Domain-Specific Multi-Level IR Rewriting for GPU: The Open Earth Compiler for GPU-accelerated Climate Simulation

TOBIAS GYSI and CHRISTOPH MÜLLER, ETH Zurich, Switzerland
OLEKSANDR ZINENKO, Google, France
STEPHAN HERHUT, Google, Germany
EDDIE DAVIS, TOBIAS WICKY, and OLIVER FUHRER, Vulcan Inc, USA
TORSTEN HOEFLER, ETH Zurich, Switzerland
TOBIAS GROSSER, University of Edinburgh, UK

Most compilers have a single core intermediate representation (IR) (e.g., LLVM) sometimes complemented with vaguely defined IR-like data structures. This IR is commonly 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.

CCS Concepts: • **Software and its engineering** → **Compilers**; **Domain specific languages**; • **Applied computing** → *Earth and atmospheric sciences;*

Additional Key Words and Phrases: Weather and climate, stencil computations, intermediate representations

Tobias Gysi also with Google.
Tobias Grosser also with ETH Zurich.
The work done at ETH Zurich has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 programme (grant agreement DAPP, No. 678880), the Swiss National Science Foundation under the Ambizione programme (grant PZ00P2168016), and ARM Holdings plc and Xilinx Inc in the context of Polly Labs.
Authors' addresses: T. Gysi, C. Müller, and T. Hoefler, ETH Zurich, Switzerland; emails: gysit@google.com, {christoph.mueller, htor}@inf.ethz.ch; O. Zinenko, Google, France; email: zinenko@google.com; S. Herhut, Google, Germany; email: herhut@google.com; E. Davis, T. Wicky, and O. Fuhrer, Vulcan Inc, USA; emails: {EddieD, TobiasW, OliverF}@vulcan.com; T. Grosser, University of Edinburgh, UK; email: tobias.grosser@ed.ac.uk.

1544-3566/2021/09-ART51 $15.00
https://doi.org/10.1145/3469030

ACM Reference format:
Tobias Gysi, Christoph Müller, Oleksandr Zinenko, Stephan Herhut, Eddie Davis, Tobias Wicky, Oliver Fuhrer, Torsten Hoefler, and Tobias Grosser. 2021. Domain-Specific Multi-Level IR Rewriting for GPU: The Open Earth Compiler for GPU-accelerated Climate Simulation. *ACM Trans. Arch. Code Optim.* 18, 4, Article 51 (September 2021), 23 pages.
https://doi.org/10.1145/3469030

1 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 [13, 38, 45, 51, 54, 58]. For example, Halide [44] automated the generation of high-performance code for image processing, XLA [31] exploited domain-specific compilation to accelerate deep learning, and Stella [24] was the first to move the weather and climate simulation to GPUs leading to 2.9× speedup [17].

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 [43] or production-oriented such as LLVM [29], 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?"

We take a practical case study-based approach to address this question by designing and completing a domain-specific compiler for weather and climate models [10, 26]. The principal computational patterns found in these codes are stencils and the Thomas algorithm. While the stencils are similar to the ones found in image processing [44] or seismic imaging [36], the Thomas algorithm demands specific control flow extensions. Besides, the bandwidth-limited low-order weather and climate stencils often require adapted optimization strategies [24]. However, 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 [53].

We propose to design domain-specific compilers using *multi-level IR rewriting*. This approach is a combination of (a) **intermediate representations (IR)** based on **Static Single Assignment form (SSA)** [50], (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 [37]. 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 [35], and to introduce reusable lower-level abstractions. The recently introduced MLIR compiler infrastructure [30] allows us to instantiate production-quality compiler IRs that follow the practice-proven IR design principles developed in LLVM [29] over the past 15 years.

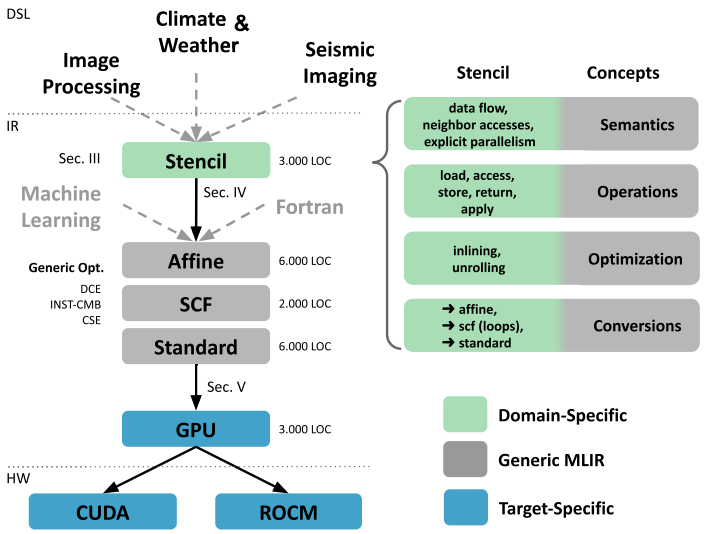

Fig. 1. The Open Earth Compiler.

The Open Earth Compiler[1] we implemented (Figure 1) is one of the first end-to-end compilation flows leveraging 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 (Section 4) designed as a target for various user-facing DSLs as well as a data structure for domain-specific transformations such as stencil inlining (Section 5.1). Stencils are then lowered through a series of IRs featuring index operations (Affine), structured control flow (SCF), and arithmetic operations (Standard), all of which are readily available in MLIR [30], 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 (Section 6) 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, 56]. 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.

[1] https://github.com/spcl/open-earth-compiler/.

Our contributions are:

- An approach to designing a modular domain-specific compiler using multi-level IR rewriting (Section 2).
- A stencil language expressed as an MLIR dialect, which encodes the high-level data flow of a stencil program as SSA def-use chains (Section 4).
- A set of transformations to tune stencil programs at a high level using simple peephole optimizations instead of conventional loop transformations (Section 5).
- A platform-neutral GPU dialect abstracting the vendor-specific GPU code generation (Section 6).
- An evaluation on benchmarks relevant to real climate models: COSMO (Europe) and FV3 (US) (Section 7).
- A comparison to the industry-leading stencil compiler and to state-of-the-art solutions for weather and climate (Section 7.4).

2 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 [30], 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 [60]. Instead of attacking these problems frontally, we design abstractions 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 [20] or dependence analysis [55]. 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 [30] 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 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

Table 1. Domain-specific to Device-specific Abstractions

Level	Concepts	Transformations	Sec.
Stencil new	- parallel stencil evaluation - value semantic - explicit data flow - compile-time access offset - compile-time domains	- inlining (+CSE) - unrolling (+CSE)	4 5.1 5.2
Standard & Affine & SCF	- multi-dimensional storage - affine index computation - parallel loop	- loop mapping - loop to GPU	5.3
GPU new	- host/device code - SIMT parallelism	- GPU outlining - host/device comp.	6

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 more amenable to modern search techniques [7, 11, 23].

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 1, 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 domain-specific compilers.

