Efficient GPU Computations of a Parallel Model for Stencils

Jean-Michel Gorius

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, 09/09/19–12/06/19

Abstract. Weather and climate models make extensive use of stencil computations to model physical phenomena. These models have grown in complexity over the past decades and make up large parts of today's HPC workloads. With new architectures switching to heterogeneous compute nodes, climate models need to be compiled and optimized for accelerator hardware. In this report, we present a compiler toolchain based on MLIR specifically targeted at compiling stencil code for GPU accelerators. We illustrate how we leverage the multi-level capabilities of MLIR by representing stencils in a high-level intermediate representation and subsequently lower this representation down to GPU code. We show that our toolchain can compile the dynamical core of the European weather and climate model.

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

1 Introduction

Climate science and weather simulation have gained a lot of attention during the past decades. With more and more concerns being raised about global warming and its implications, weather and climate modelling need to become more and more accurate and reliable. The need for increased simulation accuracy has led to the development of climate models targeting large-scale high-performance computing (HPC) infrastructures. This section introduces the field of climate and weather modelling (section 1.1) before discussing *domain-specific compilers* designed to compile climate models for heterogeneous hardware (section 1.2).

1.1 Climate and weather modelling

To simulate climate and weather, scientists rely heavily on computational models aimed at solving partial differentials equations (PDE) using numerical methods.

Over the past decades, the complexity of these models has been steadily increasing alongside the available computational power, going from simple atmospheric models [14] to fully-coupled models [9] that take into account interactions between land, ocean, human activity and the atmosphere.

Modern climate models are based on numerical methods such as finite differences [5], finite volumes [18], and finite elements [6]. These methods use stencil computations—element-wise computations on a grid accessing only a fixed neighborhood of each grid cell—to compute approximate solutions to PDEs for each simulation time step. The rapid increase in available compute power during the last decades has enabled scientists to use very high-resolution grids to solve increasingly complex differential systems. However, with the end of Dennard scaling and the slow down of Moore's law, HPC systems have started to move towards heterogeneous compute nodes, with multi-core CPUs coupled to specialized accelerators. With the current and next generation of supercomputers [20, 16] providing several specialized hardware accelerators per node, the main target for high-performance climate and weather models is switching to accelerators like GPUs [8, 3].

1.2 Domain-specific compilers

The move towards heterogeneous hardware leads to the introduction of a new parallel programming model, with new opportunities for target-specific optimizations. On the other hand, the high-level concepts expressed in climate models written and developed by domain scientists are target-agnostic, as they aim at solving problems at a higher level of abstraction. Several domain-specific compilers have been developed to bridge the gap between high-level domain-specific concepts and low-level target-specific optimizations. This section gives a brief overview of a few of these domain-specific compilers.

The Stencil Loop Language (STELLA) [12] introduces a high-level domain-specific language (DSL) along with a compiler frontend. The STELLA frontend takes the aforementioned 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. STELLA has been used for production-level weather forecasting and modelling by the Swiss National Supercomputing Center (CSCS) and the Swiss Federal Office of Meteorology and Climatology (MeteoSwiss) for many years.

Designed as a successor to STELLA, GridTools [19] takes a different approach and embeds a DSL directly in C++. GridTools makes heavy use of template metaprogramming to transform the input code at compile-time, choosing the appropriate data layout for the target architecture and providing high-level abstractions for expressing complex stencil programs. To further lift the level of abstraction at which the user has to work, MeteoSwiss together with CSCS, ETH Zurich and Vulcan Inc. have invested in the development of GtClang/Dawn [15], a source-to-source compiler based on GridTools which provides a high-level DSL designed to express climate stencil codes. The use of STELLA, GridTools and

GtClang/Dawn requires climate models to be written in a DSL, whereas many climate models are written in Fortran, with some existing models reaching several million lines of code.

The large Fortran codebase of many climate models is built on top of decades of legacy algorithms, developed and maintained by domain scientists. The CLAW compiler [4] aims at easing the transition to heterogeneous compute nodes by retaining Fortran as a base language and by providing a set of additional compiler directives to be inserted in code. This approach makes it easy to port existing code to new architectures without the need for deep code changes. Moreover, by choosing to keep Fortran as a frontend language, the CLAW compiler focuses on ease of adoption by a scientific community which has decades of experience writing code in Fortran.

