figure 17: Devito operators, and the implementation of a Checkpoint class.

run but broken down into chunks as specified by the checkpointing schedule provided by *Revolve*, each chunk being executed by calling `fwd.apply()` with arguments `t_m` and `t_M` corresponding to the timesteps to run the simulation for. Between these chunks, `cp.save()` is automatically called to save the state to a checkpoint.

On calling `revolver.apply_reverse()`, the *Revolver* calls `rev.apply()` with the relevant `t_m/t_M` arguments for the sections where the result from the forward pass is available in a checkpoint. This will be loaded by a call to `cp.load()`. For others, it will automatically call `fwd.apply()` to recompute and store in memory the results from a part of the forward operator so the reverse operator can be applied for that part.

Following this work, users of Devito can easily add optimal checkpointing to their adjoint computations by following the steps described above.

4.6 VERIFICATION

There are two possible ways of testing the numerical accuracy of an implementation - solving a problem whose solution has certain known mathematical properties and verifying these properties numerically, and comparing the results to a reference solution. Here we do both - we use the gradient test as described in Section 2.2.1 and also verify that the numerical results match those from a reference implementation. This test is carried out for a certain $\mathbf{m}$, which we call here $\mathbf{m}_0$. This is the smoothed version of a two-layer model

$\mathbf{m}_t$ i.e. the true model has two horizontal sections, each with a different value of squared slowness. The measured data required by the objective function is modeled on the true two-layer model and $\mathbf{d}\mathbf{m} = \mathbf{m}_o - \mathbf{m}_t$. The constant h then varies between 10^{-1} and 10^{-4} to verify that $\epsilon_0 = O(h)$ is a first order error and $\epsilon_1 = O(h^2)$ is a second order error. The code used for this test can be found in the repository for Devito ⁴. We used a grid with $230 \times 230 \times 230$ points. With the simulation running for 1615 timesteps, this required about 80 GB to store the full forward wavefield in memory. The first run was made with the regular gradient example that stores the entire forward wavefield in memory. This was then repeated with checkpointing, with a varying number of checkpoints. It was verified that the results from all versions matched each other exactly and also passed the gradient test mentioned previously.

4.7 IMPLEMENTATION CONCERNS

Revolve provides an optimal checkpointing schedule for a set of assumptions that we previously described in Section 4.3. We also discussed how *Revolve* was extended to remove the first four assumptions. In this section we discuss the difficulties we encountered in realising other assumptions of *Revolve* when building a practical implementation in *pyRevolve*.

4.7.1 The extra checkpoint in the beginning

An implicit assumption in *Revolve* is that initial conditions must be checkpointed or they would be lost. For this reason, *Revolve* always uses the first checkpoint to store the state before the first timestep. In the case of Devito’s applications, the initial timestep is most commonly empty i.e. a zeroed field. Even in the case where the user provides an initial state, it would be provided in externally allocated memory that *pyRevolve* does not control and will be available throughout the simulation so this checkpoint is wasteful in our opinion. We avoid storing this checkpoint by reporting one extra checkpoint to *Revolve*, and ignoring the first checkpoint that *Revolve* places at the beginning of the computation.

⁴ <https://github.com/opesci/devito>

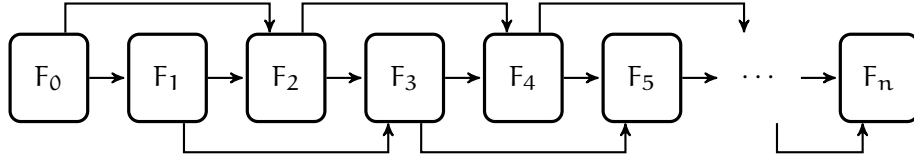


Figure 18: The non-simplified data flow graph of the forward computation showing that the data dependence is actually not just on the last timestep, but on the last two.

4.7.2 The extra forward step

Revolve, with its aim of being general, assumes that the forward operator has two modes - a taped and non-taped mode. This is provided in case the operator has some internal state within a single timestep that should be saved when this timestep is going to be immediately followed by an adjoint step (the taped mode) - this state would not be saved if the next step is not going to be an adjoint step (the non taped mode). *Revolve* calls the forward operator in the taped mode for one “extra” step just before the corresponding adjoint step - all other forward operator calls are in the non-taped mode.

In their simplest form, the forward operators based on the wave equations have no internal state within a timestep and hence have only one mode. In this scenario this *extra* forward step is unnecessary. In *pyRevolve* we currently ignore this *extra* forward step due to our knowledge of this fact.

If we were to club k forward (and corresponding reverse) timesteps (fine timesteps) as a single timestep (coarse timestep) for *Revolve*, we would now have the state of the wavefield between the fine timesteps which become the internal state for a coarse timestep. The forward operator implementing this would now have two modes - one where it is asked to save the fine timesteps (the taped mode), and one where it is not, and the “extra” timestep would become important.

4.7.3 Multi-level timestepping schemes

Revolve assumes that each computational timestep only requires the result of the immediately preceding timestep. This can be called *two-level timestepping*. However, many PDEs, including the wave-propagation PDEs employ multi-level time-stepping methods where the results of more than one previous timestep are used to calculate the next timestep. For Equation (3), for example, the most natural time-stepping

scheme requires the result of the previous two timesteps. Figure 18 shows a graphical representation of this data dependence. This implies the need to checkpoint the data for two timesteps to be able to restart the computation from an intermediate timestep.

The first solution we adopted was to simply declare each checkpoint to be the size of two timesteps in the wavefield. It is important to remember here that *Revolve* tends to checkpoint every timestep towards the end of the simulation, when memory is available to do so. The combination of the last two facts proves to be wasteful in that many timesteps are checkpointed twice. This situation gets worse as M gets closer to N and peaks when *pyRevolve* is called with $M = N$. In this scenario every timestep is being checkpointed twice. Hence, while one would expect the case of $M = N$ to behave identical to the case of no checkpointing, in practice we see twice the memory consumption and twice the data movement as the non-checkpointed version.

We later improved this by tracking metadata regarding the timestep numbers being checkpointed, and using this information to avoid moving the same timestep twice - either while loading or while saving. This change contributes to the change seen from Figure 19 to Figure 20 in Section 4.7.4.

While this reduces the data movement down to expected levels, this implementation still consumes more memory than strictly required since we are still allocating memory and calculating checkpoint schedules using the two-level time-stepping assumption – twice as much memory in the worst case.

A more complete method to address this could be precalculating the schedule with *Revolve* under the two-level time-stepping assumption and using this to predetermine the real timesteps to be checkpointed. The repeated checkpointed timesteps could be detected and removed at this point. However this might lead to the implementation using less memory than available due to the removal of these *extra* checkpoints. Using this extra memory for further checkpointing to reduce the time-to-solution might affect optimality and it might be better to develop an algorithmic extension to *Revolve* that incorporates this aspect in the scheduling.

Making *pyRevolve*'s memory allocation be fully aware of multi-level time-stepping remains future work. While it is not ideal for a memory-saving technique to be *wasting* memory, we know that this issue only becomes significant when $M \approx N$. In practice, we have seen that *pyRe-*

volve is very useful for scenarios where $M \ll N$ and there this issue does not have a significant impact.

4.7.4 Zero-cost checkpointing

Zero-cost checkpointing is *Revolve*'s assumption that the cost of saving/loading a checkpoint is negligible. This assumption is easy to realise for problems where the individual timesteps have a very high cost and the time taken to execute a timestep is much larger than the time taken to copy a checkpoint from one memory location to another. In our implementation, the wave equation solver is often memory-bound, which means an additional memory copy comes at similar (time) cost as a timestep. We studied the impact of this in the following experiment.

We setup FWI on the Overthrust velocity model[4] on a grid of $287 \times 881 \times 881$. This grid required 500mb per timestep in single precision and was to be solved for 2287 timesteps, which would require terabytes of DRAM if done without checkpointing. This problem was run multiple times with a varying number of checkpoints. The peak memory usage was tracked for each such run, as well as the total time to solution. The memory consumption was measured using the *memory_profiler* python module and the time-to-solution by using the *time* python command before and after the function to be profiled. To eliminate variation in the results, the timings are the minimum value from three runs. The tests were carried out on an Intel®Xeon®CPU E5-2640 v3 @ 2.60GHz (Haswell) with 128GB RAM.

Figure 19 shows the total time-to-solution running the same problem for a varying number of timesteps including time spent in various sections of the code. The sections in blue are the time spent saving and loading checkpoints - it is clear that the *storage overheads*, i. e. time spent loading and saving checkpoints, are dominating the overall time of computation so this assumption is far from realised at this point.

We then profiled various implementations of memcpy to find the best performing one and changed the copy/save functions to use this new implementation. We made another change that removed some unnecessary data movement, as described in Section 4.7.3 before repeating this experiment. Figure 20 shows the updated timing information in this second experiment.

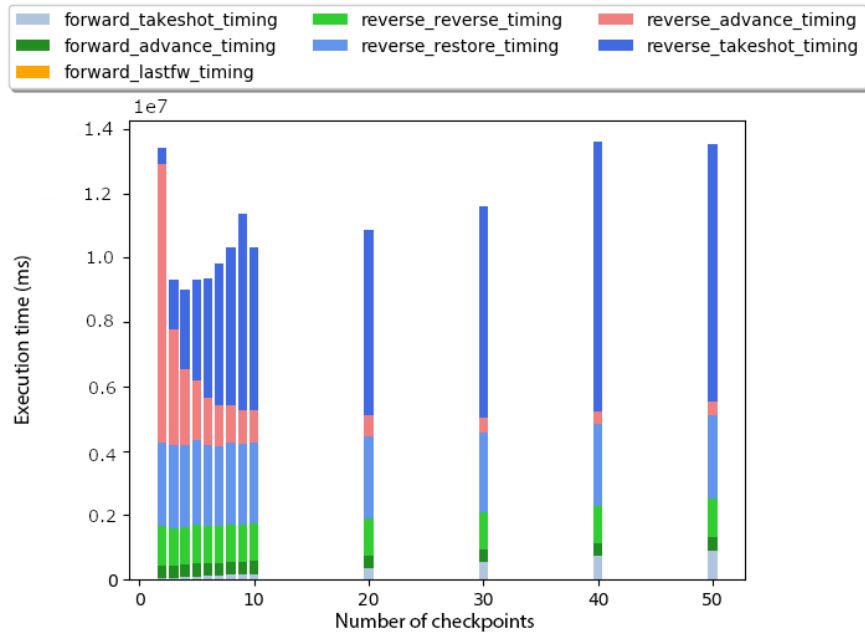


Figure 19: Time-to-solution when running the same problem with *pyRevolve* with different number of checkpoints - with time spent on different subparts of the process. The problem was a single forward-reverse pass on an FWI test case on the Overthrust velocity model[4]. The blues represent storage overheads - *Revolve* assumes these to be zero.

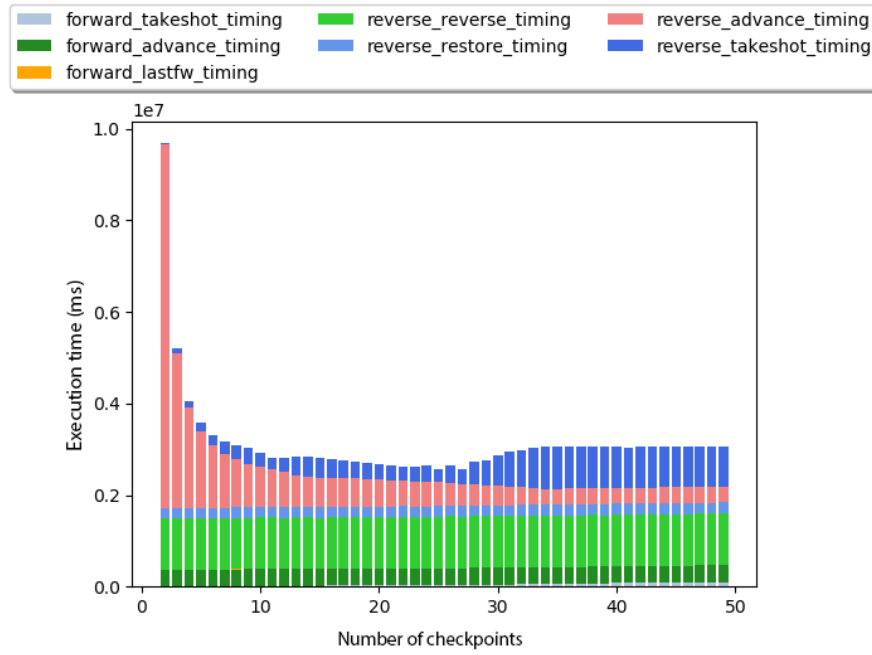


Figure 20: Same experiment as Figure 19, but now with a more efficient memcpy implementation and reduced data movement (described in Section 4.7.3)

It can be seen in Figure 20 that the storage overheads are now down to much more reasonable levels. For lower values of M , the time-to-solution drops as M increases - very close to the theoretical predictions. As M increases, the program starts spending more time reading and writing checkpoints and the storage overheads cause the time-to-solution curve to flatten out instead of continuing to decrease.

In Section 4.7.2, we discussed a method that could alleviate this issue of *costly checkpointing* further i.e. to club multiple timesteps into a single timestep for the purposes of *Revolve*. This would increase the cost of a single timestep relative to the cost of storing/retrieving checkpoints and bring us closer to the original assumption. This would also help alleviate the problem discussed in Section 4.7.5, where we discuss this method further.

This method, however, would only hide the data movement overhead behind larger blocks of computation. In the next section, we briefly discuss another method that can completely eliminate this overhead.

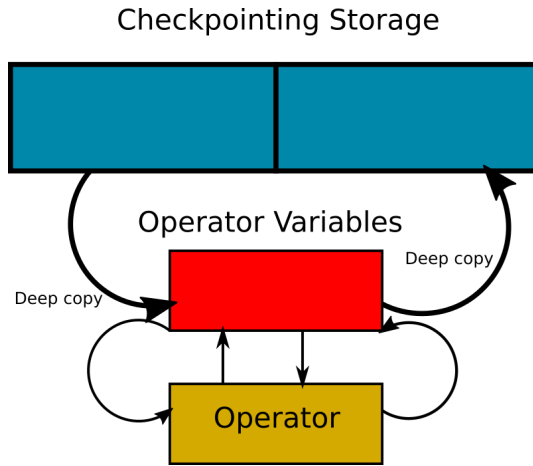


Figure 21: The existing (and most common) approach to checkpointing in Devito/pyRevolve. Checkpoints are stored and retrieved with deep copies.

4.7.4.1 In-place operators

Devito’s operators currently assume that the initial data for the computation is provided in its own data structure (`Function.data`) and the final result is also to be written to the same structure. This necessitates copying the data in before starting any intermediate compute and copying any results that are to be checkpointed, out to the checkpointing memory at the end of any partial compute. These two additional memory copies at every checkpoint introduce a significant overhead that peaks when the number of checkpoints available is the same as the number of timesteps, since in this case *Revolve* devolves to the store-all strategy, albeit the “store-all” will happen in the memory allocated for checkpointing, meaning 2 additional memory copies at every time step. This assumption is not unique to Devito and many production-ready libraries and codes suffer from this assumption. Figure 21 shows a schematic of this *conventional* approach to checkpointing.

This assumption of Devito is the root cause for our inability to realise *Revolve*’s zero-cost checkpointing assumption in practice since, an implementation could be pictured that can read and write directly from/to checkpointing memory when required. Such an implementation would indeed exhibit zero-cost checkpointing, suffering from no data movement overhead when reading/writing checkpoints.

This could be addressed by having Devito’s operators accept an *in-buffer*, from where it would read the initial data (directly from checkpointing memory), and an *outbuffer* to write the final result to check-

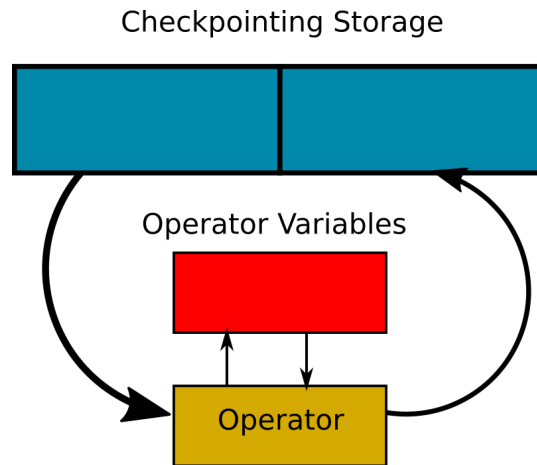


Figure 22: The suggested approach where the Operators operate directly on checkpointing memory

pointing memory. Any intermediate timesteps could utilise the Operator’s internal memory, i.e. the Devito-allocated memory. Figure 22 shows a schematic of this changed approach.

We carried out an experiment to simulate the effect of removing these copies by writing “dummy” operators, that only sleep at calling apply, and also added sleeps at checkpoint save and restores to simulate that checkpoint storage/retrieval costs the same as a computation. Figure 23 shows how the total runtime of a forward-backward pass could be reduced by moving to the approach of working directly on checkpoint memory.

Devito already provides the ability to create symbolic objects that represent externally allocated memory through the code shown in Figure 24. To convert from an operator that expects all its data to live in Devito objects to an operator that can operate on external data, one could unroll the first and last two timesteps (for 2nd order time-stepping) and carry out symbolic manipulation that changes the symbolic objects these timesteps operate on, from the original symbols representing internal memory to symbols representing external memory. A routine that can perform this transformation automatically for any given Operator could then be used to convert from internal-memory Operators to in-place Operators.⁵ A production-ready implementation of this method remains future work.

The ability of Devito’s Operators to work on external memory would prove to be useful in many other scenarios that are not directly

⁵ A prototype implementation of this method is available at <https://github.com/devitocodes/navjotk/tree/allocator3>

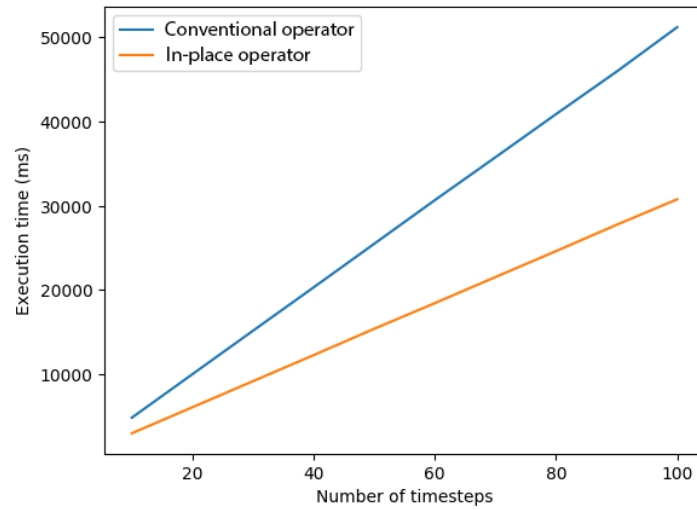


Figure 23: Comparison of execution times for the two strategies. The existing strategy is called “explicit” here and the suggested one is called “implicit”. For each experiment, the number of checkpoints was the same as the number of timesteps, to bring out the worst case since the data movement is worst when the number of checkpoints is equal to the number of timesteps.

```
u = TimeFunction(name='u', grid=grid,
                 space_order=space_order,
                 allocator=ExternalAllocator(np_array),
                 initializer=external_initializer,
                 time_order=time_order)
```

Figure 24: Example Devito code to create a symbol representing externally allocated memory

related to checkpointing. In principle, this method could be used to compute a few timesteps while carrying out any data movement - in effect providing *free* data movement across Devito-based applications. A place where this would be particularly useful is when running Devito on the GPU. The CPU could compute the first two timesteps when copying the initial conditions into GPU memory, and the last two timesteps when copying the final state out from GPU memory to host memory, effectively amortising the cost of the Host-GPU data movement.

4.7.5 Operator-call overhead

Another of *Revolve*'s assumptions is that there is no overhead associated with calling the forward and reverse operators. Another way to state this assumption is that calling an operator k times for a single timestep and calling it once for k timesteps should take the same amount of time. This is an important assumption for *Revolve* since the adjoint computation (as well as the associated forward computation) is carried out one timestep at a time.

While this granularity might be appropriate for some problems - especially those using implicit methods, this is perhaps too fine for an explicit method for the acoustic wave equation. Here, the individual timesteps are so small that the constant overheads of calling a Devito operator - preprocessing of the arguments, and any python-to-C overheads could become significant.

In the near future, Devito is likely to introduce a data-movement optimisation called *time-tiling*. This tries to minimise data movement by computing multiple timesteps in one batch. Clearly this optimisation will not be available when running one timestep at a time under *Revolve* and the impact of the missing optimisation can also be considered an operator-call overhead.

The simplest way to solve this is a method we already discussed in Section 4.7.2 and Section 4.7.4, which is to combine multiple (fine) timesteps into a single (coarse) timestep when reporting to *Revolve*. By decreasing the granularity of computational steps, this method will hide the overheads behind larger blocks of compute, while also allowing cross-timestep optimisations to function.

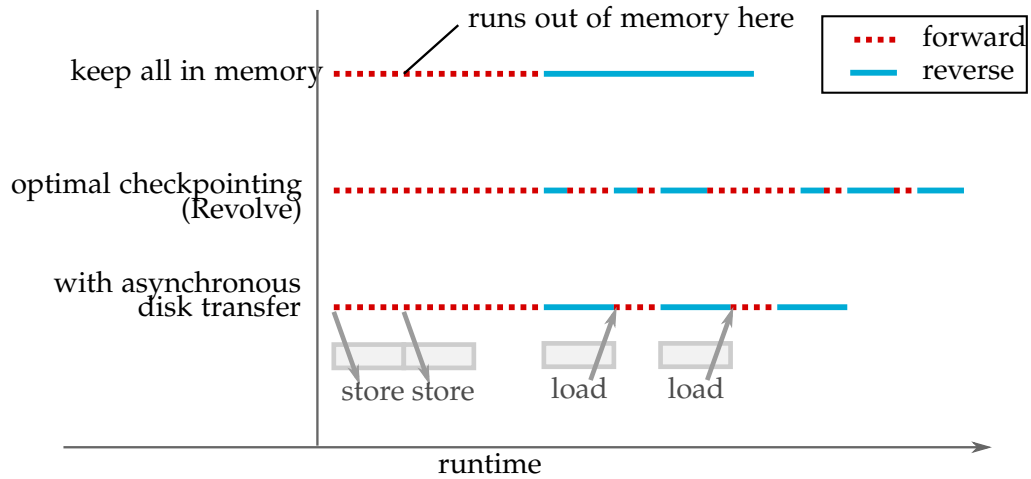


Figure 25: Timeline of events for conventional adjoint, *Revolve* checkpointing, and asynchronous multistage checkpointing.

4.7.6 Asynchronous checkpointing

In previous work we implemented a prototype for asynchronous multistage checkpointing [63]. In this, the checkpoints are always read from and written to the fastest memory, while the movement from the fastest to slower levels of memory happens asynchronously. Figure 25 shows the timeline of events in case of a computation run with asynchronous checkpointing. Our prototype was written for the Intel® Xeon Phi™ (Knights Landing) accelerator platform but the ideas could be adapted to a GPU implementation.

4.7.7 “Optimal” number of checkpoints

Under *Revolve*’s assumption of zero-cost checkpointing, there is no meaning to the term optimal number of checkpoints, since we should always use all available memory. However, when saving/loading a checkpoint comes at a cost, some recomputation is preferred to saving too many checkpoints since the program might end up spending too much time moving data from checkpoints - where it might have been quicker to use fewer checkpoints and more recomputation. An example that would demonstrate this clearly is if all available DRAM is used for the solver itself and the only checkpointing memory is available on a disk. In this scenario, saving all timesteps to disk would completely eliminate *Revolve* and this program would run at the speed of disk access - which is probably too slow.

Hence, when writing and reading checkpoints comes at a cost, it is not always optimal to use all available memory. Once *pyRevolve* supports multiple levels of memory, users might be tempted to use “too many” checkpoints leading to this issue. It would be useful for *pyRevolve* to calculate the maximal/optimal number of checkpoints for each available level of memory depending on read/write speeds and help the user with this information. Stumm and Walther [104] has a formulation and solution to this problem.

4.8 CHECKPOINTING BEYOND SEISMIC

The idea of checkpointing originated in the Algorithmic differentiation community where it is standard practice to “tape” the intermediate results of forward computations to compute the derivative as part of a “reverse-mode” pass. Hence, checkpointing is applicable wherever derivatives are calculated in the reverse mode.

FWI is comparable to other inverse problems like brain-imaging [39], shape optimisation [57], and even training a neural network.

A neural network can be seen as a generalisation of the computational graph shown in Figure 2, where the activations of neural units are to be stored during the forward pass, to be used during backpropagation. The major difference is that neural networks are not constrained to have the same checkpoint size or computational cost at different points in the network. Since both these facts violate the basic assumptions of *Revolve*, the original algorithm might not be optimal if used in its unmodified form.

The size of the corresponding computation in a neural network depends on the depth of the network and, more importantly, the input size. We assume the input is an image for this exposition. For typical input image sizes of less than 500×500 px, the computation per data point is relatively (to FWI) small, both in the number of operations and memory required. This is compensated by processing in *mini-batches*, where multiple data points are processed at the same time. This batch dimension’s size is adjusted to *fill up* the target hardware to its capacity (and no more). This is the standard method of managing the memory requirements of a neural network training pipeline.

However, for an input image that is large enough, or a network that is deep enough, it is seen that the input image, network weights, and network activations together require more memory than available on

a single node, even for a single input image ($batchsize = 1$). This is where checkpointing becomes necessary.

