DOI: 10.14529/jsfi200205

Dawn: a High Level Domain-Specific Language Compiler Toolchain for Weather and Climate Applications

Carlos Osuna[1]***, Tobias Wicky***[1]***, Fabian Thuering***[1,2]***, Torsten Hoefler***[3]***, Oliver Fuhrer***[1]

High-level programming languages that allow to express numerical methods and generate efficient parallel implementations are of key importance for the productivity of domain-scientists. The diversity and complexity of hardware architectures is imposing a huge challenge for large and complex models that must be ported and maintained for multiple architectures combining various parallel programming models. Several domain-specific languages (DSLs) have been developed to address the portability problem, but they usually impose a parallel model for specific numerical methods and support optimizations for limited scope operators. Dawn provides a high-level concise language for expressing numerical finite difference/volume methods using a sequential and descriptive language. The sequential statements are transformed into an efficient target-dependent parallel implementation by the Dawn compiler toolchain. We demonstrate our approach on the dynamical solver of the COSMO model, achieving performance improvements and code size reduction of up to 2x and 5x, respectively.

Keywords: GPGPU computing, DSL, weather and climate, code optimization, compiler, performance portability.

Introduction

High resolution weather and climate simulations are subject of an unprecedented scientific interest due to the urgent need to reduce uncertainties of climate projections. Despite the progress achieved in the last years, the uncertainties have remained large, and improvements in the projections are crucial for designing and adopting efficient mitigation measures.

There is clear evidence in the literature that increasing resolution is one of the key factors to reduce the uncertainty. The horizontal resolution of current state-of-the-art climate model projects range between 50 km and 100 km. At these resolutions, several physical processes of key importance, e.g. the formation and evolution of deep convection must be parametrized. However, these processes can be explicitly resolved on the computational grid at horizontal resolutions around one kilometer. Unfortunately, when employing explicit numerical solvers that need to respect the Courant-Friedrichs-Lewy (CFL) condition, an increase of 2x in horizontal resolution implies a factor 8x in computational cost. Since the resolution of climate models is constantly improving [24], they will quickly become a major workload on supercomputers which requires increased attention to their performance [25].

Although some of the most powerful supercomputers provide an extraordinary computational power, weather and climate models cannot take full advantage of these leadership class systems. The end of Dennard's scaling [7] has led to the adoption of many-core accelerators, hybridization and diversity of supercomputers. Weather and climate models are complex numerical codes that contain from hundred thousands to millions lines of codes, and the community is struggling to migrate and maintain the models for multiple computing architectures. Due to the lack of standard parallel programming models that can be used by compilers to parallelize models

[1]Federal Institute of Meteorology and Climatology MeteoSwiss, Zurich, Switzerland
[2]NVIDIA, Berlin, Germany
[3]Department of Computer Science, ETH Zurich, Zurich, Switzerland

implemented with sequential programming languages like Fortran on any architecture, domain scientists are forced to combine multiple pragma based models like OpenMP and OpenACC, in addition to usage of distributed memory parallelization with MPI. The result is a mixture of multiple compiler directives with redundant semantic and numerical algorithms, often combined with attempts to customize data layouts for different architectures using preprocessor conditionals. In an attempt to tackle the portability problem, numerous domain-specific languages (DSLs) are being developed and used to port parts of weather and climate models. However, they still require to specify parallel programming model semantics that is crucial to generate correct parallel implementations and instruct the DSL compiler with necessary information to apply key optimizations. This generates a significant amount of boilerplate and code redundancy and impacts the scientific productivity of the model developer. The result are error-prone implementations where the user must be careful to avoid data races when new computations are inserted into pre-existing templates. Even if the use of these DSLs is an important step towards providing a solution that allows to retain a single source code, they have not significantly improved the ease of use, safety in parallel programming and programmer productivity for domain scientists. Additionally, the scope of computations covered by existing DSLs is very limited, since they cannot deal with the analysis and optimizations of large numerical discretization algorithms with complex data dependencies. We present Dawn, a DSL compiler toolchain that offers a descriptive, high-level language for the domain of partial differential equation solvers using finite difference or volume discretizations. The Dawn DSL language provides a parallel programming language for weather applications with sequential semantics where the user does not have to consider data dependencies. The highly descriptive language where optimization and parallelization are abstracted significantly reduces the amount of necessary code to express the numerical algorithms in a parallel architecture and consequently improves maintainability and productivity of the domain scientist. In contrast to other high-level frameworks (e.g., PETSc or MATLAB), the domain scientist retains full control over the discretizations and solvers employed.

The Dawn DSL compiler aims at porting full components, within the scope of the language. This allows the toolchain to apply data locality optimizations across all the components of a model. The authors are not aware of any other DSL or programming model that enables global model optimizations with all the functionality required by the weather and climate domain. We demonstrate the Dawn DSL compiler on the full dynamical core of the COSMO weather and climate model [4]. The dynamical core of a weather and climate model solves the Navier-Stokes equations of motion of the atmosphere and its discretization generates the most complex computational patterns of a model. Our results show that it is feasible to port entire dynamical cores to a high-level descriptive DSL obtaining more efficient implementations and maintainable codes.

The contributions of this paper are as follows:

- We introduce Dawn, an open-source DSL compiler toolchain including front-end language and a comprehensive set of optimizers and code generators.
- We propose a high-level intermediate representation to interoperate tools and DSL front ends.
- We demonstrate the usability and performance of the Dawn DSL compiler on the dynamical core of COSMO, a representative weather and climate code.

The document is organized as follows: Section 2 describes the language and main features supported by the DSL for weather and climate models. Section 3 provides a comprehensive

description of the dawn compiler and the algorithms of the different DSL compiler passes. Finally Section 4 shows performance results for the dynamical core of COSMO and comparisons with the operational model running on GPUs.

1. Related Work

Numerous DSLs for stencil computations have been developed and proposed in the last decade in order to solve the performance portability problem. In the image processing field, Halide [22] provides a language and an autotuning framework to find an optimized set of parameters and strategies. PolyMage [18] provides instead a performance model heuristic. However, they lack in general functionality required for the specific domain of weather and climate, like 3D domains and a special treatment of the vertical dimension. Kokkos [5] provides a performance portable model for many core devices which has been demonstrated on the E3SM model [1]. The programming model contains useful parallel loop constructs and data layout abstractions. The CLAW DSL [3] is a Fortran based DSL for column based applications. It can apply a large set of transformations and generate OpenACC or OpenMP codes. However, it is limited to single column type of computations, like physical parametrizations, and not suitable for dynamical cores. STELLA [10] (and its successor GridTools) has been the first DSL running operationally for a weather model in a GPU-based supercomputer. The DSL supports finite differences methods on structured grids and is embedded in C++ using template metaprogramming. PSyclone [21] is a Fortran based DSL for finite elements/volumes dynamical cores being demonstrated for the NEMO ocean model and the LFric weather model. It relies on metadata provided together with the main stencil operators, in order to apply transformations and optimizations like loop fusion. Various tools and approaches such as Patus [2], Modesto [11], or Absinthe [12] tune low-level stencil implementations and could be combined with a stencil DSL. However, all of the existing approaches known to the authors provide a language where the user has the responsibility to declare the data dependencies via metadata, boilerplate of the language or need to resolve explicitly the data dependencies by choosing the computations that are fused into the same parallel component.