3 THE MLIR INFRASTRUCTURE

MLIR is a recent production compiler infrastructure that is particularly well-suited for multi-level IR rewriting due to its extensibility through *dialects* and its built-in support for declarative rewrite patterns [30]. We thus implement the Open Earth Compiler as a set of MLIR dialects and rewrite patterns designed to compose with the existing Standard, Structured Control Flow, Affine, and LLVM IR dialects.

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. There is no fixed set of operations, attributes, or types. Instead, users define their own or reuse those defined by others.

A *dialect* is a set of 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. As a result, we can extend the MLIR ecosystem by adding custom dialects. In our compiler, we use this extensibility to make stencil computations first-class by providing custom types, attributes, and operations.

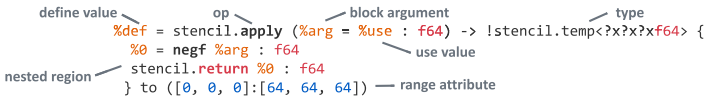

Fig. 2. Example MLIR operation that sets 64 × 64 × 64 elements of the defined value **%def** to the negative of the value **%use**.

The same holds for control flow operations such as loops or data types such as multi-dimensional arrays.

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

4 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 dozens of *stencil programs* (Section 4.4) consisting of multiple dependent *stencil operators* (Section 4.3) 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 [22], 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 (Section 4.5), following the multi-level IR rewriting principles defined in Section 2. Our dialect is explicitly decomposed (**P3**) into the high level, where we model the data flow between 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 (cf. Section 7.5), 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 to expose parallelism, and (c) using value-semantics instead of allocating storage objects to avoid costly buffer analysis.

4.1 Dialect Overview

The Stencil dialect focuses on concepts specific to stencils and relies on MLIR's Standard dialect to express the actual computation (**P2**). Figure 3 shows a stencil program that, for every point of a 64 × 64 × 64-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 grid. Inputs and outputs of a stencil program have this type. A !stencil.**temp** is a multi-dimensional collection of elements on a hyper-rectangular subdomain of the grid. Temporaries have value semantics and are initially not backed by storage. Values of this type either

```
func @sum(%in : !stencil.field<?x?x?xf64>, %out : !stencil.field<?x?x?xf64>) {
  stencil.assert %in ([-4, -4, -4]:[68, 68, 68]) : !stencil.field<?x?x?xf64>
  stencil.assert %out ([-4, -4, -4]:[68, 68, 68]) : !stencil.field<?x?x?xf64>
  %0 = stencil.load %in : (!stencil.field<?x?x?xf64>) -> !stencil.temp<?x?x?xf64>
  %1 = stencil.apply (%arg0 = %0 : !stencil.temp<?x?x?xf64>) -> !stencil.temp<?x?x?xf64> {
      %2 = stencil.access %arg0[1, 0, 0] : (!stencil.temp<?x?x?xf64>) -> f64
      %3 = stencil.access %arg0[-1, 0, 0] : (!stencil.temp<?x?x?xf64>) -> f64
      %4 = addf %2, %3 : f64
      stencil.return %4 : f64
  }
  stencil.store %1 to %out ([0, 0, 0]:[64, 64, 64]) : !stencil.temp<?x?x?xf64> to !stencil.field<?x?x?xf64>
  return
}
```

define storage shapes

stencil operator

define output domain

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

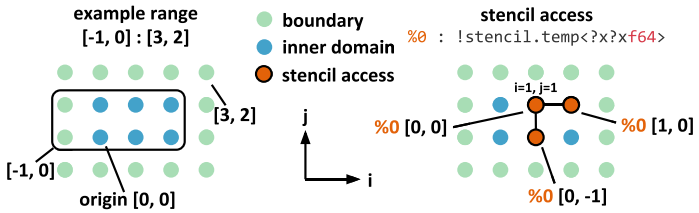

Fig. 4. Example range (left) defined by an inclusive lower and an exclusive upper bound and stencil accesses (right) expressed relative to the current position (i = 1, j = 1).

point to a subdomain of an input array or keep the results computed by a stencil operator. Both types store single- or double-precision floating-point values (f32 or f64) on a one-, two-, or three-dimensional domain.

The Stencil dialect also defines multiple operations introduced in Section 4.3 and Section 4.4. In the example, the nested region of the stencil.**apply** operation implements the stencil operator and the range attribute of the stencil.**store** operation specifies the domain written by the stencil program. Our compiler utilizes the range attribute to automatically infer the entire stencil program's access ranges and iteration domains (cf. Section 5.2).

4.2 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.

Figure 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 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**).

4.3 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.

```
func @prog(%in : !stencil.field<?x?x?xf64>, %out : !stencil.field<?x?x?xf64>) {
  /* ... */
  %0 = stencil.load %in : (!stencil.field<?x?x?xf64>) -> !stencil.temp<?x?x?xf64>
  %1 = stencil.apply (%arg0 = %0 : !stencil.temp<?x?x?xf64>) -> !stencil.temp<?x?x?xf64> {
    /* stencil 1 */
  }
  %2 = stencil.apply (%arg0 = %0 : !stencil.temp<?x?x?xf64>, %arg1 = %1 : !stencil.temp<?x?x?xf64>) -> !stencil.temp<?x?x?xf64> {
    /* stencil 2 */
  }
  /* ... */
}
```

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

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 input elements are read using the stencil.**access** operation that accesses 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. Section 5.3). Figure 4 shows the offset computation (right) for a 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 for weather and climate often implement dozens of dependent stencil operators. A stencil program thus needs additional means to orchestrate them.

4.4 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 enable the program definition. The stencil.**assert** operation specifies the index range for 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 temporary value that contains 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.

Figure 5 shows a stencil program that executes two dependent stencils. The stencil.**load** operation returns a temporary holding the accessed **%in** array elements. 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.**temp** that are not initially backed by storage and are thus also alias-free. Given the value semantics of !stencil.**temp**, the def-use graph encodes the data dependencies between the stencil operators (**P1&P3**).

