D1.2 DSL Concepts for Icosahedral Models

Nabeeh Jumah Julian Kunkel Michel Müller

Workpackage: WP1 Towards higher-level code design
Responsible institution: Kunkel
Contributing institutions: Universität Hamburg, RIKEN, IPSL, Tokyo Institute of Technology
Date of submission: March 2018

Contents

Abstract

Exploiting the power of HPC, is a main concern for the scientists in the climate and atmospheric sciences. The general-purpose languages and their compilers are not sufficient to help them get the optimal use of the computer resources. In the AIMES project WP1, we study the approach of higher-level coding to provide performance poratbility. We examine the use of a domain-specific language to provide the scientists a tool to develop software using the domain concepts. Our approach is to extend the modeling language with extensions that are developed based on abstractions of the scientific concepts. The development of the model takes place mainly using the general-purpose language. However, the use of the extensions allows coding some parts of the model from a scientific perspective instead of the machine and performance perspectives. Such approach allows the scientists to write the scientific applications with a readable code without any optimization or hardware-related details. The performance portability is offered with the help of a source-to-source translation tool that translates the source code into an optimized code with respect to a specific machine. The higher-level semantics of the language extensions help the translation process to use the performance features of the machine with the guidance of configuration information that allow the user to control the optimization process.

As a first step, at the beginning of the project, we have explored the development of language extensions to extend the Fortran language in each of the three models. The result was a set of model-specific dialects, that were published in the first deliverable for this workpackage. In collaboration with the scientists each of whom masters one of the subject models, we have chosen a set of hand optimized Fortran codes from the models, to seek for the possible opportunities for the language extensions development. We have suggested abstractions to extend the Fortran language to serve the models, and based on the suggestions we have rewritten the given codes with the language extentions. We have discussed the suggestions based on the rewritten codes, and formalized the specifications of the language extensions after the agreement on them.

In this report we discuss the work that has been done in order to develop a domain-specific language based on the model-specific dialects. The idea is to find the commonalities between the language extensions developed in the model-specific dialects to identify the scientific sbstractions that serve the domain science, and formulate domain-based language extensions.

1 Introduction

This section describes first the relation to the project according to the project proposal in Section 1.1. Then the motivation for the work that is done under this task is discussed in Section 1.2. A brief description of the methodology that we have used during this task is given in **??**. *Section 1.3 describes the structure of this document.*

1.1 Relation to the Project

This report and whitepaper contains the high-level description of the meta-dsl and illustrating examples demonstrating its use. It also summarizes experiences with converting code into the highlevel representation.
The following text is the description of the project proposal for this task and deliverable:
In this task, the meta-dsl is designed that is able to describe common operations of all icosahedral models. This abstraction extends general-purpose programming languages allowing scientists to express high-level operations naturally. It will offer a memory-layout independent formulation and allow adaption of the memory structures based on the architecture. Vendors and computer scientists are involved to enable translation of the abstraction into performant memory-layouts and code for different architectures. This approach fosters the separation of concern between scientific domain and computer science (optimization). It will demonstrate that the programming abstractions allow better performance-portability of the applications for current and future machines since architecture- specific tuning is now cleanly encapsulated within the programming systems. In a co-design between domain-scientists, vendors and computer scientists a suitable high-level representation is chosen that can be converted to efficient architure-specific code as well. During the design, we also involve a mathematician working on the field to explore how the created meta-dsl can become independent of numerical specifics such as solver and potential grid refinement.

1.2 Motivation

The diversity of hardware architectures that are used to provide performance for climate/atmospheric models is a challenge facing the development of such models. The development and the maintainability of the models is especially challenging when it needs to run on different architectures. The semantics of the general-purpose languages (GPL) limit the compilers use of the target machine's capabilities. Thus, the code needs to be

manually modified to fit a specific machine to use the performance features it provides. Running a model on many different machines requires rewriting some parts of the code to fit the features of the different architectures and hardware configurations, yielding redundant code sections which are coded for different machines. The Scientists who develop such models need to have a deep knowledge of the technical lower-level details of the different architectures, and the necessary software development skills to write codes that use their features.

In this workpackage we investigate an approach that uses higher-level code that abstracts scientific concepts instead of the machine-dependent lower-level optimization details. We develop the GGDML *(General rid Definition and Manipulation Language)* DSL which consists of a set of language extensions to extend the modeling general-purpose language.
The source code that is written with GGDML is translated with a source-to-source translation tool. This tool translates the code into an optimized general-purpose language code. The GGDML extensions help the tool to optimize the source code.

1.3 Structure of this Document

In Section 2 the computation in icosahedral models is introduced. Section 3 discusses the methodology. Then in Section 4 we review the developement of the model-specific dialects which were published in the previous deliverable (D1.1 Model-Specific Dialect Formulations). The GGDML domain-specific language extensions are discussed in Section 5. The Hybrid Fortran development is discussed in Section 6. Section 7 discusses the experiments and the results that has been done to evaluate the developments. A review of related work in the literature and known projects is dicussed in Section 8. And finally, Section 9 gives a summary and conclusions of the work described in this document.

2 Euclidean- and Icosahedral Grid Geometries

In climate models, the grid is an essential part for the development of codes which compute the values of the variables that represent fields over some space. Grids are used to discretize the space over which the variables are measured. Some models use rectangular structured grids which simply address data by Euclidean space coordinates. An advantage of modeling with such grids is the simplicity of mapping and addressing of a variable's values. The values of a variable on a regular grid is stored in a multi-dimensional array. To access a variable's value, direct addressing with an explicitly-provided index for each dimension is used. In fact this represents a simpler addressing scheme in computer memory, which has performance advantages with regards to how efficiently an implementation can make use of the available memory bandwidth, especially when running on hardware architectures that are heavily optimized for sequential access performance (e.g. GPUs). However, the main shortcoming of rectangular grids is the difficulty to account for the curvature of the earth, which becomes increasingly problematic with increasing scale of the model, that is, a rectangular grid that covers an increasing surface area contains rectangles with either different areas or different shapes depending on the position of the rectangle. Thus, rectangular grids with longitude/latitude do not fit global climate/atmospheric models. This trade-off leads to the continuing need for different grid types to support global models development. The requirement of some models for a more isotropic and equal-area global grid creates the need to go beyond Euclidean space, e.g. towards the icosahedral geometry.

An icosahedral model is one that uses an icosahedral grid, which represents the earth surface into an icosahedron. The faces of an icosahedron are further divided into smaller traiangles 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. Yet pentagonal areas still exist then. Thus we see icosahedral models use either traiangular or hexagonal grids. Variables are declared with respect to the grid. They can be declared at the centers of the cells, on the edges of the cells, or at their vertices.

3 Methodology

Our effort in this workpackage to improve the software development process is based on using higher-level language extensions, which allows to bypass the shortcomings of the lower-level semantics of the general-purpose programming languages. The higher-level semantics enable the code translation process to transform the source code in a way that exploits the capabilities of the underlying hardware. No optimization technical details need to be written in the source code. Thus, scientists from the domain science do not need to think about the

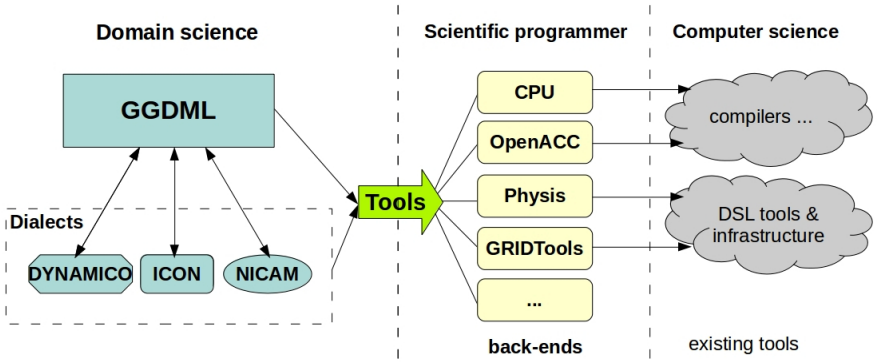


Figure 1: Separation of Concerns

hardware and performance details. In the work that is done in this workpackage we commit to the principle of separation of concerns as illustrated in Figure 1:

- Domain scientists code the problem from a scientific perspective
- Scientific programmers configure the code optimization within the source-to-source translation process

The scientists write the source code that solves a scientific problem based on scientific concepts. The GPL that the scientists generally use to build their model is used. However, the scientists can also use higher-level language extensions to write some parts of the code wherever they see that needed, although the whole code could be developed with the base language (without the extensions).
The source code is processed to translate the higher-level code into a form that is optimized with respect to a specific target machine. The code translation process is guided by configuration information that allows the translation process to make the necessary transformations in order to exploit the capabilities of the machine. The translation is prepared by scientific programmers who have the necessary technical knowledge to harness the power of the underlying architectures and hardware configurations. During this process, technical annotations can be used to direct DSLs and extensions like OpenMP.

To enable the developers to write a model's code in terms of their domain science instead of lower-level optimization details, the language extensions of the DSL are configurable again to reflect scientific concepts in the domain. We expect the DSL is developed in a co-design fashion between scientist and scientific programmers like done for, e.g., GGDML. For example, the language extensions include type specifiers that tell some hint about a variable, e.g., that it is defined over a three-dimensional grid, which reflects a scientific attribute. The same is with the iterator extension which tells that some computation is to be applied over a set of elements of a grid, which is a scientific abstraction.
The configuration controls the way the translation tool transforms the code, e.g., how to make use of the hardware to apply the computation in an iterator statement in parallel on a multicore or manycore architecture. So, a scientific programmer with expertise in GPUs for example would provide a configuration information that guides the translation tool to optimally use the GPU's processing elements to parallelize the traversal of an iterator statement over the grid elements. That information is differently written by an expert in multicore architectures to make use of the vector units and multiple cores and caching hierarchies to optimize the code for multicore processors.
The tool infrastructure is flexible allowing to design alternative DSLs while retaining some core optimizations that are independent of the frontend GPL and DSL, and the generative backend.

4 Model-Specific Dialects Reviewed

In the first deliverable of this workpackage we discussed the model-specific language extensions that were developeed based on the requirements of each of the three subject icosahedral models: Dynamico, ICON and NICAM. The requirements drove the language extensions development effort to provide different constructs to deliver the needed functionality of the extensions. The extensions included constructs to declare the model's grid-bound variables. Those extensions allow the model developer to declare a variable and tell that its values are discretized over some grid component. The higher-level extensions allowed to abstract the scientific concept without mentioning how will the variable be allocated or accessed in memory. In contrast, the computational

implementation details are left to the compilation process. The definition of the grid itself was also one part of the developed extensions. The dialects included language extensions to define the grids with a set of operators that enable specifying the dimensions and the ranges.

Among the language extensions that were developed within the model-specific dialects is the iterator construct. This constructs is an essential part of the extensions to support writing the models' kernels in an abstract code with higher-level semantics. The development of the iterator construct abstracts the details of how the kernel code is applied. It allows the developer to express what code to execute for a kernel, and over which set of grid elements. The details of the grid elements traversal and the parallelization of the kernel execution and how to read or write the data of the grid-bound variables are left to the compilation process. An abstract index is used to refer to variables values over the grid. A set of access operators allow also to refer to related grid elements, e.g. the neighbors, in a way that still uses the scientific concepts to specify grid elements relationships, instead of using memory indices to refer to the location of the variable's value in memory.

The reduction expression also was developed to simplify the coding of the stencil operations. This construct allows to write a stecil operation with a mathematical operator, e.g. addition, to a specific set of subexpressions applied repeatidly over a set of related grid elements. Using the access operators along with the reduction expression provides a good combination to write stencil codes that can be implemented on different kinds of grids.

5 GGDML: The DSL Concepts

The GGDML DSL has been developed as a set of language extensions to support the development of icosahedral-grid-based earth system models. The extensions have been developed based on the three icosahedral models Dynamico, ICON and NICAM which use icosahedral grids. The extensions were basically developed for each of the three models, while keeping in mind to specify a common set of the suggested extensions as long as possible. This common set of language extensions serves as the basis for the domain-specific language extensions that we specified as GGDML. GGDML abstracts the scientific concept of the grid and provides the necessary glue code like specifiers, expressions, iterator to access and manipulate variables and grids from a scientific point of view.