The idea of checkpointing for neural networks was first discussed in Chen et al. [20], which proposed a strategy that was not necessarily optimal. *Gradient checkpointing*, a package for Tensorflow implements the strategy set out in this paper, which reduces the memory footprint of any given computational graph that can be reduced to a chain-like graph (not necessarily uniform computation and the checkpoint size). Gruslys et al. [38] describes a dynamic programming formulation for this optimisation problem, not very different from the original *Revolve* work, that can solve any chain-like computational graph to optimality (without the uniformity assumptions). Extensions to *Revolve* have also been published in the context of neural networks [12]. In previous work, we proposed a use-case for checkpointing in neural networks [68] which has been adopted into a practical implementation [56]. The asynchronous checkpointing prototype for *pyRevolve* was also built with neural networks in mind [63].

4.9 FUTURE WORK

This work carried out so far was a proof of concept of integration with *Revolve* using high level abstractions and, as such, is still a work-in-progress in terms of use for practical applications.

The most important limitation in the current implementation is that it only supports in-memory checkpointing. Extending this to disk-aware checkpointing would be the highest-priority future work. Integration with other schedulers like *H-Revolve* would provide access to disk-aware optimal algorithms.

Another useful feature could be the ability to club multiple timesteps as a single one for checkpointing, as discussed in Section 4.7.2. As discussed previously, this would help hide many of the overheads that currently cause *pyRevolve* to be inefficient. Ideally, this would be accompanied by a performance model that can determine the best number of fine timesteps inside a coarse timestep.

We would also like to see *In-place operators*, as described in Section 4.7.4.1, to be implemented in Devito. This would eliminate any data-movement overheads we see in Devito-*pyRevolve* application code.

We would also like to improve our handling of higher-order time-stepping, as discussed in Section 4.7.3, that *pyRevolve* can then use

to eliminate the excess memory consumption that comes with the current implementation.

Further, a production-ready version of GPU-aware asynchronous checkpointing would be very useful, now that Devito has some initial GPU support.

For problems implemented with adaptive time-stepping, the number of time-steps is not known a-priori and that would require *pyRevolve* to implement online checkpointing, something that even Devito would require in future versions.

4.10 CONCLUSIONS

Through this work, we have shown that with high-level abstractions it is possible to greatly simplify the complexity of client code. We have also verified the correctness of our implementation using mathematical tests. This already enables the users of Devito to utilize *Revolve* based checkpointing in their applications to solve much bigger problems than previously possible. However, through the experiment in Section 4.6, we have seen that the overhead introduced by checkpointing is non-trivial. For this reason, there is much more work to be done to implement more features that widen the applicability of *pyRevolve*.

COMPRESSION

One accurate measurement is worth a thousand expert opinions.

—Grace Hopper

In previous chapters, we have seen the FWI problem, its implementation in Devito, the large memory footprint, and checkpointing as a solution to that footprint. In this chapter, we continue by first discussing some other methods of reducing the large memory footprint – specifically compression. We then introduce a new method that combines checkpointing and lossy compression to significantly reduce the memory footprint efficiently. This chapter was formed by merging the texts from Kukreja et al. [65] and Kukreja et al. [66].

5.1 INTRODUCTION

5.1.1 *Adjoint-based optimisation*

Like FWI, most adjoint-based optimisation problems consist of a simulation that is run forward in simulation time, producing data that is used in reverse order by a subsequent adjoint computation that is run backwards in simulation time. Many important numerical problems in science and engineering use adjoints and follow this pattern.

Since the data for each of the computed timestep in the forward simulation will be used later in the adjoint computation, it would be prudent to store it in memory until it is required again. However, the total size of this data can often run into tens of terabytes, exceeding the memory capacity of most computer systems of today. Previous work has studied recomputation or data compression strategies to work around this problem. In this chapter, we investigate a combination of compression and recomputation.

5.1.2 *Techniques used to alleviate the high memory pressure*

Many strategies have been studied to cope with the amount of data that occurs in adjoint computations - perhaps the simplest is to store

all data to a disk, to be read later by the adjoint pass in reverse order. However, the computation often takes much less time than the disk read and write, hence leaving disk speed as a bottleneck.

Domain decomposition, where a single gradient evaluation may be distributed across more than one node, is often used not only to distribute the computational workload across more processors, but also across more memory. While this strategy is very powerful, the number of compute nodes and therefore the amount of memory that can be used efficiently is limited, for example by communication overheads that start to dominate as the domain is split into increasingly small pieces. Secondly, this strategy can be wasteful if the need for memory causes more nodes to be used than can be completely utilised for computation.

Lastly, this method is not well suited for cloud-based setups since it can complicate the setup and performance will suffer due to the slow inter-node communication.

One standard approach is to save the field at only the boundaries and reconstruct the rest of the field from the boundaries to reduce the memory footprint. However, the applicability of this method is limited to time-reversible PDEs. Although Equation (3) (Chapter 2, and repeated in Section 5.2) is itself time-reversible, most of the variations used in practice are not. For this reason, we do not discuss this method in this chapter.

Another notable approach that significantly reduces the amount of memory is frequency-domain methods. However, this approach does not scale as well as time-domain approaches [60].

Hybrid approaches that combine time-domain methods, as well as frequency-domain methods, have also been tried [117]. However, this approach can be challenging because the application code must decide the user's discrete set of frequencies to achieve a target accuracy.

We already discussed checkpointing in the previous chapter. We discuss this strategy further in Section 5.5. Previous work has applied checkpointing to seismic imaging and inversion problems [25, 106]. An alternative is data compression, which is discussed in Section 5.3.

In this chapter, we extend the previous studies by *combining* checkpointing and compression. This is obviously useful when the data does not fit in the available memory even after compression, for example for very large adjoint problems, or for problems where the required accuracy limits the achievable compression ratios. Figure 26 shows a visual representation of the key idea here - the gradient com-

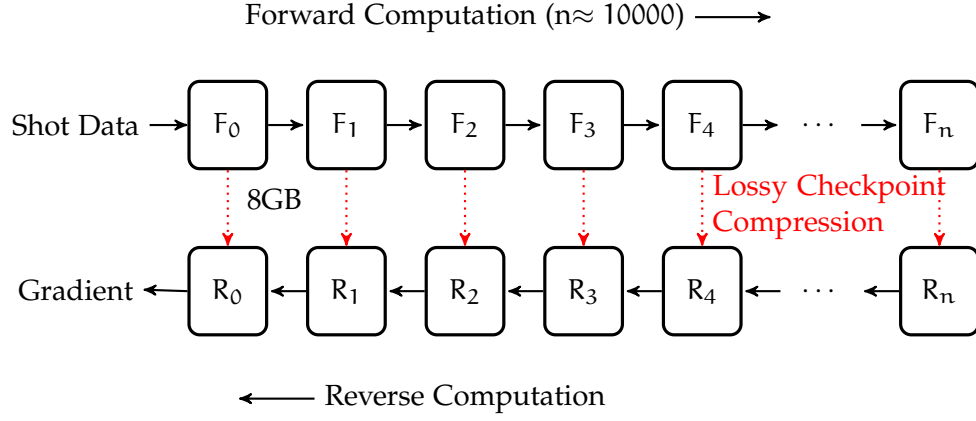


Figure 26: An illustration of the approach presented in this chapter. Checkpoints are compressed using lossy compression to combine lossy compression and the checkpoint-recompute strategies.

putation pattern introduces this high memory requirement that is alleviated by using lossy compression on the forward states. Figure ?? shows a visual depiction of the tradeoffs as we described them in this section.

Compared to the use of only checkpointing without compression, this combined method often improves performance. This is a consequence of the reduced size of stored timesteps, allowing more timesteps to be stored during the forward computation. This in turn reduces the amount of recomputation that needs to be performed. On the other hand, the compression and decompression itself take time. The answer to the question “does compression pay off?”, depends on many factors – including available memory, the required precision, the time taken to compress and decompress, and the achieved compression factors, and various problem-specific parameters like the computational intensity of the kernel involved in the forward and adjoint computations, and the number of timesteps.

Hence, the answer to the compression question depends not only on the problem one is solving (within seismic inversion, there are numerous variations of the wave equation that may be solved), but also the hardware specifics of the machine on which it is being solved. The answer might even change during the solution process of an individual problem. This brings up the need to predict whether compression pays off in a given scenario, without incurring significant overheads in answering this question. To this end, we present a performance model in Section 5.5 that answers this question.

Our performance model is agnostic of the specific optimisation problem being solved. We envision it being used in a generic checkpointing runtime that manages the checkpointed execution of an optimisation problem – like *pyRevolve*. We envision that *pyRevolve* will accept an acceptable error tolerance as an input parameter for each gradient evaluation and determine whether or not compression can pay off for that iteration.

The next question is of acceptable error tolerance. We note that there is some previous work in this area, discussing for example, the effect of bounded pointwise errors in a multi-dimensional field on computed numerical derivatives, for ZFP [1]. In the context of seismic inversion, other work discusses accuracy requirements in optimisation loops, and notes that high accuracy is only needed when already close to a minimum [14, 72]. There has been previous work on choosing the most appropriate compression algorithm based on statistical properties [108], and work that addresses convergence guarantees of trust-region based optimisation methods in the presence of gradients that are only known with a probability p . [13]

In this chapter, we shall conduct empirical experiments to measure the effect of the checkpoint error on the rest of FWI.

5.1.3 Summary of contributions

While checkpointing allows a trade-off between computational time and memory, compression allows a trade-off between memory and accuracy. In this work, we wish to combine these approaches into one three-way trade-off.

We will show that it is possible to accelerate adjoint-based computations, including FWI, by using lossy compression on the checkpoints. For a given checkpoint absolute error tolerance (*atol*), compression may or may not accelerate the computation. We shall use a performance model to help us answer this question a priori, i.e., without running any computations. Next, we evaluate how lossy compression impacts solver convergence and accuracy in FWI.

We conduct an empirical study of:

1. the use of different compression algorithms to seismic data, including six lossless and a lossy compression algorithm for floating-point data,

2. a performance model for checkpointing alone, taking into account the time taken to read and write checkpoints, and
3. an online performance model to predict whether compression would speed up an optimisation problem.
4. Propagation of errors when starting from a lossy checkpoint.
5. Effect of checkpoint errors on the gradient computation.
6. Effect of decimation/subsampling on the gradient computation.
7. Accumulation of errors through the stacking of multiple shots.
8. Effect of the lossy gradient on the convergence of FWI.

In the following sections, we start with a recap of FWI in Section 5.2 before we move on to discuss three techniques that are commonly used to alleviate this memory pressure in question - data compression in Section 5.3, other numerical approximations in Section 5.4, and checkpointing in Section 5.5. The common element in these techniques is that all three solve the problem of high memory requirement by increasing the operational intensity of the computation - doing more computations per byte transferred from memory. With the gap between memory and computational speeds growing wider as we move into the exaflop era, we expect the use of such techniques to increase moving forward.

In Section 5.5, we build a performance model based on previous work that can be used to estimate the runtime of an adjoint computation under *Revolve*. In Section 5.6, we extend this performance model to include compression. We follow this with a theoretical analysis of what that performance model predicts. In Section 5.7, we discuss the changes made in *pyRevolve* to support the experiments we do in this chapter.

In Section 5.8, we describe the setup we used for the experiments that we use to support the use of compression in FWI. This is followed by Results and Discussion in Section 5.9 and Conclusions and Future Work in Section 5.10.

5.2 RECAP OF FWI

A quick recap of the mathematics of FWI is necessary before we proceed with this chapter. FWI is an adjoint-based optimisation problem with the following objective function:

$$\underset{\mathbf{m}}{\text{minimize}} \Phi_s(\mathbf{m}) = \frac{1}{2} \|\mathbf{P}_r \mathbf{A}(\mathbf{m})^{-1} \mathbf{P}_s^\top \mathbf{q}_s - \mathbf{d}_{\text{obs}}\|_2^2, \quad (13)$$

where $\mathbf{d}_{\text{obs}}$ is the observed seismic data i.e. the reflected signals that were recorded at the receiver locations in the physical experiment, $\mathbf{q}_s$ is the source signal that was sent in the experiment, $\mathbf{P}_s^\top$ is the source-injection operator that injects the signal at the source location in the computational domain, $\mathbf{A}(\mathbf{m})^{-1}$ is the linearised operator derived from one of the wave equations, $\mathbf{P}_r$ is the receiver sampling operator that extracts the signals from the receiver locations in the wavefield, and $\mathbf{m}$ is the velocity model being solved for.

The wave equation represented as the linearised operator $\mathbf{A}(\mathbf{m})^{-1}$ can be one of many variants. Here we specify the simplest version, the isotropic acoustic equation:

$$\begin{cases} \mathbf{m} \frac{d^2 \mathbf{u}(\mathbf{x}, t)}{dt^2} - \nabla^2 \mathbf{u}(\mathbf{x}, t) = \mathbf{q}_s \\ \mathbf{u}(\cdot, 0) = 0 \\ \frac{d\mathbf{u}(\mathbf{x}, t)}{dt} \big|_{t=0} = 0 \end{cases} \quad (3)$$

Here, $\mathbf{m}$ again represents the velocity model, albeit in the form of inverse squared speed, $\mathbf{u}(\mathbf{x}, t)$ is the wavefield at location $\mathbf{x}$ and time t , and $\mathbf{q}_s$ is the source term. The solution of this (or another wave equation) would be a function $\mathbf{A}(\mathbf{m}, \mathbf{q}_s)$ that allows us to simulate the signal received at the receiver locations for a given source signal $\mathbf{q}_s$ and velocity model $\mathbf{m}$.

This completes our computation of the objective function - which forms the forward pass of the overall computation. The backpropagation is the computation of the gradient of this objective function, which is required when solving this optimisation problem with a gradient-based method. The gradient can be calculated using the following relation:

$$\nabla \Phi_s(\mathbf{m}) = \sum_{t=1}^{n_t} \mathbf{u}_{tt}[\mathbf{t}] \mathbf{v}[\mathbf{t}] = \mathbf{J}^\top \delta \mathbf{d}. \quad (29)$$

Here, $\mathbf{u}_{tt}$ is the second-time-derivative of $\mathbf{u}$, the wavefield from Equation (3), and $\mathbf{v}$ is the solution of the adjoint equation of the one used for $\mathbf{u}$. As discussed in Chapter 2, the adjoint of Equation (3) is the same equation but this is not always true.

5.3 COMPRESSION ALGORITHMS

Compression or bit-rate reduction is a concept originally from signal processing. It involves representing information in fewer bits than the original representation. Since there is usually some computation required to go from one representation to another, compression can be seen as a memory-compute tradeoff, where extra computations are used to reduce the data's memory footprint.

Perhaps the most commonly known and used compression algorithm is ZLib (from GZip) [26]. ZLib is a lossless compression algorithm, i.e., the data recovered after compressing-decompressing is an exact replica of the original data before compression. Although ZLib is targeted at text data, which is one-dimensional and often has predictable repetition, other lossless compression algorithms are designed for other kinds of data. One example is FPZIP [75], which is a lossless compression algorithm for multidimensional floating-point data.

For floating-point data, another possibility is lossy compression, where the compressed-decompressed data is not the same as the original data, but a close approximation. The precision of this approximation is often set by the user of the compression algorithm. Two popular algorithms in this class are SZ [27] and ZFP [74]. The introduction of these algorithms has renewed has significantly accelerated the use of lossy compression in scientific applications.

Lossless algorithms guarantee that the exact original data can be recovered during decompression, whereas lossy algorithms introduce an error, but often guarantee that the error does not exceed certain absolute or relative-error metrics. Typically, lossy compression is more effective in reducing the size of the data. Most popular compression packages offer various settings that allow a tradeoff between compression ratio, accuracy, and compression and decompression time.

Another difference we observed between lossless and lossy compression algorithms was that the lossless compression algorithms we evaluated tended to interpret all data as one-dimensional series only while SZ and ZFP, being designed for scientific data, take the dimen-

sionality into account directly. This makes a difference in the case of a wavefield, for example, where the data to be compressed corresponds to a smoothly varying function in (two or) three dimensions and interpreting this three-dimensional data as one-dimensional would completely miss the smoothness and predictability of the data values.

It is worth noting that another data reduction strategy is to type-cast values into a lower precision format, for example, from double precision to single precision. This can be seen as a computationally cheap lossy compression algorithm with a compression ratio of 2.

Perhaps counterintuitively, compression can not only reduce the memory footprint, but also speed up an application. Previous work has observed that the compression and decompression time can be less than the time saved from the reduction in data that needs to be communicated across MPI nodes or between a GPU and a host computer [91].

One way of using compression in adjoint-based methods is to compress all timesteps during the forward pass. If the compression ratio is sufficient to fit the compressed data in memory, compression can serve as an *alternate strategy* to checkpointing. Previous work has discussed this in the context of computational fluid dynamics [23, 86] and seismic inversion using compression algorithms specifically designed for the respective applications [14, 24]. Here, we wish to broaden the applicability so we use a generic floating-point lossy compression library that can work for these and other scientific computing applications. We also increase applicability by combining compression with checkpointing.

Since the time spent on compressing and decompressing data is often non-negligible, this raises the question of whether the computational time is better spent on this compression and decompression, or the recomputation involved in the more traditional checkpointing approach. Cyr, Shadid, and Wildey [23] performs experiments to answer the performance question as well as the propagation of errors like we do here, but their study used compression methods like PCA in the context of the *Navier-Stokes* equation. They also do not consider the combination of checkpointing and compression in a single strategy. We discuss this in more detail in Section 5.6.

5.3.1 Lossless Compression

Blosc is a library that provides optimised high-performance implementations of various lossless compressors, sometimes beyond their corresponding reference implementations [3]. For our experiments, we use this library through its python interface. The library includes implementations for six different lossless compression algorithms, namely ZLIB, ZSTD, BLOSC LZ, LZ4, LZ4HC and Snappy. All these algorithms look at the data as a one-dimensional stream of bits and at least the blosc implementations have a limit on the size of the one-dimensional array that can be compressed in one call. Therefore we use the python package *blosc-pack*, which is a wrapper over the blosc library, to implement *chunking*, i.e. breaking up the stream into chunks of a chosen size, which are compressed one at a time.

5.3.2 Lossy Compression

We use the lossy compression package ZFP [74] written in C. To use ZFP from python, we developed a python wrapper for the reference implementation of ZFP ¹. ZFP supports three compression modes, namely fixed tolerance, fixed precision and fixed rate ². The fixed-tolerance mode limits the absolute error, while the fixed-precision mode limits the error as a ratio of the range of values in the array to be compressed. The fixed-rate mode achieves a guaranteed compression ratio requested by the user, but does not provide any bounds on accuracy loss.

The fixed-rate mode could make our implementation more straightforward by offering a predictable size of compressed checkpoints, but the lack of error bounds makes this option less attractive. Moreover, ZFP claims to achieve the best "compression efficiency" in the fixed-tolerance mode, and we thus chose to focus on this mode.

SZ [28] is a more recently developed compression library, also focussed on lossy compression of floating-point scientific data, also developed in C. While we have also written a python wrapper for the reference implementation of SZ ³, a thorough comparison of ZFP and SZ remains future work.

¹ Available open source at <https://github.com/navjotk/pyzfp>

² ZFP later released a version that supports lossless compression as well. We did not try this version during this work.

³ Available at <https://github.com/navjotk/pysz>. This is less mature than pyzfp.

5.4 OTHER APPROXIMATE METHODS

There has been some recent work on alternate floating-point representations [19], although we are not aware of this technique being applied to FWI. Within FWI, many approximate methods exist, including *On-the-fly Fourier transforms* [117]. However, it is not clear whether this method can provide fine-tuned bounds on the solution’s accuracy. In contrast, other completely frequency-domain formulations can provide clearer bounds [72], but as previously discussed, this comes at the cost of a much higher computational complexity. In this chapter, we restrict ourselves to time-domain approaches only.

Another approximation commonly applied to reduce the memory pressure in FWI in the time domain is subsampling. Here, the computation for Equation (29) is split into two parts where $\nabla\Phi_s(\mathbf{m})$ is only calculated for one in n evaluations of v . This reduces the memory footprint by a factor of n , since only one-in- n values of u need to be stored. The Nyquist theorem is commonly cited as the justification for this sort of subsampling. However, the Nyquist theorem only provides a lower bound on the error - it is unclear whether an upper bound on the error has been established on this method. Although more thorough empirical measurements of the errors induced in subsampling have been done before [78], we do a brief empirical study in Section 5.9.9 as a baseline to compare the error with our method.

5.5 CHECKPOINTING

As discussed in Chapter 4, checkpointing is a strategy to store selected timesteps, and recompute others when needed. The question of “which checkpoints should be stored to get the best tradeoff between recomputation time and memory footprint?” was answered in a provably optimal way by the *Revolve* checkpointing algorithm [36]. *Revolve* makes certain assumptions that we discussed in Chapter 4, including that uniform computational cost, uniform checkpoint size and a single level of high-speed memory. Other authors have subsequently developed extensions to *Revolve* that are optimal under different conditions [8, 104]. We focus in this chapter on the classic *Revolve* algorithm, and store all checkpoints in DRAM.

In the previous chapter, we introduced the open-source software *pyRevolve*, a Python package that can automatically manage the checkpointing strategies under different scenarios with minimal modifica-

tion to the computations code [64]. In this chapter, we extend *pyRevolve* to integrate lossy compression.

The most significant advantage of checkpointing is that the numerical result remains unchanged by applying this technique. Note that this is only true when checkpointing is used by itself. When combined with a lossy technique that we are about to do in this chapter, there will of course be an error due to the lossy compression. Another advantage is that the increase in run time incurred by the recomputation is predictable.

In this section, we build on the ideas introduced in Griewank and Walther [37] to build a performance model that predicts the runtime of an adjoint computation using *Revolve* checkpointing. We call the time taken for the execution of a single forward computational step C_F and correspondingly, the time taken for a single backward step C_R . For a simulation with N timesteps, the minimum wall time required for the full forward-adjoint evaluation is given by

$$T_N = C_F \cdot N + C_R \cdot N. \quad (32)$$

If the size of a single timestep in memory is given by S , solving this adjoint problem without special techniques requires a memory of at least size $S \cdot N$. If sufficient memory is available, no checkpointing or compression is needed.

If the memory is smaller than $S \cdot N$, *Revolve* provides a strategy to solve for the adjoint field by storing a subset of the N total checkpoints and recomputing the remaining ones. The overhead introduced by this method can be broken down into the recomputation overhead O_R and the storage overhead O_S . The recomputation overhead is the amount of time spent in recomputation, given by

$$O_R(N, M) = p(N, M) \cdot C_F, \quad (33)$$

where $p(N, M)$ is the minimum number of recomputed steps from Griewank and Walther [36], given as

$$p(N, M) = \begin{cases} N(N-1)/2, & \text{if } M = 1 \\ \min_{1 \leq \tilde{N} \leq N} \{\tilde{N} + p(\tilde{N}, M) + p(N - \tilde{N}, M - 1)\}, & \text{if } M > 1 \end{cases} \quad (34)$$

where M is the number of checkpoints that can be stored in memory. Note that for $M \geq N$, O_R would be zero. For $M < N$, O_R grows rapidly as M is reduced relative to N .

In an ideal implementation, the storage overhead O_S might be zero, since the computation could be done “in-place”, but in practice, checkpoints are generally stored in a separate section of memory and they need to be transferred to a “computational” section of the memory where the computation is performed, and then the results copied back to the checkpointing memory. This copying is a common feature of checkpointing implementations, and might pose a non-trivial overhead when the computation involved in a single timestep is not very large. This storage overhead is given by:

$$O_S(N, M) = W(N, M) \cdot \frac{S}{B} + R(N, M) \cdot \frac{S}{B}, \quad (35)$$

where $W(N, M)$ is the total number of times *Revolve* writes checkpoints for a run with N timesteps and M checkpoints, and similarly $R(N, M)$ is the number of times *Revolve* reads checkpoints for a run with N timesteps and M checkpoints, and B is the bandwidth at which these copies happen. We know that at the end of the forward computation, the data required for the first reverse step is already in memory. Every other reverse step must start by reading a checkpoint, which may or may not be followed by a recomputation of the forward to propagate this checkpoint in time before running the corresponding reverse step. This last statement can be written as:

$$R(N, M) = N - 1, \quad (36)$$

which implies that Equation (35) can be rewritten as:

$$O_S(N, M) = W(N, M) \cdot \frac{S}{B} + (N - 1) \cdot \frac{S}{B}. \quad (37)$$

The total time to solution becomes

$$T_R = C_F \cdot N + C_R \cdot (N - 1) + O_R(N, M) + O_S(N, M). \quad (38)$$

5.6 PERFORMANCE MODEL INCLUDING COMPRESSION

By using compression, the size of each checkpoint is reduced and the number of checkpoints available is increased (M in Equation 34). This reduces the recomputation overhead O_R , while at the same time

adding overheads related to compression and decompression in $\mathbf{O}_S$. To be beneficial, the reduction in $\mathbf{O}_R$ must offset the increase in $\mathbf{O}_S$, leading to an overall decrease in the time to solution T .

Our performance model assumes that the compression algorithm behaves uniformly across the different time steps of the simulation, i.e. that we get the same compression ratio, compression time and decompression time, no matter which of the N possible checkpoints we compress/decompress. The storage overhead now becomes

$$\begin{aligned} \mathbf{O}_S(N, M) = & \mathbf{W}(N, M \cdot F) \cdot \left(\frac{S}{F \cdot B} + t_c \right) + \\ & (N - 1) \cdot \left(\frac{S}{F \cdot B} + t_d \right), \end{aligned} \quad (39)$$

where F is the compression ratio (i.e. the ratio between the uncompressed and compressed checkpoint), and t_c and t_d are compression and decompression times, respectively. At the same time, the recomputation overhead decreases because F times more checkpoints are now available.

To study the performance model, we first visualize it along the axis of available memory, comparing the predicted performance of the chosen compression scheme with the predicted performance of a *Revolve*-only adjoint implementation. This is shown in Figure 27 where we can distinguish three different scenarios, depending on the amount of available memory.

