ARCHITECTURE AND PERFORMANCE OF DEVITO, A SYSTEM FOR AUTOMATED STENCIL COMPUTATION *

FABIO LUPORINI†, MICHAEL LANGE‡, MATHIAS LOUBOUTIN§, NAVJOT KUKREJA†, JAN HÜCKELHEIM†, CHARLES YOUNT¶, PHILIPP WITTE‖, PAUL H. J. KELLY#, GERARD J. GORMAN†, AND FELIX J. HERRMANN§

Abstract. Stencil computations are a key part of many high-performance computing applications, such as image processing, convolutional neural networks, and finite-difference solvers for partial differential equations. Devito is a framework capable of generating highly-optimized code given symbolic equations expressed in *Python*, specialized in, but not limited to, affine (stencil) codes. The lowering process – from mathematical equations down to C++ code – is performed by the Devito compiler through a series of intermediate representations. Several performance optimizations are introduced, including advanced common sub-expressions elimination, tiling and parallelization. Some of these are obtained through well-established stencil optimizers, integrated in the back-end of the Devito compiler. The architecture of the Devito compiler, as well as the performance optimizations that are applied when generating code, are presented. The effectiveness of such performance optimizations is demonstrated using operators drawn from seismic imaging applications.

Key words. Stencil, finite difference method, symbolic processing, structured grid, compiler, performance optimization

AMS subject classifications. 65N06, 68N20

1. Introduction.

Developing software for high-performance computing requires a considerable interdisciplinary effort, as it often involves domain knowledge from numerous fields such as physics, numerical analysis, software engineering and low-level performance optimization. The result is typically a monolithic application where hardware-specific optimizations, numerical methods, and physical approximations are interwoven and dispersed throughout a large number of loops, functions, files and modules. This frequently leads to slow innovation, high maintenance costs, and code that is hard to debug and port onto new computer architectures. A powerful approach to alleviate this problem is to introduce a separation of concerns and to raise the level of abstraction by using domain-specific languages (DSLs). DSLs can be used to express numerical methods using a syntax that closely mirrors how they are expressed mathematically, while a stack of compilers and libraries is responsible for automatically

*Submitted to SIAM Journal on Scientific Computing on July 10, 2018.

Funding: This work was supported by the Engineering and Physical Sciences Research Council through grants EP/I00677X/1, EP/L000407/1, EP/I012036/1], by the Imperial College London Department of Computing, by the Imperial College London Intel Parallel Computing Centre (IPCC), and by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics and Computer Science programs under contract number DE-AC02-06CH11357.

†Department of Earth Science and Engineering, Imperial College London, London, UK, (f.luporini12@imperial.ac.uk, n.kukreja@imperial.ac.uk, j.hueckelheim@imperial.ac.uk, g.gorman@imperial.ac.uk)

‡European Centre for Medium-Range Weather Forecasts, Reading, UK, (michael.lange@ecmwf.int)

§Georgia Institute of Technology, School of Computational Science and Engineering, Atlanta GA, USA, (mlouboutin3@gatech.edu, felix.herrmann@gatech.edu)

¶Intel Corporation, (chuck.yount@intel.com)

‖Seismic Laboratory for Imaging and Modeling (SLIM), The University of British Columbia, Vancouver BC, CANADA, (pwitte.slim@gmail.com)

#Department of Computing, Imperial College London, London, SW7 2AZ, UK, (p.kelly@imperial.ac.uk)

creating the optimized low-level implementation in a general purpose programming language such as C++. While the focus of this paper is on finite-difference (FD) based codes, the DSL approach has already had remarkable success in other numerical methods such as the finite-element (FE) and finite-volume (FV) method, as documented in Section 2.

This work describes the architecture of *Devito*, a system for automated stencil computations from a high-level mathematical syntax. Devito was developed with an emphasis on FD methods on structured grids. For this reason, Devito's underlying DSL has many features to simplify the specification of FD methods, as discussed in Section 3. The original motivation was to solve large-scale partial differential equations (PDEs) in the context of seismic inverse problems, where FD solvers are commonly used for solving wave equations as part of complex workflows (e.g., data inversion using adjoint-state methods and backpropogation). Devito is equally useful as a framework for other stencil computations in general; for example, computations where all array indices are affine functions of loop variables. The Devito compiler is also capable of generating arbitrarily nested, possibly irregular, loops. This key feature is needed to support many complex algorithms that are used in engineering and scientific practice, including applications from image processing, cellular automata, and machine-learning.

One of the design goals of Devito was to enable high-productivity, so it is fully written in *Python*, with easy access to solvers, optimizers, input and output, and the wide range of other libraries in the *Python* ecosystem. At the same time, Devito transforms high-level symbolic input into optimized C++ code, resulting in a performance that is competitive with hand-optimized implementations. While the examples presented in this paper focus on using Devito from a *Python* application, exploiting the full potential of on-the-fly code generation and just-in-time (JIT) compilation, a practical advantage of generating C++ as an intermediate step is that it can be also used to generate libraries for legacy software, thus enabling incremental code modernisation.

Compared to other DSL frameworks that are used in practice, Devito uses compiler technology, including several layers of intermediate representations, to perform optimizations in multiple passes. This allows Devito to perform more complex optimizations, and to better optimize the code for individual target platforms. The fact that these optimisations are performed programmatically facilitates performance portability across different computer architectures [28]. This is important, as industrial codes are often used on a variety of platforms, including clusters with multi-core CPUs, GPUs, and many-core chips spread across several compute nodes as well as various cloud platforms. Devito also performs high-level transformations for floating-point operation (FLOP) reduction based on symbolic manipulation, as well as loop-level optimizations as implemented in Devito's own optimizer, or using a third-party stencil compiler such as YASK [40]. The Devito compiler is presented in detail in Sections 4, 5 and 6.

After the presentation of the Devito compiler, we show test cases in Section 7 that are inspired by real-world seismic-imaging problems. The paper finishes with directions for future work and conclusions in Sections 8 and 9.

2. Related work. The objective of maximizing productivity and performance through frameworks based upon DSLs has long been pursued. In addition to well-known systems such as Mathematica® and Matlab®, which span broad mathematical areas, there are a number of tools specialized in numerical methods for PDEs, some

dating back to the 1970s [6, 34, 7, 35].

2.1. DSL-based frameworks for partial differential equations. One noteworthy contemporary framework centered on DSLs is FEniCS [22], which allows the specification of weak variational forms, via UFL [2], and finite-element methods, through a high-level syntax. Firedrake [30] implements the same languages as FEniCS, although it differs from it in a number of features and architectural choices. Devito is heavily influenced by these two successful projects, in particular by their philosophy and design. Since solving a PDE is often a small step of a larger workflow, the choice of *Python* to implement these software provides access to a wide ecosystem of scientific packages. Firedrake also follows the principle of graceful degradation, by providing a very simple lower-level API to escape the abstraction when non-standard calculations (i.e., unrelated to the finite-element formulation) are required. Likewise, Devito allows injecting arbitrary expressions into the finite-difference specification; this feature has been used in real-life cases, for example for interpolation in seismic imaging operators. On the other hand, a major difference is that Devito lacks a formal specification language such us UFL in FEniCS/Firedrake. This is partly because there is no systematic foundation underpinning FD, as opposed to FE which relies upon the theory of Hilbert spaces [5]. Yet another distinction is that, for performance reasons, Devito takes control of the time-stepping loop. Other examples of embedded DSLs are provided by the OpenFOAM project, with a language for FV [13], and by PyFR, which targets flux reconstruction methods [36].

2.2. High-level approaches to finite differences. Due to its simplicity, the FD method has been the subject of multiple research projects, chiefly targeting the design of effective software abstraction and/or the generation of high performance code [14, 3, 16, 21]. Devito distinguishes itself from previous work in a number of ways including: support for the principle of graceful degradation for when the DSL does not cover a feature required by an application; incorporation of a symbolic mathematics engine; using actual compiler technology rather than template-based code generation; adoption of a native *Python* interface that naturally allows composition into complex workflows such as optimisation and machine-learning frameworks.

At a lower level of abstraction there are a number of tools targeting "stencil" computation (FD codes belong to this class), whose major objective is the generation of efficient code. Some of them provide a DSL [40, 31, 43, 29], whereas others are compilers or user-driven code generation systems, often based upon a polyhedral model, such as [4, 18]. From the Devito standpoint, the aim is to harness these tools – for example by integrating them, to maximize performance portability. As a proof of concept, we shall discuss the integration of one such tool, namely YASK [40], with Devito.

2.3. Devito and seismic imaging. Devito is a general purpose system, not restricted to specific PDEs, so it can be used for any form of the wave equation. Thus, unlike software specialized in seismic exploration, like IWAVE [32] and Madagascar [12], it suffers neither from the restriction to a small set of wave equations and discretizations, nor from the lack of portability and composability typical of a pure C/Fortran environment.

2.4. Performance optimizations. The Devito compiler can introduce three types of performance optimizations: FLOPs reduction, data locality, and parallelism. Typical FLOPs reduction transformations are common sub-expressions elimination, factorization, and code motion. A thorough review is provided in [11]. To different

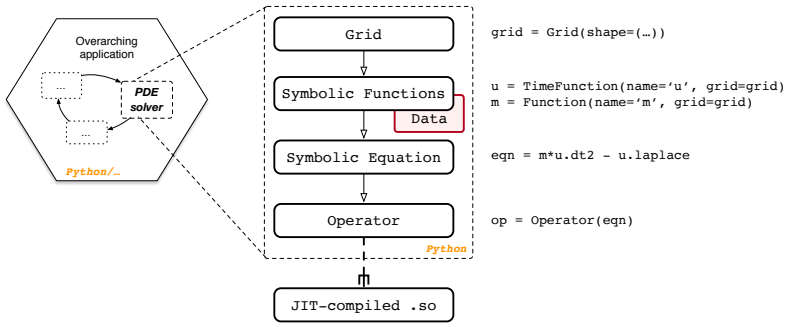


Fig. 1: The typical usage of Devito within a larger application.

extent, Devito applies all of these techniques (see Section 5.1). Particularly relevant for stencil computation is the search for redundancies across consecutive loop iterations [9, 10, 20]. This is at the core of the strategy described in Section 6, which essentially extends these ideas with optimizations for data locality. Typical loop transformations for parallelism and data locality [17] are also automatically introduced by the Devito compiler (e.g., loop blocking, vectorization); more details will be provided in Sections 5.2 and 5.3.

3. Specification of a finite-difference method with Devito. The Devito DSL allows concise expression of FD and general stencil operations using a mathematical notation. It uses *SymPy* [27] for the specification and manipulation of stencil expressions. In this section, we describe the use of Devito's DSL to build PDE solvers. Although the examples used here are for FD, the DSL can describe a large class of operations, such as convolutions or basic linear algebra operations (e.g., chained tensor multiplications).