2 A multi-level compiler for climate stencils

Numerous domain-specific compilers have been developed for compiling stencil code, with little or no code reuse at all. This creates an additional maintenance cost for compilers targeting production-ready applications. Our work aims at providing a common baseline for stencil compiler development and experimentation centered around principles borrowed from the LLVM project. In the following section, we present a multi-level compiler infrastructure built on top of the MLIR [13] framework. Our project builds upon lessons learned from previous work [10] and aims at exploring new approaches for representing stencils in a compiler IR.

2.1 A high-level intermediate representation for stencils

The following sections present a high-level intermediate representation developed as an MLIR *dialect*. This IR is designed to be able to represent *stencil programs* comprised of several calls to *stencil functions*, as is customary in climate and weather models. By basing our work on MLIR, we make it easy to extend our IR to accommodate other use cases. After describing how to express the iteration domain of a stencil computation in our IR, the following sections focus on common operations used to describe stencil programs, and on choices that we have made to represent these in a compiler. We illustrate two main algorithmic motifs that appear in climate code before discussing some of the limitations of our approach. Section 2.2 focuses on the compilation pipeline, and especially on lowering our high-level stencil IR all the way down to GPU code.

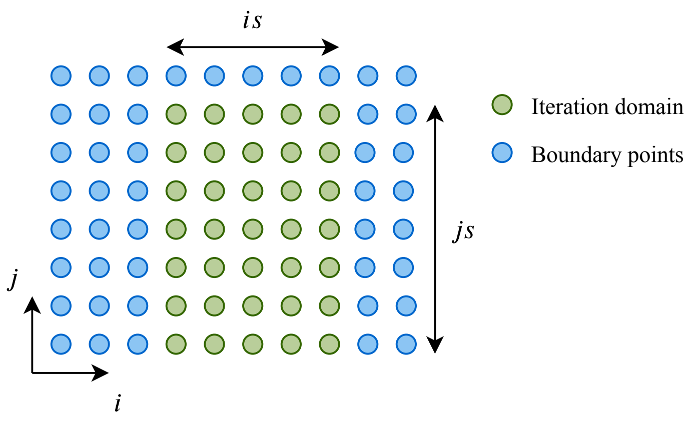
Defining iteration domains and fields Typical stencils can operate on regular grids [2] as well as on irregular grids [17]. Our work focuses primarily on the regular case, which is at the core of many climate models. By restricting ourselves to regular structures, we are able to infer more information about the compiled code, and thus leverage more optimization paths when generating code for GPUs.

Climate models often operate on 3-dimensional fields of scalar values. The following examples will use 2-dimensional fields to make their representation easier.

We represent the regular grids on which stencils operate by defining a custom IR type, `!stencil.field`. MLIR allows types to be parameterized by compile-time symbols and constants, as well as by other types. Our field type is parameterized by the type of the elements it contains. Figure 1 summarizes the description of our field type and illustrates the correspondence between the IR definition and the abstract representation of a field. To define such a field, we introduce a custom operation which creates a field instance given a name, an *iteration domain* and a *boundary*. The boundary is a set of points that are added to the iteration domain to account for the shape of the stencil access pattern, effectively preventing out-of-bounds accesses.

```
%is, %js = stencil.domain : index, index
%in = stencil.field "in" %is %js plus (-3,2) (0,1) : !stencil.field<f64>
```

(a) Defining a field in our IR.



(b) Resulting field representation.

Fig. 1: Field definition.

Fields are used as input and output parameters for each stencil program. To give the compiler more flexibility and to make the flow of data between stencils more explicit, we introduce a second type, `!stencil.view`. A view is an immutable reference to a subpart of a field. It can be read from, but never written to. Similarly to fields, views are parameterized by an element type. Figure 2 shows an example of a view defined on a field. Note that views do not have boundaries but can be defined to cover arbitrary hyper-rectangular parts of a field, including its boundary.

