Time-tiling in Devito

INTERIM REPORT

Imperial College London
Department of Computing
BEng Mathematics and Computer Science Individual Project

Nicholas Sim

Supervisors: Fabio Luporini; Paul H. J. Kelly

February 8, 2018

Abstract

This is a placeholder for the abstract.

Acknowledgements

This is a placeholder for acknowledgements.

Contents

Chapter 1

Introduction

In many engineering domains, finite-difference methods are used to solve differential equations, where an exact solution may be computationally infeasible. These approximations obviate the need for equations to be solved by hand, saving considerable labour. Optimising the computations is therefore a natural extension of this task.

For scientists using differential equations in novel ways, tools which can compute partial differential equations directly are helpful, as it enables them to refine their models quickly. Such tools are less crucial to those employing fluid-dynamical models, which are relatively well-understood and optimised.

1.1 Motivation

It is inefficient and arguably undesirable for scientists principally concerned with modelling to need to understand the structures which speed up their computation. As a means of abstraction, compilers are used to transform differential equations into stencils, then code to evaluate the equations. This reduces the effort required to comprehend the calculations, aiding maintenance and even reproducibility; generating and optimising code by hand is rarely feasible or efficient [16].

Decoupling the understanding of optimisations means that domain specialists outside of software performance and compiler technology need not keep up with computer architecture. New instructions may be introduced (such as vectorisation); not for nothing are compilers able to generate code tailored to the architecture for which a program is compiled.

1.2 Devito

Devito, a tool for efficient application of finite-difference methods, is able to generate computations directly from differential equations, achieving a notion of 'vertical integration' within the modelling ecosystem. This is extremely helpful if one is experimenting with models, or continually modelling new problems, as one can change the equations used and rapidly generate and execute the relevant computation.

In modern usage, such computations may be transformed into *stencils*. There are many compilers optimising stencil kernels with *time-tiling*, an optimisation which has

not been implemented in Devito yet. It has been previously demonstrated that time-tiling has the potential to reduce the run time of 'some Devito stencil loops by up to 27.5%' [14] when applying time-tiling against a baseline of tiling in all other dimensions, which the tool was already able to perform. In this case, another tool (CLooG) was used to perform the tiling. We extend tiling to the time dimension natively in Devito to realise this speedup integrated with its other optimisations, and evaluate the claim.

1.3 Objectives and (intended) contributions

The principal objective of this project was to implement tiling over the time dimension in Devito and evaluate its performance against that of tiling restricted to the non-time dimensions.

In summary, our contributions are:

- A survey of the background required for time-tiling, and related work. A consideration of the implications of and motivation for time-tiling. (Chapter 2)
- An implementation of skewing and time-tiling in Devito. An analysis of its legality and the necessary checks to guarantee this, and a set of relevant test cases. (Chapter 3)
- An evaluation of correctness and performance of the time-tiling transformation against the original claim and other orderings, and any actions end-users may need to take to realise performance gains. (Chapter 4)
- A discussion on further implementation work and integration with other optimisations. (to be determined)

Chapter 2

Background and related work

We first provide an overview of loop nest optimisations, techniques, and analyses that we have applied, with specific reference to the polyhedral model. This forms the basis and context for the entire report, and in particular informs our overview of Devito (Section 2.3) and the survey of related work, which composes the remainder of this chapter (Section 2.4).

This project extends a well-established idea from compiler theory, *tiling*, to another dimension (time) in Devito. This has traditionally been a challenging problem, as evaluating data dependences efficiently is beset with difficulties.

2.1 Loop tiling

Optimisations on loop nests

The bulk of computation for finite difference methods lies in loops. Loop nest optimisations seek to transform a loop, possibly changing its execution order to use data locality, parallelism, or otherwise avoid unnecessary operations.

2.1.1 Insight

To exploit data locality, we must use data before it gets evicted from the cache; ideally, data is not loaded into the cache more than once. This is a complex process [11]; nevertheless reuse does occur within sufficiently small iteration spaces. We therefore contrive small iteration spaces by partitioning the original space into smaller tiles (Figure 2.1).

Loop tiling is also commonly known as *blocking*, or perhaps less transparently *strip-mine and interchange*, as tiling is typically achieved through these two transformations.

Additional motivation As a remark, the stencils resulting from finite-difference methods tend to have high *arithmetic* intensity, while tiling is used to reduce *memory* pressure. As we note later, we can decrease arithmetic intensity (at the expense of increasing memory pressure) by eliminating common sub-expressions, which would ordinarily result in redundant computation. Further, tiling may also enable other transformations, such as loop-invariant code motion, which again reduces redundant computation.