1. If the memory is insufficient even with compression to store the entire trajectory, one can either use checkpointing only, or combine checkpointing with compression. This is the left section of the figure.
2. If the available memory is not sufficient to store the uncompressed trajectory, but large enough to store the entire compressed trajectory, we compare two possible strategies: Either use compression only, or use checkpointing only. This is the middle section of the figure.
3. If the available system memory is large enough to hold the entire uncompressed trajectory, neither compression nor checkpointing is necessary. This is the right section of the figure.

The third scenario does not need any special techniques. The second scenario was studied in previous work [23] with compression only. Our method is applicable not only for the second, but also the

first scenario which previously was only possible to solve using checkpointing alone.

We can identify many factors that make compression more likely to be beneficial compared to pure checkpointing: A very small system memory size and a large number of time steps lead to a rapidly increasing recompute factor, and compression can substantially reduce this recompute factor. This can be seen in Figures 27 and 29.

The extent to which the recompute factor affects the overall runtime also depends on the cost to compute each time step. If the compute cost per time step is large compared to the compression and decompression cost, then compression is also likely to be beneficial, as shown in Figure 28. As the time per time step increases and the compression cost becomes negligible, we observe that the ratio between the runtime of the combined method and that of pure checkpointing is only determined by the difference in recompute factors.

5.7 EXTENDING *pyrevolve* TO SUPPORT COMPRESSION

The first step in adding support for compression in *pyRevolve* was to write the python wrapper *pyzfp*. Next, we added default configuration values for each compressor supported since it was important to have meaningful defaults if we were to experiment with only *atol* values, for example. *pyRevolve* represents the compressor/decompressor function, along with its default values as a partially evaluated function.

It can be seen that compression will inevitably require some data movement as the uncompressed data is read and the compressed version written. In order to not introduce extra overheads due to this movement, it is essential to carry out compression as part of an unavoidable data movement step. A logical way to do this is to compress data as it is being copied from the operator memory to the checkpointing memory, and decompress when it is being copied from the checkpointing memory to the operator memory.

Since, in the API discussed in Chapter 4, the copy is implemented by the user of *pyRevolve*, implementing compression would require *pyRevolve* to pass the compression partially evaluated function to user code, and trust that the user will call the compression function where they wanted to carry out the copying of the data.

In Section 4.7.4, we noticed that the function used to copy the data in the checkpointing implementation is probably the single biggest

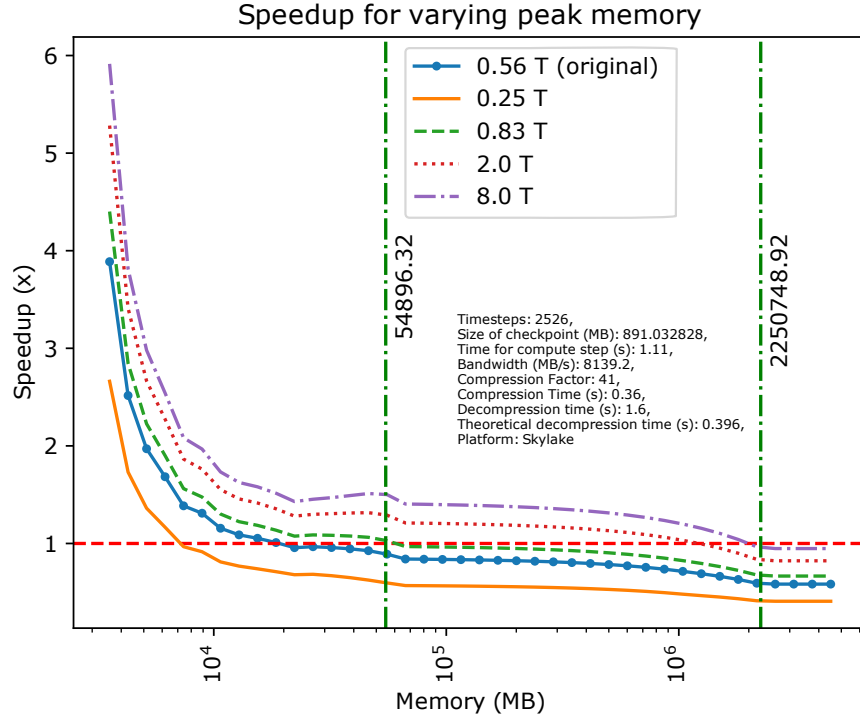


Figure 27: The speedups predicted by the performance model for varying memory. The baseline (1.0) is the performance of a *Revolve*-only implementation under the same conditions. The different curves represent kernels with differing compute times (represented here as multiples of T , where T is the time taken to compress and decompress a single timestep). The original isoacoustic compute kernel took about half the time to compute as compared to compressing/decompressing a single timestep ($0.56T$). The next two are hypothetical kernels chosen to be slightly slower and slightly slower than the original kernel. The first vertical line at 55GB marks the spot where the compressed wavefield can completely fit in memory and *Revolve* is unnecessary if using compression. The second vertical line at 2.2 TB marks the spot where the entire uncompressed wavefield can fit in memory and neither *Revolve* nor compression is necessary. The region to the right is where these optimisations are not necessary or relevant. The middle region has been the subject of past studies using compression in adjoint problems. The region to the left is the focus of this chapter.

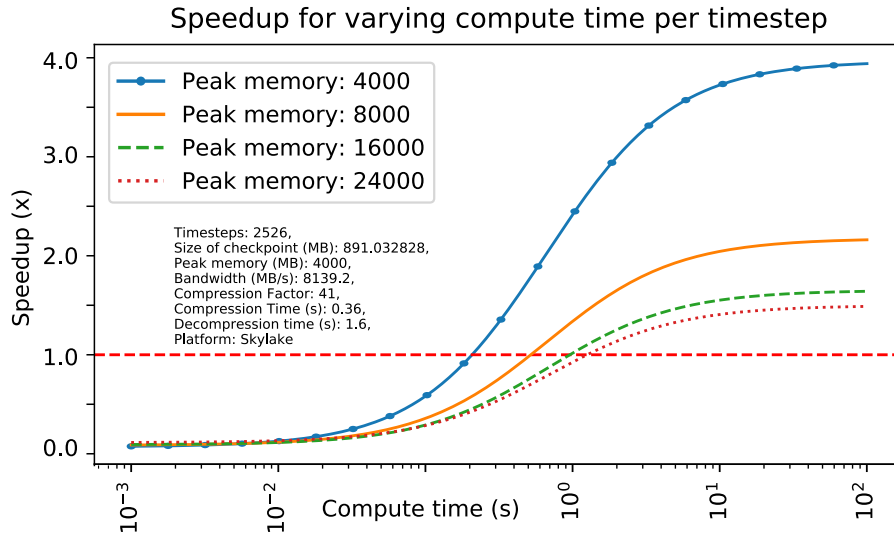


Figure 28: The speedups predicted by the performance model for varying compute cost. The baseline (1.0) is the performance of a *Revolve*-only implementation under the same conditions. The benefits of compression drop rapidly if the computational cost of the kernel that generated the data is much lower than the cost of compressing the data. For increasing computational costs, the benefits are bounded.

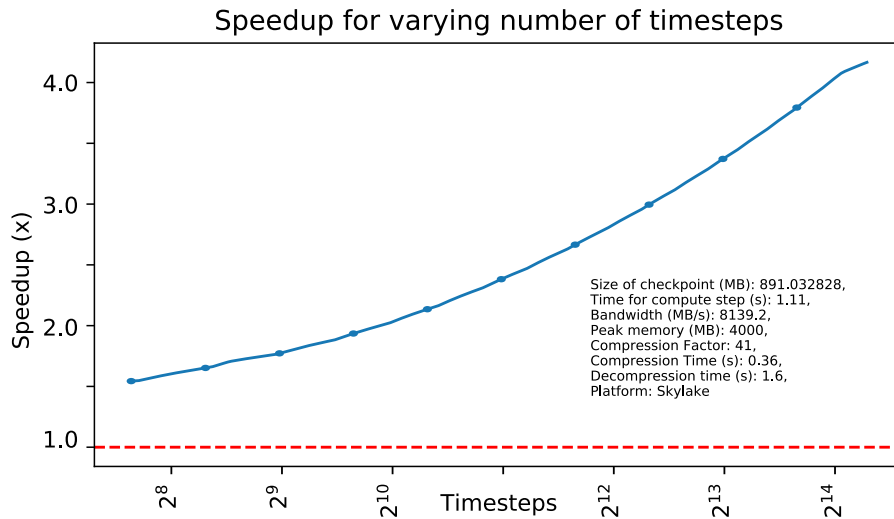


Figure 29: The speedups predicted by the performance model for varying number of timesteps. It can be seen that compression becomes more beneficial as the number of timesteps is increased.

factor controlling the overheads. Combining these two purposes, we propose that it might be better to change the API so that *pyRevolve* implements the data movement internally. Keeping this in mind, it would use a known good *memcpy* when not using compression, and the appropriate compression partial function when using compression.

For this work, we used a variation of the *pyRevolve* API where the user's checkpoint object provides pointers to the data location and the copying itself is directly managed by *pyRevolve*.

5.8 EXPERIMENTAL SETUP

REFERENCE PROBLEM We use Devito (see Chapter 3)[67, 84] to setup an acoustic wave propagation experiment. We call the kernel derived in Figure 6, the OT2 kernel because it is second-order accurate in time. We also use another formulation from Louboutin et al. [81], which is 4th-order accurate in time. We call this the OT4 kernel.

The velocity model was initialised using the SEG Overthrust model. This velocity model was then smoothed using a Gaussian function to simulate a starting guess for a complete FWI problem. The original domain was surrounded by a 40 point deep absorbing boundary layer. This led to a total of $287 \times 881 \times 881$ grid points. This was run for 4000ms with a step of 1.75ms, making 2286 timesteps. The uncompressed size of a single time step was just under 900MB in double precision. If one were to store all the timesteps, this would require 2.3TB of memory.

The spatial domain was discretised on a grid with a grid spacing of 20m, and the discretisation was 16th-order in space and second-order in time. We used 80 shots for our experiments with the sources placed along the x-dimension, spaced equally and just under the water surface. The shots were generated by modelling a Ricker source of peak frequency 8Hz. Following the method outlined in [94], we avoid inverse crime by generating the shots using a variation of Equation (3) that includes density, while using Equation (3) for inversion. The gradient was scaled by dividing by the norm of the original gradient in the first iteration. This problem solved in double precision is what we shall refer to as the *reference problem* in the rest of this chapter. Note that this reference solution itself has many sources of

error, including floating-point arithmetic and the discretisation itself.

EVOLUTION OF COMPRESSIBILITY We attempt to compress every timestep of the reference problem using the same compression setting and report on the achieved compression factor as a function of the timestep.

LOSSLESS COMPRESSION Based on the previous experiment, we choose a reference wavefield and compress it directly using a variety of compression settings. In this experiment, we experiment with a number of lossless compression algorithms and report the results.

DIRECT LOSSY COMPRESSION Here, we continue experimenting with the same reference wavefield. In this experiment we explore various settings in lossy compression. We report the errors comparing the lossy wavefield and the true reference wavefield.

PERFORMANCE MODEL VALIDATION In this experiment, we carry out gradient evaluations with a varying number of checkpoints. We use multiple computational kernels, and with compression enabled and disabled. We compare the time to solution predicted by our performance model and the measured time to solution. For benchmarking we used a dual-socket Intel®Xeon®Platinum 8180M @ 2.50 GHz (28 cores each) (Skylake) running Ubuntu 18.04 and ICC 2019.

FORWARD PROPAGATION In this experiment, we run the forward simulation for a few timesteps (about half the reference problem) and store it as a checkpoint. We then compress and decompress this through the lossy compression algorithm, getting two checkpoints - a reference checkpoint and a lossy checkpoint. We restart the simulation from each of these checkpoints and compare the two simulations' states and report on differences.

GRADIENT COMPUTATION In this experiment, we do the complete gradient computation, as shown in Figure 26 - once for the reference problem and a few different lossy settings. We report on the differences between these to show the propagation of errors.

STACKING In this experiment, we collate the gradient computed on multiple shots, and report the difference between the reference problem and the compressed version for this step.

CONVERGENCE In practice, FWI is run for only a few iterations at a time as a fine-tuning step interspersed with other imaging steps. Here we run a fixed number of FWI iterations (30) to make it easier to compare different experiments. To make this a practical test problem, we extract a 2D slice from the original 3D velocity model and run a 2D FWI instead of 3D. These experiments were run using the Dask/Kubernetes setup described in Chapter 3, on resources provided by Microsoft Azure. We compare the convergence trajectory with the reference problem and report.

SUBSAMPLING As a comparison baseline, we also use subsampling to reduce the memory footprint as a separate experiment and track the errors. The method is set up so that the forward and adjoint computations continue at the same time stepping as the reference problem above. However, the gradient computation is now not done at the same rate - it is reduced by a factor f . We plot results for varying f .

5.9 RESULTS AND DISCUSSION

5.9.1 *Evolution of compressibility*

To understand the evolution of compressibility, we compressed every timestep of a simulation to observe the evolution of compressibility through the simulation. For this, we chose ZFP in fixed tolerance mode at $atol = 10^{-4}$ (an arbitrary choice). We report the compression factors achieved in Figure 30. It can be seen that, in the beginning, the field is highly compressible since it consists of mostly zeros. The compressibility is worst towards the end of the simulation when the wave has reached most of the domain.

If the simulation had started with the field already oscillating in a wave, the compressibility curve for that simulation could be expected to be flat. This tells us that the compressibility of the last timestep of the solution is representative of the worst-case compressibility. Therefore we pick the last timestep as the reference for further experiments. A 2D cross-section of this snapshot is shown in Figure 31.

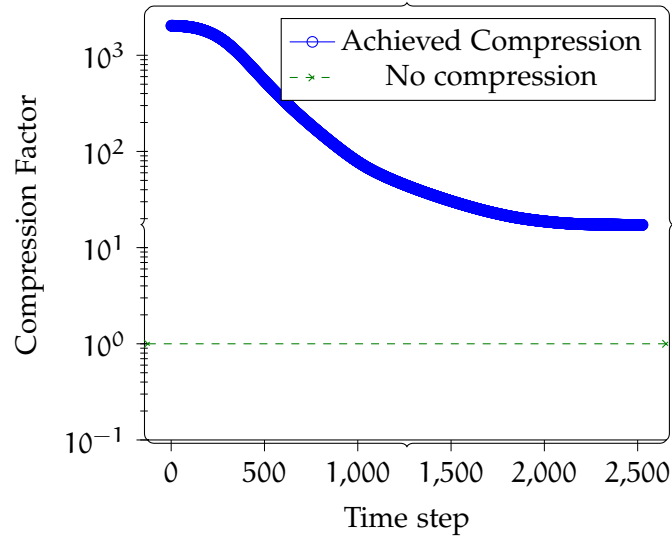


Figure 30: Evolution of compressibility through the simulation. We compressed every time step of the reference problem using an absolute error tolerance (*atol*) setting of 10^{-4} . The Compression factor achieved is plotted here as a function of the timestep number. Higher is more compression. The dashed horizontal line represents *no compression*. We can see that the first few timesteps are compressible to 1000x - since they are mostly zeros. The achievable compression factor drops as the wave propagates through the domain and seems to stabilize to 20x towards the end. We pick the last time step as the reference field for further experiments.

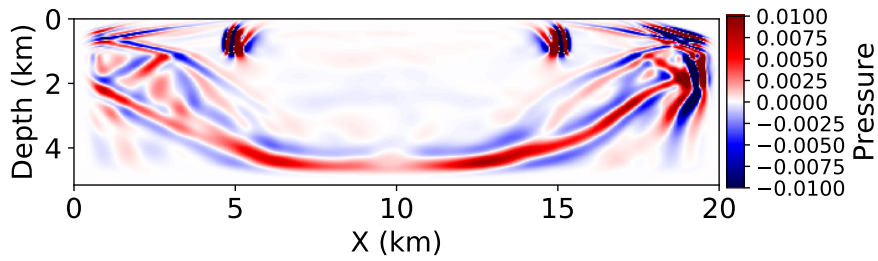


Figure 31: Reference wavefield for compression and decompression. This field was formed after a Ricker wavelet source was placed at the surface of the model and the wave propagated for 2500 timesteps. This is a vertical (x-z) cross-section of a 3D field, taken at the y source location.

Compressor	Chunk size(bytes)	Setting	Comp. time(ms)	Decomp. time(ms)	Comp. Factor
BloscLZ	1048576	6	4249.44	1288.86	1.188
LZ4	2965280	4	1371.26	920.98	1.199
LZ4HC	2097152	8	31245.16	926.69	1.265
ZLib	524288	7	30218.81	2470.04	1.291
ZStd	524288	9	117238.76	1477.34	1.312

Table 2: Some results from experimenting with all possible compressors and settings in blosc. We selected the best compression ratio seen for each compressor. "Setting" here is the choice between speed and compression, where 0 is fastest and 9 is highest compression. Blosc also provided a setting called *Shuffle mode*. We noticed our results were always better with this setting enabled than disabled.

5.9.2 Lossless compression

Table 2 shows the compression ratios and times for a few different lossless compressors and their corresponding settings. As can be seen, the compression factors achieved, and the time taken to compress and decompress can vary significantly, but it is hard to say whether this compression could be used to speed up the inversion problem.

5.9.3 Direct Lossy compression

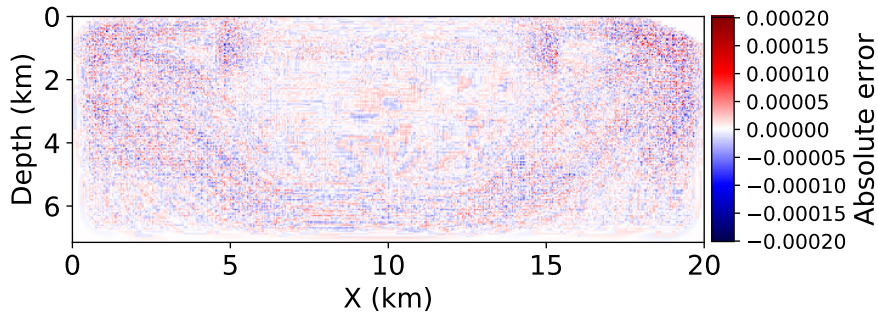


Figure 32: Direct lossy compression: Errors introduced during compression and decompression using the fixed-tolerance mode. It is interesting to note that the errors are more or less evenly distributed across the domain with only slight variations corresponding to the wave amplitude (from Figure 31). A small block-like structure characteristic of ZFP can be seen. *atol* was set to 10^{-3} - the highlighted point in Figure 33

To understand the direct effects of compression, we compressed the reference wavefield using a variety of absolute tolerance (*atol*) settings. Figure 33 shows compression ratios for different tolerance settings for

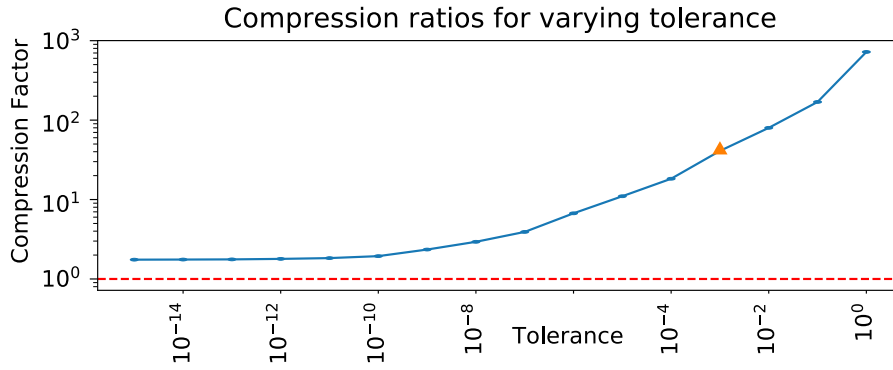


Figure 33: Direct lossy compression: Effect of tolerance settings of ZFP in fixed-tolerance mode on Compression ratio. We define compression factor as the ratio between the size of the uncompressed data and the compressed data. The dashed line represents no compression. The highlighted point is $atol = 10^{-3}$ and corresponds to the setting used for Figure 32

the fixed-tolerance mode of ZFP. The point highlighted here was the setting used for the results in Figure 32. Figure 32 shows the spatial distribution of the errors after compression and decompression, compared to the original field, for this setting. The error is a tensor of the same shape as the original field and results from subtracting the reference field and the lossy field. Figure 34 shows the Peak Signal-to-Noise Ratio achieved for each $atol$ setting. Figures 35 and 36 show some additional norms for this error tensor.

5.9.4 Performance model validation

To validate the *Revolve*-only performance model, Figure 37 shows the predicted runtime for a variety of peak memory constraints along with measured runtime for the same scenario. Figure 38 shows a comparison of predicted and measured runtimes for the OT2 kernel with compression enabled. Figure 39 repeats this experiment for the OT4 kernel which has a higher computational complexity. It can be seen that the model's prediction is close to the observed runtime in all three cases.

With this, we can establish that the performance model from Section 5.6 is useful for predicting the runtime of adjoint computations.

5.9.5 Forward propagation

Next, we executed the simulation for 500 steps and compressed the field's final state after these 500 steps. We then restarted the simu-

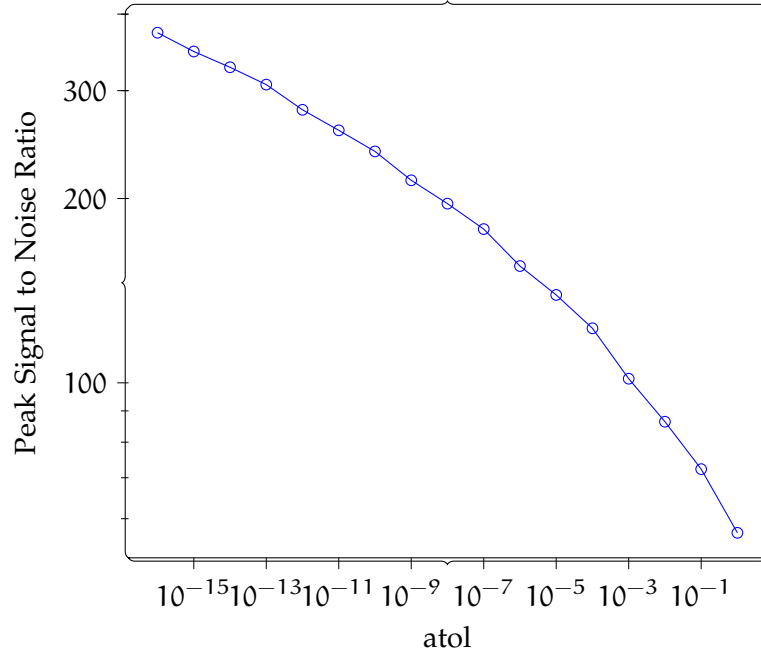


Figure 34: Direct lossy compression: We compress the wave field at the last time step of the reference solution using different $atol$ settings and report the Peak Signal to noise ratio (PSNR) achieved. Higher PSNR is lower error. The PSNR is very high for low absolute error tolerance ($atol$) and drops predictably as $atol$ is increased. See Figures 35 and 36 for more metrics on this comparison.

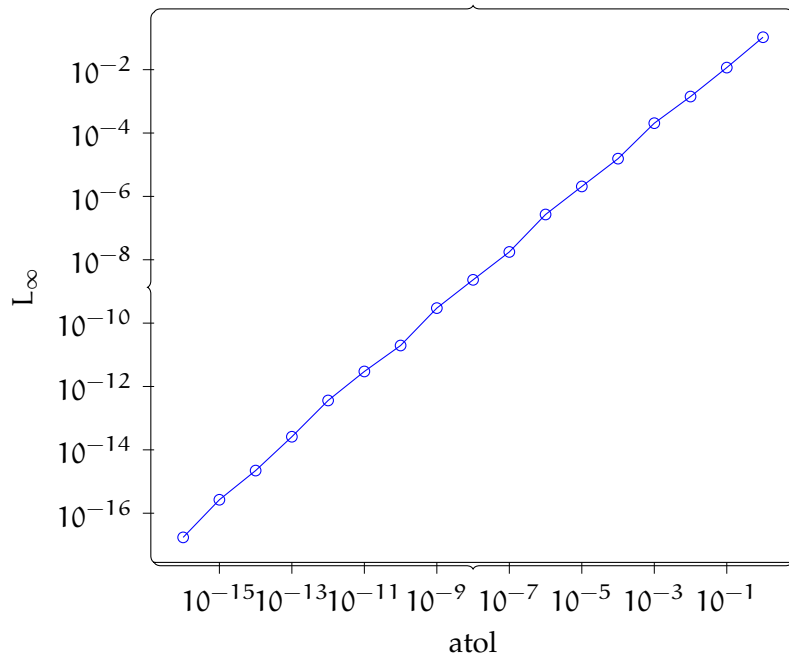


Figure 35: Direct lossy compression: L_∞ norm of error versus $atol$. This plot verifies that ZFP respects the tolerance we set.

