Domain-Specific Multi-Level IR Rewriting for GPU

The Open Earth Compiler for GPU-Accelerated Climate Simulation

Tobias Gysi[1], Christoph Müller[1], Oleksandr Zinenko[2], Stephan Herhut[2], Eddie Davis[3], Tobias Wicky[3]
Oliver Fuhrer[3], Torsten Hoefler[1], Tobias Grosser[1]
ETH Zurich[1], Google[2], Vulcan[3]
{first.last}@inf.ethz.ch, {zinenko, herhut}@google.com, {oliverf, eddied, tobiasw}@vulcan.com

***Abstract*—Traditional compilers operate on a single generic intermediate representation (IR). These IRs are usually low-level and close to machine instructions. As a result, optimizations relying on domain-specific information are either not possible or require complex analysis to recover the missing information. In contrast, multi-level rewriting instantiates a hierarchy of dialects (IRs), lowers programs level-by-level, and performs code transformations at the most suitable level. We demonstrate the effectiveness of this approach for the weather and climate domain. In particular, we develop a prototype compiler and design stencil- and GPU-specific dialects based on a set of newly introduced design principles. We find that two domain-specific optimizations (500 lines of code) realized on top of LLVM's extensible MLIR compiler infrastructure suffice to outperform state-of-the-art solutions. In essence, multi-level rewriting promises to herald the age of specialized compilers composed from domain- and target-specific dialects implemented on top of a shared infrastructure.**

I. Introduction

Domain-specific approaches are revolutionizing the generation of high-performance device-specific code and sparked the development of powerful domain-specific language (DSL) frameworks, often achieving performance numbers unattainable for general-purpose compilers [1]–[6]. For example, Halide [7] automated the generation of high-performance code for image processing, XLA [8] exploited domain-specific compilation to accelerate deep learning, and Stella [9] was the first to move the weather and climate simulation to GPUs leading to 2.9× speedup [10].

The broad success of domain-specific compilers—over time—also exposed their largest weakness: their one-off implementations mostly separated from general-purpose production compiler pipelines. Halide, XLA, Stella, and others are specialized solutions for their respective domains that are not designed with reusability in mind. The small number of reusable compiler infrastructures, research-oriented such as ROSE [11] or production such as LLVM [12], evidences of a significant effort required to design and maintain the infrastructure compared to implementing domain-specific functionality. As a result, the ongoing trend of designing standalone DSL compilers compartmentalizes the developer communities, spreads the efforts, hinders innovation transfer, and leads us to ask: "how can we design a domain-specific compiler that (a) is cleanly decoupled from user-facing front-ends, (b) makes it easy to implement domain-transformations, and (c) clearly separates potentially generic components?"

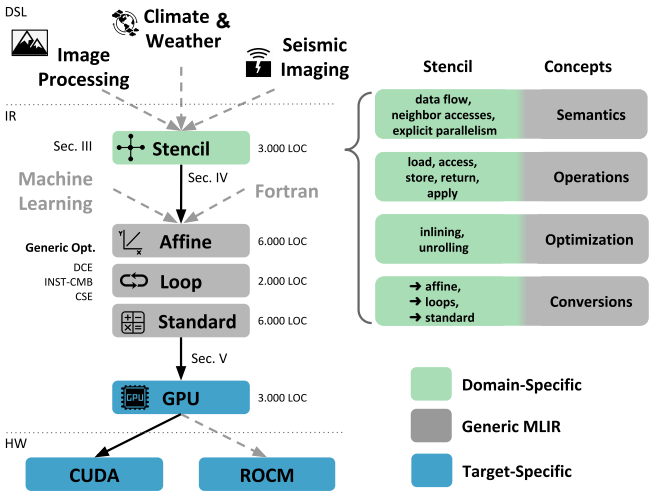

Fig. 1: The Open Earth Compiler.

We take a practical case study-based approach to addressing this question by designing and implementing a domain-specific compiler for weather and climate modeling. While this domain uses the stencil computational pattern found in image processing [7] or seismic imaging [13], it often requires radically different optimization strategies to reach maximum performance [9]. Weather and climate models [14], [15] operate on 3D domains and execute bandwidth-limited low-order stencils containing control flow. In comparison, image processing pipelines apply regular stencils to 2D data structures, and seismic imaging stencils are high-order and compute-intensive. On the other hand, the underlying abstraction of loops over multi-dimensional arrays, the arithmetic optimizations, and the conversion to device-specific GPU code is mostly identical across these domains, yet often reimplemented [16].

We propose to design DSL compilers using multi-level IR rewriting. This approach is a combination of (a) intermediate representations (IR) based on Static Single Assignment form (SSA) [17], (b) operations with high-level semantics, and (c) progressive lowering, which provides an effective framework for reusable domain-specific high-performance code generation. SSA-based IRs allow us to reuse optimizations from general-purpose compilers [18]. High-level operations concisely encode domain properties and make them readily available as, e.g., SSA data flow without a need for costly

analyses. Progressive lowering makes it natural to preserve domain information, to express transformations as high-level peephole optimizations [19], and to introduce reusable lower-level abstractions. The recently introduced MLIR compiler infrastructure [20] allows us to instantiate production-quality compiler IRs that follow the practice-proven IR design principles developed in LLVM [12] over the last 15 years.

The Open Earth Compiler we implemented (Fig. 1) is the first end-to-end compilation flow that leverages multi-level IR rewriting for high-performance code generation. Its core consists of a set of MLIR dialects, i.e., collections of domain-specific operations and transformations, and conversions between them. The Open Earth Compiler optimizes programs by progressively converting them from higher-level domain-specific dialects to lower-level platform-specific ones, using peephole-style rewrite patterns. Each dialect defines an abstraction that makes relevant analyses inexpensive and transformations convenient to implement.

The compilation process starts from the stencil dialect (Sec. IV) designed as a target for various user-facing DSLs as well as a data structure for domain-specific transformations such as stencil inlining (Sec. V-A). Stencils are then lowered through a series of IRs featuring explicit loops, affine index computations, and standard arithmetic instruction, all of which are readily available in MLIR [20], together with loop- and value-level transformations such as unrolling or common subexpression elimination. These IRs let us target a structured loop abstraction instead of low-level "goto"-based SSA IR commonly found in compiler backends. We use this structure to design a generic GPU kernel dialect (Sec. VI) and to implement loop-to-kernel conversion using simple patterns based on the parallelism information preserved from the stencil level, thus avoiding expensive GPU mapping algorithms [21], [22]. The complete pipeline transforms our high-level climate-code into a fast target-specific binary.

While the stencil dialect is generic enough to cover a range of applications (e.g., image processing or seismic imaging), our focus is excellent performance for the climate domain. The semantics of our stencil operations enable us to replace complex sequences of loop transformations with generic instruction-level transformations, e.g., redundancy elimination, requiring little analysis to ensure validity. Other domains can adapt our stencil dialect to their transformation needs, or reuse only the mid- and low-level abstractions. We demonstrate that thanks to the multi-level IR rewriting, developing a domain-specific compiler with reusable components is surprisingly simple provided a sufficiently expressive infrastructure.

Our contributions are:

- An approach to designing a modular domain-specific compiler using multi-level IR rewriting (Sec. II).
- A stencil language expressed as an MLIR dialect, which encodes the high-level data flow of a stencil program as SSA def-use chains (Sec. IV).
- A set of transformations to tune stencil programs at a high level using simple peephole optimizations instead of conventional loop transformations (Sec. V).
- A separate-compilation scheme for GPUs based on a platform-neutral GPU dialect convertible to vendor-specific GPU code (Sec. VI).
- An evaluation on benchmarks relevant to the FV3 (US) climate model (Sec. VII).

II. Multi-Level IR Rewriting

Multi-level IR rewriting promises to simplify the development of domain-specific compilers by defining a stack of reusable abstractions and by implementing the transformation at the most relevant level. The goal is to minimize the complexity of each level and to reduce the cost of analysis by encoding and preserving transformation validity preconditions directly in the IR.

