Architecture and Performance of Devito, a System for Automated Stencil Computation

FABIO LUPORINI, Imperial College London
MATHIAS LOUBOUTIN, Georgia Institute of Technology
MICHAEL LANGE, European Centre for Medium-Range Weather Forecasts
NAVJOT KUKREJA, Imperial College London
PHILIPP WITTE, Georgia Institute of Technology
JAN HÜCKELHEIM, Imperial College London
CHARLES YOUNT, Intel Corporation
PAUL H. J. KELLY, Imperial College London
FELIX J. HERRMANN, Georgia Institute of Technology
GERARD J. GORMAN, Imperial College London

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 backend 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.

CCS Concepts: • **Mathematics of computing → Mathematical software performance**; • **Software and its engineering → Compilers**; **Domain specific languages**;

Additional Key Words and Phrases: Finite-difference method, stencil, domain-specific language, symbolic processing, structured grid, compiler, performance optimization

This work was supported by the Engineering and Physical Sciences Research Council through grants EP/I00677X/1, EP/L000407/1, and EP/I012036/1, by the Imperial College London Department of Computing, by the Imperial College London Intel Parallel Computing Centre (IPCC), by the Georgia Research Alliance, by the Georgia Institute of Technology, 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. M. Louboutin, P. Witte, and F. J. Herrmann acknowledge the University of British Columbia, where part of this research was carried out.
Authors' addresses: F. Luporini, N. Kukreja, J. Hückelheim, P. H. J. Kelly, and G. J. Gorman, Imperial College London, South Kensington, London SW7 2BU, United Kingdom; emails: {f.luporini12, n.kukreja, j.hueckelheim, p.kelly, g.gorman}@imperial.ac.uk; M. Louboutin, P. Witte, and F. J. Herrmann, Georgia Institute of Technology; emails: {mlouboutin3, pwitte3, felix.herrmann}@gatech.edu; M. Lange, European Centre for Medium-Range Weather Forecasts; email: michael.lange@ecmwf.int; C. Yount, Intel Corporation; email: chuck.yount@intel.com.

0098-3500/2020/04-ART6
https://doi.org/10.1145/3374916

ACM Reference format:
Fabio Luporini, Mathias Louboutin, Michael Lange, Navjot Kukreja, Philipp Witte, Jan Hückelheim, Charles Yount, Paul H. J. Kelly, Felix J. Herrmann, and Gerard J. Gorman. 2020. Architecture and Performance of Devito, a System for Automated Stencil Computation. *ACM Trans. Math. Softw.* 46, 1, Article 6 (April 2020), 28 pages.
https://doi.org/10.1145/3374916

1 INTRODUCTION

Developing software for high-performance computing requires 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, whereas a stack of compilers and libraries is responsible for automatically creating the optimized low-level implementation in a general-purpose programming language such as C++. Although the focus of this article is 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 methods are commonly used for solving wave equations as part of complex workflows (e.g., data inversion using adjoint-state methods and backpropagation). Devito is equally useful as a framework for other stencil computations in general, such as 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. Although the examples presented in this article 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 modernization.

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 optimizations are performed programmatically

facilitates performance portability across different computer architectures [29]. 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 [42]. 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 article finishes with directions for future work and conclusions in Sections 8 and 9, respectively.

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 several tools specialized in numerical methods for PDEs, some dating back to the 1970s [6, 7, 36, 37].

2.1 DSL-Based Frameworks for PDEs

One noteworthy contemporary framework centered on DSLs is FEniCS [23], which allows the specification of weak variational forms, via UFL [2], and FE methods, through a high-level syntax. Firedrake [31] implements the same languages as FEniCS, although it differs from it in several features and architectural choices. Devito is heavily influenced by these two successful projects, particularly by their philosophy and design. Since solving a PDE is often a small step of a larger workflow, the choice of *Python* to implement this 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 FE formulation) are required. Likewise, Devito allows injecting arbitrary expressions into the FD specification; this feature has been used in real-life cases, such as for interpolation in seismic imaging operators. However, 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 [14], and by PyFR, which targets flux reconstruction methods [38].

2.2 High-Level Approaches to FDs

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 [3, 15, 17, 22]. Devito distinguishes itself from previous work in several 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, and adoption of a native *Python* interface that naturally allows composition into complex workflows such as optimization and machine-learning frameworks.

At a lower level of abstraction there are several 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 [30, 32, 42, 44], whereas others are compilers or user-driven code generation systems, often based upon a polyhedral model (e.g., [4, 19]). 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 [42], 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 [33] and Madagascar [13], 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: FLOP reduction, data locality, and parallelism. Typical FLOP reduction transformations are common sub-expressions elimination (CSE), factorization, and code motion. A thorough review is provided by Ding and Shen [11]. 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, 21]. 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 [18] 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 AN FD METHOD WITH DEVITO

The Devito DSL allows concise expression of FD and general stencil operations using a mathematical notation. It uses *SymPy* [28] 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

$$\begin{aligned} m(x,y,z)\frac{d^2u(x,y,z,t)}{dt^2} - \nabla^2 u(x,y,z,t) &= q_s, \\ u(x,y,z,0) &= 0, \\ \frac{du(x,y,z,t)}{dt}|_{t=0} &= 0, \end{aligned} \tag{1}$$

where the variables of this expression are defined as follows:

- $m(x,y,z) = \frac{1}{c(x,y,z)^2}$ is the parameterization of the sub-surface 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; and
- q_s is the source term, which is a point source in this case.

The first step toward 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.

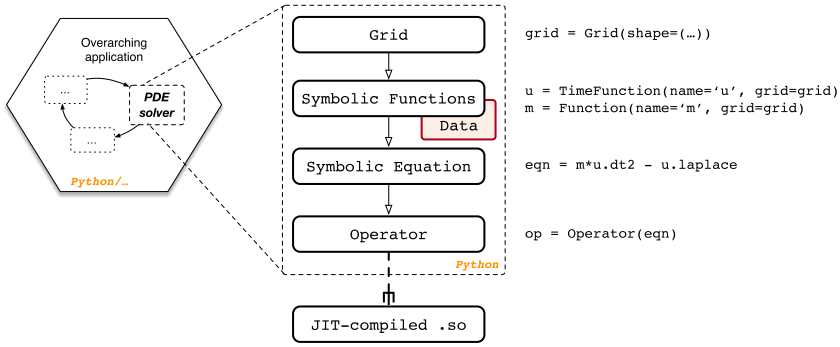


Fig. 1. The typical usage of Devito within a larger application.

