Temporal blocking of finite-difference stencil operators with sparse “off-the-grid” sources

George Bisbas  
*Imperial College London*  
London, UK  
g.bisbas18@imperial.ac.uk

Fabio Luporini  
*Devito Codes Ltd*  
London, UK  
fabio@devitocodes.com

Mathias Louboutin  
*Georgia Institute of Technology*  
Atlanta, GA  
mlouboutin3@gatech.edu

Rhodri Nelson  
*Imperial College London*  
London, UK  
rnelson@imperial.ac.uk

Gerard J. Gorman  
*Imperial College London*  
London, UK  
g.gorman@imperial.ac.uk

Paul HJ Kelly  
*Imperial College London*  
London, UK  
p.kelly@imperial.ac.uk

Abstract— Stencil kernels dominate a range of scientific applications including seismic and medical imaging, image processing, and neural networks. Temporal blocking is a performance optimisation that aims to reduce the required memory bandwidth of stencil computations by re-using data from the cache for multiple time steps. It has already been shown to be beneficial for this class of algorithms. However, optimising stencils for practical applications remains challenging. These computations often include of sparsely located operators, not aligned with the computational grid (“off-the-grid”). For example, our work is motivated by sources that inject a wavefield and measurements interpolating grid values. The resulting data dependencies make the adoption of temporal blocking much more challenging. We propose a methodology to inspect these data dependencies and reorder the computation, leading to performance gains in stencil codes where temporal blocking has not been applicable. We implement this novel scheme in the Devito domain-specific compiler toolchain. Devito implements a domain-specific language embedded in Python to generate optimised partial differential equation solvers using the finite-difference method from high-level symbolic problem definitions. We evaluate our scheme using isotropic acoustic, anisotropic acoustic and isotropic elastic wave propagators of industrial significance. Performance evaluation, after auto-tuning, shows that this enables substantial performance improvement through temporal blocking, over highly-optimised vectorized spatially-blocked code of up to 1.6x.

Index Terms—temporal blocking, stencil computations, code generation, partial differential equations, seismic imaging, domain-specific languages, wave-propagation

I. INTRODUCTION

Stencils are commonly encountered in scientific applications such as image processing [1], convolutional neural networks, weather forecasting [2], computational fluid dynamics, seismic [3], [4], and medical imaging [5]. We present a scheme that enables the application of temporal blocking, a common technique to enhance cache-locality [6]–[8], to a class of finite-difference (FD) [9], [10] stencil kernels where this is challenging. Typical stencil kernels are computational patterns that are usually functions of the nearest neighbouring point values. In a more general context, a stencil defines the iterative computation of an element in an n-dimensional spatial grid at time t as a function of neighbouring grid elements (space dependencies) at time t − 1, . . . , t − k (time dependencies). Conceptually, a typical stencil update in a scientific simulation has a 3-dimensional spatial iteration space and 1-dimensional time iteration space. Figure 1 illustrates a 1D stencil and its flow dependences. Each point is updated using values from the previous timestep and the right and left neighbours. Arrows illustrate the flow dependencies. Halo points (grey) are used to extend the computational domain by the size of a stencil radius. Wider stencils in 3D and their respective data dependences can be seen in Figure 2.

Fig. 1: Data dependences for stencil updates of a 1D-3pt Jacobi stencil. Grey points indicate halo area, arrows show the flow dependencies.

Fig. 2: An 8th-order 25-point 3-axis update. A point (red) at the edge of a block (blue) depends on a four-deep halo of neighbouring points which extends outside the block.
In addition to data dependencies of the kind illustrated in Figure 1, applications such as seismic wave-modelling carry additional dependencies owing to the interpolation of data not directly associated with grid points into the model (e.g. source injection). These positions, that are not aligned with the grid points are sparsely-distributed off-the-grid [11], [12] positions as shown in Figure 3a. They may also include receivers that interpolate neighbouring values to take measurements, as shown in Figure 3b. Sources and receivers are sets of sparsely-distributed off-the-grid points. We iterate over these sparsely-located sets through indirections applying their effect to the grid points, after iterating the 3D grid for stencil updates for each timestep. A loop nest structure illustrating this computation pattern is shown in Algorithm 1, where nt is the number of time steps; nx, ny, nz are the number of grid points along the x, y, z dimensions respectively. A(t, x, y, z) is the stencil kernel update and so is the space discretisation order. The src is of size nt×len(sources) holding the wavefield for each timestep for every source where sources is the structure holding the information for modelling source injection. The np is the number of points affected from a source and f is the function defining the type of interpolation (e.g. bilinear, trilinear). In Figure 3 an example of bilinear interpolation is shown where 4 points are affected in 2D space.