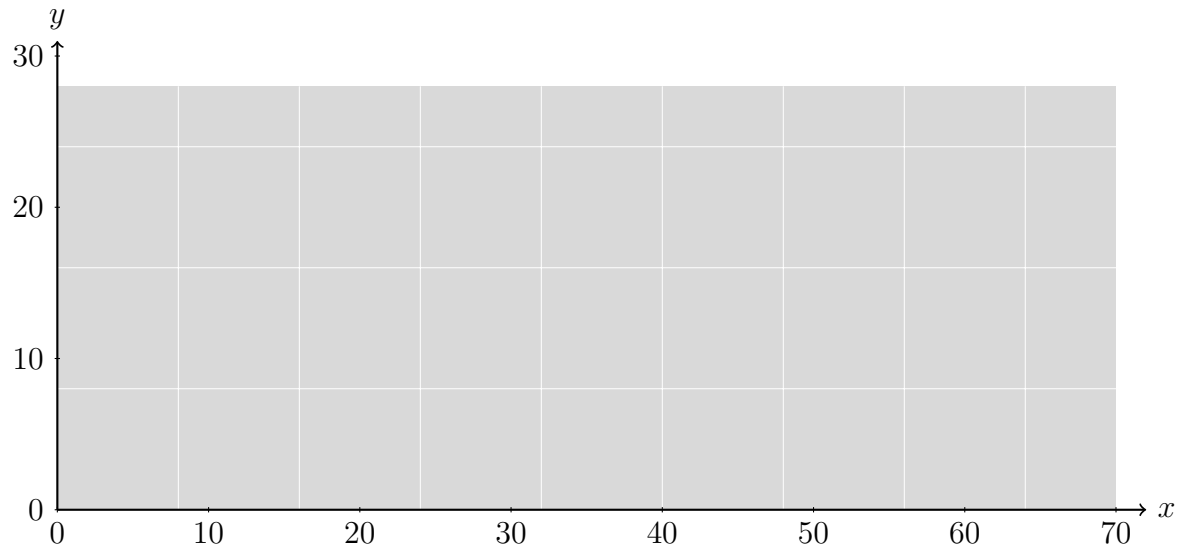

Figure 2.1: Tiles over an iteration space. Note that the tile size need not be the same in each dimension, or divide the extent of the iteration cleanly.

2.1.2 Strip-mining

```
for (int x = x_start + 1; x < x_end - 1; x++) {
  A[x] = B[x-1] + B[x+1];
}

for (int x_blk = x_start + 1; x_blk < x_end - 1; x_blk += x_blk_size) {
  for (int x = x_blk; x < min (x_end - 1, x_blk + x_blk_size); x++) {
    A[x] = B[x-1] + B[x+1]; // loop body unchanged
  }
}
```

Figure 2.2: A regular loop, then strip-mined over the variable `x`. Offsets are used on `x_start`, `x_end` to prevent out-of-bounds accesses. The `min` function avoids the need for remainder loops, in case the tile (block) size does not evenly divide the extent of the iteration. We will abbreviate the variable names in further examples.

Named after the mining practice, strip-mining involves dividing a dimension of the iteration space into strips (Figure 2.2).[1] By itself, strip-mining does not change the execution order; it is a gateway to further transformations.

We will need more loops to perform an interchange. Figure 2.3 illustrates a loop that has been strip-mined in two dimensions.

2.1.3 Loop interchange

Loop interchange is based on the observation that a change in execution order does not change the correctness of a strip-mined program. We will change the order of the loops

[1] However, you cannot divide a dimension into lateral strips, only sequential ones.

```
for (int x_blk = x_s; x_blk < x_e; x_blk += x_bs) {
  for (int x = x_blk; x < min(x_e, x_blk + x_bs); x++) {
    for (int y_blk = y_s; y_blk < y_e; y_blk += y_bs) {
      for (int y = y_blk; y < min(y_e, y_blk + y_bs); y++) {
        A[x][y] = B[x][y] + B[x][y+1];
      }
    }
  }
}
```

Figure 2.3: Strip-mining a loop nest iterating over variables `x` and `y`. Offsets have been omitted here.

to iterate over the tiles, then within them (Figure 2.4).

```
for (int x_blk = x_s; x_blk < x_e; x_blk += x_bs) {
  for (int y_blk = y_s; y_blk < y_e; y_blk += y_bs) {
    for (int x = x_blk; x < min(x_e, x_blk + x_bs); x++) {
      for (int y = y_blk; y < min(y_e, y_blk + y_bs); y++) {
        A[x][y] = B[x][y] + B[x][y+1];
      }
    }
  }
}
```

Figure 2.4: The loop nest of Figure 2.3, with the `x` and `y_blk` loops interchanged.

This is valid when each point in the iteration space does not depend on the values calculated in the same iteration. Therefore, one must be extremely careful that no data dependences cross boundaries between tiles; if they do, they must be permitted to cross only in one direction, and the tiles must be scheduled in that order.

2.2 Tiling in the time dimension

2.2.1 Motivation

Many problems involving finite difference methods are computationally bounded, rather than bounded by memory throughput. However, it is possible to reduce the operation count by exploiting the structure of expressions computed at the cost of increased memory pressure [12]. We are therefore interested to perform time-tiling to realise significant performance gains, possibly 27.5% in Devito alone [14]. This improvement is significant over the optimisation from tiling in all dimensions apart from time [3].

2.2.2 Skewing

In Section 2.1.3 we stated that interchange is valid when data dependences do not cross boundaries between tiles. This is clear, as if there are no inter-tile dependences, the tiles can be executed in any order.

