ICON DSL: A Domain-Specific Language for climate modeling

Raul Torres, Leonidas Linardakis, Julian Kunkel, Thomas Ludwig

WOLFHPC 2013

18-11-2013
1 Introduction

2 ICON Domain-Specific Language

3 Design of the translation infrastructure

4 Evaluation
   - Power6 architecture
   - Intel Westmere architecture

5 On going and Future work

6 Conclusion
Introduction
Climate simulation models

- Global climate simulations are one of the “Grand Challenges” of computing
- Composed by several hundreds of thousands of code lines in a general-purpose language
- Code complexity increases to simulate additional physical processes
- Modelers have to equilibrate efficiency and portability (not an easy task)
- Debugging and maintenance are difficult
- Common high level approaches are the usage of backend libraries or template-based operators, both being awkward expressions of mathematical operators
ICON Climate Model

- Its goal is to integrate circulation models for the atmosphere and the ocean in a unified framework
- It is being written in Fortran for several years
- It exhibits several explicit machine-dependent optimizations, i.e:
  - Nested Do loops were written originally to exploit vectorization on a vector machine
  - But for cache-based architectures, the order of the loops should be changed
  - The change was achieved by using preprocessing directives
  - What about the index order? and the memory layout?
Our proposal

We aim to provide an abstraction framework for the ICON model in the form of a Domain-Specific Language (DSL)

- It is an extension of Fortran
- New keywords hide memory dimension and layout of variables with specific model semantics
- A source-to-Source translator converts DSL code into fully compatible Fortran code, where the computation details are expressed
- It uses an Intermediate Representation (IR) suitable for simplification and high level optimizations
- It has the ability to express climate mathematical operators in an easy and natural way.
- And the capability to adapt the implementation of these operators to different architectures and parallel levels

The current implementation is preliminary, but demonstrates a great potential for adaptivity and user-friendliness.
ICON Domain-Specific Language
Keyword specification

Keywords of the DSL and their corresponding behavior are defined in a separated platform-specific file. Each new keyword is defined as 3-field tuple separated by spaces, as follows:

<keyword_name> <platform_specific_settings> <keyword_type>

- **keyword_name** : new keyword
- **platform_specific_settings** : keyword feature
- **keyword_type** : where in Fortran

Example:

Platform A: BASIC_ARRAY 1,0 declare

Platform B: BASIC_ARRAY 0,1 declare
Array declarations

Configuration:

ON CELLS \{1,2,0,3\} declare

Usage of the keyword:

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

Generated Fortran code:

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

Configuration:

SHAPE_4D {1,2,0,3} initialize

Usage of the keyword:

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

Generated Fortran code:

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

Configuration:

INLINE inline optimize

Usage of the keyword:

INLINE SUBROUTINE example_subroutine(...) 

INLINE CALL example_subroutine(...)
Design of the translation infrastructure
First approach: ANTLR Parser Generator

ANTLR has capabilities for designing of parsers for grammars, specially for DSLs. However, we encountered several burden that made development harder.

- The symbol table must be built and managed by the programmer itself
- AST usage is cumbersome
- Recovery of ignored tokens might be difficult
- The implementation of the inlining mechanism required the support of an external text replacement tool

We recommend ANTLR for:

- Design of simple grammars and translators
- Implementation of parsers
- Construction of translators between different languages
Rose Compiler (http://rosecompiler.org/)

- Source-to-source translation infrastructure developed at Lawrence Livermore National Laboratory
- Open source project
- Targets expert and non-expert audience
- Works as a library and is written in C++ mostly
- Supports C, C++, Fortran and UPC
  - Front-end that converts a given language to an AST
  - Back-end that generates Fortran code
- The AST preserves all the information of the code
- Comes with some generic analyses, transformations and optimizations at the AST level
  - Loop optimization
  - Inlining
  - Outlining
  - Auto-parallelization
Rose overview

Taken from: Semantic-Aware Automatic Parallelization of Modern Applications Using High-Level Abstractions. Liao et al.
Issues with Rose

