ICON DSL: A Domain-Specific Language for climate modeling

Raul Torres
University of Hamburg

Leonidas Linardakis
Max Planck Institute for Meteorology

Julian Kunkel
Deutsches Klimarechenzentrum

Thomas Ludwig
Deutsches Klimarechenzentrum

***Abstract*—The presence of platform-specific structures inside the ICON climate model make the code overall complex and inflexible, while the extend and number of code variations that can be expressed through preprocessing directives is limited. For cases like this, abstraction of data and loop structures offered by a Domain-Specific Language (DSL) allows a single description of the model and also flexibility for adapting to architectures. Additionally, a DSL eases the programming task for climate scientists since it provides a syntax closer to the semantics of the model domain while hiding the coding details which are important to achieve performance. We have designed a DSL as a natural extension of Fortran capable to do such abstractions. We also have formed a source-to-source translator that converts DSL enriched code into native Fortran while applying machine specific translations.**

I. Introduction

Global climate simulations are one of the Grand Challenges of computing [1], they rely on complex Earth System Model (ESM) runs, at the highest possible resolution and for long timescales. These models are often composed by several hundreds of thousands of code lines, and their complexity is increasing in order to simulate additional physical processes. It is cumbersome to express the model in general-purpose languages while still achieving good efficiency on different platforms. The current diversity and constant evolution of hardware architectures and programming models for solving computational problems makes it difficult to design efficient programs that could exploit the capabilities of all these technologies. Generally, the modeler – who is an expert in the climate domain – has to spend an increasing amount of time dealing with computation details rather than on relevant scientific questions. The scientist has to face the problem of making a reasonable trade-off between efficiency and portability.

This challenge has been experienced by the scientists developing the ICON climate model. ICON is a joint project of Max Planck Institute for Meteorology and the German Weather Service, with the goal of developing a new generation of general circulation models for the atmosphere and the ocean in a unified framework [2]. This model solves the fully compressible non-hydrostatic equations of motion at a very high horizontal resolution based on an icosahedral grid. ICON is written in Fortran, a general purpose language. Its code exhibits several explicit machine-dependent optimizations that make it harder to debug and maintain. Loop exchange, for example, is perform using pre-processor directives that irrupt between Fortran statements.

We aim to provide an abstraction framework in the context of an ICON Domain-Specific Language (ICON DSL). Such language is bundled with a reliable Source-to-Source translator that converts DSL code into fully compatible Fortran code, where the computation details are expressed. This translator uses an Intermediate Representation (IR) suitable for simplification and high level optimizations, but most important, independent from a specific hardware architecture.

Our target is not only adaptivity to architectures; we also aim for creating a real domain-specific language able to capture the semantics of the climate model. In this sense, we are currently integrating to the language the notion of sets and subsets and a proper way to loop over their elements.

To construct the ICON DSL, we have utilized a bottom-up approach; we started from the basic structures of the code that may be suitable for abstraction. In this first stage of the development process, we provide keywords which control memory dimension and layout of variables with specific model semantics. Both dimension and layout specification are hidden from the modeler, being the translator in charge of generating adequate implementation according to a selected platform. Being our DSL an extension of Fortran, it is going to be easier for scientists to adapt to it rather than having to learn a complete new language – they just have to get familiar with a few new keywords and language constructs. It is also more natural to a modeler, since language extensions are emerging with new Fortran standards every few years.

These ideas are not new. Nevertheless, the novelty of our approach consists in the attempt to make the language more natural and easier for the modelers and at the same time extending Fortran rather than creating a complete set of new rules. The features introduced by the DSL are discussed within the ICON development group and are adjusted to the modelers' requirements for expressiveness. Actually, inside the climate community, the high level approach is the usage of backend libraries or template-based operators, both being awkward expressions of mathematical operators.

The current implementation is preliminary, but demonstrates a great potential for adaptivity and user-

friendliness. New ideas of features are currently under consideration and it is our goal to gain the approval of the DSL approach by the climate modeling community, where previous similar attempts have been not successful.

This paper presents a DSL extension of Fortran for the ICON climate model, suitable for abstracting memory operations and optimizations, as well as the corresponding source-to-source translation infrastructure for automatic code generation. The paper is structured as follows: in section II related efforts in DSL design are covered; in section III the details of the DSL are discussed, while in section IV we discuss the challenges we faced while building the source-to-source translator; section V shows first performance results; finally, sections VI and VII present a discussion about on going and future work, and conclusions, respectively.