5.1 The modeling language

The three icosahedral models that were used to guide the development of the language extensions were written in Fortran. However, the nature of the solution, in which we depend on lifting the level of the semantics that are used to write the models, allows to use the technique with different languages. That is because generally the grammars of the programming languages can be extended, and the new rules could exhibit different semantical features.

The language extensions that we developed for the models which are written in the Fortran language, are usable in other languages. For example, we have used this set of language extensions to write a testcode which uses the C language as the basic modeling language. In this case, the application's code is written mainly with C, and some parts use the GGDML extensions.

The implementation of the source-to-source translation tools provides the way to allow using the language extensions to extend a specific modeling language. For example, to process the testcode that we developed with C and GGDML, a module was developed to handle the grammar of the C language. To use the language extensions to extend another language, the translation tools implementation needs to support that language.

5.2 Declarations

GGDML offers a set of declaration specifiers that allow the scientists who develop a model to mark a variable as containing values which are declared over the elements of a specific grid. The specifiers can tell, for example, that the variable has a value over each cell or edge of the grid. We have previously used the code

```
REAL , CELL, 3D :: somevar
```

to declare a variable *somevar* in the three models with the Fortran language. The same extensions are taken as part of the domain-specific language extensions. Here is an example of the same declaration also in the C language.

```
float CELL 3D somevar;
```

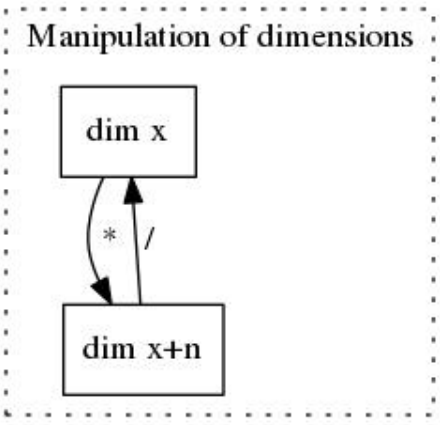


Figure 2: Dimension Operators: higher to lower dimensional grids and vice versa

To support the model-specific needs, we still allow using model-specific dialects to be used besides to the domain-specific language extensions. An example,

```
REAL , CELL, 3D, HL :: somevar
```

where we allow using model-specific declaration specifiers besides to the set of specifiers within the domain-specific language extensions.
To support multi-valued variables, array variables are also possible to use with the GGDML specifiers. An example from the test application code illustrates this

```
float CELL 2D somevar[3];
```

Although GGDML provided a set of basic specifiers, e.g., cells, edges, and vertices for the spatial position of the variables with respect to the grid, the extensions and the approach in general are designed to support extending the set of specifiers. This dynamic support for the extensibility of the tool stems from the highly configurable translation technique.

5.3 Grid Specification

The grid elements which are designated to be traversed while applying a kernel needs some way to be specified. GGDML provides language extensions to specify such ranges of grid elements. To write such a traversal range expression we can use predefined grids and a set of operators. The predefined grids are defined in the translation configurations. The operators allow to derive a range of grid elements from a grid definition to traverse.

5.3.1 The operator *

This operator gives the possibility to increase the dimensionality of the traversal range. We can multiply an expression with a new dimension to generate a range with additional one more dimension. For example, assume that the grid $gridCELL2D$ spans the set of the cells on the surface (two dimensioonal grid). Then, the expression

```
gridCELL2D * height {0 .. nLevs}
```

represents the set of the cells in the three-dimensional space. Refer to fig. 2.

5.3.2 The operator /

This operator gives the possibility to decrease the dimensionality of the traversal range. We can divide an expression by one of its dimension to generate a range with one less dimension. For example, assume that the grid $gridCELL3D$ spans the set of the cells of the three-dimensioonal grid. Then, the expression

```
gridCELL3D / height
```

represents the set of the cells on the two-dimensional surface. Refer to fig. 2.

5.3.3 The operator |

To use a RANGE with its dimention boundaries modified, we use the operator |. For example, assume we have the grid *gridCELL3D* defined with *height* dimension ranging from 0 to *nLevs*. If we want to apply a kernel to the cells of the whole grid except the lowest and the highest levels, we can use the expression

```
gridCELL3D | height {1 .. nLevs-1}
```

5.4 Iterator

Besides to the hints on the scientific attributes of the variables provided by the specifiers, GGDML provides an iterator extension as a way to express the application of a computation over the variables which are defined over the elements of the grid. The iterator statement comprises an iterator index, which allows to address one of a specific set of grid elements that are specified with a grid-specifying expression. For example, to address the cells of the three-dimensional grid. To define the set of elements over which the computation that is defined by the iterator is intended to be applied, the iterator statement comprises a special expression, which is another extension that GGDML provides as discussed in section 5.3. Those expressions specify a set of elements of a grid through the use of grid definition operators. The code example at the end of this section illustrates the idea.

5.5 Variable References

The index that is used to write the iterator represents an abstraction of a scientific concept that allows to refer to a variable at a grid element, however it does not imply any information where and how the values of the variable are stored in memory. To allow the reference to related grid elements easily, GGDML provides a basic set of operators. However, again this set is not a limited constant set, as the configurability of the translation process allows to dynamically define any operators that the developers wish to have. For example, the basic set of operators that GGDML provides includes the operator *cell.above* to refer to the cell above the cell that is being processed. Operators like *cell.neighbor* hide the indirect indices that are used in unstructured grids to refer to the related grid elements, e.g., neighbors or cell edges. Such operators abstract again the scientific concepts of the element relationships. They do not imply any information about the how the data are accessed or where they are stored.

5.6 Reduction Expressions

GGDML provides also a reduction expression that allows to simplify the coding of the computations that are applied within an iterator statement. The reduction expression removes code redundancy which happens so frequently within stencil codes, and, at the same time, allows to code sections independent of the grid type and the resulting numbers of neighbors.

To illustrate the use of GGDML, the following test code snippet demonstrates the use of the specifiers:

```
extern GVAL EDGE 3D gv_grad;
extern GVAL CELL 2D gv_o8param[3];
extern GVAL CELL 3D gv_o8par2;
extern GVAL CELL 3D gv_o8var;
```

The GVAL is a C-compiler define and we define it as float or double[1]. The specifiers are used as any other C specifier like extern. The following code demonstrates an iterator statement:

```
FOREACH cell IN grid|height{1..(g->height-1)}
{
  GVAL v0 = REDUCE(+,N={0..2},
     gv_o8param[N][cell] * gv_grad[cell.edge(N)]);

  GVAL v1 = REDUCE(+,N={0..2},
     gv_o8param[N][cell] * gv_grad[cell.edge(N).below()]);

  gv_o8var[cell] = gv_o8par2[cell]* v0
                   + (1-gv_o8par2[cell]) * v1;
}
```

[1] In the future, we will support a flexible precision of different variables that can be defined at compile time.

The iterator's grid expression here uses the GGDML grid expression modifier operator | to traverse the cells of the three-dimensional grid with the *height* dimension overridden with the boundaries 1 to the grid height -1. We can write any general-purpose language code within the iterator as a computation that will be applied over the specified grid elements. The REDUCE expression is used as follows: the value of $v0$ will be assigned the sum of the weighted values of the variable *gv_grad* multiplied by *gv_o8param* over the three edges of the cell in a triangular grid. We see here the use of multiple access operators $cell.edge(N).below()$ to access the cell below a neighboring cell.

6 Hybrid Fortran and ASUCA

Hybrid Fortran has been developed as a method for porting structured grid Fortran applications to GPU. In recognition of the advantages and disadvantages of stencil DSL- and directive-based methods (as discussed in the WACCPD 2017 proceedings paper [MA18a]) we have combined the advantages of both by employing the following characteristics in our approach:

1. it *does abstract* the parallel loops in order to achieve multiple parallelization granularities with the same code,
2. it *does not abstract* the point-wise code (i.e. the loop bodies) - allowing for code reuse,
3. it *does separate* the memory layout as defined in the user code from the layout that is effectively implemented for each architecture.

Hybrid Fortran is an open-source framework and can be accessed together with a library of sample applications[2].

In this Section we discuss how these characteristics have been achieved. Subsection 6.1 describes our approach to parallelization and granularity in Hybrid Fortran, while Subsection 6.2 discusses and compile-time defined memory layout and device memory handling. This discussion is based on relevant extracts from the proceedings of WACCPD 2017 [MA18a].

6.1 Parallel Loop Abstraction

Consider the following kernel from JMA's ASUCA reference implementation. As part of the dynamical core it is executed within the second-order Runge-Kutta scheme with high time resolution. It applies lateral and upper damping to ASUCA's grid point values.

Listing 1: Lateral and upper damping kernel applied to grid point values.

```
!$OMP PARALLEL DO
  do j = ny_mn, ny_mx
  do i = nx_mn, nx_mx
  do k = nz_mn, nz_mx
    dens_ptb_damp(k,i,j) = &
      &  mtratio_bnd * ( dens_ref_f(k,i,j) + dens_ptb_bnd(k,i,j,1) ) &
      & + tratio_bnd * ( dens_ref_f(k,i,j) + dens_ptb_bnd(k,i,j,2) ) &
      & - dens_ref_f(k,i,j)
  end do
  end do
  end do
!$OMP END PARALLEL DO
```

Using Hybrid Fortran we replace the OpenMP directives, as well as the loop instructions to be parallelized, with our parallelization DSL:

Listing 2: Lateral and upper damping kernel, modified with Hybrid Fortran.

```
@parallelRegion{ &
  & domName(i,j), domSize(nx_mn:nx_mx,ny_mn:ny_mx), &
  & startAt(nx_mn,ny_mn), endAt(nx_mx,ny_mx), template(TIGHT_STENCIL) &
& }
do k = nz_mn, nz_mx
    dens_ptb_damp(k,i,j) = &
      &  mtratio_bnd * ( dens_ref_f(k,i,j) + dens_ptb_bnd(k,i,j,1) ) &
      & + tratio_bnd * ( dens_ref_f(k,i,j) + dens_ptb_bnd(k,i,j,2) ) &
      & - dens_ref_f(k,i,j)
```

[2] Please refer to `https://github.com/muellermichel/Hybrid-Fortran`.

```
    end do
    @end parallelRegion
```

We therefore have an explicit distinction between loops that are treated as parallelizeable (and are thus restricted in their access patterns, i.e. loop carried dependencies are not supported) and loops that are always executed sequentially. The attributes `domName` and `domSize` specify the relevant domain iterators and the domain size relevant to data objects accessed within the parallel region (this relevancy will later be discussed in more detail in Section 6.2). The attributes `startAt` and `endAt` explicitly state the region boundaries, which can be a subset of the domain size (however, if omitted, the domain size is also assumed as the region boundary).

For CPU targets, Hybrid Fortran generates an OpenMP code version very similar to the reference code shown in Listing 1[3], with multi-core parallelization applied to the outermost loop. For GPU targets it defaults to CUDA Fortran kernels (thus generating all the necessary host- and device code boilerplate and data copy operations) with an option to use OpenACC kernels with CUDA compatible data structures (device pointers)[4]. The attribute `template` specifies a macro suffix used for the generated block size parameters - this allows a central configuration for the block sizes used in an application, rather than leaking this architecture-specific optimization to the user code in each kernel. If omitted, configurable default block sizes are used.

