GGDML: icosahedral models language extensions

Article

Accepted Version

GGDML: icosahedral models language extensions

Nabeeh Jumah[1], Julian Kunkel[2], Günther Zängl[3], Hisashi Yashiro[4], Thomas Dubos[5], and Yann Meurdesoif[6]

1 Universität Hamburg, Jumah@informatik.uni-hamburg.de
2 Deutsches Klimarechenzentrum, kunkel@dkrz.de
3 Deutscher Wetterdienst
4 RIKEN Advanced Institute for Computational Science
5 École Polytechnique
6 Laboratoire des sciences du climat et de l'environment

Abstract

The optimization opportunities of a codebase are not completely exploited by compilers. In fact, there are optimizations that must be done within the source code. Hence, if the code developers skip some details, some performance is lost. Thus, the use of a general-purpose language to develop a performance-demanding software -e.g. climate models- needs more care from the developers. They should take into account hardware details of the target machine.

Besides, writing a high-performance code for one machine will have a lower performance on another one. The developers usually write multiple optimized sections or even code versions for the different target machines. Such codes are complex and hard to maintain.

In this article we introduce a higher-level code development approach, where we develop a set of extensions to the language that is used to write a model's code. Our extensions form a domain-specific language (DSL) that abstracts domain concepts and leaves the lower level details to a configurable source-to-source translation process.

The purpose of the developed extensions is to support the icosahedral climate/atmospheric model development. We have started with the three icosahedral models: DYNAMICO, ICON, and NICAM. The collaboration with the scientists from the weather/climate sciences enabled agreed-upon extensions. When we have suggested an extension we kept in mind that it represents a higher-level domain-based concept, and that it carries no lower-level details.

The introduced DSL (GGDML- General Grid Definition and Manipulation Language) hides optimization details like memory layout. It reduces code size of a model to less than one third its original size in terms of lines of code. The development costs of a model with GGDML are therefore reduced significantly.

Keywords: DSL; Source-to-source translation; Fortran; Software development; Icosahedral model

1 Introduction

The compilers of general-purpose languages apply many optimizations while compiling a code repository. However, there are some optimization decisions that those compilers cannot make on behalf of the programmers as the abstraction level of the general-purpose languages is limited. Furthermore, they cannot handle some optimizations without an external guidance. Hence, some opportunities to achieve a higher performance are lost.

On the other hand, providing the optimization decisions by the programmers needs an expertise in many programming details which are far from the scientific domain knowledge. In this case, the scientists would need to learn many skills besides to their domain science. Additionally, applying the optimization within the source code harms the code structure and the understandability of it and increases the effort for any subsequent maintenance or code modifications.

In climate model development, the models' demand for performance is steadily increasing. The model developers need to exploit the hardware features to cope with this demand. The manual optimization of the source code in models with hundreds of thousands of lines of code is a tough mission. Supporting multiple target machines will harm the structure of the model's code because of the multiple optimization details for the various supported machines. Thus, it is important to develop the suitable language/tools to improve the software engineering of climate modeling. In this article, we focus on the development of GGDML (General Grid Definition and Manipulation Language), which is a higher level domain-specific language in which we provide language extensions to support climate modeling. In GGDML, we provide support for climate model development, and focus on support for icosahedral models, which have not been widely considered in software engineering. GGDML is developed under the AIMES project to offer higher level code design for the three icosahedral models DYNAMICO [1], ICON [2], and NICAM [3].

As a main contribution of this paper, we provide a set of language extensions to the Fortran language, which improves the software engineering of (icosahedral) climate models. The technique itself is not restricted to the Fortran language, in fact, it can be applied to other languages, too.

In this article, we provide an introduction to computation in icosahedral models in Section 1.1. A review on the related work is given in Section 1.2. In Section 2 we describe the software engineering of a model using GGDML. In Section 3 we discuss the language extensions, we start with a discussions of the development process, and then describe the extensions and the functions they provide, and finish the section with code examples that use GGDML to write some sample codes from the three models. In Section 4 we discuss the impact of using GGDML compared to original Fortran code. A discussion and conclusion of the work and a look on the future work are discussed in Section 5.

