Modeling Stencils in a Multi-Level Intermediate Representation*

Jean-Michel Gorius[1]

Univ Rennes, ENS Rennes – F-35000, Rennes, France
jean-michel.gorius@ens-rennes.fr

Supervisor:
Tobias Grosser
Scalable Parallel Computing Lab
ETH Zurich – Switzerland

Internship location and dates: ETH Zurich, 05/15/19–07/26/19

Abstract. Stencil computations arise in a wide variety of applications, from climate simulation to machine learning and image processing. In this report, we present a compiler intermediate representation specifically designed to express common constructs in stencil computational kernels. We describe the underlying principles of this intermediate representation and we show that it can be used to express complex stencils in the context of climate and weather simulation.

Keywords: Stencils, intermediate representation, LLVM, MLIR, climate and weather simulation, COSMO

1 Introduction

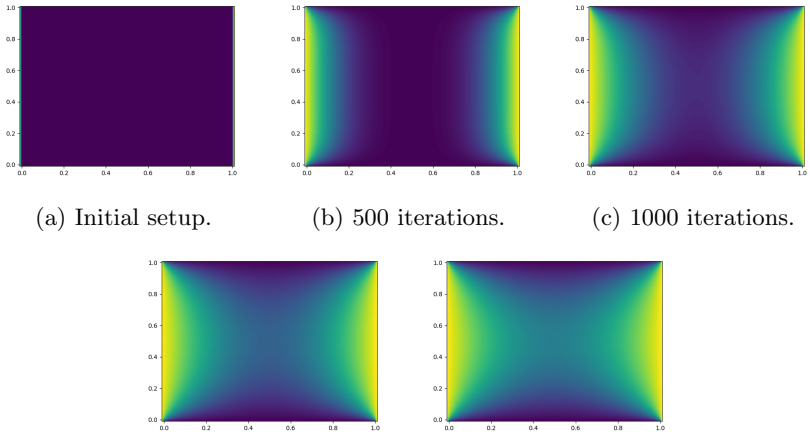
Stencil computations are used extensively in a large number of fields, including physical simulation and modeling, machine learning, computer vision and image processing. Stencils and their applications have been extensively studied during the last decades and are still at the core of many research problems. Their ubiquity has led to the development of methods aimed at easing the expression of stencil computations by domain scientists. The most prominent of those tools are stencil domain-specific languages.

1.1 Stencil computations

This section gives a brief overview of stencils and stencil computations. It is far from an exhaustive coverage of those topics and is only meant to give the reader some background knowledge about the type of computations we will focus on in the rest of this report.

* Joint research work with Nicolas Chappe, ENS Lyon, and the Swiss Federal Office of Meteorology and Climatology (MeteoSwiss).

Computational kernels can be divided in several categories, one of which is *stencil kernels*. A stencil kernel is an iterative kernel that traverses a grid and updates each grid cell according to the values of adjacent cells. Those adjacent cells are accessed according to a fixed pattern called the *stencil*. Stencils can be used on regular grids [5] as well as on irregular grids [13] and are often associated with differential calculus and numerical solving of ordinary or partial differential equations [16].



(a) Initial setup. (b) 500 iterations. (c) 1000 iterations.

(d) 1500 iterations. (e) 2000 iterations.

Fig. 1: Sample run of a Jacobi stencil over a 2-dimensional grid.

To get more familiar with the basic principles of stencils, let us consider an example of partial differential equation solving using an iterative stencil kernel. Let $F \colon \mathbb{R}^2 \to \mathbb{R}$ be a real-valued $\mathcal{C}^2$ function over a 2-dimensional space. Consider the second-order partial differential equation defined by

$$\frac{\partial^2 F}{\partial x^2} + \frac{\partial^2 F}{\partial y^2} = 0. \tag{1}$$

This kind of equation often arises in physical modeling problems such as heat diffusion and electric potential distribution modelization. Given initial boundary conditions, equation 1 can be solved iteratively by using a *Jacobi stencil*. The Jacobi stencil kernel [7] is defined as follows for all cells (i, j) in a regular grid G.