Data dependences occur when two statements reference a datum, at least one of which change it. To preserve the dependence, we must preserve the order in which these statements are executed. Figure 2.5 illustrates how dependences may look in a (1-dimensional) iteration space similar to the problems we discuss later.

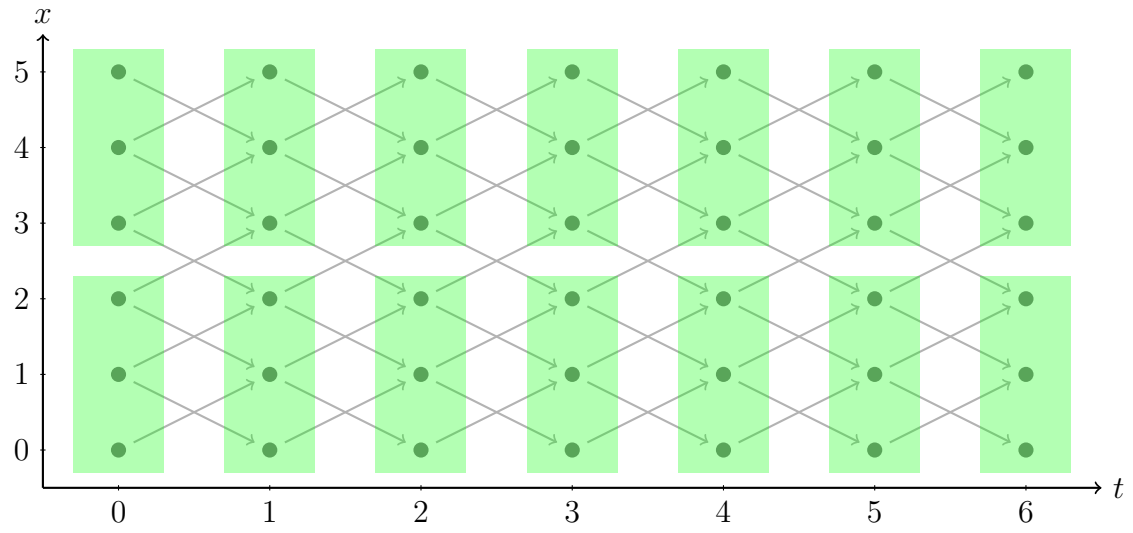

Figure 2.5: An iteration space with data dependences indicated by a forward arrow for a value derived from a dependence. It would not be valid to interchange loops over the `t` and `x` dimensions here.

We employ skewing to make the interchange valid (Figure 2.6). This solves the dependency problem [4].

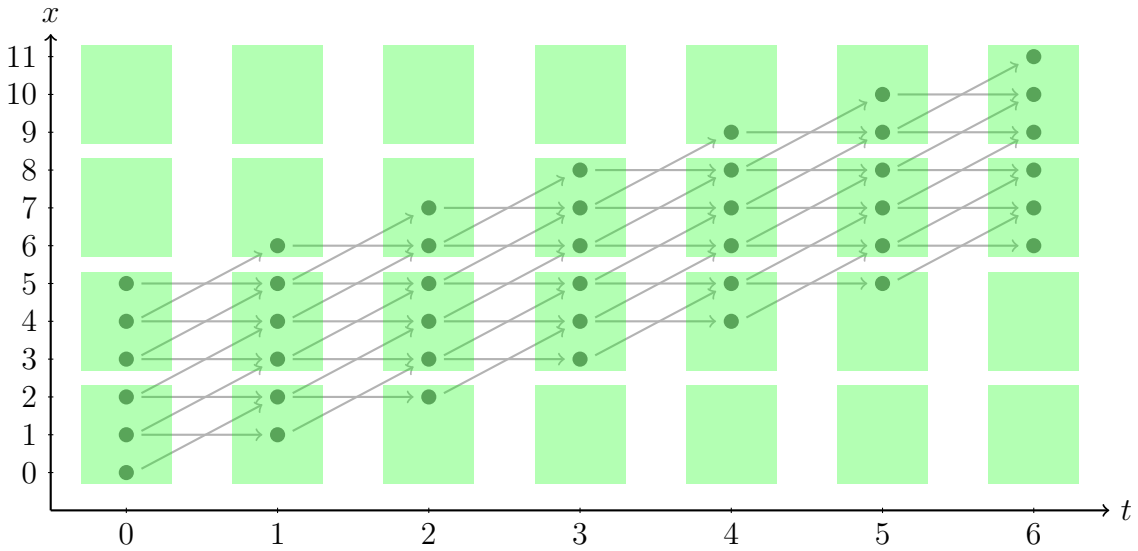

Figure 2.6: The same iteration space skewed by a factor of `t` in the `x` dimension. Note that the tiling is now valid and we can execute the tiles in either dimension first, and that we can merge tiles in the `t` dimension.

2.3 Devito

Devito [10] is a domain-specific language and code generation framework for finite-difference method computations [5]. Its main use case is building solvers for differential equations from high-level mathematical expressions, written using the symbolic library *SymPy*, and is targeted at the domain of seismic imaging.