4.5 Control Flow

Real-world stencil applications are not limited to pure data flow semantics. While we can use just-in-time compilation to handle most of the control-flow at the program level, we resort to MLIR's

```
%0 = scf.if %flag -> (f64) {
  %1 = stencil.access %arg0[0, 0, 0] : (!stencil.temp<?x?x?xf64>) -> f64
  scf.yield %1 : f64
} else {
  %2 = stencil.access %arg1[0, 0, 0] : (!stencil.temp<?x?x?xf64>) -> f64
  scf.yield %2 : f64
}
stencil.return %0 : f64
```

if %flag then %arg0 else %arg1

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

```
%0 = stencil.apply seq(dim = 2, range = 0 to 60, dir = -1)
  (%arg0 = %rhs : !stencil.temp<?x?x?xf64>,
   %arg1 = %sup : !stencil.temp<?x?x?xf64>) -> !stencil.temp<?x?x?xf64> {
  %1 = stencil.index 2 [0, 0, 0] : index
  %cst = constant 59 : index
  %2 = cmpi "eq", %1, %cst : index
  %3 = scf.if %2 -> (f64) {
    %4 = stencil.access %arg0 [0, 0, 0] : (!stencil.temp<?x?x?xf64>) -> f64
    scf.yield %4 : f64
  } else {
    %5 = stencil.access %arg0 [0, 0, 0] : (!stencil.temp<?x?x?xf64>) -> f64
    %6 = stencil.access %arg1 [0, 0, 0] : (!stencil.temp<?x?x?xf64>) -> f64
    %7 = stencil.depend 0 [0, 1] : f64
    %8 = mulf %6, %7 : f64
    %9 = addf %8, %5 : f64
    scf.yield %9 : f64
  }
  stencil.return %3 : f64
}
```

sequential loop
get current position
boundary condition
main stencil
access output 0 at offset [0, 0, 1]
(loop carried dependency)

Fig. 7. The SCF dialect and Stencil dialect extensions enable the implementation of control flow inside the stencil operator.

built-in SCF dialect and a set of Stencil dialect extensions to support control flow at the stencil operator level.

Figure 6 shows a stencil that, depending on a flag, accesses one of two arguments. The scf.**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 scf.**yield** operations. This representation makes the data flow explicit and maintains a single stencil.**return** operation per stencil. An alternative to the scf.**if** operation, is the **select** operation that chooses a value based on a condition. Supporting the scf.**if** operation requires no adaptation of our compiler (**P2**).

Figure 7 shows the backward sweep of the Thomas algorithm—an important computational motif in weather and climate. A boundary condition makes the stencil operator position-dependent, and a data dependency forces its sequential execution. We utilize an optional attribute to specify the sequential loop execution, while the stencil.**depend** operation provides access to loop-carried dependency and the stencil.**index** operation returns the current position. The latter, together with the scf.**if** operation, enables the implementation of the boundary condition. Supporting the scf.**if** operation requires no adaptation of our compiler (**P2**).

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

5 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.

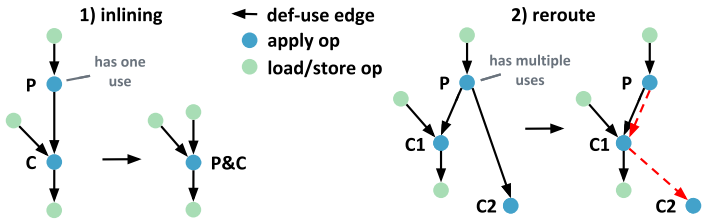

Fig. 8. Two patterns enable the iterative producer-consumer fusion for entire stencil programs. The def-use edges represent the data flow between the stencil operations.

```
%1 = stencil.apply (%arg = %0 : !stencil.temp<?x?x?xf64>) -> !stencil.temp<?x?x?xf64> {
  %2 = stencil.access %arg[1, 0, 0] : (!stencil.temp<?x?x?xf64>) -> f64
  %3 = stencil.access %arg[-1, 0, 0] : (!stencil.temp<?x?x?xf64>) -> f64
  %4 = addf %2, %3 : f64                                                    } j + 0
  %5 = stencil.access %arg[1, 1, 0] : (!stencil.temp<?x?x?xf64>) -> f64
  %6 = stencil.access %arg[-1, 1, 0] : (!stencil.temp<?x?x?xf64>) -> f64
  %7 = addf %5, %6 : f64                                                    } j + 1
  stencil.return unroll [1, 2, 1] %4, %7 : f64, f64
}
```

shifted offsets

unroll attribute

two outputs

Fig. 9. Unrolling the example stencil in the j-dimension.

5.1 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 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.

Figure 8 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, then 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, then 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 if a pattern matches.

The **stencil unrolling** pass replicates a stencil operator multiple times to update more than one grid point at once. Figure 9 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. 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 adjust the access offsets. We also adapt the stencil.**return** operation to return the results of all unrolled loop iterations and annotate 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 loop iterations often evaluate an inlined producer at the same offset. Instead of removing the redundant computation ourselves, we run the existing common subexpression elimination pass.

5.2 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 needed, since a stencil program only defines the output ranges written by the program. The pass starts from these output ranges, follows the use-def chains that define the program's data dependencies, 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. Figure 10 shows the annotated example program before lowering. We compute the access range of an operation as the minimal bounding box containing the access extents of all its consumers. If the consumer is a stencil.**load** operation, then its access extent is equal to the output range attribute. If the consumer is a stencil.**apply** operation, then the access extent is equal to its iteration domain extended by a minimal bounding box containing all stencil accesses to the consumed values.

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.

5.3 Lowering to Explicit 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 Figure 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 dimension. Another example is the scf.**parallel** operation that models a parallel multi-dimensional loop.

Figure 10 illustrates the lowering from the stencil dialect level to the MLIR Standard dialect level for the example introduced in Section 4.1. 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

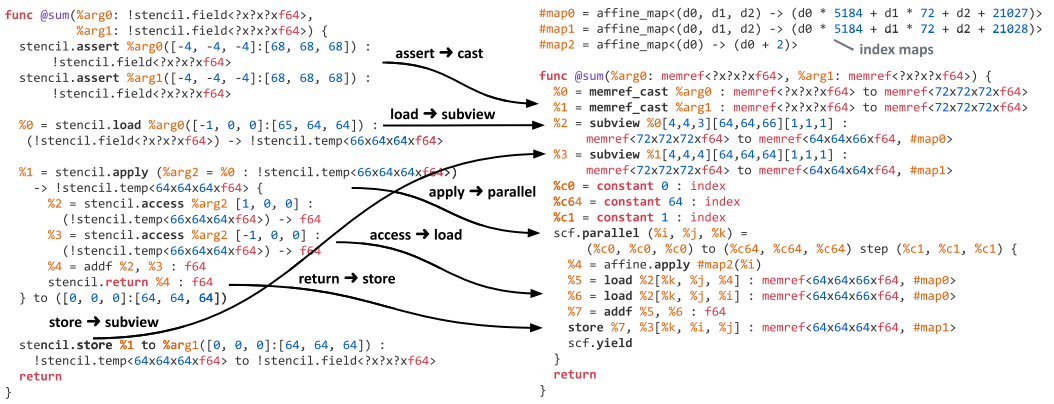

Fig. 10. Conversion of the Stencil dialect to the MLIR SCF+Affine+Standard dialects that further lower to GPU abstractions.

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**).

6 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 Section 2 and implemented the GPU dialect[2] 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.

Figure 11 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 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. GPU-specific primitives such as barrier synchronization, shuffles, and ballots are only available inside a kernel launch.

[2] https://mlir.llvm.org/docs/Dialects/GPU/.