3.1. Symbolic types. The key steps to implement a numerical kernel with Devito are shown in Figure 1. We describe this workflow, as well as fundamental features of the Devito API, using the acoustic wave equation, also known as d'Alembertian or Box operator. Its continuous form is given by:

$$
\begin{aligned}
m(x,y,z)\frac{d^2u(x,y,z,t)}{dt^2} - \nabla^2 u(x,y,z,t) &= q_s, \\
u(x,y,z,0) &= 0, \\
\frac{du(x,y,z,t)}{dt}|_{t=0} &= 0,
\end{aligned}
\tag{3.1}
$$

where the variables of this expression are defined as follows:

- $m(x,y,z) = \frac{1}{c(x,y,z)^2}$, is the parametrization of the subsurface with $c(x,y,z)$ being the speed of sound as a function of the three space coordinates (x,y,z);
- $u(x,y,z,t)$, is the spatially varying acoustic wavefield, with the additional dimension of time t;
- q_s is the source term, which is a point source in this case.

The first step towards solving this equation is the definition of a discrete computational grid, on which the model parameters, wavefields and source are defined. The computational grid is defined as a `Grid(shape)` object, where `shape` is the number

of grid points in each spatial dimension. Optional arguments for instantiating a `Grid` are `extent`, which defines the extent in physical units, and `origin`, the origin of the coordinate system, with respect to which all other coordinates are defined.

The next step is the symbolic definition of the squared slowness, wavefield and source. For this, we introduce some fundamental types.

- `Function` represents a discrete spatially varying function, such as the velocity. A `Function` is instantiated for a defined `name` and a given `Grid`.
- `TimeFunction` represents a discrete function that is both spatially varying and time dependent, such as wavefields. Again, a `TimeFunction` object is defined on an existing `Grid` and is identified by its `name`.
- `SparseFunction` and `SparseTimeFunction` represent sparse functions, that is functions that are only defined over a subset of the grid, such as a seismic point source. The corresponding object is defined on a `Grid`, identified by a `name`, and also requires the `coordinates` defining the location of the sparse points.

Apart from the grid information, these objects carry their respective FD discretization information in space and time. They also have a `data` field that contains values of the respective function at the defined grid points. By default, `data` is initialized with zeros and therefore automatically satisfies the initial conditions from equation 3.1. The initialization of the fields to solve the wave equation over a one-dimensional grid is displayed in Listing 1.

Listing 1 Setup `Functions` to express and solve the acoustic wave equation.

```
>>> from devito import Grid, TimeFunction, Function, SparseTimeFunction
>>> g = Grid(shape=(nx,), origin=(ox,), extent=(sx,))
>>> u = TimeFunction(name="u", grid=g, space_order=2, time_order=2)  # Wavefield
>>> m = Function(name="m",  grid=g)  # Physical parameter
>>> q = SparseTimeFunction(name="q", grid=g, coordinates=coordinates)  # Source
```

3.2. Discretization. With symbolic objects that represent the discrete velocity model, wavefields and source function, we can now define the full discretized wave equation. As mentioned earlier, one of the main features of Devito is the possibility to formulate stencil computations as concise mathematical expressions. To do so, we provide shortcuts to classic FD stencils, as well as the functions to define arbitrary stencils. The shortcuts are accessed as object properties and are supported by `TimeFunction` and `Function` objects. For example, we can take spatial and temporal derivatives of the wavefield `u` via the shorthand expressions `u.dx` and `u.dt` (Listing 2).

Listing 2 Example of spatial and temporal FD stencil creation.

```
>>> u.dx
-u(t, x - h_x)/(2*h_x) + u(t, x + h_x)/(2*h_x)
>>> u.dt
-u(t - dt, x)/(2*dt) + u(t + dt, x)/(2*dt)
>>> u.dt2
-2*u(t, x)/dt**2 + u(t - dt, x)/dt**2 + u(t + dt, x)/dt**2
```

Furthermore, Devito provides shortcuts for common differential operations such as the Laplacian via `u.laplace`. The full discrete wave equation can then be implemented in a single line of *Python* (Listing 3).

To solve the time-dependent wave equation with an explicit time-stepping scheme, the symbolic expression representing our PDE has to be rearranged such that it yields an update rule for the wavefield u at the next time step: $u(t + dt) = f(u(t), u(t -$

Listing 3 Expressing the wave equation.

```
>>> wave_equation = m * u.dt2 - u.laplace
>>> wave_equation
(-2*u(t, x)/dt**2 + u(t - dt, x)/dt**2 + u(t + dt, x)/dt**2)*m(x) + 2*u(t, x)/h_x
    **2 - u(t, x - h_x)/h_x**2 - u(t, x + h_x)/h_x**2
```

$dt)))$. Devito allows to rearrange the PDE expression automatically using the `solve` function, as shown in Listing 4.

Listing 4 Time-stepping scheme for the acoustic wave equation. `region=INTERIOR` ensures that the Dirichlet boundary conditions at the edges of the Grid are satisfied.

```
>>> from devito import Eq, INTERIOR, solve
>>> stencil = Eq(u.forward, solve(wave_equation, u.forward), region=INTERIOR)
>>> stencil
Eq(u(t + dt, x), -2*dt**2*u(t, x)/(h_x**2*m(x)) + dt**2*u(t, x - h_x)/(h_x**2*m(x)
    ) + dt**2*u(t, x + h_x)/(h_x**2*m(x)) + 2*u(t, x) - u(t - dt, x))
```

Note that the `stencil` expression in Listing 4 does not yet contain the point source q. This could be included as a regular `Function` which has zeros all over the grid except for a few points; this, however, would obviously be wasteful. Instead, `SparseFunction`s allow to perform operations, such as injecting a source or sampling the wavefield, at a subset of grid points determined by `coordinates`. In the case in which coordinates do not coincide with grid points, bilinear (for 2D) or trilinear (for 3D) interpolation are employed. To inject a point source into the `stencil` expression, we use the `inject` function of the `SparseTimeFunction` object that represents our seismic source (Listing 5).[1]

Listing 5 Expressing the injection of a source into a field.

```
>>> injection = q.inject(field=u.forward, expr=dt**2 * q / m)
>>> injection
[Eq(u[t + 1, INT(floor((-o_x + q_coords[p_q, 0])/h_x))], dt**2*(1 - FLOAT(-h_x*INT
    (floor((-o_x + q_coords[p_q, 0])/h_x)) - o_x + q_coords[p_q, 0])/h_x)*q[time,
    p_q]/m[INT(floor((-o_x + q_coords[p_q, 0])/h_x))] + u[t + 1, INT(floor((-o_x +
     q_coords[p_q, 0])/h_x))]),
 Eq(u[t + 1, INT(floor((-o_x + q_coords[p_q, 0])/h_x)) + 1], dt**2*FLOAT(-h_x*INT(
    floor((-o_x + q_coords[p_q, 0])/h_x)) - o_x + q_coords[p_q, 0])*q[time, p_q
    ]/(h_x*m[INT(floor((-o_x + q_coords[p_q, 0])/h_x)) + 1]) + u[t + 1, INT(floor
    ((-o_x + q_coords[p_q, 0])/h_x)) + 1])]
```

The `inject` function takes the field being updated as an input argument (in this case `u.forward`), while `expr=dt**2 * q / m` is the expression being injected. The result of the `inject` function is a list of symbolic expressions, similar to the `stencil` expression we defined earlier. As we shall see, these expressions are eventually joined together and used to create an `Operator` object – the solver of our PDE.

3.3. Boundary conditions. Simple boundary conditions (BCs), such as Dirichlet BCs, can be imposed on individual equations through special keywords (see Listing 4). For more exotic schemes, instead, the BCs need to be explicitly written (e.g., Higdon BCs [15]), just like any of the symbolic expressions defined in the Listings

[1] More complicated interpolation schemes can be defined by precomputing the grid points corresponding to each sparse point, and their respective coefficients. The result can then be used to create a `PrecomputedSparseFunction`, which behaves like a `SparseFunction` at the symbolic level.

above. For reasons of space, this aspect is not elaborated further; the interested reader may refer to [26].

3.4. Control flow. By default, the extent of a `TimeFunction` in the time dimension is limited by its time order. Hence, the shape of u in Listing 1 is $(time\_order + 1, nx) = (3, nx)$. The iterative method will then access u via modulo iteration, that is $u[t\%3, ...]$. In many scenarios, however, the entire time history, or at least periodic time slices, should be saved (e.g., for inversion algorithms). Listing 6 expands our running example with an equation that saves the content of u every 4 iterations, up to a maximum of $save = 100$ time slices.

Listing 6 Implementation of time sub-sampling.

```
>>> from devito import ConditionalDimension
>>> ts = ConditionalDimension('ts', parent=g.time_dim, factor=4)
>>> us = TimeFunction(name='us', grid=g, save=100, time_dim=ts)
>>> save = Eq(us, u)
```

In general, all equations that access `Function`s (or `TimeFunction`s) employing one or more `ConditionalDimension`s will be conditionally executed. The condition may be a number indicating how many iterations should pass between two executions of the same equation, or even an arbitrarily complex expression.

3.5. Domain, halo, and padding regions. A `Function` internally distinguishes between three regions of points.

Domain Represents the *computational domain* of the `Function` and is inferred from the input `Grid`. This includes any elements added to the *physical domain* purely for computational purposes, e.g. absorbing boundary layers.

Halo The grid points surrounding the domain region, i.e. "ghost" points that are accessed by the stencil when iterating in proximity of the domain boundary.

Padding The grid points surrounding the halo region, which are allocated for performance optimizations, such as data alignment. Normally this region should be of no interest to a user of Devito, except for precise measurement of memory allocated for each `Function`.

4. The Devito compiler. In Devito, an `Operator` carries out three fundamental tasks: generation of low-level code, JIT compilation, and execution. The `Operator` input consists of one or more symbolic equations. In the generated code, these equations are scheduled within loop nests of suitable depth and extent. The `Operator` also accepts substitution rules (to replace symbols with constant values) and optimization levels for the Devito Symbolic Engine (DSE) and the Devito Loop Engine (DLE). By default, all DSE and DLE optimizations that are known to unconditionally improve performance are automatically applied. The same `Operator` may be reused with different input data; JIT-compilation occurs only once, triggered by the first execution. Overall, this lowering process – from high-level equations to dynamically compiled and executable code – consists of multiple compiler passes, summarized in Figure 2 and discussed in the following sections (a minimal background in data dependence analysis is recommended; the unfamiliar reader may refer to a classic textbook such as [1]).

4.1. Equations lowering. In this pass, three main tasks are carried out: *indexification*, *substitution*, and *domain-alignment*.