2.3.1 Motivation

Efficient computation demands that we are able to use every optimisation at our disposal. As architecture for computation changes, introducing new instructions, GPUs and distributed systems, specialist research compilers using just a handful of related optimisations do not suffice for high-performance applications.

An equal part of optimisation is reducing the turnaround time in modelling. Providing an interface that domain specialists are familiar with reduces friction, and removes the need to construct stencils by hand. This also reduces errors by simplifying the inputs that they provide.

This combination of ease of use through the Python and *SymPy* interface and the generation of optimised and fast C code without user intervention make Devito an attractive tool.

2.3.2 Domain-specificity

Polyhedral compilers such as PLUTO [3], with its backend CLooG [2] are able to apply tiling over generic loop nests. Further, the iteration spaces which they handle are unions of convex polyhedra, again far more general than the use cases Devito is likely to encounter. This generality is not necessary when merely applying finite-difference methods to solve differential equations, not least when targeting a specific domain which uses such methods.

We can instead make use of this specificity to make informed choices in our optimisations, or streamline an auto-tuning routine. Further, we are able to combine different optimisations, at the stencil level and progressively lower levels. Finally, considering likely use cases, it may be desirable to target compilation for architectures not supported by more general compilers, such as GPUs.

2.3.3 Layers of abstraction

The separation of symbolic expressions and the underlying C code to which they are compiled is key to Devito's comprehensibility and usage. By progressive manipulation, optimisation and modification is possible at many layers. A further benefit as a more general software engineering practice is the possibility of small unit tests for individual components.

This also allows the integration of other tools, which may fit a user's needs better, and enables easy benchmarking of Devito's performance against said tools. For instance, YASK (Section 2.4.3) is being integrated as a compilation backend.

An important feature for seismic imaging is sparse-point interpolation. Devito handles this by exposing powerful lower-level APIs to allow the construction of comprehensive representations which would be tedious or impossible at higher levels of abstraction.

2.3.4 Architecture

The Devito compilation process can be divided into several stages:

1. Construction of stencil equations (symbolic kernel)
2. Grouping of expressions
3. CSE and indexing of stencil (the Devito symbolic engine, Section 3.2.1)
4. Loop optimisation and other transformations (the Devito loop engine, Section 3.2.2)
5. Code generation

Our time-tiling transformation will occur in the DSE and DLE, covered in detail in the respective sections.

2.4 Survey of related work

The following sections provide an overview of tools related to Devito and of interest to our investigation. In particular, they identify instances of time-tiling (or equivalent transformations) which have been solved and the insights required, and analyse their applicability to time-tiling in Devito.

2.4.1 Halide

Halide was conceived as a representation for image processing pipelines. Many image processing algorithms are similar to stencils: Halide specifically deals with overlapping stencils; this overlap can be compared to iteration in the time dimension.

Insight

It is possible to separate image processing algorithms ("filters") and their schedules [17]. Optimisation for an architecture then becomes an exercise in optimising the schedule and not the filters, which are reduced to kernels.[2] Halide is therefore a stencil compiler.

Modifying schedules is analogous to our loop optimisations: filters can be vectorised, tiled, interchanges are possible, etc. Halide further provides an auto-tuner which estimates, among other things, arithmetic intensity of filters and loop transformations which Devito also employs [18, 15]. Part of its analysis examines the trade-off between additional computation and memory traffic.

Applicability

The problem domain, image processing, that Halide is concerned with may appear very different to that of differential equations. However, the underlying natures of the solutions are very similar: use of stencil kernels, trade-off between redundant computation and

[2] See https://github.com/halide/CVPR2015/blob/master/blur.cpp for an example

memory pressure, etc. Analogously to time-tiling, Halide is able to compute tiles across filters.

Likewise, in Devito, *each time iteration* may contain several stencil kernels applied consecutively. To extend the analogy, Halide is concerned with tiling within *a single* timestep, while this project investigates tiling over *multiple* timesteps.

Domain-specific frameworks bring insight to a problem which general-purpose compilers may not possess; the separation of algorithm and schedule, while touted as novel, is essentially what every compiler applies during transformations such as loop interchange.

2.4.2 Pochoir

Pochoir is a compiler for stencil computations focussed on utilising parallelism and multithreading. Stencils are defined with provided C++ templates, from which the computations are generated. Pochoir implements a two-phase compilation strategy: the first involves compilation with the Pochoir template library, which ensures compliance and compatibility with the library. At this stage, one may debug the non-optimised code. The second is the optimisation phase using the Pochoir compiler[3] [20].

Insight

Cache-oblivious algorithms These seek to eliminate tuning of cuts (tile size) based on cache properties including cache sizes or replacement policies, reasoning that the correct cache-oblivious algorithm will achieve (asymptotically) the same performance [6].

Pochoir uses a cache-oblivious algorithm based on parallel cuts. These *hyperspace* and *time cuts* decompose the iteration spaces ("zoids") into smaller spaces recursively; they are chosen to improve parallelism while maintaining cache efficiency [7].