Identifying pertinent domain-specific abstractions is paramount to the multi-level rewriting. Each new abstraction increases risks of excessive complexity or, on the contrary, of incompleteness where some workloads cannot be represented. We instantiate the Open Earth Compiler using the MLIR infrastructure [20], carefully considering the abstractions it provides and introducing new ones when necessary. Our objective is to facilitate performance extraction from one of the two primary sources: parallelism and data locality, which often require conflicting transformations yielding complex optimization problems [23]. Instead of attacking these problems frontally, we design abstractions so as to extract parallelism and locality information from the domain knowledge, using the following principles.

P1 Transformation-Driven Semantics. The domain abstractions for different levels of our pipeline, e.g., stencils or GPU kernels, should favor transformation-readiness over programmer-friendliness. Our objective is to build a stack of intermediate representations that enable the compiler to reason about domain-specific programs without resorting to complex analyses such as loop extraction [24] or dependence analysis [25]. Each IR in the stack is focused on a specific set of domain transformations and designed to make all necessary information readily available. End-user usability aspects are deliberately deferred to DSL front-ends.

P2 Progressive Lowering. We aim for an effective and streamlined transformation pipeline where programs are progressively lowered [20] from a high-level domain IR to a low-level target IR. The different IR abstractions should be designed to maintain high-level semantic information as long as necessary, such that a potentially complex recovery of high-level concepts can be avoided. An important additional aspect of progressive lowering in larger domain-specific compilers is that abstractions should seamlessly compose with each other to coexist in a single module while the lowering is applied selectively.

P3 Explicit Separation. Given the abstraction composability mandated by the previous principles, it is easier to combine individual pieces of the abstraction than to disentangle a complex representation. Our incarnation of the ubiquitous separation-of-concerns approach relies on the domain-relevant

Level	Concepts	Transformations	Sec.
Stencil	- parallel stencil evaluation - value semantic - explicit data flow - compile-time access offset - compile-time domains	- inlining (+CSE) - unrolling (+CSE)	IV V-A V-B
Standard & Affine & Loop	- multi-dimensional storage - affine index computation - parallel loop	- loop mapping - loop to GPU	V-C
GPU	- host/device code - SIMT parallelism	- GPU outlining - host/device comp.	VI

TABLE I: Domain-specific to device-specific abstractions.

separation being explicit in at least some intermediate abstraction in our stack. In particular, performance-related aspects of the abstractions, such as the degree of parallelism or the memory footprints, should be present in the IR and should be modifiable separately from each other. Similarly, compile- and run-time aspects of the abstraction should be separated. In the longer term, such representations are better amenable to modern search techniques [26]–[28].

The abstractions we use enable progressive lowering (P2) from domain-specific to device-specific concepts providing clear separation (P3) between levels. As listed in Table I, each level makes specific transformations easy to implement (P1). This multi-level representation also helps us separate optimizing transformations from the lowering between the levels that constitute a large portion of DSL compilers.

III. The MLIR Infrastructure

MLIR is a recent production compiler infrastructure that is particularly well-suited for multi-level IR rewriting thanks to its extensibility through dialects and its built-in support for declarative rewrite patterns [20]. The Open Earth Compiler can be thus implemented as a set of MLIR dialects, and transformations as rewrite patterns. Furthermore, we can readily reuse Standard, Loop, Affine and LLVM IR dialects if we design our abstractions so that they compose with these.

Core MLIR concepts include operations, values, types, attributes, (basic) blocks, and regions. An operation is an atomic unit of program description. A value represents data at runtime and is always associated with a type known at compile time. Operations use values (but do not consume them) and define new values. Values can only be defined once, making the IR obey SSA form. A type holds compile-time information about a value, while attributes provide a way to attach compile-time information to operations. A block is a sequence of operations that, together with other blocks, connects to regions. A region is attached to an operation that defines its semantics. Non-trivial control flow is only allowed between an operation and the regions attached to it, and between the entry and exit points of blocks that belong to the same region. Specific operations define the structure of the control flow, for example, the last operation in a block (a terminator) can conditionally or unconditionally transfer the control flow to another block.

Fig. 2 illustrates the syntax for an example operation from our Stencil dialect. The stencil.**apply** operation uses a value

```
%def = stencil.apply (%arg = %use : f64) -> !stencil.view<ijk,f64> {
  // nested region that has one basic block taking %arg as block argument
  %0 = neg %arg : f64
  stencil.return %0 : f64
} to ([0, 0, 0]:[64, 64, 64]) // attribute defining the iteration domain
```

Fig. 2: Example MLIR operation that sets 64x64x64 elements of the defined value **%def** to the negative of the value **%use**.

%use and defines a value **%def**. Types and attributes annotate the operation with compile-time information such as the iteration domain. The nested region consist of a single basic block that implements computation performed by the operation using the basic block argument **%arg**. This hierarchical organization into blocks and regions enables infinite nesting.

There is no fixed set of operations, attributes, or types. Instead, each MLIR user can define their own or reuse those defined by others. Even MLIR's built-in functionality heavily relies on this extensibility. For example, a function is a regular operation with a region containing the function body instead of a concept on its own. Thus stencil computations can be made first-class in a stencil compiler by providing custom types, attributes, and operations. The same holds for control flow operations such as loops or data types such as multi-dimensional arrays. From an infrastructure point of view, custom operations and types are indistinguishable from common scalar operations and types.

A dialect is a set operations, attributes, and types designed to work together. There is no formal or technical restriction on how dialects are structured. Unless prescribed otherwise by the semantics of the operation, a region can contain operations from different dialects, and an operation can reference types and attributes defined by a different dialect. Therefore, new abstractions can be introduced into the MLIR ecosystem as new dialects.

IV. The Stencil Dialect

The Open Earth Compiler operates on weather and climate models. These models integrate partial differential equations forward in time commonly using either finite difference or finite volume discretization. They comprise dozes of stencil programs (Sec. IV-D) consisting of multiple dependent stencil operators (Sec. IV-C) applied across regular or irregular grids. In this work, we consider regular three-dimensional grids that partition the space into cells, each of which with six neighbors. This regularity allows a cell to be addressed via a three-component index.

In stencil programs, optimizing individual stencils is often insufficient. Instead, chains of dependent stencils or entire programs must be optimized [29], e.g., using producer-consumer fusion to obtain maximum performance. We design the Stencil dialect to represent stencil programs consisting of stencil operations connected between them and with input/output data structures through data flow, with optional control flow (Sec. IV-E), following the multi-level IR rewriting principles defined in Sec. II. Our dialect is explicitly decomposed (**P3**) into the high level, where we model the data flow between

```
func @sum(%in : !stencil.field<ijk,f64>, %out : !stencil.field<ijk,f64>) {
  stencil.assert %in ([-4, -4, -4]:[68, 68, 68]) : !stencil.field<ijk,f64>
  stencil.assert %out ([-4, -4, -4]:[68, 68, 68]) : !stencil.field<ijk,f64>
  %0 = stencil.load %in (!stencil.field<ijk,f64>) -> !stencil.view<ijk,f64>
  %1 = stencil.apply (%arg0 = %0 : !stencil.view<ijk,f64>)
    -> !stencil.view<ijk,f64> {
      %2 = stencil.access %arg0[1, 0, 0] : (!stencil.view<ijk,f64>) -> f64
      %3 = stencil.access %arg0[-1, 0, 0] : (!stencil.view<ijk,f64>) -> f64
      %4 = addf %2, %3 : f64
      stencil.return %4 : f64
  }
  stencil.store %1 to %out ([0, 0, 0]:[64, 64, 64]) :
    !stencil.view<ijk,f64> to !stencil.field<ijk,f64>
  return
}
```

Fig. 3: Example stencil program that evaluates a simple stencil on the array **%in** and stores the result to the array **%out**.