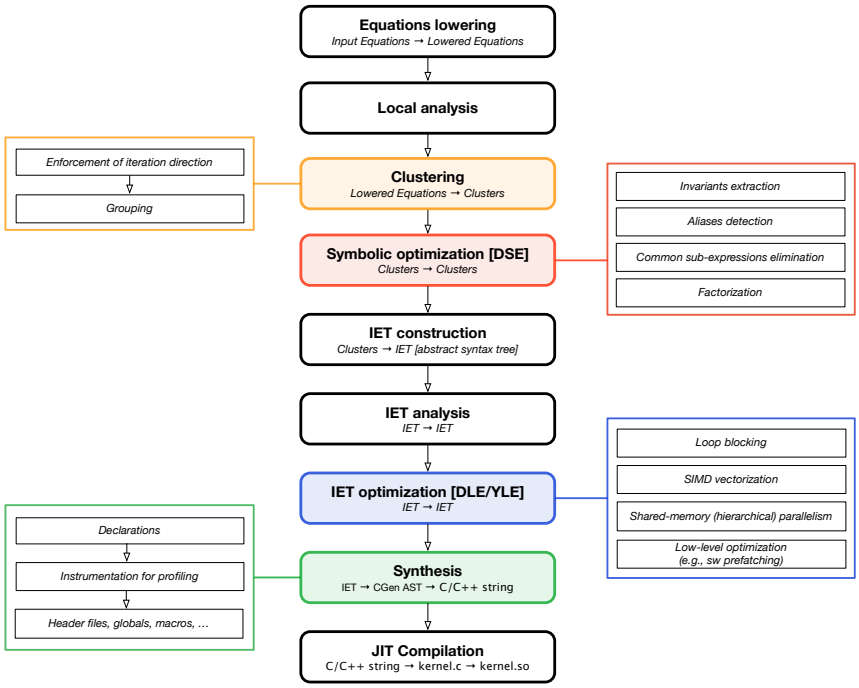


Fig. 2: Compiler passes to lower symbolic equations into shared objects through an `Operator`.

- As explained in Section 3, the input equations typically involve ore or more indexed `Function`s. The *indexification* consists of converting such objects into actual arrays. An array always keeps a reference to its originating `Function`. For instance, all accesses to u such as $u[t, x+1]$ and $u[t+1, x-2]$ would store a pointer to the same, user-defined `Function` $u(t, x)$. This metadata is exploited throughout the various compilation passes.
- During *substitution*, the user-provided substitution rules are applied. These may be given for any literal appearing in the input equations, such as the grid spacing symbols. Applying a substitution rule increases the chances of constant folding, but it makes the `Operator` less generic. The values of symbols for which no substitution rule is available are provided at execution time.
- The *domain-alignment* step shifts the array accesses deriving from `Function`s having non-empty halo and padding regions. Thus, the array accesses become logically aligned to the equation's natural domain. For instance, given the usual `Function` $u(t, x)$ having two points on each side of the x halo region, the array accesses $u[t, x]$ and $u[t, x+2]$ are transformed, respectively, into $u[t, x+2]$ and $u[t, x+4]$. When $x = 0$, therefore, the values $u[t, 2]$ and $u[t, 4]$ are fetched, representing the first and third points in the computational domain.

4.2. Local analysis. The lowered equations are analyzed to collect information relevant for the `Operator` construction and execution. In this pass, an equation is inspected "in isolation", ignoring its relationship with the rest of the input. The following metadata are retrieved and/or computed:

- input and output `Function`s;
- `Dimension`s, which are topologically ordered based on how they appear in the various array index functions; and
- two notable `Space`s: the iteration space, `ISpace`, and the data space, `DSpace`.

A `Space` is a collection of points given by the product of n compact intervals on $\mathbb{Z}$. With the notation $d[o_m, o_M]$ we indicate the compact interval $[d_m + o_m, d_M + o_M]$ over the `Dimension` d, in which d_m and d_M are parameters (specialized only at runtime), while o_m and o_M are known integers. For instance, $[x[0,0], y[-1,1]]$ describes a rectangular two-dimensional space over x and y, whose points are given by the Cartesian product $[x_m, x_M] \times [y_m - 1, y_M + 1]$. The `ISpace` and `DSpace` are two special types of `Space`. They usually span different sets of `Dimension`s. A `DSpace` may have `Dimension`s that do not appear in an `ISpace`, in particular those that are accessed only via integer indices. Likewise, an `ISpace` may have `Dimension`s that are not part of the `DSpace`, such as a reduction axis. Further, an `ISpace` also carries, for each `Dimension`, its iteration direction.

As an example, consider the equation *stencil* in Listing 4. Immediately we see that input $= [u, m]$, output $= [u]$, `Dimension`$s = [t, x]$. The compiler constructs the `ISpace` $[t[0,0]^+, x[0,0]^*]$. The first entry $t[0,0]^+$ indicates that, along t, the equation should run between $t_m + 0$ and $t_M + 0$ (extremes included) in the *forward* direction, as indicated by the symbol $+$. This is due to the fact that there is a flow dependence in t, so only a unitary positive stepping increment (i.e., $t = t + 1$) allows a correct propagation of information across consecutive iterations. The only difference along x is that the iteration direction is now arbitrary, as indicated by $*$. The `DSpace` is $[t[0,1], x[0,0]]$; intuitively, the entry $t[0,1]$ is used right before running an `Operator` to provide a default value for t_M – in particular, t_M will be set to the largest possible value that does not cause out-of-domain accesses (i.e., out-of-bounds array accesses).

4.3. Clustering. A `Cluster` is a sequence of equations having (i) same `ISpace`, (ii) same control flow (i.e., same `ConditionalDimension`s), and (iii) no dimension-carried "true" anti-dependences among them.

As an example, consider again the setup in Section 3. The equation *stencil* cannot be "clusterized" with the equations in the *injection* list as their `ISpace`s are different. On the other hand, the equations in *injection* can be grouped together in the same `Cluster` as (i) they have same `ISpace` $[t[0,0]^*, p_q[0,0]^*]$, (ii) same control flow, and (iii) there are no true anti-dependences among them (note that the second equation in *injection* does write to $u[t+1, ...]$, but as explained later this is in fact a reduction, that is a "false" anti-dependence).

4.3.1. Iteration direction. First, each equation is assigned a new `ISpace`, based upon a *global* analysis. Any of the iteration directions that had been marked as "arbitrary" ($*$) during local analysis may now be enforced to *forward* ($+$) or *backward* ($-$). This process exploits data dependence analysis.

For instance, consider the flow dependence between *stencil* and the *injection* equations. If we want u to be up-to-date when evaluating *injection*, then we eventually need all equations to be scheduled sequentially within the t loop. For this, the `ISpace`s of the *injection* equations are specialized by enforcing the direction *forward* along the `Dimension` t. The new `ISpace` is $[t[0,0]^+, p_q[0,0]^*]$.

Algorithm 1 illustrates how the enforcement of iteration directions is achieved in general. Whenever a clash is detected (i.e., two equations with `ISpace` $[d[0,0]^+, ...]$ and $[d[0,0]^-, ...]$), the original direction determined by the local analysis pass is kept (lines 11 and 13), which will eventually lead to generating different loops.

Algorithm 1: Clustering: enforcement of iteration directions (pseudocode).

```
Input: A sequence of equations ℰ.
Output: A sequence of equations ℰ′ with altered ISpace.
// Map each dimension to a set of expected iteration directions
mapper ← DETECT_FLOW_DIRECTIONS(ℰ);
for e in ℰ do
    for dim, directions in mapper do
        if len(directions) == 1 then
            // No ambiguity
            forced[dim] ← directions.pop();
        else if len(directions) == 2 then
            // No ambiguity as long as one of the two items is /Any/
            try
                directions.remove(Any);
                forced[dim] ← directions.pop();
            except
                forced[dim] ← e.directions[dim];
        else
            forced[dim] ← e.directions[dim];
        end if
    end for
    ℰ′.append(e._rebuild(directions=forced))
end for
return ℰ′
```

4.3.2. Grouping. This step performs the actual clustering, checking `ISpace`s and anti-dependences, as well as handling control flow. The procedure is shown in Algorithm 2; some explanations follow.

Algorithm 2: Clustering: grouping expressions into `Cluster`s (pseudocode)

```
Input: A sequence of equations ℰ.
Output: A sequence of clusters 𝒞.
𝒞 ← ClusterGroup();
for e in ℰ do
    grouped ← false;
    for c in reversed(𝒞) do
        anti, flow ← GET_DEPENDENCES(c, e);
        if e.ispace == c.ispace and anti.carried is empty then
            c.add(e);
            grouped ← true;
            break;
        else if anti.carried is not empty then
            c.atomics.update(anti.carried.cause);
            break;
        else if flow.cause.intersection(c.atomics) then
            // cannot search across earlier clusters
            break;
    end for
    if not grouped then
        𝒞.append(Cluster(e));
    end if
end for
𝒞 ← CONTROL_FLOW(𝒞);
return 𝒞
```

- Robust data dependence analysis, capable of tracking flow-, anti-, and output-

dependencies at the level of array accesses, is necessary. In particular, it must be able to tell whether two generic *array accesses* induce a dependence or not. The data dependence analysis performed is conservative; that is, a dependence is always assumed when a test is inconclusive. Dependence testing is based on the standard Lamport test [1]. In Algorithm 2, data dependence analysis is carried out by the function GET_DEPENDENCES.

- If an anti-dependence is detected along a `Dimension` i, then i is marked as *atomic* – meaning that no further clustering can occur along i. This information is also exploited by later `Operator` passes (see Section 4.5).
- Reductions, and in particular increments, are treated specially. They represent a special form of anti-dependence, as they do not break clustering. GET_DEPENDENCES detects reductions and removes them from the set of anti-dependencies.
- Given the sequence of equations $[E_1, E_2, E_3]$, it is possible that E_3 can be grouped with E_1, but not with its immediate predecessor E_2 (e.g., due to a different `ISpace`). However, this can only happen when there are no flow or anti-dependences between E_2 and E_3; i.e. when the `if` commands at lines 10 and 13 are not entered, thus allowing the search to proceed with the next equation. This optimization was originally motivated by gradient operators in seismic imaging kernels.
- The routine CONTROL_FLOW, omitted for brevity, creates additional `Cluster`s if one or more `ConditionalDimension`s are encountered. These are tracked in a special `Cluster` field, *guards*, as also required by later passes (see Section 4.5).

4.4. Symbolic optimization. The DSE – Devito Symbolic Engine – is a macro-pass reducing the *arithmetic strength* of `Cluster`s (e.g., their operation count). It consists of a series of passes, ranging from standard common sub-expression elimination (CSE) to more advanced rewrite procedures, applied individually to each `Cluster`. The DSE output is a new ordered sequence of `Cluster`s: there may be more or fewer `Cluster`s than in the input, and both the overall number of equations as well as the sequence of arithmetic operations might differ. The DSE passes are discussed in Section 5.1. We remark that the DSE only operates on `Cluster`s (i.e., on collections of equations); there is no concept of "loop" at this stage yet. However, by altering `Cluster`s, the DSE has an indirect impact on the final loop-nest structure.

4.5. IET construction. In this pass, the intermediate representation is lowered to an Iteration/Expression Tree (IET). An IET is an abstract syntax tree in which `Iteration`s and `Expression`s – two special node types – are the main actors. Equations are wrapped within `Expression`s, while `Iteration`s represent loops. Loop nests embedding such `Expression`s are constructed by suitably nesting `Iteration`s. Each `Cluster` is eventually placed in its own loop (`Iteration`) nest, although some (outer) loops may be shared by multiple `Cluster`s.