Fig. 3: Source injection and a receiver interpolating measurements at off-the-grid positions of a 2-D FD-grid. We assume linear interpolation.

Algorithm 1: A typical time-stepping loop nest structure for a stencil update with source injection. This stencil has one temporal and three spatial dimensions.

```
for i = 1 to nt do
  for x = 1 to nx do
    for y = 1 to ny do
      for z = 1 to nz do
        A(t, x, y, z) ≡ u[t, x, y, z] = u[t-1, x, y, z] + ∑r=1so/2 w_r
        u[t-1, x - r, y, z] + u[t-1, x + r, y, z] + u[t-1, x, y - r, z] + u[t-1, x, y + r, z] + u[t-1, x, y, z - r] + u[t-1, x, y, z + r];

for i = 1 to np do
  xs, ys, zs = map(src, i);
  u[t, xs, ys, zs] = f(src(t, s))
```

The sources set shown in Algorithm 1 provides the sparse off-the-grid coordinates for the injection. We iterate this set of coordinates that determine the affected neighbouring points. The wave amplitude is scattered to these affected points.

A. Problem overview: a running example

Space blocking [13], [14] can be applied in computational patterns similar to Algorithm 1, but applying temporal blocking is challenging as we illustrate with a 1-D example in Figure 4. White diamonds indicate off-the-grid coordinates where sparse operators are applied. Sparse operators are applied after a time iteration for the whole domain is finished. Consequently, separating the FD grid to blocks does not violate any data dependencies.

(a) Rectangular space blocking. All grid points are updated at each time-step. Sparse operators fit within space blocking as their effect is imposed after all points have been updated.

(b) Skewed/wave-front temporal blocking. Grid points are updated in “wave-fronts”. During a wave-front update, we compute values at grid points for multiple timesteps. Sparse operators may simultaneously affect points that “live” in different timesteps (i.e. source injection at \( t = 2 \) will inject to a point that is indeed in \( t = 2 \) and at the same time to a point that is at \( t = 1 \)). A point is erroneously updated with an injection before computing its stencil kernel for the corresponding timestep. We have a data dependency violation.

Fig. 4: Sparse operators do not violate data dependencies in space blocking Fig. 4a in contrast to temporal blocking Fig.4b.

In contrast, temporal blocking cannot be applied. When a sparse operator is located at an off-the-grid position between points that belong to different space blocks, data dependencies are violated yielding incorrect results. The violation occurs because updates in space may pause for a particular timestep, and computation will proceed in time rather than space. Consequently, a sparse operator may be executed and points that have not yet been updated through the kernel updates may be affected. Similarly, a point may be erroneously updated due to a forward move in time but may miss injection from a neighbouring off-the-grid operator due to space-time block constraint. Data dependencies are violated, and similar violations are raised in other variants of temporal blocking, such
as wave-front temporal blocking [15], [16] diamond temporal blocking [17], [18] and others. As an adverse side-effect, possible performance gains are limited. We aim to overcome this limitation through our contributions in this paper.

Because the set of sources is sparse, the loops generated in Algorithm 1 by modelling source injection consist of non-affine accesses as illustrated in Algorithm 1. While polyhedral tools such as PLUTO [19], [20], Polly [21], Loopy [22] and CLooG [23] manage to deal with the first uniform stencil update, they are not capable of dealing with the non-affine nature of the source injection loop nests.

The loop structure presented in Algorithm 1 is blocking time-tiling via wave-front/skewing directly as described in subsection I-A. The sparse off-the-grid nature of the source operator combined with the non-affine nature of our loop nest is limiting the capability to apply temporal blocking. The methodology presented in Section II aims to overcome this obstacle.

B. Related work

1) Improving stencil performance: stencils offer good parallelisation opportunities, ranging from Instruction Level Parallelism (ILP), SIMD register-level parallelism (SSE, AVX) to shared-memory (OpenMP, OpenACC) and task parallelism. Furthermore, distributed-memory parallelism is often employed.