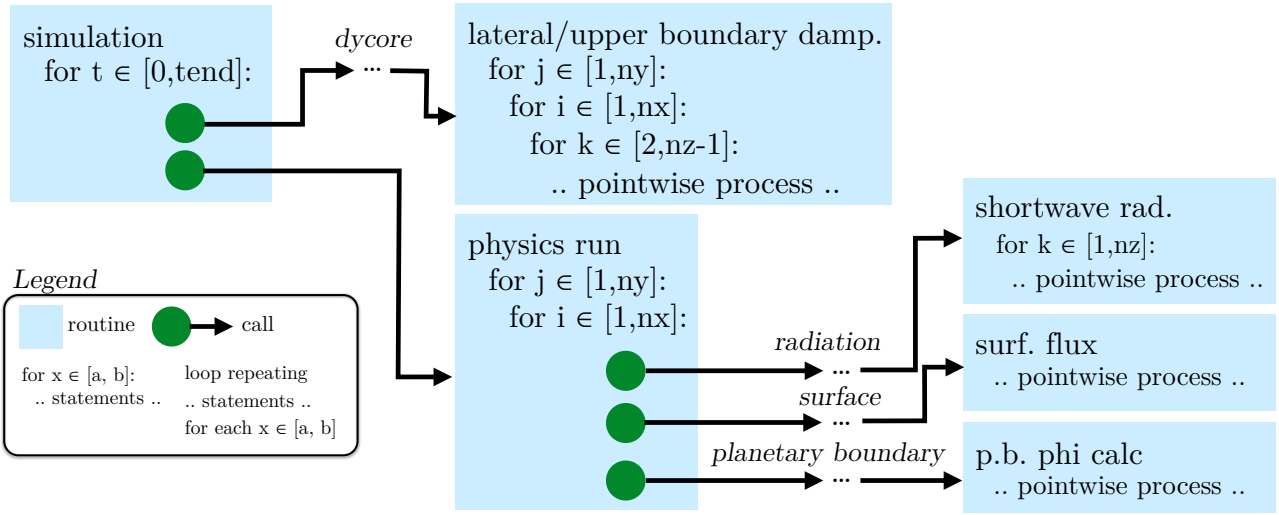


Figure 3: Simplified code structure of ASUCA.

The main advantage of this parallelization DSL is the following: replacing parallelizeable loops with the `@parallelRegion` construct allows the user to specify multiple granularities in the same code. Consider ASUCA's code structure, shown in simplified form in Figure 3. It shows two selected kernels and their embedding in the call graph - the lateral and upper boundary damping already discussed in this section, as well as the physics kernel. Many physical processes are called within this single kernel (of which three sample processes are depicted here). This code therefore has a very coarse granularity, which is problematic on GPU as discussed in the WACCPD 2017 proceedings paper [MA18a].

With Hybrid Fortran we can solve this problem as follows: An additional `appliesTo` attribute in the `@parallelRegion` statement allows the user to selectively apply parallel regions to either CPU or GPU. Applying the parallelization at different granularities therefore becomes possible[5], by enabling user-steered kernel fission. Figure 4 shows the resulting code structure, with the physics run being split into many kernels for GPU while remaining a single coarse grained kernel for CPU. The later Listing 4 gives an example of how such a kernel fission works in practice.

6.2 Compile-time Defined Memory Layout and Device Data Region

As discussed in the WACCPD 2017 proceedings paper [MA18a], it is necessary to consider two major aspects for implementing the memory layout: storage order on one hand and the compile-time defined granularity requiring

[3]Privatization is the main difference: Hybrid Fortran generated OpenMP code uses "firstprivate" as the default policy with an explicit "shared" clause for all arrays used in the kernel.

[4]OpenACC is mainly used for reduction support - Hybrid Fortran does not automatically generate reduction kernels, however it supports the "reduce" clause, which is forwarded to the generated OpenMP or OpenACC kernels.

[5]thus obviating the need for code duplication and/or deep inlining of call trees

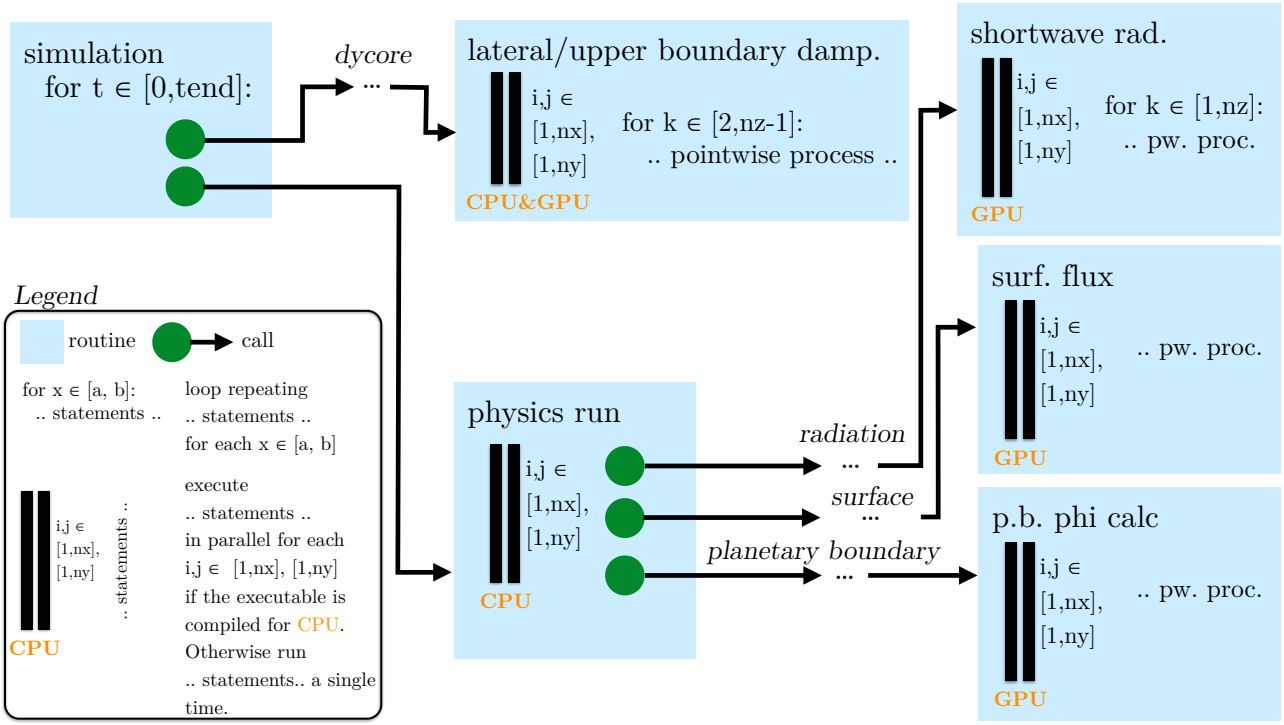


Figure 4: Simplified code structure of ASUCA using Hybrid Fortran.

a varying dimensionalities of data objects on the other. Part of Hybrid Fortran is an additional language extension, the `@domainDependant` construct, as a declarative way for the user to specify additional required information concerning data objects. This concerns memory layout as well as device memory operations, which will be discussed in this section.

6.2.1 Storage Order

Revisiting the code sample from Section 6.1, the following Listing shows the specification of the routine implementing the discussed lateral and upper damping kernel:

Listing 3: Routine implementing the lateral and upper damping kernel with Hybrid Fortran.

```
subroutine lateral_and_upper_damping()
  use ref, only : dens_ref_f
  use svar, only : dens_ptb_damp
  ! ... further imports omitted
  implicit none

  @domainDependant{ &
    & attribute(autoDom, present), &
    & accPP(AT_TIGHT_STENCIL), domPP(DOM_TIGHT_STENCIL) &
  & }
  dens_ref_f, dens_ptb_damp
  @end domainDependant

  @domainDependant{ &
    & attribute(autoDom, present), &
    & accPP(AT4_TIGHT_STENCIL), domPP(DOM4_TIGHT_STENCIL) &
  & }
  dens_ptb_bnd
  @end domainDependant

  ! ... initialisation of tratio_bnd and mtratio_bnd omitted
  ! ... kernel omitted (already shown in listing 1.2)
end subroutine
```

This shows the specification of the local module data object `dens_ptb_bnd` (density perturbation in the boundary layer) as well as the external module data objects `dens_ref_f` (reference density) and `dens_ptb_damp` (density perturbation in ASUCA grid).

The `autoDom` attribute is used to delegate the dimensions setup to the data object specification parser (which gathers this information in a separate pass from the source modules, here `ref` and `svar`), rather than having the user specify the dimensions explicitly again in the `@domainDependant` construct. The attributes `accPP` and `domPP` are employed to specify the macro names used to implement the dimension ordering for accesses and specification parts, respectively. These macros wrap all dimension lists in access expressions and specifications of respective data objects in the generated code. When `accPP` and `domPP` attributes are omitted, default macro names are used (for a code example after this conversion please refer to Listing 6). In case of Listing 3 we use explicit macro names for the dynamical core since the default macros are already used with different assumptions for the physical processes (see the paragraph on "Dimensionality Changes" below).

6.2.2 Device Data Region

Similar to OpenACC, in Hybrid Fortran we implement data regions by adding state attributes to data objects. The `present` attribute, shown in Listing 3, indicates that the respective objects are located on the device in case of GPU compilation. Analogous `transferHere` attributes are used in the main simulation routine in order to instruct Hybrid Fortran to implement the memory copy operations to- and from the device, once at the beginning and end of the simulation. For dummy variables with specified `intent`, Hybrid Fortran will use the Fortran intent information to determine the correct copy operation[6], which minimizes the potential for bugs in comparison to OpenACC's explicit `copyIn`, `copyOut` and `copy` clauses. Halo region updates, required for every timestep, are implemented explicitly in code sections guarded from CPU compilation.

6.2.3 Dimensionality Changes

Due to the compile-time defined parallelization granularity, discussed in Section 6.1, it is necessary to modify the dimensionality of data objects in certain cases in the source generation. This requires hints from the framework user. Consider the following surface flux code snippet:

Listing 4: Surface flux code snippet.

```
lt = tile_land
if (tlcvr(lt) > 0.0_r_size) then
  call sf_slab_flx_land_run( &
    ! ... inputs and further tile variables omitted
    & taux_tile_ex(lt), tauy_tile_ex(lt) &
    & )

  u_f(lt) = sqrt(sqrt(taux_tile_ex(lt) ** 2 + tauy_tile_ex(lt) ** 2))
else
  taux_tile_ex(lt) = 0.0_r_size
  tauy_tile_ex(lt) = 0.0_r_size
  ! ... further tile variables omitted
end if
! ... sea tiles code and variable summing omitted
```

Since this process is defined inside the call graph of the physics kernel, as shown in Figure 3, the relevant 2D- and 3D grid point values are already sliced and passed in as scalars or 1D-arrays, that is, data parallelism is not exposed at this level. Hybrid Fortran allows implementing this as a fine-grained kernel (as outlined in Figure 4) without modifying the computational user code, as demonstrated in the following snippet:

Listing 5: Surface flux code snippet with Hybrid Fortran.

```
@domainDependant{domName(i,j), domSize(nx,ny), attribute(autoDom, present)}
tlcvr, taux_tile_ex, tauy_tile_ex, u_f
@end domainDependant

@parallelRegion{appliesTo(GPU), domName(i,j), domSize(nx,ny)}
lt = tile_land
if (tlcvr(lt) > 0.0_r_size) then
  call sf_slab_flx_land_run( &
    ! ... inputs and further tile variables omitted
    & taux_tile_ex(lt), tauy_tile_ex(lt) &
    & )

  u_f(lt) = sqrt(sqrt(taux_tile_ex(lt) ** 2 + tauy_tile_ex(lt) ** 2))
else
```

[6] Simple examples of this feature can be found in `https://github.com/muellermichel/Hybrid-Fortran/blob/v1.00rc10/examples/demo/source/example.h90`.

```
    taux_tile_ex(lt) = 0.0_r_size
    tauy_tile_ex(lt) = 0.0_r_size
    ! ... further tile variables omitted
  end if
  ! ... sea tiles code and variable summing omitted
  @end parallelRegion
```

Using our parallelization DSL to provide additional dimensionality information, Hybrid Fortran is able to rewrite this code into a 2D kernel. Dimensions missing from the user code are inserted at the beginning of the dimension lists in access expressions and data object specifications. As an example, the expression `u_f(lt)` is converted to `u_f(AT(i,j,lt))`, employing the default ordering macro already mentioned in the paragraph "Storage Order". Dimensions are extended whenever there is a match found for `domName` or `domSize` information between data objects and parallel regions within the same routine *or* in routines called within the call graph of the same routine. It is therefore necessary for Hybrid Fortran to gather global information about the application before implementing each routine.