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 (1). The initialization of the fields to solve the wave equation over a 1D grid is displayed in Listing 1.

```
>>> 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
```

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

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).

```
>>> 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
```

Listing 2. Example of spatial and temporal FD stencil creation.

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).

```
>>> 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
```

Listing 3. Expressing the wave equation.

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 timestep: $u(t + dt) = f(u(t), u(t - dt)))$. Devito allows to rearrange the PDE expression automatically using the `solve` function, as shown in Listing 4.

```
>>> 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))
```

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

Note that the `stencil` expression in Listing 4 does not yet contain the point source q. This could be included as a regular `Function` that has zeros all over the grid except for a few points, but it would obviously be wasteful. Instead, `SparseFunctions` allow to perform operations, such as injecting a source or sampling the wavefield, at a subset of points determined by `coordinates`. In general, receivers (where the solution is to sampled) are not co-located with grid points. Therefore, an interpolation operator is needed (e.g., trilinear interpolation for 3D). To ensure a consistent discrete adjoint, source terms are implemented as the adjoint of the interpolation operator used–that is, the gather operation for interpolation becomes a scatter operation for source injection. Equation (2) gives the expressions for linear interpolation in 1D assuming the origin is zero for readability:

$$\begin{aligned}
\text{Find the two closest indices:} \quad & x_1 = \left\lfloor \frac{q_{coords}[i]}{h_x} \right\rfloor,\ x_2 = x_1 + 1 \\
\text{Interpolation coefficients:} \quad & c_1 = 1 - \frac{q_{coords}[i] - x_1}{x_2 - x_1},\ c_2 = 1 - \frac{x_2 - q_{coords}[i]}{x_2 - x_1} \\
\text{Interpolate:} \quad & q[i] = c_1 * u[t, x_1] + c_2 * u[t, x_2] \\
\text{Inject:} \quad & u[t, x_1] = q[i] * c_1,\ u[t, x_2] = q[i] * c_2.
\end{aligned} \tag{2}$$

```
>>> 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])]
```

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

To inject a point source defined at the physical location `q_coords` into the `stencil` expression, we use the `inject` function of the `SparseTimeFunction` object that represents our seismic source (Listing 5).[1]

The `inject` function takes the field being updated as an input argument (in this case `u.forward`), whereas `expr=dt**2 * q / m` is the expression being injected. The result of the `inject` function is a list of symbolic expressions that correspond to the different steps of Equation (2). 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 [16]), just like any of the symbolic expressions defined in preceding listings. For reasons of space, this aspect is not elaborated further; the interested reader may refer to Louboutin and Luporini [27].

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 (i.e., $u[t\%3, \ldots]$). 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 four iterations, up to a maximum of $save = 100$ time slices.

```
>>> 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)
```

Listing 6. Implementation of time sub-sampling.

In general, all equations that access `Functions` (or `TimeFunctions`) employing one or more `ConditionalDimensions` 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:

[1] More complicated interpolation schemes can be defined by pre-computing 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.

***Domain*:** This 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, such as absorbing boundary layers.

***Halo*:** The grid points surrounding the domain region—for instance, "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 that of Aho et al. [1]).

4.1 Equation Lowering

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

- As explained in Section 3, the input equations typically involve one or more indexed `Functions`. 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 `Functions` 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:

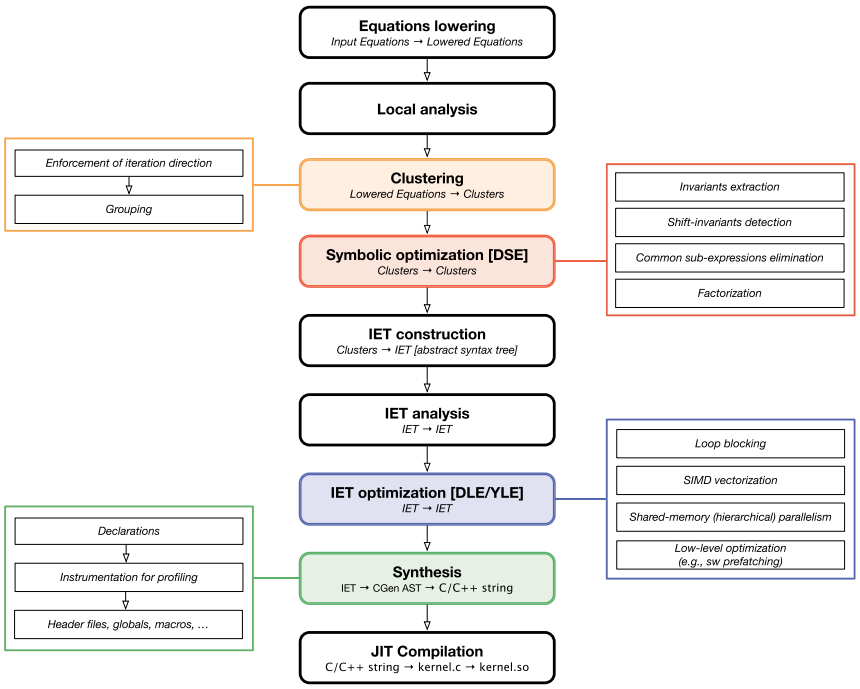


Fig. 2. Compiler passes to lower symbolic equations into shared objects through an `Operator`.