In most applications of interest, stencil kernels have low operational intensity, having few floating-point operations per byte of data accessed and are therefore memory bandwidth-bound unless blocking is used. Considerable effort has been put into caches for improving stencil performance. Rescheduling the order of computations towards increased cached memory reuse can help to improve performance.

a) Spatial cache blocking: as illustrated in [13], [14], [24]–[30] we decompose FD grids into block tiles. Space tiling techniques have also been implemented for execution on GPUs. Related work includes automated split tiling with trapezoids [31], hybrid hexagonal tiling [32] and automated HPC GPU code [33], [34], [35] as well as hybrid spatial/temporal blocking on FPGAs [36].

b) Temporal cache blocking: extending cache reuse in time-dimension led to the development of temporal blocking algorithms. To further reduce cache misses, we utilise computed values in a block to update values in the next timestep where possible. While the previous timestep for a given block is stored in cache, we start computing the next timestep for this block, not depending on the requirement to compute the first timestep for the whole grid. Spatial and temporal reuse are often fused into hybrid models (equidistant locality) to harness the advantages of both methods [36]. Plenty of research has been conducted in designing and evaluating temporal blocking schemes ranging from simple skewing [6]–[8], [16], [37] and wave-front [15] to more sophisticated such as diamond [17], [18], [38]–[41]. While narrow stencil kernels evince good temporal locality, temporal blocking gains decrease when space-order increases. The technique presented in this paper enables such schedules to be used in applications with off-the-grid operators.

2) Domain Specific Languages: improving performance is essential, but usually, comes with the price of error-prone hand-optimisation. Finding ways to automate HPC code generation led to the birth of several domain-specific Languages (DSLs) like Devito [3], [4], which is used in this paper. Several DSL/compiler frameworks are working towards the automated generation of PDE solvers such as FEniCS [42] and Firedrake [43]. Halide [1], implemented as an internal DSL in C++. OpenSBLI [44] is also another framework that generated C code from Python-based high-level abstractions targeting equations written in Einstein notation. Halide is a language targeting code generation for digital image processing featuring memory locality and vectorised computation optimisations and ported to multi-core CPUs and GPUs.

Other automatic code-generation frameworks are OPS [45] for GPU code generation and YASK [46] that is a DSL to create high-performance stencil code for implementing (FD) methods and similar applications mainly targeting Intel Skylake and Knights Landing. Lift [47], achieves performance portability on parallel accelerators by combining high-level functional data-parallel language with rewrite rules which encode algorithmic and hardware-specific optimisation choices. Stella [2] and GridTools [48] are DSLs embedded in C++ implemented using template meta-programming, focusing on performance driving HPC for weather and climate simulations.

C. Contributions

Our contributions are:

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

- We implement the algorithm directly on top of the Devito DSL harnessing the power of automated code generation, thus providing a pathway to express any similar operator in a form that exploits the benefits of time tiling with only minimal coding effort.

- We evaluate our scheme using 3D stencils encountered in wave propagation applications (isotropic acoustic, isotropic elastic and anisotropic acoustic (TTI)), each having different memory and compute requirements.

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

Our work is mainly motivated by the need to automate these optimisations for a class of problems in seismic and medical imaging. Characteristic examples of such applications include full-waveform inversion (FWI) [49] and reverse time
migration (RTM) [50]. As future work we aim to deliver these optimisations as a fully automated workflow.

The rest of the paper is organized as follows: Section II presents the approach followed to solve our problem and Section IV presents an experimental evaluation of the applicability and impact of the approach. Finally, in Section V, we discuss and summarise our work and briefly refer to our plans for future work.

II. METHODOLOGY AND IMPLEMENTATION

In this section, we describe how to enable temporal blocking for wave-propagators with sparse operators. We describe the individual steps and present the details of our implementation. The whole precomputation workflow benefits from the power of the Devito DSL [3], [4] to automatically generate code and the data structures required by our scheme in its DSL. Afterwards, we manually transform the generated loops to implement a representative temporal blocking schedule, wavefront temporal blocking (WTB) [8], [51], [52].

A. Source injection precomputation

The modelling of source injection consists of the following parameters: the number of sources, their coordinates, and their wavelet time-series. We use this data to precompute their effect on an empty grid. We assume that the sources’ coordinates are constant across the time-domain.

1) Iterate sources’ coordinates and store indices of affected points: initially, we iterate over each source and inject to an empty grid for one timestep, assuming the wavefield is not zero at the first timestep. If the wavefield is zero at the first timestep, we may inject for more timesteps. Our experiments use wavefields with non-zero values at the first time-steps. Pseudocode is illustrated in 2. We use Devito to automatically generate code for this step. Our scheme is independent of the injection and interpolation type (e.g. non-linear injection). Then, we store the non-zero grid point coordinates.

Algorithm 2: Source injection is taking place over an empty grid. No PDE stencil update is happening.