1.1 Computation in Icosahedral Models

In climate models, grids are essential abstractions for the development of the codes computing the model's variables. Grids are used to discretize the space over which the variables are measured/calculated. In various models, the developers use different kinds of grids. Some

models use structured grids, which simply address data by the Euclidean space coordinates with longitudinal and latitudinal surface dimensions. However, there are some shortcomings of such grids, which limit the model's capability to provide certain functionalities. For example, a rectangular grid for the whole earth surface contains rectangles with different sizes, which vary according to the location of the rectangle. The need of some models to offer some functions which cannot be considered with structured grids, led to go beyond such grids. Among those models are icosahedral models.

An icosahedral model is one that uses an icosahedral grid, which models the earth surface by an icosahedron. The faces of an icosahedron are further divided into smaller triangles repeatedly to a level that is enough to provide an intended resolution. Further refinements for some triangles allow for nested grids, which provide higher resolution for specific regions on the globe. ICON for instance exhibits such capability, which is not the case for simple structured grids.

In icosahedral grids, hexagons can be synthesized (with a few pentagonal areas). Thus we see icosahedral models with either triangular or hexagonal grids. The model's variables are defined with respect to the grid. They can be defined at the centers of the grid's cells, on the edges of the cells, or at their vertices (Figure 1).

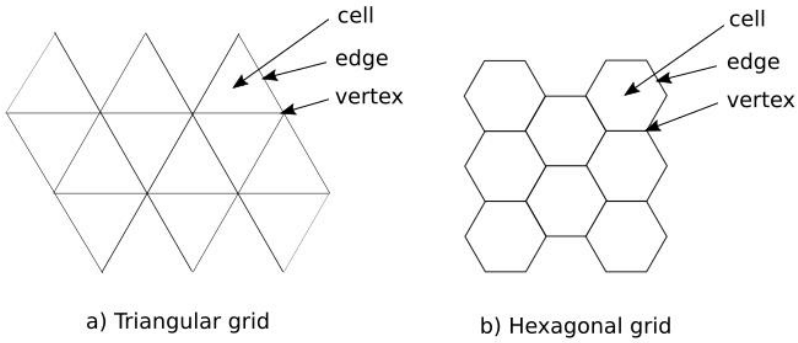

Figure 1: Icosahedral grids and variables

Moving from the structured grids to the icosahedral grids allows a model to provide new functionalities such as multigrids. However, it complicates the storage of the variables' data. N-dimensional arrays do not directly support the storage of the icosahedral-grid-bound variables like they do in the structured grids. A transformation function is then needed to address the variables. HEVI methods with a space-filling curve (e.g. Hilbert space-filling curve) for the horizontal surface enable the data addressing. Considerations like caching affect the choice of the space-filling curve.

1.2 Related work

Many research efforts have been done to improve the modeling process, but mostly they were directed towards solving the problem of performance portability, with less focus on code complexity, readability or maintainability. Some solutions comprise low-level and technical details. The approaches range from using domain libraries, to compiler directives and annotations, to general-purpose language embedded DSL constructs like C++ template

programming, to standalone DSLs that replace general-purpose languages, and finally to language extensions.

Library approaches provide high-level functions to achieve performance portability for the domain computations. The solutions [4] [5] [6] provide libraries that support regular structured grids. Tangram [7] provides a data-structure-based library which allows the programmers to explicitly specify the optimizations through using rewriting-rules within the source code. The solutions [8] and [9] moved to use the concept of active libraries, in which the code is generated during the code translation process.

Source code preprocessing based solutions use compiler directives to annotate the parts of the code that are preprocessed before being submitted to the backend compiler. Those solutions use a special front-end compiler/preprocessor to process annotated code. The solutions [10] [11] [12] [13] use the idea of annotating the source code. The annotations within the source code drive the optimization process. Gung Ho [14] separates the scientific code into high-level operations (the algorithm layer) and low-level operations that explicitly compute with the data (kernels). In between sits a layer of auto-generated code, driven partly by directives, that handles looping over data and attempts to optimize performance for different architectures and parallelization strategies. In general, with directive-based solutions, the model developers will need to care about providing lower-level (optimization) details within the source code.

General-purpose language embedded constructs, like templates in C++ or regular expressions are used in some solutions. Domain code takes benefit of higher-level abstractions built with such constructs. Lower-level implementations provide performance for a specific platform. The solutions [15] [16] [17] [18] use C++ constructs to write the model code which is translated into an architecture-optimized code. GridTools [19] generalize Stella [16] and add support for other grid types. In addition to C++, Gridtools support the translation of regular stencil code in Python into C++ Gridtools code. The generic programming with templates in the C++ language gives a strong tool, but unfortunately such feature does not exist in some modeling languages like the Fortran language.