Thanks to cut dependence analysis, the resultant trapezoids can span iterations of a time loop. Note that the domain which Pochoir operates on are unions of convex polyhedra, which are more general than the hypercube-like iteration spaces described so far.

Applicability

The Cilk Plus framework, which Pochoir uses, is intended to produce optimal scheduling for parallel tasks, and cache-oblivious algorithms may be optimal up to a constant factor, which would be interesting when targeting multiple platforms. Nevertheless, significant speedup can occur with *cache-aware* algorithms, which employ tuning for multiple levels of cache [9]. Indeed, due to a difficulty in choosing suitable base case sizes for the interior trapezoids, Pochoir includes an auto-tuner for this purpose. This is especially significant to domain specialists who are experimenting and changing models, or those who are performing stencil computations over large iteration spaces.

Finally, the Cilk Plus framework is in the process of deprecation [1], and Pochoir is not presently maintained. While the concepts behind it may be sound and applicable in specific circumstances, it would be unsuitable for integration with Devito.

[3]The Pochoir compiler will also invoke the user's Cilk Plus compiler.

2.4.3 YASK

Yet Another Stencil Kernel is a framework from Intel Corp. that transforms and optimises stencil kernels, especially targeting the Xeon Phi platform [21]. Like Pochoir, YASK provides C++ templates for stencils. Similarly to Halide, it uses a genetic-algorithm based search for its auto-tuner.

It is intended to function both as a platform for domain specialists to experiment with (higher-level) optimisations, as well as for the compilation of high-performance kernels.

Insight

Stencil kernels rapidly grow complicated. Once optimisations have been applied (possibly through a specialised stencil compiler) to stencils, general-purpose compilers are less able to explore the trade-offs of further optimisation which they would ordinarily apply, such as parallelism. YASK aims to solve this by, among other things, implementing a two-stage compiler (stencils and loops) and tuning for the execution environment [23]. We draw a comparison to Devito here.

In contrast to polyhedral research compilers such as PLUTO [3], YASK has a specific focus on combining optimisations such as vector folding [22] with the loop optimisations which they perform.

Applicability

YASK was originally created to implement vector folding, a technique which simultaneously takes advantage of SIMD instructions and reduces memory bandwidth usage. This is especially relevant in Devito, which has the transformation of arithmetically intensive computations into memory intensive computations as a basis.

Further, with its emphasis on integrating different optimisations, rather than investigating them separately, YASK is a practical tool for a workflow involving stencils. There would be no need to apply a separate stencil compiler, then a polyhedral compiler, without the guarantee that the latter would understand and be able to build open the complexities introduced by the former. One might remark, however, that the same applies when constructing stencils from differential equations: instead they are reduced to a form (C++ templates) that the compiler understands.

As noted in Section 2.3, Devito is in the process of integrating YASK as a compilation backend.

2.4.4 Firedrake

Firedrake is a tool for solving partial differential equations on unstructured meshes using the *finite-element method* [19]. It is aimed at simulation and modelling of systems through PDEs, notably separating optimisations into different layers of abstraction.

FEniCS [8], another tool in the domain of finite-element solvers, separates the usage of finite-element methods and their implementation; Firedrake takes this a step further while retaining the domain-specific language and focus which it derives.

Insight

Successfully optimising the solution of PDEs involves expertise at many levels, from numerical analysis of the equations and mesh generation, to the loop optimisations discussed here. Just as we separate stencil optimisations from polyhedral ones, we should also abstract these layers.

Again, this makes the *optimised* solution of partial differential equations accessible to domain specialists without particular expertise in computer architecture, transformation of expressions into efficient kernels [13], or mesh generation.

Applicability

Through its many layers of abstraction, Firedrake provides a natural way to access lower-level structures such as kernels and the underlying data structures. Like Devito, this allows specialists to manipulate the computation not just at the highest level of the model, but also at levels which the specialist may have insight to, but would not be otherwise exposed.

While many of the problems which Firedrake deals with are not directly related to Devito (such as unstructured meshes), the concept of abstraction is sound and very relevant to software engineering. Of particular relevance is the granularity to which abstractions are helpful at the lower levels in which we are interested, the use of kernels (albeit not stencils), and the discretisation of operators.

Hence Firedrake comes far closer to Devito than the previously-discussed tools, as it covers the full process of computation, from the models and differential equations rather than the resultant kernels. Although the main optimisation that we discuss, time tiling, functions at the kernel level, Firedrake gives a broader view of the context of the problem to be solved, and the model in which Devito resides.

2.5 Summary

In this chapter, we have given an overview of the core optimisation concepts explored and terminology used throughout this work. Further, we discussed the motivation and context of Devito specifically, its overall architecture and usage, and the importance of the time-tiling transformation; these will inform our evaluation (Chapter 4). Finally, we have examined related work in the form of tools employing transformations relevant to time-tiling and stencil computations, and most importantly, the insights which they bring to our implementation (Chapter 3).

Chapter 3

Implementation

We give an overview of the existing spatial tiling transformation in Devito (Section 3.1), then discuss our implementation of time-tiling in detail (Section 3.2). The latter extends the former, while depending crucially on an initial skewing transformation. We examine the safeguards necessary to prevent errors caused by improper skewing. Finally, we consider the implications on sparse loops, one of Devito's raisons d'être.