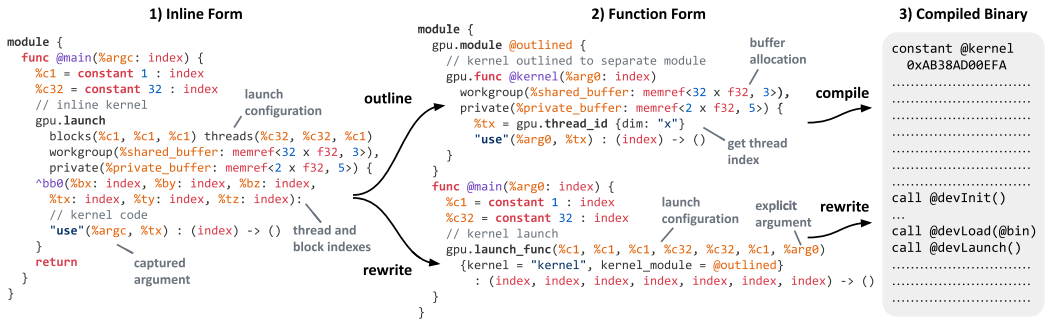

Fig. 11. Lowering of an example kernel: (1) the inline form enables host device code motion and other transformations, (2) the function form isolates the device code in a distinct module that enables device-specific optimizations and separate host/device compilation, and (3) the binary embeds the kernel as constant data.

Additionally, Figure 11 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, which can be optimized, compiled, and executed. The kernel invocations thereby lower into calls to the device driver library or runtime environment.

7 EVALUATION

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

7.1 Experimental Setup

We run our experiments on an NVIDIA Tesla V100-SXM2 with a memory bandwidth of 900 GB/s. We benchmark two domain sizes $128 \times 128 \times 60$ (small) and $256 \times 256 \times 60$ (large) for single-precision (f32) and double-precision (f64) floating-point numbers. We repeated all experiments using 128 and 256 threads per block and did not observe significant runtime differences. We thus present performance numbers for the smaller block size only. 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 initial kernel executions to avoid startup overheads. Additionally, we use the nvprof profiler to collect detailed performance data on the NVIDIA system.

Table 2. Characteristics of our Benchmarks

Name	Dims	Apply Ops	Inputs/Outputs	Arith. Ops	Access Ops	Control Flow	
hdiffsa	2	4	4 / 1	21	22	min	COSMO
hdiffsmag	2	6	8 / 2	56	38	min/max	
hadvuv	2	8	6 / 2	80	45	if	
hadvuv5th	2	8	6 / 2	112	53	if	
fastwavesuv	3	6	9 / 2	43	32	-	
p_grad_c	3	3	7 / 2	24	25	-	FV3
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	

We guarantee 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} (f32) or 10^{-10} (f64).

7.2 Benchmark Kernels

We evaluate our compiler for a set of representative benchmarks[3] derived from the dynamical cores of two popular climate and weather models. COSMO [2] is a regional numerical weather prediction model used by seven national weather services in Europe. FV3 [3] is the dynamical core 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 both codes are stencil computations on regular grids. Both models implement 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 hadvuv and hadvuv5th kernels implement third- and fifth-order horizontal advection, while the fvtp2d kernels implement a monotone two-dimensional finite volume advection operator. The hdiffsa and hdiffsmag kernels perform horizontal diffusion. The fastwavesuv kernel contains parts of the sound wave forward integration, while the p_grad_c and nh_p_grad kernels compute the three-dimensional pressure gradient. Finally, the uvbke kernel is a preprocessing step for the kinetic energy computation in FV3.

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 2 lists core characteristics of our benchmarks such as the dimensionality of the 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.

7.3 Effectiveness of Our Code Transformations

We first evaluate the effectiveness of the optimizing transformations discussed in Section 5.1. We compare four optimization levels: (1) *original*, (2) *inline*, (3) *inline+unroll(2)*, and (4) *inline+unroll(4)*. Optimization level (1) applies no optimizing transformations. Starting from level (2), we apply

[3] https://github.com/spcl/open-earth-benchmarks.

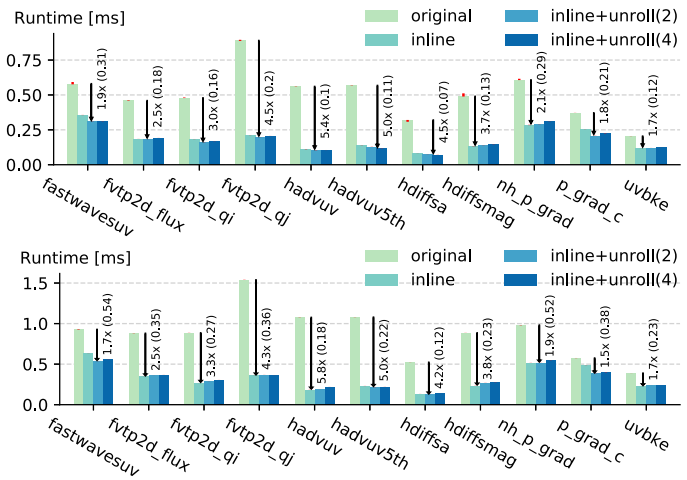

Fig. 12. Runtimes at different optimization levels for f32 (top) and f64 (bottom) floating-point values for $256 \times 256 \times 60$.

stencil inlining, and the levels (3), and (4) additionally perform stencil unrolling by factor two and four, respectively.

Figure 12 compares the runtime for all benchmarks at different optimization levels. We show data for the large problem size and f32 and f64 arithmetic and observe significant speedups for stencil inlining independent of the benchmark. In comparison, stencil unrolling has a smaller effect and sometimes is even detrimental to performance. In the plot, we annotate the speedup and the runtime of the best-performing version. We also run all benchmarks on an AMD Radeon RX 5700 system with 448 GB/s memory bandwidth. We thereby measure a geometric mean speedup of 2.7× and a total runtime of 4.1 ms for the best-performing versions (f32 arithmetic), compared to 3.0× and 1.9 ms on the NVIDIA system. We thus attain comparable speedups and bandwidth rectified performance on the AMD and NVIDIA systems despite the entirely different instruction set architectures and runtime environments, demonstrating the effectiveness of our vendor-independent GPU execution model (cf. Section 6).

Inlining all stencil operators eliminates the accesses of temporary buffers and ensures the program inputs are loaded precisely once. We thus expect stencil inlining to have a strong performance effect due to the resulting bandwidth reduction and despite the redundant computation. Figure 13 confirms that stencil inlining reduces the data movement between the device memory and the L2 cache (bottom), quantifies the extra computation (top) but also shows that stencil inlining eliminates parts of the index computation (middle). Overall the bandwidth reduction and the eliminated index computation overcompensate the additional redundant computation, especially given the low arithmetic intensity of our kernels.