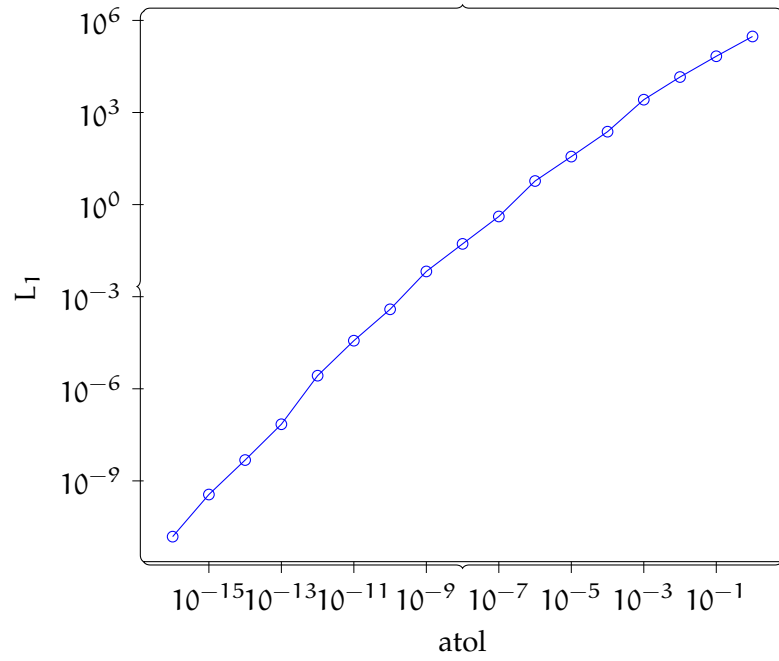


Figure 36: Direct lossy compression: L_1 norm of error versus $atol$. From the difference in magnitude between the L_∞ plot and this one, we can see how the error is spread across the domain.

Comparison of predicted and measured runtimes for OT2 kernel without compression

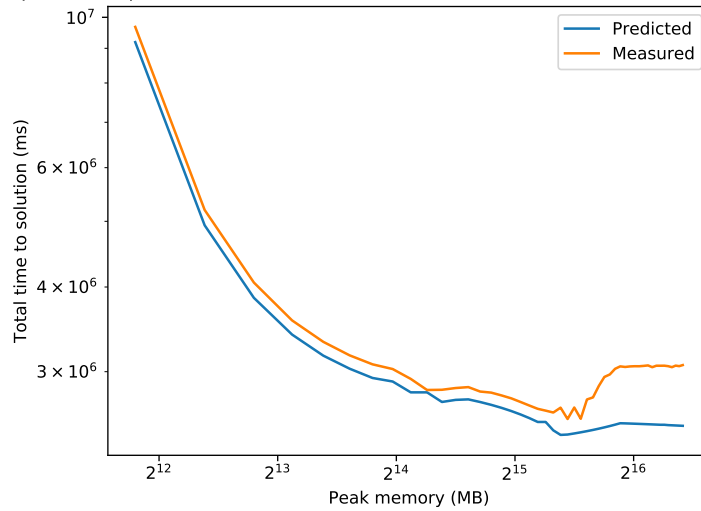


Figure 37: Predicted vs measured runtimes for the OT2 kernel, without compression.

Comparison of predicted and measured runtimes for OT2 kernel with compression

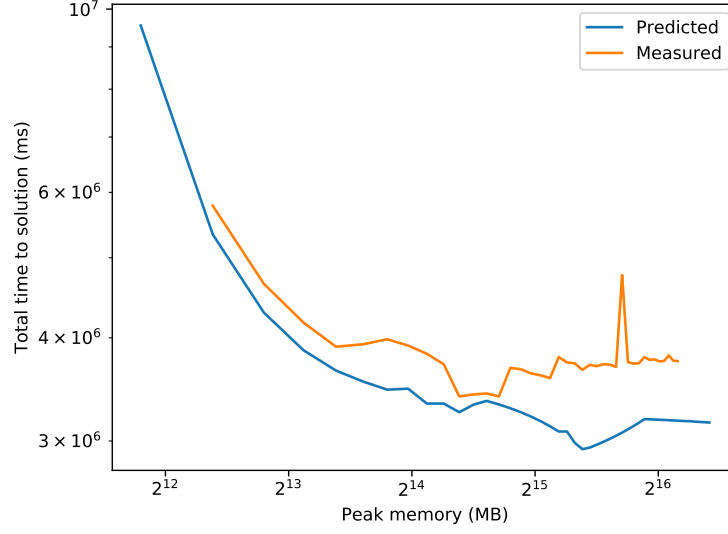


Figure 38: Predicted vs measured runtimes for the OT2 kernel, with compression enabled. The compression setting used was ZFP with $atol = 10^{-6}$

Comparison of predicted and measured runtimes for OT4 kernel with compression

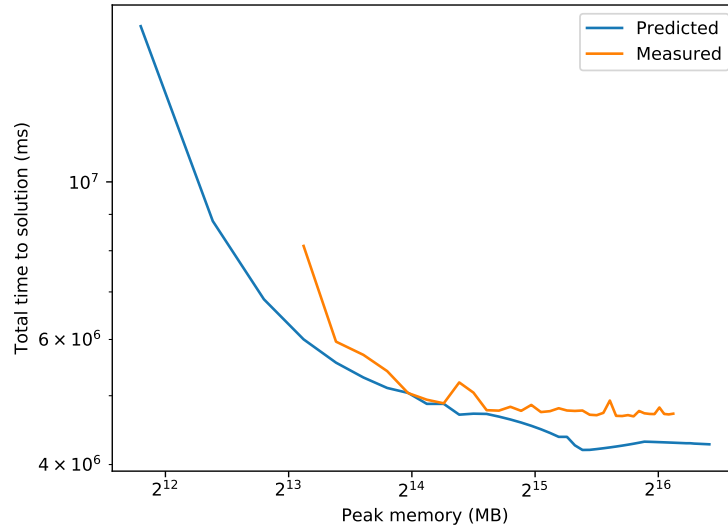


Figure 39: Predicted vs measured runtimes for the OT4 kernel, with compression enabled. The compression setting used was ZFP with $atol = 10^{-6}$

lation from step 500, comparing the progression of the simulation restarted from the lossy checkpoint vs. a *reference* simulation that was started from the original checkpoint.

Figure 40 shows the evolution of various error metrics as a function of the number of timesteps evolved. The L_∞ norm does not seem to be growing with the number of timesteps. This tells us that the maximum magnitude of pointwise error is not growing with the timesteps. However, we can see that the L_2 norm of the error is growing linearly. This could mean that the same magnitude of pointwise error is spreading to more parts of the domain - as part of the wave being modelled.

5.9.6 Gradient computation

Next, we measured the error in the gradient computation as a function of *atol*, assuming the same compression settings are used for all checkpoints.

Figure 41 shows the PSNR of the lossy gradient vs the reference gradient for varying settings of *atol*. Figure 42 shows the L_∞ and L_2 norms varying with *atol*. Apart from showing that the error in the gradient remains almost constant with changing *atol*, Figure 43 also shows that the number of timesteps does not appear to change the error by much (for a constant number of checkpoints). Figure 44 shows the L_∞ and L_2 norms of the gradient error as a function of the achieved compression factor (CF). This is provided for easy comparison of the performance of this method with subsampling. Figure 45 shows the Taylor linearisation errors of two different lossy settings vs the reference solution. Figure 46 shows the effect of the number of stored checkpoints on the gradient error.

It can be seen from the plots that the errors induced in the checkpoint compression do not propagate significantly until the gradient computation step. The *atol* compression setting does not seem to change the error in the gradient computation until a cutoff point. It is likely that the cross-correlation step in the gradient computation is acting as an error-correcting step since the adjoint computation continues at the same precision as before - the only errors introduced are in the values from the forward computation used in the cross-correlation step (the dotted arrows in Figure 26).

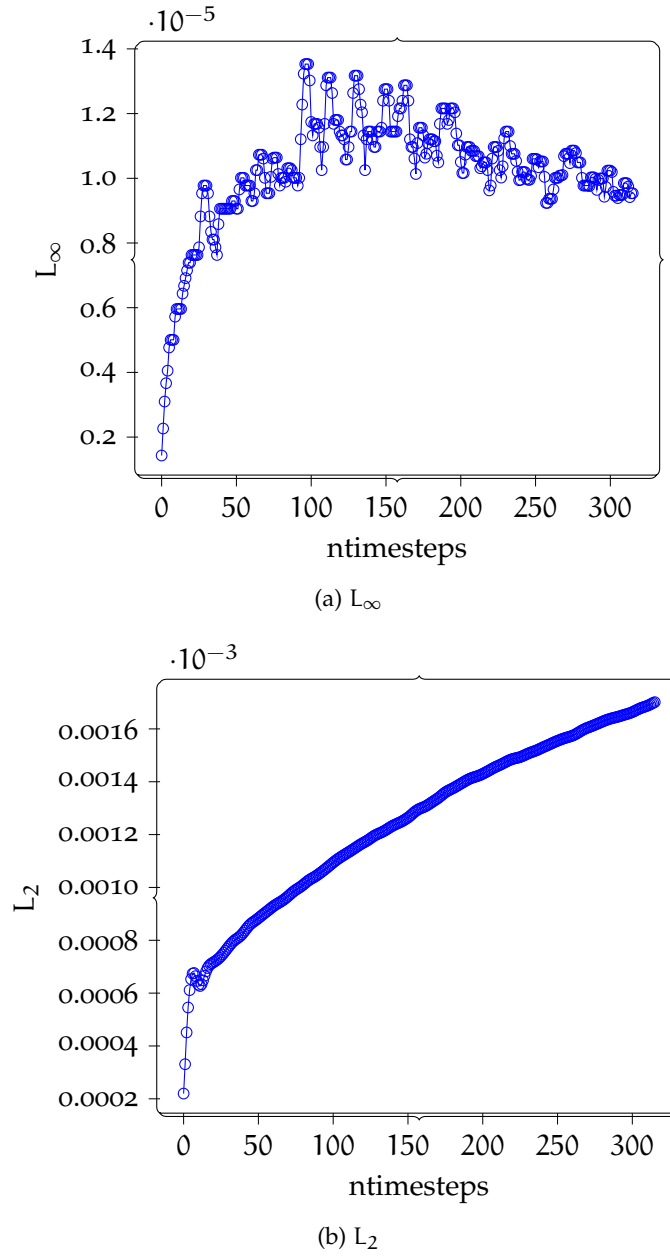


Figure 40: Forward propagation: We run the simulation for 500 timesteps. We then compress the state of the wavefield at this point using $atol = 10^{-4}$. We then continue the simulation from the lossy checkpoint and compare with the reference version. Here we report the L_∞ and L_2 norms of the error between the wavefields of these two versions as a function of the number of timesteps evolved from this lossy checkpoint. L_∞ initially grows sharply, but plateaus after about 150 timesteps. However, the growing L_2 norm tells us that the average error is going up possibly because the error is spreading around the domain without increasing in magnitude.

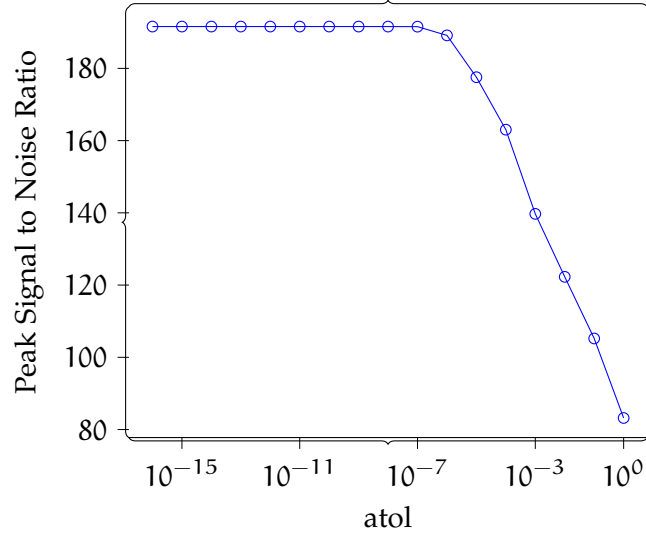


Figure 41: Gradient computation: In this experiment, we carry out the full forward-reverse computation to get a gradient for a single shot, while compressing the checkpoints at different $atol$ settings. This plot shows the PSNR of true vs lossy gradient as a function of $atol$ on the lossy checkpoints. We can see that the PSNR remains unchanged until about $atol = 10^{-6}$ and is very high even at very high values of $atol$.

5.9.7 Stacking

After gradient computation on a single shot, the next step in FWI is the accumulation of the gradients for individual shots by taking the mean of the shot-wise gradients into a single overall gradient. We call this stacking. In this experiment, we studied the accumulation of errors through this stacking process. Figure 47 shows the error in the gradient computation (compared to a similarly processed reference problem) as a function of the number of shots.

This plot shows us that the errors across the different shots are not adding up and the cumulative error is not growing with the number of shots - except for the compression setting of $atol = 10^{-1}$, which is chosen as an example of unreasonably high compression.

5.9.8 Convergence

Finally, we measure the effect of an approximate gradient on the convergence of the FWI problem. For reference, Figure 48 shows the known true velocity model for this problem as well as the final velocity model after running a *reference* FWI for 30 iterations. Figure 50

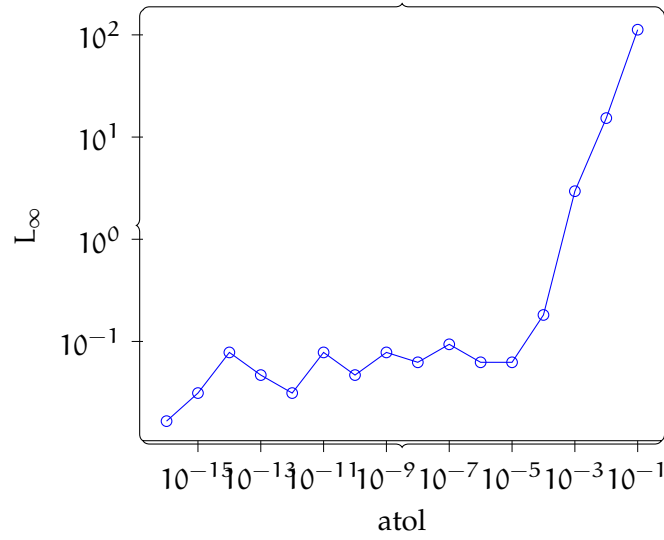
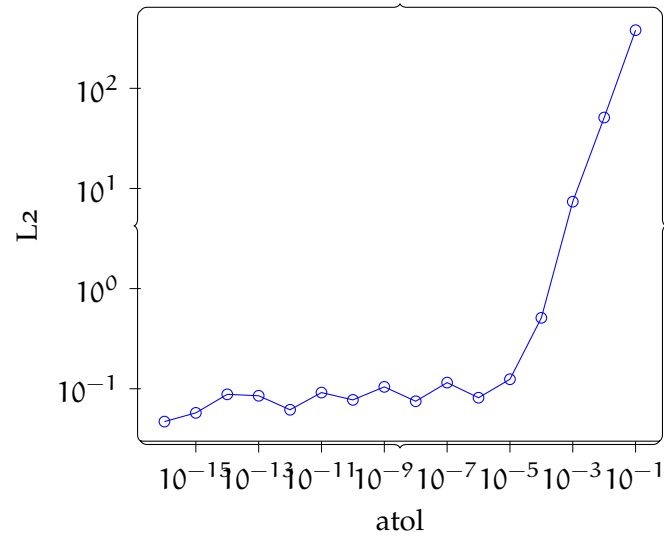
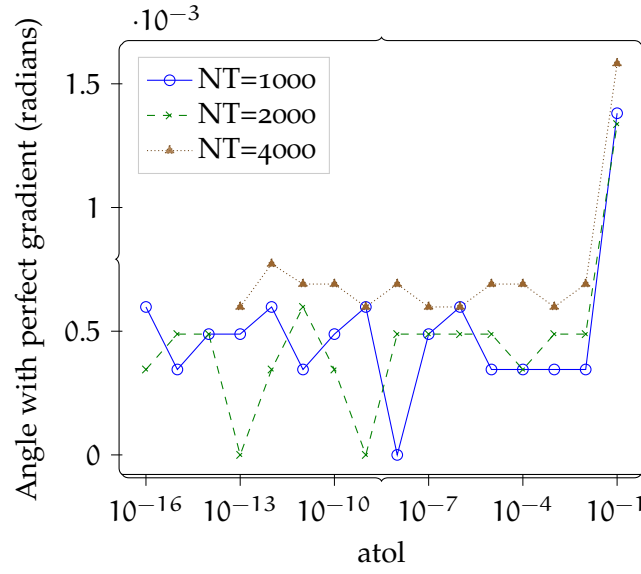
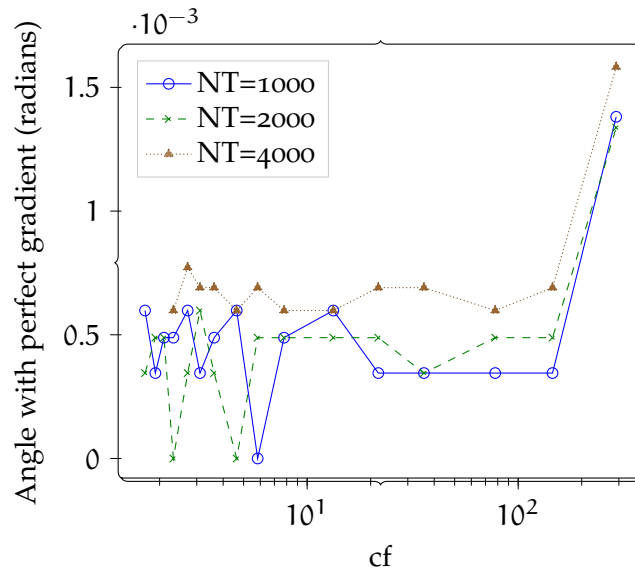
(a) L_∞ (b) L_2

Figure 42: Gradient computation: L_∞ and L_2 norms of gradient error versus $atol$. It can be seen that the error stays almost constant and very low up to a threshold value of 10^{-4}



(a) Angle vs atol



(b) Angle vs CF

Figure 43: Gradient computation: Angle between the lossy gradient vector and the reference gradient vector (in radians) vs *atol* and compression factor. If the lossy gradient vector was pointing in a significantly different direction as compared to the reference gradient, we could expect to see that on this plot. However, the angles seem too small to make a difference. The number of timesteps does not seem to affect the result by much. The results are also resilient to increasing *atol* up to 10^{-2} . Compression factors of over 100x do not seem to significantly distort the results either.

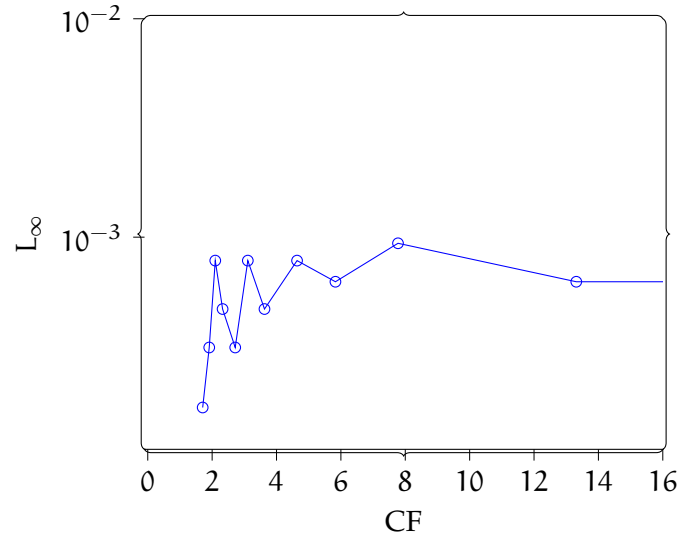
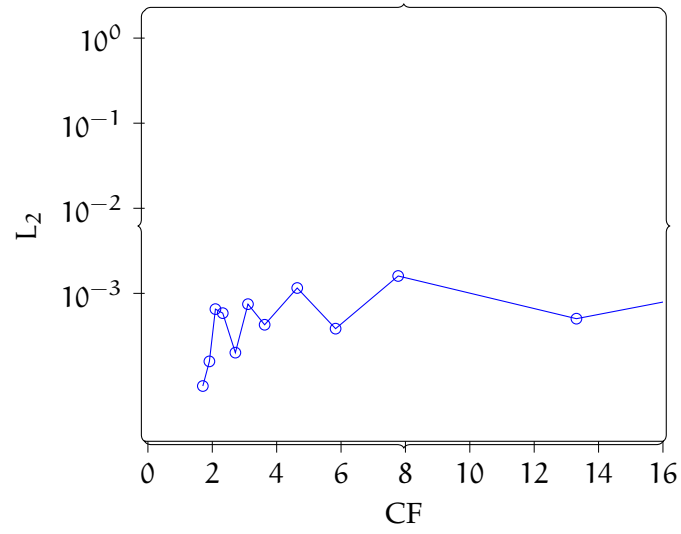
(a) L_∞ vs CF(b) L_2 vs CF

Figure 44: Gradient error: L_∞ and L_2 norms of the gradient error as a function of the achieved compression factor (CF). It can be seen that errors are negligible in the range of CF up to 16. Compare this to subsampling in Figure 51. Note that we achieved much higher CF values as part of the experiment but cut the axis in this figure to make it comparable to Figure 51

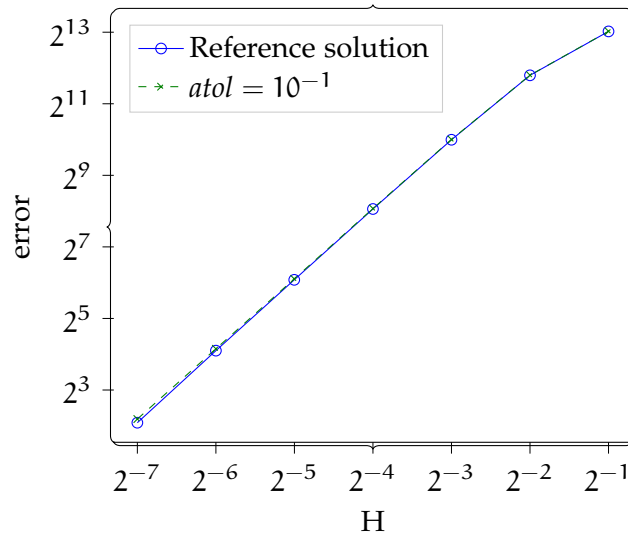
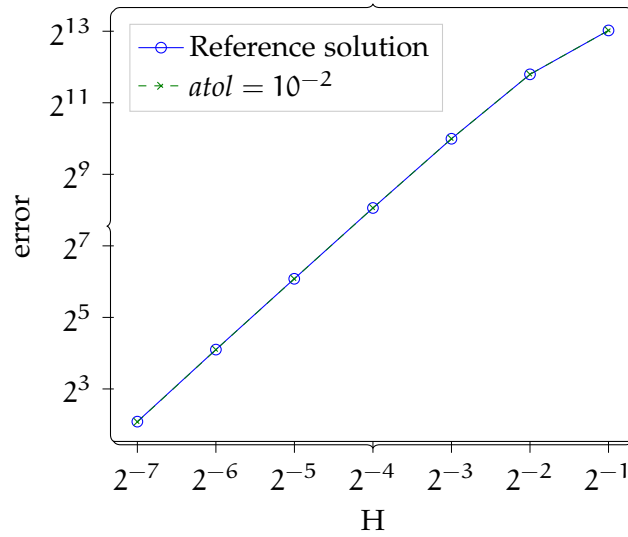
(a) 10^{-1} (b) 10^{-2}

Figure 45: Gradient linearisation: Comparison of gradient linearisation errors for two compression settings vs reference solution. H is a perturbation in the velocity model, and we measure the error induced as a function of H . The two curves in each of the plots follow each other so closely that they are indistinguishable. This confirms that the lossy gradients satisfy the Taylor-linearisation properties just as well as the reference gradient. Plots for smaller values of $atol$ look the same so we omit them here for brevity.

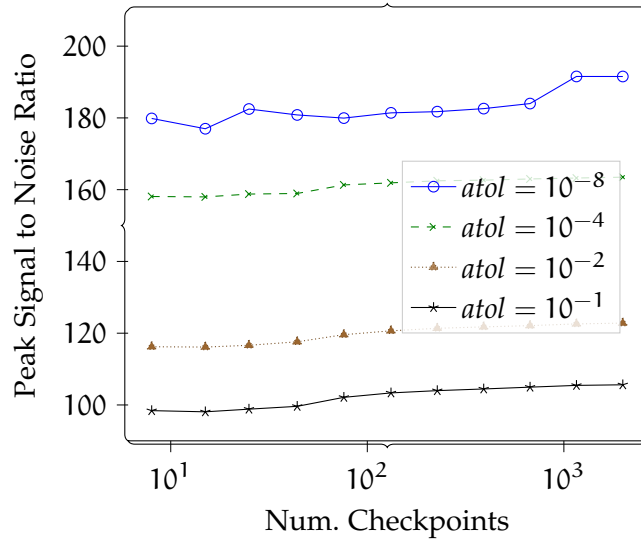


Figure 46: Gradient error: In this plot we measure the effect of varying number of checkpoints on the error in the gradient. We report PSNR of lossy vs reference gradient as a function of number of checkpoints, for four different compression settings.