II. Related work

Even though general purpose languages like Fortran or C/C++ intent to be platform independent, efficiency might depend on the code structure and memory layout which is architecture dependent. Unfortunately, the compiler cannot change the memory layout by itself because modifications could change the semantics of the program. Performance optimization requires deep knowledge about both the platform and the language capabilities, making the programming task more difficult for the domain expert. At this point, a DSL comes in handy.

The advantages of DSLs as helpers for the domain expert to express the problem in a more natural way, are well documented in the literature [3][4][5]. However, it is important to remark that, actually, most approaches converge into the idea that domain targeting also guides DSL designers to the identification of those domain-dependent computation bottlenecks and to find the most optimized ways to express them in the target general purpose language [6].

There are several DSL frameworks tackling the problem of abstraction for specific domains, in order to generate optimized code. Nevertheless, they differ from our approach in the type of DSL and the relying infrastructure used.

DSLs can be designed as complete new language. For example, in 2001, van Enlangen proposed ATMOL [7], a DSL for atmospheric models. It was build on top of Ctadel language [8], a PDE framework able to generate optimized Fortran code for HIRLAM-based models[9]. The provided abstractions for the language where straightforward. However, there are not so many documented uses of it in the literature. A more recent work is Liszt [10], a DSL framework to build mesh-based PDE solvers, crafted on top of Scala language. Similar to our approach, this DSL is able to generate code for multiple platforms while abstracting the mesh elements of the solver and allowing the construction of sets with topological relationships.

The challenge for complete new DSLs, as well as for any new language, is to achieve significant acceptance from the community and therefore avoid falling into disuse and become unsupported in new platforms; also, migrating complex domain-specific applications with years of development and more than 100,000 lines of code to another language is unfeasible. When such issues are faced, there is the option to design an extension of an existing language; this extension is intended to specialize the language in a certain domain. The advantage of this direction is that it requires little learning effort from the domain modelers, who are usually experts in the base language. Additionally, existing code can be modified incrementally, which reduces the burden to developers and increases testing cycles.

Approaches like HiCUDA [11] intent to ease the task of converting a sequential program into a GPU parallel one using annotations over a general-purpose language like C; but HiCUDA transformations are not domain-specific. A similar annotation approach but with a DSL flavor can be seen in Mint [12]. The difference with HiCUDA resides upon the fact that instead of offering general-purpose transformations, Mint focuses in provisioning a set of annotations to express parallelism only for the domain of stencil computations.

The problem with annotations resides upon the fact that they are constructions that don't integrate cleanly into the grammar of the base language and therefore break the natural programming work flow on the developer side.

It is also possible to use preprocessing macros to extend a language. However, they become unpractical and bug prone when the programmer has to deal with complex codes. For example, using macros to define the dimensions and memory layout restrict an array identifier to such features, when these arrays may have different sizes and even a different order depending on the context. In general, relying on preprocessing macros is considered to be a bad coding practice. The use of templates is a more feasible and correct approach with good results, like is the case of the DSL for the COSMO model [13].

Our approach intends to benefit from the advantages of a DSL extension, but with a cleaner integration of new keywords into the existing grammar rules. In fact, our extensions can be considered as Fortran standard targeting advanced performance portability.

III. ICON Domain-Specific Language

The ICON Domain-Specific Language acts as an extension of Fortran. The modelers only have to learn a few meaningful keywords and constructions, as well as the correct places where to put them in their Fortran codes. Such keywords are crafted to conveniently generalize platform-dependent details like memory layout or hardware-specific optimizations, and their use can help the modeler avoid expressing computational details that can make the code otherwise less portable.

There are to basic principles that support this design:

- The ability to express climate mathematical operators in an easy and natural way.

- The capability to adapt the implementation of these operators to different architectures and parallel levels

The keywords and constructions of the language extension and their corresponding behavior are defined in a separated platform-specific file. The platform specific file must be created only once and its details are hidden from the normal scientist developing the model coding. Each new keyword is defined in the platform configuration file as 3-field tuple separated by spaces, as follows:

```
<keyword_name> <platform_specific_settings> <keyword_type>
```

The *keyword_name* string clearly makes reference to the new keyword to be incorporated in the language. The *platform_specific_settings* is a string used to characterize the particular behavior of the keyword. The *keyword_type* is a string that indicates which constructions of the code can benefit from it.