Standalone DSLs like [20] and [21] specify language constructs in a domain-specific language that provides a new syntax which replaces general-purpose languages. The compilation of such a DSL's code generates code for different architectures. These DSLs must support further language features like expressions, operators, and may cover program flow and control. Such solutions require modification to existing compilers or creation of a new language compiler and force users to rewrite kernels completely with the new syntax. An additional effort is needed to integrate the generated code with the other parts of the application code. The acceptance of the domain scientists for such solutions is crucial, as declarative and functional programming differs significantly from the usually used coding styles. Thus, such DSLs are not easily accepted from the domain scientists.

In contrast to the standalone DSLs, a language extension depends on adding new types and constructs to a general-purpose language to support the domain concepts and needs modifying a compiler accordingly to generate code. The solutions [22] [23] [24] use the idea of extending a

general purpose language. The ROSE compiler infrastructure [25] was used in both [22] and [24] to perform the source-to-source translation of the source code.

In our approach, we provide a higher-level language extension DSL, that can be processed with a more compact, dynamic and configurable source-to-source translation process. We use a higher abstraction level to allow scientists/developers to focus on the scientific domain. Technical details related to target platform are provided by architecture experts in separate compilation configurations. The concept generally applies to various general-purpose languages.

2 Software Engineering with GGDML

Our approach to get around the shortcomings of existing compilers is to provide the language extensions that lift the general-purpose code on a higher abstraction level. We extend the programming language that is used to write the source code of a model with constructs based on the domain science concepts. This leads to a clear separation of concerns (see Figure 2):

The domain scientists, who write a model's code, write only the code which is necessary to deliver the intended results from a scientific perspective. Additional technical details such as optimization or hardware-specific information should not show up in the source code. Thus, they should be given the right tools and language to do it. The language extensions in GGDML provide the way for that.

The hardware details and features should not be a concern for the domain scientists. Those details will be prepared by scientific programmers. Those programmers are experienced with an architecture's details, and how to use its features in the best way to optimize an algorithm's execution performance. They provide information how to translate GGDML to various backends and potentially technical DSLs.

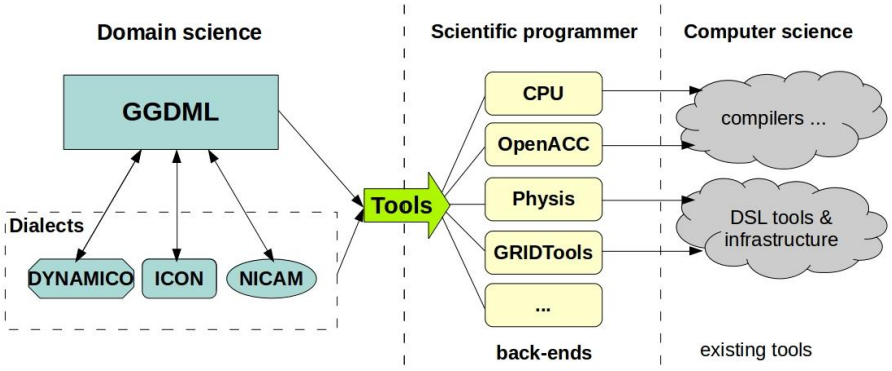

Figure 2: Software engineering with GGDML

A code repository which is written with the extended language (using GGDML) is translated into a code that is ready for a compiler or a backend tool, with the target machine features taken into account. A source-to-source translation tool handles this process. The finally generated software will be optimized for the target-machine.

3 Extending Models' Programming Language

Improving the software development process and the performance portability of the icosahedral models is the driver behind this work. This is also the goal of the AIMES project. A rewrite of a complete codebase with hundreds of thousands of lines is not acceptable. Therefore, we extend the models' language (Fortran) with domain-specific concepts relevant to the climate/atmospheric sciences. The development of the appropriate domain abstractions covered the analysis of the requirements, suggesting abstractions, the discussion and agreement in collaboration with scientists on the suggestions, and the specification of the language extensions.