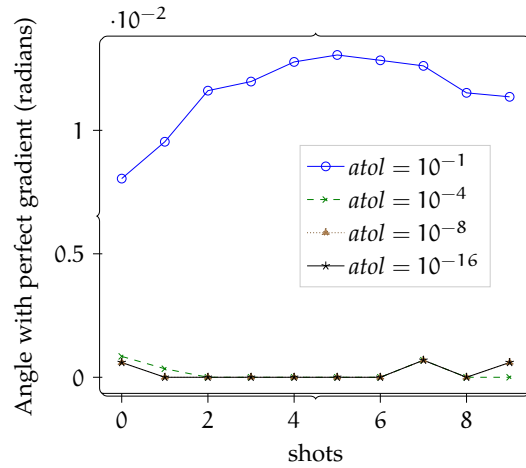
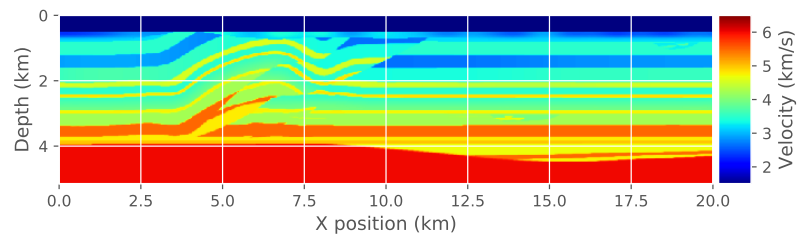
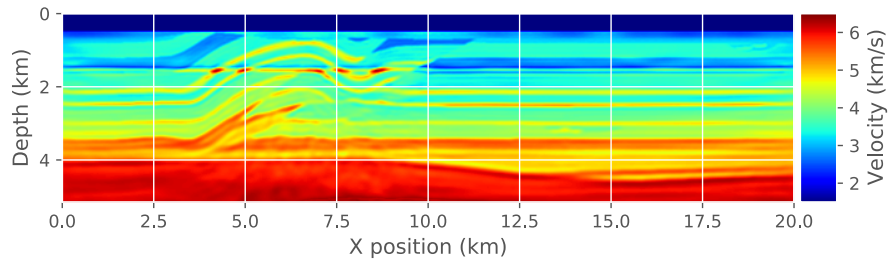


Figure 47: Shot stacking: The gradient is first computed for each shot and then added up for all the shots. In this experiment, we measure the propagation of errors through this step. This plot shows that while errors do have the potential to accumulate through the step - as can be seen from the curve for $atol = 10^{-1}$, for compression settings that are useful otherwise, the errors do not accumulate significantly.



(a) True model



(b) Final reference model

Figure 48: Reference solutions for the complete FWI problem. Above is the true solution and below is the solution after running *reference* FWI for 30 iterations

shows the final velocity model after running FWI with compression enabled at different *atol* settings - also for 30 iterations.

Figure 49 shows the convergence trajectory - the objective function value as a function of the iteration number. We show this convergence trajectory for 4 different compression settings. It can be seen that the compressed version does indeed follow a very similar trajectory as the original problem.

5.9.9 *Subsampling*

The standard assumption is that the gradient computation from Equation (29) is done at the same timestep rate as the forward and adjoint simulations. However, this computation can be done at a lower rate than the forward and adjoint simulations - this is called subsampling in time. To compare our proposed method with subsampling - which is sometimes used in industry, we run an experiment where we use subsampling in time to reduce the memory footprint. Figure 51 shows some error metrics as a function of the compression factor f .

Comparing Figure 51 with Figure 44, it can be seen that the proposed method produces significantly smaller errors for similar compression factors.

The results indicate that significant lossy compression can be applied to the checkpoints before the solution is adversely affected. This being an empirical study, we can only speculate on the reasons for this. We know that in the proposed method, the adjoint computation is not affected at all - the only effect is in the wavefield carried over from the forward computation to the gradient computation step. Since the gradient computation is a cross-correlation, we only expect correlated signals to grow in magnitude in the gradient calculation and when gradients are stacked. The optimisation steps are likely to be error-correcting as well since even with an approximate gradient ($atol > 10^{-4}$), the convergence trajectory and the final results do not appear to change much - indicating that the errors in the gradient might be cancelling out over successive iterations. There is even the possibility that these *errors* in the gradient introduce a kind of regularisation. The number of checkpoints has some effect on the error - more checkpoints incur less error for the same compression setting - as would be expected.

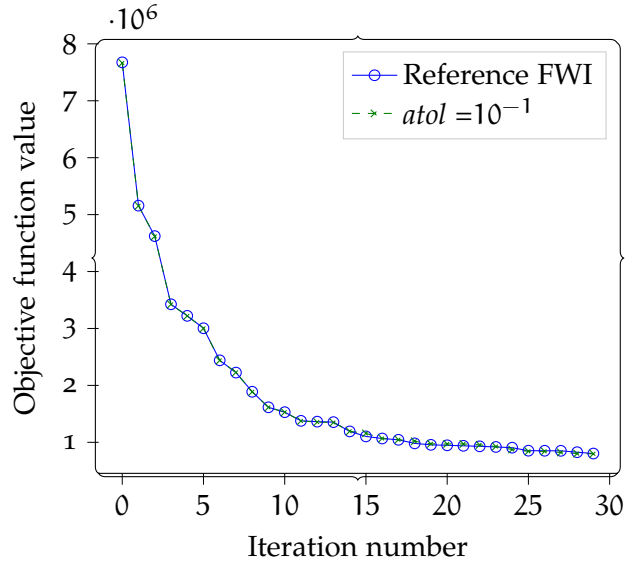
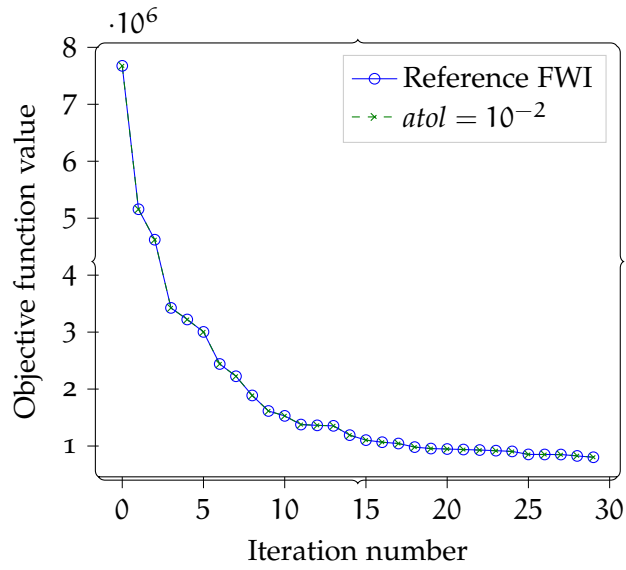
(a) $atol = 10^{-1}$ (b) $atol = 10^{-2}$

Figure 49: Convergence: As the last experiment, we run complete FWI to convergence (up to max 30 iterations). Here we show the convergence profiles of two lossy settings vs the reference problem. The reference curves are so closely followed by the lossy curve that the reference curves are hidden behind.

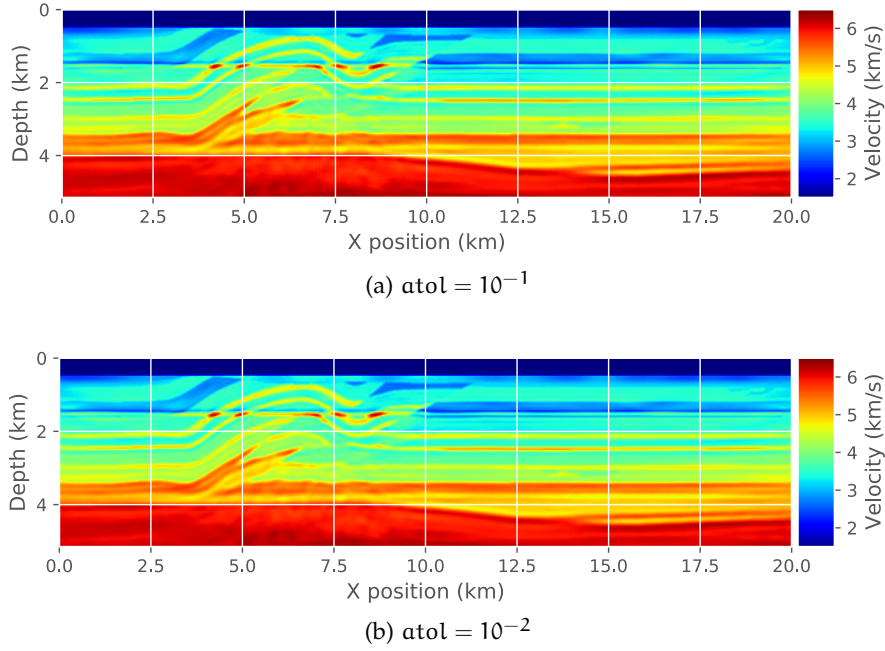
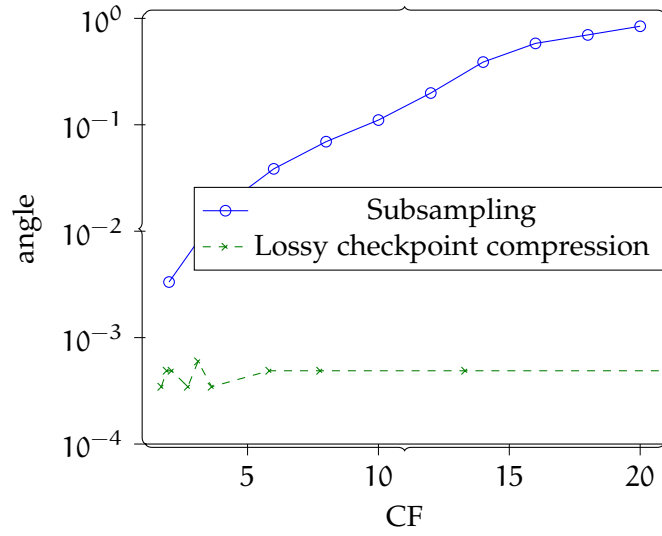


Figure 50: Final images after running FWI for two lossy settings. The two are visually indistinguishable from each other and from the reference solution in Figure 48.

5.10 CONCLUSIONS AND FUTURE WORK

In this chapter, we used compression to reduce the computational overhead of checkpointing in an adjoint computation used in seismic inversion. We developed a performance model that computes whether or not the combination of compression and checkpointing will outperform pure checkpointing or pure compression in a variety of scenarios, depending on the available memory size, the computational intensity of the application, the compression factor, and the throughput of the compression algorithm.

For the specific case of FWI, we have also shown that using lossy compression, high compression factors can be achieved without significantly impacting the convergence or final solution of the inversion solver. This is a very promising result for the use of lossy compression in FWI. The use of compression in large computations like this is especially important in the exascale era, where the gap between computing and memory speed is increasingly large. Compression can reduce the strain on the memory bandwidth by trading it off for extra computation - this is especially useful since modern CPUs are hard to saturate with computations that have a low operational intensity (OI), i. e. that do fewer computations per byte transferred.



(a) Angle

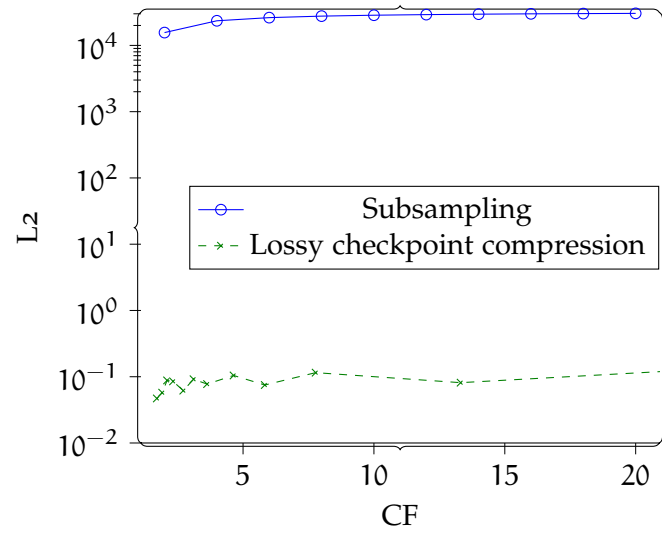
(b) L₂

Figure 51: Subsampling: We set up an experiment with subsampling as a baseline for comparison. Subsampling is when the gradient computation is carried at a lower timestepping than the simulation itself. This requires less data to be carried over from the forward to the reverse computation at the cost of solution accuracy so is comparable to lossy checkpoint compression. This plot shows L_∞ and L_2 norms of gradient error versus the compression factor CF for this experiment. Compare this to the errors in Figure 44 that apply for lossy checkpoint compression.

In future work, this could be extended by

- extending our performance model to support non-uniform compression ratios, as would be expected for example if the initial wave field is smoother and therefore more easily compressible,
- studying strategies where different compression settings (or even no compression) is used for a subset of time steps,
- exploring compression and multi-level checkpointing, including SSD or hard drives in addition to RAM storage,
- developing checkpointing strategies that are optimal even if the size of checkpoints post-compression varies and is not known a priori.
- studying the interaction between the propagation of lossy checkpoint errors and the velocity model, as well as the source frequency
- comparing the propagation of errors with other lossy compression algorithms e.g., SZ.

AUTOMATIC DIFFERENTIATION OF FINITE-DIFFERENCE SOLVERS

The purpose of computing is insight, not numbers.

—Richard Hamming

In the previous chapters, we discussed the memory footprint of backpropagation in FWI at length and proposed various solutions. As we discussed in Chapter 2, this memory footprint and the backpropagation is a characteristic feature of any derivative evaluation procedure. Through all of this, we assumed that we are given the formulation of a derivative which we then have to execute in a memory-efficient fashion.

However, requiring the users of Devito to specify the complete derivative formulation severely restricts the usage of Devito. To address this, many modern languages like PyTorch/Julia/JAX were built around the concept of Automatic Differentiation or Algorithmic Differentiation (AD). AD is a code generation technique where, given a mathematical function specified in code, we generate code for another function that computes the derivative of the original function.

During the course of this work, we also worked towards implementing this feature for Devito - we discuss this work in this chapter. It is important to note that Devito does finite-difference computation, which generates stencil operations in deeply nested loops. This chapter, a reproduction of Hückelheim et al. [50], addresses the problem of automatically generating derivatives for stencil loops. The conceptualisation of this paper, and the experiments, was done jointly by Jan and Navjot. Jan wrote most of the text of the paper while Navjot did the experimentation. All the text in this chapter is reproduced from the original publication with the exception of Section 6.6, which was written for this dissertation.

6.1 BACKGROUND

Derivatives are an important ingredient for optimisation, inverse modelling, error analysis, and related methods. Reverse-mode automatic

differentiation (AD) is used to compute these derivatives in applications including climate modelling [42], fluid dynamics [32], machine learning [11], and image processing [73]. Starting from an implementation of a differentiable function, referred to as *primal*, AD generates a new program that computes the derivative of that function. This is achieved by using operator overloading [35, 46], source-to-source [34, 40, 89, 111], or just-in-time compilation [53, 98].

In this chapter, we focus on the differentiation of *stencil loops*, which appear, for example, in convolutional neural networks and structured-grid PDE solvers. Stencil loops are characterised by a memory access pattern where each index in an output array is updated based on data gathered from a neighbourhood around that index in one or more input arrays. Stencil computations are straightforward to parallelise. For example, when OpenMP is used, each thread is guaranteed to have a different value for the counter in the parallel loop, and hence the write operations will occur at unique memory locations. The read operations, on the other hand, occur at neighbourhoods around that index, and more than one thread can read data from the same memory location. Techniques to execute stencils efficiently on multicore CPUs or GPUs have been studied extensively; see for example [59, 96].

We aim to apply reverse-mode automatic differentiation by source-to-source transformation to such stencil loops. We will refer to this approach as AD for the remainder of this work. AD, also known as backpropagation or adjoint differentiation, allows the computation of gradients with respect to an arbitrary number of function inputs, at a cost that is independent of the number of inputs. Thus, one can generate high-resolution seismic images, perform industrial-scale shape optimisation, or train large neural networks. The derivatives generated by AD are also sometimes called *adjoint* programs.

Reverse-mode AD traces derivatives backwards through a program, from the outputs to the inputs. In other words, the adjoint program is reading data from variables corresponding to the output of the primal program and is writing data to variables corresponding to the inputs of the primal. This *data flow reversal* also means that a concurrent read access in the primal program (i.e., a gather operation) can result in a concurrent write access in the derivative program (i.e., a scatter operation). Consequently, a parallelisation or vectorisation strategy that is valid for the primal program may not be valid for its derivative, and reverse-differentiating parallel programs without in-

roducing data races (while maintaining parallel efficiency) is a hard problem. Previous approaches work only for a small class of well-structured programs where array indices or pointer targets of input variables can be statically shown to be non-overlapping [31, 49].

To avoid data races and therefore undefined program behaviour, one could safeguard every potentially conflicting write access with atomic pragmas or critical sections. Depending on the hardware, this may cause the write updates to be sequentialised, reducing parallel efficiency. Additionally, such constructs incur overheads due to the necessary thread synchronisation. The cost of executing the atomic updates may slow the program execution significantly, as we demonstrate in the test cases in this chapter.

Another option is to use sum-reductions as implemented, for example, in the OpenMP runtime. These require a private copy of the output array for each thread, which significantly increases the memory footprint of the program. Other approaches include colouring schemes, which incur an overhead in computing the colouring, as well as introducing synchronisation barriers for each colour, thus reducing parallel performance.

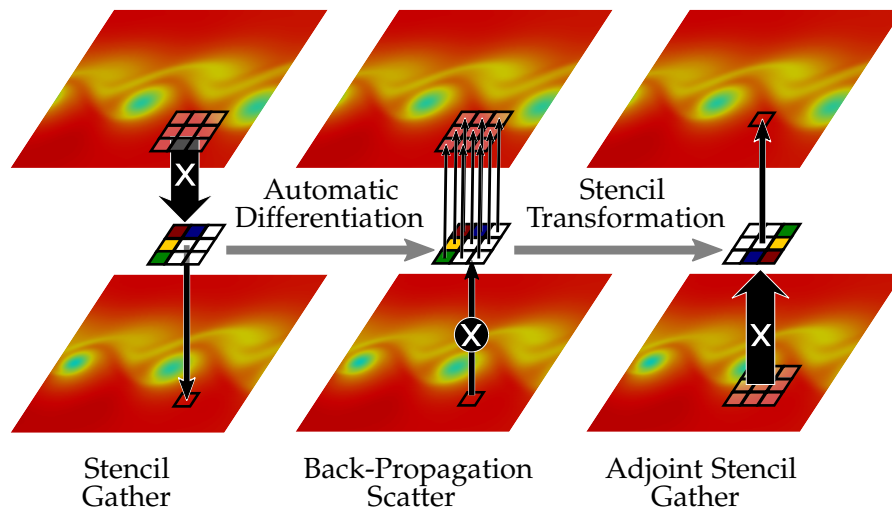


Figure 52: Stencils with gather access occur in convolutions and many other applications and are easy to parallelise. Automatic differentiation or backpropagation transforms this operation into a scatter update, which is hard to parallelise without race conditions. With adjoint stencils, this scatter update can be rearranged back into a gather/stencil and backpropagation can be performed by using only easily parallelisable stencils with gather access.

Our *adjoint stencils* technique solves this problem by implementing backpropagation using only gather operations obtained via loop

transformations of the scatter operator. It requires no additional memory and performs the bulk of the computation with no additional synchronisation barriers. All transformations are applied at compile time. The idea is illustrated in Figure 52.

The chapter is organised as follows. We summarise related work and background in Section 6.2. Following this, we describe the steps to generate adjoint stencils in Section 6.3 and present test cases in Section 6.4. We present experimental performance results in Section 6.5, and in Section 6.7, we summarise our conclusions and briefly describe future work.

6.2 RELATED WORK

AD differs from *numerical differentiation* or *finite differences* in that it computes exact derivatives of the implemented computation, with no truncation errors and without the need to choose any finite step size. Automatic differentiation also differs from symbolic differentiation in that it can handle large computations including loops, branches, and function calls efficiently.

AD for parallel programs has been addressed for distributed-memory MPI programs [48], exploiting the fact that message passing between parallel instances is explicit in the program source code and often limited to a relatively small, carefully defined interface. In this work, we focus on stencils, and not any specific parallelisation technique. Stencil compilers (e.g., YASK) can parallelise in MPI or shared memory but need the stencil structure.

Our approach is similar to that of autodiff for Halide [73]. For simple stencils, we expect both approaches to yield identical results. However, whereas the approach in [73] is intimately tied to Halide and relies on zero padding and sum reductions, our approach can be applied to any stencil compiler, such as Devito [84], HOSTS [103], and TIRAMISU [10], and requires neither padding nor reductions.

While our work uses some concepts from polyhedral compilers, it does so in a targeted way that is tailored to generating stencil loops that compute adjoints. Our framework supports non-rectangular iteration spaces, avoiding some of the over-approximation required in an interval-based representation like Halide.

A core aspect of our work is a gather-scatter conversion, similar to that described in [103]. That paper, however, deals only with performance optimisation of stencil operations, and AD is not considered. To

work with AD, the transformation must recognise that some inputs of the original function, as well as inputs to the adjoint computation, need to be gathered from the appropriate indices.

In a separate paper [52], the authors describe *Transposed Forward-Mode AD* (TF-MAD) to transform the adjoint of a stencil back into a stencil loop. The authors applied this to a structured PDE solver and observed equally good scalability for the primal and adjoint. While that paper has an aim similar to the work presented here, it did not provide an implementation to automate the necessary transformations, and it was restricted to stencils with a symmetric data flow; that is, if the output array at index i depends on the input array at index j , then the output at j must also depend on the input at i . This is not the case for most stencil loops in practice, where the input space is slightly larger than the output space because of boundary effects.

Other authors have presented automatic differentiation for various domain-specific languages [29, 93]. These have in common that the differentiation needs to operate only on a high level of abstraction and assemble hand-optimised building blocks that perform efficient adjoint operations, for example, on neural network layers, partial differential equations, or linear algebra operators. In contrast, our work is not tied to any particular application domain and can differentiate any computation that has a stencil-like structure.

6.3 METHOD AND IMPLEMENTATION

In this section we describe the procedure to generate adjoint stencils and its implementation in the stencil differentiation tool PerforAD, which we release with this work.¹

Before describing the transformations for arbitrary loop nest depths and stencil shapes, we begin with a brief summary of AD, followed by a simple example to illustrate adjoint stencils.

6.3.1 Automatic Differentiation

AD computes partial derivatives of each individual statement in a given *primal* computer program and implements a new *adjoint* program that accumulates those partial derivatives, following the chain rule, to obtain the derivative of the entire primal program. For example, if the original program contains a statement such as

¹ PerforAD v1.1, <https://github.com/jhueckelheim/PerforAD/releases/tag/1.1>

```
r = sin(u);
```

then the adjoint program will contain the corresponding statement

```
ub = cos(u)*rb;
```

where `ub` is used to store the derivative of the program output with respect to `u` and `rb` holds the derivative of the program output with respect to `r`. The multiplication of the derivative of this statement (`cos(u)`) with the output adjoint `rb` is an application of the chain rule of calculus. If a primal expression contains nonlinear functions (in this case `sin()`), the computation of the partial derivatives of that expression needs to access the original primal value (in this case, `u`).

An AD tool often distinguishes between active and passive variables. For example, a user may be interested only in the derivatives of some outputs with respect to some inputs. The inputs and outputs of interest are called independent and dependent, respectively. All intermediate variables that depend on an independent input and that influence a dependent output are called active; all other variables are called passive. Only active variables require a derivative counterpart variable, and only expressions that involve an active variable need to be differentiated. PerforAD supports these requirements by allowing a user to specify which arrays are active.

PerforAD is not a general-purpose AD tool. Rather, it can be seen as an AD-aware loop transformation tool that focuses on the efficient creation of adjoint stencil loops. A general-purpose AD tool is currently necessary to differentiate the entire program, except for the stencil loops that can be handled by PerforAD. The tool does not contain its own parser front-end and instead relies on the caller to supply a high-level description of the stencil computation. In the test cases in this chapter, this description was manually created. Automating this process remains future work. PerforAD is designed in a modular fashion to simplify the creation of new front-ends (for example, to parse Fortran or C code) and back-ends (for example, to print Fortran or C code). PerforAD is written in Python and makes extensive use of the symbolic math library SymPy [88] for its internal computation, as well as for its user interface.

6.3.2 1D Example

This section explains the adjoint stencil transformation step by step using a one-dimensional example. This explanation is accompanied by the illustration in Figure 53.

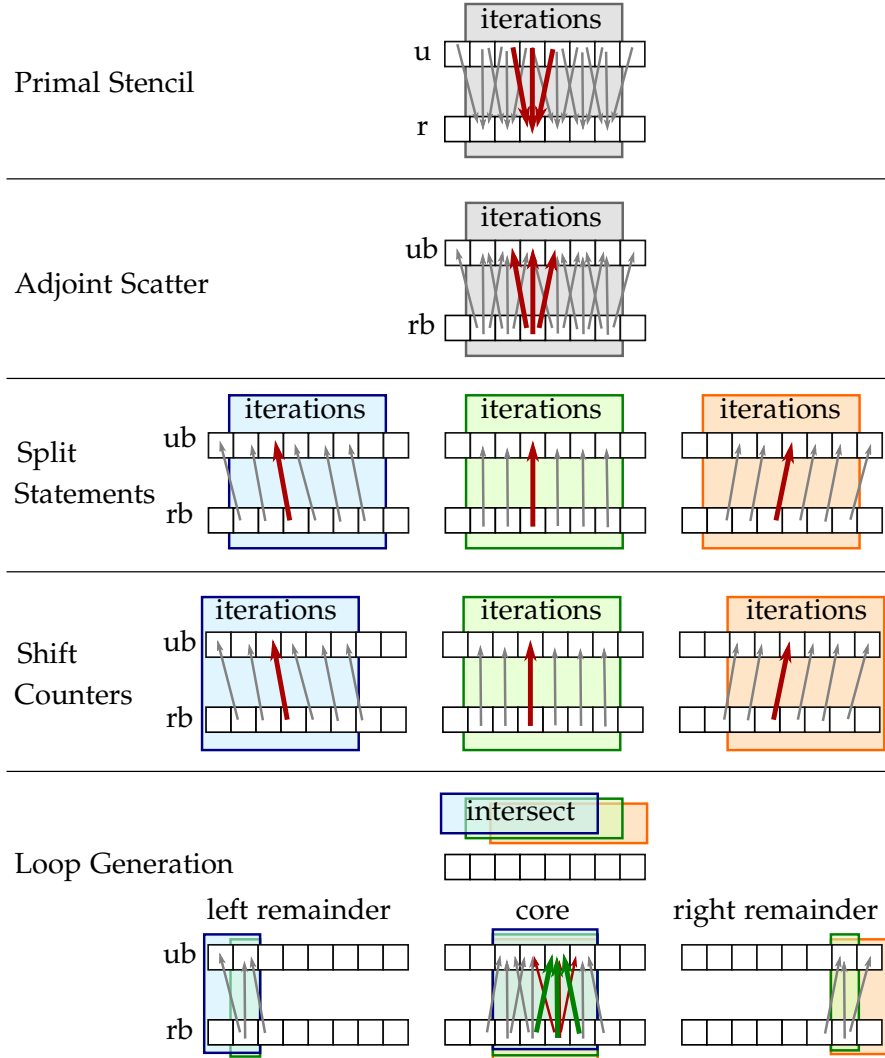


Figure 53: Adjoint stencil transformation step by step, for a one-dimensional three-point stencil. This illustration does not show the read accesses to the primal array *u* that may still be needed in the derivative code. Red arrows in the primal and adjoint show a group of updates that is performed in the same loop iteration. Those same updates are then highlighted in the split, shifted, and core loop. After regeneration, the red updates are distributed among several iterations, and one iteration in the core loop performs a set of updates as shown by green arrows.