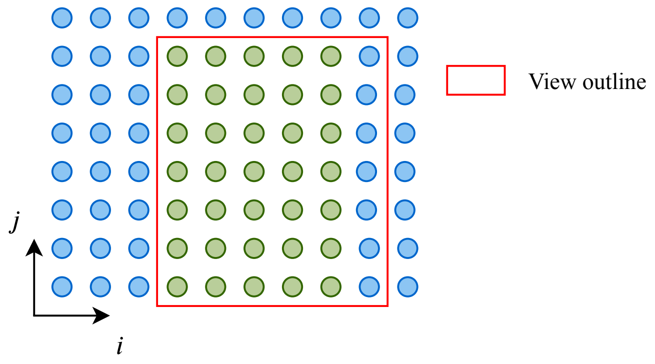
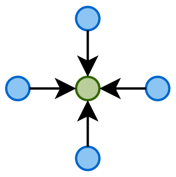


Fig. 2: Outline of a view defined on the field of figure 1.

Stencil kernels Stencil code is organized around a set of kernels that are executed on multiple fields to produce one or more outputs. In our IR, we represent individual kernels by functions operating on views. Figure 3 gives an example of such a function for a simple Laplace operator kernel. Individual kernels access their parameters using the `access` operation, which takes a view as well as an offset and returns the corresponding element. Offsets are always defined relatively to the current iteration position. This behavior is similar to kernels written in CUDA C, where each thread can access data relative to the current thread and block indices.

```
stencil.func @lap(%in : !stencil.view<f64>) -> f64 {
  %0 = stencil.access %in[ 0, 0] : f64
  %1 = stencil.access %in[ 1, 0] : f64
  %2 = stencil.access %in[ 0, 1] : f64
  %3 = stencil.access %in[-1, 0] : f64
  %4 = stencil.access %in[ 0,-1] : f64
  %cst = constant 4.0 : f64
  %5 = addf %1 %2 : f64
  %6 = addf %3, %4 : f64
  %7 = addf %5, %6 : f64
  %8 = mulf %cst, %7 : f64
  %9 = subf %0, %8 : f64
  stencil.return %9
}
```

(a) Defining the Laplace operator.



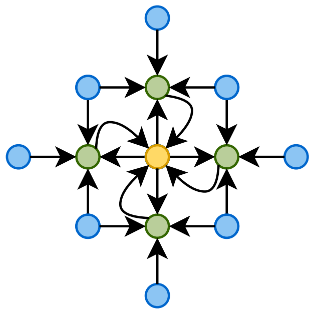
(b) The Laplace operator pattern.

Fig. 3: A simple Laplace operator.

To increase the composability of our stencil kernels, we introduce a `call` operation, which applies a given kernel on one or more views, but by shifting the current iteration position by a given offset. An example where this pattern can be useful is given in figure 4. By using `call`, we can reuse the Laplace kernel defined in figure 3 and build a more complex access pattern out of it. This is a common idiom in stencil code and especially in climate code, where an atmospheric parameter is computed using the same stencil kernel on all the neighbours of the current iteration position.

```
stencil.func @laplap(%in : !stencil.view<f64>) -> f64 {
  %0 = stencil.access %in[0, 0] : f64
  %1 = stencil.call @lap(%in)[ 1, 0] : f64
  %2 = stencil.call @lap(%in)[ 0, 1] : f64
  %3 = stencil.call @lap(%in)[-1, 0] : f64
  %4 = stencil.call @lap(%in)[ 0,-1] : f64
  %cst = constant 4.0 : f64
  %5 = addf %1 %2 : f64
  %6 = addf %3, %4 : f64
  %7 = addf %5, %6 : f64
  %8 = mulf %cst, %7 : f64
  %9 = subf %0, %8 : f64
  stencil.return %9
}
```

(a) Laplace operator composition.



(b) Resulting access pattern.

Fig. 4: Composing stencils.