3.1 Collaborative Extension Development

To develop the language extensions, we worked together with the domain scientists, each of whom is an expert with one of the three icosahedral models DYNAMICO [1], ICON [2], and NICAM [3], in a co-design approach:

- The domain scientists have suggested the code parts which are the most relevant. They have chosen the most complex or compute-intensive and time-consuming code regions.
- An abstraction has been extracted by recognizing the domain concepts and operations in these compute intensive code parts. During this process, we tried to identify commonalities in the three models and create a representation that expresses all the three models. Technical requirements for performance were considered during this abstraction process.
- We rewrote codes from the models according to the suggestions.
- We discussed with scientists the abstractions and code examples.

This process was repeated several times and thus, the specifications were iteratively refined until all the requirements were met.

3.2 Extensions and Domain-Specific Concepts

GGDML provides the extensions necessary for:

- declaring model's variables with respect to a grid.
- defining the grid itself and its subsets of cells, edges or vertices.
- declaring the model's variables that are defined over subsets of a grid.
- referencing the variables defined over a grid.
- traversing a grid to allow stencil operations over the whole grid or subsets of it. The original Fortran code that is written within an iterator is kept, but it is given the ability to reference the grid variables with language extensions.
- simplifying the stencil codes in which an operation is applied over multiple neighbors.

Those extensions ease the coding process, and hide memory layout details. Memory layout and access details are not defined by GGDML, nor are they part of the translation tools. Such lower-level details are platform dependent. GGDML lifts the code to abstract higher-level concepts. A provided configuration guides the translation process to generate the optimal memory layout and the access patterns that are suitable for the target hardware.

3.2.1 Grid Definition

Given that the model's variables are defined over a grid at its cells' centers, on their edges, or at their vertices, the DSL introduces the RANGE statement to define such grids (and subsets) in n-dimensional spaces.

A RANGE statement captures where variables are defined with respect to grids; at the cells' centers, on their edges, or at their vertices. A RANGE statement also captures the dimension of the space behind the grid. This dimension differs from the grid's dimensionality. That is because the icosahedral grids are composed of triangular/hexagonal shaped cells that fill a two dimensional space, in which we cannot simply use two dimensional indices to address data. To adapt for such an inherent attribute of the icosahedral grids, a RANGE defined over a dimensional space could be defined by a simple set with no dimensions (i.e., one dimensional with a filling curve mapping).

3.2.2 Operators

To simplify defining structured/semi-structured grids, GGDML uses Cartesian product of sets. For instance, when we deal with either a structured or an icosahedral grid for a surface and want to extend the grid into a three dimensional space, we multiply the surface grid with the set of vertical levels (Figure 3).

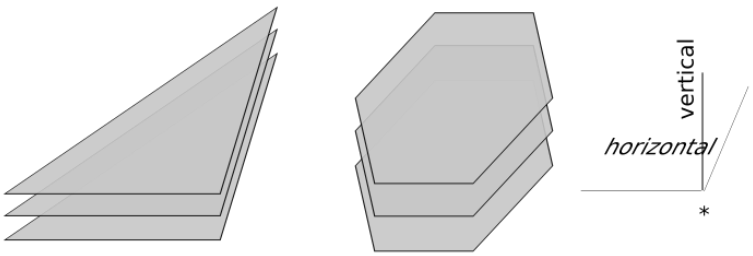

Figure 3: A 3D grid cell

Applying the Cartesian product operation on ranges, which are sets, yields higher dimensional grids. In addition, GGDML supports other operators that allow using the ranges that have already been defined in a model to define new ones. One of those operators allows dropping a dimension from a range, to go from higher to lower dimensional grids, the inverse operation of the Cartesian product. GGDML provides other operators which allow overriding a dimension in terms of its range of values, using the union operation, and the exclusion of ranges.

3.2.3 Data Definition and Manipulation

Statements to declare variables defined over a grid, and manipulating them are provided in GGDML. A grid is traversed via an iterator statement; "FOREACH", which allows accessing a variable's values over a grid/grid-subset. Whenever the variable's values can be traversed independently, we can safely use the "FOREACH" statement to get an optimized performance through parallelization and various optimization techniques. In such traversals, GGDML allows referencing the variables' values at the grid and navigating the elements around. For example, we can reference the cell above/below, the neighboring cells in a stencil operation, or the edges/vertices of a cell, and so forth. Navigating around an element of the grid simplifies coding

very much for the scientists. It eliminates the need for the indirect addressing of a variable's values or any repeatedly-used addressing details.