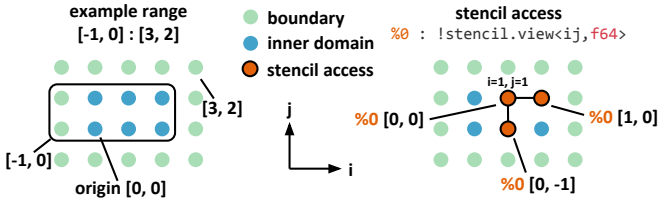

Fig. 4: Example range (left) defined by an inclusive lower and an exclusive upper bound and stencil accesses (right) expressed relative to the current position (i=1, j=1).

operators, and the low level, where we model the parallel execution of individual operators. The former enables flow rerouting transformations where a stencil operator can be seen as a unit (as opposed to lower-level IRs where a stencil is a collection of arithmetic instructions), while the latter supports parallelism exploitation (**P1**). The level separation also participates in progressive lowering (**P2**).

The dialect is not designed as a user-facing DSL, but as a compiler IR that supports transformations (**P1**&**P3**) by (a) keeping the stencil concepts high-level so they can be moved as a unit, (b) imposing no specific execution order so as to expose parallelism, and (c) using value-semantics instead of allocating storage objects to avoid costly buffer analysis.

A. Dialect Overview

The Stencil dialect focuses on concepts specific to stencils and relies on MLIR's Standard dialect to express the actual computation (**P2**). Fig. 3 shows a stencil program that, for every point of a 64x64x64-element domain, adds the left and the right neighbor of the input array **%in** and stores the result to the output array **%out**. The "stencil" prefix identifies the operations and types from this dialect.

The dialect defines two types. A !stencil.**field** is a multi-dimensional array that stores an element for all points of the regular grid. Inputs and outputs of a stencil program have this type. A !stencil.**view** is a multi-dimensional collection of elements for a hyper-rectangular subdomain of the regular grid. A view either points to a subdomain of an input array or keeps the results computed by a stencil operator. Intermediate results of this type have value semantics and are initially not backed by storage. Both types contain single- or double-precision floating-point values (f32 or f64) on a one-, two-, or three-dimensional (i, j, and k enumerate the grid dimensions) domain.

The Stencil dialect also defines six operations. In the example, the stencil.**assert** operations specify the static shape of the arrays. The stencil.**load** operation takes the input array and returns a view that points to the input elements consumed by the subsequent stencil. The stencil.**apply** operation executes the stencil and defines a result view that keeps the results of the computation. Its nested region implements the stencil operator for one point of the iteration domain. The stencil.**access** operations read the input view at a constant offset relative to the current position in the iteration domain, while the stencil.**return** operation sets the output at the current position. In between, we use the Standard dialect to sum the left and the right neighbor elements. The stencil.**store** operation finally stores the result of the computation to the output array. Its range attribute thereby specifies the domain written by the stencil program. Our compiler utilizes the output ranges to automatically infer the access ranges and iteration domains for the entire stencil program (cf. Sec. V-B).

B. Shapes & Domains

The range notation is essential to specify stencil iteration domains and access ranges, especially given that stencils may be accessing inputs with indices that are outside of their computation domain, e.g., on boundaries.

Fig. 4 shows our range notation (left) for a two-dimensional domain. The origin denotes the lower bound of the computation domain and has all coordinates set to zero. Ranges are specified given the absolute coordinates of an inclusive lower bound and an exclusive upper bound separated by a colon.

On GPUs, integer index computations are a significant performance bottleneck. As real-world stencil programs often execute stencils repeatedly for the same problem size, it is desirable for the stencil dialect to support size specialization for just-in-time compilation. It does so by defining storage shapes and iteration domains as numeric compile-time attributes (**P3**).

C. Stencil Operators

A stencil operator performs element-wise computations on all elements of a regular grid except for some constant-width boundary. It accesses the elements of input arrays at constant offsets relative to the coordinates of the output element.

The stencil.**apply** operation contains a region that implements the stencil operator in terms of scalar operations on domain elements. The scalar operations are applied to all domain elements as in a loop nest. Stencil operator inputs and outputs correspond to values used and defined by this operation. Inputs are assumed to not alias, and element-wise computations are assumed to be independent (**P3**).

Individual elements of inputs are accessed using the stencil.**access** operation that reads an element at a constant offset. The lowering of the Stencil dialect later adds the constant access offset to the index of the current iteration (cf. Sec. V-C). Fig. 4 shows the offset computation (right) for a

```
func @prog(%in : !stencil.field<ijk,f64>, %out : !stencil.field<ijk,f64>) {
  /* ... */
  %0 = stencil.load %in : (!stencil.field<ijk,f64>) -> !stencil.view<ijk,f64>
  %1 = stencil.apply (%arg0 = %0 : !stencil.view<ijk,f64>)
    -> !stencil.view<ijk,f64> { /* stencil 1 */ }
  %2 = stencil.apply (%arg0 = %0 : !stencil.view<ijk,f64>,
                      %arg1 = %1 : !stencil.view<ijk,f64>)
    -> !stencil.view<ijk,f64> { /* stencil 2 */ }
  /* ... */
}
```

Fig. 5: Stencil program that evaluates two dependent stencils.

two-dimensional stencil iteration domain. The region of the stencil operator must be terminated by a single stencil.**return** operation that accepts the value of the output element as an argument. Together, the stencil.**access** and stencil.**return** operations specify the memory access pattern of the stencil operator. Both of them are only valid as part of the stencil operator definition.

Real-world stencil programs from the weather and climate domain often implement dozens of dependent stencil operators. A stencil program thus needs additional means to orchestrate them.

D. Stencil Programs

A stencil program executes a sequence of dependent stencil operators. It loads the data from the input arrays, implements the stencil operators inline, and stores the results to the output arrays. The SSA def-use graph of the program thus specifies the high-level data flow between the stencil operators (**P2**). Having both the high-level data flow and the inlined stencil operators in a single function facilitates code transformations across multiple stencil operators, eliminating any need for complex interprocedural analysis (**P1**).

Three additional operations are part of the program definition. The stencil.**assert** operation specifies the index range or an input or output array. A valid stencil program needs to define the index range for all input and output arrays. The stencil.**load** operation returns a view to all input array elements accessed by dependent stencils. Conversely, the stencil.**store** operation stores the output of a stencil operator to the output array elements denoted by its range attribute.

Fig. 5 shows a stencil program that executes two dependent stencils. The stencil.**load** operation returns a view to the **%in** array. The second stencil operator uses the result of the first. In the end, the stencil.**store** operation stores the values computed by the second stencil to the **%out** array.

All stencil program parameters have to be alias-free and are either loaded from or stored to as a unit. Intermediate results are kept in values of type !stencil.**view** that are not initially backed by storage and are thus also alias-free. Given the value semantics of !stencil.**view**, the def-use graph encodes the data dependencies between the stencil operators (**P1&P3**).

E. Control Flow

Real-world stencil applications are not limited to pure data flow semantics. We can use eager execution and just-in-time compilation to handle most of the control-flow at the program

```
%0 = loop.if %flag -> (f64) {
  %1 = stencil.access %arg0[0, 0, 0] : (!stencil.view<ijk,f64>) -> f64
  loop.yield %1 : f64
} else {
  %2 = stencil.access %arg1[0, 0, 0] : (!stencil.view<ijk,f64>) -> f64
  loop.yield %2 : f64
}
stencil.return %0 : f64
```

if %flag then %arg0 else %arg1

Fig. 6: The Loop dialect enables the implementation of control flow inside the stencil operator.

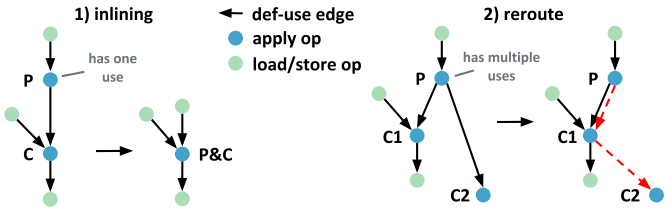

Fig. 7: Two patterns enable the iterative producer-consumer fusion for entire stencil programs. The def-use edges represent the data flow between the stencil operations.

level and resort to MLIR's built-in Loop dialect to implement dynamic control flow inside the stencil operators.