6.3 Transformed Code

Revisiting Listing 5, the following code is generated when applying Hybrid Fortran with the OpenACC backend:

Listing 6: Surface flux code snippet after conversion with OpenACC backend.

```
!$acc kernels deviceptr(taux_tile_ex) deviceptr(tauy_tile_ex)  &
!$acc& deviceptr(tlcvr) deviceptr(u_f)
!$acc loop independent vector(CUDA_BLOCKSIZE_Y)
outerParallelLoop0: do j=1,ny
!$acc loop independent vector(CUDA_BLOCKSIZE_X)
  do i=1,nx
      ! *** loop body *** :
    lt = tile_land
    if (tlcvr( AT(i,j,lt) )> 0.0_r_size) then
            call sf_slab_flx_land_run(&
                ! ... inputs and further tile variables omitted
                & taux_tile_ex( AT(i,j,lt) ), tauy_tile_ex( AT(i,j,lt) ) &
                & )
      u_f( AT(i,j,lt) )= sqrt(sqrt(taux_tile_ex( AT(i,j,lt) )** 2 + &
          & tauy_tile_ex( AT(i,j,lt) )** 2))
    else
      taux_tile_ex( AT(i,j,lt) )= 0.0_r_size
      tauy_tile_ex( AT(i,j,lt) )= 0.0_r_size
      ! ... further tile variables omitted
    end if
    ! ... sea tiles code and variable summing omitted
  end do
end do outerParallelLoop0
!$acc end kernels
```

Device data is interoperable with the CUDA Fortran backend, thus device pointers are used instead of passing the management to OpenACC. OpenACC directives together with this data type can thus be directly used in the user code as well, i.e. it remains interoperable with device code generated by Hybrid Fortran.

As noted in Section 6.2, storage ordering macros (here `AT()`) are applied to all array access statements. For the thread block setup, the configurable default sizes `CUDA_BLOCKSIZE_X/Y` are used since no template is specified for the parallel region at hand. Parallel region loops (here for indices `i` and —verb—j— are set up explicitly to parallelize. Other loops, such as the loop over `k`, use a `!$acc loop seq` directive to explicitly avoid parallization and give the framework user full expressiveness over the desired granularity.

Applying CUDA Fortran backend to the same user code produces the following host code (here shown together with the routine header and footer):

Listing 7: Surface flux host code snippet after conversion with CUDA Fortran backend.

```
subroutine hfd_sf_slab_flx_tile_run( &
  ! ... inputs omitted
& )
  use cudafor
  type(dim3) :: cugrid, cublock
  integer(4) :: cugridSizeX, cugridSizeY, cugridSizeZ, &
    & cuerror, cuErrorMemcopy
  ! ... other imports and specifications omitted
```

```
  cuerror = cudaFuncSetCacheConfig( &
    & hfk0_sf_slab_flx_tile_run, cudaFuncCachePreferL1)
  cuerror = cudaGetLastError()
  if(cuerror .NE. cudaSuccess) then
    ! error logging omitted
    stop 1
  end if
  cugridSizeX = ceiling(real(nx) / real(CUDA_BLOCKSIZE_X))
  cugridSizeY = ceiling(real(ny) / real(CUDA_BLOCKSIZE_Y))
  cugridSizeZ = 1
  cugrid = dim3(cugridSizeX, cugridSizeY, cugridSizeZ)
  cublock = dim3(CUDA_BLOCKSIZE_X, CUDA_BLOCKSIZE_Y, 1)
  call hfk0_sf_slab_flx_tile_run <<< cugrid, cublock >>>( &
    ! ... inputs and further tile variables omitted
    & nx, ny, tile_land, u_f, tlcvr & ! required data objects are
    & taux_tile_ex, tauy_tile_ex &    ! automatically passed to kernel
    & )
  cuerror = cudaThreadSynchronize()
  cuerror = cudaGetLastError()
  if(cuerror .NE. cudaSuccess) then
    ! error logging omitted
    stop 1
  end if
end subroutine
```

The prefix `hfd_` is added to host routines that use device data. Hybrid Fortran also duplicates the code for a pure host version of these routines (without a name change in order to remain interoperable with code that is not passed through Hybrid Fortran). In contexts where the data is not residing on the device, such as the setup part of an application, Hybrid Fortran automatically chooses the host version when generating the call statements at compile-time. Code residing within parallel regions is moved within a separeted kernel routine (using prefix `hfki_` with `i` representing the kernel number). In case of the surface flux sample shown here, the kernel routine is generated as follows:

Listing 8: Surface flux device code snippet after conversion with CUDA Fortran backend.

```
attributes(global) subroutine hfk0_sf_slab_flx_tile_run(&
  ! ... inputs and further tile variables omitted
  &, nx, ny, tile_land, u_f, tlcvr &
  &, taux_tile_ex, tauy_tile_ex &
  & )
  use cudafor
  use pp_vardef ! defines r_size
  implicit none
  real(r_size), device :: u_f(DOM(nx,ny,ntlm))
  real(r_size), device :: tlcvr(DOM(nx,ny,ntlm))
  real(r_size), device :: taux_tile_ex(DOM(nx,ny,ntlm))
  real(r_size), device :: tauy_tile_ex(DOM(nx,ny,ntlm))
  integer(4), value :: lt
  integer(4), value :: nx
  integer(4), value :: ny
  integer(4), value :: tile_land
  ! ... other imports and specifications omitted

  i = (blockidx%x - 1) * blockDim%x + threadidx%x + 1 - 1
  j = (blockidx%y - 1) * blockDim%y + threadidx%y + 1 - 1
  if (i .GT. nx .OR. j .GT. ny) then
    return
  end if
  ! *** loop body *** :
  lt = tile_land
  if (tlcvr( AT(i,j,lt) )> 0.0_r_size) then
    call hfd_sf_slab_flx_land_run( &
      ! ... inputs and further tile variables omitted
      taux_tile_ex( AT(i,j,lt) ), tauy_tile_ex( AT(i,j,lt) ) &
      & )
    ! ... rest of loop body already shown in listing 1.6
end subroutine
```

The specification part of these kernels is automatically generated, applying device state information and converting input scalars to pass-by-value[7], among other transformations.

[7] reduction kernels are thus not supported with this backend - we use the OpenACC backend selectively for this purpose, see

It is notable that CUDA Fortran requires a fairly large amount of boiler plate code for grid setup, iterator setup, host- and device code separation as well as memory- and error handling - Hybrid Fortran allows the user to pass on the responsibility for that to the framework. Compared with the code generated by OpenACC however (assembly-like CUDA C or NVVM intermediate representation), the Hybrid Fortran generated CUDA Fortran code remains easily readable to programmers experienced with CUDA. Experience shows that this is a productivity boost, especially in the debugging and manual performance optimization phase of a project.

Regarding the OpenMP backend, since the surface flux example is parallelized at a much more coarse-grained level for CPU, the generated CPU code for the sample at hand is a one-to-one copy of the user code shown earlier in Listing 4. The parallelization is generated at a higher level in the call graph (by use of a parallel region construct with `appliesTo(CPU)` clause) as follows:

Listing 9: Parallelization of physical processes on CPU.

```
!$OMP PARALLEL DO DEFAULT(firstprivate)
!$OMP& SHARED( ... inputs and outputs omitted ... )
    outerParallelLoop0: do j=1,ny
    do i=1,nx
      call physics_main(i, j, &
        ! ... inputs and outputs omitted
        & )
    end do
  end do outerParallelLoop0
!$OMP END PARALLEL DO
```

6.4 Code Transformation Method

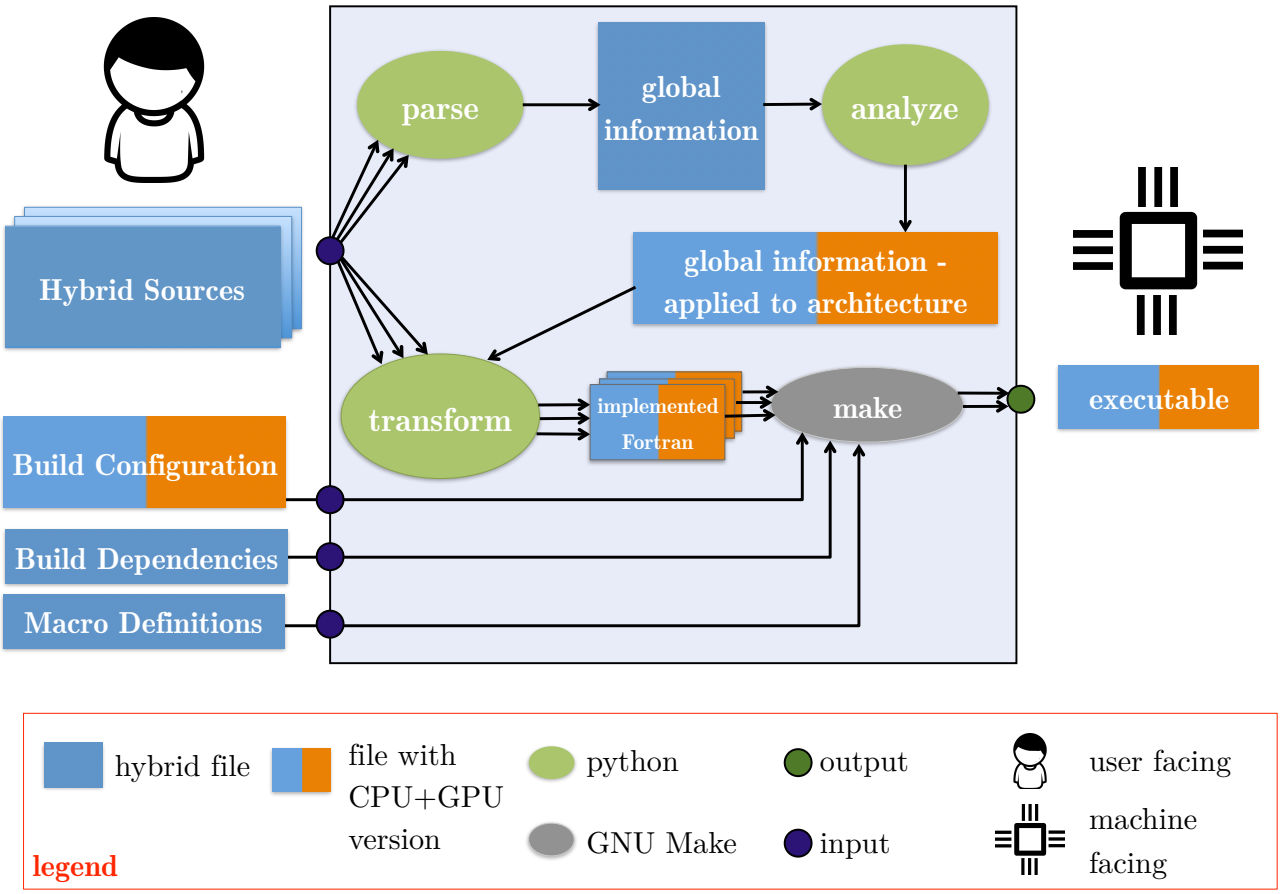


Figure 5: Hybrid Fortran software components and build workflow.