Consider again our running acoustic wave equation example. There are three `Cluster`s in total: C_1 for *stencil*, C_2 for *save*, and C_3 for the equations in *injection*. We use Algorithm 3 – an excerpt of the actual cluster scheduling algorithm – to explain how this sequence of `Cluster`s is turned into an IET. Initially, the *schedule* list is empty, so when C_1 is handled two nested `Iteration`s are created (line 15), respectively for the `Dimension`s t and x. Subsequently, C_2's `ISpace` and the current *schedule* are compared (line 5). It turns out that t appears among C_2's guards, hence the for loop is exited at line 12 without inspecting the second and last iteration. Thus, $index = 1$, and the previously built `Iteration` over t is reused. Finally, when processing C_3, the for loop is exited at the second iteration due to line 6, since $p_q! = x$. Again, the t `Iteration` is reused, while a new `Iteration` is constructed for

Algorithm 3: An excerpt of the cluster scheduling algorithm, turning a list (of `Clusters`) into a tree (IET). Here, the fact that different `Clusters` may eventually share some outer `Iterations` is highlighted.

Input: A sequence of `Clusters` $\mathcal{C}$.
Output: An Iteration/Expression Tree.

```
schedule ← list();
for c in C do
    root ← None;
    index ← 0;
    for i0, i1 in zip(c.ispace, schedule) do
        if i0 != i1 or i0.dimension in c.atomics then
            break;
        end if
        root ← schedule[i1];
        index ← index + 1;
        if i0.dim in c.guards then
            break;
        end if
    end for
    ⟨build as many Iterations as Dimensions in c.ispace[index:] and nest them inside root⟩;
    ⟨update schedule⟩;
    ⟨...⟩
end for
```

the `Dimension` p_q. Eventually, the constructed IET is as in Listing 7.

Listing 7 Graphical representation of the IET produced by the cluster scheduling algorithm for the running example.

```
for t = t_m to t_M:
 |-- for x = x_m to x_M:
 |     |-- <Eq(u[t+1,x], ...)>
 |
 |-- if t % 4 == 0
 |     |-- for x = x_m to x_M:
 |           |-- <Eq(us[t/4, x], ...)>
 |
 |-- for p_q = p_q_m to p_q_M:
       |-- <Eq(u[t+1,f(p_q)], ...)>
       |-- <Eq(u[t+1,g(p_q)], ...)>
```

4.6. IET analysis. The newly constructed IET is analyzed to determine `Iteration` properties such as `sequential`, `parallel`, and `vectorizable`, which are then attached to the relevant nodes in the IET. These properties are used for loop optimization, but only by a later pass (see Section 4.7). To determine whether an `Iteration` is `parallel` or `sequential`, a fundamental result from compiler theory is used – the i-th `Iteration` in a nest comprising n `Iterations` is parallel if for all dependences D, expressed as distance vectors $D = (d_0, ..., d_{n-1})$, either $(d_1, ..., d_{i-1}) > 0$ or $(d_1, ..., d_i) = 0$ [1].

4.7. IET optimization. This macro-pass transforms the IET for performance optimization. Apart from runtime performance, this pass also optimizes for rapid JIT compilation with the underlying C compiler. A number of loop optimizations are introduced, including loop blocking, minimization of remainder loops, SIMD vectorization, shared-memory (hierarchical) parallelism via OpenMP, software prefetching. These will be detailed in Section 5. A *backend* (see Section 4.9) might provide its own

loop optimization engine.

4.8. Synthesis, dynamic compilation, and execution. Finally, the IET adds variable declarations and header files, as well as instrumentation for performance profiling, in particular, to collect execution times of specific code regions. Declarations are injected into the IET, ensuring they appear as close as possible to the scope in which the relative variables are used, while honoring the OpenMP semantics of private and shared variables. To generate C code, a suitable tree visitor inspects the IET and incrementally builds a *CGen* tree [19], which is ultimately translated into a string and written to a file. Such files are stored in a software cache of Devito-generated `Operator`s, JIT-compiled into a shared object, and eventually loaded into the *Python* environment. The compiled code has a default entry point (a special function), which is called directly from *Python* at `Operator` application time.

4.9. Operator specialization through backends. In Devito, a *backend* is a mechanism to specialize data types as well as `Operator` passes, while preserving software modularity (inspired by [25]).

One of the main objectives of the backend infrastructure is promoting software composability. As explained in Section 2, there exist a significant number of interesting tools for stencil optimization, which we may want to integrate with Devito. For example, one of the future goals is to support GPUs, and this might be achieved by writing a new backend implementing the interface between Devito and third-party software specialized for this particular architecture.

Currently, two backends exist:

`core` the default backend, which relies on the DLE for loop optimization.

`yask` an alternative backend using the YASK stencil compiler to generate optimized C++ code for Intel® Xeon® and Intel® Xeon Phi™ architectures [40]. Devito transforms the IET into a format suitable for YASK, and uses its API for data management, JIT-compilation, and execution. Loop optimization is performed by YASK through the YASK Loop Engine (YLE).

The *core* and *yask* backends share the compilation pipeline in Figure 2 until the loop optimization stage.

5. Automated performance optimizations. As discussed in Section 4, Devito performs symbolic optimizations to reduce the arithmetic strength of the expressions, as well as loop transformations for data locality and parallelism. The former are implemented as a series of compiler passes in the DSE, while for the latter there currently are two alternatives, namely the DLE and the YLE (depending on the chosen execution backend).

Devito abstracts away the single optimizations passes by providing users with a certain number of optimization levels, called "modes", which trigger pre-established sequences of optimizations – analogous to what general-purpose compilers do with, for example, `-O2` and `-O3`. In Sections 5.1, 5.2, and 5.3 we describe the individual passes provided by the DSE, DLE, and YLE respectively, while in Section 7.1 we explain how these are composed into modes.

5.1. DSE - Devito Symbolic Engine. The DSE passes attempt to reduce the arithmetic strength of the expressions through FLOP-reducing transformations [11]. They are illustrated in Listings 8-11, which derive from the running example used throughout the article. A detailed description follows.

- **Common sub-expression elimination (CSE).** Two implementations are available: one based upon *SymPy*'s `cse` routine and one built on top of more basic

SymPy routines, such as `xreplace`. The former is more powerful, being aware of key arithmetic properties such as associativity; hence it can discover more redundancies. The latter is simpler, but avoids a few critical issues: (i) it has a much quicker turnaround time; (ii) it does not capture integer index expressions (for increased quality of the generated code); and (iii) it tries not to break factorization opportunities. A generalized common sub-expressions elimination routine retaining the features and avoiding the drawbacks of both implementations is still under development. By default, the latter implementation is used when the CSE pass is selected.

Listing 8 An example of common sub-expressions elimination.

```
>>> 9.0*dt*dt*u[t, x + 1] - 18.0*dt*dt*u[t][x + 2] + 9.0*dt*dt*u[t, x + 3]
temp0 = dt*dt
9.0*temp0*u[t, x + 1] - 18.0*temp0*u[t][x + 2] + 9.0*temp0*u[t, x + 3]
```

- **Factorization.** This pass visits each expression tree and tries to factorize FD weights. Factorization is applied without altering the expression structure (e.g., without expanding products) and without performing any heuristic search across groups of expressions. This choice is based on the observation that a more aggressive approach is only rarely helpful (never in the test cases in Section 7), while the increase in symbolic processing time could otherwise be significant. The implementation exploits the *SymPy* `collect` routine. However, while `collect` only searches for common factors across the immediate children of a single node, the DSE implementation recursively applies `collect` to each `Add` node (i.e., an addition) in the expression tree, until the leaves are reached.

Listing 9 An example of FD weights factorization.

```
>>> 9.0*temp0*u[t, x + 1] - 18.0*temp0*u[t][x + 2] + 9.0*temp0*u[t, x + 3]
9.0*temp0*(u[t, x + 1] + u[t, x + 3]) - 18.0*temp0*u[t][x + 2]
```

- **Extraction.** The name stems from the fact that sub-expressions matching a certain condition are pulled out of a larger expression, and their values are stored into suitable scalar or tensor temporaries. For example, a condition could be "*extract all time-varying sub-expressions whose operation count is larger than a given threshold*". A tensor temporary may be preferred over a scalar temporary if the intention is to let the *IET construction* pass (see Section 4.5) place the pulled sub-expressions within an outer loop nest. Obviously, this comes at the price of additional storage. This peculiar effect – trading operations for memory – will be thoroughly analyzed in Sections 6 and 7.

Listing 10 An example of time-varying sub-expressions extraction. Only sub-expressions performing at least one floating-point operation are extracted.

```
>>> 9.0*temp0*(u[t, x + 1] + u[t, x + 3]) - 18.0*temp0*u[t][x + 2]
temp1[x] = u[t, x + 1] + u[t, x + 3]
9.0*temp0*temp1[x] - 18.0*temp0*u[t][x + 2]
```

- **Detection of aliases.** The Alias-Detection Algorithm implements the most advanced DSE pass. In essence, an alias is a sub-expression that is redundantly computed at multiple iteration points. Because of its key role in the Cross-Iteration

Redundancy-Elimination algorithm, the formalization of the Alias-Detection Algorithm is postponed until Section 6.

Listing 11 An example of alias detection.

```
>>> 9.0*temp0*u[t, x + 1] - 18.0*temp0*u[t][x + 2] + 9.0*temp0*u[t, x + 3]
temp[x] = 9.0*temp0*u[t, x]
temp[x + 1] - 18.0*temp0*u[t][x + 2] + temp[x + 3]
```

5.2. DLE - Devito Loop Engine. The DLE transforms the IET via classic loop optimizations for parallelism and data locality [17]. These are summarized below.

- **SIMD Vectorization.** Implemented by enforcing compiler auto-vectorization via special `pragma`s from the OpenMP 4.0 language. With this approach, the DLE aims to be performance-portable across different architectures. However, this strategy causes a significant fraction of vector loads/stores to be unaligned to cache boundaries, due to the stencil offsets. As we shall see, this is a primary cause of performance loss.
- **Loop Blocking.** Also known as "tiling", this technique implemented by replacing `Iteration` trees in the IET. The current implementation only supports blocking over fully-parallel `Iteration`s. Blocking over dimensions characterized by flow- or anti-dependences, such as the time dimension in typical explicit finite difference schemes, is instead work in progress (this would require a preliminary pass known as loop skewing; see Section 8 for more details). On the other hand, a feature of the present implementation is the capability of blocking across particular *sequences* of loop nests. This is exploited by the Cross-Iteration Redundancy-Elimination algorithm, as shown in Section 6.3. To determine an optimal block shape, an `Operator` resorts to empirical auto-tuning.
- **Parallelism.** Shared-memory parallelism is introduced by decorating `Iteration`s with suitable OpenMP `pragma`s. The OpenMP `static` scheduling is used. Normally, only the outermost fully-parallel `Iteration` is annotated with the parallel `pragma`. However, heuristically nested fully-parallel `Iteration`s are `collapsed` if the core count is greater than a certain threshold. This pass also ensures that all array temporaries allocated in the scope of the parallel `Iteration` are declared as `private` and that storage is allocated where appropriate (stack, heap).