2. Abstractions for Weather and Climate

2.1. The Weather and Climate Domain

The domain we target are computational patterns of weather and climate models on structured grids where each grid cell can be addressed by an index triplet (i, j, k). We focus on algorithmic motifs with direct addressing where a series of operators are applied to update grid point values. Further, no explicit dependency of the operator on the horizontal positional indices (i, j) is assumed (with the exception of boundary conditions). This domain contains discretizations of partial differential equations using finite difference or volume methods as well as physical parametrizations. The main computational patterns resulting from these numerical discretizations are compact horizontal stencils in the horizontal plane and implicit solvers like tridiagonal solve in the vertical dimension.

In the following sections we introduce the Dawn DSL frontend (GTClang) and an intermediate representation (IR) designed as a minimal set of orthogonal concepts that can be used to represent these computational patterns with a high-level of abstraction.

2.2. GTClang Frontend

GTClang [20] is a DSL frontend that provides a high-level descriptive language for expressing finite difference/volume methods on structured grids. GTClang takes the view of a series of computations at a single horizontal location (i, j). Unlike other approaches such as the STELLA DSL [10], the language assumes a sequential model where all the data dependencies and data hazards will be resolved by the toolchain when constructing a parallel implementation from the sequential specification. The frontend language is embedded in C++ using the Clang compiler [15] to parse and analyze the C++ abstract syntax tree (AST). It provides a high level of interoperability, allowing to escape the DSL language within the same translation unit.

Figure 1 shows the main language elements of the DSL for an horizontal stencil example, while Fig. 2 shows how to execute the generated backend specific implementation from a C++ driver program.

```
globals {
  double eddlat, eddlon;
  double r_earth = 6371.229e3;
}
stencil_function avg {
  storage data;
  direction d;
  Do {
    return (data[d+1] - data[d-1])*0.5;
  }
}
stencil_function delta {
  storage data;
  offset off;
  Do {
    return (data[off] - data);
  }
}
stencil_function laplacian {
  storage data;
  Do{
    return data[i+1] + data[i-1] +
           data[j+1] + data[j-1] -
           4.0*data;
  }
}

stencil hd_smag {
 // Input-Output fields
 storage u, v;

 // Input fields
 storage hdmaskvel;
 storage[j] crlat;

 // Temporaries
 var T_sqr_s, S_sqr_uv;

 Do {
  vertical_region(k_start, k_end) {
   var frac_1_dx = crlat * eddlon;
   var frac_1_dy = eddlat / r_earth;

   var T_s = delta(j-1, v) * frac_1_dy -
       delta(i-1, u) * frac_1_dx;
   T_sqr_s = T_s * T_s;

   var S_uv = delta(j+1, u) * frac_1_dy +
       delta(i+1, v) * frac_1_dx;
   S_sqr_uv = S_uv * S_uv;

   var smag_u = math::sqrt((avg(i+1,
       T_sqr_s) +
       avg(j-1, S_sqr_uv))) - hdmaskvel;

   smag_u = math::min(0.5, math::max(0.0,
        smag_u));

   var smag_v = math::sqrt((avg(j+1,
       T_sqr_s) +
       avg(i-1, S_sqr_uv))) - hdmaskvel;
   smag_v = math::min(0.5, math::max(0.0,
        smag_v));

   u += smag_u * laplacian(u);
   v += smag_v * laplacian(v);
}}}
```

Figure 1. Smagorinsky diffusion operator example implemented using the GTClang front-end language. For simplicity some functions like delta, avg are omitted

The main language elements shown in the example are the following:

- **`stencil`** is the main computation concept that contains the declaration of all the input-output (`storage`) and temporary (`var`) fields and the `Do` body with the sequence of stencil-like computations.
- **`vertical_region`** allows to specialize computations for different regions of the vertical dimension. Atmospheric codes require to specialize equations at the boundaries or at custom regions of the vertical dimension. Since weather models do not need to specialize computations for regions of the horizontal plane, the semantic of this keyword is restricted to the

```
1
2  // define a runtime domain as a triplet of
3  // sizes and halos on each direction/dimension
4  domain dom(128,128,80,halos{3,3,3,3,0,0});
5
6  // declare all storages
7  metadata storage_info(128,128,80);
8  storage u(storage_info, "u"), v(storage_info,"v");
9  //...
10
11 hd_smag(domain, u, v, hdmaskvel, crlat);
12 hd_smag.run();
```

Figure 2. Execution in a C++ program of the smagorinsky computation declared in Fig. 1

vertical dimension. `k_start`/`k_end` are builtin identifiers marking the vertical levels of the top/bottom of the atmosphere.

- **`var`** declares a variable for a temporary computation. The dimensionality and type of memory where the temporary field will be stored is derived by specific analysis passes (Section 3.2.2).
- **`stencil_function`**, allows to define numerical functions that can be used within the `vertical_region` to increase readability of the numerical algorithm. They can be parametrized with fields (line 20), grid-point offsets (line 14) and dimensions (line 7).
- **`storage[dim]`**, declares fields of certain dimensionality (default is a 3D storage). The grid dimensionality of var declarations is deduced by an analysis pass of the toolchain, while it is explicitly for input-output storage fields.
- **`i,j,k`** are builtin identifiers for each of the cartesian dimensions.
- Neighbor grid point field access operator **`[]`**, like `u[i+1]` allows to access fields at neighboring grid points of the center of the stencil operation.
- **`global`** defines global scalar parameters, like model configuration variables, with a global scope. They can be defined at compile time or runtime.

The language assumes an array-like notation, where a loop over the entire domain is implicit and indices on dimensions are only used when accessing neighbor grid points. A GTClang statement:

```
b = a[i+1]
```

would be equivalent to the explicit array notation

```
b[h:end-h] = a[h+1:end-h+1],
```

where `h` is the halo size.

The main drawback of the array notation is that for a parallel compiler is not trivial to determine in general which statements can be fused in the same parallel region due to data dependencies. Indeed not all the statements of Fig. 1 can be inserted in the same (i, j) parallel region due to dependencies, e.g., between lines 53 and 46. Other languages or DSLs like GridTools or Kokkos delegate the responsibility for splitting the computations into parallel regions that should not contain data dependencies to the user.