In this section we discuss code transformation method involved in implementing Hybrid Fortran's characteristics described earlier. This process is applied transparently for the user, i.e. it is applied automatically by the

also the discussion in the footnotes to Section 6.1.

means of a provided common Makefile[8]. Figure 5 gives an overview of the process and the components involved. We discuss this process in order of execution - each of the following enumerated items corresponds to one transformation phase:

1. To simplify the parsing in subsequent phases, Fortran continuation lines are merged.
2. Facilitating later phases, the application's call graph and parallel region directives are parsed globally ("parse" phase in Figure 5).
3. Using the `appliesTo` information in kernels and the call graph, the position of each routine in relation to kernels is computed. Possible positions are "has kernel(s) in its caller graph", "contains a kernel itself" and "is called inside a kernel" ("analyze" phase in Figure 5).
4. In two passes, module data object specifications are parsed and then linked against all routines with imports of such objects, together with the locally defined objects.
5. A global application model is generated, with model classes representing the modules, routines and code regions. This model can be regarded as a target hardware independent intermediate representation.
6. Each routine object is assigned an implementation class depending on the target architecture[9]. For each coding pattern, a separate class method of the implementation class is called by the model objects - e.g. CUDA parallelization boilerplate is generated. Using the previously gathered global kernel positioning and data object dimension information, data objects are transformed according to the behavior discussed in Section 6.2. Implementation class methods return strings that are concatenated by the model objects into source files ("transform" phase in Figure 5).
7. Code lines are split using Fortran line continuations in order to adhere to line limits imposed by Fortran compilers.
8. Macros generated by Hybrid Fortran (to implement storage reordering and configurable block sizes) are processed by using the GNU compiler toolchain. Subsequently, a user specified compiler and linker is employed in order to create the CPU and GPU executables. A common makefile is provided with the framework, however the build dependency graph is user-provided in the format of makefile rules[10] ("make" phase in Figure 5).

This process makes it possible to have a unified source input and create executables targeted for either multi-core CPU or many-core GPU.

7 Experiments

In this section we discuss some experiments that have been done to evaluate the achieved work that has been already finished under this workpackage.

7.1 GGDML

We have already done some experiments to evaluate our approach. First, we describe the application that has been used as a testbed code. Then, the machines that have been used to run the tests are described. Finally, we discuss the tests results.

7.1.1 Test Application

A testbed code in the C-programming language is used to test the approach. The application is an icosahedral-grid-based code, that maps variables to the cells and edges of a three-dimensional grid. The two dimensional surface is mapped to one dimension using a Hilbert space-filling-curve. The curve is partitioned into blocks. The testbed runs in time steps during each of which the model components are called to do their computations – a component can be considered a scientific process like radiation. Each component provides a compute function that calls the necessary kernels that are needed to update some variables. All the kernels are written with the GGDML extensions. The translation tool is called to translate the application's code into the different variants to run on the test machines with different memory layouts.

[8] See also the "Getting Started" section in `https://github.com/muellermichel/Hybrid-Fortran/blob/v1.00rc10/doc/Documentation.pdf`.

[9] Hybrid Fortran allows the user to switch between varying backend implementations per routine, such as OpenACC and CUDA Fortran - the user specified information as well as the defaults given by the build system call thus steers this implementation class.

[10] alternatively, a dependency generator script can be configured as well

Table 1: Single Node CPU with OpenMP

	Serial	2 Threads	4 Threads	8 Threads	16 Threads	32 Threads
3D	1.97	3.74	7.05	13.78	24.15	46.94
3D-1D	1.99	3.95	7.59	14.43	24.98	48.87

Table 2: Single node GPU

	Serial	P100			V100		
		performance GF/s	Memory throughput GB/s		performance GF/s	Memory throughput GB/s	
			read	write		read	write
3D	1.97	220.38	91.34	56.10	854.86	242.59	86.98
3D-1D	1.99	408.15	38.75	43.87	1240.19	148.49	57.12

7.1.2 Test System

Two machines have been used to run the tests. The first is the supercomputer Mistral at the German Climate Computing Center (DKRZ). Mistral offers dual socket Intel Broadwell nodes (Intel Xeon E5-2695 v4 @ 2.1GHz). The second machine is NVIDIA's PSG cluster, where we used the Haswell CPUs (Intel(R) Xeon(R) CPU E5-2698 v3 @ 2.30GHz). The GPU tests were run on NVIDIA's PSG cluster on two types of GPUs: P100 and V100.
To compile the codes and run them on Mistral we used OpenMPI version 1.8.4 and GCC version 7.1. On the PSG cluster we have used the OpenMPI version 1.10.7 and the PGI compiler version 17.10.

7.1.3 Results

In this experiment, we evaluate the application's performance for a single node. First, we translate the source code into a serial code and run it on the PSG cluster to evaluate the performance improvements on CPU and GPU. We translated it again for OpenMP to run on the Haswell multicore processors. The OpenMP version has been run with different numbers of threads. The application was also translated to run on the two types of GPUs; the P100 and the V100. We tested two memory layouts:

- **3D**: a three-dimensional addressing with three-dimensional array
- **3D-1D**: a transformed addressing that maps the original three-index addresses into an 1D index.

All the tests have been run with a 3D grid of 1024x1024x64 for 100 time steps using 32-bit floating point variables. The results for running the OpenMP tests are shown in Table 1
While the change between the two chosen memory layouts have not shown much impact on the performance on the Haswell processor, we see the impact clear when running the same code on the GPUs. The results for running the same code with the two different memory layouts on both GPU machines are shown in Table 2. We also include the measured memory throughput into the table, which we measure with NVIDIA's 'nvprof' tool. The change of the memory layout means transforming the addresses from a three-dimensional array indices to a one-dimensional array index, which means cutting down the amount of the data that needs to be read from the memory in each kernel. The caching hierarchy of the Haswell processor hides the impact by using the cached values of the additional data that needs to be read in the three-dimensional indices. However, the use of the code transformation to use the one-dimensional index while translating the code to run on the GPU allowed to get the performance gain.

To evaluate the scalability of the testbed code on multiple nodes with GPUs, we have translated the code for GPU-accelerated machines using MPI and we have run it on 1-4 nodes. Figure 6 shows the performance of the application when it is run on the P100-accelerated machines. The figure shows the performance achieved in both cases when measuring the strong and the weak scalability. The performance has been measured to find the maximum achievable performance when no halo exchange is performed, and to find the performance of an optimized code with halo exchange. The performance gap reflects the cost of the data movement from and into the GPU's memory as limited by the PCIe3 bus and along the network using Infiniband. This gap differs according to the data placement of the elements that need to be communicated to other nodes. Thus, putting the elements in an order in which halo elements are closer to each other in memory reduces the time for the data cop from and into the GPU's memory. The scalability (both strong and weak) is shown in Table 3.

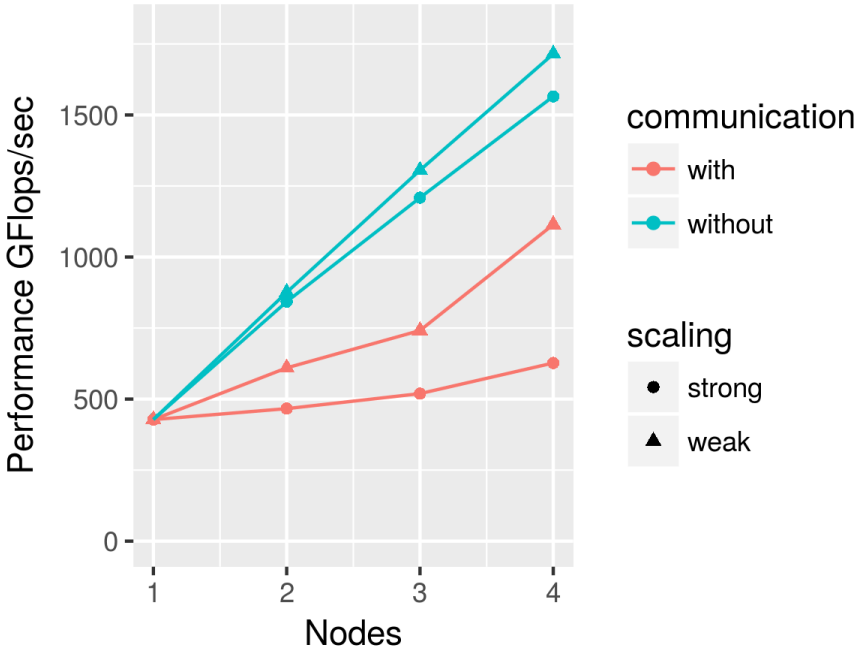


Figure 6: P100 scalability

Table 3: P100 scalability

Number of nodes	strong scaling			weak scaling		
	without communication	with communication	ratio	without communication	with communication	ratio
2	1.97	1.09	55%	2.07	1.43	70%
3	2.82	1.21	43%	3.05	1.73	58%
4	3.65	1.47	40%	4.01	2.60	65%

The table shows how the performance improves with the nodes. Also, it shows the ratio that is achieved when running the code with respect to the maximum performance gain (that is achieved without halo exchange). The computing time spent each time step for the whole grid (1024x1024x64 elements) is measured to be 8.34ms. The communication times spent during each time step are shown in Table 4.
The communication times between different numbers of MPI processes running in different mappings over nodes are recorded, Table 4 shows the measured values on the PSG cluster. We have run the application in 2,4,8,16,32,64, and 128 processes over 1,2, and 4 nodes. For multiple nodes, we mapped the MPI processes to the nodes in three ways: cyclic, blocked with balanced numbers of processes on each node, and in blocks where the processes subsequently fill the nodes. The time was measured over 1000 time steps in each case. The measured times show that optimizing the communication time is essential to achieve better performance, and that optimizing the data movement from/into the GPU's memory is essential to minimize the halo exchange time.

Table 4: Communication time per time step (in ms)

# processes	1	2 nodes			4 nodes		
		cyclic	block (balanced)	block (unbalanced)	cyclic	block (balanced)	block (unbalanced)
2	1.21	1.18	1.11	1.21			
4	1.03	0.93	0.86	1.18	0.88	0.90	1.24
8	1.00	0.84	0.77	1.52	0.77	0.75	1.58
16	0.80	0.83	0.56	1.59	0.69	0.54	1.60
32	1.29	0.77	0.64	1.26	0.69	0.51	1.24
64		1.33	0.82	0.78	0.84	0.52	0.77
128					1.48	1.32	1.23

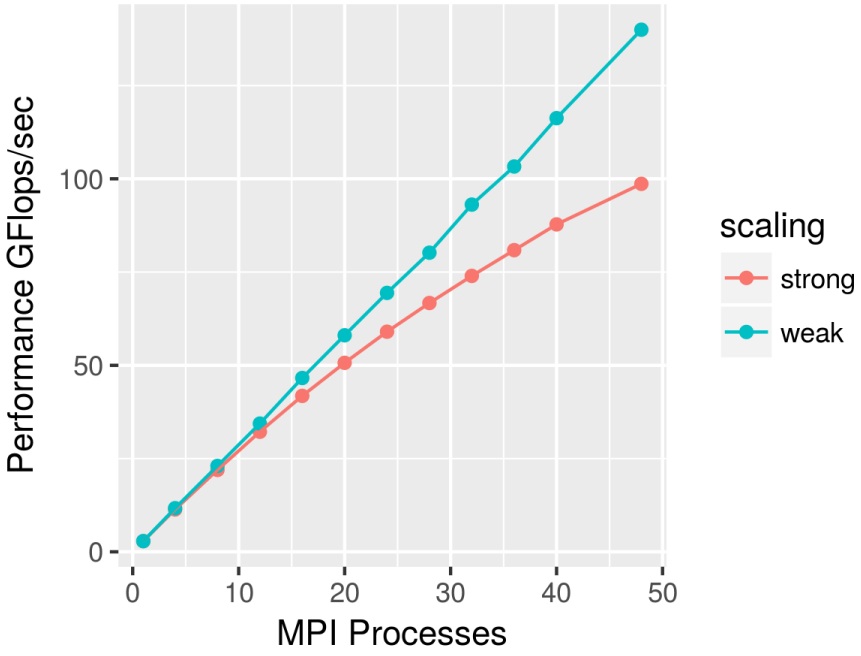
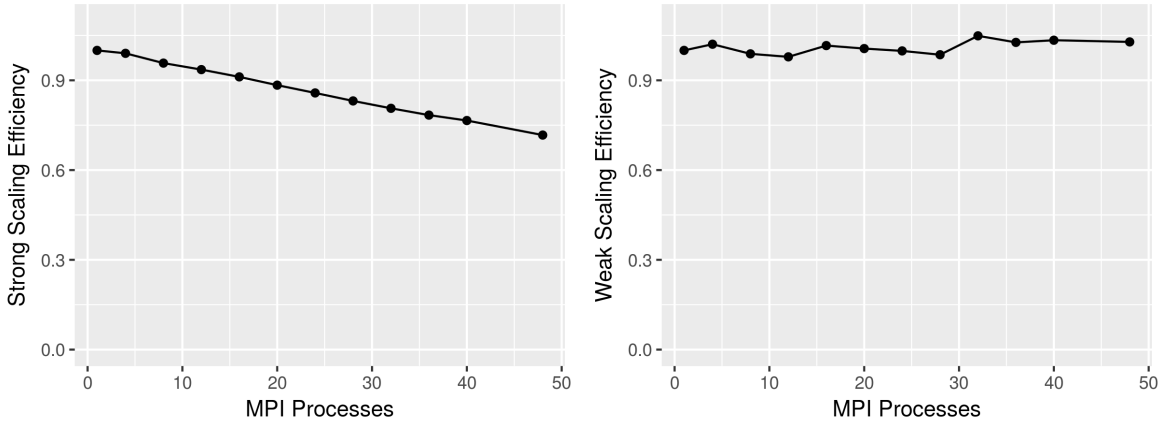