Summarizing, the DLE applies a sequence of typical stencil optimizations, aiming to reach a minimum level of performance across different architectures. As we shall see, the effectiveness of this approach, based on simple transformations, deteriorates on architectures strongly conceived for hierarchical parallelism. This is one of the main reasons behind the development of the `yask` backend (see Section 4.9), described in the following section.

5.3. YLE - YASK Loop Engine. "YASK" (Yet Another Stencil Kernel) is an open-source C++ software framework for generating high-performance implementations of stencil codes for Intel® Xeon® and Intel® Xeon Phi™ processors. Previous publications on YASK have discussed its overall structure [40] and its application to the Intel® Xeon Phi™ x100 family (code-named Knights Corner) [37] and Intel® Xeon Phi™ x200 family (code-named Knights Landing) [38, 33] many-core CPUs. Unlike Devito, it does not expose a symbolic language to the programmer or create stencils from finite-difference approximations of differential equations. Rather, the programmer provides simple declarative descriptions of the stencil equations using a C++ or Python API. Thus, Devito operates at a level of abstraction higher than

that of YASK, while YASK provides performance portability across Intel architectures and is more focused on low-level optimizations. Following is a sample of some of the optimizations provided by YASK:[2]

- **Vector-folding.** In traditional SIMD vectorization, such as that provided by a vectorizing compiler, the vector elements are arranged sequentially along the unit-stride dimension of the grid, which must also be the dimension iterated over in the inner-most loop of the stencil application. Vector-folding is an alternative data-layout method whereby neighboring elements are arranged in small *multi-dimensional* tiles. Figure 3 illustrates three ways to pack eight double-precision floating-point values into a 512-bit SIMD register. Figure 3a shows a traditional 1D "in-line" layout, and 3b and 3c show alternative 2D and 3D "folded" layouts. Furthermore, these tiles may be ordered in memory in a dimension independent of the dimensions used in vectorization [37]. The combination of these two techniques can significantly increase overlap and reuse between successive stencil-application iterations, reducing the memory-bandwidth demand. For stencils that are bandwidth-bound, this can provide significant performance gains [37, 33].
- **Software prefetching.** Many high-order or staggered-grid stencils require many streams of data to be read from memory, which can overwhelm the hardware prefetchers. YASK can be directed to automatically generate software prefetch instructions to improve the cache hit rates, especially on Xeon Phi CPUs.
- **Hierarchical parallelism.** Dividing the spatial domain into tiles to increase temporal cache locality is a common stencil optimization as discussed earlier. When implementing this technique, sometimes called "cache-blocking", it is typical to assign each thread to one or more small rectilinear subsets of the domain in which to apply the stencil(s). However, if these threads share caches, one thread's data will often evict data needed later by another thread, reducing the effective capacity of the cache. YASK addresses this by employing two levels of OpenMP parallelization: the outer level of parallel loops are applied across the cache-blocks, and an inner level is applied across sub-blocks within those tiles. In the case of the Xeon Phi, the eight hyper-threads that share each L2 cache can now cooperate on filling and reusing the data in the cache, rather than evicting each other's data.

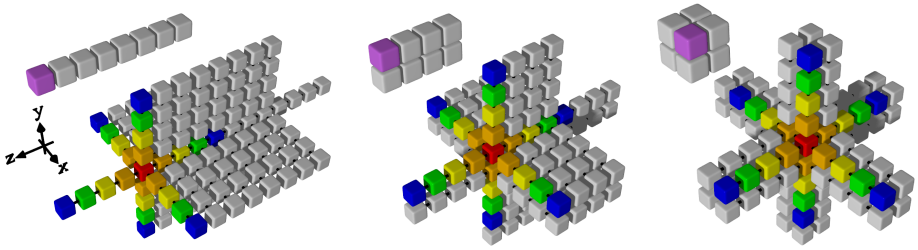
YASK also provides other optimizations, such as temporal wave-fron tiling, as well as MPI support. These features, however, are not exploited by Devito yet. The interested reader may refer to [38, 39].

To obtain the best of both tools, we have integrated the YASK framework into the Devito package. In essence, the Devito `yask` backend exploits the intermediate representation of an `Operator` to generate YASK kernels. This process is based upon sophisticated compiler technology. In *Devito v3.1*, roughly 70% of the Devito API is supported by the `yask` backend[3].

6. The Cross-Iteration Redundancy-Elimination Algorithm. Aliases, or "cross-iteration redundancies" (informally introduced in Section 5.1), in FD operators depend on the differential operators used in the PDE(s) and the chosen discretization scheme. From a performance viewpoint, the presence of aliases is a non-issue as long as the operator is memory-bound, while it becomes relevant in kernels with a high arithmetic intensity. In Devito, the Cross-Iteration Redundancy-Elimination (CIRE) algorithm attempts to remove aliases with the goal of reducing the operation count. As

[2]Not all YASK features are currently used by Devito.

[3]At the time of writing, reaching feature-completeness is one the major on-going development efforts



a. $1 \times 1 \times 8$ 1D fold b. $1 \times 2 \times 4$ 2D fold c. $2 \times 2 \times 2$ 3D fold

Fig. 3: Various folds of 8 elements [37]. The smaller diagram in the upper-left of each sub-figure illustrates a single SIMD layout, and the larger diagram shows the input values needed for a typical 25-point stencil, as from an 8th-order finite-difference approximation of an isotropic acoustic wave. Note that the $1 \times 1 \times 8$ 1D fold corresponds to the traditional in-line vectorization.

shown in Section 7, the CIRE algorithm has considerable impact in seismic imaging kernels. The algorithm is implemented through the orchestration of multiple DSE and DLE/YLE passes, namely *extraction of candidate expressions (DSE)*, *detection of aliases (DSE)*, *loop blocking (DLE/YLE)*.

6.1. Extraction of candidate expressions. The criteria for extraction of candidate sub-expressions are:

- Any *maximal time-invariant* whose operation count is greater than $Thr_0 = 10$ (floating point arithmetic only). The term "maximal" means that the expression is not embedded within a larger time-invariant. The default value $Thr_0 = 10$, determined empirically, provides systematic performance improvements in a series of seismic imaging kernels. Transcendental functions are given a weight in the order of tens of operations, again determined empirically.
- Any *maximal time-varying* whose operation count is greater than $Thr_1 = 10$. Such expressions often lead to aliases, since they typically result from taking spatial and time derivatives on `TimeFunction`s. In particular, cross-derivatives are a major cause of aliases.

  This pass leverages the *extraction* routine described in Section 5.1.

6.2. Detection of aliases. To define the concept of aliasing expressions, we first need to formalize the notion of *translated operands*. Here, an operand is regarded as the arithmetic product of a scalar value (or "coefficient") and one or more indexed objects. An indexed object is characterized by a label (i.e., its name), a vector of n dimensions, and a vector of n displacements (one for each dimension). We say that an operand o_1 is translated with respect to an operand o_0 if o_0 and o_1 have same coefficient, label, and dimensions, and if their displacement vectors are such that one is the translation of the other (in the classic geometric sense). For example, the operand $2 * u[x, y, z]$ is translated with respect to the operand $2 * u[x+1, y+2, z+3]$ since they have same coefficient (2), label (u), and dimensions ($[x, y, z]$), while the displacement vectors $[0, 0, 0]$ and $[1, 2, 3]$ are expressible by means of a translation.

Now consider two expressions e_0 and e_1 in fully-expanded form (i.e., a non-nested sum-of-operands). We say that e_0 is an alias of e_1 if the following conditions hold:

- the operands in e_0 (e_1) are expressible as a translation of the operands in e_1 (e_0);
- the same arithmetic operators are applied to the involved operands.

For example, consider $e = u[x] + v[x]$, having two operands $u[x]$ and $v[x]$; then:

- **u[x-1] + v[y-1]** is *not* an alias of e, due to a different dimension vector.
- **u[x] + w[x]** is *not* an alias of e, due to a different label.
- **u[x+2] + v[x]** is *not* an alias of e, since a translation cannot be determined.
- **u[x+2] + v[x+2]** is an alias of e, as the operands $u[x+2]$ and $v[x+2]$ can be expressed as a translation of $u[x]$ and $v[x]$ respectively, with $T(o_d) = o_d + \mathbf{2}$ and o_d representing the displacement vector of an operand.

The relation "e_0 is an alias of e_1" is an equivalence relation, as it is at the same time reflexive, symmetric, and transitive. Thanks to these properties, the turnaround times of the Alias-Detection Algorithm are extremely quick (less than 2 seconds running on an Intel® Xeon® E5-2620 v4 for the challenging `tti` test case with `so`=16, described in Section 7.2), despite the $O(n^2)$ computational complexity (with n representing the number of candidate expressions, see Section 6.1).

Algorithm 4 highlights the fundamental steps of the Alias-Detection Algorithm. In the worst case scenario, all pairs of candidate expressions are compared by applying the aliasing definition given above. Aggressive pruning, however, is applied to minimize the cost of the search. The algorithm uses some auxiliary functions: (i) CALCULATE_DISPLACEMENTS returns a mapper associating, to each candidate, its displacement vectors (one for each indexed object); (ii) COMPARE_OPS(e_1, e_2) evaluates to true if e_1 and e_2 perform the same operations on the same operands; (iii) IS_TRANSLATED(d_1, d_2) evaluates to true if the displacement vectors in d_2 are pairwise-translated with respect to the vectors in d_1 by the same factor. Together, (ii) and (iii) are used to establish whether two expressions alias each other (line 8).

Eventually, m sets of aliasing expressions are determined. For each of these sets $G_0, ..., G_{m-1}$, a *pivot* – a special aliasing expression – is constructed. This is the key for operation count reduction: the pivot p_i of $G_i = \{e_0, ..., e_{k-1}\}$ will be used in place of $e_0, ..., e_{k-1}$ (thus obtaining a reduction proportional to k). A simple example is illustrated in Listing 11.

Algorithm 4: The Alias-Detection Algorithm (pseudocode).

```
Input: A sequence of expressions ℰ.
Output: A sequence of Alias objects 𝒜.
1  displacements ← CALCULATE_DISPLACEMENTS(ℰ);
2  𝒜 ← list();
3  unseen ← list(ℰ);
4  while unseen is not empty do
5      top ← unseen.pop();
6      G = Alias(top);
7      for e in unseen do
8          if COMPARE_OPS(top, e) and IS_TRANSLATED(displacements[top], displacements[e]) then
9              G.append(e);
10             unseen.remove(e);
11         end if
12     end for
13     𝒜.append(G)
14 end while
15 return 𝒜
```

Several optimizations for data locality, not shown in Algorithm 4, are also applied. The interested reader may refer to the documentation and the examples of *Devito v3.1* for more details; below, we only mention the underlying ideas.