$$G'(i,j) = \frac{1}{4}\left(G(i-h,j) + G(i+h,j) + G(i,j-h) + G(i,j+h)\right), \tag{2}$$

where G' is the output 2-dimensional grid and h is the size of a grid cell. Figure 1 illustrates the output of a sample run of the Jacobi stencil kernel. The iteration

is computed on the domain $\mathscr{D} = [0,1]^2$ with initial conditions such that

$$\mathscr{D}(i,j) = \begin{cases} 0 & \text{if } 0 < i < 1 \text{ and } 0 < j < 1, \\ 1 & \text{otherwise.} \end{cases} \tag{3}$$

Stencil code development often requires deep knowledge of the target application domain. This domain-specific knowledge often requires a higher level of abstraction to be easily expressible in source code. The need for new abstractions has led to the development of numerous stencil-based domain-specific languages.

1.2 Stencil domain-specific languages

One of the primary purposes of domain-specific languages (DSL) is to raise the level of abstraction at which the user of the language can express domain-specific constructs [10]. In the case of stencil computations, this higher-level of abstraction is crucial to enable domain scientists to develop custom stencil kernels without having to delve into the implementations details inherent to this kind of computations. Typical details one would like to avoid when developing a stencil kernel include memory layout optimizations, data dependency resolution and multi-stage splitting [4]. The following paragraphs give a broad overview of stencil DSLs developed for various application domains, some of which are used in production environments.

Image processing and computer vision make heavy use of stencil-based transformation and analysis kernels [8]. Ragan-Kelley et al. [15] developed a fully-featured optimizing compiler infrastructure built to support the Halide DSL. Halide is an image processing DSL that enables the user to express image processing pipelines in a high-level language. The underlying compiler implements numerous optimization passes which allow it to produce very efficient compute kernels.

Breaking the boundaries of a single application domain, complete software platforms such as Simflowny [2], and later Simflowny 2 [3], aim at generalizing compilation and optimization techniques for stencil DSLs to larger application fields. In the case of Simflowny and Simflowny 2, the authors present a general-purpose software platform broadly aimed at physical modeling and simulation.

Weather and climate simulation are a major user of stencil computations in the context of High Performance Computing (HPC). Stencil DSLs specifically targeted at climate and weather simulation applications, such as the Stencil Loop Language (STELLA) [9] and GridTools [17], enable the user to express complex computational patterns, abstracting away the convoluted optimizations necessary to get the most out of the HPC clusters the simulations are being run on. STELLA introduces a high-level DSL along with a compiler frontend which takes this DSL as an input and emits C++ template code, generating efficient loops and memory access patterns depending on the target architecture. This infrastructure is complemented by a modular code generation backend, which can emit CPU or GPU code. GridTools takes a different approach and embeds a DSL directly in the C++ language. It is designed as a successor to STELLA [17]

and is developed by the Swiss National Supercomputing Center (CSCS) and the Swiss Federal Office of Meteorology and Climatology (MeteoSwiss).

2 Background and motivation

A growing number of stencil DSLs are being developed and maintained alongside completely dedicated compiler infrastructures. This leads to little or no code reuse at all and creates an additional maintenance cost for DSLs aimed at production-ready applications. Our work aims at providing a common baseline for stencil DSL development and experimentation centered around principles borrowed from the LLVM project. In the following sections, we give a brief overview of the LLVM project (section 2.1) before introducing a multi-level intermediate representation (MLIR) based around the key concepts of LLVM (section 2.2). We make use of the MLIR infrastructure to build a custom compiler intermediate representation (IR) that is able to represent patterns and constructs commonly found in stencil kernels.

2.1 The LLVM project

The LLVM project is a collection of modular and reusable components used to create complete compiler infrastructures [11]. The key idea behind LLVM is the use of a common intermediate representation, the LLVM IR, coupled with an optimizer. What set the LLVM IR apart from other existing compiler IRs at the time it was first introduced is its textual representation; LLVM IR can be stored in a file in a textual form which is easily readable and understandable by the compiler developers. This in turn makes it easier to debug and analyze the results of transformation and optimization passes.