Suppose that a primal program contains the following parallel gather operation for an iteration space $i \in [1, n-1]$:

```
#pragma omp parallel for private(i)
for ( i=1; i<=n - 1; i++ ) {
    r[i] = c[i]*(2.0*u[i-1]-3.0*u[i]+4*u[i+1]);
}
```

A straightforward reverse-mode differentiation of this loop to compute the derivatives of r with respect to u would yield the following scatter operation, where ub and rb represent the adjoint variables that correspond to u and r :

```
for ( i=1; i<=n-1; i++ ) {
    ub[i-1] += 2.0 * c[i] * rb[i];
    ub[i] -= 3.0 * c[i] * rb[i];
    ub[i+1] += 4.0 * c[i] * rb[i];
}
```

If we assume that floating-point summation is associative (we come back to this point in Section 6.3.5), we can split this into three loops.

```
for ( i=1; i<=n-1; i++ ) {
    ub[i-1] += 2.0 * c[i] * rb[i];
}
for ( i=1; i<=n-1; i++ ) {
    ub[i] -= 3.0 * c[i] * rb[i];
}
for ( i=1; i<=n-1; i++ ) {
    ub[i+1] += 4.0 * c[i] * rb[i];
}
```

We can now substitute the loop counter i in the first, second, and third loop with $j:=i-1$, $j:=i$, and $j:=i+1$, respectively, and obtain three loops that each use the loop counter j as write index and have iteration spaces $j \in [0, n-2]$, $j \in [1, n-1]$ and $j \in [2, n]$.

```
for ( j=0; j<=n-2; j++ ) {
    ub[j] += 2.0 * c[j+1] * rb[j+1];
}
for ( j=1; j<=n-1; j++ ) {
    ub[j] -= 3.0 * c[j] * rb[j];
}
for ( j=2; j<=n; j++ ) {
    ub[j] += 4.0 * c[j-1] * rb[j-1];
}
```

We observe that the iteration space of the three loops intersects for $j \in [2, n-2]$. Only the iteration space of the first loop contains $j = 0$,

that of the first and second loops contain $j = 1$, that of the second and third contain $j = n - 1$, and the iteration space of the third loop contains $j = n$. We can therefore compute the same result using the following parallel loop and remainder statements.

```
ub[0] += 2.0 * c[1] * rb[1];
ub[1] += 2.0 * c[2] * rb[2];
ub[1] -= 3.0 * c[1] * rb[1];
ub[n-1] -= 3.0 * c[n-1] * rb[n-1];
ub[n-1] += 4.0 * c[n-2] * rb[n-2];
ub[n] += 4.0 * c[n-1] * rb[n-1];
#pragma omp parallel for private(j)
for ( j=2; j<=n-2; j++ ) {
    ub[j] += 2.0 * c[j+1] * rb[j+1];
    ub[j] -= 3.0 * c[j] * rb[j];
    ub[j] += 4.0 * c[j-1] * rb[j-1];
}
```

Assuming that n is sufficiently large, the time spent executing the remainder statements will be insignificant compared with that spent inside the loop, which contains only updates to $ub[j]$ that can easily be merged into a single statement to obtain

```
#pragma omp parallel for private(j) shared(rb,ub,c)
for ( j=2; j<=n-2; j++ ) {
    ub[j] += 4.0 * c[j-1] * rb[j-1]
           - 3.0*c[j] * rb[j]
           + 2.0 * c[j+1] * rb[j+1];
}
```

This adjoint stencil loop has the same set of read and write indices and can be parallelised in the same way as the primal stencil loop. Note that the constant factors 4.0 and 2.0 have swapped their position compared with the primal stencil.

6.3.3 Multidimensional Stencils

We now present the generation of adjoint stencil loops for any number of dimensions and for any stencil shape, as implemented in `PerforAD`.

6.3.3.1 From Inputs to Adjoint Statements

The tool requires as input a symbolic expression that represents the computation performed by a single iteration of the innermost loop. In addition, a list of symbolic objects representing loop counters and symbolic expressions for loop bounds is required. Also, the user must

provide a list of active variables for which counterpart derivative variables will be generated. Two complete examples of PerforAD input scripts are given in Section 6.4.

If the loop body is sufficiently simple, the provided expression is differentiated with respect to all active input variables separately using SymPy’s symbolic differentiation capabilities. We note that even though we use symbolic differentiation for the statement inside the innermost loop, the overall derivative is assembled by using automatic differentiation techniques. For large loop bodies where symbolic differentiation is too inefficient, instead of providing a concrete expression such as

$$r_{i,j} = u_{i-1,j} + 2u_{i,j-1},$$

the user can provide an uninterpreted function such as

$$r_{i,j} = f(a, b)(a = u_{i-1,j}, b = u_{i,j-1}).$$

PerforAD will then generate adjoint code where the derivatives of f are also uninterpreted function calls. For example, the derivative of f with respect to its input a , evaluated at $(u_{i-1,j}, u_{i,j-1})$, would be written as

$$r_{i,j} = \text{derivative}(f(a, b), a)(a = u_{i-1,j}, b = u_{i,j-1}).$$

A back-end could easily replace this expression by the appropriate call to a derivative function that was created manually or by AD.

Each of these generated expressions represents the partial derivative of the stencil with respect to one input (for example, $\frac{\partial f}{\partial L}$). The derivative expressions that are generated in this way will be denoted by $S_1, S_2, \dots, S_M$, where the number of such expressions M is bounded by the stencil size. For example, the above two-point stencil would yield $M = 2$. To implement backpropagation, each expression needs to be multiplied with the adjoint variable that is associated with the output of the primal loop body (in this example, $\bar{r}_{i,j}$), and the result must be assigned to the adjoint variable that is associated with the input with respect to which this expression was differentiated (in this example, $\bar{u}_{i-1,j}$). This yields the expression $\bar{u}_{i-1,j} + = \frac{\partial f}{\partial a} \cdot \bar{r}_{i,j}$. Without the following steps, the derivative expressions would collectively implement the scatter operation that is typical for conventional adjoints of stencil loops.

6.3.3.2 Shifting the Indices

To transform the scatter into a gather operation, PerforAD determines the constant offset $\mathbf{o}$ that is used by each derivative expression (in the above example, a two-dimensional vector $\mathbf{o} = [-1; 0]$). All indices of that expression are increased by $-\mathbf{o}$. In our example, this yields $\bar{u}_{i,j} = \frac{\partial f}{\partial \mathbf{L}} \cdot \bar{r}_{i+1,j}$. In order to preserve the semantics of the adjoint computation, the offset is stored along with the shifted expression and is later used to adjust the loop bounds accordingly. The result of this step is a list of tuples, each containing an expressions and an offset. All expressions now write to the same output index.

If the output of the primal stencil function depends on its inputs in a nonlinear way, then the derivative computation requires read access to the primal input, and any array accesses also need to occur with shifted indices. For our example with

$$\text{derivative}(f(\mathbf{a}, \mathbf{b}), \mathbf{a})(\mathbf{a} = \mathbf{u}_{i-1,j}, \mathbf{b} = \mathbf{u}_{i,j-1}),$$

the derivative with offsets shifted by $\mathbf{o} = [-1; 0]$ would become

$$\text{derivative}(f(\mathbf{a}, \mathbf{b}), \mathbf{a})(\mathbf{a} = \mathbf{u}_{i,j}, \mathbf{b} = \mathbf{u}_{i+1,j-1}).$$

In some cases, the shifted derivative functions will read data from indices that did not occur in any primal expression (here, $i+1, j-1$).

6.3.3.3 Generating the Core Loop Nest

The offsets are now used to identify the largest iteration space in which it is legal to execute all shifted derivative expressions generated in the previous step. We will refer to the loop that covers this space as the *core loop nest*. The loop bounds of the core loop nest depend on the bounds of the primal loop nest supplied by the user, as well as the offsets of the shifted expressions. In each dimension i , the iteration space of the primal loop will be denoted as $i \in [s_i, e_i]$ where s_i, e_i refer to the primal lower and primal upper bound, respectively. The i component of the offset vector of a given adjoint expression S_l will be denoted as $o_i(S_l)$. The lower core loop bound is then given by the original lower loop bound given by the user, plus the maximum offset in that dimension. Conversely, the upper loop bound is given by the

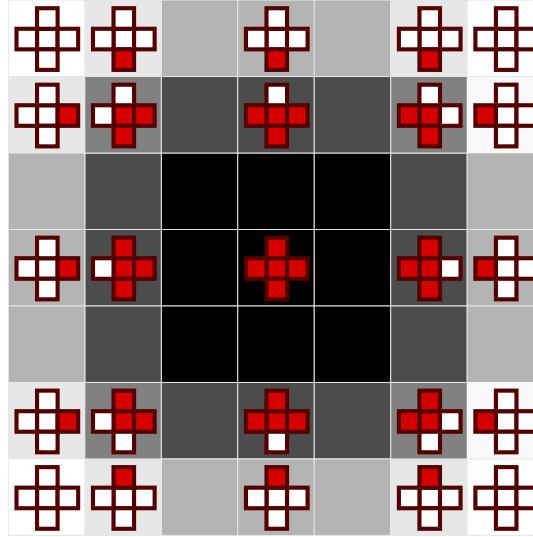


Figure 54: Iteration spaces of adjoint stencil loop nests for a five-point stencil in two dimensions (that is, reading from $[i - 1, j], [i + 1, j], [i, j], [i, j - 1], [i, j + 1]$). Each of the large squares represents one point in the iteration space. Areas with uniform colour are part of the same loop nest. The core loop is represented by the dark area in the centre of the image. Within each loop nest, a stencil gather operation is implemented. The read indices are a subset of the original five-point star stencil. The subset that is actually implemented in that particular loop nest is shown by filled red boxes; the expressions that are not valid in this area are shown with empty red boxes.

original upper loop bound minus the minimum offset. Formally, the adjoint core loop bounds in i -dimension are

$$\left[s_i + \max_{l \in [1, M]} (o_i(S_l)), \quad s_e - \min_{l \in [1, M]} (o_i(S_l)) \right].$$

The core loop nest implements a gather operation with as many distinct read indices as the primal loop. If the iteration space is much larger than the stencil size, this loop nest will perform most of the adjoint computation. The core loop nest can be easily parallelised.

6.3.3.4 Boundary Treatment

In addition to the core loop nest, we need to generate boundary loop nests that execute the appropriate subset of derivative expressions wherever they are valid outside the core area. To achieve these, we split the set of derivative expressions into subsets such that each subset contains only expressions with the same iteration space. Usually more than one correct splitting exists. In the current implementation, the loop nests are generated such that their iteration spaces are dis-

joint; that is, every output index is accessed only within one loop nest that contains all statements that update this location. As a result, no synchronisation barriers are needed between the generated loop nests. Note that the boundary loops are also stencil gather operations, albeit with smaller numbers of read indices, and can be easily parallelised; see Figure 54.

The current splitting strategy results in a relatively large code size: with a primal stencil that gathers data from n points in each of d dimensions, the number of generated adjoint loop nests is at most $(2n - 1)^d$. For the one-dimensional three-point stencil in Section 6.3.2, this resulted in five adjoint loops, of which all but the core loop could be unrolled, since they performed only one iteration. For a dense 3×3 stencil in two dimensions, the number of adjoint loops would be 25, and for a dense three-dimensional $3 \times 3 \times 3$ stencil, 125. If the primal stencil is not dense but instead a star-shaped stencil such as the one shown in Section 6.4.1, then 53 loop nests are needed.

To avoid generating such a large amount of code, the strategy could be changed to instead generate only one remainder loop on each side of the core loop in each dimension, containing all derivative expressions. Since not all expressions are valid throughout the whole remainder, each statement would need to be guarded by an `if`-condition. While this might not be ideal for performance, the effect would be limited to the remainder loop nests, which are at most $d - 1$ -dimensional. A polyhedral compiler could be used to implement many other strategies, with different trade-offs between the code size and the number of branches.

Another strategy is possible if the primal stencil is always executed on entire arrays and the AD tool can control the memory allocation for both primal and adjoint code. In this case, the input arrays could be padded with zeroes, which would allow all shifted adjoint expressions to be executed throughout the entire domain.

6.3.4 Restrictions

The transformation implemented in `PerforAD` requires that there be a set of input arrays from which data is read and a set of output arrays that are written in an assignment or incremented by using the `+=` operator. The sets of read and write arrays must not intersect; that is, no array can be both input and output (with the aforementioned exception of the `+=` operator). In addition, read access is allowed from constants

that are not active for differentiation. All output arrays are accessed only by using the loop counters as indices. For a nested loop with counter variables $[i_1, i_2, \dots]$, a multidimensional array may be written or incremented by using a permuted subset of the counter variables as indices, for example, $r[i_1][i_4][i_3]$. All input arrays are read at indices that are constant integer offsets of the loop counters, for example, $u[i_1+3][i_3-2][i_4-1]$. The offsets must be known at compile time for expressions to be shifted and remainder loops to be generated. The loop nest must be perfect; that is, no statement may appear other than inside the innermost loop. All loop bounds must be affine functions.

6.3.5 *Associativity of Floating-Point Summation*

We assume that the contributions to each adjoint output index can be reordered arbitrarily without affecting the final result. This assumption would be true for real numbers but is not true for floating-point numbers because of roundoff effects. If this effect is important for some application, PerforAD could be modified to respect a particular ordering that the user chooses. The ordering would still be deterministic if the adjoint code is executed in parallel. This is possible because all updates to a given index are collected within the same iteration. In contrast, the scatter operation generated by a conventional AD tool could not be easily parallelised in a way that produces a deterministic floating-point result.

6.3.6 *Verification*

To verify the correctness of the adjoint code generated by PerforAD, we compared it with adjoint code generated by the AD tools ADIC [89] and Tapenade [40]. We used the test cases described in Section 6.4 and checked for equality of the computed adjoints. For all the input test cases in this chapter, the outputs of all the three approaches were in full agreement.

6.4 TEST CASES

We have selected two test cases to demonstrate the performance of adjoint stencils in PerforAD. Both test cases are structured-mesh discretisations of partial differential equations. The generated adjoint

stencil code faces two performance challenges. First, a large number of remainder loops may cause large adjoint code sizes and potentially slow the execution. Second, the use of symbolic differentiation applied to the loop body may cause unnecessary computations for complicated primal loop bodies, especially since PerforAD makes no attempt to identify common sub-expressions within the same loop nest. We selected a test case with a deep loop nest and another one with a complicated loop body to test our tool in the presence of these challenges.

The first test case is a solver for wave equations with spatially varying wave propagation speed. The computation is performed in a three-deep loop nest, corresponding to the three spatial dimensions. A large number of iterations is usually performed with this type of solver, which makes this a good showcase for performance evaluations.

The second test case is a one-dimensional Burgers equation solver. The loop body is more challenging to differentiate, because it is non-linear and only piecewise differentiable. As a result, the generated adjoint is more complex and contains branches.

For both test cases, we generate adjoint stencils using PerforAD and generate conventional adjoint code with scatter memory access using the Tapenade [40] AD tool. Tapenade generates serial output code. We thus additionally create a parallelised version of the conventional adjoint code by adding OpenMP pragmas to the Tapenade-generated code, safeguarding the scattered memory access with atomic pragmas to avoid data races.

6.4.1 Wave Equation

The wave equation and its adjoint are solved numerically in applications including seismic imaging, where performance on high-performance computing systems has been identified as a bottleneck [5]. We perform one single time step on a $1000 \times 1000 \times 1000$ three-dimensional grid. The wave equation is given by

$$\frac{\partial^2 u}{\partial t^2} = a^2 \cdot \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right) \text{ for } \vec{x} \in \Omega \subset \mathbb{R}^3, t \in (0, T].$$

```

import sympy as sp
from perforad import *
##### 3D Wave Equation Example #####
# Define symbols
c = sp.Function("c")
u_1 = sp.Function("u_1"); u_1_b = sp.Function("u_1_b")
u_2 = sp.Function("u_2"); u_2_b = sp.Function("u_2_b")
u = sp.Function("u") ; u_b = sp.Function("u_b")
i,j,k,C,D,n = sp.symbols("i,j,k,C,D,n")
# Build stencil expression
u_xx = u_1(i-1,j,k) - 2*u_1(i,j,k) + u_1(i+1,j,k)
u_yy = u_1(i,j-1,k) - 2*u_1(i,j,k) + u_1(i,j+1,k)
u_zz = u_1(i,j,k-1) - 2*u_1(i,j,k) + u_1(i,j,k+1)
expr = 2.0*u_1(i,j,k) - u_2(i,j,k) + c(i,j,k)*D*(u_xx+u_yy+u_zz)
# Build LoopNest object for this expression
lp = makeLoopNest(lhs=u(i,j,k), rhs=expr, counters = [i,j,k],
                 bounds={i:[1,n-2],j:[1,n-2],k:[1,n-2]})
# Output primal and adjoint files
printfunction(name="wave3d", loopnestlist=[lp])
printfunction(name="wave3d_perf_b",
              loopnestlist=lp.diff({u:u_b, u_1:u_1_b, u_2: u_2_b
                                   })))

```

Figure 55: Python script that uses the PerforAD package to generate C code, which implements a three-dimensional wave equation solver. The input and output arrays are represented in PerforAD using SymPy function objects. All scalars, including loop counters and bounds, are represented by SymPy symbol objects. The SymPy expression object `expr`, a list of loop counters, and a dictionary with bounds associated with each counter are passed to the `makeLoopNest()` function, which is part of PerforAD. It returns a `LoopNest` object, which encapsulates all necessary data to represent a stencil computation, and provides functions to generate the primal and adjoint C code.

By using finite differences to discretise the differential operators in space and time with spatial and temporal resolution dx and dt and by letting $c := a^2$ and

$$D := \left(\frac{dt}{dx}\right)^2 = \left(\frac{dt}{dy}\right)^2 = \left(\frac{dt}{dz}\right)^2,$$

a primal solver to solve this equation, as well as its adjoint, can be generated with the Python script shown in Figure 55.

The primal and adjoint stencil loop shown in Figure 56 are generated by PerforAD and shown alongside the conventional adjoint generated by Tapenade, which is not a stencil loop and thus had to be manually parallelised.

```

// Stencil primal
#pragma omp parallel for private(k,j,i)
for ( i=1; i<=n - 2; i++ )
  for ( j=1; j<=n - 2; j++ )
    for ( k=1; k<=n - 2; k++ )
      u[i][j][k] += D*(-6*u_1[i][ j][ k] + u_1[i][ j][ k - 1]
        + u_1[i][ j][ k + 1] + u_1[i][ j - 1][ k]
        + u_1[i][ j + 1][ k] + u_1[i - 1][ j][ k]
        + u_1[i + 1][ j][ k])*c[i][ j][ k]
        + 2.0*u_1[i][ j][ k] - u_2[i][ j][ k];

// PerforAD stencil adjoint
for ( i=2; i<=n - 3; i++ )
  for ( j=2; j<=n - 3; j++ )
    for ( k=2; k<=n - 3; k++ ) {
      u_1_b[i][j][k] += D*c[i][ j][ k + 1]*u_b[i][ j][ k + 1];
      u_1_b[i][j][k] += D*c[i][ j + 1][ k]*u_b[i][ j + 1][ k];
      u_1_b[i][j][k] += D*c[i + 1][ j][ k]*u_b[i + 1][ j][ k];
      u_1_b[i][j][k] += (-6*D*c[i][ j][ k] + 2.0)*u_b[i][ j][ k];
      u_2_b[i][j][k] += -u_b[i][ j][ k];
      u_1_b[i][j][k] += D*c[i - 1][ j][ k]*u_b[i - 1][ j][ k];
      u_1_b[i][j][k] += D*c[i][ j - 1][ k]*u_b[i][ j - 1][ k];
      u_1_b[i][j][k] += D*c[i][ j][ k - 1]*u_b[i][ j][ k - 1];
    }

// Tapenade adjoint, manually parallelised
#pragma omp parallel for private(i, j, k, tempb)
for ( i = n-2; i > 0; --i)
  for ( j = n-2; j > 0; --j)
    for ( k = n-2; k > 0; --k) {
      tempb = D*c[i][j][k]*ub[i][j][k];
      #pragma omp atomic
      u_1b[i][j][k - 1] = u_1b[i][j][k - 1] + tempb;
      #pragma omp atomic
      u_1b[i][j][k] = u_1b[i][j][k] + 2.0*ub[i][j][k] - 6*tempb;
      #pragma omp atomic
      u_1b[i][j][k + 1] = u_1b[i][j][k + 1] + tempb;
      #pragma omp atomic
      u_1b[i][j - 1][k] = u_1b[i][j - 1][k] + tempb;
      #pragma omp atomic
      u_1b[i][j + 1][k] = u_1b[i][j + 1][k] + tempb;
      #pragma omp atomic
      u_1b[i - 1][j][k] = u_1b[i - 1][j][k] + tempb;
      #pragma omp atomic
      u_1b[i + 1][j][k] = u_1b[i + 1][j][k] + tempb;
      u_2b[i][j][k] = u_2b[i][j][k] - ub[i][j][k];
    }
}

```

Figure 56: One time step for the primal and adjoint wave equation solver implemented in C, as generated by PerforAD. The output array is written to at the central index i, j, k , while the input arrays are read in the neighbourhood around that centre. The loop is parallelised. In contrast, the conventional adjoint implements a scatter operation to a neighbourhood around the centre and is manually parallelised by using atomic updates.

```

import sympy as sp
from perforad import *
##### 1D Burgers Equation Example #####
# Build stencil expression
ap = sp.functions.Max(u_1(i),0)
am = sp.functions.Min(u_1(i),0)
uxm = u_1(i)-u_1(i-1)
uxp = u_1(i+1)-u_1(i)
ux = ap*uxm+am*uxp
expr = u_1(i) - C * ux + D * (u_1(i+1) + u_1(i-1) - 2.0*u_1(i))
# Build LoopNest object for this expression
lp = makeLoopNest(lhs=u(i), rhs=expr, counters = [i],
                  bounds={i:[1,n-2]})
# Output primal and adjoint files
printfunction(name="burgers1d", loopnestlist=[lp])
printfunction(name="burgers1d_perf_b",
              loopnestlist=lp.diff({u:u_b, u_1:u_1_b}))

```

Figure 57: Python script to generate the Burgers equation solver. An upwinding scheme is implemented by using min and max functions, which are piecewise differentiable and correctly handled by SymPy and PerforAD.

6.4.2 Burgers Equation

Computational fluid dynamics (CFD) is another application that routinely uses adjoints and relies on efficient parallelisation to compute industrial-scale test cases. The Burgers equation is often used as a prototype in CFD solver development, because it is scalar and thus relatively easy to implement, but it still shows some of the challenging nonlinear behaviour that makes CFD simulations difficult. In one spatial dimension, it can be written as

$$\frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} = \nu \frac{\partial^2 u}{\partial x^2}.$$

An interesting challenge in the context of this work is that the convective term is nonlinear. A common way of discretising this term is *upwinding*, where the finite difference formula depends on the sign of u as

$$u \frac{\partial u}{\partial x} \approx \frac{1}{dx} \cdot (\max(u_i, 0) \cdot (u_i - u_{i-1}) + \min(u_i, 0) \cdot (u_{i+1} - u_i)).$$

By letting $C := \frac{dt}{dx}$ and $D := \nu * \frac{dt}{dx^2}$, the primal and adjoint stencils can be implemented by using the PerforAD script shown in Figure 57, yielding the C code shown in Figure 58.

```

// Stencil Primal
for ( i=1; i<=n - 2; i++ ) {
    u[i] += -C*((-u_1[i] + u_1[i + 1])*fmin(0, u_1[i])
              + (u_1[i] - u_1[i - 1])*fmax(0, u_1[i]))
            + D*(-2.0*u_1[i] + u_1[i - 1] + u_1[i + 1])
            + u_1[i];
}

// PerforAD Stencil Adjoint
for ( i=2; i<=n - 3; i++ ) {
    u_1_b[i] += (C*fmax(0, u_1[i + 1]) + D)*u_b[i + 1];
    u_1_b[i] += (-C*((-u_1[i] + u_1[i + 1])*((-u_1[i]>=0)
        ?1.0:0.0)
              + (u_1[i] - u_1[i - 1])*((u_1[i]>=0)?1.0:0.0)
              + fmax(0, u_1[i]) - fmin(0, u_1[i]))
              - 2.0*D + 1)*u_b[i];
    u_1_b[i] += (-C*fmin(0, u_1[i - 1]) + D)*u_b[i - 1];
}

```

Figure 58: Primal and adjoint stencil solvers for the Burgers equation in one dimension. The upwinding scheme results in the use of ternary operators in the adjoint code.

The stencil is nonlinear, and the value of u is required at various points in the derivative computation. The adjoint code generated by Tapenade handles the min and max functions differently from PerforAD. Instead of evaluating these functions within the adjoint code, Tapenade creates a loop that evaluates the functions separately and pushes the results onto a stack. These precomputed values are popped from the stack during the adjoint computation. The resulting code cannot easily be parallelised, since the order in which values are added and removed from the stack is crucial.