Stencil unrolling removes redundant computations and improves the data-locality in case the data accesses of the unrolled loop iterations overlap. Figure 13 confirms that stencil unrolling reduces both redundant computation (top) and index computation (middle). We also observe less device memory transactions (bottom) for the three-dimensional fastwavesuv, nh_p_grad, and p_grad_c kernels due to improved data-locality. At the same time, stencil unrolling increases the register pressure and reduces the available parallelism. Unrolling the hdiffsmag kernel, for example, increases the register usage from 72 to 96 registers and conversely reduces the occupancy from 40% to 29%. As a result, unrolling increases the runtime from 0.235 ms to 0.248 ms. We thus

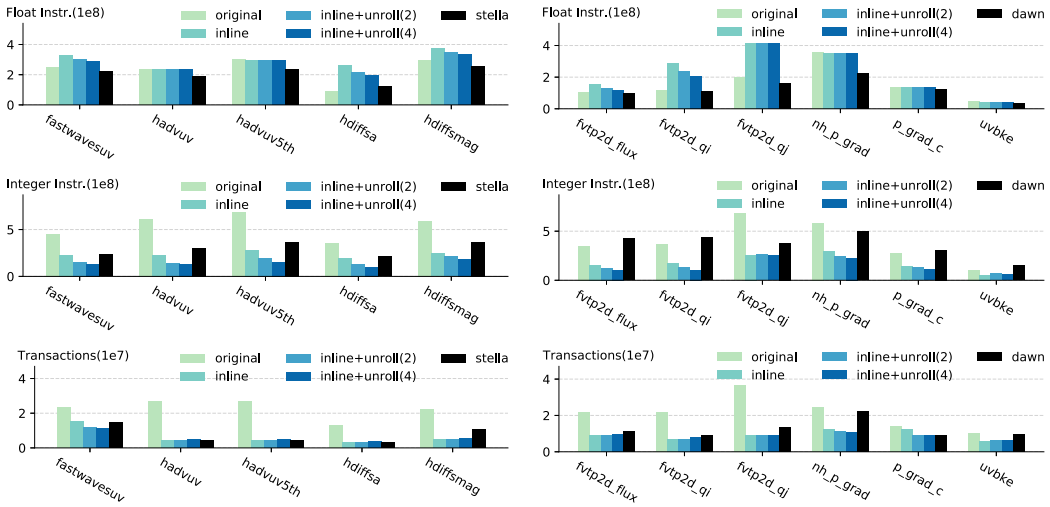

Fig. 13. Number of float instructions, integer instructions, and the device memory transactions at different optimization levels and compared to Stella (COSMO) (left) and FV3 (Dawn) (right) implementations (cf. Section 7.4) for f64 floating-point values and $256 \times 256 \times 60$ (collected using nvprof).

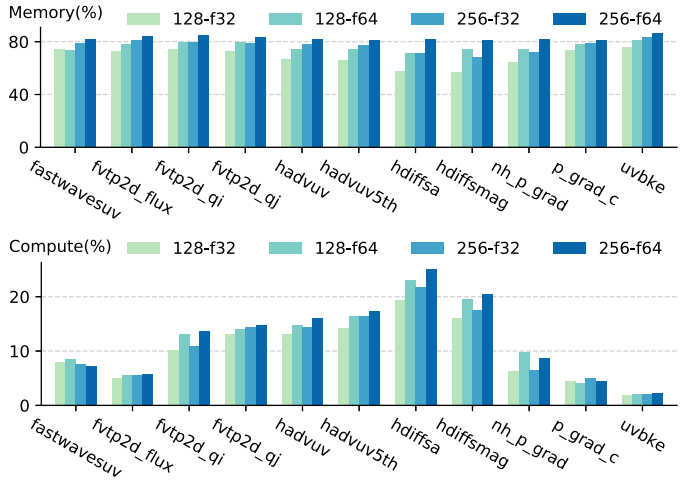

Fig. 14. Utilization of the peak compute throughput (top) and the peak memory bandwidth (bottom) for the best-performing kernel variants in percent (collected using nvprof).

do not expect all stencil programs to benefit from stencil unrolling. Instead, the optimization's effectiveness depends on the complexity of the stencil program (register pressure), the available parallelism, and the potential bandwidth reduction.

Figure 14 illustrates the memory bandwidths and compute throughputs achieved by the best-performing kernel versions. We observe very high memory bandwidth utilization up to 86% and low compute utilization below 22%. These results demonstrate the importance of aggressive stencil inlining and confirm the redundant computation is less of a concern.

In summary, we show that our code transformations yield significant speedups. Selecting the optimal unroll factor or finding good fusion choices for a specific benchmark and target system

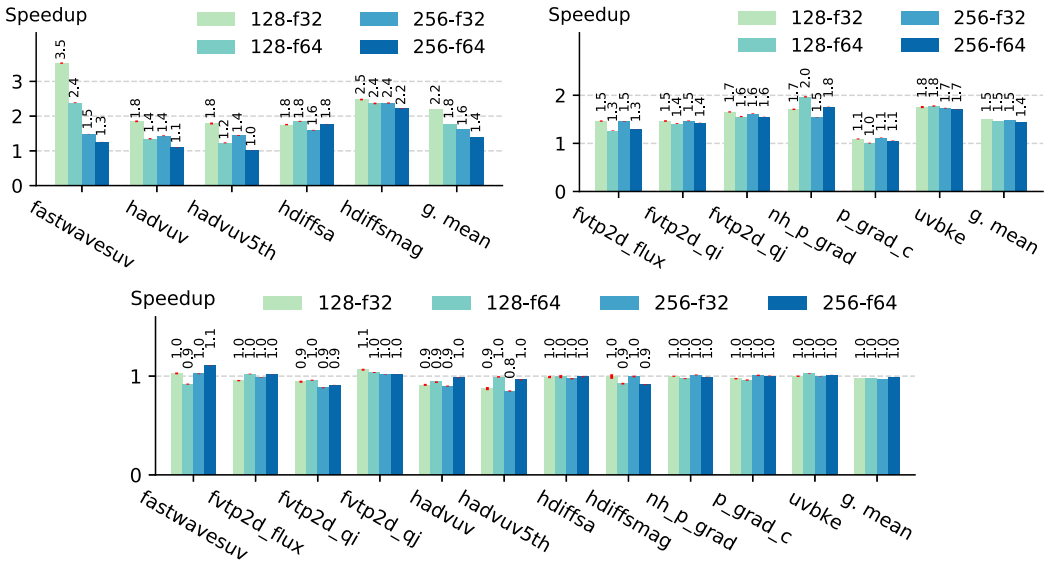

Fig. 15. Speedup of our compiler over Stella (COSMO) (top left), Dawn (FV3) (top right), and Halide (bottom) for f32 and f64 floating-point values for $128 \times 128 \times 60$ and $256 \times 256 \times 60$.

combination is not the scope of our work. Instead, we employ empirical tuning to find the best unroll factor and always fuse all stencil operators (optimizing larger stencil programs will require a fusion heuristic).

7.4 Comparison to State-of-the-art Solutions

We now compare the runtime of our optimized kernels to Stella [24] and Dawn [42] generated CUDA [41] implementations. Stella is a C++ embedded DSL used to run the GPU version of COSMO in production. Dawn is a research compiler that lowers a high-level climate IR to efficient CUDA code. Both implement overlapped tiling [27] using shared memory and stream data [46] in registers along the k-dimension. This execution model limits the redundant computation to the tile boundaries reduces the parallelism in the k-dimension. In comparison, our stencil inlining plus unrolling performs redundant computation at the thread level but requires no thread synchronization during the kernel execution.