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

2) Generate sparse binary mask and unique IDs: using the nonzero indices we populate two arrays. The first array (Fig.5b) is a binary integer mask with 1s at indices where u is nonzero. Ones are shown as filled bullet circles, with a green background, Fig. 5b. The second one (Fig.5c) is populated with unique ascending values for each unique point affected. It is quite common to encounter points being affected by more than one source. Figures show an x-y plane slice of the 3D grid.

Fig. 5: Illustration of the four steps through which a source impact is aligned to the computational grid. The figures show an x-y plane slice of the 3D grid.

3) Decompose wavefields: knowing the unique positions affected and their coordinates we now use Devito’s source injection mechanism to decompose the off-the-grid wavefields to on-the-grid per point wavefields. Pseudocode for that workflow presented in Algorithm 3. src_dcmp now replaces src in our computations. Instead of having sources at off-the-grid positions (Fig.5a) we now have decomposed aligned sources (Fig.5d).

Algorithm 3: Decomposing the source injection wavefields.

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

4) Fuse iteration spaces: by using the aligned structure src_dcmp, we can now fuse the source injection loop inside the kernel update iteration space. There is no sources loop as sparse data can be expressed in terms of 3D coordinates. We fuse the source injection loop at the same level as the stencil update z loop. The source mask SM is used to add (if 1) or not (if 0) the impact and SID is used to indirect to the impact values using the traversed grid coordinates. The resulting loop structure is illustrated in the following pseudocode 4 and also offers SIMD vectorization opportunities over the z2 loop.
Algorithm 4: Stencil kernel update with fused source injection.

for \( t = 1 \) to \( nt \) do
  for \( x = 1 \) to \( nx \) do
    for \( y = 1 \) to \( ny \) do
      for \( z = 1 \) to \( nz \) do
        \[ u[t, x, y, z2] \equiv A(t, x, y, z, s); \]
        \[ I[t, x, y, z, s] \equiv \{ \text{zinds} = \text{Sp_SID}[x, y, z2]; \}
        \[ u[t, x, y, z2] \equiv \{ \text{zinds} = \text{Sp_SM}[x, y, z2]; \}
        \[ u[t, x, y, z2] \equiv \{ \text{zinds} = \text{Sp_SID}[x, y, z2]; \}
    end for
  end for
end for

5) Reducing the iteration space size: the 3D structures that used to iterate through sources are, in the general case, few in number. Multiplication by zero operations are dominant as SM is as a consequence massively sparse. To alleviate this issue, we need a structure to perform only the necessary iterations in the z dimension. We aggregate nonzero occurrences along the z-axis keeping count of them in a structure named nnz_mask. We reduce the size of SM and SID cutting off z-slices where all elements are zero. For naming convention, we use Sp_SM and Sp_SID for the new structures. These structures can help to reduce the size of the iteration space and modify the second innermost z loop only to perform the necessary computation. Pseudocode for the new structure is illustrated in Algorithm 5.

![Fig. 6: We aggregate nonzero occurrences along the z axis keeping count of them. We also reduce the size of SM and SID cutting off z-slices where all elements are zero.](image)

Algorithm 5: Stencil kernel update with fused - reduced size iteration space - source injection.

for \( t = 1 \) to \( nt \) do
  for \( x = 1 \) to \( nx \) do
    for \( y = 1 \) to \( ny \) do
      for \( z = 1 \) to \( nz \) do
        \[ A(t, x, y, z, s); \]
        \[ u[t, x, y, z2] \equiv \{ \text{zinds} = \text{Sp_SID}[x, y, z2]; \}
        \[ u[t, x, y, z2] \equiv \{ \text{zinds} = \text{Sp_SM}[x, y, z2]; \}
    end for
  end for
end for

Finally, we present a methodology that aligns the source injection impact to the grid points, thus generating a structure where TB is now applicable.

B. Applying wave-front temporal blocking

We present the loop transformations to apply WTB to loop structure in Algorithm 5. In temporal blocking, we extend space blocking so that multiple timesteps are evaluated in a subset of the overall problem domain. In WTB space-time wave-fronts traverse our domain computing grid point values. For naming convention, as used in [15], we are going to use the term “block” for spatial-only grouping and “tile” when multiple temporal updates are allowed. Fig. 7 shows grid point updates as they happen in WTB. The green point is updated using orange values. This stencil kernel has a radius of size two, thus allowing a margin of 2 points to preserve data dependencies. Only two timesteps are kept in memory (for time order one problems), so the green value substitutes the yellow one in the buffer. Stencil radius affects the wave-front angle. Wave-front angle is a parameter that defines the ratio of spatial indices needed to update one point in the next time step. Thus, it gets steeper with higher stencil radius. WTB can also be applied to staggered grids. In this case, two or more grids may be updated often having inter-dependencies [15]. It is then necessary to shift the wave-front angle by an amount depending on the stencil radius of data dependencies in each loop as shown in Figure 8b.