Since GTClang does not expose these parallel concepts in its language, the numerical methods can be implemented in a sequential manner without considering data hazards or having to split computations into different parallel region components. This increases safety and productivity of the scientific development compared to other programming models that expose parallel constructs in their language.

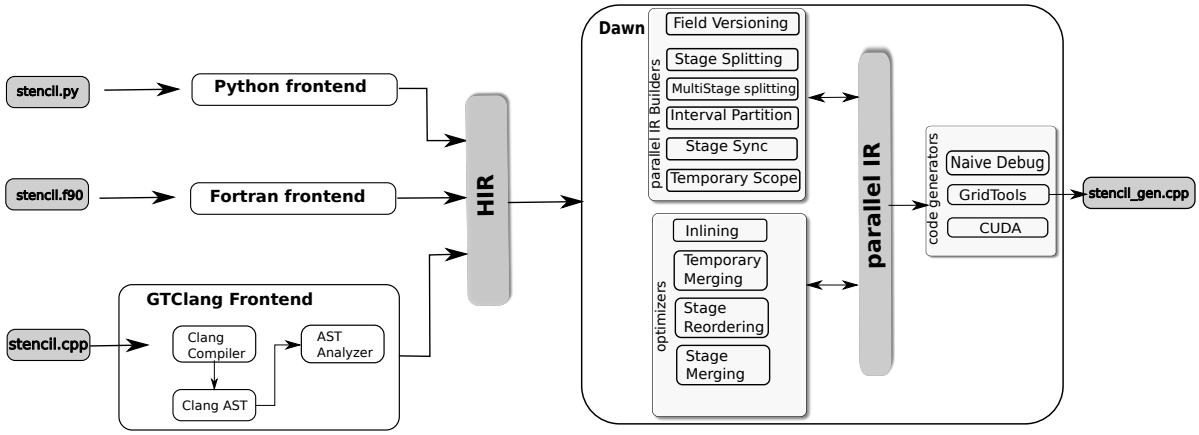


Figure 3. Architecture design of the dawn compiler

2.3. The High-Level Intermediate Representation

The high-level intermediate representation (HIR) is a representation that captures all (and only) the concepts required to express a high level language for weather and climate applications like the GTClang language presented in Section 2.2. The aim of the HIR is to provide a standard, programming language agnostic representation that enables sharing and reusing tools and optimizers such as Dawn across multiple frontend languages. Other DSLs and code-to-code translators like Fortran-based DSLs [3, 17] with the ability to parse and generate an IR will be able to transform the parsed code into HIR and use the Dawn compiler toolchain. The Dawn implementation uses Google protocol buffers to provide a specification of the HIR and thus communicates in a language-neutral manner between the DSL frontend and the DSL toolchain.

A comprehensive description of the HIR specification can be found in [19].

3. Compiler Structure

In this section, we will present the dawn [20] compiler structure and a description of all the components from the HIR to the code generation. The different layers of the compiler toolchain, including the GTClang front end, the standard interface HIR and the Dawn compiler, are shown in Fig. 3.

3.1. The Dawn Parallel IR

The HIR represents the user specification of computations in a sequential form. In order to be able to generate efficient parallel implementations from the HIR, we need to define a parallel model and map the HIR computations into that parallel model.

The Dawn compiler uses a parallel model that consist of a pipeline of computations (*Stages*) that are executed in parallel for the horizontal plane (i, j). *MultiStages* contains a list of *Stages* that are sequentially executed for each horizontal plane. Finally each *MultiStages* is executed over a vertical range. The vertical data dependencies in the user-specified computations determine the execution strategy of the vertical looping: forward, backward, or parallel. Multiple stages are then fused within the same parallel kernel connected by temporary computations stored in on-chip memory. The horizontal plane is tiled in order to fit the temporary computations into limited-size caches. However, stencil computations accessing data from grid cells of

neighboring tiles will create data races in the case of horizontal data dependencies, since parallel tile computations can not be synchronized in general. We construct a parallel model based on redundant computations [10], where all intermediate computations are computed privately by each tile.

In order to code-generate implementations that follow this parallel model, Dawn defines a parallel IR that enriches the HIR with additional concepts such as *Stages* and *MultiStages* of the parallel model. Different optimizer-passes are responsible for creating a valid parallel IR representation from the HIR. The main data structure of the parallel IR is defined as a tree, shown in Fig. 4.

Computation:
Scope of a stencil computation, contains one or more parallel kernels and fields allocated to it
-*[MultiStage]*:
A parallel kernel over the entire 3D domain, where the vertical loop direction can be specified
-vertical loop strategy: (forward —— backward —— parallel)
-*[IntervalComputation]*:
A set of stage computations defined within a vertical region
-range: [k0,k1]
-*[Stage]*:
A group of computations performed in parallel in the horizontal (i, j) plane
-*[AST]*:
The AST of the arithmetic statements

Figure 4. Parallel IR data model tree. The operator [] denotes an array of nodes

Figure 5 shows the parallel IR representation of the smagorinsky example. Since the example does not contain vertical dependencies, the organization of the HIR computations into *MultiStages* and interval computations information is trivially computed.

```
1
2  -Computation:
3    -MultiStage:
4      -vertical_loop_strategy: parallel
5      -IntervalComputation: [k_start,k_end]
6        -Stage:
7          var frac_1_dx = crlat * eddlon;
8          var frac_1_dy = eddlat / r_earth;
9
10         var T_s = delta(j-1, v) * frac_1_dy -
11           delta(i-1, u) * frac_1_dx;
12         T_sqr_s = T_s * T_s;
13         var S_uv = delta(j+1, u) * frac_1_dy +
14           delta(i+1, v) * frac_1_dx;
15         S_sqr_uv = S_uv * S_uv;
16       -Stage:
17         var smag_u = math::sqrt((avg(i+1, T_sqr_s) +
18         avg(j-1, S_sqr_uv))) - hdmaskvel;
19         smag_u = math::min(0.5, math::max(0.0, smag_u));
20       -Stage:
21         var smag_v = math::sqrt((avg(j+1, T_sqr_s) +
22         avg(i-1, S_sqr_uv))) - hdmaskvel;
23         smag_v = math::min(0.5, math::max(0.0, smag_v));
24
25         u += smag_u * laplacian(u);
26         v += smag_v * laplacian(v);
27 }}}
```

Figure 5. Parallel IR of the smagorinsky diffusion operator defined in Fig. 1

The parallel IR defines different types of data storages:

- User-declared fields: N dimensional fields that are owned by the user with a scope that goes beyond the *Computation*. The GTClang front end declares them using the *storage* keyword.
- Temporary fields: storage with a scope limited to any of the levels of the parallel IR. The data allocation and dimensionality of the field will depend on the scope and will be managed by the toolchain.
- global parameters: scalar basic types to describe non gridded data, like model configuration switches.