Fig. 6 shows a stencil that, depending on a flag, accesses one of two arguments. The loop.**if** operation conditionally executes either the "then" or the "else" region. In contrast to a regular if-else, the operation returns a result value that is set by the loop.**yield** operations. This representation makes the data flow explicit and maintains a single stencil.**return** operation per stencil. An alternative to the loop.**if** operation, is the **select** operation that chooses a value based on a condition. Supporting the loop.**if** operation requires no adaptation of our compiler (**P2**).

Our choice of the built-in MLIR Loop dialect exemplifies how progressive lowering, explicit separation, and composable abstractions enable reuse of compiler components in the multi-level IR rewriting scheme.

V. Stencil Transformations

We distinguish three categories of transformations that work on the Stencil dialect: 1) performance optimizations, 2) transformations to prepare the lowering, and 3) the actual lowering.

A. Optimizing Transformations

All optimizing transformations implemented for the Stencil dialect operate at a high-level and neither introduce explicit loops nor storage allocations (**P1**).

The **stencil inlining** pass applies fusion on the def-use graph of the stencil program. In particular, we repeatedly apply a stencil specific variant of producer-consumer fusion that replaces all accesses to producer results by inline computation. If the consumer accesses the producer at multiple offsets, we thus perform redundant computation for every point in the iteration domain. Inlining stencils in an arbitrary order may introduce circular dependencies. An input of the consumer may, for example, depend on another stencil that transitively

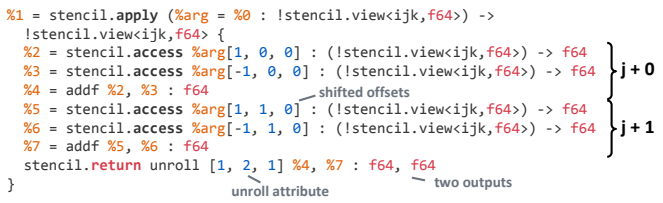

Fig. 8: Unrolling two iterations of the example stencil along the j-dimension.

depends on an output of the producer stencil. Instead of developing an algorithm to fuse the stencils in a valid order, we implement patterns that match and rewrite small subgraphs and use MLIR's greedy rewriter to apply them step-by-step.

Fig. 7 shows our inlining patterns. The inlining pattern matches a producer P and a consumer C if the producer has a single consumer. If the pattern matches, we remove the producer stencil and inline the computation into the consumer. Additionally, we update the argument and result lists of the fused stencils. The reroute pattern matches a producer P and its consumers C1 to CN. If the pattern matches, we route all outputs of the producer through the consumer that executes next. The red (dashed) arrows mark the rerouted data dependencies. The former pattern implements the actual inlining, while the latter pattern prepares an inlining step.

Our inlining implementation introduces redundant computation even if the consumer accesses the same offset multiple times and always inlines the entire producer even if only one of its outputs is accessed. Dead code elimination and common subexpression elimination later clean up the code. These transformations rely on the stencil accesses being side-effect free (the stencil inputs are immutable and do not alias with the outputs). Our compiler currently implements no fusion heuristic and continuous inlining as long as one of the patterns applies.

The **stencil unrolling** pass replicates a stencil operator multiple times to update more than one grid point at once. Fig. 8 shows an unrolled version of our example program.

Unrolling is another example of a classical loop transformation implemented by our high-level dialect. Instead of transforming loops, our implementation annotates the high-level Stencil dialect and directly lowers to unrolled loops. In particular, we only modify the nested region attached to the stencil.**apply** operation but not its interface. Initially, we replicate the stencil computation once for every unrolled loop iteration and adjusts the access offsets. We also adapt the stencil.**return** operation to return the results of all unrolled loop iterations and annotate the unroll factor and dimension using an optional attribute.

The unrolling pass supports all unroll dimensions and unroll factors. Yet, the lowering is currently limited to unroll factors that divide the domain size evenly.

Inlining and unrolling improve the performance of stencil programs. Especially inlining reduces the off-chip data movement at the cost of introducing redundant computation. Unrolling can eliminate parts of the redundant computation since the unrolled stencil operator often evaluates the producer several times at the same offset. Instead of removing the redundant computation ourselves, we run the existing common subexpression elimination pass.

B. Preparing the Lowering

After optimizing the stencil program, we infer all access ranges and iteration domains to prepare the lowering (**P2**).

The **shape inference** pass derives the access ranges for the input arrays and stencil operators of the program. It is necessary since a stencil program only defines the output ranges written by the program. The pass then starts from these output ranges and follows the use-def chains that define the dependencies of the stencil program and transitively extends the access ranges.

Our algorithm walks all operations of the stencil program in reverse order and annotates the access ranges using optional range attributes (Sec. V-C shows the lowering of the annotated example program). We compute these ranges as the minimal bounding box that contains all access extents of operations that consume a value defined by the current operation. If the consumer is a stencil.**load** operation, its access extent is equal to the output range attribute. If the consumer is a stencil.**apply** operation, the access extent is equal to the iteration domain extended by a minimal bounding box that contains all stencil accesses of the consumed values. Once the access range of a stencil.**load** operation is known, we also verify the input array is large enough.

Although the access extent analysis seemingly contradicts the progressive lowering idea (**P2**), it does not aim at recovering information that has been there before. Instead, it automates the error-prone manual access range specification.

The **shape shifting** pass shifts all offsets and ranges of the stencil program to the positive range. This translation enables the lowering of the stencil types to their MLIR counterparts.

Shape inference and shifting enable the lowering and have no performance impact.

C. Lowering to Loops

The stencil lowering applies conversion patterns to translate the individual stencil operations to their MLIR counterparts (**P2**). It is the last domain-specific part of our compilation pipeline, outlined in Fig. 1, that lowers our high-level stencil programs towards executable code.

Even at the Standard dialect level, MLIR provides rather high-level abstractions. The memref is a structured multi-dimensional buffer abstraction. It can have static or dynamic sizes, and an optional layout attribute defines the index computation if the layout diverges from the row-major format. This layout attribute also allows one to define strided hyper-rectangular views into a memory buffer, for example, with offsets and non-unit steps along each of the dimensions. Another example is the loop.**parallel** operation that models a parallel multi-dimensional loop.

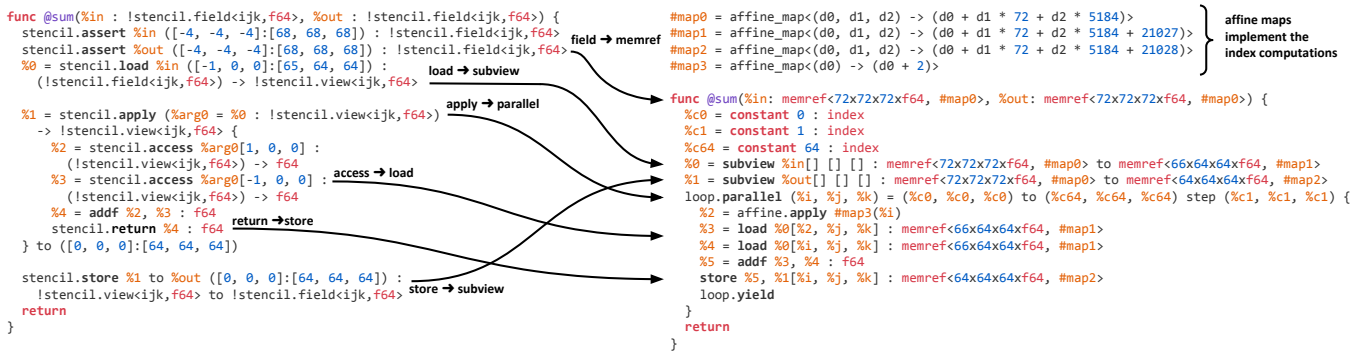

Fig. 9: Conversion of the Stencil dialect to the MLIR Loop+Affine+Standard dialects that further lower to GPU abstractions.