- input and output `Functions`;
- `Dimensions`, which are topologically ordered based on how they appear in the various array index functions; and
- two notable `Spaces`: 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), whereas o_m and o_M are known integers. For instance, $[x[0,0], y[-1,1]]$ describes a rectangular 2D 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 `Dimensions`. A `DSpace` may have `Dimensions` that do not appear in an `ISpace`, particularly those that are accessed only via integer indices. Likewise, an `ISpace` may have `Dimensions` 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 $= [u]$, and `Dimensions` $= [t, x]$. 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 because there is a flow dependency 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 `ConditionalDimensions`), and (iii) no dimension-carried "true" anti-dependencies 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 `ISpaces` are different. However, the equations in *injection* can be grouped together in the same `Cluster` because (i) they have same `ISpace` $[t[0,0]^*, p_q[0,0]^*]$, (ii) they have the same control flow, and (iii) there are no true anti-dependencies among them (note that the second equation in *injection* does write to $u[t+1,\ldots]$, but as explained later this is in fact a reduction, which is a "false" anti-dependency).

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 dependency 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 `ISpaces` 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` $[d[0,0]^+,\ldots]$ and $[d[0,0]^-,\ldots]$), 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
1  mapper ← DETECT_FLOW_DIRECTIONS(E);
2  for e in E do
3      for dim, directions in mapper do
4          if len(directions) == 1 then
               // No ambiguity
5              forced[dim] ← directions.pop();
6          else if len(directions) == 2 then
               // No ambiguity as long as one of the two items is /Any/
7              try
8                  directions.remove(Any);
9                  forced[dim] ← directions.pop();
10             except
11                 forced[dim] ← e.directions[dim];
12         else
13             forced[dim] ← e.directions[dim];
14         end if
15     end for
16     E'.append(e._rebuild(directions=forced))
17 end for
18 return E'
```

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

- 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 dependency or not. The data-dependence analysis performed is conservative—that is, a dependency 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_DEPENDENCIES.
- If an anti-dependency 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 particularly increments, are treated specially. They represent a special form of anti-dependency, 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—that is, 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 `Clusters` if one or more `ConditionalDimensions` are encountered. These are tracked in a special `Cluster` field, *guards*, as also required by later passes (see Section 4.5).

ALGORITHM 2: Clustering: grouping expressions into `Clusters` (pseudocode).

```
   Input: A sequence of equations E.
   Output: A sequence of clusters C.
1  C ← ClusterGroup();
2  for e in E do
3      grouped ← false;
4      for c in reversed(C) do
5          anti, flow ← GET_DEPENDENCIES(c, e);
6          if e.ispace == c.ispace and anti.carried is empty then
7              c.add(e);
8              grouped ← true;
9              break;
10         else if anti.carried is not empty then
11             c.atomics.update(anti.carried.cause);
12             break;
13         else if flow.cause.intersection(c.atomics) then
               // cannot search across earlier clusters
14             break;
15     end for
16     if not grouped then
17         C.append(Cluster(e));
18     end if
19 end for
20 C ← CONTROL_FLOW(C);
21 return C
```

4.4 Symbolic Optimization

The DSE is a macro-pass reducing the *arithmetic strength* of `Clusters` (e.g., their operation count). It consists of a series of passes, ranging from standard CSE to more advanced rewrite procedures, applied individually to each `Cluster`. The DSE output is a new ordered sequence of `Clusters`: there may be more or fewer `Clusters` than in the input, and both the overall number of equations and the sequence of arithmetic operations might differ. The DSE passes are discussed in Section 5.1. We remark that the DSE only operates on `Clusters` (i.e., on collections of equations); there is no concept of "loop" at this stage yet. However, by altering `Clusters`, 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 `Iterations` and `Expressions`—two special node types—are the main actors. Equations are wrapped within `Expressions`, whereas `Iterations` represent loops. Loop nests embedding such `Expressions` are constructed by suitably nesting `Iterations`. Each `Cluster` is eventually placed in its own loop (`Iteration`) nest, although some (outer) loops may be shared by multiple `Clusters`.

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 C.
   Output: An Iteration/Expression Tree.
1  schedule ← list();
2  for c in C do
3  |  root ← None;
4  |  index ← 0;
5  |  for i0, i1 in zip(c.ispace, schedule) do
6  |  |  if i0 ≠ i1 or i0.dimension in c.atomics then
7  |  |  |  break;
8  |  |  end if
9  |  |  root ← schedule[i1];
10 |  |  index ← index + 1;
11 |  |  if i0.dim in c.guards then
12 |  |  |  break;
13 |  |  end if
14 |  end for
15 |  ⟨build as many Iterations as Dimensions in c.ispace[index:] and nest them inside root⟩;
16 |  ⟨update schedule⟩;
17 |  ⟨...⟩
18 end for
```

Consider again our running acoustic wave equation example. There are three `Clusters` 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 `Clusters` is turned into an IET. Initially, the *schedule* list is empty, so when C_1 is handled, two nested `Iterations` are created (line 15) for the `Dimensions` t and x, respectively. Subsequently, C_2's `ISpace` and the current *schedule* are compared (line 5). It turns out that t appears among C_2's guards, and 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

```
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)], ...)>
```

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

at the second iteration due to line 6, since $p_q \neq x$. Again, the t `Iteration` is reused, whereas a new `Iteration` is constructed for the `Dimension` p_q. Eventually, the constructed IET is as in Listing 7.

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, although 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 dependencies D, expressed as distance vectors $D = (d_0, \ldots, d_{n-1})$, either $(d_1, \ldots, d_{i-1}) > 0$ or $(d_1, \ldots, d_i) = 0$ [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. Several loop optimizations are introduced, including loop blocking, minimization of remainder loops, SIMD vectorization, shared-memory (hierarchical) parallelism via OpenMP, and 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 [20], which is ultimately translated into a string and written to a file. Such files are stored in a software cache of Devito-generated `Operators`, 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 Markall et al. [26]).

One of the main objectives of the backend infrastructure is promoting software composability. As explained in Section 2, a significant number of interesting tools exist 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 [42]. 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, whereas 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, whereas in Section 7.1, we explain how these are composed into modes.

5.1 The 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 through 11, which derive from the running example used throughout the article. A detailed description follows:

- ***Common sub-expressions elimination*:** 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 CSE 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.

```
>>> 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]
```

Listing 8. An example of CSE.