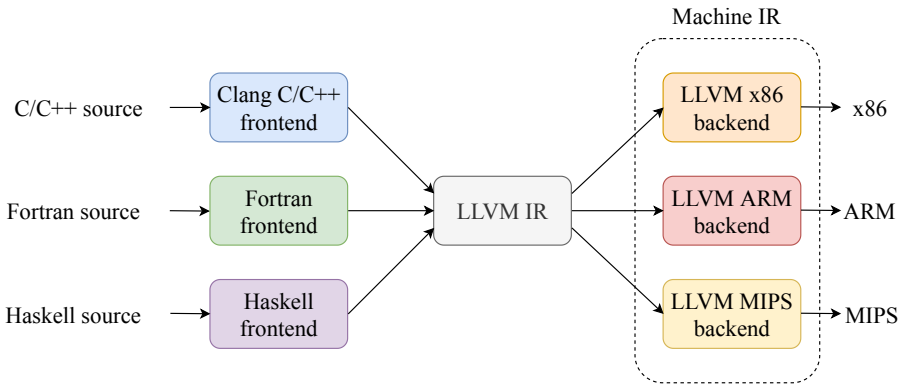


Fig. 2: The LLVM architecture. Modular frontends generate LLVM IR, which is then passed on to modular backends for machine code generation.

Another key concept of the LLVM infrastructure is its modular architecture, which is illustrated in figure 2. A set of different frontends can be plugged onto the LLVM optimizer by generating LLVM IR and another set of architectural backends can be added to perform architecture-specific optimizations as well as code generation.

The architecture depicted in figure 2 is well-suited for languages like C or Fortran, but it soon appeared that there was a need for some kind of higher-level IR to ease the development of language-specific high-level optimization passes. Languages like Swift, Rust and Julia have their own frontend to the LLVM optimizer, but they also include additional intermediate representations before generating LLVM IR, as shown in figure 3.

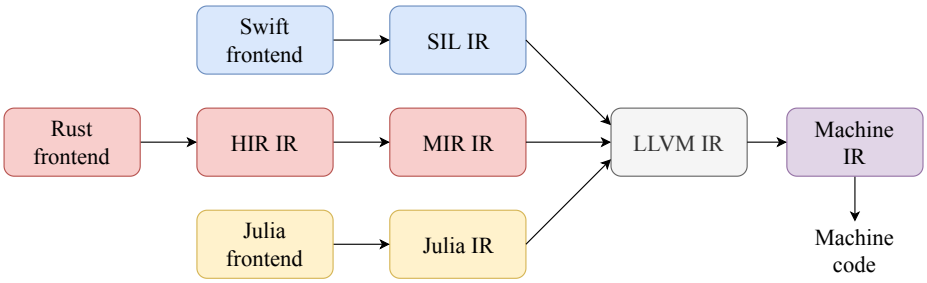


Fig. 3: Modern languages introduce higher-level intermediate representations before going down to LLVM IR.

This new kind of compiler design, which makes use of several layers of intermediate representations, *lowering* from one to the other as the compilation process goes on, is what inspired the development of MLIR, a multi-level intermediate representation [12].

2.2 MLIR: a Multi-Level Intermediate Representation

Originally designed to be used in the TensorFlow environment [1, 12], MLIR (for *Multi-Level Intermediate Representation*) is a compiler infrastructure built on top of LLVM and aimed at easing the development of high-level IRs. It builds upon lessons learned from the LLVM ecosystem and provides a robust set of tools to design and prototype new compiler intermediate representations.

MLIR is a textual intermediate representation based on *Static Single Assignment* (SSA) form [6], which means that values are defined and assigned to only once in the entire program. SSA form is also used by the LLVM compiler infrastructure, as it enables for easier optimization pass writing.

The core components of the MLIR infrastructure are presented in figure 4. MLIR is built on top of the concept of *dialects*. A dialect is akin to a namespace containing custom operations and custom types. When one develops an intermediate representation using MLIR, it usually takes the form a dialect. The basic

code units in MLIR are *operations*. An operation represents a given computation and can take a number of *operands* and return zero, one or more *results*. Each operation can have *attributes* attached to it. An attribute is a constant value known at compile time and that can carry information relevant to the execution and/or compilation context of its attached operation. Custom parsing, printing and correctness verification methods can be attached to any given MLIR operation.