We further benchmark the Halide [44] image processing framework. It is an industry-grade compiler that lowers a domain-specific data flow language to target specific code. While its data flow language is a good fit for plain stencil computations, it is not designed to express loop-carried dependencies as they appear in weather and climate (cf. Section 4.5). In contrast to our compiler and the other domain-specific solutions, Halide performs aggressive floating-point optimizations that affect the precision of the results (e.g., fused multiply-add, approximate divisions, and operation reordering). While these limitations prevent a broad adoption in weather and climate, we still compare to Halide, since it is a widely accepted reference for stencil compilers.

Figure 15 compares for all benchmarks the best-performing variant generated by the Open Earth Compiler to their Stella (COSMO) and Dawn (FV3) counterparts as well as to Halide variants. We are performance-competitive to Halide and attain geometric mean speedups that range from 1.4× to 2.2× compared to Stella and Dawn.

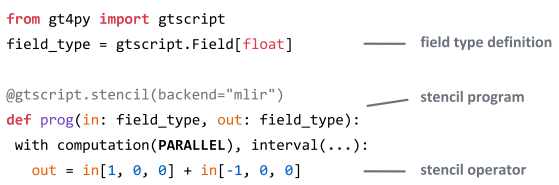

Fig. 16. GT4Py version of the example stencil program.

We attribute the performance of our compiler to the simple execution model. It fuses all stencil operators and stores temporaries in registers to limit the data movement, and it introduces redundant computation instead of thread synchronizations to avoid parallelization overheads. Its only disadvantage is the redundant computation, which due to the low arithmetic intensity of our kernels shown in Figure 14, is less critical. In Figure 13, we indeed observe lower device memory bandwidth requirements (bottom) and notable amounts of redundant computation (top) compared to Stella (COSMO) and Dawn (FV3) implementations. Having compile-time information about storage shapes and iteration domains in return eliminates parts of the index computation (middle). Avoiding synchronizations at the cost of additional computation and utilizing compile-time information, thus turn out to be beneficial compared to the Stella and Dawn execution models.

In contrast, Halide and our compiler perform stencil inlining and unrolling. We configure both compilers to inline all stencil operators, tune the unroll factor using the same search space, and enable our compiler's fast math flag to account for Halide's aggressive floating-point optimizations. As a result, the two compilers attain similar performance.

The compilation times and the sizes of the generated binaries across all compilers and kernels are comparable. We measure compilation times and binary sizes ranging from 3 to 12 seconds and 50 to 800 kB, respectively. Our compiler thereby spends 95% of the execution time in the low-level GPU code generation, demonstrating the domain-specific compilation's low cost. Low-level overheads consequently hide possible compile-time differences among the different domain-specific solutions.

Our compiler outperforms Stella and Dawn, demonstrating the potential of stencil inlining and unrolling compared to overlapped tiling and streaming. While Stella and Dawn's execution model may have been the design sweet-spot for the target hardware at the time, stencil inlining and unrolling seem favorable on modern hardware. We also attain performance parity compared to Halide, the standard solution in the image processing domain, emphasizing our compiler's quality.

7.5 Lowering User-facing Code to the Stencil Dialect

We design our Stencil dialect as a target for user-facing domain-specific languages. To study the feasibility of such a lowering, we integrate our compiler with **GridTools for Python (GT4Py)** [5], an embedded domain-specific language for weather and climate.

In Figure 16, we show a GT4Py version of the example stencil program introduced in Section 4.1. GT4Py lowers the user code to an internal IR, consisting of a compute domain, data field descriptors, and a set of computations in the form of an **abstract syntax tree (AST)**. We traverse the AST to emit MLIR, compile the resulting MLIR program to a binary, and produce Python bindings to link the generated binary to the calling program.

The functional lowering demonstrates our compiler's applicability in the context of an end-to-end solution for the weather and climate domain.

8 RELATED WORK

Accelerated systems made programming model innovations inevitable. Kokkos [16] and Raja [6] are C++ performance portability layers. Kruse and Finkel [28] propose compiler directives to control generalized loop transformations. PENCIL [9] and Polly-ACC [21] automate the accelerator mapping using the polyhedral model. DaCe [12] 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 [13, 31]. The age of stencil compilers started even earlier: Halide [44] and Polymage [38] tune image processing pipelines, Pochoir [54] implements cache-oblivious tiling, SDSLc [45] supports many targets (SIMD, GPU, and FPGA), Panda [51] supports distributed memory, and YASK [58] specifically targets Intel processors. Lift [25] has also been shown effective for stencil codes. This diversity demonstrates the importance of a shared compiler infrastructure.

Multiple projects work on solutions for weather and climate. The CLIMA [1] effort develops a novel earth system model using the Julia language. The LFRic [8] climate modeling system relies on the Python-based PSyclone compiler. The Dawn [42] compiler lowers a high-level IR that has sequential semantics. Stella [24] and GridTools [4] use C++ template metaprogramming to support CPU and GPU systems. CLAW [14] and Hybrid Fortran [39] extend Fortran to achieve performance portability. Despite their heterogeneity, all of them could benefit from a shared compiler infrastructure.

Several frameworks support the development of domain-specific compilers. AnyDSL [32] supports partial evaluation using minimal annotations in the Impala front end language. Lightweight modular staging [49] is a technique that uses Scala's type system to transform codes before their execution. It forms the basis of the Delite [53] compiler framework. Lua script similarly supports staging via the Terra [15] low-level language. Lift [52] finally combines a functional language and rewrite rules to generate performance portable code. MLIR [30] 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 [18, 19, 27, 33, 34, 40] and fusion [22, 48, 57] are the core optimizations for bandwidth-limited low-order stencils as they appear in weather and climate. Other works optimize the resource utilization [46, 47] or discuss the optimization of high-order stencils [45, 59]. Our compiler implements a variant of overlapped tiling [27] that introduces redundant computation for every thread.

9 CONCLUSION

We introduced multi-level IR rewriting, an approach to build 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 11 stencil programs relevant to existing climate models, COSMO and FV3, demonstrates that the Open Earth Compiler, compared to state-of-the-art solutions for weather and climate, generates up to 3.4× faster code and delivers a geomean speedup between 1.4× and 1.8× across problem sizes and precisions. We suggest that multi-level IR rewriting and the associated design

principles are a promising approach to rapidly design and deploy domain-specific compilers that can take advantage of the reusable components of the MLIR ecosystem.

ACKNOWLEDGMENTS

We thank Jean-Michel Gorius for his foundational stencil compiler work and the continuous support of our project. We appreciate the assistance of Carlos Osuna and the MeteoSwiss compiler team throughout the project. Finally, we thank Hannes Vogt for providing the initial out-of-tree development setup and Felix Thaler for sharing performance results that inspired our work.