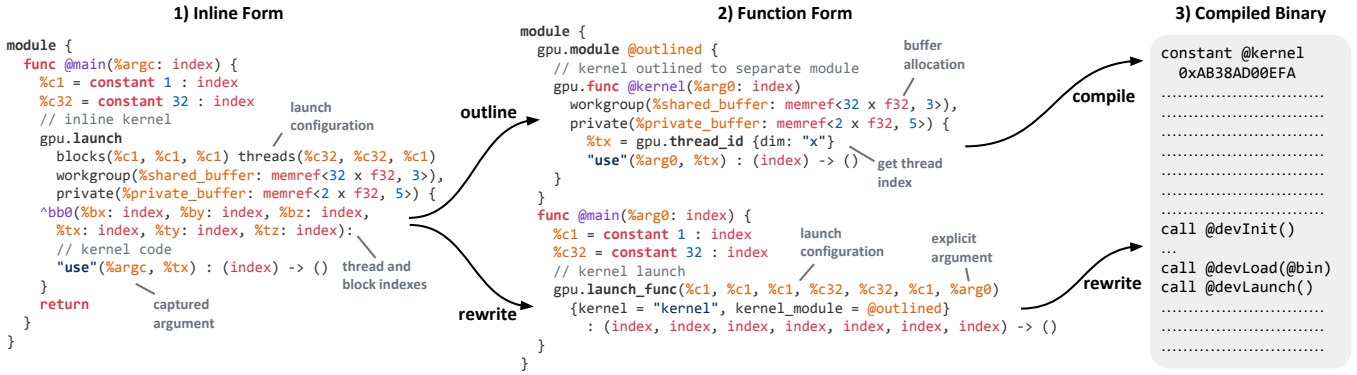

Fig. 10: Lowering of an example kernel: 1) the inline form enables host device code motion and other transformations, 2) the function form isolates the device code in a distinct module that enables device-specific optimizations and separate host/device compilation, and 3) the binary embeds the kernel as constant data.

Fig. 9 illustrates the lowering from the stencil dialect level to the MLIR Standard dialect level for the example introduced in Sec. IV-A. We define six conversion patterns that introduce explicit loops, index computations, memory accesses, and temporary storage. After this lowering, detecting stencil operators or access offsets requires analysis. Implementing domain-specific transformations consequently becomes harder. In turn, by introducing loops and temporary storage, we settle to program execution order but still maintain the parallel semantics needed for the subsequent GPU lowering (**P2**).

VI. The GPU Dialect

Since GPUs remain a platform of choice to achieve high performance, we construct our multi-level compiler to target these devices. We designed following the principles defined in Sec. II and implemented the GPU dialect for MLIR to this end with the goal of abstracting the GPU execution model in a vendor-independent way. In particular, it generalizes MLIR's NVVM, ROCm, and SPIR-V representations and thus separates unified platform-independent device mapping (**P1**) from platform-specific code generation (**P3**). The GPU dialect is not intended as a generic SIMT execution model (**P3**), nor as a raising target from lower-level abstractions (**P2**). The dialect exposes a set of GPU-specific concepts: hierarchical thread structure (blocks, threads, warps); synchronization through barriers; memory hierarchy (global, shared, private, constant memory); standard computational primitives such as parallel reductions. It is also designed to support separate host/device compilation in a single module (**P3**). The latter is made possible by MLIR modules recursively containing other modules that can be processed differently.

Fig. 10 shows the two forms of a kernel launch during the GPU lowering. The inline form uses the gpu.**launch** operation to define the kernel inline. A nested region implements the kernel, and basic block arguments provide access to the thread and block identifiers. Explicit parameter handling is not needed since the values defined outside of the nested region remain visible. The function form uses the gpu.**func** operation to implement the kernel as a separate function in a dedicated module launched by the gpu.**launch_func** operation that represents the kernel invocation. Special operations provide access to the thread and block identifiers. All non-constant kernel arguments are passed in explicitly, while the constants are propagated into the kernel functions. Both the inline and the function form accept a GPU grid configuration and support the declarative allocation of buffers in the different levels of the GPU memory hierarchy. The kernel code expresses the computation for a single thread, following the SIMT model, and specialized mechanisms provide access to the thread and

block identifiers. Thereby GPU-specific primitives such as barrier synchronization, shuffles, and ballots are only available inside a kernel launch.

Additionally, Fig. 10 illustrates the main steps of the GPU lowering, starting from the inline form (left), through the function form (middle), down to the compiled binary (right). A parallel loop nest can be converted in-place to the inline form, using loop bounds as GPU grid configuration. After the conversion, we apply common subexpresison and dead code elimination, canonicalization and propagate constants inside the GPU kernel to minimize host/device memory traffic. Common SSA-based transformations apply seamlessly across the host/device boundary thanks to the kernel being inlined with no visibility restrictions. The kernel is then outlined into a separate function in a dedicated GPU module. Functions called by the kernel are copied into the module, and values defined outside the kernel are passed in as function arguments. This results in kernels living in a separate module to enable the separate host/device optimization and compilation. The kernel bodies are no longer visible to intra-procedural optimizations on the host code. The GPU module is finally converted through a dedicated dialect to a platform-specific representation (e.g., PTX), and using the vendor compiler (e.g., ptxas) compiles further down to a binary. The resulting binary is embedded as a global constant into the original module. This approach enables, e.g., multi-versioning to support multiple architectures or kernel specialization for different-sized workloads. The original module extended with the binary constants then becomes a regular host module, that can be optimized, compiled, and executed. The kernel invocations thereby lower into calls to the device driver library or runtime environment.

VII. Evaluation

We evaluate the Open Earth Compiler on real-world stencil programs derived from the most performance critical parts of the FV3 climate model and compare its performance to state-of-the-art code generation techniques.

A. Experimental Setup

We run our experiments on an Nvidia Tesla V100-SXM2 that has a memory bandwidth of 900 GB/s. We use the CUDA toolkit 10.1 with driver version 435.21 and run our experiment for two domain sizes 128x128x60 (small) and 256x256x60 (large) for single-precision (f32) and double-precision (f64) floating-point numbers. For all benchmarks, we report the median runtime of 100 measurements, and red error bars show the quartile runtime to quantify the measurement error. We do not time the first kernel execution due to setup overheads such as host to device data copies. Additionally, we use the nvprof profiler to collect memory bandwidths and compute throughputs. Finally, we ensure correctness by comparing the outputs of all optimized kernel variants to naive C versions and ensure the results are within a relative error of 10^{-5} for single-precision (f32) and 10^{-10} for double-precision (f64) floating-point numbers.

Name	Dims	Apply Ops	Inputs/ Outputs	Arith. Ops	Access Ops	Control Flow
p_grad_c	3	3	7 / 2	24	25	-
nh_p_grad	3	5	8 / 2	47	48	-
uvbke	2	2	4 / 2	12	12	-
fvtp2d_qi	2	5	5 / 2	27	23	if
fvtp2d_qj	2	8	6 / 3	49	39	if
fvtp2d_flux	2	5	7 / 2	28	22	if

TABLE II: Characteristics of our benchmarks.

B. Benchmark Kernels

We evaluate our compiler for a set of representative benchmarks derived from the FV3 [30] dynamical core. It is part of the CM4 and GEOS-5 global climate models and the global weather prediction system of the US National Weather Service. The dominant algorithmic motif of FV3 is stencil computations on regular grids. It implements dozens of stencil operators to perform the numerical forward integration in time. Due to the explicit time integration, most stencils are purely horizontal with bounded domains of dependence. Some use the Thomas algorithm to perform implicit integration in the vertical direction.