3.1 Spatial tiling in Devito

Under the existing tiling transformation, tiling is performed over every dimension but the innermost, which benefits from vectorisation. In the generated stencils, skewing is not required, as dependencies do not cross tile boundaries, instead referencing values computed in the previous time iteration.

3.1.1 Remainder loops

When discussing loop tiling in Section 2.1, we constrained our tiles with `min` constraints. To deal with the case when the tile size does not divide the extent of the iteration space, Devito instead implements *remainder loops*, which Figures 3.1 and 3.2 illustrate.

3.2 Skewing and time-tiling

As outlined in Section 2.2, we implement time-tiling in two stages: skewing in the DSE (Section 3.2.1), and tiling in the DLE (Section 3.2.2).

3.2.1 The Devito symbolic engine (DSE)

The Devito symbolic engine is responsible for stencil optimisation and common sub-expression elimination. At this point the indices and loops have not been generated yet. Although we are working with loop transformations, we chose to perform skewing here, as skewing is a modification on the canonical indexing scheme.

In our implementation, inner loops are skewed by a factor of time, and the loop bounds skewed by the same factor, negated. This ensures that all accesses refer to the same data

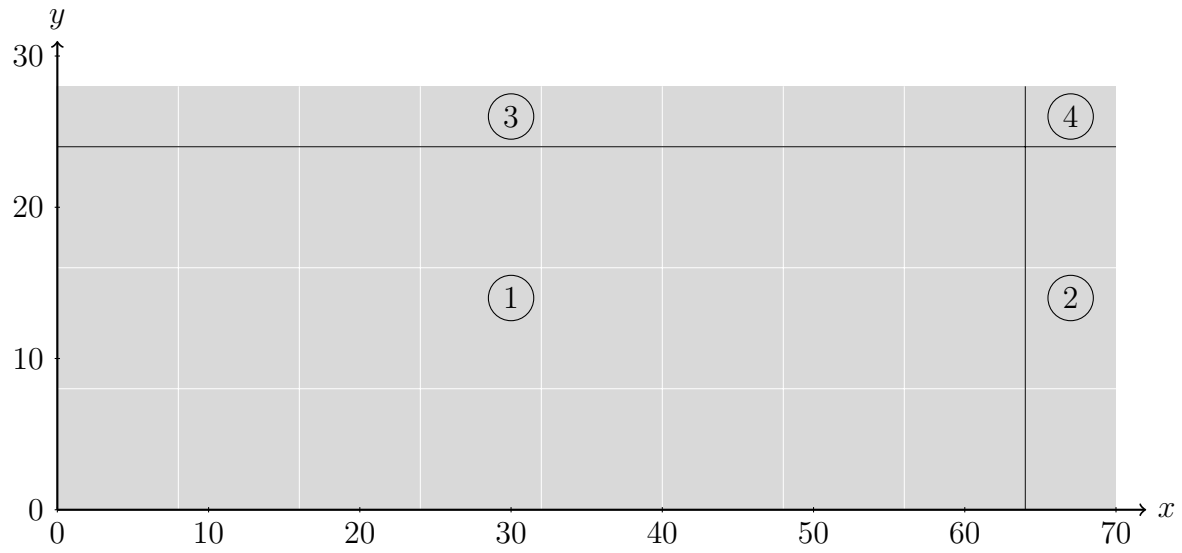

Figure 3.1: Tiles over an iteration space. Note that the tile size need not be the same in each dimension, or divide the extent of the iteration cleanly.

```
for (int x_blk = x_s; x_blk < x_e - (x_e-x_s)%x_bs; x_blk += x_bs)
  for (int y_blk = y_s; y_blk < y_e - (y_e-y_s)%y_bs; y_blk += y_bs)
    for (int x = x_blk; x < x_blk + x_bs; x++)
      for (int y = y_blk; y < y_blk + y_bs; y++)
        A[x][y] = B[x][y] + B[x][y+1];  // Nest 1

for (int x = x_e - (x_e-x_s)%x_bs; x < x_e; x++)
  for (int y_blk = y_s; y_blk < y_e - (y_e-y_s)%y_bs; y_blk += y_bs)
    A[x][y] = B[x][y] + B[x][y+1];  // Nest 2

for (int x_blk = x_s; x_blk < x_e - (x_e-x_s)%x_bs; x_blk += x_bs)
  for (int y = y_e - (y_e-y_s)%y_bs; y < y_e; y++)
    A[x][y] = B[x][y] + B[x][y+1];  // Nest 3

for (int x = x_e - (x_e-x_s)%x_bs; x < x_e; x++)
  for (int y = y_e - (y_e-y_s)%y_bs; y < y_e; y++)
    A[x][y] = B[x][y] + B[x][y+1];  // Nest 4
```

Figure 3.2: Replacement of `min` constraints with remainder loops from Figure 2.4. First the main tiles, then the remainder in `x` then `y` dimensions, and finally the remainders in both dimensions. Braces removed for concision.

as before the transformation, making it valid. Note that skewing *does not* change the execution order of the loops.