REFERENCES

[1] 2020. CLIMA. Retrieved from https://github.com/climate-machine/CLIMA/.
[2] 2020. Consortium for Small-scale Modeling. Retrieved from http://www.cosmo-model.org/.
[3] 2020. FV3: Finite-Volume Cubed-Sphere Dynamical Core. Retrieved from https://www.gfdl.noaa.gov/fv3/.
[4] 2020. GridTools. Retrieved from https://github.com/GridTools/gridtools.
[5] 2020. GT4Py. Retrieved from https://github.com/gridtools/gt4py.
[6] 2020. RAJA. Retrieved from https://github.com/LLNL/RAJA.
[7] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. 2019. Learning to optimize halide with tree search and random programs. *ACM Trans. Graph.* 38, 4 (July 2019). DOI: https://doi.org/10.1145/3306346.3322967
[8] S. V. Adams, R. W. Ford, M. Hambley, J. M. Hobson, I. Kavčič, C. M. Maynard, T. Melvin, E. H. Müller, S. Mullerworth, A. R. Porter, M. Rezny, B. J. Shipway, and R. Wong. 2019. LFRic: Meeting the challenges of scalability and performance portability in weather and climate models. *J. Parallel Distrib. Comput.* 132 (2019), 383–396. DOI: https://doi.org/10.1016/j.jpdc.2019.02.007
[9] 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. 2015. PENCIL: A platform-neutral compute intermediate language for accelerator programming. In *Proceedings of the International Conference on Parallel Architecture and Compilation (PACT'15)*. 138–149.
[10] Michael Baldauf, Axel Seifert, Jochen Förstner, Detlev Majewski, Matthias Raschendorfer, and Thorsten Reinhardt. 2011. Operational convective-scale numerical weather prediction with the COSMO model: Description and sensitivities. *Month. Weath. Rev.* 139, 12 (2011), 3887–3905.
[11] Ulysse Beaugnon, Antoine Pouille, Marc Pouzet, Jacques Pienaar, and Albert Cohen. 2017. Optimization space pruning without regrets. In *Proceedings of the 26th International Conference on Compiler Construction*. 34–44.
[12] Tal Ben-Nun, Johannes de Fine Licht, Alexandros Nikolaos Ziogas, Timo Schneider, and Torsten Hoefler. 2019. Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'19)*.
[13] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. USENIX Association, 578–594. Retrieved from https://www.usenix.org/conference/osdi18/presentation/chen.
[14] Valentin Clement, Sylvaine Ferrachat, Oliver Fuhrer, Xavier Lapillonne, Carlos E. Osuna, Robert Pincus, Jon Rood, and William Sawyer. 2018. The CLAW DSL: Abstractions for performance portable weather and climate models. In *Proceedings of the Platform for Advanced Scientific Computing Conference (PASC'18)*. Association for Computing Machinery, New York, NY. DOI: https://doi.org/10.1145/3218176.3218226
[15] Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. 2013. Terra: A multi-stage language for high-performance computing. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13)*. Association for Computing Machinery, New York, NY, 105–116. DOI: https://doi.org/10.1145/2491956.2462166
[16] H. C. Edwards and C. R. Trott. 2013. Kokkos: Enabling performance portability across manycore architectures. In *Proceedings of the Extreme Scaling Workshop (XSW'13)*. 18–24.
[17] Oliver Fuhrer, Carlos Osuna, Xavier Lapillonne, Tobias Gysi, Ben Cumming, Mauro Bianco, Andrea Arteaga, and Thomas Schulthess. 2014. Towards a performance portable, architecture agnostic implementation strategy for weather and climate models. *Supercomput. Front. Innov.* 1, 1 (2014).
[18] Tobias Grosser, Albert Cohen, Justin Holewinski, P. Sadayappan, and Sven Verdoolaege. 2014. Hybrid hexagonal/classical tiling for GPUs. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and*

Optimization (CGO'14). Association for Computing Machinery, New York, NY, 66–75. DOI: https://doi.org/10.1145/2544137.2544160

[19] Tobias Grosser, Albert Cohen, Paul H. J. Kelly, J. Ramanujam, P. Sadayappan, and Sven Verdoolaege. 2013. Split tiling for GPUs: Automatic parallelization using trapezoidal tiles. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units (GPGPU'13)*. Association for Computing Machinery, New York, NY, 24–31. DOI: https://doi.org/10.1145/2458523.2458526

[20] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. 2012. Polly—Performing polyhedral optimizations on a low-level intermediate representation. *Parallel Process. Lett.* 22, 04 (2012), 1250010.

[21] Tobias Grosser and Torsten Hoefler. 2016. Polly-ACC transparent compilation to heterogeneous hardware. In *Proceedings of the International Conference on Supercomputing (ICS'16)*. Association for Computing Machinery, New York, NY. DOI: https://doi.org/10.1145/2925426.2926286

[22] Tobias Gysi, Tobias Grosser, and Torsten Hoefler. 2015. MODESTO: Data-centric analytic optimization of complex stencil programs on heterogeneous architectures. In *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS'15)*. Association for Computing Machinery, New York, NY, 177–186. DOI: https://doi.org/10.1145/2751205.2751223

[23] T. Gysi, T. Grosser, and T. Hoefler. 2019. 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 (PACT'19)*. 370–382.

[24] Tobias Gysi, Carlos Osuna, Oliver Fuhrer, Mauro Bianco, and Thomas C. Schulthess. 2015. 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 (SC'15)*. Association for Computing Machinery, New York, NY. DOI: https://doi.org/10.1145/2807591.2807627

[25] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. 2018. High performance stencil code generation with lift. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'18)*. Association for Computing Machinery, New York, NY, 100–112. DOI: https://doi.org/10.1145/3168824

[26] Lucas M. Harris and Shian-Jiann Lin. 2013. A two-way nested global-regional dynamical core on the cubed-sphere grid. *Month. Weath. Rev.* 141, 1 (2013), 283–306. DOI: https://doi.org/10.1175/MWR-D-11-00201.1

[27] Justin Holewinski, Louis-Noël Pouchet, and P. Sadayappan. 2012. High-performance code generation for stencil computations on GPU architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing (ICS'12)*. Association for Computing Machinery, New York, NY, 311–320. DOI: https://doi.org/10.1145/2304576.2304619

[28] M. Kruse and H. Finkel. 2018. User-directed loop-transformations in Clang. In *Proceedings of the IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC'18)*. 49–58. DOI: https://doi.org/10.1109/LLVM-HPC.2018.8639402

[29] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'04)*. IEEE, 75–86.

[30] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling compiler infrastructure for domain specific computation. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO'21)*. 2–14. DOI: https://doi.org/10.1109/CGO51591.2021.9370308

[31] Chris Leary and Todd Wang. 2017. XLA: TensorFlow, compiled. In *Proceedings of the TensorFlow Dev Summit*.