In addition to the parallel IR tree, a program that assembles operators in a model requires a control flow that can schedule the different computations. The *control flow AST* contains the sequence of AST nodes to define a control flow (conditionals, loop iterations, etc.) and nodes for calls to

- *Computations* defined in the parallel IR;
- boundary conditions;
- halo exchanges.

All the analyses and optimizations are performed across multiple *Computation* nodes without control flow dependencies like conditionals or iterations. Therefore program dependence graphs and control dependence analyses are not used by the toolchain. The *control flow AST* is only stored as part of the parallel IR for code generation purposes.

3.2. Optimization Process

A comprehensive list of compiler passes are responsible for organizing the HIR statements into a valid parallel IR and run optimization algorithms to prepare the IR for efficient code generation. The passes are organized in three different categories: parallel IR builders, optimization passes and safety checkers. The safety checkers contain checks for detection of ill-formed numerical codes like access to uninitialized temporaries or write-after-write (WAW) data hazards. The following will describe the most relevant passes of the first two categories:

3.2.1. Parallel IR Builders

Field Versioning. Numerical discretizations often update a field reusing the same field identifier for readability of the code and to minimize memory footprint. In the following PDE example

$$\frac{\delta u}{\delta t} = f(u) + g(u), \tag{1}$$

that is discretized to

$$u_{t+1} = u_t + \Delta t(f(u_t) + g(u_t)) \tag{2}$$

if $f(u_t)$ or $g(u_t)$ involves an access to neighbour grid points in the parallel dimension, the field update can generate data races if the right-hand side (RHS) and the left-hand side (LHS) are evaluated in the same parallel region. Often this is solved by using a double buffering technique in the model implementation:

$$u\_out = u\_in + dt * (f(u\_in) + g(u\_in)). \tag{3}$$

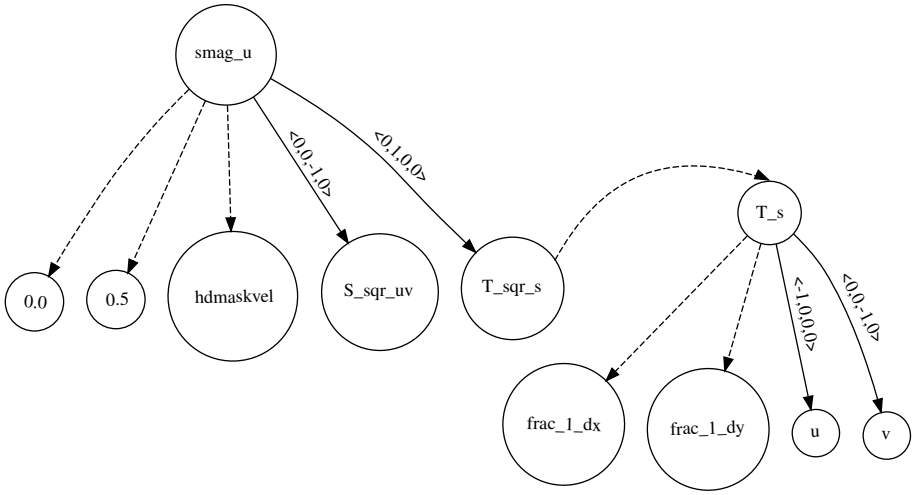


Figure 6. Horizontal data dependency DAG of a subset of the smagorinsky diffusion operator (Fig. 1)

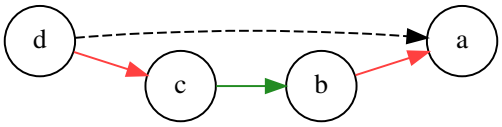


Figure 7. Vertical data dependency DAG of the anti dependent pattern example (Fig. 8)

Since the high-level DSL abstracts away the details of parallelization, it is not possible to know if RHS and LHS are evaluated within the same parallel region. Therefore GTClang allows to update fields with stencil computations that generate write-after-read (WAR) data dependencies. The field versioning pass will create versions of the field to ensure data hazards are resolved in parallel regions. Read-after-write (RAW) are resolved by the stage splitting pass and write-after-write (WAW) are legal but should be protected with a compiler warning, since the numerical algorithm might be ill-formed.

The field versioning pass operates on the following DAG formulation: field accesses are represented by nodes in the DAG where edges connect fields according to data dependencies. Edges are annotated with the horizontal *stencil extent* with the notation `<iminus, iplus, jminus, jplus>`.

There are two type of edges: solid lines edges for data dependencies with a (not null) horizontal stencil extent and dashed lines for connecting data with null extents or literals (see Fig. 6).

The algorithm identifies WAR hazards by finding strongly connected components (SCC) in the DAG where at least one of the nodes is a non temporary field and contains at least a solid edge. Temporary fields have a special allocation with redundant halos per block which allow WAR, unless they are within a single statement.

Stage Splitting. The stage splitting pass will organize the original sequence of statements, $(stmt_n)$, into *stages* of the parallel IR in order to resolve (RAW) data dependencies that would introduce data races in the horizontal parallelization.

An example of such data races is observed in the horizontal diffusion smagorinsky example in the following lines:

```
1 T_sqr_s = T_s * T_s;
2 var smag_u = math::sqrt((avg(i+1, T_sqr_s) + avg(j-1, S_sqr_uv))) - hdmaskvel;
```

Algorithm 1: Stage splitting algorithm

```
for stmt in J(stmt_n) do
    D' ← D ;
    insert stmt accesses in D';
    if ∃{i, j} such that D_ij == 1 and ∃j' such that D_jj' ! = 0 then
        create new stage from D';
        D ← ∅ ;
        insert stmt accesses in D ;
    else
        D ← D'
```

The stage splitting pass resolves this by placing statements with data dependencies in separate *stages*.

The algorithm finds a partition of the DAG (see Fig. 6) where any solid edge can only be connected to leaves of the DAG. Let D_{ij} be the adjacency matrix of the DAG, where elements of the matrix can take two values for elements that are connected: 1 for a solid edge and 2 for dashed edges. The algorithm (Algorithm 1) iterates over the statements in the reverse sequence of statements, $J(stmt_n)$, where J is the backward identity matrix.

Multistage Splitting. It is common in weather and climate applications to find vertical implicit solvers that introduce a loop-carried dependence (see Fig. 8).

Anti dependence patterns are also allowed for read-only fields or field accesses before a write.

```
vertical_region(k_start, k_start) {
  phi = 0;
}
vertical_region(k_start, k_end) {
  phi = phi[k-1];
}
```

```
vertical_region ( k_start+1 , k_end-1 ) {
  b = (a[k+1])*0.5;
  c = b[k-1];
  d = c[k+1]*a;
}
```

Figure 8. Examples of (left) vertical solver and (right) vertical antidependence pattern

On the contrary, an anti dependence pattern on a field after a write statement would access outdated or uninitialized data. The Multistage splitting identifies anti dependence patterns on temporary accesses and fixes them by splitting and creating a new multistage with reverse vertical loop ordering.