![Fig. 7: Illustration of stencil kernel update in WTB. The green point is updated using the orange values. This stencil kernel has a space order of 4, thus allowing a margin of 2 points on the right in order not to violate data dependencies. Only two timesteps are kept in memory (for time order one problems), so the green value substitutes the yellow one in memory. Figure partially adapted from [15].](image)

After the precomputing source injection, data dependencies are now aligned with the computational grid. Applying temporal blocking is now feasible. We split the time-space iteration space into tiles as shown in Figure 8a. Each tile is then partitioned into space blocks. By applying the transformations required from wave-front temporal blocking to Algorithm 4 we
Algorithm 6: The figure shows the loop structure after applying our proposed scheme.

```
1 for z_tile in time_tiles do
2   for xtile in xtiles do
3     for ytile in ytiles do
4       for t in tile do
5         OpenMP parallelism
6         for xblk in xtile do
7           for yblk in ytile do
8             for y in yblk do
9               SIMD vectorization
10              for z = 1 to nz do
11                A(t, x - time, y - time, z, s);
12                for z2 = 1 to nnz_mask[x](y) do
13                  f(t, x - time, y - time, z2, s);
```

The next section provides more details about the kernels that were evaluated, their data dependencies and their inherent loop structure.

III. Structure of Wave-Propagation Kernels

The kernels we chose to illustrate our method and demonstrate its capabilities are wave-equation explicit time steppers. These stencils have drastically different computational properties depending on the representation of the physics [53] and therefore demonstrate both the performance and the flexibility of our method. Harnessing the DSL and the code-generation powers of Devito we can generate these kernels in C just from a few lines of Python code.

A. Isotropic acoustic

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

\[
\begin{align*}
    m(x) \frac{\partial^2 u(t, x)}{\partial t^2} - \Delta u(t, x) &= \delta(x_s)q(t) \\
    u(0, .) &= \frac{\partial u(t, x)}{\partial t}(0, .) = 0
\end{align*}
\]

(1)

where \( u(t, x) \) is the pressure wavefield, \( x_s \) is the point source position, \( q(t) \) is the source time signature, \( d(t, .) \) is the measured data at positions \( x_r \) and \( m(x) \) is the squared slowness. This equation writes in few lines with Devito symbolic API as follows:

```python
from devito import solve, Eq, Operator
update = Eq(u.forward, solve(eq, u.forward))
```

Listing 1: Wave-equation symbolic definition

The discretized acoustic wave-equation is generally memory-bound due to the low computational count of the standard Laplacian [53], [54].

B. Anisotropic acoustic

The second wave-equation kernel we consider is the most commonly used in industrial application for subsurface imaging (RTM, FWI) [55]–[59]. This equation is a pseudo-acoustic anisotropic equation that consists of a coupled system of two scalar PDEs. Unlike the most simple acoustic isotropic equation, this formulation takes into account direction-dependent propagation speeds that translate into the discretized equation into a rotated anisotropic laplacian. Such a kernel increases the operation count drastically [53]. For example, the first dimension \( x \) component of the laplacian is defined as:

\[
G_{xx} = D_x^T D_x \\
D_x = \cos(\theta) \cos(\phi) \frac{\partial}{\partial x} + \cos(\theta) \sin(\phi) \frac{\partial}{\partial y} - \sin(\theta) \frac{\partial}{\partial z}
\]

(2)

where \( \theta \) is the (spatially dependent) tilt angle (rotation around \( z \)), \( \phi \) is the (spatially dependent) azimuth angle (rotation around \( y \)). A more detailed description of the physics and discretization can be found in [55], [56].

C. Isotropic elastic

Finally, we consider the isotropic elastic equation. Unlike the two previous acoustic approximation, this equation has two major properties. First, this is a first-order system in time, which allow us to extend our work to a smaller range of local
data dependency along time. By consequence, we demonstrate
that the benefits of time-blocking, and our implementation of
it, is not limited to a single pattern along the time dimension.
Second, this equation is a coupled system of a vectorial and a
tensorial PDE, which increases the data movement drastically
(one or two versus nine state parameters) on the wavefield but
also contains non-scalar expressions of the source and receiver
expressions that involve multiple wavefields.