- The pivot of G_i is *constructed*, rather than selected out of $e_0, ..., e_{k-1}$, so that

it could coexist with as many other pivots as possible within the same `Cluster`. For example, consider again Listing 11: there are infinite possible pivots `temp[x + s] = 9.0*temp0*u[t, x + s]`, and the one with $s = 0$ is chosen. However, this choice is not random: the Alias-Detection Algorithm chooses pivots based on a global optimization strategy, which takes into account all of the m sets of aliasing expressions. The objective function consists of choosing s so that multiple pivots will have identical `ISpace`, and thus be scheduled to the same `Cluster` (and, eventually, to the same loop nest).

- Conservatively, the chosen pivots are assigned to array variables. A second optimization pass, called *index bumping and array contraction* in *Devito v3.1*, attempts to turn these arrays into scalar variables, thus reducing memory consumption. This pass is based on data dependence analysis, which essentially checks whether a given pivot is required only within its `Cluster` or by later `Cluster`s as well. In the former case, the optimization is applied.

6.3. Loop blocking for working-set minimization. In essence, the CIRE algorithm trades operation for memory – the (array) temporaries to store the aliases. From a run-time performance viewpoint, this is convenient only in arithmetic-intensive kernels. Unsurprisingly, we observed that storing temporary arrays spanning the entire grid rarely provides benefits (e.g., only when the operation count reductions are exceptionally high). We then considered the following options.

1. Capturing redundancies arising along the innermost dimension only. Thus, only scalar temporaries would be necessary. This approach presents three main issues, however: (i) only a small percentage of all redundancies are captured; (ii) the implementation is non-trivial, due to the need for circular buffers in the generated code; (iii) SIMD vectorization is affected, since inner loop iterations are practically serialised. Some previous articles followed this path [9, 10].
2. A generalization of the previous approach: using both scalar and array temporaries, without searching for redundancies across the outermost loop(s). This mitigates issue (i), although the memory pressure is still severely affected. Issue (iii) is also unsolved. This strategy was discussed in [20].
3. Using loop blocking. Redundancies are sought and captured along all available dimensions, although they are now assigned to array temporaries whose size is a function of the block shape. A first loop nest produces the array temporaries, while a subsequent loop nest consumes them, to compute the actual output values. The block shape should be chosen so that writes and reads to the temporary arrays do not cause high latency accesses to the DRAM. An illustrative example is shown in Listing 12.

The CIRE algorithm uses the third approach, based on cross-loop-nest blocking. This pass is carried out by the DLE, which can introduce blocking over sequences of loops (see Section 5.2).

7. Performance evaluation. We outline in Section 7.1 the compiler setup, computer architectures, and measurement procedure that we used for our performance experiments. Following that, we outline the physical model and numerical setup that define the problem being solved in Section 7.2. This leads to performance results, presented in Sections 7.3 and 7.4.

7.1. Compiler and system setup. We analyse the performance of generated code using enriched roofline plots. Since the DSE transformations may alter the

Listing 12 The loop nest produced by the CIRE algorithm for the example in Listing 11. Note that the block loop (line 2) wraps both the producer (line 3) and consumer (line 5) loops. For ease of read, unnecessary information are omitted.

```
1 for t = t_m to t_M:
2   for xb = x_m to x_M, xb += blocksize:
3     for x = xb to xb + blocksize + 3, x += 1
4       temp[x] = 9.0*temp0*u[t, x]
5     for x = xb to xb + blocksize; x += 1:
6       u[t+1,x,y] = ... + temp[x + 1] - 18.0*temp0*u[t][x + 2] + temp[x + 3] + ...
```

operation count by allocating extra memory, only by looking at GFlops/s performance and runtime jointly can a quality measure of code syntheses be derived.

For the roofline plots, Stream TRIAD was used to determine the attainable memory bandwidth of the node. Two peaks for the maximum floating-point performance are shown: the ideal peak, calculated as

$$\#[\text{cores}] \cdot \#[\text{avx units}] \cdot \#[\text{vector lanes}] \cdot \#[\text{FMA ports}] \cdot [\text{ISA base frequency}]$$

and a more realistic one, given by the LINPACK benchmark. The reported runtimes are the minimum of three runs (the variance was negligible). The model used to calculate the operational intensity assumes that the time-invariant `Function`s are reloaded at each time iteration. This is a more realistic setting than a "compulsory-traffic-only" model (i.e., an infinite cache).

We had exclusive access to two architectures: an Intel® Xeon® Platinum 8180 (formerly code-named Skylake) and an Intel® Xeon Phi™ 7250 (formerly code-named Knights Landing), which will be referred to as `skl8180` and `knl7250`. Thread pinning was enabled with the program `numactl`. The Intel® compiler `icc version 18.0` was used to compile the generated code. The experiments were run with *Devito v3.1* [41]. The experimentation framework with instructions for reproducibility is available at [42]. All floating point operations are performed in single precision, which is typical for seismic imaging applications.

Any arbitrary sequence of DSE and DLE/YLE transformations is applicable to an `Operator`. Devito, provides three preset optimization sequences, or "modes", which vary in aggressiveness and affect code generation in three major ways:

- the time required by the Devito compiler to generate the code,
- the potential reduction in operation count, and
- the potential amount of additional memory that might be allocated to store (scalar, tensor) temporaries.

A more aggressive mode might obtain a better operation count reduction than a non-aggressive one, although this does not necessarily imply a better time to solution as the memory pressure might also increase. The three optimization modes – `basic`, `advanced`, and `aggressive`– apply the same sequence of DLE/YLE transformations, which includes OpenMP parallelism, SIMD vectorization, and loop blocking. However, they vary in the number, type, and order of DSE transformations. In particular,
`basic` enables common sub-expressions elimination only;
`advanced` enables `basic`, then factorization, extraction of time-invariant aliases;
`aggressive` enables `advanced`, then extraction of time-varying aliases.
Thus, `aggressive` triggers the full-fledged CIRE algorithm, while `advanced` uses only a relaxed version (based on time invariants). All runs used loop tiling with a block shape that was determined individually for each case using auto-tuning. The

auto-tuning phase, however, was not included in the measured experiment runtime. Likewise, the code generation phase is not included in the reported runtime.

7.2. Test case setup. In the following sections, we benchmark the performance of operators modeling the propagation of acoustic waves in two different models: isotropic and Tilted Transverse Isotropy (TTI, [44]), henceforth `isotropic` and `tti`, respectively. These operators were chosen for their relevance in seismic imaging techniques [44].

Acoustic `isotropic` modeling is the most commonly used technique for seismic inverse problems, due to the simplicity of its implementation, as well as the comparatively low computational cost in terms of FLOPs. The `tti` wave equation provides a more realistic simulation of wave propagation and accounts for local directional dependency of the wave speed, but comes with increased computational cost and mathematical complexity. For our numerical tests, we use the `tti` wave equation as defined in [44]. The full specification of the equation as well as the finite difference schemes and its implementation using Devito are provided in [24, 23]. Essentially, the `tti` wave equation consists of two coupled acoustic wave equations, in which the Laplacians are constructed from spatially rotated first derivative operators. As indicated by Figure 4, these spatially rotated Laplacians have a significantly larger number of stencil coefficients in comparison to its isotropic equivalent which comes with an increased operational intensity.

The `tti` and `isotropic` equations are discretized with second order in time and varying space orders of 4, 8, 12 and 16. For both test cases, we use zero initial conditions, Dirichlet boundary conditions and absorbing boundaries with a 10 point mask (Section 3.5). The waves are excited by injecting a time-dependent, but spatially-localized seismic source wavelet into the subsurface model, using Devito's sparse point interpolation and injection as described in Section 3.1. We carry out performance measurements for two velocity models of 512^3 and 768^3 grid points with a grid spacing of 20 m. Wave propagation is modeled for 1000 ms, resulting in 327 time steps for `isotropic`, and 415 time steps for `tti`. The time-stepping interval is chosen according to the Courant-Friedrichs-Lewy (CFL) condition [8], which guarantees stability of the explicit time-marching scheme and is determined by the highest velocity of the subsurface model and the grid spacing.

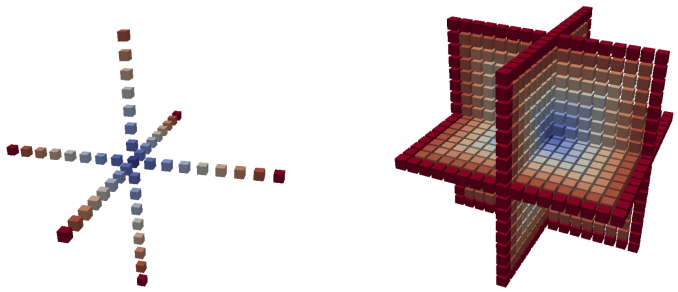


Fig. 4: Stencils of the acoustic Laplacian for the `isotropic` (left) and `tti` (right) wave equations and `so`=16. The anisotropic Laplacian corresponds to a spatially rotated version of the isotropic Laplacian. The color indicates the distance from the central coefficient.

7.3. Performance: acoustic wave in isotropic model. This section illustrates the performance of `isotropic` with the `core` and `yask` backends. To relieve the exposition, we show results for the DSE in `advanced` mode only; the `aggressive` has no impact on `isotropic`, due to the memory-bound nature of the code [23].

The performance of `core` on `skl8180`, illustrated in Figure 5a (`yask` uses slightly smaller grids than `core` due to a flaw in the API of *Devito v3.1*, which will be fixed in *Devito v3.2*), degrades as the space order (henceforth, `so`) increases. In particular, it drops from 59% of the attainable machine peak to 36% in the case of `so`=16. This is the result of multiple issues. As `so` increases, the number of streams of unaligned virtual addresses also increases, causing more pressure on the memory system. Intel® VTune™ revealed that the lack of split registers to efficiently handle split loads was a major source of performance degradation. Another major issue for `isotropic` on `core` concerns the quality of the generated SIMD code. The in-line vectorization performed by the auto-vectorizer produces a large number of pack/unpack instructions to move data between vector registers, which introduces substantial overhead. Intel® VTune™ also confirmed that, unsurprisingly, `isotropic` is a memory-bound kernel. Indeed, switching off the DSE basically did not impact the runtime, although it did increase the operational intensity of the four test cases.

The performance of `core` on `knl7250` is not as good as that on `skl8180`. Figure 5b shows an analogous trend to that on `skl8180`, with the attainable machine peak systematically dropping as `so` increases. The issue is that here the distance from the peak is even larger. This simply suggests that `core` is failing at exploiting the various levels of parallelism available on `knl7250`.