Currently, there are 3 different types specified: Array declarations, array initializers and optimizers. The potential options for platform specific settings depend on the type of the declaration.

Example:
The configuration is illustrated on an example for a declaration of a 2-D variable:

```
BASIC_ARRAY {1,0} declare
```

This declaration has the following meaning:

- BASIC_ARRAY is the new keyword which can be used within the source code.
- The platform specific string ({1,0}) specifies that an array of this kind will be 2-dimensional and all index references will swap index positions
- `declare` states that the keyword can only be used for declaration statements.

On another platform, the configuration can be adapted to use another memory layout:

```
BASIC_ARRAY {0,1} declare
```

A. Array declarations

At an array declaration, users can add a meaningful keyword to characterize the variable. This keyword encodes in itself information like the dimension and the memory layout. Therefore, the dimension part can be avoided in the definition.

1) Platform-specific configuration: For multiple-dimension array declarations, `platform_specific_settings` specifies the dimension and the memory layout (how the indexes must be interchanged). `keyword_type` specifies that the keyword must be used in a declaration statement.

Example configuration:

```
ON_CELLS {1,2,0,3} declare
```

The platform-specific configuration {1,2,0,3} is interpreted as follows:

- Index 0 goes to position 2
- Index 1 goes to position 0
- Index 2 goes to position 1
- Index 3 remains in position 3

2) Usage of the keyword: The new keyword must be written in any place between the type specification and the double colons of a declaration statement in Fortran. Nevertheless, it is considered only when the variable is an array. If not, it is ignored by the DSL parser. Notice the lack of a dimension specification in the Fortran declaration. The modeler does not have to take care of a special arrangement of indexes.

Example usage:

```
REAL, ON_CELLS, POINTER :: my_variable
my_variable( i , j , k, l) = 2
```

3) Generated Fortran code: The source-to-source translator takes both the code and platform-specific file and transforms the declaration line as follows:

- The keyword is removed from declaration
- The dimension specifier is added to declaration
- At variable indexing references, the index interchange for the specified platform is applied

Thus, by applying the machine configuration the code of 2), it is translated into:

```
REAL, DIMENSION(:,:,:,:), POINTER :: my_variable
my_variable( j , k , i, l) = 2
```

B. Array initialization

Array initialization can benefit from the index interchange feature too. In this case, the used keyword encodes only the memory layout, because this feature works only with one-dimensional arrays. These special arrays store in each position, size information for other multiple-dimension arrays.

1) Platform-specific configuration: The 3-field tuple in the configuration file for this keyword is similar to the one used for array declarations. The only difference is that `keyword_type` must indicate that the keyword can only be used at an initialization statement.

Example configuration:

```
SHAPE_4D {1,2,0,3} initialize
```

2) Usage of the keyword: The keyword must be written after the equal symbol and right before the opening bracket of the initialization statement. The characters, usually used inside the brackets, must be avoided. Recall that `a, b, c, d` make reference to the new values that each position of the array is going to have.

Example usage:

```
my_variable = SHAPE_4D( a, b, c, d )
```

3) Generated Fortran code: At translation time, the statement is transformed as follows:

- The keyword is removed
- The dimension characters inside the brackets are added
- Indexes are interchanged according to the configuration file

Thus, by applying the machine configuration the code of 2), it is translated into:

```
my_variable = (/ b , c , a, d /)
```

C. Optimizers

Optimizers can be associated to keywords, which can be used to dictate how certain blocks of code are transformed to improve performance. At the moment we just support on-demand inlining.

1) Platform-specific configuration: In this case, the 3-field tuple changes as follows: `keyword_type` characterize the keyword as an optimizer while `platform_specific_settings` denotes the specific optimizer to be applied. Both values are utilized by the translator to restrain the use of the keyword in specific blocks of code.

Example configuration:

```
INLINE inline optimize
```

2) Usage of the keyword: Optimizers usually are applied to blocks of code. Inlining works only with subroutines, but the keyword can be used in two contexts: one is to be written before a subroutine definition; this forces inlining in all uses of the subroutine.