Figure 7: MPI process scalability



(a) Strong Scaling Efficiency

(b) Weak Scaling Efficiency

Figure 8: Scaling Efficiency

To evaluate the scalability of the generated code with multiple MPI processes on CPU nodes, we have run it with over 1,4,8,12,16,20,24,28,32,36,40, and 48 nodes. The performance is shown in Figure 7.
Both the strong and the weak scalability efficiency are calculated according to the equations

$$Efficiency_{strong} = T_1/(N \cdot T_N) \cdot 100\% \tag{1}$$

$$Efficiency_{weak} = T_1/T_N \cdot 100\% \tag{2}$$

and the results are shown in Figure 8. The efficiency is slightly below 100% up to 48 MPI processes for the weak scaling measurements. The Strong scaling measurements decrease from 100% at one process to about 70% at 48 processes in a linear trend.
The performance of the generated code that uses OpenMP with the MPI is also evaluated. The code has been generated for OpenMP and MPI and run with multiple numbers of nodes and using different numbers of cores on each node. We have run the code on 1,4,8,12,16,20,24,28,32,36,40 and 48 nodes and 1,2,4,8,16,32, and 36 cores per node. The measurements are shown in Figure 9.

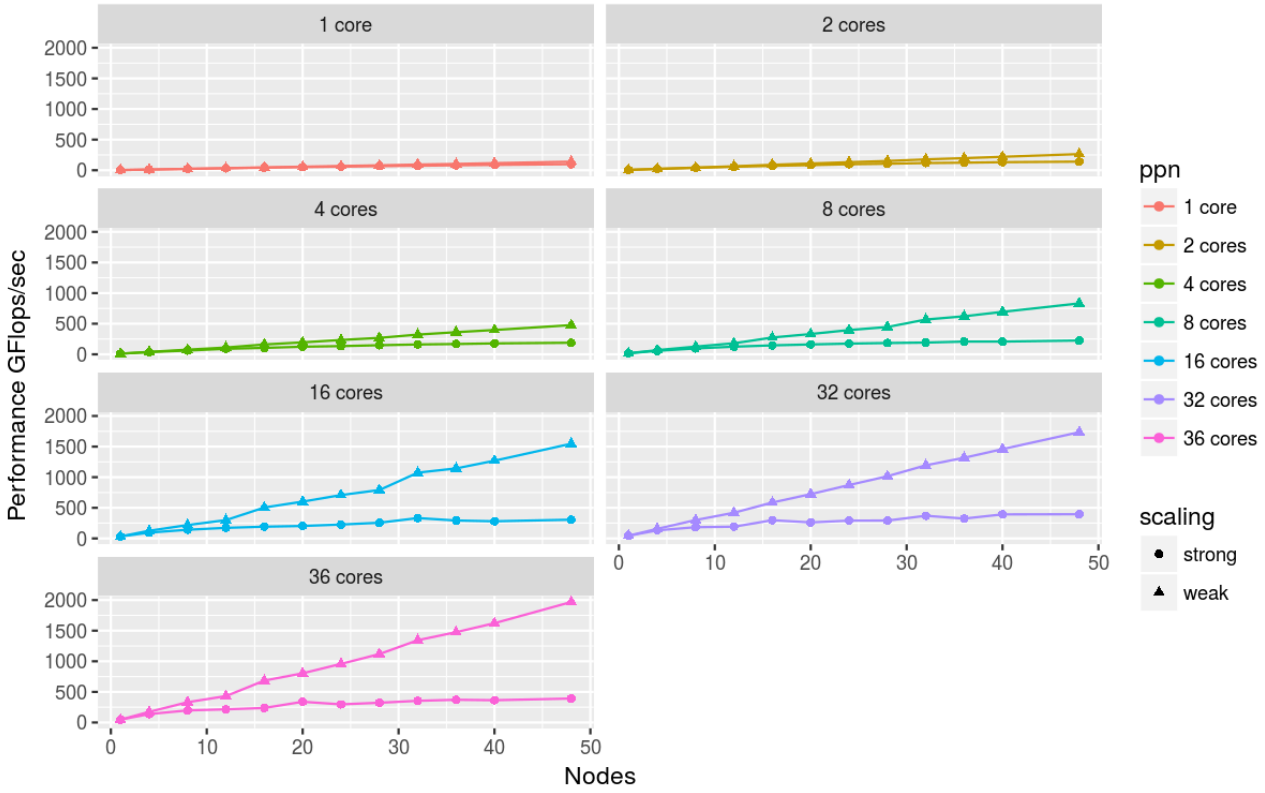


Figure 9: MPI+OpenMP scalability

7.2 Hybrid Fortran and ASUCA

The following presents performance results from our ACM paper [MA18b] as well as productivity results from the WACCPD 2017 proceedings [MA18a].

7.2.1 Productivity

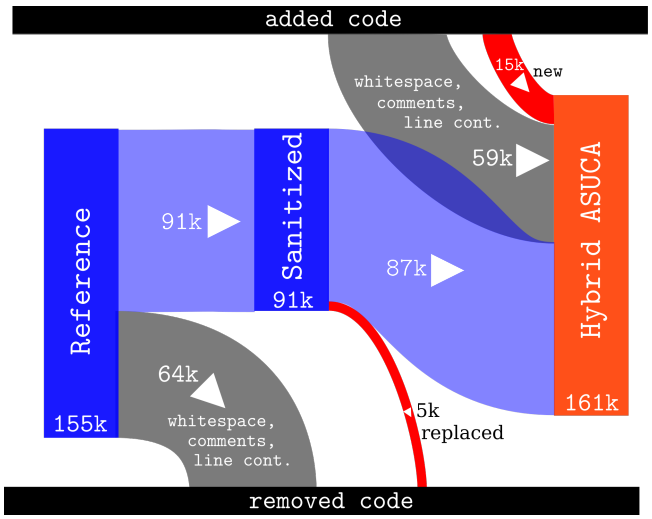
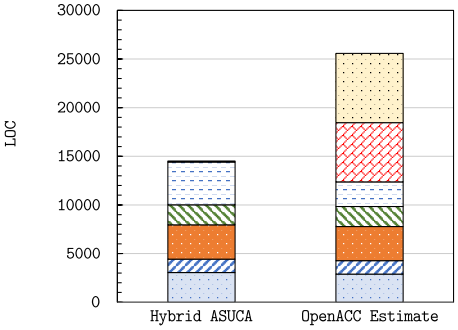


Figure 10: Flow of code lines from reference implementation to Hybrid ASUCA by number of lines of code.

To examine the productivity of our solution we have analyzed the code and compare it against the reference implementation[11]. The high-level results of this analysis is shown in Figure 10. In order to gain GPU support

[11]Since the input to this analysis is the closed source ASUCA codebase, full reproducibility cannot be provided in this context. However the intermediate data, the method employed to gather this data as well as a sample input is provided and documented in `https://github.com/muellermichel/hybrid-asuca-productivity-evidence/blob/master/asuca_productivity.xlsx`.

in addition to the already existing multi-core and multi-node parallelization, the code has grown by less than 4% in total, from 155k lines of code to 161k. Sanitizing the two code versions (removing white space, comments and merging continued lines), the code has grown by 12%, from 91k to 102k lines of code. 95% of the sanitized reference code is used as-is in the new implementation, while 5% or approximately 5k lines of code is replaced with approximately 15k new code lines.



	Hybrid ASUCA	OpenACC Estimate
CPU-only physics	0	7122
storage order macros	116	6098
parallelization & data layout DSL	4398	2521
long-wave radiation	2059	2059
modified data spec./init	3519	3519
routine & call signatures	1381	1381
other	3046	2884

Figure 11: New code required for Hybrid ASUCA vs. estimate of equivalent OpenACC implementation (LOC stands for "lines of code").

Code changes and additions have the largest impact in terms of productivity. We have analyzed the additional 15k lines of code in more detail. Figure 11 shows a breakdown of these changes and compares them to an estimate of what would be required with an OpenACC-based implementation. The following methodology has been used for this analysis:

1. for the parallelization- and data layout DSL line count we have used information parsed for Hybrid ASUCA, as well as the OpenACC backend available in Hybrid Fortran, to acquire an accurate count for the directives required for an OpenACC-based implementation,
2. since OpenACC does not offer a granularity abstraction, we have used the Hybrid ASUCA's parsed global application information to arrive at a set of routines that require a kernel positioning change (see the discussion in Section 6.4) - the resulting code lines require duplication for the CPU in an equivalent OpenACC implementation (shown as "CPU-only physics" in figure 11),
3. we have used the OpenACC backend in Hybrid Fortran to count the lines of code where storage order macros are introduced in order to achieve a compile-time defined data layout.

"Parallelization & data layout DSL" refers to the number of code lines for `@parallelRegion` and `@domainDependant` directives in case of Hybrid Fortran, and OpenACC `!$acc` directivies in case of OpenACC. Hybrid Fortran replaces the requirement for code changes to implement a varying data layout ("storage order macros", 6098 lines of code, "LOC") as well as a code duplication for multiple parallelizations with varying granularities ("CPU-only physics", 7122 LOC) with a higher number of DSL code lines compared to OpenACC (4398 vs. 2521 LOC). "Modified data specifications / initializations" (3519 LOC) as well as "routine & call signature" (1381 LOC) refers to changes applied to the setup of data and call parameter lists, respectively. These changes are necessary due to device code limitations and optimizations and are largely required for both the Hybrid Fortran version as well as a potential OpenACC solution, so we use the result from Hybrid ASUCA as an estimate for what would be required with OpenACC. Finally, one physics module concerning long-wave radiation (2059 LOC), has been replaced with a version that uses less local memory per thread (a factor of 10 improvement in that regard) to make it more GPU-friendly. We again estimate that an OpenACC version would have approximately the same code size.

This result shows that an equivalent OpenACC implementation of ASUCA can be estimated to require approximately 11k LOC in additional changes compared to the Hybrid Fortran-based implementation. When comparing to the sanitized reference codebase, an OpenACC user code would require approximately 28% of code lines to be changed or added, while Hybrid Fortran currently requires 16%.

7.2.2 Comparing Hybrid Fortran with OpenACC and Model

This section facilitates the previously discussed reduced weather application in order to compare Hybrid Fortran with OpenACC and a performance model.

7.2.2.1 Performance Portable Storage Order Table 5 shows the impact, storage order has on execution time. In the case of the currently discussed application, choosing a sub-optimal storage order impacts CPU execution time negatively by 35%, while on GPU the slowdown is 7.7x. This verifies the necessity of a flexible storage order for applications with similar data structures as ASUCA.