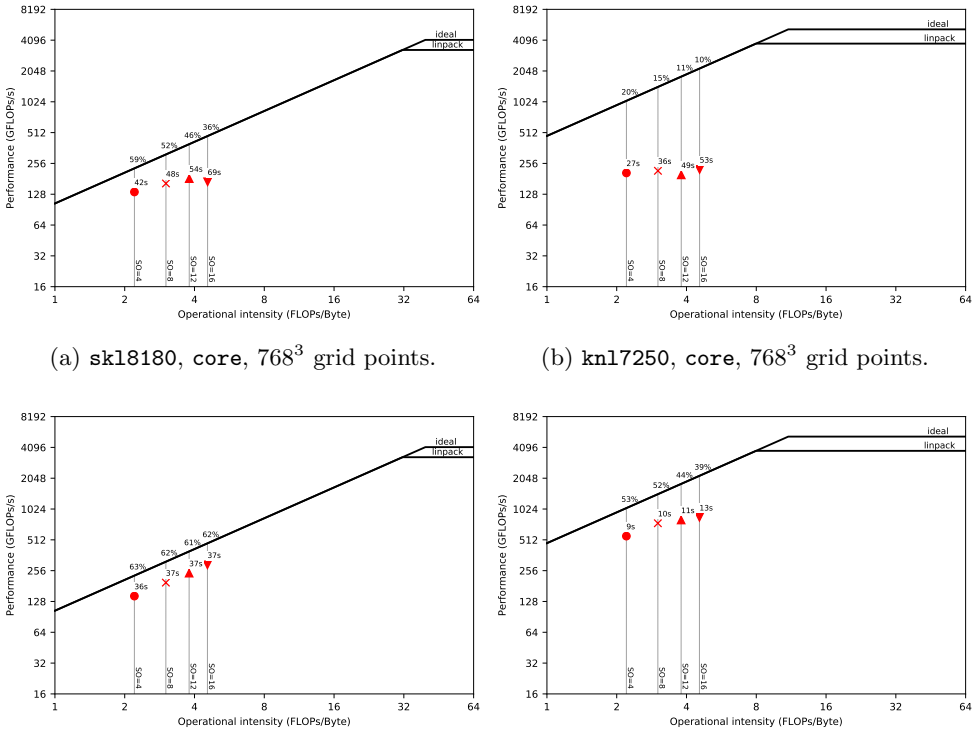
The `yask` backend overcomes all major limitations to which `core` is subjected. On both `skl8180` and `knl7250`, `yask` outperforms `core`, essentially since it does not suffer from the issues presented above. Vector folding minimizes unaligned accesses; software prefetching helps especially for larger values of `so`; hierarchical OpenMP parallelism is fundamental to leverage shared caches. The speed-up on `knl7250` is remarkable, since even in the best scenario for `core` (`so`=4), `yask` is roughly 3× faster, and more than 4× faster when `so`=12.

7.4. Performance: acoustic wave in tilted transverse isotropy model. This sections illustrates the performance of `tti` with the `core` backend. `tti` cannot be run on the `yask` backend in *Devito v3.1* as some fundamental features are still missing; this is part of our future work (more details in Section 8).

Unlike `isotropic`, `tti` significantly benefits from different levels of DSE optimizations, which play a key role in reducing the operation count as well as the register pressure. Figure 6 displays the performance of `tti` for the usual range of space orders on `skl8180` and `knl7250`, for two different cubic grids.

Generally, `tti` does not reach the same level of performance as `isotropic`. This is not surprising given the complexity of the PDEs (e.g., in terms of differential operators), which translates into code with much higher arithmetic intensity. In `tti`, the memory system is stressed by a considerably larger number of loads per loop iteration than in `isotropic`. On `skl8180`, we ran some profiling with Intel® VTune™. We determined that one of the major issues is the pressure on both L1 cache (lack of split registers, unavailability of "fill buffers" to handle requests to the other levels of the hierarchy) and DRAM (bandwidth and latency). Clearly, this is only a summary from some sample kernels – the actual situation varies depending on the DSE optimizations as well as the `so` employed.

It is remarkable that on both `skl8180` and `knl7250`, and on both grids, the



(a) `skl8180`, `core`, 768^3 grid points.

(b) `knl7250`, `core`, 768^3 grid points.

(c) `skl8180`, `yask`, 748^3 grid points.

(d) `knl7250`, `yask`, 748^3 grid points.

Fig. 5: Performance of `isotropic` on multiple Devito backends and architectures.

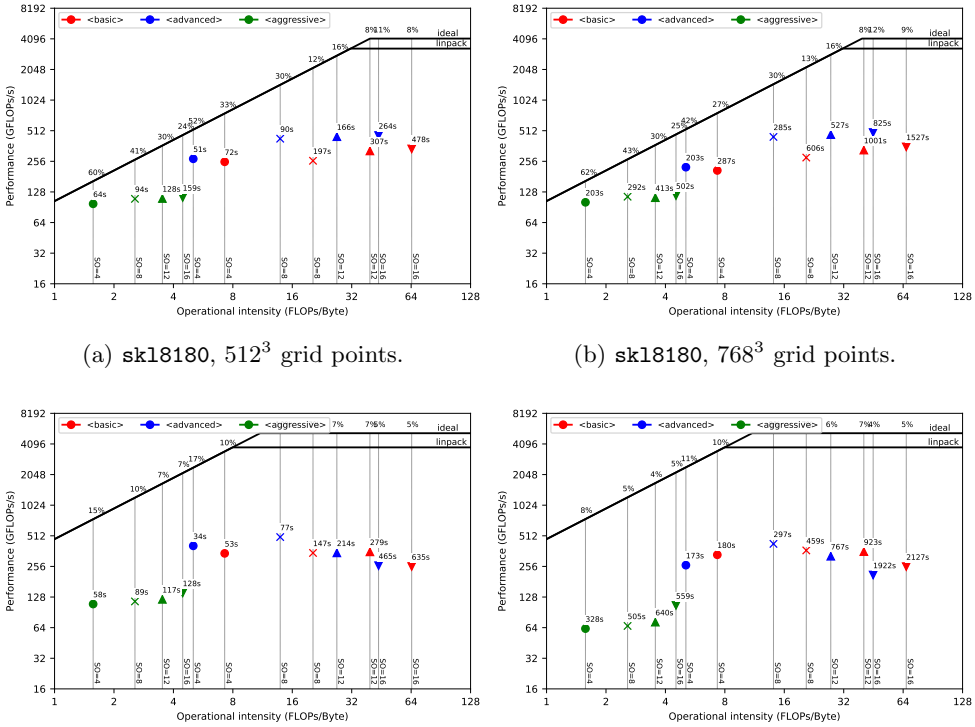
cutoff point beyond which `advanced` results in worse runtimes than `aggressive` is `so`=8. One issue with `aggressive` is that to avoid redundant computation, not only additional memory is required, but also more data communication may occur through caches, rather than through registers. In Figure 12, for example, we can easily deduce that `temp` is first stored, and then reloaded in the subsequent loop nest. This is an overhead that `advanced` does not pay, since temporaries are communicated through registers, for as much as possible. Beyond `so`=8, however, this overhead is overtaken by the reduction in operation count, which grows almost quadratically with `so`, as reported in Table 1.

Table 1: Operation counts for different DSE modes in `tti`

so	basic	advanced	aggressive
4	299	260	102
8	857	707	168
12	1703	1370	234
16	2837	2249	300

The performance on `knl7250` is overall disappointing. This is unfortunately caused by multiple factors – some of which already discussed in the previous sec-

tions. These results, and more in general, the need for performance portability across future (Intel® or non-Intel®) architectures, motivated the ongoing `yask` project. Here, the overarching issue is the inability to exploit the multiple levels of parallelism typical of architectures such as `knl7250`. Approximately 17% of the attainable peak is obtained when `so`=4 with `advanced` (best runtime out of the three DSE modes for the given space order). This occurs when using 512^3 points per grid, which allows the working set to completely fit in MCDRAM (our calculations estimated a size of roughly 7.5GB). With the larger grid size (Figure 6d), the working set increases up to 25.5GB, which exceeds the MCDRAM capacity. This partly accounts for the 5× slow down in runtime (from 34s to 173s) in spite of only a 3× increase in number of grid points computed per time iteration.



(a) `skl8180`, 512^3 grid points.

(b) `skl8180`, 768^3 grid points.

(c) `knl7250`, 512^3 grid points.

(d) `knl7250`, 768^3 grid points.

Fig. 6: Performance of `tti` on `core` for different architectures and grids.

8. Further work. While many simulation and inversion problems such as full-waveform inversion only require the solver to run on a single shared memory node, many other applications require support for distributed memory parallelism (typically via MPI) so that the solver can run across multiple compute nodes. The immediate plan is to leverage `yask`'s MPI support, and perhaps to include MPI support into `core` at a later stage. Another important feature is staggered grids, which are necessary for a wide range of FD discretization methods (e.g. modelling elastic wave propagation). Basic support for staggered grids is already included in *Devito v3.1*, but currently

only through a low-level API – the principle of graceful degradation in action. We plan to make the use of this feature more convenient.

As discussed in Section 7.4, the `yask` backend is not feature-complete yet; in particular, it cannot run the `tti` equations in the presence of array temporaries. As `tti` is among the most advanced models for wave propagation used in industry, extending Devito in this direction has high priority.

There also is a range of advanced performance optimization techniques that we want to implement, such as "time tiling" (i.e., loop blocking across the time dimension), on-the-fly data compression, and mixed-precision arithmetic exploiting application knowledge. Finally, there is an on-going effort towards adding an `ops` [31] backend, which will enable code generation for GPUs and also supports distributed memory parallelism via MPI.

9. Conclusions. Devito is a system to automate high-performance stencil computations. While Devito provides a *Python*-based syntax to easily express FD approximations of PDEs, it is not limited to finite differences. A Devito `Operator` can implement arbitrary loop nests, and can evaluate arbitrarily long sequences of heterogeneous expressions such as those arising in FD solvers, linear algebra, or interpolation. The compiler technology builds upon years of experience from other DSL-based systems such as FEniCS and Firedrake, and wherever possible Devito uses existing software components including *SymPy* and *NumPy*, and YASK. The experiments in this article show that Devito can generate production-level code with compelling performance on state-of-the-art architectures.

REFERENCES

[1] *Compilers: principles, techniques, and tools*, Pearson/Addison Wesley, Boston, MA, USA, second ed., 2007, http://www.loc.gov/catdir/toc/ecip0618/2006024333.html.

[2] M. S. Alnæs, A. Logg, K. B. Ølgaard, M. E. Rognes, and G. N. Wells, *Unified Form Language: a domain-specific language for weak formulations of partial differential equations*, ACM Transactions on Mathematical Software (TOMS), 40 (2014), p. 9.

[3] 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*, arXiv preprint arXiv:1702.04715, (2017).

[4] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, *A practical automatic polyhedral parallelizer and locality optimizer*, in Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '08, New York, NY, USA, 2008, ACM, pp. 101–113, https://doi.org/10.1145/1375581.1375595, http://doi.acm.org/10.1145/1375581.1375595.

[5] S. C. Brenner and L. R. Scott, *The Mathematical Theory of Finite Element Methods*, vol. 15, Springer New York, New York, NY, 2008, https://doi.org/10.1007/978-0-387-75934-0, http://dx.doi.org/10.1007/978-0-387-75934-0.