Writing stencil programs Iteration domains, fields and kernels are the basic building blocks used to define stencil programs. The usual flow of such a program is as follows: load data from the input fields, apply a series of kernels on the input data, merge the results and store them to the output fields. Figure 5a shows an example of a simple stencil program that computes values sequentially on an input field, and figure 5b illustrates how we can compute different values at the bottom, in the middle and at the top of the iteration domain.

To load data from an input field and create a view on it, our IR introduces a `load` operation. This operation creates a handle to the input field, which will be used by subsequent computations on this field. Note that we do not specify any dimensions or offset for the view. Usually, when writing stencil kernels, the user only knows the size of the desired output. We provide a compiler analysis pass that infers the shapes and offsets of each view in a stencil program based on the desired output shape. This gives our compiler a lot of flexibility when it comes to splitting or merging computations, as it can split and merge input

```
%1 = stencil.load %in : !stencil.view<f64>
%2 = stencil.apply @first_kernel(%1) : !stencil.view<f64>
%3 = stencil.apply @second_kernel(%2) : !stencil.view<f64>
stencil.store %3 to %out[0:%is,0:%js]
```

(a) Applying stencils sequentially.

```
%1 = stencil.load %in : !stencil.view<f64>
%t = stencil.apply @top(%1) : !stencil.view<f64>
%m = stencil.apply @middle(%1) : !stencil.view<f64>
%b = stencil.apply @bottom(%1) : !stencil.view<f64>
%5 = stencil.combine %t[0:%is,%js-2:%js],
                     %m[0:%is,3:%js-2],
                     %b[0:%is,0:3] : !stencil.view<f64>
stencil.store %5 to %out[0:%is,0:%js]
```

(b) Splitting the application domain.

Fig. 5: Example stencil programs.

views depending on the most efficient data representation. The output shape is defined in the `store` operation, which takes a view to store to a given output field and targeting a given domain size. The domain is defined using a syntax similar to Python ranges, where the form `[b:e]` defines the half-open integer range $[b, e)$.

Applying stencil kernels to a set of input fields is handled by the `apply` operation. This operation takes a kernel name (or *symbol*) as an input, as well as a list of arguments to pass to the kernel. We can think of `apply` as being similar to a CUDA kernel call: it abstracts away a loop nest that applies the kernel on the entire computation domain and returns a view containing the result of the stencil computation at each iteration point.

The last operation that appears in figure 5b is `combine`. The `combine` operation takes a list of views associated with domains and *combines* them into a single view. Conceptually, `combine` can be seen as building a view V from several blocks forming a partition of V. Figure 6 shows the view produced by the call to `combine` in figure 5b.

Dealing with vertical dependencies Climate models rely on two main algorithmic motifs: *horizontal stencils* and *vertical solvers.* In the context of climate modelling, a stencil is an element-wise computation that accesses a fixed set of neighbours, with *no vertical dependencies.* On the other hand, vertical solvers are used to solve tridiagonal systems of PDEs and operate on the vertical dimension of the atmosphere. They introduce vertical dependencies between multiple computation layers. Even though they operate in essentially the same way as

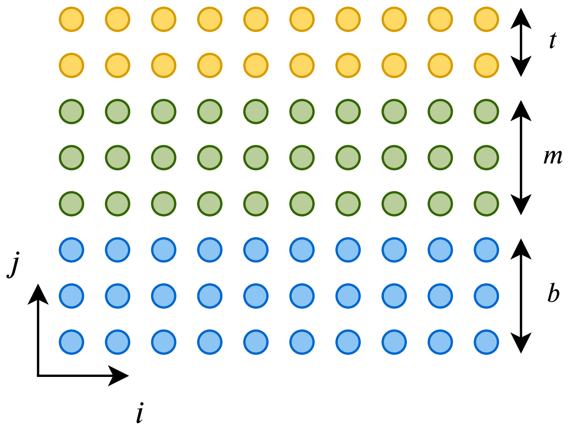


Fig. 6: Combining multiple views.

stencils, solvers often receive a special treatment in climate code, as they cannot easily be parallelized.