Example usage:

```
INLINE SUBROUTINE example_subroutine(...)
```

But when it is written before a specific subroutine call, only that subroutine call is forced to be inlined:

Example usage:

```
INLINE CALL example_subroutine(...)
```

3) Generated Fortran code: The transformations depend on the type of optimizer. For the inlining optimizer, the transformations performed are as follows:

- The keyword used at subroutine declaration is deleted
- At subroutine call, the complete call is properly replaced with the body of the inlined subroutine
- Declarations for dummy arguments inside the pasted subroutine body are deleted
- Names of new variables of the subroutine are changed to avoid conflicts with other variables in the containing scope

IV. Design of the Source-to-source translation infrastructure

At the beginning of the development process, we used the ANTLR Parser Generator[14] as the base infrastructure. Our decision was based on the its capabilities for designing of parsers for grammars, specially for DSLs. The provision of an AST represented a strong point to semantical transformations. However, we encountered several burden that made development harder.

- The symbol table must be built and managed by the programmer itself, which can be easy for simple languages but not for Fortran, where one has to be aware about the scope level not only at the current file but also at the imported modules
- Dealing with the AST is cumbersome, due to the fact it is represented as a manually written reduced grammar (tree grammar) that keeps only the meaningful tokens. The programmer has to transform the original grammar (parser grammar) into the reduced version using rewriting rules. But the rewriting rules don't generate the reduced grammar automatically. Instead, the programmer has to make sure that the rewriting rules correspond exactly to the desired reduced grammar. This is time consuming and error-prone for a general purpose language like Fortran
- Even though transformations on the reduced grammar are easier to express, one has to create a complete set of the language again, but recovery of the ignored tokens is not that easy.
- The implementation of the inlining mechanism required the support of an external text replacement tool.

The characteristics of ANTLR make it more suitable for tasks like: design simple grammars and translators, implementation of parsers, and construction of translators between different languages. For the case of the design of language extensions, more powerful tools should be used.

Another tool that was taken into account was TXL [15], a powerful language to perform text transformations based on example but with a complex mechanism to manipulate the intermediate representation; moreover, the source code is still not open to the community, a relevant requirement for our project. We also considered the use of LLVM[16] which provides a low-level intermediate representation of the Static Single Assignment form but that does not fit with our purpose to keep our transformations in the higher possible level. Nevertheless, the wide spectrum of hardware architectures that LLVM targets make it a tool to consider when we decide to leave the high level abstractions.

We have built our translator on top of the Rose Compiler [17]. Rose is able to parse most of the Fortran expressions of our codes by using Open Fortran Parser (developed under ANTLR). It also provides a robust high level tree structure as intermediate representation without losing almost none of the original tokens. It is called SAGE III and provides an object