The algorithm is similar to stage splitting. It processes a graph where edges are colored green for in loop data dependence and red for anti dependencies (see Fig. 7).

The algorithm traverses each edge of the graph. If a red edge is found, the loop order is reversed to fix the anti dependence. If a red and a green edges are traversed, then the multistage is split. Edges on leaves are ignored, since they connect to read-only data.

Interval Computation Partition. The user DSL code is provided as an ordered sequence of (in general) overlapping *vertical region* computations. In order to be able to fuse computations of the different interval regions into a single vertical loop, the interval computation partition pass will reorganize the ordered set into a non overlapping ordered set of *interval computations* of the parallel IR (Section 3.1). The sequence can be described as an interval graph where edges connect interval nodes that are overlapping. From there, the interval partitioning algorithm derives a set of non-overlapping *interval computations* as follows:

Let e(X,Y) be any edge that connects two nodes X and Y.

for *e(X,Y) : G* **do**
  $G' \leftarrow \{G \setminus \{X, Y\}, (X \cap Y), (X \ominus Y)\}$;

where the set operations on *interval computations* are defined as:

$$\{I^x, (stmt_n^x)\} \cap \{I^y, (stmt_n^y)\} = \{I^x \cap I^y, (stmt_n^x, stmt_n^y)\} \tag{4}$$

$$\{I^x, (stmt_n^x)\} \ominus \{I^y, (stmt_n^y)\} = \begin{cases} \{I^x \setminus I^y, (stmt_n^x)\} \\ \{I^y \setminus I^x, (stmt_n^y)\} \end{cases} \tag{5}$$

assuming that $IC^x = \{I^x, (stmt_n^x)\}$ and $IC^x <= IC^y$ in the ordered set of *interval computations*. Transformations are iteratively applied until non of the nodes are connected.

Scope of temporaries. The toolchain will size the dimensionality of the temporary storages according to the scope of the temporary usages in the parallel IR tree (Fig. 4). The scope of temporaries pass will determine the lifetime and scope of each temporary, and optimize the dimensionality required accordingly.

3.2.2. Optimization passes

The first instance of the parallel IR would generate legal parallel implementations but is still non optimized and requires further transformations in order to produce efficient implementations. This is done with a set of optimizations presented in this section. All the optimizations performed in the toolchain are domain specific based on analyses of the domain specific, high-level information stored in the parallel IR.

Stage Reordering. The compiler passes discussed in Section 3.2.1 are necessary to map a sequential description of the numerical algorithms into the parallel IR model from Section 3.1. However, the splitting algorithms tend to generate a large number of *Stages* and *MultiStages*. The stage reordering will reorder *Stages* according to data dependencies in order to group and merge them together, increasing the data locality of the algorithms. The algorithm operates on the stage dependency DAG, where a stage S_1 depends on stage S_2 if and only if:

- The vertical intervals where S_1 and S_2 are defined overlap.
- Both access at least one common field with the policies defined as in Tab. 1.

Table 1. IO policies to consider a data hazard between stages

		S1 policy		
		Input	Output	Input-Output
S2 policy	Input		x	x
	Output	x		x
	Input-Output	x	x	x

Table 1 extends the definition of write-after-read (WAR), write-after-write (WAW) and read-after-write (RAW) hazards [14] for a compiler framework without single static assignment, where input-output accesses to a field are allowed for a single statement.

The algorithm iterates over all the stages in a reverse order and finds the leftmost position in the tree where the stage can be moved, accordingly to the following criteria:

- If the *Stage* is moved into another *MultiStage*, the loop orders must be compatible. A forward order *Stage* can be inserted into a parallel but not into a backward order *MultiStage*.
- A *Stage* S_1 can be moved over S_2 only if there is no path from S_1 to S_2 in the stage dependency DAG.
- Placing a stage into a new *MultiStage* should not introduce any anti dependence in the vertical dimension (see multistage splitting pass).

Stage Merging. As a result of the stage and multistage splitting pass, potentially many fine grained stages might be generated. The stage reordering pass will naturally group stages together that are connected via data dependencies, increasing data locality. Since every stage requires synchronize, the stage merging pass will merge various stages into a single one. It contains two modes:

- merge stages that are trivially mergeable since they specify the same level of redundant computations [10];
- merge stages even if they specify different level of redundant computations.

Temporary Merging. As a result of the field versioning pass, or usage of many temporaries, the parallel IR usually contains more temporary allocations than required. This increases the memory footprint and the load on the scratchpad memory. The temporary merging pass will reduce the number of temporaries to the minimum required. The pass operates on a reduced version of the data dependency DAG (Fig. 6) where only temporary fields are represented as nodes, and where two nodes are connected if there is a path that connects them in the original DAG. A coloring algorithm of the DAG of temporaries will identify the required number of temporary fields, where nodes with the same color will share the same temporary identifier.

Inlining. The stage splitting pass will generate a pipeline of stage computations that require synchronization of the parallel computational units, since stages are connected via horizontal data dependencies. The stage computations are executed in a tiled decomposition of the domain, allowing to cache the data that connects the different stages in some type of fast on-chip memory.

```
stencil hori_diff {
 storage dphi, phi, c;

 Do {
  vertical_region(k_start, k_end){
   lap = -4*phi + c*(phi[i+1] +
      phi[i-1] + phi[j+1] + phi[j-1);
   dphi = -4*lap + c*(lap[i+1] +
      lap[i-1] + lap[j+1] + lap[j-1);
}}};
```

```
stencil hori_diff {
 storage dphi, phi, c;

 Do {
  vertical_region(k_start, k_end){
   dphi = (16+4*c*c)*phi
     -8*c*(phi[i-1]+phi[i+1]+phi[j-1]+phi[j+1])+
     c*c*(phi[i+2]+phi[i-2]+phi[j+2]+phi[j-2] +
     2*c*c*(phi[i+1,j+1]+phi[i+1,j-1]+
     phi[i-1,j+1]+phi[i-1,j-1]));
}}};
```

Figure 9. Fourth-order diffusion operator as a two stage Laplacian operator in DSL form (left) and the dawn inlined version (right)

Alternatively, any intermediate computation that is not part of the input/output field declaration of the computation can be inlined, avoiding the memory operations but generating a larger stencil matrix computation. An example of a double Laplacian stage computation of a fourth-order diffusion operator is inlined in Fig. 9.