We chose to make the use of vertical solvers explicit in our IR by introducing a sequential loop construct named `for`. An example of vertical solver can be found in figure 7. The `for` operation introduces an iteration variable named `%j` that traverses the range `[1:%js]`. The iteration variable is used in the body of the `for` operation to extract layers from views using the `combine` operation. The `(%a=%2)` construct that precedes the operation body introduces an *accumulator* view. This view is initialized with the content of `%2` and is then used to carry out the computation for one iteration. At the end of the `for` operation's body, the `yield` operation yields a new value for the accumulator, which is used in the next iteration.

```
%1 = stencil.load %inout : !stencil.view<f64>
%2 = stencil.apply @bottom(%1) : !stencil.view<f64>
%3 = stencil.for [%j=1:%js] (%a=%2) {
  %b = stencil.apply @main(%1, %a) : !stencil.view<f64>
  %c = stencil.combine
      %a[0:%is,0:%js]
      %b[0:%is,%j:%j+1] : !stencil.view<f64>
  stencil.yield %c
}
stencil.store %3 to %inout[0:%is,0:%js]
```

Fig. 7: Example of a vertical solver.

Discussion The operations described in the previous section form a dialect in MLIR that is suitable to represent a vast majority of commonly used stencil codes. However, it comes with a few limitations. By focusing on climate stencil code, we make some assumptions about the computational patterns that can arise in the compiled code. For example, we implicitly assume that stencil kernels operate solely on their input parameters and return one or more views. This is the most common use case for stencils, but it can sometimes be useful to alter some kind of global state inside of a kernel. We deliberately chose not to handle this case, as it makes data flow analysis much harder by introducing side effects.

Another limitation arises from the way we chose to handle domain sizes. By inferring the shapes of the views used in stencil programs from the shape of the output fields, we assume that the input fields are large enough for the computation to be carried out without any out-of-bound accesses. This choice shifts the responsibility of ensuring that enough input memory is allocated to the user. We chose to not enforce this requirement in our compiler, as stencil code is often meant to be interfaced with some external code that provides the input and output fields and handles the general control flow for the entire computation. In the future, we might envision adding support for control flow operations in our dialect as well as facilities to allocate memory, effectively enabling the compilation of full climate models, including their control logic.

2.2 Compiling stencil code for accelerators

The MLIR infrastructure is well-suited for domain-specific applications, allowing users to define their own intermediate representations and making it easy to work at multiple levels of abstractions in the same compiler IR. We take advantage of this approach by progressively lowering the stencil dialect defined in section 2.1 to GPU code while applying multiple rounds of optimizations at various levels of abstraction, effectively writing *trans-abstraction* optimization passes.

The design of our stencil IR allows us to operate on very high-level concepts and to optimize them without having to deal with low-level target-specific details. We can therefore reorder stencil kernels, fuse or split them, or inline function calls while staying at a high-level of abstraction. Once those high-level optimizations are applied, we *lower* our stencil IR to a mix of *affine loops* and basic stencil operations. Affine loops are provided by MLIR in the form of yet another dialect: the `affine` dialect. The latter provides a simple abstraction to model polyhedral transformations on programs. Contrary to other polyhedral optimization frameworks such as Polly [11], MLIR takes a top-down approach on polyhedral transformations: where tools like Polly try to recover high-level information from the compiler IR (*e.g.* the LLVM IR), MLIR allows the compiler to work on a high-level loop representation before going down to a more low-level IR. This approach makes it easier to work on loop patterns, which are explicitly represented in the IR by loop operations, rather than trying to recover a loop structure from a set of low-level operations. Going down from the stencil dialect to the affine dialect enables us to reuse the optimization passes developed by the

MLIR team to process loops, such as loop tiling, loop fusion and loop-invariant code motion.