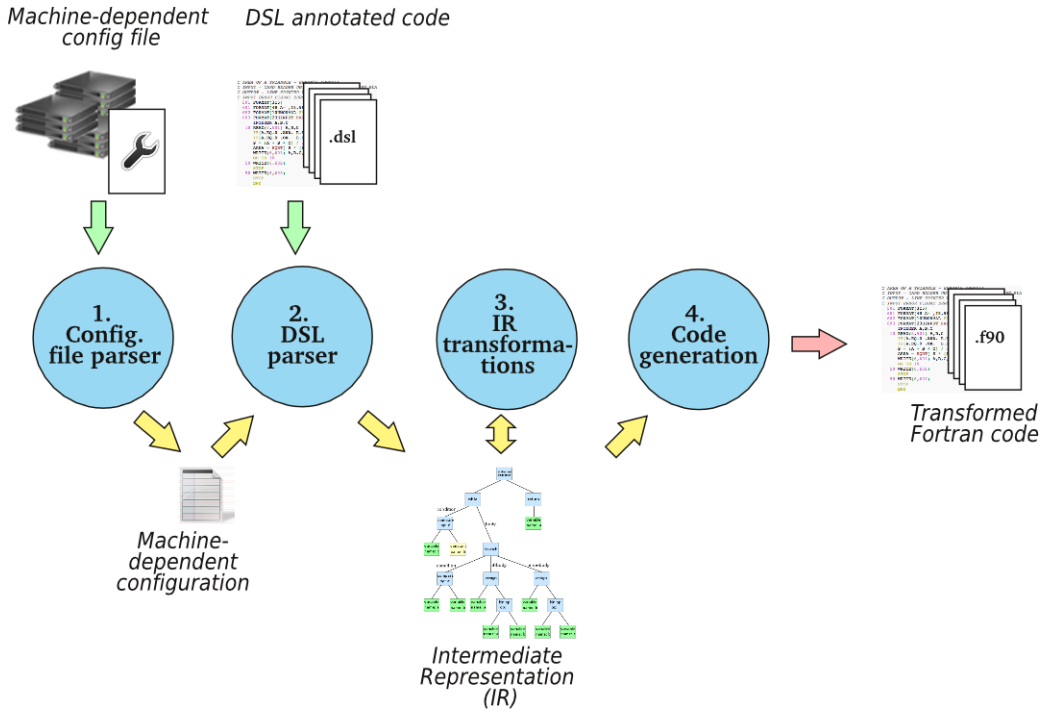

Fig. 1. Translation infrastructure

oriented API [18].

The symbol table is automatically built, we do not have to care about conflicts or missing parts. Once we have a valid AST, the unparsing is done automatically (with some modifications on the tree, there is also the possibility of generating C/C++ code)

However, some issues had to be fixed to parse our extension and perform the DSL transformations:

- Rose Compiler provides no interface to design a language extension. We developed a parser module that reads the code, detects our keywords and replace them with customized pragma annotations. After this first transformation, code can be passed to Rose Compiler.
- A few correctly parsed Fortran statements have no corresponding action to build nodes on the AST. We implemented the missing actions inside our version of Rose.
- Pragma annotations of the kind of Open MP are given nodes in C or C++ codes, but not in Fortran codes. In Fortran they are stored as comments but sometimes are misplaced, which changes the parallel semantics of the program. We had to trick Rose by make them appear as function calls –not an elegant solution– but one that preserves the semantics. We plan to change this in the future.
- Rose creates a sort of header files for Fortran modules, but they do not store the semantics of the our extension. The solution is to replace the Rose header file with the one we generate in an intermediate stage.

Once the AST is built, the transformations are easy to made, in terms of adding or deleting nodes and subtrees.

The translation of extended Fortran code into native Fortran works as follows (see Figure 1):

1) A machine-dependent configuration file is parsed, where the particular details of the platform are specified.
2) The DSL enriched Fortran code is parsed, the symbol table and the intermediate representation, called Abstract Syntax Tree (AST), are constructed, without losing any information about the source code.
3) Before unparsing, the tree is modified to transform the provided abstractions according to those particularities of the platform.
4) As a final step, native Fortran code is generated by traversing the modified tree.

V. Evaluation

The original code was optimized initially for a vector machine (NEC), but when the code was executed on current cache based machines, there was a bottleneck in the memory bandwidth. The machine-dependent file had to be tailored to use an optimized memory layout for IBM Power6 and Intel Westmere architectures. The memory layout was determined manually to make a better use of the available cache levels. However, different configurations were tried to find the correct one, because there are several arrays with differences in dimension and their corresponding loops. Further, the DSL abstractions were applied on the ICON testbed code and a synthetic test for the ICON dynamical core was used with a configuration of 20480 cells x 78 levels. The DSL keyword

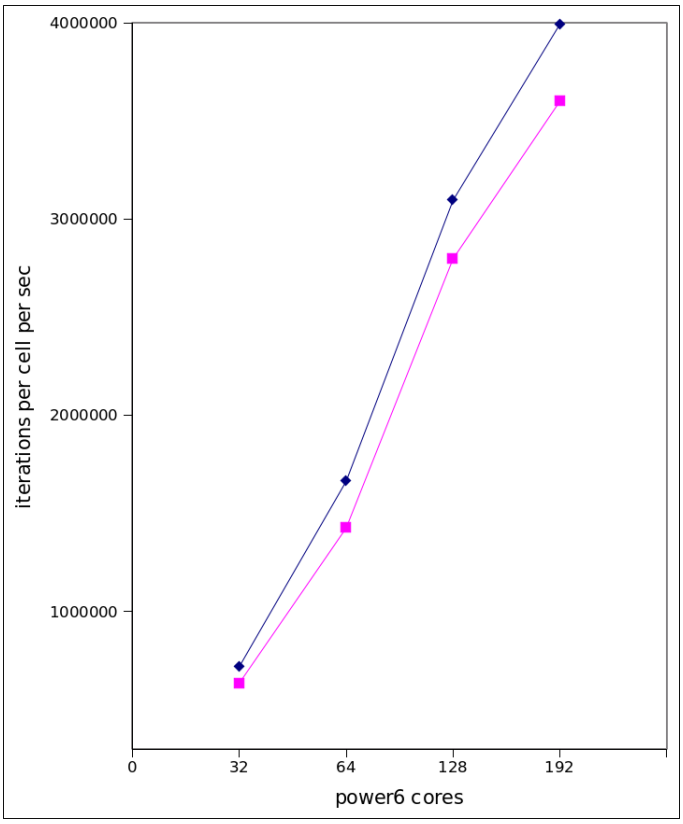

Fig. 2. Performance comparison between code with and without DSL keywords for IBM Power 6 architecture