The elastic isotropic wave-equation, parametrized by the Lamé
parameters $\lambda, \mu$ and the density $\rho$, is defined as [60]:

$$\frac{1}{\rho} \frac{\partial v}{\partial t} = \nabla \cdot \tau$$
$$\frac{\partial \tau}{\partial t} = \lambda \text{tr} (\nabla v) I + \mu (\nabla v + (\nabla v)^T)$$

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

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

IV. EXPERIMENTAL EVALUATION

We outline in subsection IV-A the compiler setup, computer
architectures, and measurement procedure used for our per-
formance experiment. We aim to demonstrate the performance
improvement achieved by our approach, illustrate its potential
for impact on important applications, and probe the limits of
its applicability.

A. Compiler and system setup

To evaluate our scheme we used Virtual Machines in
Azure with two architectures: Intel® Xeon® Processor E5
v4 Family (formerly called Broadwell) and Intel® Xeon®
Scalable Processors (formerly called Skylake 8171M). For
our experiments, access was obtained on VMs (on Microsoft
Azure) called Standard_E16s_v3 and Standard_E32s_v3. The
first system E16s_v3 has a single socket 8-core Intel Broadwell
E5-2673 v4 CPUs with AVX2 support. Each Intel Broadwell
CPU has three levels of cache: L1 (32KB) and L2 (256KB)
caches private to each core and a 50MB L3 cache shared per
socket. The second system has single-socket 16-core
IntelSkylake Platinum 8171M CPUs with AVX512 support.
Each Intel Skylake CPU has three levels of cache: L1 (32KB)
and L2 (1MB) caches private to each core and a 35.75MB
L3 cache shared per socket. Both systems run Ubuntu 18.04.4
and compiler versions used were GCC 7.5.0* and ICC 2021.1
Beta 20200602 and relying on the OpenMP programming
model including OpenMP parallelism and SIMD vectorization.
Our wave-front temporal blocking scheme employs OpenMP-
based dynamic scheduling. Thread pinning was enabled using
the environment variables OMP_PROC_BIND (for GCC) and
KMP_AFFINITY (for ICC). Experiments were run with De-
vito v.4.2.3. The experimentation framework and instructions
on reproducibility are available at V-A.

B. Test case setup

We evaluate the performance of operators relevant to seis-
mic imaging. We model the propagation of waves for three
different models: isotropic acoustic and isotropic elastic,
anisotropic acoustic (TTI). The acoustic and elastic wave
equations are discretized with second order in time while TTI
with second-order, and we study varying space orders of 4,
8, 12. For all test cases, we use zero initial conditions and
damping fields with absorbing boundary layers. Waves are
injected using one time-dependent, spatially localized seismic
source wavelet into the subsurface model. We benchmark
velocity models of $512^3$ grid points, with a grid spacing of
10 for isotropic and elastic and 20 for TTI. Wave propagation
is modelled for $512$ms, resulting in 228 time-steps for isotropic
acoustic, 436 for isotropic elastic and 587 for anisotropic
acoustic. The time-stepping interval is selected with regards to
the Courant-Friedrichs-Lewy (CFL) condition [61], ensuring
the stability of the explicit time-stepping scheme, determined
by the highest velocity of the subsurface model and the grid
spacing.

C. Autotuning temporally blocked code

It should be noted that the parameter space for temporal
blocking schemes and especially for pipelined blocking is
enormous. We report results obtained from guidance and
experience from the state-of-the-art codes and literature [62].
Simulation codes are hard to generalize in terms of perfor-
ance as multiple configurations may be used from case to
case. The performance of an operator depends upon many
factors, such as grid shape, discretization space order, tile and
block shapes, number of other fields, number of timesteps,
platforms and others. In order to tune our C code for the
underlying hardware, we swept over the whole parameter
space to find the global performance maxima. We executed
our experiments using the best-performing tile and block sizes,
ensuring a fair comparison versus Devito ’s aggressively tuned
optimised spatially-blocked and vectorized code. Figure 9
illustrates the speedup achieved for each evaluated model for
several space order discretizations. We measured the GFlops/s
and GPoints/s performance of the operators.

D. Results discussion

Figure 9 illustrates the speedup achieved for each of our
wave propagation models. All of our models show speedup
for space order four discretization on both platforms. Acoustic
benefits the most with around 1.6x, and TTI follows with 1.4x.
Elastic wave propagation is accelerated by 1.3x on Broadwell
and 1.22x on Skylake. Concerning space order 8, which is a
commonly used practice, we observe speedups of 1.1x or more
for acoustic, elastic on Broadwell and acoustic, TTI on Sky-
lake. No significant performance gains are observed for space
order 12, excluding some gains of around 5% on Broadwell
with isotropic elastic and TTI. Figure 11 shows the roofline
performance of the isotropic acoustic kernels for the Broad-
well microarchitecture. The roofline is a cache-aware roofline
model representing cumulative (L1+L2+LLC+DRAM) traffic-based Arithmetic Intensity for application kernels \(^1\). We showcase improvement for the acoustic model breaking the roof of L3 cache.