The affine dialect and the remaining stencil operations are then translated to so-called standard operations, which are part of MLIR's core `standard` dialect. This dialect models low-level concepts and basic operations such as integer and floating-point arithmetic as well as memory access through a set of minimal abstractions called *memory references*. At this point in the compilation pipeline, the high-level information encoded in our stencil dialect has been entirely translated to low-level operations. We can now fully take advantage of the existing lowering, transformation and optimization passes provided by MLIR to target either CPUs by using the provided conversion from standard operations to LLVM IR, or GPUs by lowering to the MLIR GPU dialect. We chose GPUs as our main target, as the capabilities of such accelerators can be leveraged to execute massively parallel stencil computations such as the horizontal stencils found in climate models. Once the relevant standard operations are converted to operations from the GPU dialect, MLIR offers to target NVIDIA hardware by translating the MLIR GPU dialect to LLVM NVVM operations, or to target AMD hardware by using the LLVM ROCDL support. We also wrote a custom CUDA C backend for MLIR to make it easier to see and debug our compilation pipeline. Additionally, this backend is used in the following section to compare our code generation against the GtClang/Dawn compiler output.

3 Validation

In order to validate our design and to set clear performance goals for future developments of our toolchain, we developed a proof-of-concept compiler based on the ideas described in the past sections. In this section, we demonstrate that our toolchain is able to compile the entire COSMO dynamical core [1]. We further give some insights into the current performance of our compiler by comparing it to the GtClang/Dawn infrastructure described in section 1.2. GtClang/Dawn is currently used by MeteoSwiss, CSCS and Vulcan Inc. to compile and run the COSMO dynamical core on the Piz Daint supercomputer [7].

To be able to compile the COSMO model, which is implemented in the GtClang/Dawn DSL, we implemented an adaptor that translates from the GtClang/Dawn internal representation to a prototype version of our stencil IR. This IR models a subset of the concepts we describe in section 2.1 and is intended to be entirely replaced by the latter in the very near future.

The dynamical core of the COSMO model is comprised of a large set of stencil kernels, ranging from horizontal diffusion computations to large-scale vertical solvers computing vertical atmospheric advection phenomena. Figure 8 show the relative performance of our generated GPU code compared to the GtClang/Dawn output. All the measurements were realized on a common atmospheric grid size of 128x128x80 cells using the `nvprof` profiler with CUDA 10.2 on a NVIDIA Quadro M1200 GPU (Maxwell architecture), and by using the CUDA backend of both compilers.