- ***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), whereas the increase in symbolic processing time could

otherwise be significant. The implementation exploits the *SymPy* `collect` routine. However, although `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.

```
>>> 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]
```

Listing 9. An example of FD weights factorization.

- ***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.

```
>>> 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]
```

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

- ***Detection of shif invariants*:** In essence, a shift invariant is a sub-expression that is redundantly computed at multiple iteration points. Because of its key role in the shift-invariants elimination (SIE) algorithm, the explanation of how shift invariants are detected is postponed until Section 6.

```
>>> 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]
```

Listing 11. An example of shift-invariant detection. The shift-invariant $9.0 * temp0 * u[t, x]$ is assigned to the vector temporary $temp[x]$ so that it can be used in place of the two sub-expressions $9.0 * temp0 * u[t, x + 1]$ and $9.0 * temp0 * u[t, x + 3]$.

5.2 The Devito Loop Engine

The DLE transforms the IET via classic loop optimizations for parallelism and data locality [18]. These are summarized as follows:

- ***SIMD vectorization*:** Implemented by enforcing compiler auto-vectorization via special `pragmas` 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 `Iterations`. Blocking over dimensions characterized by flow- or anti-dependencies, such as the time dimension in typical explicit FD schemes, is instead work in progress (this would require a preliminary pass known as loop skewing; see Section 8 for more details). However, a feature of the present implementation is the capability of blocking across particular

sequences of loop nests. This is exploited by the SIE 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 `Iterations` with suitable OpenMP `pragmas`. The OpenMP `static` scheduling is used. Normally, only the outermost fully parallel `Iteration` is annotated with the parallel `pragma`. However, heuristically nested fully parallel `Iterations` 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).

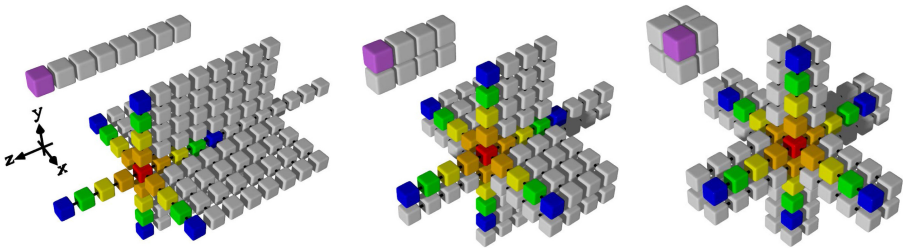
Summarizing, the DLE applies a sequence of typical stencil optimizations, aiming to reach a minimum level of performance across different architectures. As we shall see, the effectiveness of this approach, based on simple transformations, deteriorates on architectures strongly conceived for hierarchical 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 The YASK Loop Engine

YASK—Yet Another Stencil Kit[2]—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 [42] and its application to the Intel Xeon Phi x100 family (code named Knights Corner) [39] and Intel Xeon Phi x200 family (code named Knights Landing) [12, 35, 40] many-core CPUs. Unlike Devito, it does not expose a symbolic language to the programmer or create stencils from FD 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, whereas 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:

- ***Vector folding*:** In traditional SIMD vectorization, such as that provided by an auto-vectorizing compiler, the vector elements are arranged sequentially along the unit-stride dimension of the grid, which is also the dimension iterated over in the innermost loop of the stencil kernel. 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 3(a) shows a traditional 1D "in-line" layout, and Figure 3(b) and (c) 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 [39]. 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 [35, 39].
- ***Software prefetching*:** Many high-order or staggered-grid stencils require multiple streams of data to be read from memory, which can overwhelm the hardware prefetchers. YASK can 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

[2]Formerly, Yet Another Stencil Kernel.



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 eight elements [39]. The smaller diagram in the upper left of each sub-figure illustrates a single SIMD layout, which is also the configuration of the output elements from a single SIMD computation. The larger diagram shows the SIMD input values needed for a typical 25-point stencil (e.g., from an eighth-order FD approximation of an isotropic acoustic wave). The colored elements highlight the first element in the output layout (purple element) and the corresponding elements in the inputs (red through blue elements, where the different colors indicate the distance from the center). Note that the $1 \times 1 \times 8$ 1D fold corresponds to the traditional in-line vectorization.

this technique, sometimes called *cache blocking*, 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 tiling and MPI support that are not exploited by Devito at the time of writing. The interested reader may refer to Yount and Duran [40] and Yount et al. [41].

To leverage both the symbolic processing of Devito and the low-level optimizations of YASK, 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. In *Devito v3.1*, roughly 70% of the Devito API is supported by the `yask` backend.[3]

6 THE SIE ALGORITHM

Shift invariants, 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 shift invariants is a non-issue as long as the operator is memory bound, whereas it becomes relevant in kernels with a high arithmetic intensity. In Devito, the SIE algorithm attempts to remove shift invariants with the goal of reducing the operation count. As shown in Section 7, the SIE 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 shift invariants (DSE), and loop blocking (DLE/YLE).

6.1 Extraction of Candidate Expressions

The criteria for extraction of candidate sub-expressions are as follows:

[3] At the time of writing, reaching feature completeness is one the major ongoing development efforts.

- 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 shift-invariants, since they typically result from taking spatial and time derivatives on `TimeFunctions`. In particular, cross derivatives are a major cause of shift invariants.

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

6.2 Detection of Shif Invariants

To define the concept of shift-invariant expressions, we first need to formalize the notion of *shifted 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 shifted 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 shifted with respect to the operand $2 * u[x + 1, y + 2, z + 3]$ since they have same coefficient (2), label (u), and dimensions ($[x, y, z]$), whereas 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 shifted with respect to e_1 if the following conditions hold:

- the operands in e_0 (e_1) are shifted with respect to 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* shifted with respect to e, due to a different dimension vector.
- **u[x] + w[x]** is *not* shifted with respect to e, due to a different label.
- **u[x+2] + v[x]** is *not* shifted with respect to e, since it cannot be expressed as a translation of e.
- **u[x+2] + v[x+2]** is shifted with respect to e, as it can be expressed through the translation $T(x) = x + 2$.