To exclude this effect and compare PerforAD with a parallelised adjoint code that uses scatter operations, we manually modified the Tapenade output to instead evaluate the min and max functions within the adjoint computation itself, and we removed the stack access. Following this, we made the loop OpenMP-parallel and added atomic pragmas. We ran the primal and all adjoint solvers on a one-dimensional problem with one time step on 1 billion cells.

6.5 EXPERIMENTAL RESULTS

We obtained timing results on two test systems for the primal and adjoint, as well as conventional adjoint solvers for the wave equation, and the Burgers equation. The first test system, referred to as *Broad-*

Scalability of the Wave Equation on Broadwell

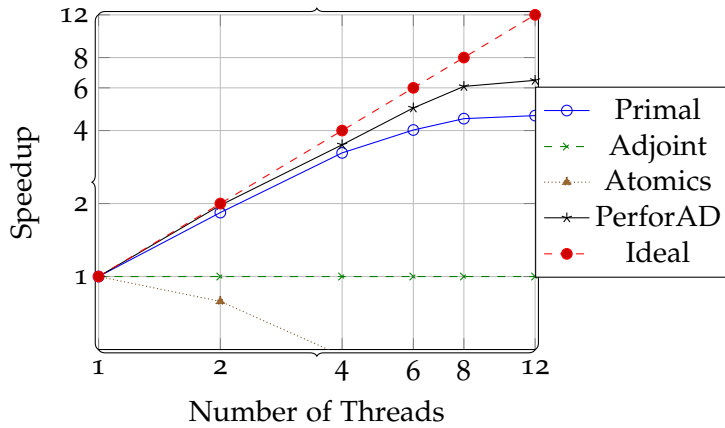


Figure 59: Speedups for the wave equation solver on a Broadwell processor, using up to 12 threads. The Tapenade-generated code with manual parallelisation does not scale at all. The primal and PerforAD-generated adjoint benefit from using all 12 cores.

well, uses a dual socket system with 12-core Intel®Xeon®E5-2650 v4 Broadwell CPUs, with a combined total of 24 physical cores. To eliminate NUMA effects on our experiments, we limited the number of threads to 12, and used `numactl` to restrict memory and computation to a single socket and `OMP_PROC_BIND` to ensure thread pinning. The other test system, referred to as *KNL*, runs an Intel® Xeon Phi™ (Knights Landing) 7210 processor with 64 physical cores and up to 256 threads, in `KMP_AFFINITY=scatter` mode. The examples are compiled using the Intel C Compiler version 18 using the flags `-O3 -fopenmp -xHost`.

6.5.1 Broadwell

Since PerforAD introduces overhead by differentiating the loop body separately for each function input using SymPy, we expect the serial performance of the generated adjoint stencils to be worse than that of Tapenade-generated adjoint code. On the other hand, we expect adjoint stencils to have parallel scalability at least equivalent to that of the primal stencil.

All of this is indeed confirmed in our experiments. Figure 61 shows absolute runtimes for our wave equation test case on Broadwell. The serial runtime of adjoint stencils is 64% higher than that of Tapenade adjoints, at 8.52s compared with 5.43s. However, using atomic pragmas on the conventional adjoint slows that code, and it requires 91s

Scalability of the Burgers Equation on Broadwell.

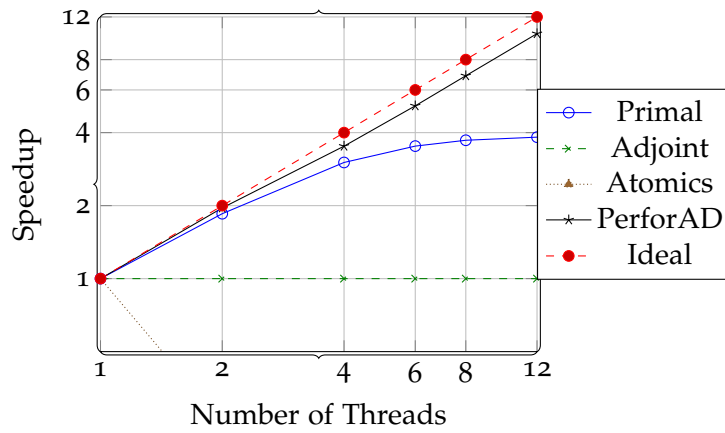


Figure 60: Speedups for the Burgers equation solver. The PerforAD-generated adjoint has near-perfect scalability.

Runtimes of the Wave Equation on Broadwell

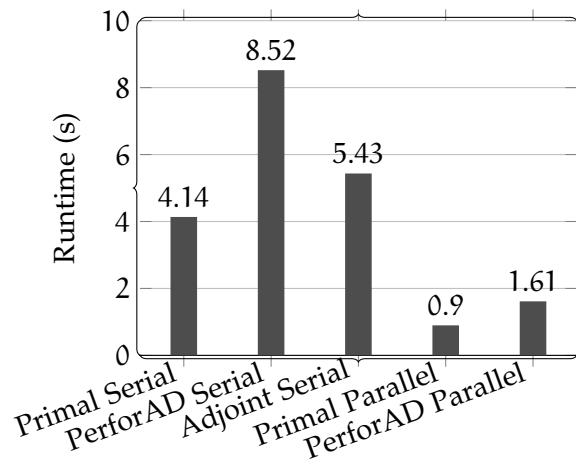


Figure 61: Absolute runtimes for wave equation primal and adjoint stencils and conventional adjoints in serial, as well as best observed primal and adjoint stencil run time in parallel. The best-observed performance of adjoint stencils was with 12 threads and is faster than the conventional adjoint by a factor of $3.4\times$.

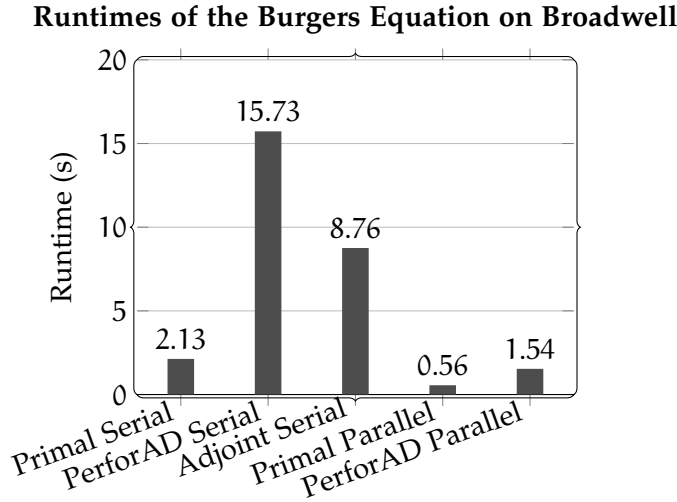


Figure 62: Absolute runtimes for the Burgers equation primal and adjoint stencils and conventional adjoints in serial, as well as best observed primal and adjoint stencil runtime in parallel. Despite being slower in serial, the adjoint stencil outperforms conventional adjoints by a factor of $5.7\times$.

even if only one thread is used, and it slows down further with any additional thread that is added to the computation. This result is in agreement with the reported slowdown of using atomics in [96]. In contrast, the adjoint stencil code scales well and performs slightly better than the conventional adjoint if 2 threads are used, and significantly better if more threads are added. Similarly, Figure 62 shows absolute runtimes for the Burgers equation test case, where the serial performance of adjoint stencils is worse but, because of the improved scalability, outperforms the conventional adjoint solver if at least 2 threads are used. The scalability of all wave equation solvers on Broadwell is presented in Figure 59, and for the Burgers equation solvers in Figure 60.

6.5.2 *Knights Landing (KNL)*

As can be seen in Figure 63, on the KNL system, the primal wave equation solver scales up to 16 threads, then plateaus. The solver cannot efficiently scale to the full number of threads, probably because the memory bandwidth is saturated. Adjoint stencils scale better than the primal stencils do (up to 32 threads), probably because the loop body contains more operations. The better scalability means that the performance gap between the primal and adjoint stencil is slightly smaller for parallel than for the serial execution.

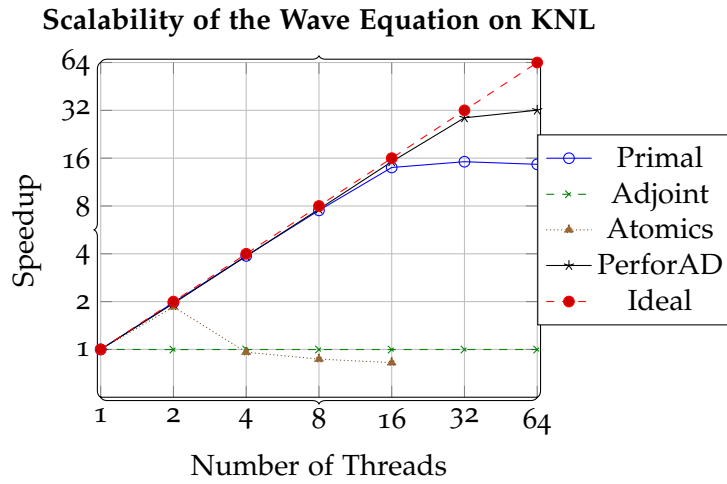


Figure 63: Speedups for the wave equation solver on KNL, using up to 64 threads. The primal loop scales well only up to 16 threads. The adjoint stencil loop generated by PerforAD continues to scale up to 32 threads. The manually parallelised adjoint solver scales only up to 2 threads.

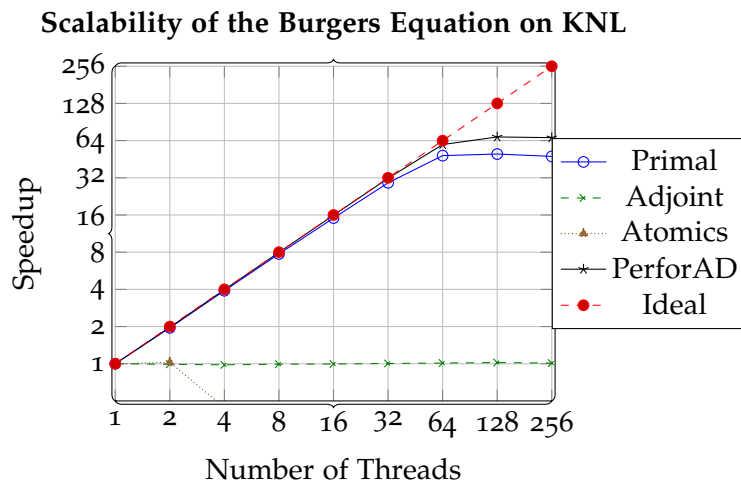


Figure 64: Speedups for the Burgers equation solver, showing near-perfect scalability up to 64 threads for the primal and adjoint stencil solver on a KNL processor. The scatter adjoints with atomics do not scale at all.

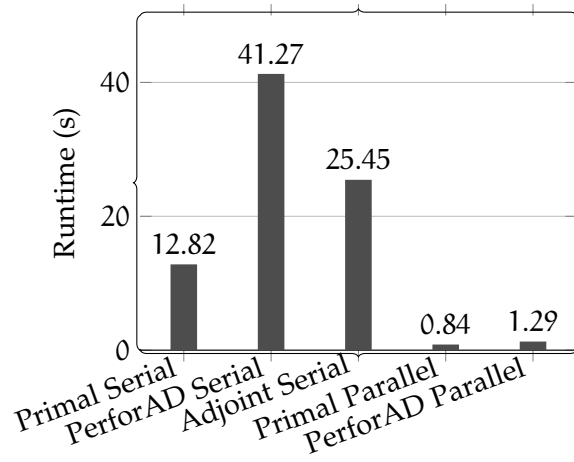
Runtimes of the Wave Equation on KNL

Figure 65: Absolute runtimes for wave equation solvers. The fastest primal stencil used 128 threads, while the fastest adjoint stencil used 256 threads. Since the conventional adjoint code does not scale well, adjoint stencils lead to a much-reduced runtime in parallel, over $19\times$ faster than the best runtime of the conventional adjoint code.

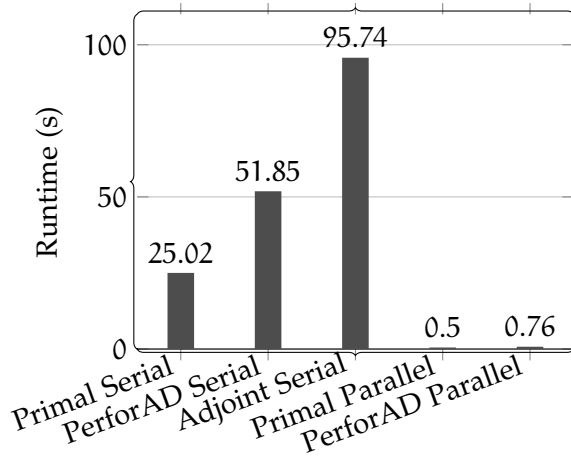
Runtimes Burgers Equation on KNL

Figure 66: Absolute runtimes for the Burgers equation solvers. The adjoint stencil loop is faster than the conventional adjoint code generated by Tapenade. The latter is slowed by the access to the intermediate value stack that is used to store the results of the min and max functions. This, combined with the differences in scalability, results in a $125\times$ run time difference between Tapenade and PerforAD.

The Tapenade wave equation adjoint is not parallelised and hence does not scale. The manually parallelised scatter adjoint with atomics scales up to 2 threads, then becomes slower with any additional thread. Its absolute runtime is always worse than that of the serial adjoint, seen in Figure 65. The best-observed runtime for stencil adjoints is more than $19\times$ better than that of the conventional adjoint code.

Figure 64 shows that for the Burgers equation, adjoint and primal stencils scale well up to 64 threads. Figure 66 shows that parallelised conventional adjoints with atomics always perform worse than serial conventional adjoints in our experiment. On KNL, we used the original Tapenade output that precomputes min and max functions on a stack for our serial run times. Hence, the PerforAD-generated adjoint stencil is faster even in serial. The manually parallelised Tapenade adjoint is identical to that used on Broadwell, with no stack access. Adjoint stencils outperform conventional adjoints by a factor of $125\times$.

6.6 IMPLEMENTATION IN DEVITO

In this chapter, we have discussed a transformation that can convert a scatter operation to a gather operation. This transformation comes as a second step after the differentiation itself. From an API perspective, it would probably be better to implement the differentiation externally to Devito's Operator API, because, among other reasons, Sympy provides an easy API to differentiate expressions that could be used directly by the user.

The transformation itself could be implemented as a pass inside the compiler - triggered by some heuristics that detect this sort of a scatter operation. This would be useful even if the user provides a similar scatter operation directly - Devito could use the same pass to rearrange that scatter into a more efficient gather operation.

This is enough to differentiate the core stencil of Devito-generated code. However, there are some other components of the code that would need special treatment. The first of these are the *sources* and *receivers*. These are implemented in Devito as sparse operations i.e. not stencil operations. Assuming bilinear/trilinear interpolation, a source is injected by injecting its value on 4 points in a 2D domain, and 6 on a 3D domain. This would be more than $4/6$ in case of a more complicated interpolation function. Similarly a receiver is interpolated by gathering its value from $4/6$ points in 2D/3D. For real-valued source functions, we know that sources and receivers are adjoints of each

other. Hence automatic differentiation for these objects could be done at a higher level, where a source is simply replaced by a receiver, and the source expression appropriately differentiated according to the chain rule. Further thought might be required to handle complex-valued source functions.

One key feature of the method discussed here is the complete treatment of the boundary region. We briefly discussed the possibility of padding the domain with zeros to simplify the treatment of the boundary region. However, that simplified approach would only work if exactly one stencil was being applied to the whole domain. In a scenario where different stencils might be applied to different parts of the domain, the approach with explicit treatment of boundaries would be the only correct one.

Different stencils being applied to different parts of the domain are a common resulting feature when applying advanced boundary conditions in the PDE problems being set up in Devito. This more complete approach of complete boundary treatment would therefore handle advanced PDE boundary conditions naturally.

6.7 CONCLUSION AND FUTURE WORK

We have presented adjoint stencils, a method for automatic differentiation or backpropagation of gather stencil loops in a way that uses only gather stencil operations. The test cases in our work demonstrate that adjoint stencils are as scalable on CPU and Intel® Xeon Phi™ (Knights Landing) systems as are the original computations, and can outperform programs generated by conventional automatic differentiation by orders of magnitude.

We plan to test our method also on GPU systems, and to explore similar transformations not only for stencil computations but more generally for programs that have been performance-optimised. We aim to extend PerforAD by combining it with conventional automatic differentiation tools, as well as polyhedral compilers [10, 16], to target more applications. It would also be useful to integrate our tool with domain-specific language compilers [59, 62, 84] to make these transformation available to a larger audience. The implementation of a production-ready version of this method in Devito remains future work.

CONCLUSION

In mathematics you don't understand things. You just get used to them.

—John von Neumann

With this work, we have built a framework that allows experimentation with memory-compute tradeoffs that are essential to building a high-performance backpropagation pipeline. Through this framework, we have provided the ability to execute problems that would be too large to fit in memory otherwise. This has enabled significant follow-on research, not only by us but also many other researchers [65, 81, 85, 116–118]. Through these experiments, we incorporated lossy compression into the checkpointing framework and extended the traditional memory-compute tradeoff into a three-way tradeoff involving memory, compute, and precision. Our experiments show that the precision required in FWI is not very high and the results remain unchanged even after extreme lossy compression.

A key design decision of *pyRevolve* was that the core of the framework executes in python. From prior knowledge about the *Revolve* algorithm, we were aware that *Revolve* always calls the reverse operator for one timestep at a time, and even the forward operator is called for a single timestep a majority of the time. The consequence of this decision - that the time loop will essentially run in python when using *pyRevolve*, was deemed to be an acceptable compromise at the time of initial design. This decision was based on past experience with Firedrake [97], where the time loop runs entirely in python, and this fact has not been seen as the bottleneck in computations.

A clear difference between Firedrake and Devito is the fact that Firedrake focusses on implicit timestepping schemes, while Devito's focus is on explicit timestepping schemes. The most relevant effect of this difference is that the computational load inside a single timestep is much smaller in Devito, implying that any inter-timestep overheads would be more significant in Devito than in Firedrake. This was very clearly seen in practice since there is an undeniable slowdown when using *pyRevolve* as compared to a naive forward-reverse adjoint run, even in the case (perhaps more so) where *pyRevolve* is asked to checkpoint every single timestep.

Through various profiling experiments that we showed in Chapter 4, we identified the bottlenecks in performance. The first of these was excessive data movement, which we reduced somewhat. To continue reducing the data movement to the theoretical minimum, changes were required in how Devito constructs its Operators (Section 4.7.4.1) and we did not get the opportunity to complete that work. Had we got to the point where we identified the python-to-C function call overhead as the bottleneck, we had the option to consider *CFFI* as the interface Devito uses to call C functions from python instead of the current *Ctypes*.

It is likely that continuing down this path of profiling to identify the most significant bottleneck and then addressing that might lead us to a satisfactory level of performance even while staying within the current high-level checkpointing framework in python. However, a quicker alternative might be to move the majority of the checkpointing operations to the C code. One way to do this that would be more extensible and maintainable could be to precompute a schedule in python, that is then provided to an executor in C. This method would have the advantages that it could continue to be a general library that can be used without Devito, and that it could be interfaced with different schedulers that are most likely written in python. A clear disadvantage would be that this method would not support online schedules i.e. any scenario where the scheduler can not precompute the optimal schedule before the start of execution. Two scenarios come to mind that might require this - the first is adaptive timestepping. When using adaptive timestepping, the number of timesteps is not known in advance, so the scheduler needs to be called at every time-step. Another scenario where this might be necessary is if we have a compression-aware scheduler in the future. Since the size of a compressed checkpoint is not easy to predict, the scheduler would need constant feedback on the amount of memory remaining for future checkpoints after every checkpoint. To support online schedules in this way, we would be required to maintain high-performance C implementations of at least the online schedulers, that could be called from within the C timestepping loop.

Another open question is that of the right level of abstraction for checkpointing, compression and AD. At this point ¹, Devito identifies as a DSL for finite-difference only. Under this strict definition, perhaps the three features of checkpointing, compression and AD do

¹ These definitions can change with time.

not belong inside Devito. This is because, while an FD DSL is essential for building high-performance kernels for inversion, the FD DSL has no abstractions to capture and represent the optimisation problem. For example, in its current structure, Devito has no knowledge of the objective function. Without this, any user API for automatic differentiation would be unnecessarily complicated, as compared to pytorch, for example. Hence, an abstraction for the objective function is essential to build a good user API for AD in Devito.

A second feature is that, in the current form, Devito sees the forward and reverse operator as two independent operators. The relationship that the reverse operator is the gradient of (and perhaps derived from) the forward operator will be easier to capture with the introduction of the AD API. Once the forward and reverse operator are captured into a single higher-level abstraction – this would be the “right” place for checkpointing to exist, since all the information required to build a checkpointing schedule is available together at this level of abstraction.

While the implementation of compression belongs right next to checkpointing, the compressor requires an *atol* parameter that we assumed as given throughout our work on compression in Chapter 5. We previously referred to work that can provide a schedule for changing precision as the optimisation proceeds [13]. This sort of schedule would require feedback from the FWI optimiser at every iteration and would therefore require an abstraction to capture the optimiser itself.

Hence, we envision these features to live at a level of abstraction that is above what is currently Devito. Either the definition of Devito will adapt to include this higher level of abstraction, or this *inversion framework* will be given another name ².

While we have only studied FWI during this work, the ideas of checkpointing and compression can probably also be applied to the training of neural networks. Since the computations required to train a NN are much smaller than the ones during FWI, the minimum requirements on the hardware are much smaller for NN-training. However, with the advent of Edge computing, it is becoming increasingly common to see NN-training on small devices that are heavily constrained on CPU and memory resources. In this scenario of NN-training on the edge, the techniques discussed in this thesis become much more relevant.

² We called a previous attempt *Conway*, but that name and framework did not stick.

In recent times, we have also seen an increase in the use of domain-specific hardware for machine learning (e.g. Cerebras/Graphcore). It is common for such hardware to compromise memory capacity for memory bandwidth. This again, brings in the need for memory-compute tradeoffs of the kind we have discussed in this work.

Medical imaging, as an application area, combines both the previous scenarios where we want to use domain-specific hardware at the source of the data to quickly compute the image. Using cloud technologies here might bring in data-privacy concerns that are avoided by combining domain-specific hardware and edge computing.

There is also a lot of follow-on research into reduced precision computation, as well as alternate floating-point representations, that is informed by our work on lossy compression of checkpoints. These methods become highly relevant as we are moving to the Exaflop era - the compute speeds are growing much faster than memory speeds - making memory-compute tradeoffs important.

BIBLIOGRAPHY

- [1] URL: <https://computation.llnl.gov/projects/floating-point-compression/zfp-and-derivatives>.
- [2] Martin S. 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.” In: *ACM Transactions on Mathematical Software (TOMS)* 40.2 (2014), p. 9.
- [3] Francesc Altèd. “Why modern CPUs are starving and what can be done about it.” In: *Computing in Science & Engineering* 12.2 (2010), p. 68.
- [4] Fred Aminzadeh, N Burkhard, J Long, Tim Kunz, and P Duclos. “Three dimensional SEG/EAGE models—An update.” In: *The Leading Edge* 15.2 (1996), pp. 131–134.
- [5] M. Araya-Polo et al. “Assessing Accelerator-Based HPC Reverse Time Migration.” In: *IEEE Transactions on Parallel and Distributed Systems* 22.1 (Jan. 2011), pp. 147–162. ISSN: 1045-9219. DOI: [10.1109/TPDS.2010.144](https://doi.org/10.1109/TPDS.2010.144).
- [6] A Arbona, B Miñano, A Rigo, C Bona, C Palenzuela, A Artigues, C Bona-Casas, and J Massó. “Simflowny 2: An upgraded platform for scientific modeling and simulation.” In: *arXiv preprint arXiv:1702.04715* (2017).
- [7] Guillaume Aupy and Julien Herrmann. “Periodicity in optimal hierarchical checkpointing schemes for adjoint computations.” In: *Optimization Methods and Software* 32.3 (2017), pp. 594–624.
- [8] Guillaume Aupy, Julien Herrmann, Paul Hovland, and Yves Robert. “Optimal multistage algorithm for adjoint computation.” In: *SIAM Journal on Scientific Computing* 38.3 (2016), pp. C232–C255.
- [9] John Backus. “The history of Fortran I, II, and III.” In: *History of programming languages I*. ACM. 1978, pp. 25–74.

- [10] 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*. IEEE Press. 2019, pp. 193–205.
- [11] Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. "Automatic differentiation in machine learning: a survey." In: *Journal of Machine Learning Research* 18 (2018), pp. 1–43.
- [12] Olivier Beaumont, Julien Herrmann, Guillaume Pallez, and Alena Shilova. "Optimal memory-aware backpropagation of deep join networks." In: *Philosophical Transactions of the Royal Society A* 378.2166 (2020), p. 20190049.
- [13] Jose Blanchet, Coralia Cartis, Matt Menickelly, and Katya Scheinberg. "Convergence rate analysis of a stochastic trust-region method via supermartingales." In: *INFORMS journal on optimization* 1.2 (2019), pp. 92–119.
- [14] Christian Boehm, Mauricio Hanzich, Josep de la Puente, and Andreas Fichtner. "Wavefield compression for adjoint methods in full-waveform inversion." In: *Geophysics* 81.6 (2016), R385–R397.
- [15] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. "A Practical Automatic Polyhedral Parallelizer and Locality Optimizer." In: *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '08. Tucson, AZ, USA: ACM, 2008, pp. 101–113. ISBN: 978-1-59593-860-2. DOI: [10.1145/1375581.1375595](https://doi.org/10.1145/1375581.1375595). URL: <http://doi.acm.org/10.1145/1375581.1375595>.
- [16] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. "A practical automatic polyhedral program optimization system." In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. June 2008.
- [17] George Boole. *An investigation of the laws of thought: on which are founded the mathematical theories of logic and probabilities*. Vol. 2. Walton and Maberly, 1854.