3.2.4 Stencils

Stencil operations play an important role in the abstraction process. It happens frequently that the calculation of a variable's value needs repeating an operation over a set of related elements like the neighboring cells. GGDML provides an extension to simplify such situations. Stencil operations are used frequently in the climate models, so, such an abstraction enhances the code readability, and cuts down the size of the source code.

3.3 Code Examples

In this part, we provide code examples from the three models before and after rewriting with GGDML. The examples demonstrate using GGDML to define grids, and to manipulate the grid-connected variables. RANGE statements, FOREACH statements, components and neighbor references, and stencil REUDCE constructs are demonstrated.

3.3.1 ICON

The following Fortran code from the ICON model uses directives to handle optimization for different architectures. The loops are interchanged in order to fit the target architecture in each section. The loop indices, which iterate the grid edges, are used in an indirect addressing to reference some variables defined over the cells around the iterated edges.

```
#ifdef __LOOP_EXCHANGE

 DO je = i_startidx, i_endidx
!DIR$ IVDEP, PREFERVECTOR
  DO jk = nflat_gradp(jg)+1, nlev
#else
 DO jk = nflat_gradp(jg)+1, nlev
  DO je = i_startidx, i_endidx
#endif
   ! horizontal gradient of Exner pressure,
   ! Taylor-expansion-based reconstruction
   z_gradh_exner(je,jk,jb) = p_patch%edges%inv_dual_edge_length(je,jb)* &
    (z_exner_ex_pr(icidx(je,jb,2),ikidx(2,je,jk,jb),icblk(je,jb,2)) +   &
     p_nh%metrics%zdiff_gradp(2,je,jk,jb)*                              &
    (z_dexner_dz_c(1,icidx(je,jb,2),ikidx(2,je,jk,jb),icblk(je,jb,2)) + &
     p_nh%metrics%zdiff_gradp(2,je,jk,jb)*                              &
     z_dexner_dz_c(2,icidx(je,jb,2),ikidx(2,je,jk,jb),icblk(je,jb,2)))- &
    (z_exner_ex_pr(icidx(je,jb,1),ikidx(1,je,jk,jb),icblk(je,jb,1)) +   &
     p_nh%metrics%zdiff_gradp(1,je,jk,jb)*                              &
    (z_dexner_dz_c(1,icidx(je,jb,1),ikidx(1,je,jk,jb),icblk(je,jb,1)) + &
     p_nh%metrics%zdiff_gradp(1,je,jk,jb)*                              &
     z_dexner_dz_c(2,icidx(je,jb,1),ikidx(1,je,jk,jb),icblk(je,jb,1)))))
  ENDDO
 ENDDO
```

Equivalent code rewritten with GGDML:

```
FOREACH edge IN grid%edges

        ! horizontal gradient of Exner pressure,
        ! Taylor-expansion-based reconstruction
```

```
              z_gradh_exner(edge) = edge%inv_dual_edge_length*  &
               (z_exner_ex_pr(edge%cell(2)) +                   &
                p_nh%metrics%zdiff_gradp(2,edge)*               &
               (z_dexner_dz_c(edge%cell(2),1) +                 &
                p_nh%metrics%zdiff_gradp(2,edge)*               &
                z_dexner_dz_c(edge%cell(2),2)) -                &
               (z_exner_ex_pr(edge%cell(1)) +                   &
                p_nh%metrics%zdiff_gradp(1,edge)*               &
               (z_dexner_dz_c(edge%cell(1),1) +                 &
                p_nh%metrics%zdiff_gradp(1,edge)*               &
                z_dexner_dz_c(edge%cell(1),2))))
END FOREACH
```

The code written with GGDML uses the iterator that iterates the edges of the grid. The abstract (edge) index is used to reference the variables instead of explicitly using indices that impact the performance because of memory layout. Using (edge%cell) to refer to the cells around an edge simplifies the indirect addressing. This way, GGDML hides the memory layout and connectivity information.