The relation "e_0 is shifted with respect to e_1" is an equivalence relation, as it is at the same time reflexive, symmetric, and transitive. Thanks to these properties, the turnaround times for detecting shift invariants 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 shift invariants detection. In the worst case scenario, all pairs of candidate expressions are compared by applying the shift-invariant definition given previously. 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; and (iii) IS_TRANSLATED(d_1, d_2) evaluates to true if the displacement vectors in d_2 are pairwise shifted with respect to the vectors in d_1 by the same factor. Together, (ii) and (iii) are used to establish whether two expressions are shifted (line 8). From an implementation point of view, these functions exploit key *SymPy* expression properties (e.g., immutability, deterministic ordering of operands) and operators (e.g., for structural equality testing), so they eventually result rather simply.

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

ALGORITHM 4: Detection of shift invariants (pseudocode).

```
   Input: A sequence of expressions E.
   Output: A sequence of shift-invariant 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 = ShiftInvariant(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; in the following, we only mention the underlying ideas:

- The pivot of G_i is *constructed*, rather than selected out of $e_0, \ldots, e_{k-1}$, 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 pivots are chosen based on a global optimization strategy, which takes into account all of the m sets of shift invariants. 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 `Clusters` as well. In the former case, the optimization is applied.

6.3 Loop Blocking for Working-Set Minimization

In essence, the SIE algorithm trades operation for memory—the (array) temporaries to store the shift invariants. From a runtime 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; and (iii) SIMD vectorization is affected, since inner loop iterations are practically serialized. 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 Kronawitter et al. [21].
(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, whereas 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 SIE 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).

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

Listing 12. The loop nest produced by the SIE 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 clarity, unnecessary information is omitted.

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 analyze the performance of generated code using enriched roofline plots. Since the DSE transformations may alter the 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 `Functions` 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`, respectively. 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* [43]. The experimentation framework with instructions for reproducibility is available from the Devito Team [34]. All FLOPs 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, we have the following:

`basic` enables: CSE.
`advanced` enables: factorization; extraction of time-invariant shift invariants; detection of shift invariants; all `basic` passes.
`aggressive` enables: extraction of time-varying shift invariants; all `advanced` passes.

Thus, `aggressive` triggers the full-fledged SIE algorithm, whereas `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 [45]), henceforth `isotropic` and `tti`, respectively. These operators were chosen for their relevance in seismic imaging techniques [45].

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 by Zhang et al. [45]. The full specification of the equation and the FD schemes and its implementation using Devito are provided in Louboutin et al. [24, 25]. 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

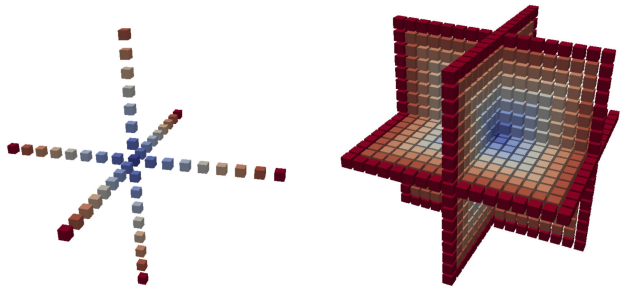


Fig. 4. Stencils of the acoustic Laplacian for the `isotropic` (left) and `tti` (right) wave equations and space-order of 16. The anisotropic Laplacian corresponds to a spatially rotated version of the isotropic Laplacian. The color indicates the distance from the central coefficient.

rotated Laplacians have a significantly larger number of stencil coefficients in comparison to its isotropic equivalent that 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 BCs, 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 sub-surface 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 1,000 ms, resulting in 327 timesteps for `isotropic` and 415 timesteps 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 sub-surface model and the grid spacing.

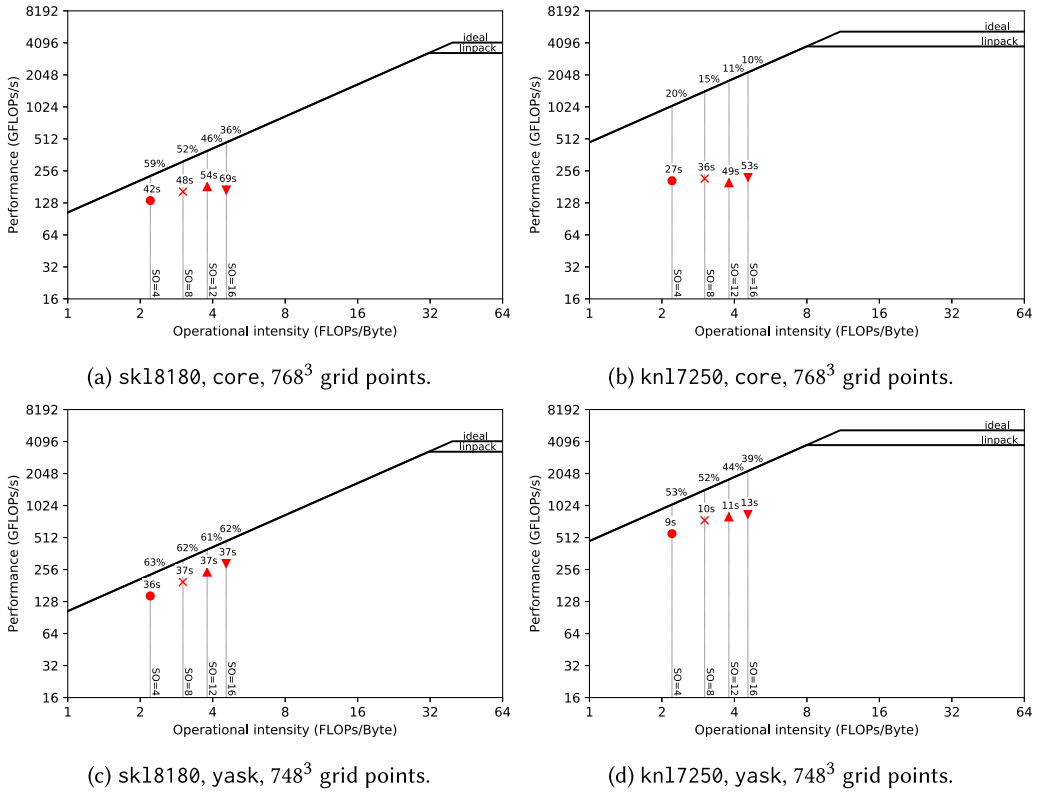
7.3 Performance: Acoustic Wave in Isotropic Model

This section illustrates the performance of `isotropic` with the `core` and `yask` backends. To simplify 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 [24].

The performance of `core` on `skl8180`, illustrated in Figure 5(a) (`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 5(b) 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



(a) sk18180, core, 768^3 grid points.