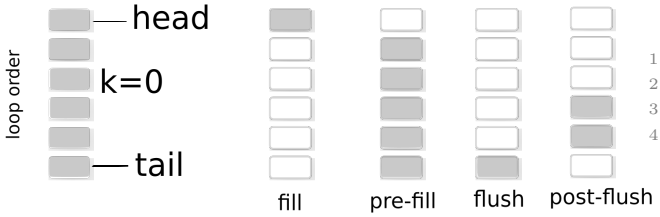
The algorithm traverses the reverse sequence of statements of each interval computation finding assignments on temporary fields that are only accessed in the parallel dimensions (i, j). The right hand side of the assignment is stored as the definition of a function that computes the temporary and the assignment stmt is removed. If the assignment statement depends on local

variable declarations, they need to be promoted to temporary fields (with the right dimensionality) before recursively inlining all data dependencies of the RHS of the assignment.

A second iteration over the reverse sequence of statements will replace all field accesses to the temporary by the function definition evaluated at the stencil offset of the temporary access.

The pre-computation and inlined algorithms exhibit completely different arithmetic intensity. Performance of the different versions will depend on the computation (stencil shape, memory accesses, ...) and the hardware architecture and its memory subsystem.

Vertical register caching detection. An important optimization for implicit vertical solvers is to cache in registers a ring buffer of near k accesses reused through the vertical iteration, as in the following example: The ring buffer keeps values of vertical levels that will still be accessed from the vertical iteration of neighbour vertical levels, and it will synchronize values with the field in main memory whenever required according to the input-output pattern of the computation. Often the levels at the head of the ring buffer needs to be filled from the main memory for input accesses while values of the tail of the ring buffer must be flushed into main memory from the ring buffer. Additionally a pre-fill/post-flush operation of the ring buffer before/after processing the *Interval Computation* might be needed to synchronize the required sections of the ring buffer (Fig. 10).




```
1 vertical_region(k_start+3, k_end-2) {
2   a = (a[k+2] + a[k+1] + a +
3        a[k-1] + a[k-2])*0.2;
4 }
```

Figure 10. (Left) representation of the ring buffer and the different sections that will require synchronization with main memory for the vertical average example operation (right)

3.3. Code Generation

The last step in the toolchain is the code generation. The optimizer passes have organized the original HIR into a structured parallel IR that contains all the components required for generating an efficient parallel implementation with stencil computations, halo exchanges and boundary conditions. Dawn supports three code generators:

- a naive C++ sequential generator, used for debugging purposes;
- a GridTools DSL generator;
- a specialized CUDA generator.

All the existing generators make use of different GridTools components like the multidimensional storage facility and the halo exchange library. In particular the use of the multidimensional storage allows to escape the DSL language by using the storage objects that abstract away the details of memory layouts of the fields.

4. Experimental Results

In this section we present performance results obtained for the most relevant dynamical core operators of COSMO. They provide a diverse set of computational patterns of weather

and climate application in Cartesian grids. The performance baseline of the dynamical core of COSMO has been analyzed in detail for NVIDIA GPUs [8].

The benchmarks are collected on the Piz Daint system at the Swiss National Supercomputing Centre (CSCS). The nodes are equipped with a 12-core Intel Xeon CPU (E5-2690 v3 @ 2.60 GHz) and an NVIDIA Tesla P100. Each node has 16 GB of HBM2 memory on the GPU. The nominal peak memory bandwidth of the GPU is 732 GB/s. All executables were compiled using gcc 7.3 and the nvvc compiler shipped with CUDA 10.0. Timing information is collected using CUDA events. All the experiments are verified by comparing the output of the optimized generated code with the naive C++ code generation on input data artificially generated using smooth mathematical functions.

4.1. COSMO Dynamical Core Results

Figure 11 shows the performance comparison between the production GPU code using the STELLA DSL and Dawn CUDA generation, for the most relevant stencils of the dynamical core of COSMO and a domain size of 128x128x80. Although Dawn does not support yet the STELLA tracer functionality that performs the same computation on multiple tracer fields in the same parallel kernel, two tracer operators (AdvPDBottX/Y) were added where the optimization in STELLA has been disabled for the comparison purposes. The data shown in the plots are showing the harmonic mean $\tilde{x}^{(h)} = \frac{n}{\sum_{i=1}^{n}(1/x_i)}$ [6, 13]. Errors were removed from Fig. 11 since they are not perceptible at the scale of the figure.

The performance of the Dawn CUDA backend obtained on P100 GPUs outperforms the STELLA optimized GPU production code, with performance improvements that vary from 2.62x (for HoriAdvPPTP operator) to same performance.

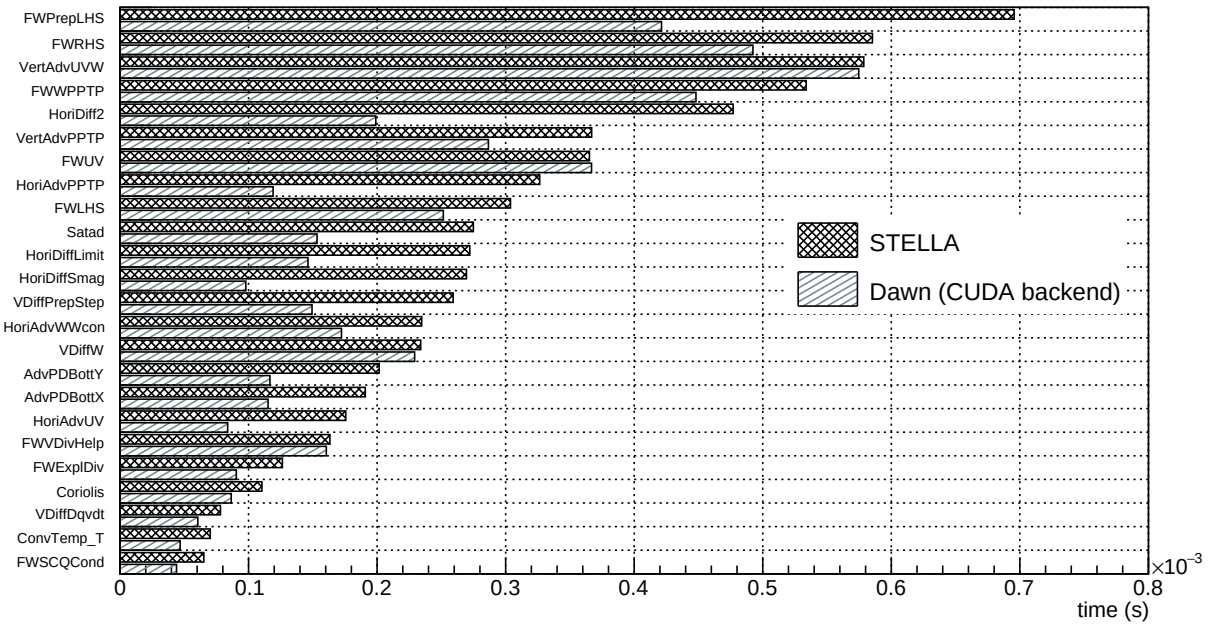


Figure 11. Performance of individual stencil objects of the COSMO dynamical core on P100 NVIDIA GPUs

In order to understand importance of the most relevant optimizers of Dawn, Tab. 2 evaluates the impact of disabling them for some of the components of the dynamical core of COSMO.