All our benchmarks are part of the explicit integration. The fvtp2d kernels implement a monotone two-dimensional finite volume advection operator, the p_grad_c and nh_p_grad kernels compute the three-dimensional pressure gradient, and the uvbke kernel is a preprocessing step for the kinetic energy computation.

Each benchmark executes an entire stencil program consisting of multiple stencil operators being applied on the three-dimensional domain. The stencil operators have different dimensionality (from one- to three-dimensional), have different width (two- to five-point), and some of them contain dynamic control flow. Table II list core characteristics of our benchmarks such as the dimensionality of the stencil access patterns or the number of stencil operators, input/output arrays, arithmetic operations, and stencil.**access** operations. We observe that all kernels have a low arithmetic intensity (arithmetic operations per memory access), which explains why our compiler focuses on transformations to increase the data-locality.

C. Effectiveness of our Code Transformations

We first evaluate the effectiveness of the code transformations discussed in Sec. V-A. In total, we compare different optimization levels: 1) original, 2) inline, 3) inline+unroll(2), and 4) inline+unroll(4). Optimization level one applies no optimizing transformations (cf. Sec. V-A). Starting from optimization level two we apply stencil inlining, and the optimization levels three and four additionally perform stencil unrolling by factor two and four, respectively.

Fig. 11 compares the runtime for all benchmarks at different optimization levels. We show the speedups for the large problem size using f32 and f64 arithmetic. We observe significant speedups for stencil inlining independent of the benchmark. In comparison, stencil unrolling has a smaller

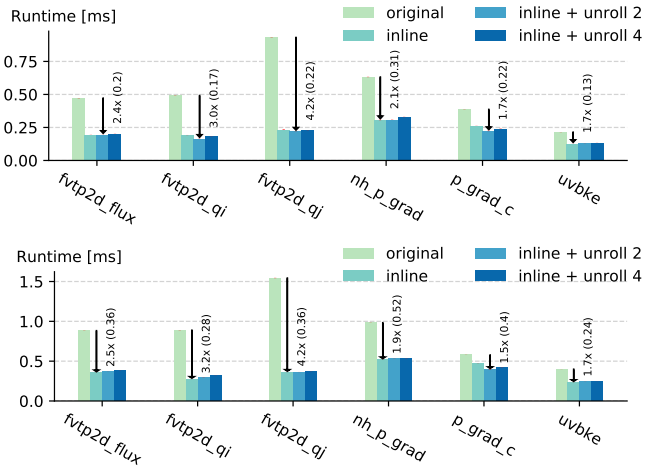

Fig. 11: Runtimes at different optimization levels for f32 (top) and f64 (bottom) floating-point values for 256x256x60.

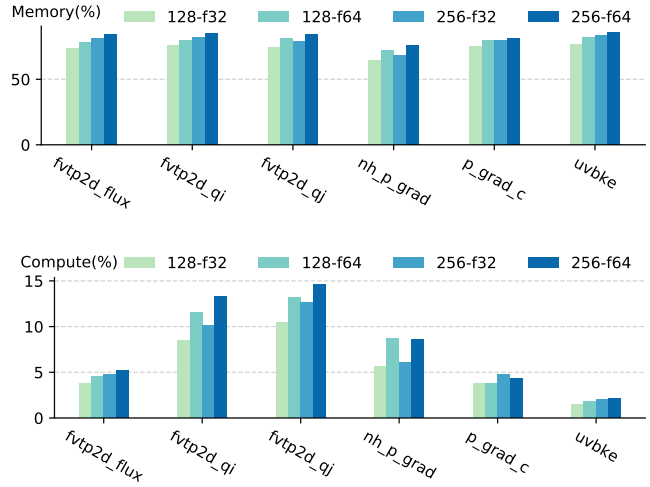

Fig. 12: Utilization of the peak compute throughput (top) and the peak memory bandwidth (bottom) for the best performing kernel variants in percent (these measurements are profiling results collected using nvprof).

effect and sometimes is even detrimental to performance. In the plot, we annotate the speedup and the runtime of the best performing version.

We expect stencil inlining to have such a strong effect on performance, since fusing the stencil operators significantly reduces the required memory bandwidth. After fusing all stencil operators, we do not need to store and load any temporary buffer, and all inputs of the stencil program are only loaded once. At the same time, stencil inlining also introduces redundant computation. However, due to the low arithmetic intensity, the bandwidth reduction overcompensates the additional computation. In contrast, stencil unrolling removes redundant computation at the cost of higher register pressure and reduced parallelism. As a consequence, we do not expect all stencil programs to benefit from stencil unrolling.

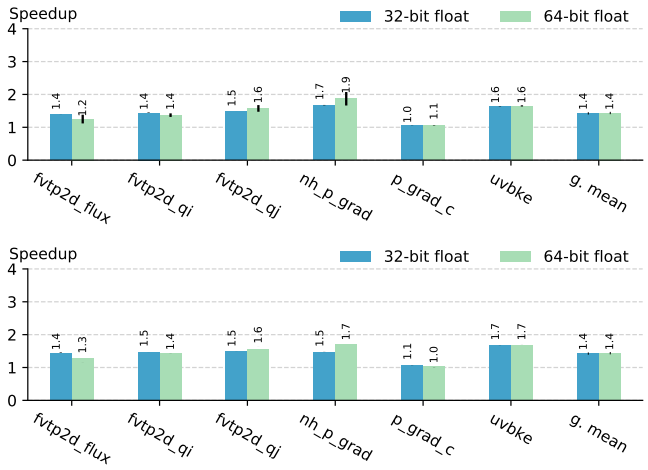

Fig. 13: Speedup of our compiler over Dawn for 128x128x60 (top) and 256x256x60 (bottom).

Instead, the effectiveness of the optimization depends on the complexity of the stencil program (register pressure) and the amount of redundant computation introduced due to stencil inlining.

Fig. 12 illustrates the memory bandwidths and compute throughputs achieved by the best performing kernel versions. We observe very high memory bandwidth utilization and low compute utilization. These results demonstrate the importance of aggressive stencil inlining and confirm the redundant computation is not critical.

In summary, we show that our code transformations yield significant speedups. Selecting the optimal unroll factor or finding good fusion choices is not the scope of this work. We thus employ empirical tuning to find the best unroll factor and fuse all stencil operators (optimizing larger stencil programs will require a fusion heuristic).

D. Comparison to the Dawn Compiler

We now compare the runtime of the kernels optimized by our compiler to Dawn [31] generated CUDA [32] implementations. Dawn is a research compiler that lowers high-level stencil programs to efficient CUDA code. It implements overlapped tiling [33] using shared memory and stream data [34] in registers along the k-dimension. This execution model limits the redundant computation to the tile boundaries. In comparison, our stencil inlining plus unrolling performs additional redundant computation at the thread level but requires no thread synchronization during the kernel execution.

Fig. 13 compares for all benchmarks the best performing variant generated by the Open Earth Compiler to their Dawn counterparts. We outperform Dawn and measure a geometric mean speedup of 1.4x.

We attribute the performance of our compiler to the simple execution model. It limits the data movement by fusing all stencil operators and storing temporaries in registers, and it avoids parallelization overheads by performing no thread

synchronizations during the kernel execution. Its only disadvantage is the substantial redundant computation, which due to the low arithmetic intensity of our kernels shown in Fig. 12, is less critical. Recomputing instead of synchronizing, thus, turns out to be beneficial.

Our compiler, despite its simplicity, outperforms Dawn on raw stencil programs. The results demonstrate the quality of our code generation and the potential of stencil inlining plus unrolling compared to overlapped tiling and streaming, the standard solution in the field.

VIII. Related Work

Accelerated systems made programming model innovations inevitable. Kokkos [35] and Raja [36] are C++ performance portability layers. PENCIL [37] and Polly-ACC [21] automate the accelerator mapping using the polyhedral model. DaCe [38] allows performance engineers to select and develop target-specific transformations. All approaches are generic and, for the same level of performance and automation, solve a more complex problem than a domain-specific compiler.