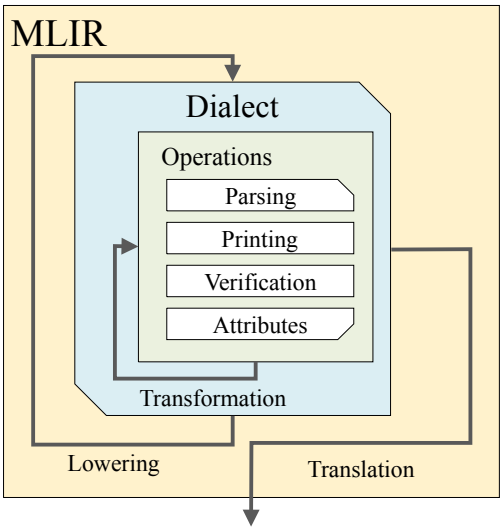


Fig. 4: The MLIR infrastructure.

In addition to giving the user a way to represent custom operations and types, MLIR provides facilities to implement rewriting passes. Those passes can be divided in two categories: *transformation passes* and *lowering passes*. A transformation pass takes MLIR code as an input and rewrites the operations from a given dialect $\mathscr{D}$ to other operations from the same dialect. This can be useful when writing passes such as function call inlining and constant propagation. On the other hand, a lowering pass rewrites operations from a dialect $\mathscr{D}_1$ to operations from another dialect $\mathscr{D}_2$. As the name implies, lowering passes are often used to lower the level of abstraction of the IR so as to enable new kinds of optimizations and rewriting techniques.

After all the required transformation and lowering passes have been run on the input code, the resulting IR can now be used to emit code, be it executable machine code, LLVM IR or higher-level source code. This *translation* phase makes use of a user-provided backend which is in charge of traversing the MLIR representation and emitting corresponding target instructions.

3 Designing a stencil intermediate representation

The capabilities offered by MLIR make it a good candidate for developing domain-specific intermediate representations. This section presents our attempt at modeling common concepts found in stencil kernels and integrate them in a compiler IR. More specifically, section 3.1 presents some key concepts of stencil computations and how they are modeled inside our IR and section 3.2 gives a brief overview of our IR's syntax. The following sections present the transformation and lowering passes we have implemented (section 3.3) and the last section (section 3.4) takes a look at the code generation backend.

3.1 Key concepts

Our stencil intermediate representation is inspired by the concepts expressed in a domain-specific language designed by MeteoSwiss for their production-ready climate and weather simulation model [17]. However, some of the ideas that it expresses are generic and are not tied to the field of climate science, making them applicable to a wide range of stencil-based computational workflows. The following paragraphs describe each core element of our stencil IR.

Data types. To represent stencil computations, we will need two custom data types, namely *fields* and *offsets*. A field represents a 2-or 3-dimensional grid containing values of a given base type. This base type can be any of the following: `i1` (boolean value), `i16/32/64` (integer value of a given width) or `f16/32/64` (half, single and double-precision floating-point numbers). Offsets are used when reading a particular cell inside the grid to indicate its relative position to the current cell.

Basic operations. Our IR provides usual arithmetic (`add`, `sub`, `mul`, etc.), comparison (`le`, `gt`, `eq`, etc.) and boolean (`and`, `or`, etc.) operations. We also introduce some convenience operations to compute the minimum and maximum of two values, as well as the square root of a floating-point number.

Control flow. A stencil kernel is executed iteratively on each cell of an input domain. The only control-flow operation that we need is a conditional branching operation. We define it in the form of a high-level `if` construct, to which we attach a code region for the *then* part of the operation and one for the *else* part. The use of regions (brace-delimited code blocks) allows us to have a much higher-level representation of such an operation, compared to the use of conditional jumps and labels in the LLVM IR.

Vertical regions. Many stencil codes operating on a 3-dimensional grid require the same operations to be executed for each layer in the grid. We introduce the concept of a *vertical region*, on operation returning no results and wrapping a region of the IR to be executed for each grid layer. Later on, this operation is lowered to a loop on the vertical axis.