3.2.2 The Devito loop engine (DLE)

The DLE performs loop transformations including loop fission and tiling, while also marking loops to be executed in parallel or that denormal numbers should be flushed. Before it is invoked, the loops are built from the previously-manipulated expressions;

clearly it would not be possible to implement tiling before this stage.

We then change the tiling algorithm to use the `min` scheme, rather than remainder loops. As we noted in the previous section, this will not change the execution order: we now have skewed tiles on a skewed iteration space (Figure 3.3). We now need to 'straighten' the tiles by aligning the loop bounds.

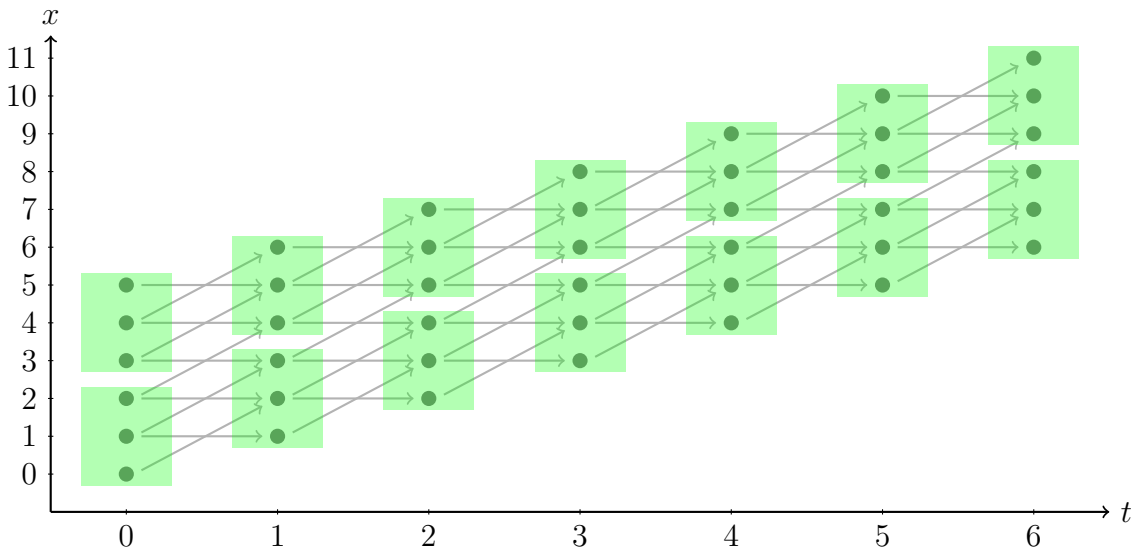

Figure 3.3: Skewed tiles on a skewed iteration space. Dependencies between blocks have not changed, and interchange is not valid.

3.3 Remaining work

The implementation work that remains can be divided into several tasks:

Straightening of tiles (end Feb.) This will make the interchange valid, enabling time-tiling. There are a number of concerns:

- The actual straightening, involving some reasoning about loop bounds, use of `max` – trivial
- Modification to tile the time loop, possibly by incorporating the `blockshape` parameter – moderate difficulty due to test cases
- Removal of 'empty' blocks – trivial; does not affect correctness but might produce speedup[1]
- Making this work with time buffering – some reasoning about skewing factor required
- Test cases – important; partially done
- Removal of test cases assuming a remainder loop structure.
- Verification that a given skewing factor is valid – deferred

[1] CLooG does it, but one has difficulty imagining that there will indeed be a noticeable speedup

Verification of skewing validity This may be challenging.[2] For the moment, skewing will work via manual input of skewing factors, hence shifting the responsibility of this onto the user. Clearly, this is not a viable strategy for code acceptance and usage, but it will (minimally) enable our evaluation.

DSE aggressive mode Currently, no reason to suppose that this cannot work with `min`-bounded instead of remainder loops. Need to prove/disprove this, and make it work. Revert to remainder loops if necessary. In particular, we must examine the references made to certain variables.[3]

Sparse loop interpolation Gather more context and understanding of the problem will be important for reasoning about how this affects time-tiling. Ideally, eventually a proof of what it does/why it does not.

Detection of when to apply skewing and calculation of the skewing factor Whether time-tiling is an appropriate optimisation, for a given stencil. Entirely beyond the scope of this work.

[2]It should be quite easy with knowledge of the data dependence vectors.

[3]Of particular interest are `blockshape` (the tile size) and dimension extents (e.g. symbolic extent vs offsets).

Chapter 4

Evaluation

Previous evaluation suggested that Devito gains considerable speedup from time-tiling. This chapter focusses on evaluating this statement with general time-tiling implemented in Devito.

Objective We want to evaluate the performance of the time-tiling transformation against that of tiling only in the other dimensions.

4.1 Functional correctness

This will be done primarily using test cases, some of which have already been written to test the features already implemented. Where possible, existing test cases are reused, when the output is expected not to vary.

Additionally, while Devito is targeted at seismic imaging, it can also be used to apply finite-difference methods to general differential equations. We are therefore able to perform testing against many natural examples, rather than being dependent on hand-crafted stencils.