Machine learning today drives the development of domain-specific compilers [2], [8]. The development of stencil compilers started even earlier: Halide [7] and Polymage [1] tune image processing pipelines, Pochoir [6] implements cache-oblivious tiling, SDSLc [3] supports many targets (SIMD, GPU, and FPGA), Panda [5] supports distributed memory, and YASK [4] specifically targets Intel processors. Lift [39] has also been shown effective for stencil codes. The variety of different solutions demonstrates the importance of a shared compiler infrastructure.

Multiple projects work on solutions for weather and climate. The CLIMA [40] effort develops a novel earth system model using the Julia language. The LFRic [41] climate modeling system relies on the Python-based PSyclone compiler. Stella [9] and GridTools [42] use C++ template metaprogramming to support CPU and GPU systems. CLAW [43] and Hybrid Fortran [44] extend Fortran to achieve performance portability. Despite their heterogeneity, all of these approaches could benefit from a shared compiler infrastructure.

Several frameworks support the development of domain-specific compilers. AnyDSL [45] supports partial evaluation using minimal annotations in the Impala front end language. Lightweight modular staging [46] is a technique that uses Scala's type system to transform codes before their execution. It forms the basis of the Delite [16] compiler framework. Lua script similarly supports staging via the Terra [47] low-level language. Lift [48] finally combines a functional language and rewrite rules to generate performance portable code. MLIR [20] is the only full-fledged compiler infrastructure among these contenders, not limited in terms of optimizations, and not tied to a particular front end language.

Stencil optimizations for GPU targets are a well-researched topic. Tiling [33], [49]–[53] and fusion [29], [54], [55] are the core optimizations for bandwidth-limited low-order stencils as they appear in weather and climate. Other works optimize the resource utilization [34], [56] or discuss the optimization of high-order stencils [3], [57]. Our compiler implements a variant of overlapped tiling [33] that introduces redundant computation for every thread.

IX. Conclusion

We introduced multi-level IR rewriting, an approach to building reusable components for domain-specific compilers. This approach is illustrated through the design and implementation of the Open Earth Compiler, which provides a high-performance compilation flow for weather and climate modeling. We demonstrated that thanks to multi-level IR rewriting, a small yet self-consistent set of high-level operations specifically designed for stencil computations is sufficient to achieve better performance than state-of-the-art DSL compilers. Contrary to the latter, the Open Earth Compiler relies on existing and new reusable compiler abstractions, including the GPU kernel abstraction we introduced, by decoupling domain-specific and target-specific code transformations. Our evaluation of six stencil programs relevant to existing climate models, demonstrates that the Open Earth Compiler generates code that is up to 1.9x faster than the state-of-the-art. We suggest that multi-level IR rewriting and the associated design principles is a promising approach to rapidly design and deploy domain-specific compilers that can take advantage of reusable components of the MLIR ecosystem.

REFERENCES

[1] R. T. Mullapudi, V. Vasista, and U. Bondhugula, "PolyMage: Automatic optimization for image processing pipelines," in Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ser. ASPLOS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 429–443. [Online]. Available: https://doi.org/10.1145/2694344.2694364

[2] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: An automated end-to-end optimizing compiler for deep learning," in 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). Carlsbad, CA: USENIX Association, Oct. 2018, pp. 578–594. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/chen

[3] P. Rawat, M. Kong, T. Henretty, J. Holewinski, K. Stock, L.-N. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan, "SDSLc: A multi-target domain-specific compiler for stencil computations," in Proceedings of the 5th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing, ser. WOLFHPC '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2830018.2830025

[4] C. Yount, J. Tobin, A. Breuer, and A. Duran, "YASK—yet another stencil kernel: A framework for hpc stencil code-generation and tuning," in 2016 Sixth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC), 2016, pp. 30–39.

[5] M. Sourouri, S. B. Baden, and X. Cai, "Panda: A compiler framework for concurrent CPU+GPU execution of 3d stencil computations on GPU-accelerated supercomputers," Int. J. Parallel Program., vol. 45, no. 3, p. 711–729, Jun. 2017. [Online]. Available: https://doi.org/10.1007/s10766-016-0454-1

[6] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson, "The pochoir stencil compiler," in Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures, ser. SPAA '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 117–128. [Online]. Available: https://doi.org/10.1145/1989493.1989508

[7] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, ser. PLDI '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 519–530. [Online]. Available: https://doi.org/10.1145/2491956.2462176

[8] C. Leary and T. Wang, "XLA: Tensorflow, compiled," TensorFlow Dev Summit, 2017.

[9] T. Gysi, C. Osuna, O. Fuhrer, M. Bianco, and T. C. Schulthess, "STELLA: A domain-specific tool for structured grid methods in weather and climate models," in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, ser. SC '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2807591.2807627

[10] O. Fuhrer, C. Osuna, X. Lapillonne, T. Gysi, B. Cumming, M. Bianco, A. Arteaga, and T. Schulthess, "Towards a performance portable, architecture agnostic implementation strategy for weather and climate models," Supercomputing Frontiers and Innovations, vol. 1, no. 1, 2014. [Online]. Available: https://superfri.org/superfri/article/view/17

[11] D. Quinlan, "ROSE: Compiler support for object-oriented frameworks," Parallel Processing Letters, vol. 10, no. 02n03, pp. 215–226, 2000.

[12] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in International Symposium on Code Generation and Optimization, 2004. CGO 2004. IEEE, 2004, pp. 75–86.

[13] G. A. McMechan, "Migration by Extrapolation of Time-Dependent Boundary VALUES," Geophysical Prospecting, vol. 31, pp. 413–420, Jun. 1983.

[14] L. M. Harris and S.-J. Lin, "A two-way nested global-regional dynamical core on the cubed-sphere grid," Monthly Weather Review, vol. 141, no. 1, pp. 283–306, 2013. [Online]. Available: https://doi.org/10.1175/MWR-D-11-00201.1

[15] M. Baldauf, A. Seifert, J. Förstner, D. Majewski, M. Raschendorfer, and T. Reinhardt, "Operational convective-scale numerical weather prediction with the COSMO model: Description and sensitivities," Monthly Weather Review, vol. 139, no. 12, pp. 3887–3905, 2011.

[16] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "Delite: A compiler architecture for performance-oriented embedded domain-specific languages," ACM Trans. Embed. Comput. Syst., vol. 13, no. 4s, Apr. 2014. [Online]. Available: https://doi.org/10.1145/2584665

[17] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Global value numbers and redundant computations," in Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, 1988, pp. 12–27.

[18] S. S. Muchnick, Advanced Compiler Design and Implementation. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998.

[19] W. M. McKeeman, "Peephole optimization," Communications of the ACM, vol. 8, no. 7, pp. 443–444, 1965.

[20] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: A compiler infrastructure for the end of moore's law," 2020.

[21] T. Grosser and T. Hoefler, "Polly-ACC transparent compilation to heterogeneous hardware," in Proceedings of the 2016 International Conference on Supercomputing, ser. ICS '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2925426.2926286

[22] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for CUDA," ACM Trans. Archit. Code Optim., vol. 9, no. 4, Jan. 2013. [Online]. Available: https://doi.org/10.1145/2400682.2400713

[23] O. Zinenko, S. Verdoolaege, C. Reddy, J. Shirako, T. Grosser, V. Sarkar, and A. Cohen, "Modeling the conflicting demands of parallelism and temporal/spatial locality in affine scheduling," in Proceedings of the 27th International Conference on Compiler Construction, ser. CC 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 3–13. [Online]. Available: https://doi.org/10.1145/3178372.3179507

[24] T. Grosser, A. Groesslinger, and C. Lengauer, "Polly—performing polyhedral optimizations on a low-level intermediate representation," Parallel Processing Letters, vol. 22, no. 04, p. 1250010, 2012.

[25] N. Vasilache, C. Bastoul, A. Cohen, and S. Girbal, "Violated dependence analysis," in Proceedings of the 20th annual international conference on Supercomputing, 2006, pp. 335–344.