Execution context. To support the use of vertical regions on variable size domains, we provide an operation that enables access to the execution environment of the stencil code. This operation makes it easier for the user to define runtime constants and to access them inside the IR without having to explicitly know their values. A typical use case for such instruction is to retrieve the bounds of a vertical region in a stencil code designed to run at different grid resolution levels.

Field operations. Stencil computations sometimes require the use of a temporary field to store intermediate values defined for the entire iteration domain. The `temp` operation allocates such a field and implicitly marks it as temporary. Optimization passes can then make use of this information to promote the temporary field to a regular field or chose to inline the computations. This can in turn be used to optimize the overall memory layout of the stencil computation [4]. We also provide a `read` operation taking a field F and an offset (i, j) as a parameter and returning the value of $F(i_0 + i, j_0 + j)$, where (i_0, j_0) is the current location in the iteration space. Additionally, a `write` operation, taking a field F and a value v, writes the latter to $F(i_0, j_0)$.

Function calls and global variables. The last set of operations included in our stencil IR includes function calls (`call` operation) and global variable handling. Global variables are used extensively in stencil code to share state between different *stages* of the same stencil kernel. Stages are usually used to split a computation into successive passes, structuring the code and avoiding some data races. We provide three operations for global variable handling, namely `declare_global`, `set_global` and `get_global`.

The operations described above form the basis of complex stencil kernels used, for example, in climate and weather simulation applications.

3.2 Quick syntax overview

This section gives a brief overview of the syntax of our intermediate representation. We will use the example of the 2-dimensional Jacobi stencil described in section 1.1 to illustrate the syntax of the IR. For the sake of the example, we will iterate this stencil for each layer in a 3-dimensional domain whose vertical extent is only known at runtime. The bounds of the vertical domain will be named `kstart` and `kend`. Figure 5 shows the MLIR code representing the computation expressed in equation 2. The `stencil` prefix in front of operations and the `!stencil` prefix in front of types indicates that those are part of the same dialect, namely the *stencil* dialect.

The first lines (up to line 17) define the `@jacobi` function. The attached attribute on line 2 specifies that it is a `stencil.function`, which means that it can be called from a regular function to compute a value at a given point in the grid. The definition itself is a straightforward translation of equation 2. Note that the function operates on a field containing double-precision floating-point

numbers (`f64`) and that it returns an `f64`. The type annotations following some of the operations are used by the integrated type checker to ensure that the operations are applied to well-typed arguments.

```
1  func @jacobi(%G: !stencil<"field:f64">) -> f64
2    attributes {stencil.function} {
3      %off0 = stencil.constant_offset 1 0 0
4      %off1 = stencil.constant_offset -1 0 0
5      %off2 = stencil.constant_offset 0 1 0
6      %off3 = stencil.constant_offset 0 -1 0
7      %0 = stencil.read(%G, %off0) : f64
8      %1 = stencil.read(%G, %off1) : f64
9      %2 = stencil.read(%G, %off2) : f64
10     %3 = stencil.read(%G, %off3) : f64
11     %cst = stencil.constant 0.25 : f64
12     %4 = stencil.add(%0, %1) : f64
13     %5 = stencil.add(%4, %2) : f64
14     %6 = stencil.add(%5, %3) : f64
15     %res = stencil.mul(%6, %cst) : f64
16     return %res : f64
17 }
18
19 func @jacobi_stencil(%in: !stencil<"field:f64">,
20                      %out: !stencil<"field:f64">) {
21     %kstart = stencil.context "kstart" : i64
22     %kend = stencil.context "kend" : i64
23     stencil.vertical_region(%kstart, %kend) {
24         %val = stencil.call @jacobi(%G) : (!stencil<"field:f64">) -> f64
25         stencil.write(%out, %val) : f64
26     }
27     return
28 }
```

Fig. 5: Expressing a simple Jacobi stencil in our intermediate representation.

The stencil kernel is defined from line 19 to line 28. It defines a function taking two fields as arguments. The first one is the input field and the second one is the output field. The bounds of the vertical domain are retrieved by the `stencil.context` operations on lines 21–22 and are then used to define a vertical region on line 23. The value `%val` is computed at each point of the domain and written to `%out`. Type annotations, namely the function type attached to `stencil.call` and the value type attached to `stencil.write` are used to ensure type-correctness.