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

(c) sk18180, yask, 748^3 grid points.

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

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

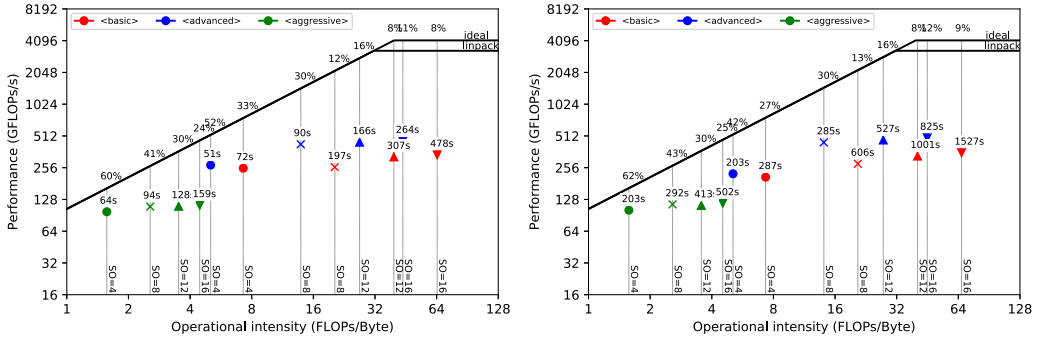
earlier. Vector folding reduces memory-read streams, software prefetching helps especially for larger values of so, and hierarchical OpenMP parallelism is fundamental to leverage shared caches. The speedup on knl7250 is remarkable, as even in the best scenario for core (so =4), yask is roughly 3× faster, and it is more than 4× faster when so = 12.

7.4 Performance: Acoustic Wave in the TTI Model

This sections illustrates the performance of tti with the core backend. tti cannot be run on the yask backend in *Devito v3.1* because some fundamental features are still missing; this is part of our future work (more details in Section 8).

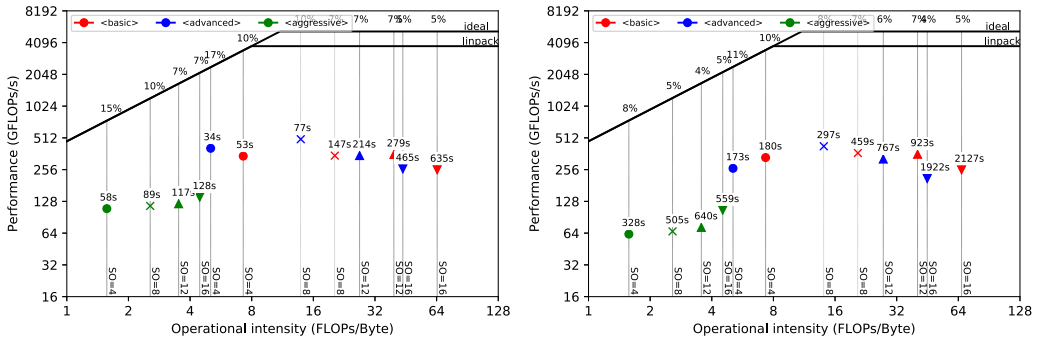
Unlike isotropic, tti significantly benefits from different levels of DSE optimizations, which play a key role in reducing the operation count as well as the register pressure. Figure 6 displays the performance of tti for the usual range of space orders on sk18180 and knl7250 for two different cubic 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 sk18180, we ran performance-profiling analyses using Intel VTune. We determined that the major issues are pressure on both L1 cache (lack of split registers, insufficient "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 and the so employed.



(a) sk18180, 512^3 grid points.

(b) sk18180, 768^3 grid points.



(c) kn17250, 512^3 grid points.

(d) kn17250, 768^3 grid points.

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

Table 1. Operation Counts for Different DSE Modes in `tti`

so	basic	advanced	aggressive
4	299	260	102
8	857	707	168
12	1,703	1,370	234
16	2,837	2,249	300

It is notable that on both `sk18180` and `kn17250`, and on both grids, the 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 is additional memory required but also more data communication may occur through caches rather than through registers. As shown later 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.

The performance on `kn17250` is overall disappointing. This is unfortunately caused by multiple factors—some of which already were discussed in the previous sections. 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 7.5 GB). With the larger grid size (Figure 6(d)), the working set increases up to 25.5 GB, which exceeds the MCDRAM capacity. This partly accounts for the 5× slow down in runtime (from 34 to 173 seconds) despite only a 3× increase in number of grid points computed per time iteration.

7.5 Overhead Summary

To run an `Operator`, there are four major sources of overhead:

- ***Code generation*:** The phase during which the high-level symbolic specification is lowered into C/C++.
- ***Compilation into a shared object*:** The generated C/C++ file is compiled into a shared object by a C/C++ compiler with optimizations enabled. The time spent in this phase highly depends on the quality of the generated code.
- ***Calling the shared object*:** This requires analyzing the user input, provided at `Operator` application time, and forwarding it to the loaded shared object.
- ***Auto-tuning*:** This step is optional. Its impact varies greatly across different problem sizes and even across backends (`core`, `yask`).

On `skl8180`, Devito's turnaround times for all of these four phases are extremely quick, even in the most complex problems in which hundreds of lines of code are generated (e.g., high-order `tti`). The Intel compiler took less than 7 seconds to build `tti so` =16 at the maximum optimization level, whereas the `Operator` required around 3 seconds to emit the C code (with DSE `aggressive`). Calling the loaded shared object from *Python* takes negligible time, despite the extensive checks to validate the arguments. Auto-tuning took 3 minutes to complete (heuristic-based search); however, from a user perspective, this is hardly relevant because auto-tuning is disabled (by default) until production or benchmark runs. All of these times improve, even significantly, as the arithmetic complexity of a problem decreases (e.g., at lower orders or when considering `isotropic`).

On `knl7250`, due to weaker single-core performance, the overheads are more pronounced. It took slightly more than 1 minute to produce a shared object for `tti so` =16. Auto-tuning took around 15 minutes. Since it is unlikely that a `knl7250` will ever be used as a development platform, these overheads are easily amortized out during production runs.

With the `yask` backend, the compilation times tend to increase, although the order of magnitude is still the same as with `core`. All other phases are substantially unchanged.

For all experiments, we report the time spent in each of these phases in the logs available from the Devito Team [34].

8 FURTHER WORK

Although 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., modeling 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 ongoing effort toward adding an `ops` [32] backend, which will enable code generation for GPUs and also supports distributed memory parallelism via MPI.