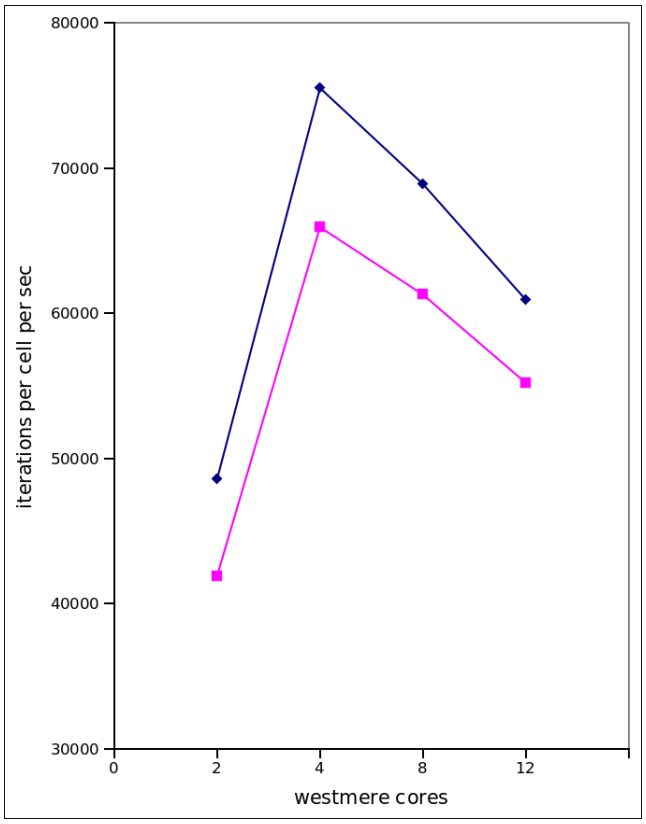

Fig. 3. Performance comparison between code with and without DSL keywords on a Intel Westmere architecture

for inlining was not used. Finally, generated Fortran code was compiled and executed on the mentioned architectures.

A. *Power6 architecture*

With the appropriate machine-specific configuration the efficiency of central data structures of ICON could be improved, obtaining up to 17% of speedup (see table I and figure 2).

B. *Intel Westmere architecture*

For the case of Westemere, up to 16% of speedup was obtained (see table II and figure 3).

On table III we can see the behavior of some hardware counter groups for the Westmere architecture. These counters were obtained using the Likwid Lightweight performance tools[19]. Memory bandwidth was increased by 14% and the miss rate of L1 and L2 caches were reduced by 68% and 20% respectively. It is notable that the proper index interchanges provided an reduction by 7% on the retired instructions counter and by 6% on the cycles per instruction. This could be due to the fact that continuous memory access provides more room for compiler optimizations, for example, loop unrolling.

VI. On going and Future work

In order to make full use of the ability to define the memory access patterns, loop structures should also be abstracted. This abstraction also targets to improve the expressiveness of the DSL, in a way closer to the modelers' domain. We are currently considering to use the notion of sets and subsets, which is the natural mathematical construction for identifying indexed based operations. Our goal is to use the DSL abstraction to create a similar code but hiding the indexing details. An example of such an abstraction is:

```
type(t_int_state), intent(in)            :: ptr_int
real(wp), EDGES_3D, intent(in)       :: vec_e
intent(wp), CELLS_3D, intent(inout) :: div_vec_c
SUBSET, CELLS_3D, intent(in)     :: cells_subset
ELEMENT, CELLS_3D                :: cell
ELEMENT, EDGES_OF_CELL           :: edge

FOR cell in cells_subset DO
  div_vec_c(cell) = 0.0_wp
  FOR edge in cell%edges DO
    div_vec_c(cell) =  div_vec_c(cell) + &
    & vec_e(edge) * ptr_int%geofac_div(edge)
  END FOR
END FOR
```