3.3.2 DYNAMICO

The following Fortran code from the DYNAMICO model uses two nested loops with a directive to vectorize the inner loop which iterates the horizontal grid. The horizontal loop index is used to calculate the indices of the neighbors in a stencil operation.

```
DO l=ll_begin,ll_end

!DIR$ SIMD
  DO ij=ij_begin,ij_end

   berni(ij,l) = .5*(geopot(ij,l)+geopot(ij,l+1)) &
    + 1/(4*Ai(ij))*(le(ij+u_right)*de(ij+u_right)*u(ij+u_right,l)**2 + &
                    le(ij+u_rup)*de(ij+u_rup)*u(ij+u_rup,l)**2 +       &
                    le(ij+u_lup)*de(ij+u_lup)*u(ij+u_lup,l)**2 +       &
                    le(ij+u_left)*de(ij+u_left)*u(ij+u_left,l)**2 +    &
                    le(ij+u_ldown)*de(ij+u_ldown)*u(ij+u_ldown,l)**2 + &
                    le(ij+u_rdown)*de(ij+u_rdown)*u(ij+u_rdown,l)**2 )
  ENDDO
ENDDO
```

Equivalent code rewritten with GGDML:

```
RANGE,CELL,3D gc = ij{ ij_omp_begin_ext .. ij_omp_end_ext }*l {1 .. llm}


FOREACH cell IN gc
 berni(cell) = .5*(geopot(cell)+geopot(cell%above)) + 1/(4*Ai(cell%ij))
  * REDUCE(+, N={1..6}
  le(cell%neighbour(N)%ij)*de(cell%neighbour(N)%ij)
                                          *u(cell%neighbour(N))**2)
END FOREACH
```

The rewritten code uses the FOREACH statement to iterate the set of cells and update the variable (berni) at each of the iterated cells. Using the REDUCE extension along with the

(cell%neighbour) to refer to the neighbours of a cell simplifies the code of the stencil operation and eliminates the duplicate code over each neighbour.

3.3.3 NICAM

The following Fortran code from the NICAM model uses three nested loops with an OpenCL directive to harness parallel execution capabilities. The code defines variables to help calculate the indices that are necessary to reference the variables over the neighbour cells within the stencil operation.

```
    do d = 1, ADM_nxyz

    do l = 1, ADM_lall
!OCL PARALLEL
    do k = 1, ADM_kall
       do n = OPRT_nstart, OPRT_nend
          ij     = n
          ip1j   = n + 1
          ijp1   = n     + ADM_gall_1d
          ip1jp1 = n + 1 + ADM_gall_1d
          im1j   = n - 1
          ijm1   = n     - ADM_gall_1d
          im1jm1 = n - 1 - ADM_gall_1d

          grad(n,k,l,d) = cgrad(n,l,0,d) * scl(ij    ,k,l) &
                        + cgrad(n,l,1,d) * scl(ip1j  ,k,l) &
                        + cgrad(n,l,2,d) * scl(ip1jp1,k,l) &
                        + cgrad(n,l,3,d) * scl(ijp1  ,k,l) &
                        + cgrad(n,l,4,d) * scl(im1j  ,k,l) &
                        + cgrad(n,l,5,d) * scl(im1jm1,k,l) &
                        + cgrad(n,l,6,d) * scl(ijm1  ,k,l)
       enddo
       grad(          1:OPRT_nstart-1,k,l,d) = 0.0_RP
       grad(OPRT_nend+1:ADM_gall     ,k,l,d) = 0.0_RP
    enddo
    enddo
    enddo
```

Equivalent code rewritten with GGDML:

```
  RANGE, CELL, 3D g1 =  GRID%cells | g{OPRT_nstart..OPRT_nend}

  FOREACH cell in g1
    do d = 1, ADM_nxyz
          grad(cell,d) = REDUCE(+,N={0..6},
                        cgrad(cell%g,cell%l,N,d) * scl(cell%neighbor(N)) )
    enddo
  END FOREACH

  FOREACH cell in GRID%cells | g{1..OPRT_nstart-1 , OPRT_nend+1 .. gall}
    do d = 1, ADM_nxyz
            grad(cell,d) = 0.0_PRECISION
    enddo
  END FOREACH
```

Using GGDML operators to define the RANGE in the first line helps iterating a subset of the grid cells. Within the iterator, the use of (cell%neighbour) removes the complexity of the index

calculations which are necessary to reference neighbours. Also the REDUCE extension simplifies the stencil operation over the neighbours.