3.3 Transformation and lowering passes

In addition to defining the syntax and semantics of the stencil operations in our MLIR dialect, we implemented a few transformation and lowering passes. This section gives an overview of those passes and presents the overall compilation pipeline, from a stencil DSL to a relatively low-level intermediate representation suited for conversion to C code.

The overall compilation pipeline is depicted in figure 6. We take a stencil DSL source file as an input and translate it to an IR file. The DSL we use is the one developed by MeteoSwiss and used in conjunction with their `gtclang` frontend, which is based on LLVM `clang`. We adapted it to output MLIR code.

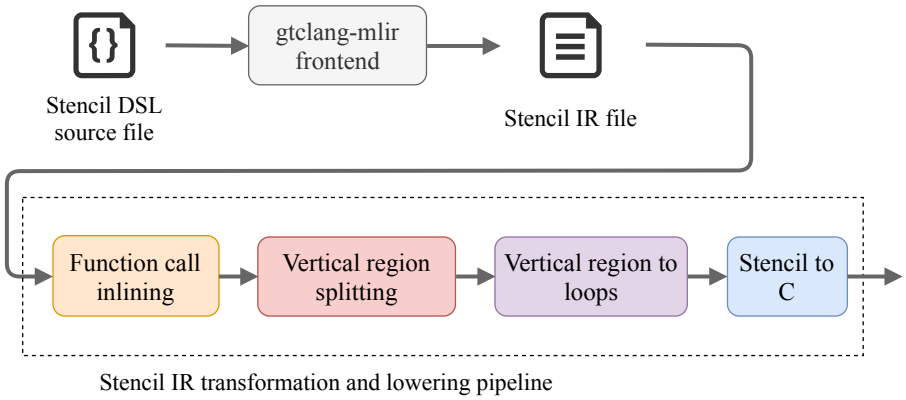


Fig. 6: The stencil DSL compilation pipeline.

The resulting MLIR source file is processed through a series of transformation and lowering passes. Those passes do jot perform any kind of complex optimizations and are only aimed at generating correct executable C code. The implementation of optimization passes will be the subject of future work on this project. The following paragraphs give more details about each individual pass. The passes are presented in the order in which they are executed by the compilation infrastructure.

Function call inlining. The first pass takes care of inlining every function call appearing in the source code. This enables us to get rid of the functions annotated with the `stencil.function` attribute and to generate a monolithic stencil kernel. We take a naive approach and inline every function call recursively, but future versions of our stencil IR compiler might want to introduce a cost model to direct the inlining choices.

Vertical region splitting. The execution of a stencil on a vertical domain may cause data races due to data-dependency relations between `read` and `write` operations. To avoid these kind of interference, we split any given vertical region

after each write operation it contains. The result of this transformation is illustrated in figure 7. The values required to compute the value argument to each `write` operation are copied to the newly created vertical regions to ensure that they are available in the current scope.

```
stencil.vertical_region(%k0, %k1)
{
    // ...
    stencil.write(%out0, %0): f64
    // ...
    stencil.write(%out1, %1): f64
}
```

(a) Before splitting.

```
stencil.vertical_region(%k0, %k1)
{
    // ...
    stencil.write(%out0, %0): f64
}
stencil.vertical_region(%k0, %k1)
{
    // ...
    stencil.write(%out1, %1): f64
}
```

(b) After splitting.

Fig. 7: Splitting vertical regions to avoid data races.

Vertical region to loops. Following the splitting of vertical regions, we introduce a lowering pass. This pass takes vertical regions as an input and outputs *affine loops*. Affine loops are loop constructs defined in the MLIR `affine` dialect. They represent for loops with affine upper and lower bounds and a constant step. By lowering to these operations, we are able to take advantage of the polyhedral compilation concepts integrated into MLIR. Polyhedral compilation techniques can be used to optimize loops by performing data-dependency analyses and adapting loop bounds to expose parallelism [14]. The lowering to affine loops also ensures that the loop bounds in the horizontal dimensions are correct with respect to the elements read by the computational kernel. If necessary, it extends the loop bounds to account for boundary conditions in the access patterns.