Table 5: Influence of Storage Order on Execution Time, $nx = ny = nz = 128$

	IJK Order	KIJ Order
CPU Single Core	1.73s	1.28s
GPU (OpenACC) (Fastest Implementation)	0.10s	0.77s

Table 6: Execution Time with "Naive" Parallelization

$nx \cdot ny \cdot nz$	128^3	256^3
CPU Single Core Measurement	1.28s	8.20s
CPU Single Core Model w/ cache	0.74s	5.70s
CPU Single Core Model w/o cache	1.77s	13.91s
CPU 6 Core Measurement	0.40s	4.25s
CPU 6 Core Model w/ cache	0.38	2.84s
CPU 6 Core Model w/o cache	0.87	6.77s
GPU Measurement	163.13s	n/a

7.2.2.2 "Naive" Parallelization In order to establish a baseline performance we apply a basic parallelization to the reduced weather application with OpenACC and OpenMP directives. Storage order is made variable across the whole application by using macros for accessing and specifying multi-dimensional arrays. We are using I-J-K order for GPU and K-I-J order for the CPU implementation. Table 6 shows the resulting CPU performance from this parallelization.

We conclude that the measured CPU performance is already well within the models with perfect and no cache. The GPU performance is however very slow - more than 400x slower than the six core CPU version, which is not in agreement with the models we have constructed. This is to be expected however, since no data region has yet been defined, without which the data is being copied over the slow PCI express bus for every kernel invocation. Further discussion for the reduced weather application therefore focuses on GPU performance.

Table 7: Execution Time with Data Region

$nx \cdot ny \cdot nz$	128^3	256^3
GPU Measurement	0.095s	0.63s
GPU Model w/ cache	0.027s	0.19s
GPU Model w/o cache	0.087s	0.66s

7.2.2.3 Data Region The amount of I/O to and from the GPU can be greatly reduced by using a data region in order to avoid host-to-device data copies at every kernel invocation. Table 7 shows the resulting performance with OpenACC.

7.2.2.4 Block Size As Table 8 shows, up to 38% performance improvement is possible over the version discussed in the previous paragraph for the OpenACC implementation by changing the default block size using vector clauses for each parallel loop. PGI accelerator uses a 128×1 block size for its OpenACC implementation by default while 32×16 has been experimentally determined to be the best block size for this application except for a small degradation for the smallest measured grid size.

Table 8: Block Size Impact on Execution Time

$nx \cdot ny \cdot nz$	128^3	256^3	$256^2 \cdot 512$
GPU Automatic Block size	0.095s	0.63s	2.99s
GPU 32 x 16 Block size	0.10s	0.55s	2.16s
GPU Model w/ cache	0.027s	0.19s	0.69s
GPU Model w/o cache	0.087s	0.66	2.60s

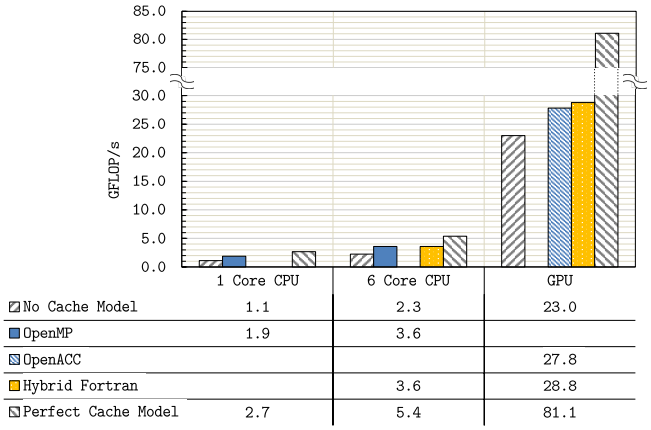


Figure 12: Comparing performance with the reduced weather application for handwritten vs. Hybrid Fortran generated vs. model on 256^3 Grid.

7.2.2.5 Performance Comparison Figure 12 shows the performance results for a reduced weather application with the Hybrid Fortran based implementation in comparison with performance models models as well as a pure the OpenMP/OpenACC based approach. For a further discussion of this reduced application and its models please refer to the ACM TOPC paper [MA18b]). For each of the implementations the highest performing version (32 x 16 block size with data region) has been selected for this comparison. This shows that measured CPU performance aligns well in between the two models (with and without cache) while for the TSUBAME 2.5 Kepler based GPU architecture, L1 cache appears to be less effective for this application and the measured performance is closer to the model without cache. It is noteworthy that the Hybrid Fortran generated code performs as well or better than the equivalent OpenACC code for both target architectures in this example.

Overall we observe a speedup of 8x for the Hybrid Fortran version vs. 6-core CPU, compared to the speedup of 7.7x for the OpenACC version. The Hybrid Fortran implementation's speedup is thus very close to the bandwidth increase 8.2x between the two architectures.

On CPU the Hybrid Fortran implementation performs the same as the manually coded OpenMP implementation, which is unsurprising given that the CPU code generated by Hybrid Fortran is practically identical.

7.2.3 Hybrid ASUCA Kernel Performance

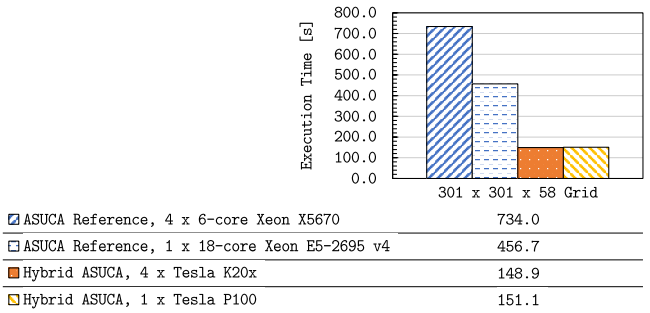


Figure 13: Execution time measurements for 75 long time steps of ASUCA executed in four different configurations

In this section, performance results for the Hybrid Fortran implementation of ASUCA (which we call "Hybrid ASUCA") are discussed for a 301 x 301 x 58 grid that is small enough for single GPU or single socket execution with the latest architecture (Tesla P100 on Reedbush-H), yet still allows a useful performance analysis in terms

of occupancy. This allows to draw conclusions for the kernel performance as opposed to the communication overhead (which impacts performance more strongly, the more nodes are used for the same grid size, i.e. when applying strong scaling as will be shown in Section 7.2.4). Additionally, an older system (TSUBAME 2.5 with Tesla K20x) is used for comparison purposes. Since the two compared GPUs differ in their available device memory (16 GB for P100 vs. 6 GB for K20x) we compare four K20x GPUs to one P100.

Figure 13 shows the execution times for this configuration on four hardware/software configurations as listed. A speedup of 4.9x has been achieved by the port on Kepler GPU vs. Westmere 6-core Xeon X5670. On newer hardware however (Pascal vs. Broadwell) the speedup has so far been a more modest 3.1x, which is partly explained by the lower memory bandwidth difference between the two comparisons and the increased caching performance on Broadwell- versus Westmere CPU architecture (as caching generally has a lower impact on GPU compared to CPU).

7.2.4 Hybrid ASUCA Strong Scaling GPU Results

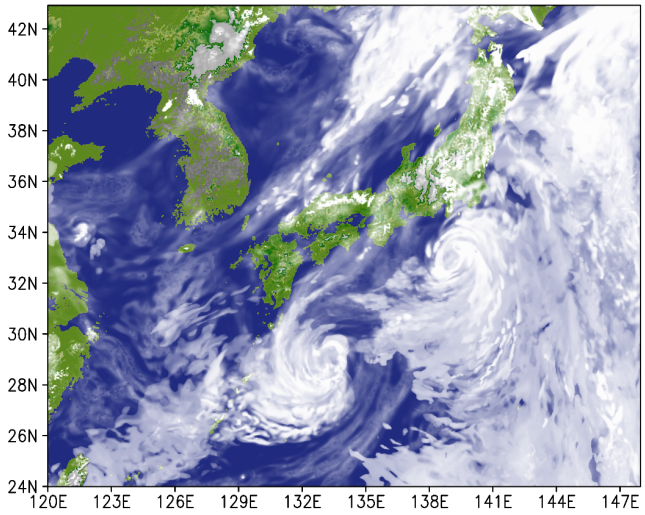


Figure 14: Total cloud cover result with ASUCA using a 2km resolution grid with a real world weather situation

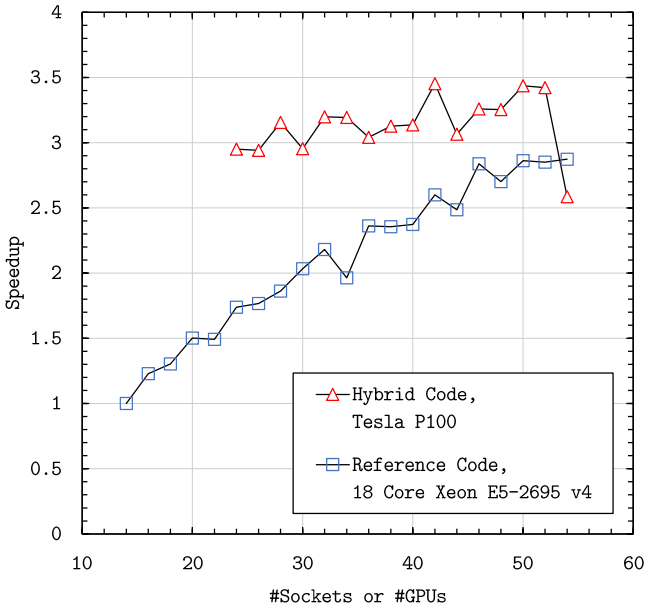


Figure 15: Strong scaling speedup on 1581 x 1301 x 58 ASUCA Grid.

Using a full scale production grid, Hybrid ASUCA has been tested on the new Tokyo University cluster "Reedbush H" [**?**]. This cluster has two 18-core Xeon E5-2695 v4 CPUs per node as well as two NVIDIA Tesla P100 GPUs per node. At least seven nodes (14 sockets) or 24 GPUs are required for this test due to the grid's

memory requirements. A visualization of the resulting cloud cover from this simulation is depicted in Figure 14.

Figure 15 shows that 24 GPUs can replace more than 50 18-core CPU sockets. When comparing the same number of GPUs and CPU sockets, the GPU is up to 76% faster.

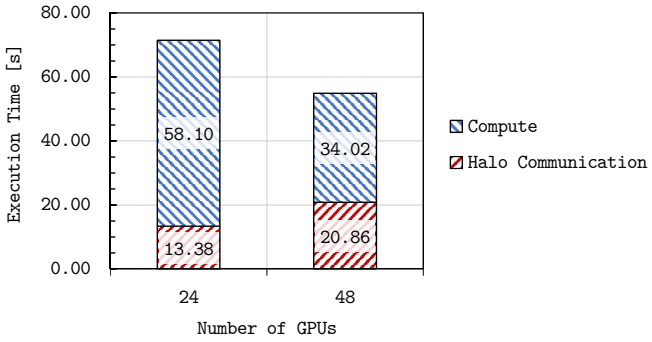


Figure 16: Impact of communication for strong scaling on 1581 x 1301 x 58 ASUCA Grid.

The following factors influence scalability and will need to be improved in order to achieve better strong scaling:

1. MPI communications code has been used as-is, with no further optimizations applied with respect to the targeted cluster. When testing the impact of communication on performance on TSUBAME 3.0 using the minimally required 24 GPUs, as shown in Figure 16, communication requires approximately 13.4s or 18.7% of the overall runtime of 71.5s (to compute a 600 seconds simulation of the full regional grid in 2km resolution). When doubling the number of GPUs this increases to 20.9s while the compute time decreases from 58.1s to 34.0s, thus the communication then takes 38% of the runtime. Overlapping communication and computations has been shown to be effective in enabling better scaling by Shimokawabe et al. [**?**], thus this approach will be the first step to improve performance at larger scales.

2. Since GPUs require a large enough problem size per chip in order to have a sufficient number of threads to fill all schedulers, strong scaling is limited when the problem size per GPU becomes too small.