Notice in the example code that the modeler will not make use of explicit indexes on DSL loops, it is only necessary to establish the corresponding sets and the operations among the elements of the set. The notion of subsets has been already implemented in the ICON ocean model in the form of Fortran structures and looks like this:

```
type(t_int_state), type(in) :: ptr_int
real(wp), intent(in)        :: vec_e(:,:,:)
real(wp), intent(inout)     :: div_vec_c(:,:,:)
type(t_subset_range_3D)     :: cells_subset
type(t_grid_cells), pointer :: cell_cells
integer                     :: cell_idx_start, cell_idx_end, ...
integer                     :: edge_cell_idx, edge_idx, ...
```

Cores	32	64	128	192
NO_DSL iterations/sec	635479	1426037	2798150	3601217
DSL iterations/sec	719527	1664402	3096318	3993947
Speedup	13%	17%	11%	11%

TABLE I. ACHIEVED ITERATIONS PER CELLS PER SEC FOR DIFFERENT NUMBER OF CORES ON A IBM POWER6 ARCHITECTURE

Cores	2	4	8	12
NO_DSL iterations/sec	41914	65937	61292	55209
DSL iterations/sec	48574	75521	68908	60927
Speedup	16%	14%	12%	10%

TABLE II. ACHIEVED ITERATIONS PER CELLS PER SEC FOR DIFFERENT NUMBER OF CORES ON A INTEL WESTMERE ARCHITECTURE

Performance Counter	NO DSL	DSL	Improvement
Retired instructions	1.68322e+12	1.5579e+12	7% reduction
Cycles per instruction	0.546809	0.514415	6% reduction
L1 cache misses rate	0.0170913	0.00532005	68% reduction
L2 cache misses rate	0.00518718	0.00410406	20% reduction
Memory bandwidth (MB/sec)	1221.44	1422.61	14% increase

TABLE III. PERFORMANCE COUNTERS ON A INTEL WESTMERE ARCHITECTURE

```
8
9  cell_cells =>  cells_subset%cells
10
11 DO cell_block = cells_subset%start_block, &
12 & cells_subset%end_block
13    ...
14   DO cell_idx = cell_idx_start, cell_idx_end
15      ...
16     DO cell_level = cells_subset%start_level, &
17     & cells_subset%end_level
18        ...
19        div_vec_c(cell_level, cell_idx, cell_block) = 0.0_wp
20        ...
21        DO edge_cell_idx = 1, cell_cells%num_edges(cell_idx, &
22        & cell_block)
23           ...
24           div_vec_c(cell_level, cell_idx, cell_block) = ...
25           ...
26        ENDDO
27     ENDDO
28   ENDDO
29 ENDDO
```

As seen on the example code, at translation time, all the indexing information will be written, and the DSL loops will be transformed into 4 nested Fortran loops. This will also give the opportunity for automatic parallelization, for example, by inserting OpenMP pragmas around the outermost loop. Emerging architectures based on accelerators or heterogeneous hardware can be targeted for example by using annotations of the type of OpenACC. Specific architectures like Nvidia accelerators can be targeted by creating CUDA kernels from the generated loops. It is notable that this approach provides the ability to easily shape the actual array dimensions to meet the requirements for different levels of parallelism (blocks, thread groups, threads, vectors, etc.).

The syntax for such new features is currently under evaluation and discussion in the ICON development group.

Performance keeps being a key driver for the development of the DSL; heterogeneity in current hardware architectures makes totally feasible that fixed memory layouts could not benefit from different memory schemes in the same platform. Hence, an important upgrade for the translator is the enhancement of keywords related to index interchange to encode more than one swapping possibilities and the ability to select the correct exchange and memory representation according to the execution context.

Moreover, optimizers should be extended to permit code to explode better the advantages of emerging accelerator architectures; therefore and in opposite to inlining, code outlining can be helpful to consistently separate blocks of code with computation intensity that can be optimally mapped as kernels into those accelerators.

VII. Conclusion

We have presented ICON DSL as a Fortran extension that eases the modeling process for the climate expert, allows code portability and facilitates performance improvement. The strength of this approach relies on the fact that there is no need to learn a new language, because new keywords can be assimilated easily into existing Fortran code. With the proposed DSL features, array declarations and initializers can take advantage of memory layout abstractions while subroutine calls can be easily optimized by being inlined. Automatically generated code exhibited a significant improvement on IBM Power6 and Intel Westmere architectures when the appropriate set of index interchanges were expressed in the configuration file of the DSL.