[6] A. F. Cárdenas and W. J. Karplus, *Pdel—a language for partial differential equations*, Communications of the ACM, 13 (1970), pp. 184–191.

[7] G. O. Cook Jr, *Alpal: A tool for the development of large-scale simulation codes*, tech. report, Lawrence Livermore National Lab., CA (USA), 1988.

[8] R. Courant, K. Friedrichs, and H. Lewy, *On the partial difference equations of mathematical physics*, International Business Machines (IBM) Journal of Research and Development, 11 (1967), pp. 215–234, https://doi.org/10.1147/rd.112.0215.

[9] K. Datta, S. Williams, V. Volkov, J. Carter, L. Oliker, J. Shalf, and K. Yelick, *Auto-tuning the 27-point stencil for multicore*, in In In Proc. iWAPT2009: The Fourth International Workshop on Automatic Performance Tuning, 2009.

[10] S. J. Deitz, B. L. Chamberlain, and L. Snyder, *Eliminating redundancies in sum-of-product array computations*, in Proceedings of the 15th International Conference on Supercomputing, ICS '01, New York, NY, USA, 2001, ACM, pp. 65–77, https://doi.org/10.1145/377792.377807, http://doi.acm.org/10.1145/377792.377807.

[11] Y. Ding and X. Shen, *Glore: Generalized loop redundancy elimination upon ler-notation*, Proc. ACM Program. Lang., 1 (2017), pp. 74:1–74:28, https://doi.org/10.1145/3133898, http://doi.acm.org/10.1145/3133898.

[12] S. Fomel, P. Sava, I. Vlad, Y. Liu, and V. Bashkardin, *Madagascar: open-source software project for multidimensional data analysis and reproducible computational experiments*, Journal of Open Research Software, 1 (2013), p. e8, https://doi.org/http://dx.doi.org/10.5334/jors.ag.

[13] T. O. Foundation, *OpenFOAM v5 User Guide*, https://cfd.direct/openfoam/user-guide/.

[14] 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), no. CSTN-187, Las Vegas, USA, 22-25 July 2013, WorldComp, p. SER3829.

[15] R. L. Higdon, *Numerical absorbing boundary conditions for the wave equation*, Mathematics of Computation, 49 (1987), pp. 65–90, http://www.jstor.org/stable/2008250.

[16] C. T. Jacobs, S. P. Jammy, and N. D. Sandham, *Opensbli: A framework for the automated derivation and parallel execution of finite difference solvers on a range of computer architectures*, CoRR, abs/1609.01277 (2016), http://arxiv.org/abs/1609.01277.

[17] J. Jeffers and J. Reinders, *High Performance Parallelism Pearls Volume Two: Multicore and Many-core Programming Approaches*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st ed., 2015.

[18] A. Klöckner, *Loo.py: transformation-based code generation for GPUs and CPUs*, in Proceedings of ARRAY '14: ACM SIGPLAN Workshop on Libraries, Languages, and Compilers for Array Programming, Edinburgh, Scotland., 2014, Association for Computing Machinery, https://doi.org/{10.1145/2627373.2627387}.

[19] A. Klckner, *Cgen - c/c++ source generation from an ast*. https://github.com/inducer/cgen, 2016.

[20] S. Kronawitter, S. Kuckuk, and C. Lengauer, *Redundancy elimination in the exastencils code generator*, in ICA3PP Workshops, 2016.

[21] C. Lengauer, S. Apel, M. Bolten, A. Grösslinger, F. Hannig, H. Köstler, U. Rüde, J. Teich, A. Grebhahn, S. Kronawitter, S. Kuckuk, H. Rittich, and C. Schmitt, *Exastencils: Advanced stencil-code engineering*, in Euro-Par 2014: Parallel Processing Workshops - Euro-Par 2014 International Workshops, Porto, Portugal, August 25-26, 2014, Revised Selected Papers, Part II, 2014, pp. 553–564, https://doi.org/10.1007/978-3-319-14313-2_47, https://doi.org/10.1007/978-3-319-14313-2_47.

[22] A. Logg, K.-A. Mardal, G. N. Wells, et al., *Automated Solution of Differential Equations by the Finite Element Method*, Springer, 2012, https://doi.org/10.1007/978-3-642-23099-8.

[23] M. Louboutin, M. Lange, F. J. Herrmann, N. Kukreja, and G. Gorman, *Performance prediction of finite-difference solvers for different computer architectures*, Computers & Geosciences, 105 (2017), pp. 148–157, https://doi.org/https://doi.org/10.1016/j.cageo.2017.04.014, https://www.slim.eos.ubc.ca/Publications/Public/Journals/ComputersAndGeosciences/2016/louboutin2016ppf/louboutin2016ppf.pdf. (Computers & Geosciences).

[24] M. Louboutin, M. Lange, F. Luporini, N. Kukreja, P. A. Witte, P. Velesko, G. Gorman, and F. J. Herrmann, *Devito: A portable and flexible mathematical api for geophysical applications*. 2018.

[25] G. R. Markall, F. Rathgeber, L. Mitchell, N. Loriant, C. Bertolli, D. A. Ham, and P. H. J. Kelly, *Performance-portable finite element assembly using pyop2 and fenics*, in 28th International Supercomputing Conference, ISC, Proceedings, J. M. Kunkel, T. Ludwig, and H. W. Meuer, eds., vol. 7905 of Lecture Notes in Computer Science, Springer, 2013, pp. 279–289, https://doi.org/10.1007/978-3-642-38750-0_21, http://dx.doi.org/10.1007/978-3-642-38750-0_21.

[26] Mathias Louboutin, Fabio Luporini, *Boundary conditions in Devito*, in preparation (2018).

[27] A. Meurer, C. P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, T. Rathnayake, S. Vig, B. E. Granger, R. P. Muller, F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, M. J. Curry, A. R. Terrel, v. Roučka, A. Saboo, I. Fernando, S. Kulal, R. Cimrman, and A. Scopatz, *Sympy: symbolic computing in python*, PeerJ Computer Science, 3 (2017), p. e103, https://doi.org/10.7717/peerj-cs.103, https://doi.org/10.7717/peerj-cs.103.

[28] S. J. Pennycook, J. Sewall, and V. Lee, *A metric for performance portability*, arXiv preprint arXiv:1611.07409, (2016).

[29] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, *Halide: A language and compiler for optimizing parallelism, locality, and recomputation*

in image processing pipelines, in Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, New York, NY, USA, 2013, ACM, pp. 519–530, https://doi.org/10.1145/2491956.2462176, http://doi.acm.org/10.1145/2491956.2462176.

[30] F. Rathgeber, D. A. Ham, L. Mitchell, M. Lange, F. Luporini, A. T. T. Mcrae, G.-T. Bercea, G. R. Markall, and P. H. J. Kelly, *Firedrake: Automating the finite element method by composing abstractions*, ACM Trans. Math. Softw., 43 (2016), pp. 24:1–24:27, https://doi.org/10.1145/2998441, http://doi.acm.org/10.1145/2998441.

[31] I. Z. Reguly, G. R. Mudalige, M. B. Giles, D. Curran, and S. McIntosh-Smith, *The ops domain specific abstraction for multi-block structured grid computations*, in Proceedings of the Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing, WOLFHPC '14, Piscataway, NJ, USA, 2014, IEEE Press, pp. 58–67, https://doi.org/10.1109/WOLFHPC.2014.7, http://dx.doi.org/10.1109/WOLFHPC.2014.7.

[32] W. W. Symes, D. Sun, and M. Enriquez, *From modelling to inversion: designing a well-adapted simulator*, Geophysical Prospecting, 59 (2011), pp. 814–833, https://doi.org/10.1111/j.1365-2478.2011.00977.x, http://dx.doi.org/10.1111/j.1365-2478.2011.00977.x.

[33] J. Tobin, A. Breuer, A. Heinecke, C. Yount, and Y. Cui, *Accelerating seismic simulations using the intel xeon phi knights landing processor*, in Proceedings of ISC High Performance 2017 (ISC17), to appear 2017.

[34] Y. Umetani, *Deqsol a numerical simulation language for vector/parallel processors*, Proc. IFIP TC2/WG22, 1985, 5 (1985), pp. 147–164.

[35] R. Van Engelen, L. Wolters, and G. 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.

[36] F. Witherden, A. Farrington, and P. Vincent, *Pyfr: An open source framework for solving advectiondiffusion type problems on streaming architectures using the flux reconstruction approach*, Computer Physics Communications, 185 (2014), pp. 3028 – 3040, https://doi.org/https://doi.org/10.1016/j.cpc.2014.07.011, http://www.sciencedirect.com/science/article/pii/S0010465514002549.

[37] C. Yount, *Vector folding: Improving stencil performance via multi-dimensional simd-vector representation*, in Proceedings of the IEEE 17th International Conference on High Performance Computing and Communications (HPCC), Aug 2015, pp. 865–870, https://doi.org/10.1109/HPCC-CSS-ICESS.2015.27.

[38] C. Yount and A. Duran, *Effective use of large high-bandwidth memory caches in HPC stencil computation via temporal wave-front tiling*, in Proceedings of the 7th International Workshop in Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems held as part of ACM/IEEE Supercomputing 2016 (SC16), PMBS'16, Nov 2016.

[39] C. Yount, A. Duran, and J. Tobin, *Multi-level spatial and temporal tiling for efficient hpc stencil computation on many-core processors with large shared caches*, Future Generation Computer Systems, (2017), https://doi.org/https://doi.org/10.1016/j.future.2017.10.041, http://www.sciencedirect.com/science/article/pii/S0167739X17304648.

[40] C. Yount, J. Tobin, A. Breuer, and A. Duran, *Yask–yet another stencil kernel: a framework for hpc stencil code-generation and tuning*, in Proceedings of the 6th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing held as part of ACM/IEEE Supercomputing 2016 (SC16), WOLFHPC'16, Nov 2016, https://doi.org/10.1109/WOLFHPC.2016.08.

[41] Zenodo/devito, *Devito v3.1*, October 2017, https://doi.org/10.5281/zenodo.836688.

[42] Zenodo/devito-performance, *Devito Experimentation Framework*, July 2018, https://doi.org/TODO.

[43] Y. Zhang and F. Mueller, *Auto-generation and auto-tuning of 3d stencil codes on gpu clusters*, in Proceedings of the Tenth International Symposium on Code Generation and Optimization, CGO '12, New York, NY, USA, 2012, ACM, pp. 155–164, https://doi.org/10.1145/2259016.2259037, http://doi.acm.org/10.1145/2259016.2259037.

[44] Y. Zhang, H. Zhang, and G. Zhang, *A stable tti reverse time migration and its implementation*, GEOPHYSICS, 76 (2011), pp. WA3–WA11, https://doi.org/10.1190/1.3554411, https://doi.org/10.1190/1.3554411, https://arxiv.org/abs/https://doi.org/10.1190/1.3554411.