3. In our ASUCA code version, as in the given reference, we do not have a distributed file I/O system as used in production. Due to the large amount of data, even though for this test the output is only run at the beginning and end of the simulation, it still has a strong impact on the overall execution time, resulting in part of the discrepancy between for the speedups between the 301 x 301 and 1581 x 1301 grid sizes.

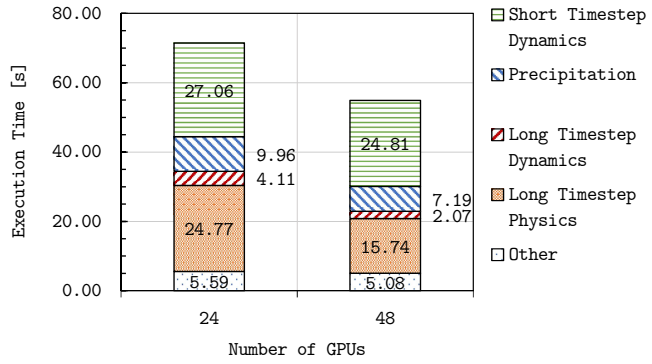


Figure 17: Impact of modules for strong scaling on 1581 x 1301 x 58 ASUCA Grid.

Figure 17 categorizes the performance impact of the different modules of ASUCA on performance, including communication. The simulation of fast moving sound- and gravity waves has the highest impact, followed by radiation- and boundary layer physics. Since the physics calculations do not require communication, the impact of fast moving dynamics increases with the number of nodes, rendering it the most important optimization target for larger scale simulations.

8 Related Work

Many research efforts were directed towards solving the problem of performance portability. 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 that models can use to perform some computation with high performance. Bianco et al. [BV12] provide a library for stencil computations. They use generic programming capabilities of C++ language to provide a solution for regular (structured) grid based applications. The active library OP2 [MGR+12] provides a framework to generate optimized code for multiple back-ends. It provides an API to define unstructured meshes and connectivity maps with C/C++ and Fortran. OPS [RMG+14] is another active library that provides C++ domain specific abstractions for multi-block structured grid based applications. It uses source-to-source translation to generate platform specific code that makes use of optimized back-end libraries for different configurations. Tangram [CDRH15] provides optimized code for CPU and GPU platforms through data-structure-based libraries. Besides, it allows programmers to explicitly specify optimizations through using rewriting-rules within code. Shimokawabe et al. [SAO14] [SAO16] provide a C++ based framework that is added to user code to provide performance portability for regular structured grids. Programmers provide stencil computations as C++ functions that update grid points, and the framework translates them to CPU/GPU optimized codes. It also produces any needed CUDA and MPI codes.

Source code preprocessing based solutions use compiler directives to annotate parts of code that is preprocessed before being submitted to backend compiler. Those solutions use a special front-end compiler/preprocessor to process annotated code. HMPP [DBB07] uses directives along with a runtime to generate accelerator high performance code. Parallelization and optimization decisions are provided by source code also in the work of Christen et al. [CSB11], where they provide a DSL for code generation and auto tuning of stencil codes for manycore and multicore processor based systems. Mint [UCB11] uses annotations to translate stencil computations from C to optimized CUDA C. Annotations within the source code drive the optimization process. In Gung Ho [FGH+13], scientific code is separated into high-level operations acting on whole fields (the algorithm layer) and low-level operations that explicitly compute with the data (kernels). In between sits a layer of autogenerated code, driven partly by directives, that handles looping over data and attempts to optimize performance for different architectures and parallelization strategies. In CLAW [Cla] project, optimization and implementation details like loop optimization and domain decomposition are explicitly specified with annotations added within 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. In the C++ library Kokkos [ETS14] C++ constructs are used to support different memory layouts for manycore architectures. The C++ stencil library Stella [FOL+14] was developed for structured grids in climate models. It uses domain concepts through a DSL to code kernels logic using C++ constructs. GridTools [Gri] generalize Stella 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. Berényi's work [Ber15] extends C++ language with an embedded DSL for AST manipulation. Constructs of this embedded DSL provide parallelization. YASK [You25] is a C++ framework that provides constructs to specify stencils and kernels. It provides a specialized source-to-source translator to convert scalar C++ stencil code into optimized C++ code. Optimization includes SIMD optimization in addition to many other optimizations that harness the power of Xeon-Phi processor.

Some source-to-source translation solutions specify language constructs in a domain-specific language that provides a new syntax which replaces general-purpose languages. Compilation of such DSLs code generates code for different architectures. These DSLs need to support further language features like expressions, operators, and may cover program flow and control. Acceptance of such solutions is crucial, e.g., declarative and functional programming differs significantly from usually used coding styles and thus, is not easily accepted from the domain scientists. Example DSLs that are tight to the scientific domain are Atmol [AvE01] and Liszt [DJP+11]. 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. Additional work is needed to integrate the generated code with other parts of the application code.

In contrast to standalone DSLs, Language extension depends on adding new types and constructs to a general-purpose language to support domain concepts and needs modifying a compiler accordingly to generate code. In Physis [MSNM11], code is written in C++ but extended with some domain concepts. It provides a source-to-source translator built on top of ROSE [Qui00]. Code is translated by this source-to-source translator into

the target platform code; CUDA code for GPU platforms and C for CPUs. It also uses a runtime component for each platform to achieve high performance. It generates MPI code for distributed compute resources. PyOP2 [RMM+12] provides a parallelization solution for numerical kernels over unstructured meshes through an embedded DSL. On problem-specific parameter unavailability, PyOP2 uses just-in-time kernel compilation and parallelization scheduling. Torres et al. [TLKL13] extend Fortran language to support different index permutations in multidimensional arrays in ICON model. They also use ROSE to do source-to-source translation. However, the solution was heavy-weight and the Fortran parser required many adjustments to run on the complex model code.

We build on the concept of a language extension DSL, with a more compact and dynamic configurable compilation tool. The concept applies to various general-purpose languages in general.

9 Summary and Conclusions

We developed a set of language extensions based on the dialects that we discussed in the previous deliverable (D1.1 Model-Specific Dialect Formulations). The set of the language extensions are named GGDML and provide higher-level abstractions of scientific concepts related to grids. In this report we discuss GGDML and its concepts.

To test GGDML, We used it to write a test application. The code is smaller than hand-written code. We did not need to optimize the source code, we just used the GGDML extensions where applicable. To test the performance and the optimization process, we prepared different configuration files to run the test application on different run configurations. We generated code for both multi-core processor machines, and for nodes with GPUs, both to run on single node and multi-node configurations. The translation tool could successfully apply the expected optimizations to the source code using the semantics of the GGDML extensions. The performance results of the experiments show that the codes that were generated show scalability over multiple nodes in strong and weak scaling analysis.

With our work on Hybrid Fortran we have shown that it is possible for large structured grid Fortran applications to

1. achieve a GPU implementation without rewriting major parts of the computational code,
2. abstract the memory layout and
3. allow for multiple parallelization granularities.

With our proposed method, a regional scale weather prediction model of significant importance to Japan's national weather service has been ported to GPU, showing a speedup of up to 4.9x on single GPU compared to a single Xeon socket. When scaling up to 24 Tesla P100, less than half the number of GPUs is required compared to contemporary Xeon CPU sockets to achieve the same result. Approximately 95% of the existing codebase has been reused for this implementation and our implementation has grown by less than 4% in total, even though it is now supporting GPU as well as CPU.

Acknowledgement

This work was supported by the German Research Foundation (DFG) through the Priority Programme 1648 „Software for Exascale Computing“ (SPPEXA).

References

[AvE01] Robert A van Engelen. Atmol: A domain-specific language for atmospheric modeling. *CIT. Journal of computing and information technology*, 9(4):289–303, 2001.

[Ber15] Dániel Berényi. C++ EDSL for parallel code generation. In *Grid, Cloud & High Performance Computing in Science (ROLCG), 2015 Conference*, pages 1–5. IEEE, 2015.

[BV12] Mauro Bianco and Ugo Varetto. A generic library for stencil computations. *arXiv preprint arXiv:1207.1746*, 2012.

[CDRH15] Li-Wen Chang, Abdul Dakkak, Christopher I Rodrigues, and Wenmei Hwu. Tangram: a high-level language for performance portable code synthesis. *In Programmability Issues for Heterogeneous Multicores*, 2015.

[Cla] CSCS Claw. `https://github.com/C2SM-RCM`. Accessed: 2016-11-22.

[CSB11] Matthias Christen, Olaf Schenk, and Helmar Burkhart. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 676–687. IEEE, 2011.

[DBB07] Romain Dolbeau, Stéphane Bihan, and François Bodin. Hmpp: A hybrid multi-core parallel programming environment. In *Workshop on general purpose processing on graphics processing units (GPGPU 2007)*, volume 28, 2007.

[DJP+11] Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, 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*, page 9. ACM, 2011.

[ETS14] H Carter Edwards, Christian R Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202–3216, 2014.

[FGH+13] R Ford, MJ Glover, DA Ham, CM Maynard, SM Pickles, G Riley, and N Wood. Gung ho: A code design for weather and climate prediction on exascale machines. In *Proceedings of the Exascale Applications and Software Conference*, 2013.

[FOL+14] Oliver Fuhrer, Carlos Osuna, Xavier Lapillonne, Tobias Gysi, Ben Cumming, Mauro Bianco, Andrea Arteaga, and Thomas Christoph Schulthess. Towards a performance portable, architecture agnostic implementation strategy for weather and climate models. *Supercomputing frontiers and innovations*, 1(1):45–62, 2014.

[Gri] CSCS GridTools. `http://www2.cosmo-model.org/content/consortium/developers/2016_01/Gridtools_python.pdf`. Accessed: 2016-11-22.

[MA18a] Michel Müller and Takayuki Aoki. Hybrid fortran: High productivity GPU porting framework applied to japanese weather prediction model. In *WACCPD: Accelerator Programming Using Directives 2017*, pages 20–41. Springer International Publishing, 2018.

[MA18b] Michel Müller and Takayuki Aoki. New high performance GPGPU code transformation framework applied to large production weather prediction code, 2018. Preprint as accepted for ACM TOPC.

[MGR+12] GR Mudalige, MB Giles, I Reguly, C Bertolli, and PH J Kelly. Op2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures. In *Innovative Parallel Computing (InPar), 2012*, pages 1–12. IEEE, 2012.

[MSNM11] Naoya Maruyama, Kento Sato, Tatsuo Nomura, and Satoshi Matsuoka. 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)*, pages 1–12. IEEE, 2011.

[Qui00] Dan Quinlan. ROSE: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(02n03):215–226, 2000.

[RMG+14] István Z Reguly, Gihan R Mudalige, Michael B Giles, Dan Curran, and Simon McIntosh-Smith. 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*, pages 58–67. IEEE, 2014.

[RMM+12] Florian Rathgeber, Graham R Markall, Lawrence Mitchell, Nicolas Loriant, David A Ham, Carlo Bertolli, and Paul HJ Kelly. Pyop2: A high-level framework for performance-portable simulations on unstructured meshes. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 1116–1123. IEEE, 2012.

[SAO14] Takashi Shimokawabe, Takayuki Aoki, and Naoyuki Onodera. 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*, pages 251–261. IEEE Press, 2014.

[SAO16] Takashi Shimokawabe, Takayuki Aoki, and Naoyuki Onodera. High-productivity framework for large-scale gpu/cpu stencil applications. *Procedia Computer Science*, 80:1646–1657, 2016.

[TLKL13] Raul Torres, Leonidas Linardakis, TL Julian Kunkel, and Thomas Ludwig. 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.

[UCB11] Didem Unat, Xing Cai, and Scott B Baden. Mint: realizing cuda performance in 3d stencil methods with annotated c. In *Proceedings of the international conference on Supercomputing*, pages 214–224. ACM, 2011.

[You25] Chuck Yount. Recipe: Building and Running YASK (Yet Another Stencil Kernel) on Intel® Processors. `https://software.intel.com/en-us/articles/recipe-building-and-running-yask-yet-another-stencil-kernel-on-intel-processors`, 2016 (Accessed: 2016-11-25).