Stencil to C. The final lowering pass converts our stencil IR operations as well as the affine loops introduced in the preceding pass to a C MLIR dialect. The C dialect implements a small subset of the features of the C language and expresses them in MLIR syntax. We use this dialect as a bridge between our high-level stencil operations and the C code generation backend described in the next section. This allows us to map each stencil construct to a set of C operations without having to do this transformation during the translation pass run by the code generation backend.

The use of successive transformation and lowering passes is very similar to the way the LLVM optimizer transforms LLVM IR. It makes it easier to develop,

debug and test each pass individually. The development speed is also increased by the use of a human-readable textual intermediate representation.

3.4 Translation to C code

We provide a custom code generation backend that takes an MLIR C dialect as an input and outputs corresponding C code. The code we generate makes used of a very small subset of C which is sufficient to express the computations involved in stencil kernels. The generated C code is then plugged into a GridTools wrapper. This allows us to run the produced stencil C code and test it against the output of a reference implementation written using the GridTools embedded DSL. The next section provides some insights into the results of those tests.

4 Validation

In this section, we show that our stencil IR can express most of the constructs used in a production-level DSL. Section 4.1 looks at the test coverage of our implementation by using the `gtclang` test suite and section 4.2 discusses the applicability of our approach to express and transform production-ready climate simulation code.

4.1 Test coverage

Our code generation backend adds a GridTools wrapper around our stencil C code. This allows us to easily integrate our compilation pipeline in the `gtclang` test suite and to compare the output of our stencils to the one produced by the unmodified MeteoSwiss DSL compilation pipeline. We use the latter as a reference to assert the correctness of our code generation and to assert the extent of our feature coverage.

We achieve a feature coverage of about 90% with respect to available DSL features in `gtclang`. These results were obtained by compiling our generated code and comparing its output to the `gtclang` code generation test cases. We estimate that less than a week of work would be needed to get a feature-complete implementation of a stencil intermediate representation. As a side note, our experiments allowed us to uncover two bugs in the MeteoSwiss DSL compiler.

4.2 COSMO dynamic core

In addition to testing our compiler toolchain against the `gtclang` test suite, we also compiled parts of the COSMO dynamical weather model. The Consortium for Small-scale Modelling (COSMO) develops a weather model for local weather prediction and climate simulation. This model is used in eight countries to provide accurate local-level climate simulation for weather forecast and simulation purposes.

The COSMO model integrates a dynamical core which is used to run stencil computations on weather and climate data. This dynamical core is comprised of 37 stencil kernels. At the time of this writing, we can compile eight of those kernels and successfully run six of them. The remaining 29 kernels all make use of yet unimplemented features, but we plan to support all of them in the very near future.

5 Conclusion and future work

Stencil computations are at the core of numerous numerical applications for which domain-specific languages have been extensively developed. In this report, we showed that we can drastically increase code reuse between those different stencil DSL implementations by making use of a shared compiler intermediate representation and shared optimization passes. MLIR makes it easy to develop new intermediate representations targeted at domain-specific applications. We showed that one can implement a nearly feature-complete stencil IR in a small time frame and that it can be used to run parts of a production-level stencil codebase.

We plan to continue working on this project so as to achieve feature completeness and full test coverage. Additionally, we started to work closely with MeteoSwiss and the CSCS to develop an interoperability layer between our stencil intermediate representation and their custom DSL compiler toolchain. This would allow us to compile and run the entire COSMO dynamical core set of stencils.

Acknowledgements

This work would not have been possible without the collaboration of Nicolas Chappe (ENS Lyon, France), with whom we developed the IR and who adapted `gtclang` to our needs. We would like to thank Tobias Grosser and Tobias Gysi (SPCL, ETH Zurich, Switzerland) for their guidance and advice on this project. We would also like to thank Carlos Osuna, Tobias Wicki and Giacomo Serafini (MeteoSwiss), Hannes Vogt (CSCS) and the MeteoSwiss and CSCS research teams for their valuable feedback and advice. Their technical insights were of great value during the course of this project.