Acknowledgment

This work is conducted within the frame of Work Package 2 in the ICOMEX project. ICOMEX is funded by the DFG as a part of the G8Initiative (GZ: LU 1353/51, LI 2125/11, ZA 268/91). Raul Torres would like to acknowledge the support from Colciencias.

References

[1] W. Washington, "Challenges in climate change science and the role of computing at the extreme scale," in *Proc. of the Workshop on Climate Science*, 2008.

[2] G. Zaengl, "The icosahedral nonhydrostatic (icon) model: formulation of the dynamical core and physics-dynamics coupling," Multiscale Numerics for the Atmosphere and Ocean at Isaac Newton Institute for Mathematical Sciences, September 2012.

[3] A. V. Deursen, P. Klint, and J. Visser, "Domain-specific languages: An annotated bibliography," *ACM SIGPLAN NOTICES*, vol. 35, pp. 26–36, 2000.

[4] N. Oliveira, M. J. ao Varanda Pereira, P. R. Henriques, and D. da Cruz, "Domain-Specic Languages - A Theoretical Survey," in *Proceedings of the 3rd Compilers, Programming Languages, Related Technologies and Applications (CoRTA'2009)*, 2009, pp. 35–46.

[5] M. Fowler, *Domain-specific languages*, Addison-Wesley, Ed. Addison-Wesley, 2011.

[6] S. Guyer and C. Lin, "Broadway: A compiler for exploiting the domain-specific semantics of software libraries," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 342–357, 2005.

[7] R. A. V. Engelen, "Atmol: A domain-specific language for atmospheric modeling," *the Journal of Computing and Information Technology*, vol. 4, pp. 289–303, 2002.

[8] R. van Engelen, L. Wolters, and G. Cats, "Ctadel: a generator of multi-platform high performance codes for pde-based scientific applications," in *Proceedings of the 10th international conference on Supercomputing*, ser. ICS '96. New York, NY, USA: ACM, 1996, pp. 86–93. [Online]. Available: http://doi.acm.org/10.1145/237578.237589

[9] P. Unden, L. Rontu, H. Jrvinen, P. Lynch, J. Calvo, G. Cats, J. Cuxart, K. Eerola, C. Fortelius, J. A. Garcia-Moya, C. Jones, Geert, G. Lenderlink, A. Mcdonald, R. Mcgrath, B. Navascues, N. W. Nielsen, V. Degaard, E. Rodriguez, M. Rummukainen, K. Sattler, B. H. Sass, H. Savijarvi, B. W. Schreur, R. Sigg, and H. The, *HIRLAM-5 Scientific Documentation*, 2002.

[10] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan, "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*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 9:1–9:12. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063396

[11] T. D. Han and T. S. Abdelrahman, "hicuda: a high-level directive-based language for gpu programming," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-2. New York, NY, USA: ACM, 2009, pp. 52–61. [Online]. Available: http://doi.acm.org/10.1145/1513895.1513902

[12] D. Unat, X. Cai, and S. B. Baden, "Mint: realizing cuda performance in 3d stencil methods with annotated c," in *Proceedings of the international conference on Supercomputing*, ser. ICS '11. New York, NY, USA: ACM, 2011, pp. 214–224. [Online]. Available: http://doi.acm.org/10.1145/1995896.1995932

[13] P. Messmer, "Porting cosmo to hybrid architectures," Programming weather, climate, and earth-system models on heterogeneous multi-core platforms, September 2012.

[14] T. Parr, *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007.

[15] J. R. Cordy, "The txl source transformation language," *Sci. Comput. Program.*, vol. 61, no. 3, pp. 190–210, Aug. 2006. [Online]. Available: http://dx.doi.org/10.1016/j.scico.2006.04.002

[16] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[17] D. Quinlan, "Rose: Compiler support for object-oriented frameworks," *Parallel Processing Letters*, vol. 10, no. 02n03, pp. 215–226, 2000. [Online]. Available: http://www.worldscientific.com/doi/abs/10.1142/S0129626400000214

[18] *ROSE User Manual*, Lawrence Livermore National Laboratory, Livermore, CA 94550, July 2013.

[19] J. Treibig, G. Hager, and G. Wellein, "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments," in *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops*, 2010.