E. Corner cases

**Number of injecting sources:** although our test cases use a single source, it is interesting to explore how our model performs with the presence of more sparse operators. Each source is decomposed in its surrounding grid points, so the overhead is affected not only by the number of sources but by number of grid points affected as well. We evaluate the overhead induced for two cases a) an increasing number of sparsely located sources: in this case, we have an increasing number of sources located at an x-y plane slice of the 3D grid, a scenario which can be of practical interest and b) an increasing number of sources densely and uniformly located all over the 3D grid. Figure 10 shows that for an isotropic acoustic wave propagation, the increasing number of sources is not affecting performance gains except in the case of really densely located sources where our scheme is not taking advantage of the structure sparsity. Still, though we observe gains of around 1.4x compared to 1.55x previously.

V. Conclusions

This paper introduced a mechanism to enable temporal blocking in stencil computations involving sparse off-the-grid operators as encountered for example with sources and receivers in seismic inversion problems. We applied wavefront temporal blocking to wave-propagators ranging from isotropic acoustic to more advanced, such as isotropic elastic and anisotropic acoustic (TTI). Experimental evaluation of the improved kernels on Broadwell and Skylake microarchitectures showed compelling evidence of substantial acceleration of at least 1.5x for low and at least 1.1x for medium space order wave-propagation kernels.

A. Code availability

An implementation of the methods described in this paper is available in a Devito fork repository under the MIT open-source license at georgebisbas/devito. See the README.md for instructions on how to reproduce the results in the paper.

B. Future work

Achieving performance improvement with high-space order kernels requires further research work. Methods like stencil retiming [63] have shown promise in alleviating this performance bottleneck, and a possible combination with temporal blocking may be promising. Another possible solution can be data layout transformations [46]. Near-term plans include...

---

\(^1\)https://software.intel.com/content/www/us/en/develop/articles/integrated-roofline-model-with-intel-advisor.html
Fig. 11: Cache-aware roofline model on Broadwell for isotropic acoustic model space order 4 (triangles), 8 (circles) and 12 (squares). Red markers show the performance of spatially blocked vectorized kernels, while yellow ones show the performance of our temporal blocking scheme.

the evaluation of our scheme on more diverse platforms (Knights Landing, ARM, GPUs). The next step then is the full automation and integration in the Devito DSL [3]. We aim to deliver automated, scalable optimisations on generated code beyond the current roofline performance limit of our kernels. The evaluation results presented in this paper are mainly motivated by the seismic imaging domain; however, the target applications are not only limited to this scope.

ACKNOWLEDGMENTS

This research is funded by the Engineering and Physical Sciences Research Council (EPSRC) grants and HiPEDS Center for Doctoral Training. The author thanks Richard M. Veras, Navjot Kukreja, John Washbourne and Giacomo La Scala for the fruitful discussions as well as the whole Devito community.

REFERENCES

[1] J. Ragan-Kelley, A. Adams, D. Sharlet, C. Barnes, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand, “Halide: decoupling algorithms from schedules for high-performance image processing,” Communications of the ACM, vol. 61, no. 1, pp. 106–115, 2017.

[2] T. Gysi, C. Osuna, O. Fuhrer, M. Bianco, and T. C. Schulteess, “Stella: A domain-specific tool for structured grid methods in weather and climate models,” in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, ser. SC ’15. New York, NY, USA: Association for Computing Machinery, 2015.

[3] F. Luporini, M. Louboutin, M. Lange, N. Kukreja, P. Witte, J. Hückelheim, C. Yount, P. H. J. Kelly, F. J. Herrmann, and G. J. Gorman, “Architecture and performance of devito, a system for automated stencil computation,” ACM Trans. Math. Softw., vol. 46, no. 1, Apr. 2020.

[4] M. Louboutin, M. Lange, F. Luporini, N. Kukreja, P. A. Witte, F. J. Herrmann, P. Velesko, and G. J. Gorman, “Devito (v3.1.0): an embedded domain-specific language for finite differences and geophysical exploitation,” Geoscience Model Development, vol. 12, no. 3, pp. 1165–1187, 2019.