4 Results

We have taken two relevant kernels from each of the three models, and analyzed the achieved code reduction. An overview of the results is shown in Figure 4 and Figure 5. The numbers demonstrate the impact on the code length when porting the code to GGDML.

	lines (LOC)		words		characters	
	before DSL	with DSL	before DSL	with DSL	before DSL	with DSL
ICON1	13	7	238	174	317	258
ICON2	53	24	163	83	2002	916
NICAM1	27	13	148	69	924	408
NICAM2	90	11	344	53	1487	363
DYNAMICO1	14	5	115	54	402	272
DYNAMICO2	13	5	30	20	402	218
total	210	65	1038	453	5534	2435
	30.95%		43.64%		44.0%	

Figure 4: Impact of GGDML on LOC

In average, we cut down the LOC to less than one third (~31%) of the original code. Better reductions are achieved in stencil codes (NICAM example No.2, reduced to 12% of the original LOC).

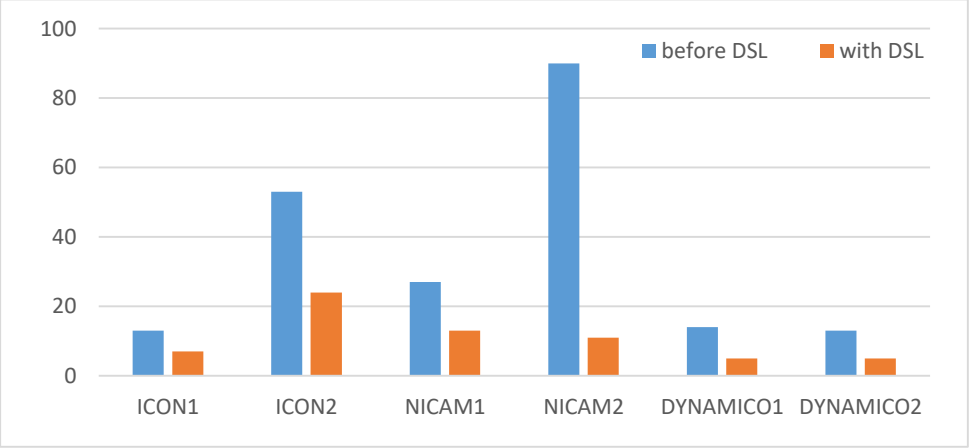

Figure 5: LOC before and after using GGDML

Influence on readability and maintainability: Reducing the important code metrics like code duplication, WTF/Minute -in code review, in some cases, boundary conditions could be removed thus reducing the cyclomatic complexity.

We provide the estimated benefits in Figure 6. According to the COCOMO method the code reduction offered by GGDML could save half the estimated development costs.

Software Project	Codebase	Effort Applied	Dev. Time (Months)	People Required	Dev. Costs (M€)
Semi-detached	Fortran	2462	38.5	64	12.3
	DSL	1133	29.3	39	5.7
Organic	Fortran	1295	38.1	34	6.5
	DSL	625	28.9	22	3.1

Figure 6: Cost estimates with COCOMO

5 Discussion and Conclusion

In this article, we introduced GGDML, a set of extensions to the Fortran language to improve the software engineering of climate/atmospheric modeling. The developed extensions are driven by the shortcomings of compilers, in particular, with respect to performance portability. In the design we use a bottom-up approach to account for the requirements. However, we re-engineer the language extension top-down to provide a consistent perspective from the domain science point of view. We make abstractions for computation intensive parts of three existing models, up to the level of the domain concepts. This leads to a set of concepts reflecting the domain science/application, and bypassing the low-level implementation details. The hardware related information are eliminated from the structure of a code written with GGDML. Applying the DSL reduces the code size significantly (code with GGDML is less than one third the size of the original Fortran code) and impacts the development process and costs.

For future work, we will continue the improvement and refinement of GGDML. A priority is to develop a reliable source-to-source translation tool to effectively support our solution.

6 Acknowledgements

This work was supported in part by the German Research Foundation (DFG) through the Priority Programme 1648 “Software for Exascale Computing” (SPPEXA) (GZ: LU 1353/11-1).

7 References

1. Dubos T, Dubey S, Tort M, Mittal R, Meurdesoif Y, Hourdin F. DYNAMICO, an icosahedral hydrostatic dynamical core designed for consistency and versatility. Geoscientific Model Development Discussions. 2015; 8(2): p. 1749-1800.

2. Zängl G. ICON: The icosahedral nonhydrostatic modelling framework of DWD and MPI-M. In Proc. ECMWF Seminar on Numerical Methods for Atmosphere and Ocean Modelling; 2013.

3. Satoh M, Tomita H, Yashiro H, Miura H, Kodama C, Seiki T, et al. The non-hydrostatic icosahedral atmospheric model: Description and development. Progress in Earth and Planetary Science. 2014; 1(1): p. 1.