References

1. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D.G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., Zheng, X.: TensorFlow: A System for Large-Scale Machine Learning. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). pp. 265–283. USENIX Association, Savannah, GA (2016), https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

2. Arbona, A., Artigues, A., Bona-Casas, C., Massó, J., Miñano, B., Rigo, A., Trias, M., Bona, C.: Simflowny: A general-purpose platform for the management of physical models and simulation problems. Computer Physics Communications **184**(10), 2321–2331 (oct 2013). https://doi.org/10.1016/j.cpc.2013.04.012
3. Arbona, A., Miñano, B., Rigo, A., Bona, C., Palenzuela, C., Artigues, A., Bona-Casas, C., Massó, J.: Simflowny 2: An upgraded platform for scientific modelling and simulation. Computer Physics Communications **229**, 170–181 (aug 2018). https://doi.org/10.1016/j.cpc.2018.03.015
4. Bianco, M., Cumming, B.: A Generic Strategy for Multi-stage Stencils. In: Lecture Notes in Computer Science, pp. 584–595. Springer International Publishing (2014). https://doi.org/10.1007/978-3-319-09873-9_49
5. Bianco, M., Varetto, U.: A Generic Library for Stencil Computations. CoRR (2012)
6. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: An efficient method of computing static single assignment form. In: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. POPL '89, ACM Press, New York, NY, USA (1989). https://doi.org/10.1145/75277.75280
7. Ford, W.: Numerical Linear Algebra with Applications. In: Ford, W. (ed.) Numerical Linear Algebra with Applications, chap. 20, pp. 469–490. Elsevier, Boston (2015). https://doi.org/10.1016/c2011-0-07533-6
8. Forsyth, D.A., Ponce, J.: Computer Vision: A Modern Approach. Pearson (2002), http://cmuems.com/excap/readings/forsyth-ponce-computer-vision-a-modern-approach.pdf
9. Gysi, T., Osuna, C., Fuhrer, O., Bianco, M., Schulthess, T.C.: STELLA: A Domain-specific Tool for Structured Grid Methods in Weather and Climate Models. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC '15, ACM Press, New York, NY, USA (2015). https://doi.org/10.1145/2807591.2807627
10. Karsai, G., Krahn, H., Pinkernell, C., Rumpe, B., Schindler, M., Vlkel, S.: Design Guidelines for Domain Specific Languages. CoRR (2014)
11. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization, 2004. CGO '04, IEEE, Washington, DC, USA (2004). https://doi.org/10.1109/cgo.2004.1281665
12. Lattner, C., Shpeisman, T.: MLIR: Multi-Level Intermediate Representation for Compiler Infrastructure. EuroLLVM 2019 (Apr 2019), https://llvm.org/devmtg/2019-04/slides/Keynote-ShpeismanLattner-MLIR.pdf
13. Opršal, I., Zahradník, J.: Elastic finite-difference method for irregular grids. GEOPHYSICS **64**(1), 240–250 (jan 1999). https://doi.org/10.1190/1.1444520
14. Pugh, W.: Uniform Techniques for Loop Optimization. In: Proceedings of the 5th international conference on Supercomputing. pp. 341–352. ICS '91, ACM Press, New York, NY, USA (1991). https://doi.org/10.1145/109025.109108
15. Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., Amarasinghe, S.: Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In: Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation. ACM Press, New York, NY, USA (2013). https://doi.org/10.1145/2491956.2462176
16. Roth, G., Mellor-Crummey, J., Kennedy, K., Brickner, R.G.: Compiling stencils in high performance Fortran. In: Proceedings of the 1997 ACM/IEEE conference on Supercomputing. ACM Press, New York, NY, USA (1997). https://doi.org/10.1145/509593.509605

17. Thaler, F., Hoefler, T., Moosbrugger, S., Osuna, C., Bianco, M., Vogt, H., Afanasyev, A., Mosimann, L., Fuhrer, O., Schulthess, T.C.: Porting the COSMO Weather Model to Manycore CPUs. In: Proceedings of the Platform for Advanced Scientific Computing Conference. PASC '19, ACM Press, New York, NY, USA (2019). https://doi.org/10.1145/3324989.3325723