- [18] Alfonso F Cárdenas and Walter J Karplus. "PDEL—a language for partial differential equations." In: *Communications of the ACM* 13.3 (1970), pp. 184–191.
- [19] Yohan Chatelain, Eric Petit, Pablo de Oliveira Castro, Ghislain Lartigue, and David Defour. "Automatic exploration of reduced floating-point representations in iterative methods." In: *European Conference on Parallel Processing*. Springer. 2019, pp. 481–494.
- [20] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. "Training deep nets with sublinear memory cost." In: *arXiv preprint arXiv:1604.06174* (2016).
- [21] Matthias Christen, Olaf Schenk, and Helmar Burkhart. "PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures." In: *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*. IPDPS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 676–687. ISBN: 978-0-7695-4385-7. DOI: [10.1109/IPDPS.2011.70](https://doi.org/10.1109/IPDPS.2011.70). URL: <http://dx.doi.org/10.1109/IPDPS.2011.70>.
- [22] Grant O Cook Jr. *ALPAL: A tool for the development of large-scale simulation codes*. Tech. rep. Lawrence Livermore National Lab., CA (USA), 1988.
- [23] ERIC C Cyr, JN Shadid, and T Wildey. "Towards efficient backward-in-time adjoint computations using data compression techniques." In: *Computer Methods in Applied Mechanics and Engineering* 288 (2015), pp. 24–44.
- [24] F Rubio Dalmau, M Hanzich, J de la Puente, and N Gutiérrez. "Lossy data compression with DCT transforms." In: *EAGE Workshop on High Performance Computing for Upstream*. 2014.
- [25] Debanjan Datta, David Appelhans, Constantinos Evangelinos, and Kirk Jordan. "An Asynchronous Two-Level Checkpointing Method to Solve Adjoint Problems on Hierarchical Memory Spaces." In: *Computing in Science & Engineering* 20.4 (2018), pp. 39–55.
- [26] Peter Deutsch and Jean-Loup Gailly. *Zlib compressed data format specification version 3.3*. Tech. rep. 1996.

- [27] Sheng Di and Franck Cappello. "Fast error-bounded lossy HPC data compression with SZ." In: *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2016, pp. 730–739.
- [28] Sheng Di, Dingwen Tao, Xin Liang, and Franck Cappello. "Efficient Lossy Compression for Scientific Data based on Point-wise Relative Error Bound." In: *IEEE Transactions on Parallel and Distributed Systems* (2018).
- [29] Patrick E Farrell, David A Ham, Simon W Funke, and Marie E Rognes. "Automated derivation of the adjoint of high-level transient finite element programs." In: *SIAM Journal on Scientific Computing* 35.4 (2013), pp. C369–C393.
- [30] Michael Fehler and P Joseph Keliher. *SEAM phase 1: Challenges of subsalt imaging in tertiary basins, with emphasis on deepwater Gulf of Mexico*. Society of Exploration Geophysicists, 2011.
- [31] Michael Förster. "Algorithmic Differentiation of Pragma-Defined Parallel Regions: Differentiating Computer Programs Containing OpenMP." PhD thesis. RWTH Aachen, 2014.
- [32] MB Giles, D Ghate, and MC Duta. "Using automatic differentiation for adjoint CFD code development." In: (2005).
- [33] A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, 2008.
- [34] Andreas Griewank. "On automatic differentiation." In: *Mathematical Programming: recent developments and applications* 6.6 (1989), pp. 83–107.
- [35] Andreas Griewank, David Juedes, and Jean Utke. "Algorithm 755: ADOL-C: a package for the automatic differentiation of algorithms written in C/C++." In: *ACM Transactions on Mathematical Software (TOMS)* 22.2 (1996), pp. 131–167.
- [36] Andreas Griewank and Andrea Walther. "Algorithm 799: Revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation." In: *ACM Transactions on Mathematical Software (TOMS)* 26.1 (2000), pp. 19–45.
- [37] Andreas Griewank and Andrea Walther. "Algorithm 799: revolve: an implementation of checkpointing for the reverse or adjoint mode of computational differentiation." In: *ACM Transactions on Mathematical Software (TOMS)* 26.1 (2000), pp. 19–45.

- [38] Audrunas Gruslys, Rémi Munos, Ivo Danihelka, Marc Lanctot, and Alex Graves. “Memory-efficient backpropagation through time.” In: *Advances in Neural Information Processing Systems*. 2016, pp. 4125–4133.
- [39] Lluís Guasch, Oscar Calderon Agudo, Meng-Xing Tang, Parashkev Nachev, and Michael Warner. “Full-waveform inversion imaging of the human brain.” In: *npj Digital Medicine* 3.1 (2020), pp. 1–12.
- [40] Laurent Hascoet and Valérie Pascual. “The Tapenade automatic differentiation tool: Principles, model, and specification.” In: *ACM Trans. Math. Softw.* 39.3 (May 2013), 20:1–20:43. ISSN: 0098-3500. DOI: [10.1145/2450153.2450158](https://doi.org/10.1145/2450153.2450158).
- [41] K. A. Hawick and D. P. Playne. “Simulation Software Generation using a Domain-Specific Language for Partial Differential Field Equations.” In: *11th International Conference on Software Engineering Research and Practice (SERP’13)*. CSTN-187. Las Vegas, USA: WorldComp, July 2013, SER3829.
- [42] Patrick Heimbach, Chris Hill, and Ralf Giering. “An efficient exact adjoint of the parallel MIT general circulation model, generated via automatic differentiation.” In: *Future Generation Computer Systems* 21.8 (2005), pp. 1356–1371.
- [43] Tom Henretty, Richard Veras, Franz Franchetti, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan. “A Stencil Compiler for Short-vector SIMD Architectures.” In: *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*. ICS ’13. Eugene, Oregon, USA: ACM, 2013, pp. 13–24. ISBN: 978-1-4503-2130-3. DOI: [10.1145/2464996.2467268](https://doi.org/10.1145/2464996.2467268). URL: <http://doi.acm.org/10.1145/2464996.2467268>.
- [44] Julien Herrmann and Guillaume Pallez. “H-Revolve: A Framework for Adjoint Computation on Synchronous Hierarchical Platforms.” In: *ACM Transactions on Mathematical Software* (2020).
- [45] Graham J Hicks. “Arbitrary source and receiver positioning in finite-difference schemes using Kaiser windowed sinc functions.” In: *Geophysics* 67.1 (2002), pp. 156–165.
- [46] Robin J Hogan. “Fast reverse-mode automatic differentiation using expression templates in C++.” In: *ACM Transactions on Mathematical Software (TOMS)* 40.4 (2014), p. 26.

- [47] Grace Murray Hopper. “The education of a computer.” In: *Proceedings of the 1952 ACM national meeting (Pittsburgh)*. ACM. 1952, pp. 243–249.
- [48] Paul Dennis Hovland. “Automatic differentiation of parallel programs.” PhD thesis. University of Illinois at Urbana-Champaign, 1997.
- [49] Jan Hückelheim, Paul D. Hovland, Michelle Mills Strout, and Jens-Dominik Müller. “Reverse-mode algorithmic differentiation of an OpenMP-parallel compressible flow solver.” In: *International Journal for High Performance Computing Applications* (2017). DOI: [10.1177/1094342017712060](https://doi.org/10.1177/1094342017712060).
- [50] Jan Hückelheim, Navjot Kukreja, Sri Hari Krishna Narayanan, Fabio Luporini, Gerard Gorman, and Paul Hovland. “Automatic differentiation for adjoint stencil loops.” In: *Proceedings of the 48th International Conference on Parallel Processing*. 2019, pp. 1–10.
- [51] Jan Hückelheim, Ziqing Luo, Fabio Luporini, Navjot Kukreja, Michael Lange, Gerard Gorman, Stephen Siegel, Matthew Dwyer, and Paul Hovland. “Towards self-verification in finite difference code generation.” In: *Proceedings of the First International Workshop on Software Correctness for HPC Applications*. ACM. 2017, pp. 42–49.
- [52] J.C. Hückelheim, P.D. Hovland, M.M. Strout, and J.-D. Müller. “Parallelizable adjoint stencil computations using transposed forward-mode algorithmic differentiation.” In: *Optimization Methods and Software* 33.4-6 (2018), pp. 672–693. DOI: [10.1080/10556788.2018.1435654](https://doi.org/10.1080/10556788.2018.1435654).
- [53] Michael Innes. “Don’t unroll adjoint: differentiating SSA-Form programs.” In: *arXiv preprint arXiv:1810.07951* (2018).
- [54] Kenneth E. Iverson. *A Programming Language*. New York, NY, USA: John Wiley & Sons, Inc., 1962. ISBN: 0-471430-14-5.
- [55] 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.” In: *CoRR abs/1609.01277* (2016). URL: <http://arxiv.org/abs/1609.01277>.

- [56] Luke Jacobs, Akhil Kodumuri, Jim James, Seongha Park, and Yongho Kim. "Multiperspective Automotive Labeling." In: *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2020, pp. 927–936.
- [57] Antony Jameson, Luigi Martinelli, and NA Pierce. "Optimum aerodynamic design using the Navier–Stokes equations." In: *Theoretical and computational fluid dynamics* 10.1-4 (1998), pp. 213–237.
- [58] John L Jones. "A survey of automatic coding techniques for digital computers." PhD thesis. Massachusetts Institute of Technology, 1954.
- [59] Shoaib Kamil, Cy Chan, Leonid Oliker, John Shalf, and Samuel Williams. "An auto-tuning framework for parallel multicore stencil computations." In: *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE. 2010, pp. 1–12.
- [60] H Knibbe, WA Mulder, CW Oosterlee, and C Vuik. "Closing the performance gap between an iterative frequency-domain solver and an explicit time-domain scheme for 3D migration on parallel architectures." In: *Geophysics* 79.2 (2014), S47–S61.
- [61] Marcel Köster, Roland Leißa, Sebastian Hack, Richard Membarth, and Philipp Slusallek. "Platform-Specific Optimization and Mapping of Stencil Codes through Refinement." In: *Proceedings of the 1st International Workshop on High-Performance Stencil Computations*. 2014, pp. 1–6.
- [62] Stefan Kronawitter and Christian Lengauer. "Polyhedral search space exploration in the ExaStencils code generator." In: *ACM Trans. Archit. Code Optim.* 15.4 (Oct. 2018), 40:1–40:25. ISSN: 1544-3566. DOI: [10.1145/3274653](https://doi.org/10.1145/3274653). URL: <http://doi.acm.org/10.1145/3274653>.
- [63] Navjot Kukreja, Jan Hückelheim, and Gerard J Gorman. "Back-propagation for long sequences: beyond memory constraints with constant overheads." In: *arXiv preprint arXiv:1806.01117* (2018).
- [64] Navjot Kukreja, Jan Hückelheim, Michael Lange, Mathias Louboutin, Andrea Walther, Simon W Funke, and Gerard Gorman. "High-level python abstractions for optimal checkpointing in inversion problems." In: *arXiv preprint arXiv:1802.02474* (2018).

- [65] Navjot Kukreja, Jan Hückelheim, Mathias Louboutin, Paul Hovland, and Gerard Gorman. “Combining checkpointing and data compression to accelerate adjoint-based optimization problems.” In: *European Conference on Parallel Processing*. Springer. 2019, pp. 87–100.
- [66] Navjot Kukreja, Jan Hueckelheim, Mathias Louboutin, John Washbourne, Paul HJ Kelly, and Gerard J Gorman. “Lossy Checkpoint Compression in Full Waveform Inversion.” In: *arXiv preprint arXiv:2009.12623* (2020).
- [67] Navjot Kukreja, Mathias Louboutin, Felipe Vieira, Fabio Luporini, Michael Lange, and Gerard Gorman. “Devito: Automated fast finite difference computation.” In: *2016 Sixth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*. IEEE. 2016, pp. 11–19.
- [68] Navjot Kukreja, Alena Shilova, Olivier Beaumont, Jan Hückelheim, Nicola Ferrier, Paul Hovland, and Gerard Gorman. “Training on the Edge: The why and the how.” In: *arXiv preprint arXiv:1903.03051* (2019).
- [69] Rubin H Landau. *A first course in scientific computing: symbolic, graphic, and numeric modeling using Maple, Java, Mathematica, and Fortran90*. Princeton University Press, 2011.
- [70] Michael Lange, Navjot Kukreja, Mathias Louboutin, Fabio Luporini, Felipe Vieira, Vincenzo Pandolfo, Paulius Velesko, Paulius Kazakas, and Gerard Gorman. “Devito: towards a generic finite difference DSL using symbolic Python.” In: *2016 6th Workshop on Python for High-Performance and Scientific Computing (PyHPC)*. IEEE. 2016, pp. 67–75.
- [71] Michael Lange, Navjot Kukreja, Fabio Luporini, Mathias Louboutin, Charles Yount, Jan Hückelheim, and Gerard J Gorman. “Optimised finite difference computation from symbolic equations.” In: *arXiv preprint arXiv:1707.03776* (2017).
- [72] Tristan van Leeuwen and Felix J Herrmann. “3D frequency-domain seismic inversion with controlled sloppiness.” In: *SIAM Journal on Scientific Computing* 36.5 (2014), S192–S217.
- [73] Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. “Differentiable programming for image processing and deep learning in Halide.” In: *ACM Transactions on Graphics (TOG)* 37.4 (2018), p. 139.

- [74] Peter Lindstrom. "Fixed-rate compressed floating-point arrays." In: *IEEE transactions on visualization and computer graphics* 20.12 (2014), pp. 2674–2683.
- [75] Peter G Lindstrom et al. *FPZIP*. Tech. rep. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2017.
- [76] Seppo Linnainmaa. "Taylor expansion of the accumulated rounding error." In: *BIT Numerical Mathematics* 16.2 (1976), pp. 146–160.
- [77] Anders Logg, Kent-Andre Mardal, Garth N. Wells, et al. *Automated Solution of Differential Equations by the Finite Element Method*. Springer, 2012. ISBN: 978-3-642-23098-1. DOI: [10.1007/978-3-642-23099-8](https://doi.org/10.1007/978-3-642-23099-8).
- [78] Mathias Louboutin* and Felix J Herrmann. "Time compressively sampled full-waveform inversion with stochastic optimization." In: *SEG Technical Program Expanded Abstracts 2015*. Society of Exploration Geophysicists, 2015, pp. 5153–5157.
- [79] Mathias Louboutin, Michael Lange, Felix J Herrmann, Navjot Kukreja, and Gerard Gorman. "Performance prediction of finite-difference solvers for different computer architectures." In: *Computers & Geosciences* 105 (2017), pp. 148–157.
- [80] Mathias Louboutin, Michael Lange, Fabio Luporini, Navjot Kukreja, Philipp A Witte, Felix J Herrmann, Paulius Velesko, and Gerard J Gorman. "Devito: an embedded domain-specific language for finite differences and geophysical exploration." In: *arXiv preprint arXiv:1808.01995* (2018).
- [81] Mathias Louboutin, Michael Lange, Fabio Luporini, Navjot Kukreja, Philipp A Witte, Felix J Herrmann, Paulius Velesko, and Gerard J Gorman. "Devito (v3. 1.0): an embedded domain-specific language for finite differences and geophysical exploration." In: *Geoscientific Model Development* 12.3 (2019), pp. 1165–1187.
- [82] Mathias Louboutin, Philipp Witte, Michael Lange, Navjot Kukreja, Fabio Luporini, Gerard Gorman, and Felix J Herrmann. "Full-waveform inversion, Part 1: Forward modeling." In: *The Leading Edge* 36.12 (2017), pp. 1033–1036.
- [83] Mathias Louboutin, Philipp Witte, Michael Lange, Navjot Kukreja, Fabio Luporini, Gerard Gorman, and Felix J Herrmann.

- "Full-waveform inversion, Part 2: Adjoint modeling." In: *The Leading Edge* 37.1 (2018), pp. 69–72.
- [84] Fabio Luporini, Michael Lange, Mathias Louboutin, Navjot Kukreja, Jan Hückelheim, Charles Yount, Philipp Witte, Paul HJ Kelly, Gerard J Gorman, and Felix J Herrmann. "Architecture and performance of Devito, a system for automated stencil computation." In: *arXiv preprint arXiv:1807.03032* (2018).
 - [85] James R Maddison, Daniel N Goldberg, and Benjamin D Goddard. "Automated calculation of higher order partial differential equation constrained derivative information." In: *SIAM Journal on Scientific Computing* 41.5 (2019), pp. C417–C445.
 - [86] OANA Marin, MICHEL Schanen, and PF Fischer. *Large-scale lossy data compression based on an a priori error estimator in a spectral element code*. Tech. rep. ANL/MCS-P6024-0616, 2016.
 - [87] Richard Membarth, Frank Hannig, Jürgen Teich, and Harald Köstler. "Towards domain-specific computing for stencil codes in HPC." In: *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion: IEEE*. 2012, pp. 1133–1138.
 - [88] Aaron Meurer et al. "SymPy: symbolic computing in Python." In: *PeerJ Computer Science* 3 (Jan. 2017), e103. ISSN: 2376-5992. DOI: [10.7717/peerj-cs.103](https://doi.org/10.7717/peerj-cs.103). URL: <https://doi.org/10.7717/peerj-cs.103>.
 - [89] Sri Hari Krishna Narayanan, Boyana Norris, and Beata Winnicka. "ADIC2: Development of a component source transformation system for differentiating C and C++." In: *Procedia Computer Science* 1.1 (2010), pp. 1845–1853.
 - [90] Isaac Newton. *Letter to Nathaniel Hawes, 25 May*. Letter. May 1694.
 - [91] Molly A. O'Neil and Martin Burtcher. "Floating-point Data Compression at 75 Gb/s on a GPU." In: *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*. ACM, 2011. DOI: [10.1145/1964179.1964189](https://doi.org/10.1145/1964179.1964189).
 - [92] Carlos Osuna, Oliver Fuhrer, Tobias Gysi, and Mauro Bianco. "STELLA: A domain-specific language for stencil methods on structured grids." In: *Poster Presentation at the Platform for Advanced Scientific Computing (PASC) Conference, Zurich, Switzerland*. 2014.

- [93] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. “Automatic differentiation in PyTorch.” In: (2017).
- [94] Bas Peters, Brendan R Smithyman, and Felix J Herrmann. “Projection methods and applications for seismic nonlinear inverse problems with multiple constraints.” In: *Geophysics* 84.2 (2019), R251–R269.
- [95] R-E Plessix. “A review of the adjoint-state method for computing the gradient of a functional with geophysical applications.” In: *Geophysical Journal International* 167.2 (2006), pp. 495–503.
- [96] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. “Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines.” In: *ACM SIGPLAN Notices* 48.6 (2013), pp. 519–530.
- [97] Florian Rathgeber, David A Ham, Lawrence Mitchell, Michael Lange, Fabio Luporini, Andrew TT McRae, Gheorghe-Teodor Bercea, Graham R Markall, and Paul HJ Kelly. “Firedrake: automating the finite element method by composing abstractions.” In: *ACM Transactions on Mathematical Software (TOMS)* 43.3 (2016), pp. 1–27.
- [98] Jarrett Revels, Miles Lubin, and Theodore Papamarkou. “Forward-mode automatic differentiation in Julia.” In: *arXiv preprint arXiv:1607.07892* (2016).
- [99] Alan Richardson. “Seismic Full-Waveform Inversion Using Deep Learning Tools and Techniques.” In: *arXiv preprint arXiv:1801.07232* (2018).
- [100] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. “Learning representations by back-propagating errors.” In: *nature* 323.6088 (1986), pp. 533–536.
- [101] Michel Schanen, Oana Marin, Hong Zhang, and Mihai Anitescu. *Asynchronous two-level checkpointing scheme for large-scale adjoints in the spectral-element solver nek5000*. Tech. rep. Argonne National Lab.(ANL), Argonne, IL (United States), 2015.

- [102] Julia Sternberg and Michael Hinze. "A memory-reduced implementation of the Newton-CG method in optimal control of nonlinear time-dependent PDEs." In: *Optimization Methods & Software* 25.4 (2010), pp. 553–571.
- [103] Kevin Stock, Martin Kong, Tobias Grosser, Louis-Noël Pouchet, Fabrice Rastello, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. "A framework for enhancing data reuse via associative reordering." In: *ACM SIGPLAN Notices*. Vol. 49. 6. ACM. 2014, pp. 65–76.
- [104] Philipp Stumm and Andrea Walther. "Multistage approaches for optimal offline checkpointing." In: *SIAM Journal on Scientific Computing* 31.3 (2009), pp. 1946–1967.
- [105] Philipp Stumm and Andrea Walther. "New algorithms for optimal online checkpointing." In: *SIAM Journal on Scientific Computing* 32.2 (2010), pp. 836–854.
- [106] William W. Symes. "Reverse time migration with optimal checkpointing." In: *GEOPHYSICS* 72.5 (2007), SM213–SM221. DOI: [10.1190/1.2742686](https://doi.org/10.1190/1.2742686). eprint: <https://doi.org/10.1190/1.2742686>. URL: <https://doi.org/10.1190/1.2742686>.
- [107] Yuan Tang, Rezaul Alam Chowdhury, Bradley C Kuszmaul, Chi-Keung Luk, and Charles E Leiserson. "The pochoir stencil compiler." In: *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*. ACM. 2011, pp. 117–128.
- [108] Dingwen Tao, Sheng Di, Xin Liang, Zizhong Chen, and Franck Cappello. "Optimizing Lossy Compression Rate-Distortion from Automatic Online Selection between SZ and ZFP." In: *arXiv preprint arXiv:1806.08901* (2018).
- [109] Yukio Umetani. "DEQSOL A numerical Simulation Language for Vector/Parallel Processors." In: *Proc. IFIP TC2/WG22, 1985* 5 (1985), pp. 147–164.
- [110] Didem Unat, Xing Cai, and Scott B Baden. "Mint: realizing CUDA performance in 3D stencil methods with annotated C." In: *Proceedings of the international conference on Supercomputing*. ACM. 2011, pp. 214–224.
- [111] Jean Utke, Uwe Naumann, Mike Fagan, Nathan Tallent, Michelle Strout, Patrick Heimbach, Chris Hill, and Carl Wunsch. "OpenAD/F: A modular open-source tool for automatic differenti-

- ation of Fortran codes." In: *ACM Transactions on Mathematical Software (TOMS)* 34.4 (2008), p. 18.
- [112] Robert Van Engelen, Lex Wolters, and Gerard Cats. "Ctadel: A generator of multi-platform high performance codes for pde-based scientific applications." In: *Proceedings of the 10th international conference on Supercomputing*. ACM. 1996, pp. 86–93.
 - [113] Jean Virieux and Stéphane Operto. "An overview of full-waveform inversion in exploration geophysics." In: *Geophysics* 74.6 (2009), WCC1–WCC26.
 - [114] Andrea Walther. "Bounding the number of processes and checkpoints needed in time-minimal parallel reversal schedules." In: *Computing* 73.2 (2004), pp. 135–154.
 - [115] Qiqi Wang, Parviz Moin, and Gianluca Iaccarino. "Minimal Repetition Dynamic Checkpointing Algorithm for Unsteady Adjoint Calculation." In: *SIAM Journal on Scientific Computing* 31.4 (May 2009), pp. 2549–2567. DOI: [10.1137/080727890](https://doi.org/10.1137/080727890).
 - [116] Philipp A Witte, Mathias Louboutin, Navjot Kukreja, Fabio Luporini, Michael Lange, Gerard J Gorman, and Felix J Herrmann. "A large-scale framework for symbolic implementations of seismic inversion algorithms in Julia." In: *Geophysics* 84.3 (2019), F57–F71.
 - [117] Philipp A Witte, Mathias Louboutin, Fabio Luporini, Gerard J Gorman, and Felix J Herrmann. "Compressive least-squares migration with on-the-fly Fourier transforms." In: *Geophysics* 84.5 (2019), pp. 1–76.
 - [118] Philipp A Witte, Mathias Louboutin, Henryk Modzelewski, Charles Jones, James Selvage, and Felix J Herrmann. "An Event-Driven Approach to Serverless Seismic Imaging in the Cloud." In: *IEEE Transactions on Parallel and Distributed Systems* 31.9 (2020), pp. 2032–2049.
 - [119] Philipp Witte, Mathias Louboutin, Keegan Lensink, Michael Lange, Navjot Kukreja, Fabio Luporini, Gerard Gorman, and Felix J Herrmann. "Full-waveform inversion, part 3: optimization." In: *The Leading Edge* 37.2 (2018), pp. 142–145.
 - [120] C. Yount. "Vector Folding: Improving Stencil Performance via Multi-dimensional SIMD-vector Representation." In: *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cy-*

- berspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*. Oct. 2015, pp. 865–870. DOI: [10.1109/HPCC-CSS-ICSS.2015.27](https://doi.org/10.1109/HPCC-CSS-ICSS.2015.27).
- [121] Yu Zhang, Houzhu Zhang, and Guanquan Zhang. “A stable TTI reverse time migration and its implementation.” In: *GEO-PHYSICS* 76.3 (2011), WA3–WA11. DOI: [10.1190/1.3554411](https://doi.org/10.1190/1.3554411). eprint: <http://dx.doi.org/10.1190/1.3554411>. URL: <http://dx.doi.org/10.1190/1.3554411>.

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both $\text{\LaTeX}$ and $\text{\LyX}$:

<https://bitbucket.org/amiede/classicthesis/>

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

<http://postcards.miede.de/>

Final Version as of February 18, 2021 (`classicthesis` version 1.1).