4. Bianco M, Varetto U. A generic library for stencil computations. arXiv preprint arXiv:1207.1746. 2012.

5. Shimokawabe T, Aoki T, Onodera N. High-productivity framework on GPU-rich supercomputers for operational weather prediction code ASUCA. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis; 2014. p. 251-261.

6. Shimokawabe T, Aoki T, Onodera N. High-productivity Framework for Large-scale GPU/CPU Stencil Applications. Procedia Computer Science. 2016; 80: p. 1646-1657.

7. Chang LW, Dakkak A, Rodrigues CI, Hwu W. Tangram: a high-level language for performance portable code synthesis. In Programmability Issues for Heterogeneous Multicores. 2015.

8. Reguly IZ, Mudalige GR, Giles MB, Curran D, McIntosh-Smith S. The OPS domain specific abstraction for multi-block structured grid computations. In Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC), 2014 Fourth International Workshop on; 2014. p. 58-67.

9. Mudalige G, Giles M, Reguly I, Bertolli C, Kelly PJ. OP2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures. In Innovative Parallel Computing (InPar), 2012; 2012. p. 1-12.

10. Dolbeau R, Bihan S, Bodin F. HMPP: A hybrid multi-core parallel programming environment. In Workshop on general purpose processing on graphics processing units (GPGPU 2007); 2007.

11. Christen M, Schenk O, Burkhart H. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International; 2011. p. 676-687.

12. CSCS. CSCS Claw [WWW]. [cited 2017 4 5. Available from: https://github.com/C2SM-RCM.

13. Unat D, Cai X, Baden SB. Mint: realizing CUDA performance in 3D stencil methods with annotated C. In Proceedings of the international conference on Supercomputing; 2011. p. 214-224.

14. Ford R, Glover M, Ham D, Maynard C, Pickles S, Riley G, et al. Gung Ho: A code design for weather and climate prediction on exascale machines. In Proceedings of the Exascale Applications and Software Conference; 2013.

15. Edwards HC, Trott CR, Sunderland D. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. Journal of Parallel and Distributed Computing. 2014; 74(12): p. 3202-3216.

16. Fuhrer O, Osuna C, Lapillonne X, Gysi T, Cumming B, Bianco M, et al. Towards a performance portable, architecture agnostic implementation strategy for weather and climate models. Supercomputing frontiers and innovations. 2014; 1(1): p. 45-62.

17. Berényi D. C++ EDSL for parallel code generation. In Grid, Cloud & High Performance Computing in Science (ROLCG), 2015 Conference; 2015. p. 1-5.

18. Yount C. Recipe: Building and Running YASK (Yet Another Stencil Kernel) on Intel® Processors [WWW].; 2017 [cited 2017 4 5. Available from: https://software.intel.com/en-us/articles/recipe-building-and-running-yask-yet-another-stencil-kernel-on-intel-processors.

19. CSCS. CSCS GridTools [WWW]. [cited 2017 4 5. Available from: http://www2.cosmo-model.org/content/consortium/developers/2016_01/Gridtools_python.pdf.

20. A van Engelen R. ATMOL: A domain-specific language for atmospheric modeling. CIT. Journal of computing and information technology. 2001; 9(4): p. 289-303.

21. DeVito Z, Joubert N, Palacios F, Oakley S, Medina M, Barrientos M, et al. Liszt: a domain specific language for building portable mesh-based PDE solvers. In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis; 2011. p. 9.

22. Maruyama N, Sato K, Nomura T, Matsuoka S. Physis: an implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers. In 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC); 2011. p. 1-12.

23. Rathgeber F, Markall GR, Mitchell L, Loriant N, Ham DA, Bertolli C, et al. PyOP2: A high-level framework for performance-portable simulations on unstructured meshes. In High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:; 2012. p. 1116-1123.

24. Torres R, Linardakis L, Kunkel TJ, Ludwig T. ICON DSL: A domain-specific language for climate modeling. In International Conference for High Performance Computing, Networking, Storage and Analysis, Denver, Colo.[Available at h ttp://sc13. supercomputing. org/sites/default/files/WorkshopsArchive/track139. html.]; 2013.

25. Quinlan D. ROSE: Compiler support for object-oriented frameworks. Parallel Processing Letters. 2000; 10(02n03): p. 215-226.