[5] L. Guasch, O. C. Agudo, M.-X. Tang, P. Nachev, and M. Warner, “Full-waveform inversion imaging of the human brain,” NPJ Digital medicine, vol. 3, no. 1, pp. 1–12, 2020.

[6] D. Wonnacott, “Using time skewing to eliminate idle time due to memory bandwidth and network limitations,” in Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000, pp. 171–180, May 2000.

[7] Guohua Jin, J. Mellor-Crummey, and R. Fowler, “Increasing temporal locality with skewing and recursive blocking,” in SC ’01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing, 2001, pp. 57–57.

[8] D. Wonnacott, “Achieving scalable locality with time skewing,” International Journal of Parallel Programming, vol. 30, pp. 181–221, 2004.

[9] R. J. LeVeque, Finite difference methods for ordinary and partial differential equations: steady-state and time-dependent problems. Siam, 2007, vol. 98.

[10] G. Smith, Numerical Solution of Partial Differential Equations: Finite Difference Methods. Clarendon, 1985.

[11] H. Yan, J. Xu, and X. Zhang, “Compressed sensing radar imaging of off-grid sparse targets,” in 2015 IEEE Radar Conference (RadarCon), 2015, pp. 0690–0693.

[12] C. Poon and G. Peyré, “Degrees of freedom for off-the-grid sparse estimation,” arXiv preprint arXiv:1911.03577, 2019.

[13] M. Wolfe, “More iteration space tiling,” Proceedings of the 1989 ACM/IEEE conference on Supercomputing - Supercomputing ’89, pp. 655–664, 1989.

[14] M. E. Wolf and M. S. Lam, “A data locality optimizing algorithm,” ACM SIGPLAN Notices, vol. 26, no. 6, pp. 30–44, 1991.

[15] C. Yount and A. Duran, “Effective use of large high-bandwidth memory caches in hpc stencil computation via temporal wavefront tiling,” in 2016 7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), 2016, pp. 65–75.

[16] G. Wellein, G. Hager, T. Zeiser, M. Wittmann, and H. Fehske, “Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization,” Proceedings - International Computer Software and Applications Conference, vol. 1, pp. 579–586, 2009.

[17] I. J. Bertolacci, C. Olschanowsky, B. Harshbarger, B. L. Chamberlain, D. G. Wonnacott, and M. M. Stout, “Parameterized diamond tiling for stencil computations with chapel parallel iterators,” in Proceedings of the 29th ACM on International Conference on Supercomputing, ser. ICS ’15. New York, NY, USA: ACM, 2015, pp. 197–206.

[18] T. Malas, G. Hager, H. Ltaief, H. Stengel, G. Wellein, and D. Keyes, “Multicore-optimized wavefront diamond blocking for optimizing stencil updates,” SIAM Journal on Scientific Computing, vol. 37, no. 4, pp. C439–C464, 2015.

[19] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, “A practical automatic polyhedral parallelizer and locality optimizer,” in ACM SIGPLAN Notices, vol. 43, no. 6, ACM, 2008, pp. 101–113.

[20] U. Bondhugula, “Compiling affine loop nests for distributed-memory parallel architectures,” in SC’13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. IEEE, 2013, pp. 1–12.

[21] T. Grosser, A. Größlinger, and C. Lengauer, “Polly - performing polyhedral optimizations on a low-level intermediate representation,” Parallel Process. Lett., vol. 22, 2012.

[22] A. Klöckner, “Loo.py: transformation-based code generation for GPUs and CPUs,” in Proceedings of ARRAY ’14: ACM SIGPLAN Workshop on Libraries, Languages, and Compilers for Array Programming. Edinburgh, Scotland.: Association for Computing Machinery, 2014.

[23] C. Bastoul, “Code generation in the polyhedral model is easier than you think,” in Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques. IEEE Computer Society, 2004, pp. 7–16.

[24] J. Ramanujam, “Iteration Spaces for Multicomputers,” Computer, vol. i, 1992.

[25] J. Xue, “On tiling as a loop transformation,” Parallel Processing Letters, vol. 7, no. 04, pp. 409–424, 1997.

[26] M. Frigo and V. Strumpen, “Cache oblivious stencil computations,” in Proceedings of the 19th Annual International Conference on Supercomputing, ser. ICS ’05. New York, NY, USA: ACM, 2005, pp. 361–366.

[27] ———, “The cache complexity of multithreaded cache oblivious algorithms,” in Proceedings of the Eighteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, ser. SPAA ’06. New York, NY, USA: Association for Computing Machinery, 2006, p. 271–280.