Table 2. Time (ms) for P100 GPU of some components of the dynamical core of COSMO measured with some of the Dawn optimizations disabled

	Full Opt	No stage reordering/stage merging
FastWaves	2.6	4.5
		No vert register caching
VertUVW	0.57	0.86
		No temporary merging
HoriDiffSmag	0.098	0.13

Table 3. Computational time (ms) of one time step of the fast waves solver of the dynamical core of COSMO for a P100 GPU

	time (ms)
STELLA	3.08902
Dawn	2.680

4.2. Global Optimizations

The STELLA based dynamical core of COSMO [10] was implemented as a set of complex stencil operators (see Fig. 11) that are glued together by a C++ driver code. The driver code connects all stencil components, implements time iterations, manage data storages, and performs other administrative functions. Contrary to Dawn, STELLA requires the user to organize the code in components that do not contain data dependencies.

Therefore, there is a compromise in designing how many computations are fused within each stencil component that will increase data locality but at the same time increases the complexity of the data dependencies that need to be understood by the user to produce correct and efficient code. Most of the stencils STELLA stencils of the dynamical core define no more than 4–5 *Stages* and 2–3 *MultiStages*. This approach has the following drawbacks:

- It limits the data locality optimizations that can be performed by the DSL since only a limited scope of computations are expressed within a DSL stencil object.
- It requires infrastructure that glues all the DSL stencil objects together increasing boilerplate required to composed a full dynamical core.

One of the advantages of GTClang/Dawn compared to other approaches is the possibility to express entire models within the DSL language. We demonstrate this on the fast waves component of the dynamical core of COSMO, that is composed by 11 STELLA stencil objects, and a total size of approximately 4K lines of code.

The GTClang/Dawn implementation of the fast waves reduces the amount of lines of code to approximately 800. The performance results are summarized in Tab. 3.

The main optimizer pass that allows to integrate large stencil computations is the *stage reordering pass*. The same fast waves without the stage reordering pass takes 4.5 ms (Tab. 2). As shown in Tab. 3, the CUDA implementation generated by Dawn outperforms the version in production using the STELLA DSL. However, the *stage reordering pass* is an algorithm that tends to fuse computations together as much as possible within the same kernel, which

in general will not lead to the most efficient implementation. An optimal solution should be driven by a performance model that minimizes HBM2/DDR memory accesses and at the same time minimizes on-chip resource consumption like shared memory or registers. The optimization problem is NP-hard and not solved for the complexity of a full dynamical core, although there are approaches that find solutions for a smaller problem sizes [11, 12, 23].

4.3. Maintainability and Model Development Productivity

One of the recurring parameters for maintainability across various models is lines of code (LOC). In order to quantify the gain in maintainability, we measured this for the fast waves of the dynamical core of COSMO, where the GTClang implementation (800 LOC) requires a factor 5x less than the operational code (4200 LOC). With similar order of magnitude the original Fortran implemention of the COSMO consortium requires 5000 LOC, as well as the optimized CUDA generated code of Dawn.

Therefore, GTClang/Dawn considerably reduces the amount of code required to express numerical algorithms which will increase the model development productivity and improve the model maintainability. However, there are other metrics in addition to LOC that contribute to significantly improve the maintainability:

Lack of Parallelism. Since the Dawn DSL does not expose parallelism to the user, a significant amount of boilerplate code as well as code complexity is no longer present in the user code. This does not only decrease the LOC but also increases safety, since parallel errors typical of programming models like OpenMP/OpenACC cannot occur.

Lack of Optimization. Since all the optimizations are performed by the Dawn toolchain, the GTClang language mostly expresses only the numerical algorithm. All optimizations and hardware dependent code, such as tiling, register or scratch-pad software managed cache, loop nesting, etc., hinder the scientific development. The use of a high-level language like GTClang increases considerably the readability of the numerical algorithm and model development productivity.

Reduction of Driver Code. As shown for the fast waves, the possibility to develop full models within the DSL, removes the necessity of complex infrastructure to glue all the stencil objects, data management and driver code that increases the overall maintenance of the model.

Conclusions

We have presented Dawn, a high-level DSL compiler toolchain that solves the performance portability problem of weather and climate applications. The DSL compiler is designed as a modern modular compiler. We demonstrated the usage of Dawn on the dynamical core of COSMO and presented performance results for the CUDA back end of Dawn on P100 GPUs. All the stencil computations outperform the optimized production code using the STELLA DSL, obtaining speedup factors of up to 2x. This significantly reduces the amount of code required (up to a factor of 5x). More importantly, the lack of explicit parallelism in the HIR and GTClang language provides a safe (against parallel errors) and highly productive scientific development environment.

The dynamical core is the most computationally expensive component of the model, accounting for 60 % of the total simulation time. Furthermore, it is the most complex in terms of computational patterns. Other components like the physical parametrizations contain column

based only computations that are subset of the patterns supported by dawn for dynamical cores. Therefore the DSL toolchain proposed is applicable to entire models. Although the version of dawn presented here is demonstrated on the COSMO model, its applicability is not restricted to regional models. Additional development projects are porting the advection operators of the operational ocean model of NEMO [9]. Furthermore, the dawn collaboration is extending the toolchain in order to support two new categories of global weather and climate models:

- Cube sphere grids (adding support for corner and edge specializations). Current developments are working on porting the dynamical core of the FV3 model to DSL using dawn [16].
- Global models on triangular grids. New language extensions will allow to port models that can not be described with Cartesian operators. A version of the dynamical core of ICON [26] model is being developed using the DSL based on dawn.

Dawn is the only high-level DSL compiler known to the authors that allows to express an entire weather and climate model using a concise, simple and sequential language, and delivers an optimized model implementation that outperforms operational models even on modern GPU architectures.

Future developments will allow to apply this novel DSL language and compiler to a wider range of global models, and demonstrate its applicability on two of the most renowned models like FV3 and ICON.