- Rose Compiler provides no interface to design a language extension
- A few correctly parsed Fortran statements have no corresponding action to build nodes on the AST
- Pragma annotations of the kind of Open MP are given nodes in C or C++ codes, but not in Fortran codes
- Same for Inlining mechanism
- Rose creates a sort of header files for Fortran modules, but they do not store the semantics of the our extension
Translation infrastructure

The translation of extended Fortran code into native Fortran works as follows:

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.
Machine-dependent config file

DSL annotated code

1. Config. file parser

2. DSL parser

3. IR transformations

4. Code generation

Machine-dependent configuration

Intermediate Representation (IR)

Figure: Translation infrastructure
Evaluation
Considerations

- Original code was optimized initially for a vector machine (NEC)
- A memory bandwidth bottleneck on current cache-based machines was detected
- We utilized an optimized memory layout for IBM Power6 and Intel Westmere architectures
- It was determined manually to make a better use of the available cache levels
- The DSL abstractions were applied on the ICON testbed code
- A synthetic test data was used with a configuration of 20480 cells x 78 levels
- The DSL keyword for inlining was not used
- Generated Fortran code was compiled and executed on the mentioned architectures
IBM Power6

With the appropriate machine-specific configuration the efficiency of central data structures of ICON could be improved, obtaining up to 17% of speedup

<table>
<thead>
<tr>
<th>Cores</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>192</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO_DSL iterations/sec</td>
<td>635479</td>
<td>1426037</td>
<td>2798150</td>
<td>3601217</td>
</tr>
<tr>
<td>DSL iterations/sec</td>
<td>719527</td>
<td>1664402</td>
<td>3096318</td>
<td>3993947</td>
</tr>
<tr>
<td>Speedup</td>
<td>13%</td>
<td>17%</td>
<td>11%</td>
<td>11%</td>
</tr>
</tbody>
</table>

**Table:** Achieved iterations per cells per sec for different number of cores on an IBM Power6 architecture
Figure: Performance comparison between code with and without DSL keywords for IBM Power 6 architecture
For the case of Westmere, up to 16% of speedup was obtained.

<table>
<thead>
<tr>
<th>Cores</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO_DSL iterations/sec</td>
<td>41914</td>
<td>65937</td>
<td>61292</td>
<td>55209</td>
</tr>
<tr>
<td>DSL iterations/sec</td>
<td>48574</td>
<td>75521</td>
<td>68908</td>
<td>60927</td>
</tr>
<tr>
<td>Speedup</td>
<td>16%</td>
<td>14%</td>
<td>12%</td>
<td>10%</td>
</tr>
</tbody>
</table>

**Table:** Achieved iterations per cells per sec for different number of cores on an Intel Westmere architecture.
Figure: Performance comparison between code with and without DSL keywords on a Intel Westmere architecture
## Intel Westmere architecture

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

**Table:** Performance counters on a Intel Westmere architecture
On going and Future work
On going work: Loop abstraction

```fortran
  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
END FOR
```
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, ...

cell_cells => cells_subset%cells

DO cell_block = cells_subset%start_block, &
& cells_subset%end_block
  ... DO cell_idx = cell_idx_start, cell_idx_end
  ... DO cell_level = cells_subset%start_level, &
& cells_subset%end_level
    ... div_vec_c(cell_level, cell_idx, cell_block) = 0.0_wp
    ... DO edge_cell_idx = 1, cell_cells%num_edges(cell_idx, &
& cell_block)
      ... div_vec_c(cell_level, cell_idx, cell_block) = ...
ENDDO
ENDDO
ENDDO
ENDDO
ENDDO
Future work

- Opportunities for automatic parallelization
- Emerging architectures based on accelerators or heterogeneous hardware can be targeted
- Different levels of parallelism (blocks, thread groups, threads, vectors, etc.) can be exploited
- Usage of different memory layouts on a single architecture
- Outlining can be used to build kernels
Conclusion
ICON DSL as a Fortran extension:

- It eases the modeling process for the climate expert
- It allows code portability and facilitates performance improvement
- There is no need to learn a new language
- Array declarations and initializers can take advantage of memory layout abstractions
- 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.
Thanks!
Danke!
Gracias!