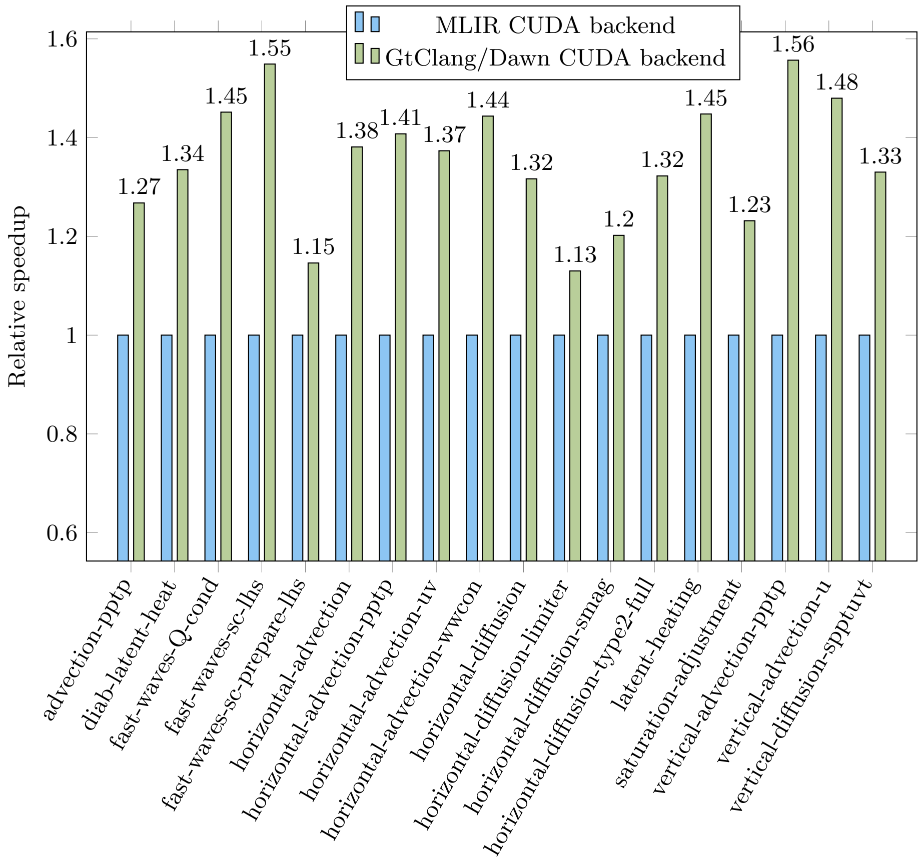


Fig. 8: Relative speedup of the GtClang/Dawn output compared to our MLIR implementation.

The plot in figure 8 shows the expected speedup we could gain by implementing the optimizations performed by the GtClang/Dawn toolchain. We plan to implement these optimizations in subsequent iterations of our toolchain. The biggest difference between our toolchain and the GtClang/Dawn compiler is the way we handle vertical dependencies. In its current state, our compiler separates each layer of the vertical computation in its own GPU kernel, which makes the execution of vertical solvers sequential. On the other hand, GtClang/Dawn detects vertical solvers that can be executed in parallel and generates a fused kernel that executes most of the computation in parallel. Only the remaining dependent code segments are executed sequentially. A speedup of more than 30% in figure 8 is almost always the symptom of vertical solvers that can be partially executed in parallel.

Another important class of optimization that our compiler does not perform at the time of this writing is memory layout transformations and shared memory allocation. Stencil computations in climate code are often memory-bound, and we expect memory optimizations to provide an additional 30% speedup over the current execution time. In the future, we plan to implement data layout transformation passes to generate efficient coalesced memory accesses as well as passes to select the type of memory each field should be stored in for a given part of the computation (global memory vs. shared memory). To further optimize memory accesses, we plan to introduce *register buffering*. This optimization makes use of a circular buffer of registers to store values retrieved from accesses to sequential elements in memory by multiple iterations of a loop. For example, instead of reading the value of a field at indices `i` and `i+1` from memory at each iteration, we read the element at index `i+1` and store it in a buffer of two registers, shifting the contents of the latter by one. By doing so, we divide the number of memory accesses by a factor of two.

Our work lays down the foundations for the development of a multi-level stencil compiler for climate, which opens up several routes for further improvement of the optimization process at many different levels of abstraction.

4 Conclusion and future work

Stencils are at the core of many computational domains. Several domain-specific compilers targeted at optimizing stencil codes have been developed, often in close relationship to a domain-specific language. In this report, we presented a frontend-agnostic compiler infrastructure for compiling stencil code targeting GPU accelerators. By representing stencil constructs in a high-level compiler intermediate representation, we are able to optimize code using domain knowledge before going down to lower-level IRs. This approach allows us to leverage trans-abstraction optimizations which can tailor high-level constructs to the final target machine with great flexibility. We developed a proof-of-concept toolchain which is able to compile the dynamical core stencils of the European climate model, and we set a clear goal for performance improvements and optimization development.

Future work will focus on finalizing the transition from our prototype IR to the IR described in section 2.1 as well as on the implementation of high-level optimization passes. We also expect low-level passes to be improved in the near future as the needs of the MLIR community evolve to more HPC-focused applications. By building on top of the MLIR infrastructure, we will directly benefit from these improvements in our toolchain without having to make important changes. Eventually, the modular nature of our compiler toolchain will allow for an easy exploration of new target architectures and optimization paths, *e.g.* by using machine learning to select optimizations based on the input stencil and the target platform.

Acknowledgements

First and foremost, I would like to thank my supervisor, Tobias Grosser (SPCL, ETH Zurich, Switzerland), for his insights during the development of this project, for his advice on research but also on team management, and for enabling me to present our work at the MLIR4HPC workshop in Atlanta, GA. Large parts of this project have been developed in close collaboration with Tobias Gysi (SPCL, ETH Zurich, Switzerland), without whom our stencil dialect would probably not look the same. Many thanks to him for taking part in the design of the stencil IR and for bearing with me during our countless design meetings. I would like to thank Tobias Wicky (Vulcan Inc.), Hannes Vogt (CSCS) and Matthias Roethlin (MeteoSwiss) for their technical support and insights regarding GtClang and Dawn, and for their attendance during the MLIR workshop that we organized at ETH. Finally, I would like to thank the management teams at Vulcan Inc., MeteoSwiss and CSCS for enabling such a collaboration, particularly Oliver Fuhrer (Vulcan Inc.), Carlos Osuna (MeteoSwiss) and Mauro Bianco (CSCS).