9 CONCLUSION

Devito is a system to automate high-performance stencil computations. Although Devito provides a *Python* -based syntax to easily express FD approximations of PDEs, it is not limited to FDs. 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*, as well as 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] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman (Eds.). 2007. *Compilers: Principles, Techniques, and Tools* (2nd ed.). Pearson/Addison Wesley, Boston, MA. http://www.loc.gov/catdir/toc/ecip0618/2006024333.html.

[2] Martin S. Alnæs, Anders Logg, Kristian B. Ølgaard, Marie E. Rognes, and Garth N. Wells. 2014. Unified form language: A domain-specific language for weak formulations of partial differential equations. *ACM Transactions on Mathematical Software* 40, 2 (2014), 9.

[3] A. Arbona, B. Miñano, A. Rigo, C. Bona, C. Palenzuela, A. Artigues, C. Bona-Casas, and J. Massó. 2017. Simflowny 2: An upgraded platform for scientific modeling and simulation. arXiv:1702.04715.

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

[5] Susanne C. Brenner and L. Ridgway Scott. 2008. *The Mathematical Theory of Finite Element Methods*. Vol. 15. Springer, New York, NY. DOI: https://doi.org/10.1007/978-0-387-75934-0

[6] Alfonso F. Cárdenas and Walter J. Karplus. 1970. PDEL—A language for partial differential equations. *Communications of the ACM* 13, 3 (1970), 184–191.

[7] Grant O. Cook Jr. 1988. *ALPAL: A Tool for the Development of Large-Scale Simulation Codes*. Technical Report. Lawrence Livermore National Laboratory, Livermore, CA.

[8] R. Courant, K. Friedrichs, and H. Lewy. 1967. On the partial difference equations of mathematical physics. *International Business Machines (IBM) Journal of Research and Development* 11, 2 (March 1967), 215–234. DOI: https://doi.org/10.1147/rd.112.0215

[9] Kaushik Datta, Samuel Williams, Vasily Volkov, Jonathan Carter, Leonid Oliker, John Shalf, and Katherine Yelick. 2009. Auto-tuning the 27-point stencil for multicore. In *Proceedings of iWAPT 2009: The 4th International Workshop on Automatic Performance Tuning*.

[10] Steven J. Deitz, Bradford L. Chamberlain, and Lawrence Snyder. 2001. Eliminating redundancies in sum-of-product array computations. In *Proceedings of the 15th International Conference on Supercomputing (ICS'01)*. ACM, New York, NY, 65–77. DOI: https://doi.org/10.1145/377792.377807

[11] Yufei Ding and Xipeng Shen. 2017. GLORE: Generalized loop redundancy elimination upon LER-notation. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (Oct. 2017), Article 74, 28 pages. DOI: https://doi.org/10.1145/3133898

[12] Albert Farres, Claudia Rosas, Mauricio Hanzich, Alejandro Duran, and Charles Yount. 2018. Performance optimization of fully anisotropic elastic wave propagation on 2nd generation Intel Xeon Phi processors. In *Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW'18)*.

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

[14] The OpenFOAM Foundation. n.d. *OpenFOAM v5 User Guide*. Retrieved March 17, 2020 from https://cfd.direct/openfoam/user-guide/.

[15] K. A. Hawick and D. P. Playne. 2013. Simulation software generation using a domain-specific language for partial differential field equations. In *Proceedings of the 11th International Conference on Software Engineering Research and Practice (SERP'13)*. SER3829.

[16] Robert L. Higdon. 1987. Numerical absorbing boundary conditions for the wave equation. *Mathematics of Computation* 49, 179 (1987), 65–90. http://www.jstor.org/stable/2008250.

[17] Christian T. Jacobs, Satya P. Jammy, and Neil D. Sandham. 2016. OpenSBLI: A framework for the automated derivation and parallel execution of finite difference solvers on a range of computer architectures. arXiv:1609.01277.

[18] Jim Jeffers and James Reinders. 2015. *High Performance Parallelism Pearls Volume Two: Multicore and Many-Core Programming Approaches*. Morgan Kaufmann, San Francisco, CA.

[19] Andreas Klöckner. 2014. 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*. DOI: https://doi.org/10.1145/2627373.2627387

[20] Andreas Klöckner. 2016. CGen - C/C++ Source Generation from an AST. Retrieved March 17, 2020 from https://github.com/inducer/cgen.

[21] Stefan Kronawitter, Sebastian Kuckuk, and Christian Lengauer. 2016. Redundancy elimination in the ExaStencils code generator. In *Proceedings of the International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP'16)*.

[22] Christian Lengauer, Sven Apel, Matthias Bolten, Armin Größlinger, Frank Hannig, Harald Köstler, Ulrich Rüde, et al. 2014. ExaStencils: Advanced stencil-code engineering. In *Euro-Par 2014: Parallel Processing Workshops*. Lecture Notes in Computer Science, Vol. 8806. Springer, 553–564. DOI: https://doi.org/10.1007/978-3-319-14313-2_47

[23] Anders Logg, Kent-Andre Mardal, and Garth N. Wells (Eds.). 2012. *Automated Solution of Differential Equations by the Finite Element Method*. Springer. DOI: https://doi.org/10.1007/978-3-642-23099-8