[26] U. Beaugnon, A. Pouille, M. Pouzet, J. Pienaar, and A. Cohen, "Optimization space pruning without regrets," in Proceedings of the 26th International Conference on Compiler Construction, 2017, pp. 34–44.

[27] A. Adams, K. Ma, L. Anderson, R. Baghdadi, T.-M. Li, M. Gharbi, B. Steiner, S. Johnson, K. Fatahalian, F. Durand, and J. Ragan-Kelley, "Learning to optimize halide with tree search and random programs," ACM Trans. Graph., vol. 38, no. 4, Jul. 2019. [Online]. Available: https://doi.org/10.1145/3306346.3322967

[28] T. Gysi, T. Grosser, and T. Hoefler, "Absinthe: Learning an analytical performance model to fuse and tile stencil codes in one shot," in 2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT), 2019, pp. 370–382.

[29] T. Gysi, T. Grosser, and T. Hoefler, "MODESTO: Data-centric analytic optimization of complex stencil programs on heterogeneous architectures," in Proceedings of the 29th ACM on International Conference on Supercomputing, ser. ICS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 177–186. [Online]. Available: https://doi.org/10.1145/2751205.2751223

[30] "FV3: Finite-volume cubed-sphere dynamical core," https://www.gfdl.noaa.gov/fv3/, 2020.

[31] "Dawn," https://github.com/MeteoSwiss-APN/dawn, 2020.

[32] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," Queue, vol. 6, no. 2, p. 40–53, Mar. 2008. [Online]. Available: https://doi.org/10.1145/1365490.1365500

[33] J. Holewinski, L.-N. Pouchet, and P. Sadayappan, "High-performance code generation for stencil computations on GPU architectures," in Proceedings of the 26th ACM International Conference on Supercomputing, ser. ICS '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 311–320. [Online]. Available: https://doi.org/10.1145/2304576.2304619

[34] P. S. Rawat, C. Hong, M. Ravishankar, V. Grover, L.-N. Pouchet, and P. Sadayappan, "Effective resource management for enhancing performance of 2d and 3d stencils on GPUs," in Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit, ser. GPGPU '16. New York, NY, USA: Association

for Computing Machinery, 2016, p. 92–102. [Online]. Available: https://doi.org/10.1145/2884045.2884047

[35] H. C. Edwards and C. R. Trott, "Kokkos: Enabling performance portability across manycore architectures," in 2013 Extreme Scaling Workshop (xsw 2013), 2013, pp. 18–24.

[36] "RAJA," https://github.com/LLNL/RAJA, 2020.

[37] R. Baghdadi, U. Beaugnon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, A. Betts, A. F. Donaldson, J. Ketema, J. Absar, S. v. Haastregt, A. Kravets, A. Lokhmotov, R. David, and E. Hajiyev, "PENCIL: A platform-neutral compute intermediate language for accelerator programming," in 2015 International Conference on Parallel Architecture and Compilation (PACT), 2015, pp. 138–149.

[38] A. N. Ziogas, T. Ben-Nun, G. I. Fernández, T. Schneider, M. Luisier, and T. Hoefler, "A data-centric approach to extreme-scale ab initio dissipative quantum transport simulations," in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3357156

[39] B. Hagedorn, L. Stoltzfus, M. Steuwer, S. Gorlatch, and C. Dubach, "High performance stencil code generation with Lift," in Proceedings of the 2018 International Symposium on Code Generation and Optimization, ser. CGO 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 100–112. [Online]. Available: https://doi.org/10.1145/3168824

[40] "CLIMA," https://github.com/climate-machine/CLIMA/, 2020.

[41] S. Adams, R. Ford, M. Hambley, J. Hobson, I. Kavčič, C. Maynard, T. Melvin, E. Müller, S. Mullerworth, A. Porter, M. Rezny, B. Shipway, and R. Wong, "LFRic: Meeting the challenges of scalability and performance portability in weather and climate models," Journal of Parallel and Distributed Computing, vol. 132, pp. 383 – 396, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731518305306

[42] "GridTools," https://github.com/GridTools/gridtools, 2020.

[43] V. Clement, S. Ferrachat, O. Fuhrer, X. Lapillonne, C. E. Osuna, R. Pincus, J. Rood, and W. Sawyer, "The CLAW DSL: Abstractions for performance portable weather and climate models," in Proceedings of the Platform for Advanced Scientific Computing Conference, ser. PASC '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3218176.3218226

[44] M. Müller and T. Aoki, "Hybrid Fortran: High productivity GPU porting framework applied to japanese weather prediction model," in Accelerator Programming Using Directives, S. Chandrasekaran and G. Juckeland, Eds. Cham: Springer International Publishing, 2018, pp. 20–41.

[45] R. Leißa, K. Boesche, S. Hack, A. Pérard-Gayot, R. Membarth, P. Slusallek, A. Müller, and B. Schmidt, "AnyDSL: A partial evaluation framework for programming high-performance libraries," Proc. ACM Program. Lang., vol. 2, no. OOPSLA, Oct. 2018. [Online]. Available: https://doi.org/10.1145/3276489

[46] T. Rompf and M. Odersky, "Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs," in Proceedings of the Ninth International Conference on Generative Programming and Component Engineering, ser. GPCE '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 127–136. [Online]. Available: https://doi.org/10.1145/1868294.1868314

[47] Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan, and J. Vitek, "Terra: A multi-stage language for high-performance computing," in Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, ser. PLDI '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 105–116. [Online]. Available: https://doi.org/10.1145/2491956.2462166

[48] M. Steuwer, T. Remmelg, and C. Dubach, "LIFT: A functional data-parallel ir for high-performance GPU code generation," in 2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), 2017, pp. 74–85.

[49] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey, "3.5-D blocking optimization for stencil computations on modern CPUs and GPUs," in SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, 2010, pp. 1–13.

[50] N. Maruyama and T. Aoki, "Optimizing stencil computations for NVIDIA kepler GPUs," in Proceedings of the 1st international workshop on high-performance stencil computations, Vienna, 2014, pp. 89–95.

[51] T. Grosser, A. Cohen, J. Holewinski, P. Sadayappan, and S. Verdoolaege, "Hybrid hexagonal/classical tiling for GPUs," in Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization, ser. CGO '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 66–75. [Online]. Available: https://doi.org/10.1145/2544137.2544160

[52] T. Grosser, A. Cohen, P. H. J. Kelly, J. Ramanujam, P. Sadayappan, and S. Verdoolaege, "Split tiling for GPUs: Automatic parallelization using trapezoidal tiles," in Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units, ser. GPGPU-6. New York, NY, USA: Association for Computing Machinery, 2013, p. 24–31. [Online]. Available: https://doi.org/10.1145/2458523.2458526

[53] K. Matsumura, H. R. Zohouri, M. Wahib, T. Endo, and S. Matsuoka, "AN5D: Automated stencil framework for high-degree temporal blocking on GPUs," in Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization, ser. CGO 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 199–211. [Online]. Available: https://doi.org/10.1145/3368826.3377904

[54] P. S. Rawat, M. Vaidya, A. Sukumaran-Rajam, M. Ravishankar, V. Grover, A. Rountev, L. Pouchet, and P. Sadayappan, "Domain-specific optimization and generation of high-performance GPU code for stencil computations," Proceedings of the IEEE, vol. 106, no. 11, pp. 1902–1920, 2018.

[55] M. Wahib and N. Maruyama, "Scalable kernel fusion for memory-bound GPU applications," in SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2014, pp. 191–202.

[56] P. S. Rawat, F. Rastello, A. Sukumaran-Rajam, L.-N. Pouchet, A. Rountev, and P. Sadayappan, "Register optimizations for stencils on GPUs," in Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ser. PPoPP '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 168–182. [Online]. Available: https://doi.org/10.1145/3178487.3178500

[57] T. Zhao, P. Basu, S. Williams, M. Hall, and H. Johansen, "Exploiting reuse and vectorization in blocked stencil computations on CPUs and GPUs," in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356210