References

1. Bertagna, L., Deakin, M., Guba, O., Sunderland, D., Bradley, A.M., Tezaur, I.K., Taylor, M.A., Salinger, A.G.: Hommexx 1.0: A performance portable atmospheric dynamical core for the energy exascale earth system model. Geoscientific Model Development Discussions 2018, 1–23 (2018), DOI: 10.5194/gmd-2018-218

2. Christen, M., Schenk, O., Burkhart, H.: Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In: 2011 IEEE International Parallel Distributed Processing Symposium, 16-20 May 2011, Anchorage, AK, USA. pp. 676–687. IEEE (2011), DOI: 10.1109/IPDPS.2011.70

3. Clement, V., Ferrachat, S., Fuhrer, O., Lapillonne, X., Osuna, C.E., Pincus, R., Rood, J., Sawyer, W.: The CLAW DSL: Abstractions for performance portable weather and climate models. In: Proceedings of the Platform for Advanced Scientific Computing Conference, Basel, Switzerland. pp. 2:1–2:10. ACM, New York, NY, USA (2018), DOI: 10.1145/3218176.3218226

4. Doms, G., Baldauf, M.: A Description of the Nonhydrostatic Regional COSMO-Model – Part I: Dynamics and Numerics. COSMO – Consortium for Small-Scale Modelling (2015), `http://cosmo-model.org/content/model/documentation/core/cosmoDyncsNumcs.pdf`

5. Edwards, H.C., Trott, C.R., Sunderland, D.: Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. Journal of Parallel and Distributed Computing 74(12), 3202–3216 (2014), DOI: 10.1016/j.jpdc.2014.07.003

6. Fleming, P.J., Wallace, J.J.: How not to lie with statistics: the correct way to summarize benchmark results. Communications of the ACM 29(3), 218–221 (1986)

7. Frank, D.J., Dennard, R.H., Nowak, E., Solomon, P.M., Taur, Y., Wong, H.S.P.: Device scaling limits of Si MOSFETs and their application dependencies. Proceedings of the IEEE 89(3), 259–288 (2001), DOI: 10.1109/5.915374

8. Fuhrer, O., Chadha, T., Hoefler, T., Kwasniewski, G., et al.: Near-global climate simulation at 1 km resolution: establishing a performance baseline on 4888 GPUs with COSMO 5.0. Geoscientific Model Development 11(4), 1665–1681 (2018), DOI: 10.5194/gmd-11-1665-2018

9. Gurvan, M., Bourdall-Badie, R., Bouttier, P.A., Bricaud, C., et al.: NEMO ocean engine (2017), DOI: 10.5281/zenodo.3248739

10. Gysi, T., Osuna, C., Fuhrer, O., Bianco, M., Schulthess, T.C.: STELLA: a domain-specific tool for structured grid methods in weather and climate models. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 15-20 Nov. 2015, Austin, TX, USA. pp. 1–12. IEEE (2015), DOI: 10.1145/2807591.2807627

11. Gysi, T., Grosser, T., Hoefler, T.: MODESTO: Data-centric Analytic Optimization of Complex Stencil Programs on Heterogeneous Architectures. In: Proceedings of the 29th International Conference on Supercomputing, Newport Beach, CA, USA. pp. 177–186. ACM, New York, NY, USA (2015), DOI: 10.1145/2751205.2751223

12. Gysi, T., Grosser, T., Hoefler, T.: Absinthe: Learning an Analytical Performance Model to Fuse and Tile Stencil Codes in One Shot. In: Proceedings of the 28th International Conference on Parallel Architectures and Compilation Techniques, 23-26 Sept. 2019, Seattle, WA, USA. pp. 370–382. IEEE (2019), DOI: 10.1109/PACT.2019.00036

13. Hoefler, T., Belli, R.: Scientific Benchmarking of Parallel Computing Systems. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 15-20 Nov. 2015, Austin, TX, USA. pp. 1–12. ACM (2015), DOI: 10.1145/2807591.2807644

14. Kuck, D.J., Kuhn, R.H., Padua, D.A., Leasure, B., Wolfe, M.: Dependence graphs and compiler optimizations. In: Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Williamsburg, Virginia. pp. 207–218. ACM, New York, NY, USA (1981), DOI: 10.1145/567532.567555

15. Lattner, C.: LLVM and Clang: Next generation compiler technology. In: The BSD conference, May 2008, Ottawa, Canada. vol. 5 (2008)

16. Lin, S.J.: A "vertically Lagrangian" finite-volume dynamical core for global models. Monthly Weather Review 132(10), 2293–2307 (2004), DOI: 10.1175/1520-0493(2004)132¡2293:AVLFDC¿2.0.CO;2

17. Melvin, T., Mullerworth, S., Ford, R., Maynard, C., Hobson, M.: LFRic: Building a new Unified Model. In: EGU General Assembly Conference Abstracts. EGU General Assembly Conference Abstracts, vol. 19, p. 13021 (2017)

18. Mullapudi, R.T., Vasista, V., Bondhugula, U.: Polymage: Automatic optimization for image processing pipelines. SIGARCH Comput. Archit. News 43(1), 429–443 (2015), DOI: 10.1145/2786763.2694364

19. Osuna C., Clement V.: MeteoSwiss-APN/HIR 0.0.1 (2019), DOI: 10.5281/zenodo.2629314

20. Osuna C., Thuering F., Wicky T., Dahm J., et al.: MeteoSwiss-APN/dawn: 0.0.2 (2020), DOI: 10.5281/zenodo.3870862

21. Porter, A.R., Appleyard, J., Ashworth, M., Ford, R.W., Holt, J., Liu, H., Riley, G.D.: Portable multi- and many-core performance for finite-difference or finite-element codes – application to the free-surface component of NEMO (NEMOLite2D 1.0). Geoscientific Model Development 11(8), 3447–3464 (2018), DOI: 10.5194/gmd-11-3447-2018

22. Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., Amarasinghe, S.: 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, Seattle, Washington, USA. pp. 519–530. ACM, New York, NY, USA (2013), DOI: 10.1145/2491956.2462176

23. Rawat, P.S., Rastello, F., Sukumaran-Rajam, A., Pouchet, L.N., Rountev, A., Sadayappan, P.: Register optimizations for stencils on GPUs. ACM SIGPLAN Notices 53(1), 168–182 (2018), DOI: 10.1145/3178487.3178500

24. Schär, C., Fuhrer, O., Arteaga, A., Ban, N., et al.: Kilometer-scale climate models: Prospects and challenges. Bulletin of the American Meteorological Society 101(5), E567–E587 (2020), DOI: 10.1175/BAMS-D-18-0167.1

25. Schulthess, T., Bauer, P., Fuhrer, O., Hoefler, T., Schaer, C., Wedi, N.: Reflecting on the goal and baseline for exascale computing: a roadmap based on weather and climate simulations. Computing in Science and Engineering (CiSE) 21(1), 30–41 (2019), DOI: 10.1109/MCSE.2018.2888788

26. Zangl, G., Reinert, D., Ripodas, P., Baldauf, M.: The ICON (ICOsahedral Non-hydrostatic) modelling framework of DWD and MPI-M: Description of the non-hydrostatic dynamical core. Quarterly Journal of the Royal Meteorological Society 141(687), 563–579 (2015), DOI: 10.1002/qj.2378