[32] Roland Leißa, Klaas Boesche, Sebastian Hack, Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, André Müller, and Bertil Schmidt. 2018. AnyDSL: A partial evaluation framework for programming high-performance libraries. *Proc. ACM Program. Lang.* 2, OOPSLA (Oct. 2018). DOI: https://doi.org/10.1145/3276489

[33] Naoya Maruyama and Takayuki Aoki. 2014. Optimizing stencil computations for NVIDIA Kepler GPUs. In *Proceedings of the 1st International Workshop on High-performance Stencil Computations*. 89–95.

[34] Kazuaki Matsumura, Hamid Reza Zohouri, Mohamed Wahib, Toshio Endo, and Satoshi Matsuoka. 2020. 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 (CGO'20)*. Association for Computing Machinery, New York, NY, 199–211. DOI: https://doi.org/10.1145/3368826.3377904

[35] William M. McKeeman. 1965. Peephole optimization. *Commun. ACM* 8, 7 (1965), 443–444.

[36] G. A. McMechan. 1983. Migration by extrapolation of time-dependent boundary VALUES. *Geophys. Prospect.* 31 (June 1983), 413–420. DOI: https://doi.org/10.1111/j.1365-2478.1983.tb01060.x

[37] Steven S. Muchnick. 1998. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA.

[38] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. 2015. PolyMage: Automatic optimization for image processing pipelines. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*. Association for Computing Machinery, New York, NY, 429–443. DOI: https://doi.org/10.1145/2694344.2694364

[39] Michel Müller and Takayuki Aoki. 2018. Hybrid Fortran: High productivity GPU porting framework applied to Japanese weather prediction model. In *Accelerator Programming Using Directives*, Sunita Chandrasekaran and Guido Juckeland (Eds.). Springer International Publishing, Cham, 20–41.

[40] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey. 2010. 3.5-D Blocking optimization for stencil computations on modern CPUs and GPUs. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13.

[41] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable parallel programming with CUDA. *Queue* 6, 2 (Mar. 2008), 40–53. DOI: https://doi.org/10.1145/1365490.1365500

[42] Carlos Osuna, Tobias Wicky, Fabian Thuering, Torsten Hoefler, and Oliver Fuhrer. 2020. Dawn: A high-level domain-specific language compiler toolchain for weather and climate applications. *Supercomput. Front. Innov.* 7, 2 (2020).

[43] Dan Quinlan. 2000. ROSE: Compiler support for object-oriented frameworks. *Parallel Process. Lett.* 10, 02n03 (2000), 215–226.

[44] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13)*. Association for Computing Machinery, New York, NY, 519–530. DOI: https://doi.org/10.1145/2491956.2462176

[45] Prashant Rawat, Martin Kong, Tom Henretty, Justin Holewinski, Kevin Stock, Louis-Noël Pouchet, J. Ramanujam, Atanas Rountev, and P. Sadayappan. 2015. 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 (WOLFHPC'15)*. Association for Computing Machinery, New York, NY. DOI: https://doi.org/10.1145/2830018.2830025

[46] Prashant Singh Rawat, Changwan Hong, Mahesh Ravishankar, Vinod Grover, Louis-Noël Pouchet, and P. Sadayappan. 2016. Effective resource management for enhancing performance of 2D and 3D stencils on GPUs. In *Proceedings of the 9th Workshop on General Purpose Processing Using Graphics Processing Unit (GPGPU'16)*. Association for Computing Machinery, New York, NY, 92–102. DOI: https://doi.org/10.1145/2884045.2884047

[47] Prashant Singh Rawat, Fabrice Rastello, Aravind Sukumaran-Rajam, Louis-Noël Pouchet, Atanas Rountev, and P. Sadayappan. 2018. Register optimizations for stencils on GPUs. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'18)*. Association for Computing Machinery, New York, NY, USA, 168–182. DOI: https://doi.org/10.1145/3178487.3178500

[48] P. S. Rawat, M. Vaidya, A. Sukumaran-Rajam, M. Ravishankar, V. Grover, A. Rountev, L. Pouchet, and P. Sadayappan. 2018. Domain-specific optimization and generation of high-performance GPU code for stencil computations. *Proc. IEEE* 106, 11 (2018), 1902–1920.

[49] Tiark Rompf and Martin Odersky. 2010. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. In *Proceedings of the 9th International Conference on Generative Programming and Component Engineering (GPCE'10)*. Association for Computing Machinery, New York, NY, 127–136. DOI: https://doi.org/10.1145/1868294.1868314

[50] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1988. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 12–27.

[51] Mohammed Sourouri, Scott B. Baden, and Xing Cai. 2017. Panda: A compiler framework for concurrent CPU+GPU execution of 3D stencil computations on GPU-accelerated supercomputers. *Int. J. Parallel Program.* 45, 3 (June 2017), 711–729. DOI: https://doi.org/10.1007/s10766-016-0454-1

[52] M. Steuwer, T. Remmelg, and C. Dubach. 2017. LIFT: A functional data-parallel IR for high-performance GPU code generation. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO'17)*. 74–85.

[53] Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2014. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Trans. Embed. Comput. Syst.* 13, 4s (April 2014). DOI: https://doi.org/10.1145/2584665

[54] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. 2011. The Pochoir stencil compiler. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'11)*. Association for Computing Machinery, New York, NY, 117–128. DOI: https://doi.org/10.1145/1989493.1989508

[55] Nicolas Vasilache, Cédric Bastoul, Albert Cohen, and Sylvain Girbal. 2006. Violated dependence analysis. In *Proceedings of the 20th International Conference on Supercomputing*. 335–344.

[56] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral parallel code generation for CUDA. *ACM Trans. Archit. Code Optim.* 9, 4 (Jan. 2013). DOI: https://doi.org/10.1145/2400682.2400713

[57] M. Wahib and N. Maruyama. 2014. Scalable kernel fusion for memory-bound GPU applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 191–202.

[58] C. Yount, J. Tobin, A. Breuer, and A. Duran. 2016. YASK—Yet Another Stencil Kernel: A framework for HPC stencil code-generation and tuning. In *Proceedings of the 6th International Workshop on Domain-specific Languages and High-level Frameworks for High-performance Computing (WOLFHPC'16)*. 30–39.

[59] Tuowen Zhao, Protonu Basu, Samuel Williams, Mary Hall, and Hans Johansen. 2019. 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 (SC'19)*. Association for Computing Machinery, New York, NY. DOI: https://doi.org/10.1145/3295500.3356210

[60] Oleksandr Zinenko, Sven Verdoolaege, Chandan Reddy, Jun Shirako, Tobias Grosser, Vivek Sarkar, and Albert Cohen. 2018. Modeling the conflicting demands of parallelism and temporal/spatial locality in affine scheduling. In *Proceedings of the 27th International Conference on Compiler Construction (CC'18)*. Association for Computing Machinery, New York, NY, 3–13. DOI: https://doi.org/10.1145/3178372.3179507

Received December 2020; revised April 2021; accepted May 2021