[24] Mathias Louboutin, Michael Lange, Felix J. Herrmann, Navjot Kukreja, and Gerard Gorman. 2017. Performance prediction of finite-difference solvers for different computer architectures. *Computers & Geosciences* 105 (08 2017), 148–157. DOI: https://doi.org/10.1016/j.cageo.2017.04.014

[25] M. Louboutin, M. Lange, F. Luporini, N. Kukreja, P. A. Witte, F. J. Herrmann, P. Velesko, and G. J. Gorman. 2019. Devito (v3.1.0): An embedded domain-specific language for finite differences and geophysical exploration. *Geoscientific Model Development* 12, 3 (2019), 1165–1187. DOI: https://doi.org/10.5194/gmd-12-1165-2019

[26] Graham R. Markall, Florian Rathgeber, Lawrence Mitchell, Nicolas Loriant, Carlo Bertolli, David A. Ham, and Paul H. J. Kelly. 2013. Performance-portable finite element assembly using PyOP2 and FEniCS. In *Supercomputing*. Lecture Notes in Computer Science, Vol. 7905. Springer, 279–289. DOI: https://doi.org/10.1007/978-3-642-38750-0_21

[27] Mathias Louboutin and Fabio Luporini. 2019. Boundary conditions in Devito. In preparation.

[28] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, et al. 2017. SymPy: Symbolic computing in Python. *PeerJ Computer Science* 3 (Jan. 2017), e103. DOI: https://doi.org/10.7717/peerj-cs.103

[29] Simon J. Pennycook, J. D. Sewall, and V. W. Lee. 2016. A metric for performance portability. arXiv:1611.07409.

[30] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. 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)*. ACM, New York, NY, 519–530. DOI: https://doi.org/10.1145/2491956.2462176

[31] Florian Rathgeber, David A. Ham, Lawrence Mitchell, Michael Lange, Fabio Luporini, Andrew T. T. Mcrae, Gheorghe-Teodor Bercea, Graham R. Markall, and Paul H. J. Kelly. 2016. Firedrake: Automating the finite element method by composing abstractions. *ACM Transactions on Mathematical Software* 43, 3 (Dec. 2016), Article 24, 27 pages. DOI: https://doi.org/10.1145/2998441

[32] István Z. Reguly, Gihan R. Mudalige, Michael B. Giles, Dan Curran, and Simon McIntosh-Smith. 2014. The OPS domain specific abstraction for multi-block structured grid computations. In *Proceedings of the 4th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC'14)*. IEEE, Los Alamitos, CA, 58–67. DOI: https://doi.org/10.1109/WOLFHPC.2014.7

[33] William W. Symes, Dong Sun, and Marco Enriquez. 2011. From modelling to inversion: Designing a well-adapted simulator. *Geophysical Prospecting* 59, 5 (2011), 814–833. DOI: https://doi.org/10.1111/j.1365-2478.2011.00977.x

[34] The Devito Team. 2018. Devito Experimentation Framework v1.0. Retrieved March 17, 2020 from https://github.com/opesci/devito-performance/releases/tag/v1.0.

[35] Josh Tobin, Alexander Breuer, Alexander Heinecke, Charles Yount, and Yifeng Cui. 2017. Accelerating seismic simulations using the Intel Xeon Phi Knights Landing processor. In *Proceedings of the 2017 ISC High Performance Conference (ISC'17)*.

[36] Yukio Umetani. 1985. DEQSOL: A numerical simulation language for vector/parallel processors. In *Proceedings of the 1985 IFIP TC2/WG22 Conference*, Vol. 5. 147–164.

[37] Robert Van Engelen, Lex Wolters, and Gerard Cats. 1996. CTADEL: A generator of multi-platform high performance codes for PDE-based scientific applications. In *Proceedings of the 10th International Conference on Supercomputing*. ACM, New York, NY, 86–93.

[38] F. D. Witherden, A. M. Farrington, and P. E. Vincent. 2014. PyFR: An open source framework for solving advectionâĂŞdiffusion type problems on streaming architectures using the flux reconstruction approach. *Computer Physics Communications* 185, 11 (2014), 3028–3040. DOI: https://doi.org/10.1016/j.cpc.2014.07.011

[39] Charles Yount. 2015. 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'15)*. 865–870. DOI: https://doi.org/10.1109/HPCC-CSS-ICESS.2015.27

[40] Charles Yount and Alejandro Duran. 2016. 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 (SC'16/PMBS'16)*.

[41] Charles Yount, Alejandro Duran, and Josh Tobin. 2019. Multi-level spatial and temporal tiling for efficient HPC stencil computation on many-core processors with large shared caches. *Future Generation Computer Systems* 92 (March 2019), 903–919. DOI: https://doi.org/10.1016/j.future.2017.10.041

[42] Charles Yount, Josh Tobin, Alexander Breuer, and Alejandro Duran. 2016. 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 (SC'16/WOLFHPC'16)*. DOI: https://doi.org/10.1109/WOLFHPC.2016.08

[43] Zenodo/Devito. 2017. Devito v3.1. Software used in Architecture and performance of Devito, a system for automated stencil computation. DOI: https://doi.org/10.5281/zenodo.836688

[44] Yongpeng Zhang and Frank Mueller. 2012. Auto-generation and auto-tuning of 3D stencil codes on GPU clusters. In *Proceedings of the 10th International Symposium on Code Generation and Optimization (CGO'12)*. ACM, New York, NY, 155–164. DOI: https://doi.org/10.1145/2259016.2259037

[45] Yu Zhang, Houzhu Zhang, and Guanquan Zhang. 2011. A stable TTI reverse time migration and its implementation. *Geophysics* 76, 3 (2011), WA3–WA11. DOI: https://doi.org/10.1190/1.3554411 arXiv:https://doi.org/10.1190/1.3554411

Received July 2018; revised August 2019; accepted December 2019