References

1. Baldauf, M.: The COSMO model: towards cloud-resolving NWP. In: Seminar on Recent Developments in Numerical Methods for Atmosphere and Ocean Modelling, 2-5 September 2013. pp. 107–121. ECMWF, ECMWF, Shinfield Park, Reading (2014)
2. Bianco, M., Varetto, U.: A Generic Library for Stencil Computations. CoRR (2012)
3. Brown, N., Nigay, A., Weiland, M., Hill, A., Shipway, B.: Porting the microphysics model CASIM to GPU and KNL Cray machines. In: Proceedings of the Cray User Group (2017)
4. Clement, V., Ferrachat, S., Fuhrer, O., Lapillonne, X., Osuna, C.E., Pincus, R., Rood, J., Sawyer, W.B.: The CLAW DSL: Abstractions for Performance Portable Weather and Climate Models. In: PASC (2018)
5. Coiffier, J.: Fundamentals of Numerical Weather Prediction. Cambridge University Press (2011)
6. Cotter, C.J., Shipton, J.: Mixed finite elements for numerical weather prediction. Journal of Computational Physics **231**(21), 7076–7091 (Aug 2012)
7. CSCS, ETH Zürich: "Piz Daint", one of the most powerful supercomputers in the world. Available at https://www.cscs.ch/computers/piz-daint/ (2019/12/10)
8. Cumming, B., Osuna, C., Gysi, T., Bianco, M., Lapillonne, X., Fuhrer, O., Schulthess, T.C.: A Review of The Challenges and Results of Refactoring the Community Climate Code COSMO for Hybrid Cray HPC Systems. In: Proceedings of the Cray User Group (2013)
9. Delworth, T.L., Rosati, A., Anderson, W., Adcroft, A.J., Balaji, V., Benson, R., Dixon, K., Griffies, S.M., Lee, H.C., Pacanowski, R.C., Vecchi, G.A., Wittenberg, A.T., Zeng, F., Zhang, R.: Simulated Climate and Climate Change in the GFDL CM2.5 High-Resolution Coupled Climate Model. Journal of Climate **25**(8), 2755–2781 (2012)
10. Gorius, J.M.: Modeling Stencils in a Multi-Level Intermediate Representation (2019)

11. Grosser, T., Groesslinger, A., Lengauer, C.: Polly — Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. Parallel Processing Letters **22**(04) (2012)
12. 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)
13. Lattner, C., Shpeisman, T.: MLIR: Multi-Level Intermediate Representation for Compiler Infrastructure. EuroLLVM 2019 (Apr 2019)
14. Manabe, S., Wetherald, R.T.: Thermal Equilibrium of the Atmosphere with a Given Distribution of Relative Humidity. Journal of the Atmospheric Sciences **24**(3), 241–259 (1967)
15. MeteoSwiss, CSCS, ETH Zürich, Vulcan Inc.: GTClang and Dawn. Available at https://github.com/MeteoSwiss-APN/dawn
16. Oak Ridge National Laboratory: Frontier spec sheet. Available at https://www.olcf.ornl.gov/wp-content/uploads/2019/05/frontier_specsheet.pdf (2019/12/10)
17. Opršal, I., Zahradník, J.: Elastic finite-difference method for irregular grids. GEOPHYSICS **64**(1), 240–250 (jan 1999)
18. Putman, W.M., Lin, S.J.: Finite-volume transport on various cubed-sphere grids. Journal of Computational Physics **227**(1), 55–78 (2007)
19. 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)
20. The Top500 Project: Top500 lists. Available at https://www.top500.org/lists/top500/ (2019/12/10)