4.2 Initial evaluation

We will perform evaluation against a variety of stencils, including the acoustic wave equation (AWE) stencil used in [14]. In all cases, consideration will be necessary to deal with time-dimension buffering. This evaluation would comprise both timing and memory usage, as well as examining the loop structures of the generated code.

For a more in depth analysis, we would consider memory analyses measuring cache misses and memory traffic, to ensure that improvements are indeed gleaned from time-tiling rather than other factors.

4.3 Extended evaluation

As mentioned above, Devito gives rise to many natural test cases. Implementation and time permitting, we could evaluate the benefit of time-tiling against a plethora of problems, such as inversion, or other operator orderings.

Bibliography

[1] M. Anoop. Migrate your application to use OpenMP or Intel(R) TBB instead of Intel(R) Cilk(TM) Plus. https://software.intel.com/en-us/articles/migrate-your-application-to-use-openmp-or-intelr-tbb-instead-of-intelr-cilktm-plus. Online; accessed 2018-02-02.

[2] Cédric Bastoul. Code Generation in the Polyhedral Model Is Easier Than You Think. In *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, pages 7–16, Juan-les-Pins, France, September 2004.

[3] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[4] Pierre Boulet, Alain Darte, Georges-André Silber, and Frédéric Vivien. Loop parallelization algorithms: From parallelism extraction to code generation. *Parallel Computing*, 24(3-4):421–444, May 1998.

[5] Gerard Gorman et al. Devito: Symbolic Finite Difference Computation. http://www.opesci.org/devito-public. Online; accessed 2018-01-24.

[6] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-Oblivious Algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pages 285–, Washington, DC, USA, 1999. IEEE Computer Society.

[7] Matteo Frigo and Volker Strumpen. The Cache Complexity of Multithreaded Cache Oblivious Algorithms. In *Proceedings of the Eighteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '06, pages 271–280, New York, NY, USA, 2006. ACM.

[8] Robert C. Kirby and Anders Logg. A Compiler for Variational Forms. *ACM Trans. Math. Softw.*, 32(3):417–444, September 2006.

[9] Markus Kowarschik and Christian Weiß. An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms. In *Algorithms for Memory Hierarchies – Advanced Lectures, volume 2625 of Lecture Notes in Computer Science*, pages 213–232. Springer, 2003.

[10] Navjot Kukreja, Mathias Louboutin, Felippe Vieira, Fabio Luporini, Michael Lange, and Gerard Gorman. Devito: automated fast finite difference computation. *CoRR*, abs/1608.08658, 2016.

[11] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems - ASPLOS-IV*. ACM Press, 1991.

[12] Fabio Luporini, David A. Ham, and Paul H. J. Kelly. An Algorithm for the Optimization of Finite Element Integration Loops. *ACM Transactions on Mathematical Software*, 44(1):1–26, March 2017.

[13] Fabio Luporini, Ana Lucia Varbanescu, Florian Rathgeber, Gheorghe-Teodor Bercea, J. Ramanujam, David A. Ham, and Paul H. J. Kelly. Cross-Loop Optimization of Arithmetic Intensity for Finite Element Local Assembly. *ACM Transactions on Architecture and Code Optimization*, 11(4):1–25, January 2015.

[14] Dylan McCormick. Applying the Polyhedral Model to Tile Loops in Devito. Master's thesis, Imperial College of Science, Technology and Medicine, 2017.

[15] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. Automatically Scheduling Halide Image Processing Pipelines. *ACM Trans. Graph.*, 35(4):83:1–83:11, July 2016.

[16] Kristian B. Ølgaard and Garth N. Wells. Optimizations for quadrature representations of finite element tensors through automated code generation. *ACM Transactions on Mathematical Software*, 37(1):1–23, January 2010.

[17] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines. *ACM Trans. Graph.*, 31(4):32:1–32:12, July 2012.

[18] 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 *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

[19] Florian Rathgeber, David A. Ham, Lawrence Mitchell, Michael Lange, Fabio Luporini, Andrew T. T. McRae, Gheorghe-Teodor Bercea, Graham R. Markall, and Paul H. J. Kelly. Firedrake: Automating the finite element method by composing abstractions. *ACM Transactions on Mathematical Software*, 43(3):1–27, dec 2016.

[20] 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*, SPAA '11, pages 117–128, New York, NY, USA, 2011. ACM.

[21] Charles Yount. YASK. https://01.org/yask. Online; accessed 2018-01-29.

[22] Charles Yount. Vector Folding: improving stencil performance via multi-dimensional SIMD-vector representation. In *High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), 2015 IEEE 12th International Conferen on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on*, pages 865–870. IEEE, 2015.

[23] Charles Yount, Josh Tobin, Alexander Breuer, and Alejandro Duran. YASK-yet Another Stencil Kernel: A Framework for HPC Stencil Code-generation and Tuning. In *Proceedings of the Sixth International Workshop on Domain-Specific Languages and High-Level Frameworks for HPC*, WOLFHPC '16, pages 30–39, Piscataway, NJ, USA, 2016. IEEE Press.